

# Contents

[Azure Architecture Center](#)

[Cloud Fundamentals](#)

[Application architecture guide](#)

[Introduction](#)

[Architecture styles](#)

[Overview](#)

[N-tier application](#)

[Web-queue-worker](#)

[Microservices](#)

[CQRS](#)

[Event-driven architecture](#)

[Big data](#)

[Big compute](#)

[Choosing a compute service](#)

[Overview](#)

[Decision tree](#)

[Compute comparison](#)

[Choosing a data store](#)

[Overview](#)

[Data store comparison](#)

[Design principles](#)

[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

Pillars of software quality

Data architecture guide

Introduction

Traditional RDBMS workloads

Overview

Online transaction processing (OLTP)

Data warehousing

Online analytical processing (OLAP)

Extract, transform, and load (ETL)

Big data architectures

Overview

Batch processing

Real time processing

Machine learning at scale

Non-relational data stores

Scenarios

Advanced analytics

Data lakes

Free-form text search

Interactive data exploration

Natural language processing

Time series solutions

Working with CSV and JSON files

Technology choices

Analytical data stores

Analytics and reporting

Batch processing

Cognitive services

Data storage

- Machine learning
- Natural language processing
- Pipeline orchestration
- Real-time message ingestion
- Search data stores
- Stream processing
- Cross-cutting concerns
  - Data transfer
  - Extending on-premises data solutions to the cloud
  - Securing data solutions
- Cloud design patterns
  - Overview
  - Categories
    - Availability
    - Data management
    - Design and implementation
    - Messaging
    - Management and monitoring
    - Performance and scalability
    - Resiliency
    - Security
  - Ambassador
  - Anti-corruption Layer
  - Backends for Frontends
  - Bulkhead
  - Cache-Aside
  - Circuit Breaker
  - Command and Query Responsibility Segregation (CQRS)
  - Compensating Transaction
  - Competing Consumers
  - Compute Resource Consolidation
  - Event Sourcing

[External Configuration Store](#)

[Federated Identity](#)

[Gatekeeper](#)

[Gateway Aggregation](#)

[Gateway Offloading](#)

[Gateway Routing](#)

[Health Endpoint Monitoring](#)

[Index Table](#)

[Leader Election](#)

[Materialized View](#)

[Pipes and Filters](#)

[Priority Queue](#)

[Queue-Based Load Leveling](#)

[Retry](#)

[Scheduler Agent Supervisor](#)

[Sharding](#)

[Sidecar](#)

[Static Content Hosting](#)

[Strangler](#)

[Throttling](#)

[Valet Key](#)

[Best practices for cloud applications](#)

[API design](#)

[API implementation](#)

[Autoscaling](#)

[Background jobs](#)

[Caching](#)

[Content Delivery Network](#)

[Data partitioning](#)

[Monitoring and diagnostics](#)

[Naming conventions](#)

[Transient fault handling](#)

[Retry guidance for specific services](#)

[Performance antipatterns](#)

[Overview](#)

[Busy Database](#)

[Busy Front End](#)

[Chatty I/O](#)

[Extraneous Fetching](#)

[Improper Instantiation](#)

[Monolithic Persistence](#)

[No Caching](#)

[Synchronous I/O](#)

[Azure for AWS Professionals](#)

[Overview](#)

[Services comparison](#)

[Example Scenarios](#)

[Overview](#)

[AI](#)

[Hotel reservation chatbot](#)

[Image classification](#)

[Apps](#)

[Computer-aided engineering](#)

[Decentralized trust between banks](#)

[Web application monitoring](#)

[DevOps with containers](#)

[DevOps with Azure DevOps](#)

[SAP for dev/test](#)

[SAP for production](#)

[E-commerce front-end](#)

[E-commerce API management](#)

[E-commerce product search](#)

[Data and Analytics](#)

[IoT for construction](#)

Data warehousing and analytics

Automotive IoT data

Real-time fraud detection

Scalable order processing

Infrastructure

Computational fluid dynamics (CFD)

Linux virtual desktops

Decomposing monolithic applications

Highly scalable WordPress

Multi-tier Windows

HPC video rendering

Reference Architectures

Overview

AI

Batch scoring for deep learning models

Big data

Enterprise BI with SQL Data Warehouse

Automated enterprise BI with SQL Data Warehouse and Data Factory

Stream processing with Azure Stream Analytics

Enterprise integration

Simple enterprise integration

Hybrid networks

Overview

VPN

ExpressRoute

ExpressRoute with VPN failover

Hub-spoke topology

Hub-spoke topology with shared services

Identity management

Overview

Integrate on-premises AD with Azure AD

Extend AD DS to Azure

[Create an AD DS forest in Azure](#)

[Extend AD FS to Azure](#)

[N-tier applications](#)

[N-tier application with SQL Server](#)

[Multi-region N-tier application](#)

[N-tier application with Cassandra](#)

[Deploy a Linux VM](#)

[Deploy a Windows VM](#)

[Network DMZ](#)

[DMZ between Azure and on-premises](#)

[DMZ between Azure and the Internet](#)

[Highly available network virtual appliances](#)

[SAP](#)

[SAP NetWeaver for AnyDB](#)

[SAP S/4HANA](#)

[SAP HANA on Azure Large Instances](#)

[Serverless](#)

[Serverless web application](#)

[Serverless event processing](#)

[VM workloads](#)

[Jenkins server](#)

[SharePoint Server 2016](#)

[Web applications](#)

[Basic web application](#)

[Improved scalability](#)

[Multi-region deployment](#)

[Design Guides](#)

[Build microservices on Azure](#)

[Introduction](#)

[Domain analysis](#)

[Identifying microservice boundaries](#)

[Data considerations](#)

- [Interservice communication](#)
- [API design](#)
- [Ingestion and workflow](#)
- [API gateways](#)
- [Logging and monitoring](#)
- [CI/CD](#)
- [Manage multitenant identity](#)
  - [Introduction](#)
  - [The Tailspin scenario](#)
  - [Authentication](#)
  - [Claims-based identity](#)
  - [Tenant sign-up](#)
  - [Application roles](#)
  - [Authorization](#)
  - [Secure a web API](#)
  - [Cache access tokens](#)
  - [Client assertion](#)
  - [Protect application secrets](#)
  - [Federate with a customer's AD FS](#)
  - [Run the Surveys application](#)
- [Migrate from Cloud Services to Service Fabric](#)
  - [Migrate a Cloud Services application to Service Fabric](#)
  - [Refactor a Service Fabric application](#)
- [Design Review Framework](#)
  - [Design for resiliency](#)
  - [Failure mode analysis](#)
  - [Availability checklist](#)
  - [DevOps checklist](#)
  - [Resiliency checklist \(general\)](#)
  - [Resiliency checklist \(Azure services\)](#)
  - [Scalability checklist](#)
- [Enterprise Cloud Adoption](#)
  - [Introduction](#)

## [Getting Started](#)

[Overview](#)

[How does Azure work?](#)

[What is cloud resource governance?](#)

[Resource access governance in Azure](#)

## [Governance](#)

[Overview](#)

[Governance design for a single team](#)

[Governance design for multiple teams](#)

## [Infrastructure](#)

[Deploy a basic workload](#)

## [Operations](#)

[Overview](#)

[Establish an operational fitness review](#)

## [Appendix](#)

[Azure enterprise scaffold](#)

[Implementing Azure enterprise scaffold](#)

## Azure Application Architecture Guide

A guide to designing scalable, resilient, and highly available applications, based on proven practices that we have learned from customer engagements.

## Reference Architectures

A set of recommended architectures for Azure. Each architecture includes best practices, prescriptive steps, and a deployable solution.

## Enterprise Cloud Adoption

This guide outlines a process for creating an organization-wide cloud adoption strategy. It focuses on organizational readiness, governance, and infrastructure.



## Build Microservices on Azure

This design guide takes you through the process of designing and building a microservices architecture on Azure. A reference implementation is included.

## Azure Data Architecture Guide

A structured approach to designing data-centric solutions on Microsoft Azure.

## Cloud Best Practices

Best practices for cloud applications, covering aspects such as auto-scaling, caching, data partitioning, API design, and others.

## Design for Resiliency

Learn how to design resilient applications for Azure.

## Azure Building Blocks

Simplify deployment of Azure resources. With a single settings file, deploy complex architectures in Azure.

## Cloud Design Patterns

Design patterns for developers and solution architects. Each pattern describes a problem, a pattern that addresses the problem, and an example based on Azure.

## **Design Review Checklists**

Checklists to assist developers and solution architects during the design process.

### **Azure Virtual Datacenter**

When deploying enterprise workloads to the cloud, organizations must balance governance with developer agility. Azure Virtual Datacenter provides models to achieve this balance with an emphasis on governance.

### **Azure for AWS Professionals**

Leverage your AWS experiences in Microsoft Azure.

### **Performance Antipatterns**

How to detect and fix some common causes of performance and scalability problems in cloud applications.

---

## Build your skills with Microsoft Learn

□

### **Pillars of a great Azure architecture**

□

### **Design for security in Azure**

□

### **Design for performance and scalability in Azure**

□

### **Design for efficiency and operations in Azure**

□

### **Design for availability and recoverability in Azure**

This guide presents a structured approach for designing applications on Azure that are scalable, resilient, and highly available. It is based on proven practices that we have learned from customer engagements.

## Introduction

The cloud is changing the way applications are designed. 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 state is distributed. Operations are done in parallel and asynchronously. The system as a whole must be resilient when failures occur. Deployments must be automated and predictable. Monitoring and telemetry are critical for gaining insight into the system. The Azure Application Architecture Guide is designed to help you navigate these changes.

TRADITIONAL ON-PREMISES	MODERN CLOUD
Monolithic, centralized Design for predictable scalability Relational database Strong consistency Serial and synchronized processing Design to avoid failures (MTBF) Occasional big updates Manual management Snowflake servers	Decomposed, de-centralized Design for elastic scale Polyglot persistence (mix of storage technologies) Eventual consistency Parallel and asynchronous processing Design for failure (MTTR) Frequent small updates Automated self-management Immutable infrastructure

This guide is intended for application architects, developers, and operations teams. It's not a how-to guide for using individual Azure services. After reading this guide, you will understand the architectural patterns and best practices to apply when building on the Azure cloud platform. You can also download an [e-book version of the guide](#).

## How this guide is structured

The Azure Application Architecture Guide 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

Two technology choices should be decided early on, because they affect the entire architecture. These are the choice of compute service and data stores. *Compute* refers to the hosting model for the computing resources that your applications runs on. *Data stores* includes databases but also storage for message queues, caches, logs, and anything else that an application might persist to storage.

Learn more:

- [Choosing a compute service](#)
- [Choosing a data store](#)

### Design principles

We have identified ten high-level design principles that will make your application more scalable, resilient, and manageable. These design principles apply to any architecture styles. Throughout the design process, keep these ten high-level design principles in mind. Then consider the set of best practices for specific aspects of the architecture, such as auto-scaling, caching, data partitioning, API design, and others.

Learn more:

- [Design principles](#)

## Quality pillars

A successful cloud application will focus on five pillars of software quality: Scalability, availability, resiliency, management, and security. Use our design review checklists to review your architecture according to these quality pillars.

- [Quality pillars](#)

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.

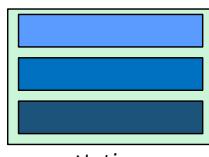
We have identified a set of architecture styles that are commonly found in cloud applications. The article for each style includes:

- 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.

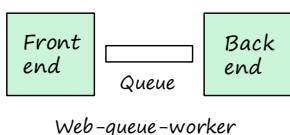
### N-tier



**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.

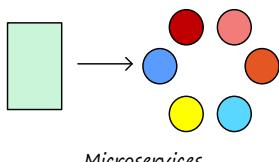
### Web-Queue-Worker



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 a 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

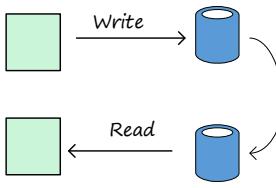


If your application has a more complex domain, consider moving to a **Microservices** architecture. A microservices application is composed of many small, independent services. Each service implements a single business capability. Services are loosely coupled, communicating through API contracts.

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.

### CQRS

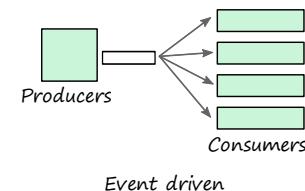
The **CQRS** (Command and Query Responsibility Segregation) style separates read and write operations into separate models. This isolates the parts of the system that update data from the parts that read the data. Moreover, reads can be executed against a materialized view that is physically separate from the write database. That lets you scale the read and write workloads independently, and optimize the materialized view for queries.



CQRS makes the most sense when it's applied to a subsystem of a larger architecture. Generally, you shouldn't impose it across the entire application, as that will just create unneeded complexity.

Consider it for collaborative domains where many users access the same data.

### Event-driven architecture



**Event-Driven Architectures** use a publish-subscribe (pub-sub) mode 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.
CQRS	Read/write segregation. Schema and scale are optimized separately.	Collaborative domain where lots of users access the same data.

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*.

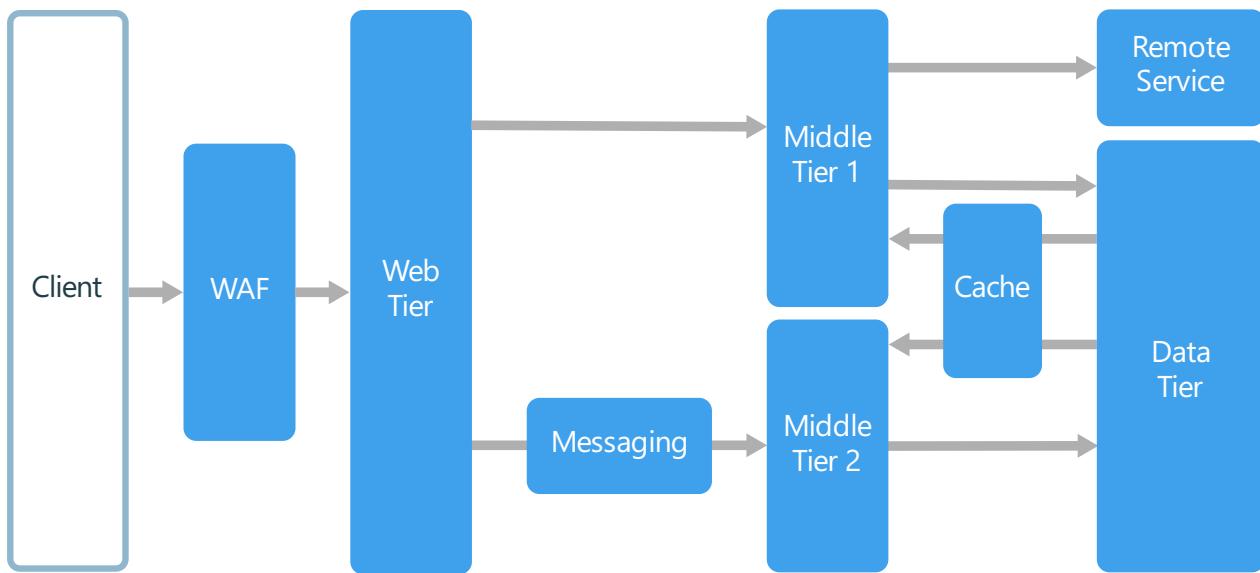
Here are some of the types of challenges to consider when selecting an architecture style:

- **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 "[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 such as always-once semantics and eventual consistency.
- **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?

# N-tier architecture style

8/30/2018 • 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

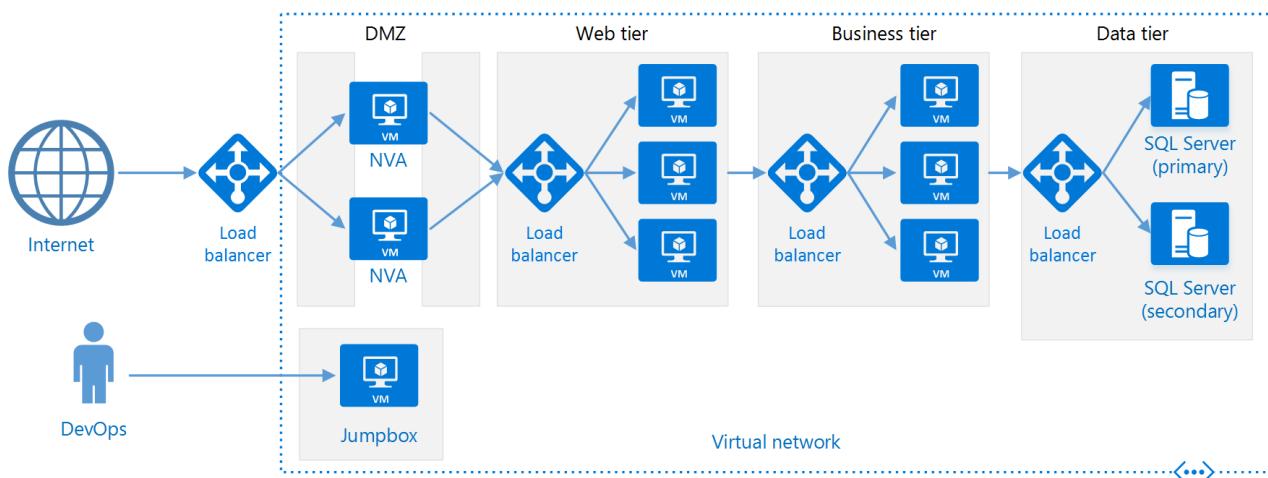
- It's easy to end up with a middle tier that just does CRUD operations on the database, adding extra latency without doing any useful work.
- Monolithic design prevents independent deployment of features.
- Managing an IaaS application is more work than an application that uses only managed services.
- 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 semi-static data. See [Caching best practices](#).
- Configure 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 VM 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 (NSG) 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 (NSGs) restrict access to each tier. For example, the database tier only allows access from the business tier.

For more details and a deployable Resource Manager template, see the following reference architectures:

- [Run Windows VMs for an N-tier application](#)
- [Run Linux VMs for an N-tier application](#)

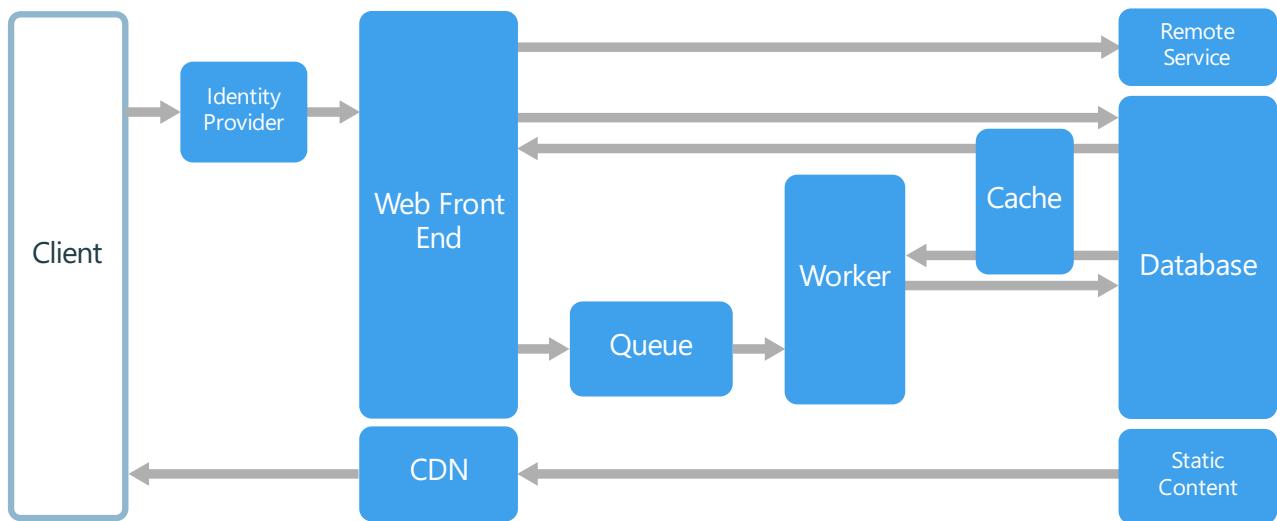
### **Additional considerations**

- 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 VM 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 an NSG 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

8/30/2018 • 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

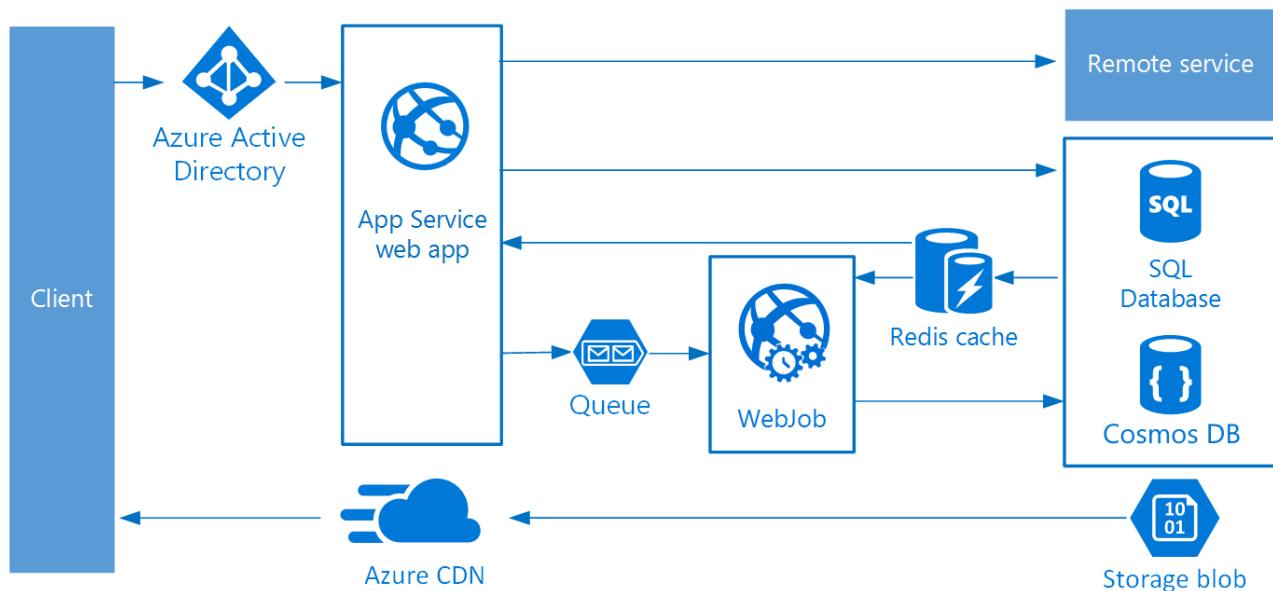
- Without careful design, the front end and the worker can become large, monolithic components that are difficult to maintain and update.
- There may be hidden dependencies, if the front end and worker share data schemas or code modules.

## 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 a WebJob. The web app and the WebJob 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 Redis Cache 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](#).

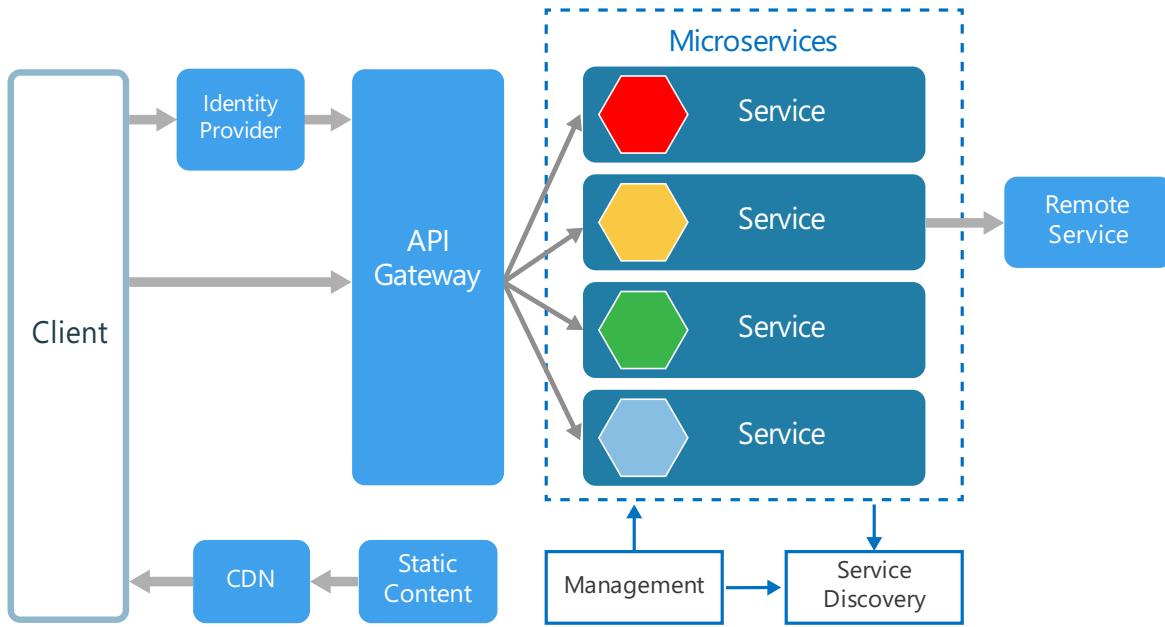
## Additional considerations

- 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 WebJob into separate App Service plans. That way, they are hosted on separate VM instances and 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.

# Microservices architecture style

8/30/2018 • 7 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. For detailed guidance about building a microservices architecture on Azure, see [Designing, building, and operating microservices on Azure](#).



In some ways, microservices are the natural evolution of service oriented architectures (SOA), but there are differences between microservices and SOA. Here are some defining characteristics of a microservice:

- In a microservices architecture, services are small, independent, and loosely coupled.
- 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.
- 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.** The management component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth.

**Service Discovery.** Maintains a list of services and which nodes they are located on. Enables service lookup to find the endpoint for a service.

**API Gateway.** The API gateway is the entry point for clients. Clients don't call services directly. Instead, they call the API gateway, which forwards the call to the appropriate services on the back end. The API gateway might aggregate the responses from several services and return the aggregated response.

The 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.

## When to use this architecture

Consider this architecture style for:

- Large applications that require a high release velocity.
- Complex applications that need to be highly scalable.
- Applications with rich domains or many subdomains.
- An organization that consists of small development teams.

## Benefits

- **Independent deployments.** You can update a service without redeploying the entire application, and roll back or roll forward an update if something goes wrong. Bug fixes and feature releases are more manageable and less risky.
- **Independent development.** A single development team can build, test, and deploy a service. The result is continuous innovation and a faster release cadence.
- **Small, focused teams.** Teams can focus on one service. The smaller scope of each service makes the code base easier to understand, and it's easier for new team members to ramp up.
- **Fault isolation.** If a service goes down, it won't take out the entire application. However, that doesn't mean you get resiliency for free. You still need to follow resiliency best practices and design patterns. See [Designing resilient applications for Azure](#).
- **Mixed technology stacks.** Teams can pick the technology that best fits their service.
- **Granular scaling.** Services can be scaled independently. At the same time, the higher density of services per VM means that VM resources are fully utilized. Using placement constraints, a services can be matched to a VM profile (high CPU, high memory, and so on).

## Challenges

- **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 test.** Developing against service dependencies requires a different approach. Existing tools are not necessarily 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.

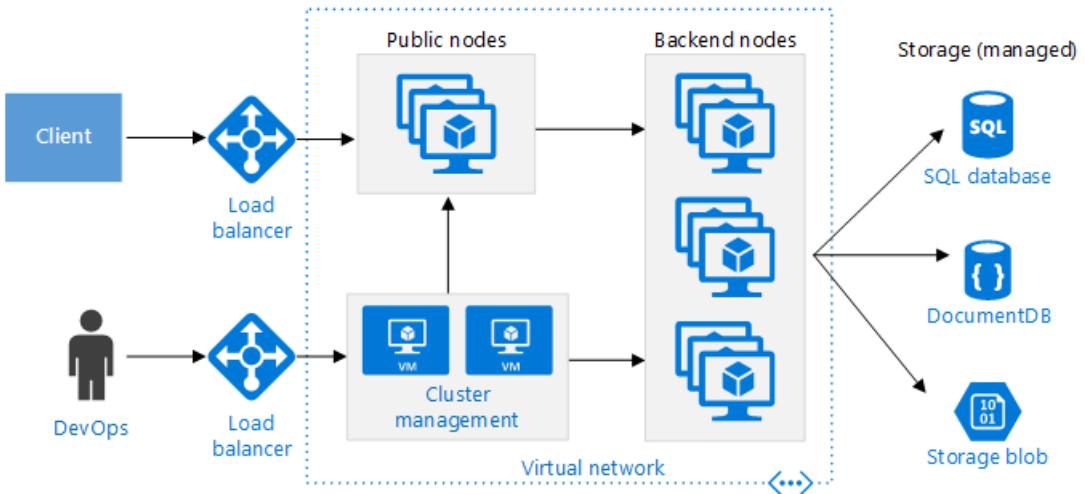
- **Network congestion and latency.** The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns.
- **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.
- **Skillset.** 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 resilient applications](#).

## Microservices using Azure Container Service

You can use [Azure Container Service](#) to configure and provision a Docker cluster. Azure Container Services supports several popular container orchestrators, including Kubernetes, DC/OS, and Docker Swarm.



**Public nodes.** These nodes are reachable through a public-facing load balancer. The API gateway is hosted on these nodes.

**Backend nodes.** These nodes run services that clients reach via the API gateway. These nodes don't receive Internet traffic directly. The backend nodes might include more than one pool of VMs, each with a different hardware profile. For example, you could create separate pools for general compute workloads, high CPU workloads, and high memory workloads.

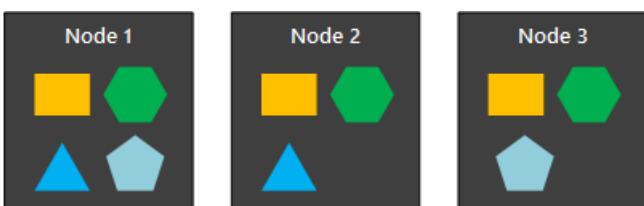
**Management VMs.** These VMs run the master nodes for the container orchestrator.

**Networking.** The public nodes, backend nodes, and management VMs are placed in separate subnets within the same virtual network (VNet).

**Load balancers.** An externally facing load balancer sits in front of the public nodes. It distributes internet requests to the public nodes. Another load balancer is placed in front of the management VMs, to allow secure shell (ssh) traffic to the management VMs, using NAT rules.

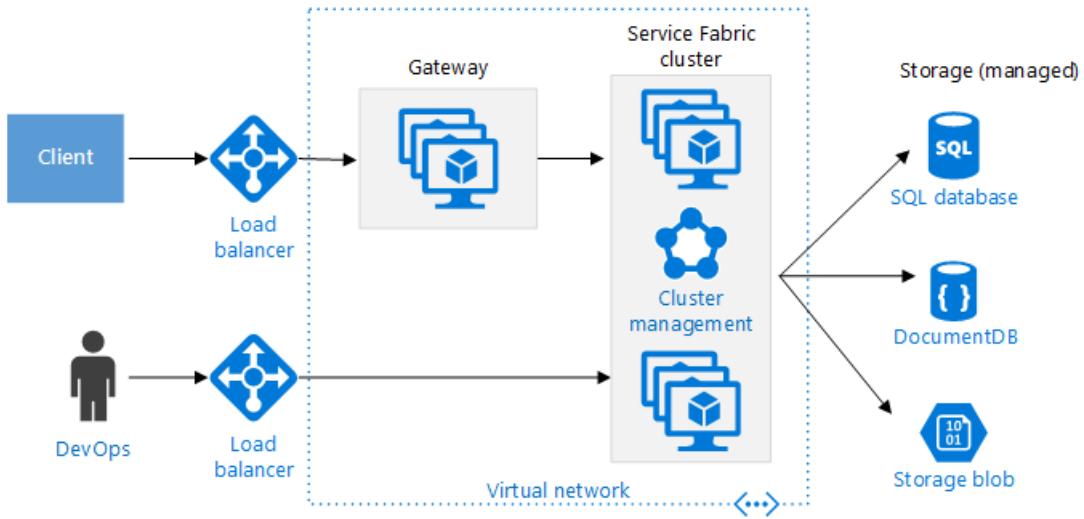
For reliability and scalability, each service is replicated across multiple VMs. However, because services are also relatively lightweight (compared with a monolithic application), multiple services are usually packed into a single VM. Higher density allows better resource utilization. If a particular service doesn't use a lot of resources, you don't need to dedicate an entire VM to running that service.

The following diagram shows three nodes running four different services (indicated by different shapes). Notice that each service has at least two instances.



## Microservices using Azure Service Fabric

The following diagram shows a microservices architecture using [Azure Service Fabric](#).



The Service Fabric cluster is deployed to one or more VM scale sets. You might have more than one VM scale set in the cluster, in order to have a mix of VM types. An API Gateway is placed in front of the Service Fabric cluster, with an external load balancer to receive client requests.

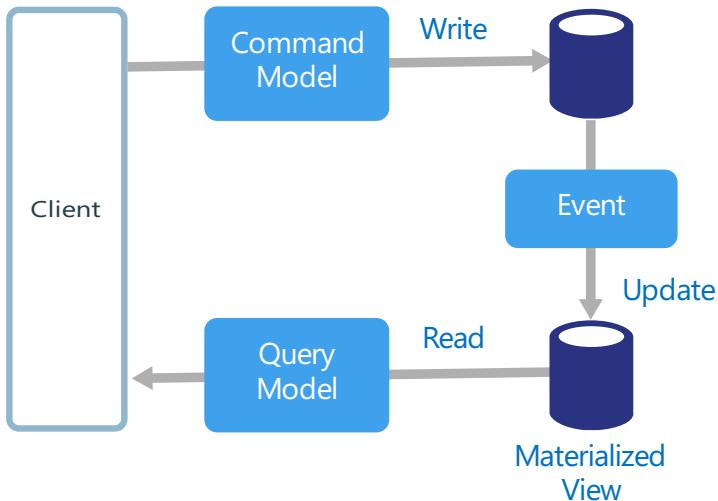
The Service Fabric runtime performs cluster management, including service placement, node failover, and health monitoring. The runtime is deployed on the cluster nodes themselves. There isn't a separate set of cluster management VMs.

Services communicate with each other using the reverse proxy that is built into Service Fabric. Service Fabric provides a discovery service that can resolve the endpoint for a named service.

# CQRS architecture style

8/30/2018 • 3 minutes to read • [Edit Online](#)

Command and Query Responsibility Segregation (CQRS) is an architecture style that separates read operations from write operations.



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.

Another potential problem is that read and write workloads are often asymmetrical, with very different performance and scale requirements.

CQRS addresses these problems by separating reads and writes into separate models, using **commands** to update data, and **queries** to read data.

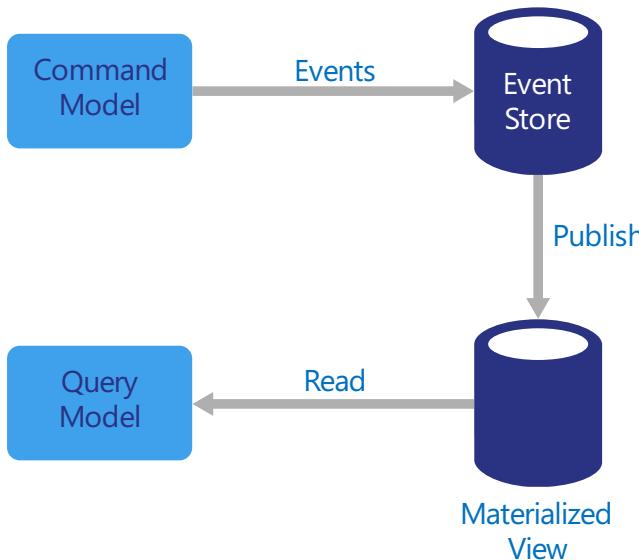
- Commands should be task based, rather than data centric. ("Book hotel room," not "set ReservationStatus to Reserved.") Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

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. Updating the database and publishing the event must occur in a single transaction.

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.



## When to use this architecture

Consider CQRS for collaborative domains where many users access the same data, especially when the read and write workloads are asymmetrical.

CQRS is not a top-level architecture that applies to an entire system. Apply CQRS only to those subsystems where there is clear value in separating reads and writes. Otherwise, you are creating additional complexity for no benefit.

## Benefits

- **Independently 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.

## Challenges

- **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.
- **Eventual consistency.** If you separate the read and write databases, the read data may be stale.

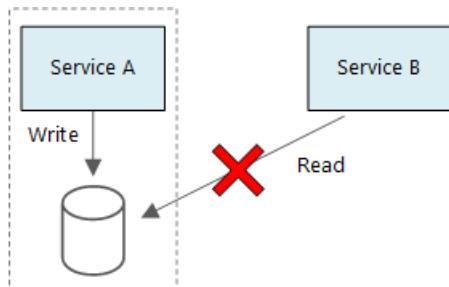
## Best practices

- For more information about implementing CQRS, see [CQRS Pattern](#).

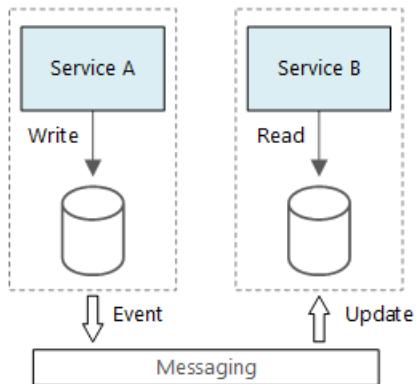
- Consider using the [Event Sourcing](#) pattern to avoid update conflicts.
- Consider using the [Materialized View pattern](#) for the read model, to optimize the schema for queries.

## CQRS in microservices

CQRS can be especially useful in a [microservices architecture](#). One of the principles of microservices is that a service cannot directly access another service's data store.



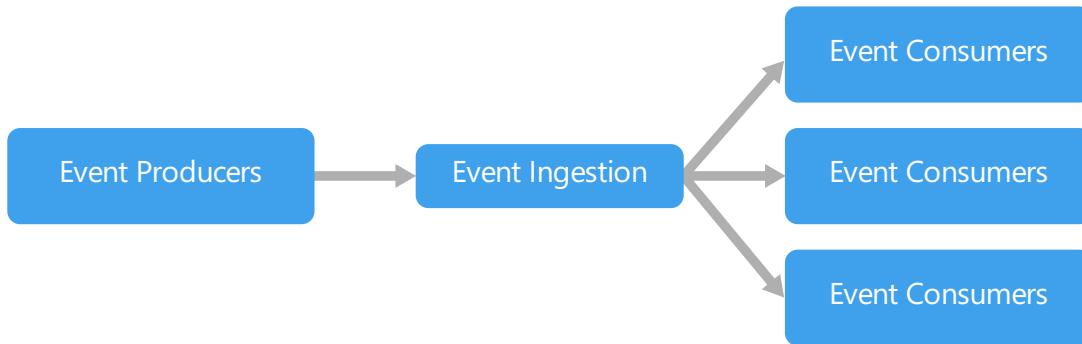
In the following diagram, Service A writes to a data store, and Service B keeps a materialized view of the data. Service A publishes an event whenever it writes to the data store. Service B subscribes to the event.



# Event-driven architecture style

8/30/2018 • 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.

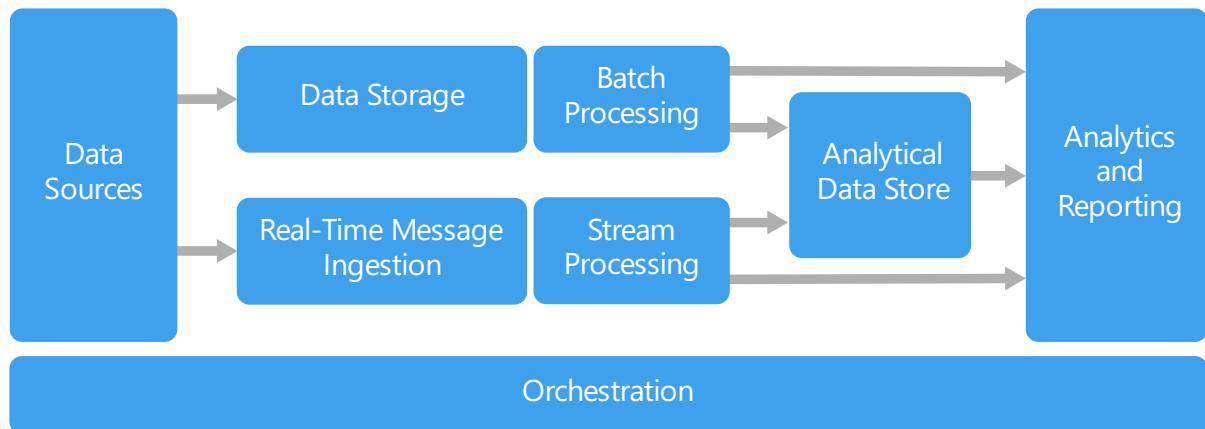
## Challenges

- 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.

# Big data architecture style

8/30/2018 • 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 SQL Data Warehouse 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 as 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 Data Warehouse, 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.

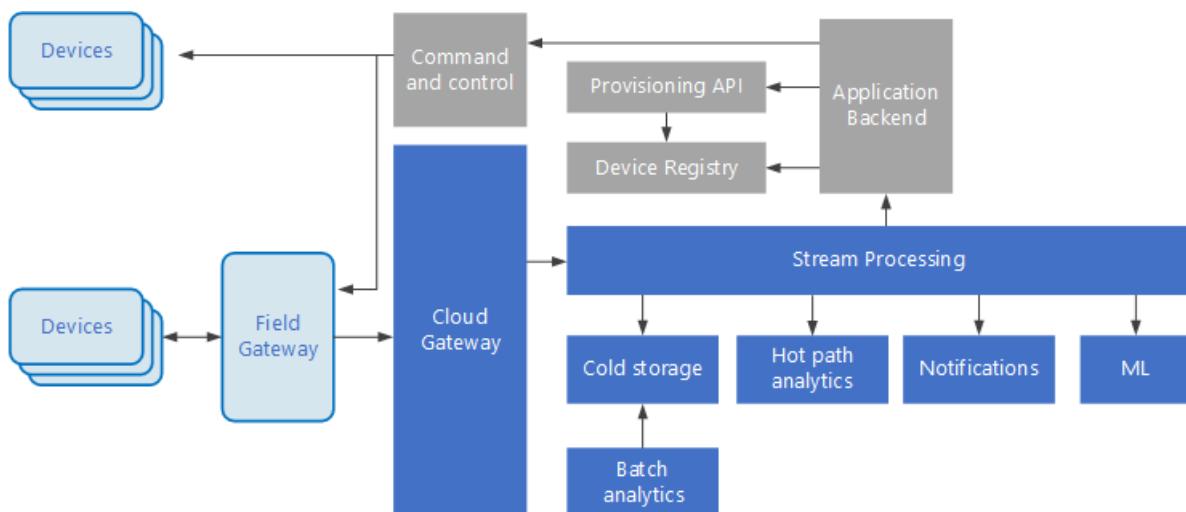
## Best practices

- **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 under-utilized 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.

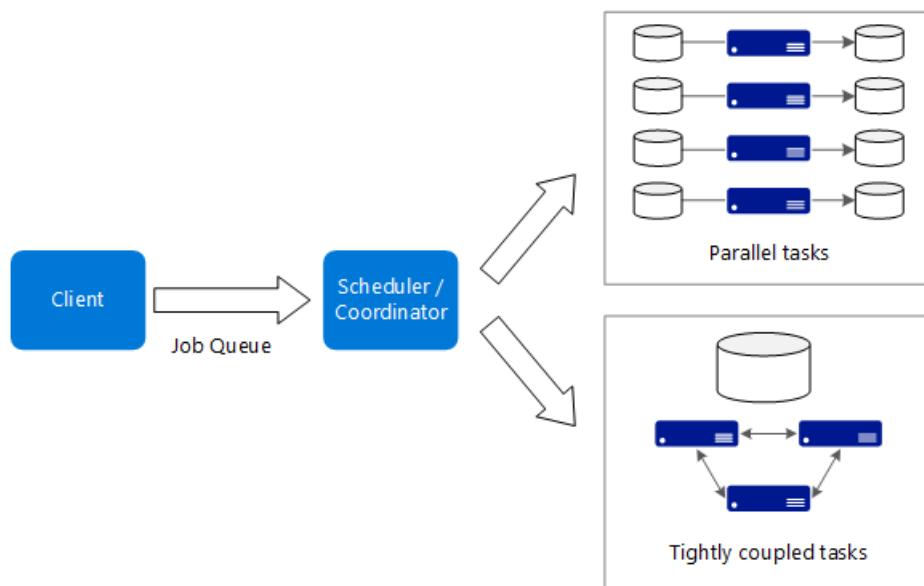
- The **device registry** is a database of the provisioned devices, including the device IDs and usually device metadata, such as location.
- The **provisioning API** is a common external interface for provisioning and registering new devices.
- Some IoT solutions allow **command and control messages**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

# Big compute architecture style

8/30/2018 • 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.

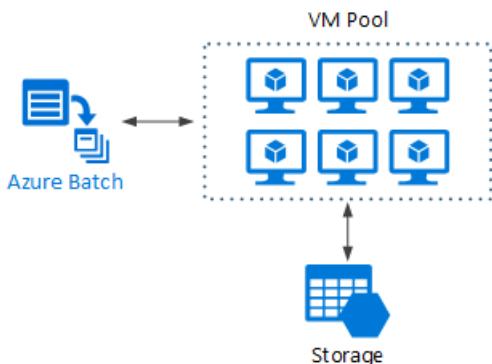
## Challenges

- 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.

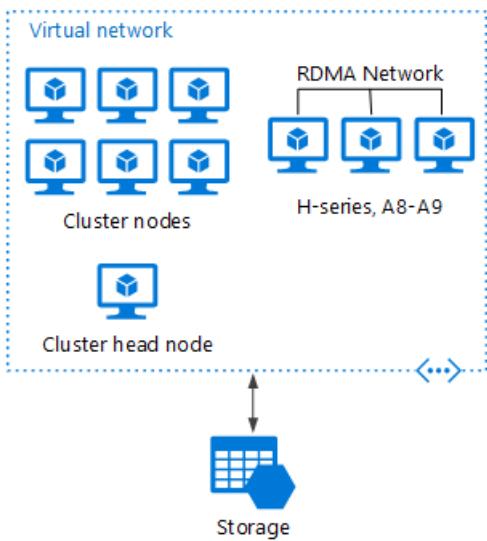


## Big compute running on Virtual Machines

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 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

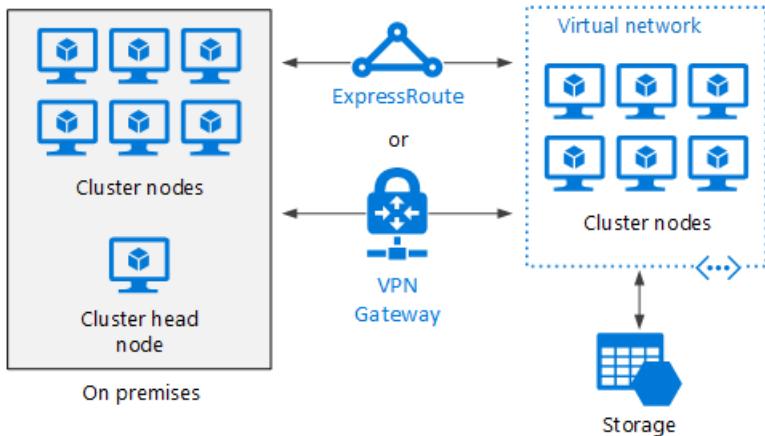
In this scenario, the HPC cluster is created entirely within Azure.



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.



# Overview of Azure compute options

10/5/2018 • 3 minutes to read • [Edit Online](#)

The term *compute* refers to the hosting model for the computing resources that your application runs on.

## Overview

At one end of the spectrum is **Infrastructure-as-a-Service** (IaaS). With IaaS, you provision the VMs that you need, along with associated network 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. For example, instead of creating individual VMs, you specify an instance count, and the service will provision, configure, and manage the necessary resources. Azure App Service is an example of a PaaS service.

There is a spectrum from IaaS to pure PaaS. For example, Azure VMs can auto-scale by using VM Scale Sets. This automatic scaling capability isn't strictly PaaS, but it's the type of management feature that might be found in a PaaS service.

**Functions-as-a-Service** (FaaS) goes even further in removing the need to worry about the hosting environment. Instead of creating compute instances and deploying code to those instances, you simply deploy your code, and the service automatically runs it. You don't need to administer the compute resources. These services make use of serverless architecture, and seamlessly scale up or down to whatever level necessary to handle the traffic. Azure Functions are a FaaS service.

IaaS gives the most control, flexibility, and portability. FaaS provides simplicity, elastic scale, and potential cost savings, because you pay only for the time your code is running. PaaS falls somewhere between the two. In general, the more flexibility a service provides, the more you are responsible for configuring and managing the resources. FaaS services automatically manage nearly all aspects of running an application, while IaaS solutions require you to provision, configure and manage the VMs and network components you create.

## Azure compute options

Here are the main compute options currently available in Azure:

- [Virtual Machines](#) are an IaaS service, allowing you to deploy and manage VMs inside a virtual network (VNet).
- [App Service](#) is a managed PaaS offering for hosting web apps, mobile app back ends, RESTful APIs, or automated business processes.
- [Service Fabric](#) is a distributed systems platform that can run in many environments, including Azure or on premises. Service Fabric is an orchestrator of microservices across a cluster of machines.
- [Azure Container Service](#) lets you create, configure, and manage a cluster of VMs that are preconfigured to run containerized applications.
- [Azure Container Instances](#) offer 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.
- [Azure Functions](#) is a managed FaaS service.
- [Azure Batch](#) is a managed service for running large-scale parallel and high-performance computing (HPC) applications.
- [Cloud Services](#) is a managed service for running cloud applications. It uses a PaaS hosting model.

When selecting a compute option, here are some factors to consider:

- Hosting model. How is the service hosted? What requirements and limitations are imposed by this hosting environment?
- DevOps. Is there built-in support for application upgrades? What is the deployment model?
- Scalability. How does the service handle adding or removing instances? Can it auto-scale based on load and other metrics?
- Availability. What is the service SLA?
- Cost. In addition to the cost of the service itself, consider the operations cost for managing a solution built on that service. For example, IaaS solutions might have a higher operations cost.
- What are the overall limitations of each service?
- What kind of application architectures are appropriate for this service?

## Next steps

To help select a compute service for your application, use the [Decision tree for Azure compute services](#)

For a more detailed comparison of compute options in Azure, see [Criteria for choosing an Azure compute service](#).

# Decision tree for Azure compute services

10/24/2018 • 2 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. 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, looking at aspects such as:

- Feature set
- Service limits
- Cost
- SLA
- [Regional availability](#)
- Developer ecosystem and team skills
- [Compute comparison tables](#)

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

For more information about your options for hosting containers in Azure, see <https://azure.microsoft.com/overview/containers/>.

## Flowchart

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

## Next steps

For additional criteria to consider, see [Criteria for choosing an Azure compute service](#).

# Criteria for choosing an Azure compute service

8/9/2018 • 3 minutes to read • [Edit Online](#)

The term *compute* refers to the hosting model for the computing resources that your applications runs on. The following tables compare Azure compute services across several axes. Refer to these tables when selecting a compute option for your application.

## Hosting model

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

## DevOps

CRITERIA	VIRTUAL MACHINES	APP SERVICE	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Local debugging	Agnostic	IIS Express, others <sup>1</sup>	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	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 (per service)	Deployment slots	Rolling update	Not applicable	

### Notes

- 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.
- See [Resource Manager providers, regions, API versions and schemas](#).

## Scalability

CRITERIA	VIRTUAL MACHINES	APP SERVICE	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Auto-scaling	VM scale sets	Built-in service	VM Scale Sets	Built-in service	Not supported	Not supported	N/A
Load balancer	Azure Load Balancer	Integrated	Azure Load Balancer	Integrated	Integrated	No built-in support	Azure Load Balancer
Scale limit <sup>1</sup>	Platform image: 1000 nodes per VMSS, Custom image: 100 nodes per VMSS	20 instances, 100 with App Service Environment	100 nodes per VMSS	200 instances per Function app	100 nodes per cluster (default limit)	20 container groups per subscription (default limit).	20 core limit (default limit).

### Notes

- See [Azure subscription and service limits, quotas, and constraints](#).

## Availability

CRITERIA	VIRTUAL MACHINES	APP SERVICE	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
SLA	SLA for Virtual Machines	SLA for App Service	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	Not supported	Traffic manager	Not supported	Not Supported

## Other

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

# Choose the right data store

6/11/2018 • 10 minutes to read • [Edit Online](#)

Modern business systems manage increasingly large volumes of data. Data may be ingested from external services, generated by the system itself, or created by users. These data sets may have extremely varied characteristics and processing requirements. Businesses use data to assess trends, trigger business processes, audit their operations, analyze customer behavior, and many other things.

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 towards a specific workload or usage pattern. The term *polyglot persistence* is used to describe solutions that use a mix of data store technologies.

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 *multimodel* 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.

## Relational database management systems

Relational databases organize data as a series of two-dimensional tables with rows and columns. Each table has its own columns, and every row in a table has the same set of columns. This model is mathematically based, and 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 is in contrast to most NoSQL data stores, particularly key/value types, where the schema-on-read model assumes that the client will be imposing its own interpretive schema on data coming out of the database, and is agnostic to the data format being written.

An RDBMS 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, the underlying structures do not lend themselves to scaling out by distributing storage and processing across machines. Also, information stored in an RDBMS, must be put into a relational structure by following the normalization process. While this process is well understood, it can lead to inefficiencies, because of the need to disassemble logical entities into rows in separate tables, and then reassemble the data when running queries.

Relevant Azure service:

- [Azure SQL Database](#)

- [Azure Database for MySQL](#)
- [Azure Database for PostgreSQL](#)

## Key/value 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.

The diagram shows a table representing a key-value store. The table has two columns: 'Key' and 'Value'. The 'Key' column contains values like 'AAAAAA', 'AABAB', 'DFA766', and 'FABCC4'. The 'Value' column contains binary strings. A callout box labeled 'Opaque to data store' points to the 'Value' column, indicating that the stored values are not interpretable by the storage system itself.

Key	Value
AAAAAA	11010011101010011010111...
AABAB	100110000101100110101110...
DFA766	00000000001010101101010...
FABCC4	1110110110101010100101101...

Key/value stores are highly optimized for applications performing simple lookups, but are less suitable for systems that need to query data across different key/value stores. Key/value stores are also not optimized for scenarios where querying by value 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, but key/values stores usually do not have this type of lookup capability for 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:

- [Cosmos DB](#)
- [Azure Redis Cache](#)

## Document databases

A document database is conceptually similar to a key/value store, except that it stores a collection of named fields and data (known as documents), each of which could be simple scalar items or compound elements such as lists and child collections. The data in the fields of a document can be encoded in a variety of ways, including XML, YAML, JSON, BSON, or even stored as plain text. Unlike key/value stores, the fields in 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 may contain information that would be spread across several relational tables in an RDBMS.

A document store does not require that all documents have the same structure. This free-form approach provides a great deal of flexibility. 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"       }

The application can retrieve documents by using the document key. This 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.

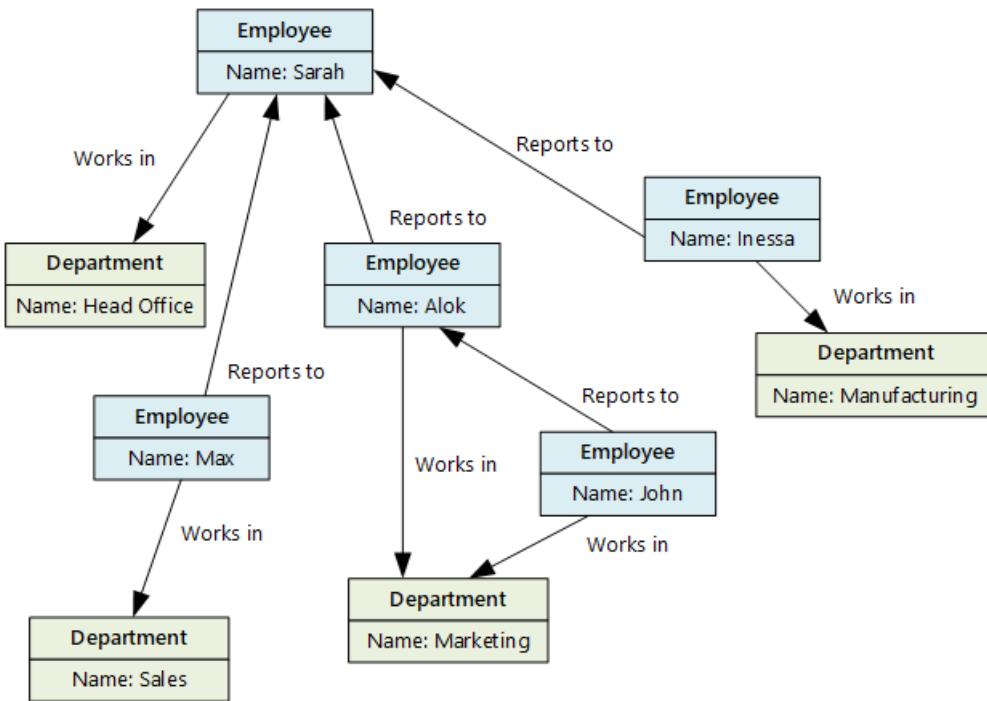
Many document databases support in-place updates, enabling an application to modify the values of specific fields in a document without rewriting the entire document. Read and write operations over multiple fields in a single document are usually atomic.

Relevant Azure service: [Cosmos DB](#)

## Graph databases

A graph database stores two types of information, nodes and edges. You can think of nodes as entities. Edges which specify the relationships between nodes. 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 database 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 database 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 very complex analyses very quickly. Many graph databases provide a query language that you can use to traverse a network of relationships efficiently.

Relevant Azure service: [Cosmos DB](#)

## 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.

Relevant Azure service: [HBase in HDInsight](#)

## Data analytics

Data analytics stores provide massively parallel solutions for ingesting, storing, and analyzing data. This data is distributed across multiple servers using a share-nothing architecture to maximize scalability and minimize dependencies. The data is unlikely to be static, so these stores must be able to handle large quantities of information, arriving in a variety of formats from multiple streams, while continuing to process new queries.

Relevant Azure services:

- [SQL Data Warehouse](#)
- [Azure Data Lake](#)

## Search Engine Databases

A search engine database supports the ability to search for information held in external data stores and services. A search engine database can be used to index massive volumes of data and provide near real-time access to these indexes. Although search engine databases are commonly thought of as being synonymous with the web, many large-scale systems use them to provide structured and ad-hoc search capabilities on top of their own databases.

The key characteristics of a search engine database are the ability to store and index information very quickly, and provide fast response times for search requests. 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).

Relevant Azure service: [Azure Search](#)

## Time Series Databases

Time series data is a set of values organized by time, and a time series database is a database that is optimized for this type of data. Time series databases 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. 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.

Time series databases are good for storing telemetry data. Scenarios include IoT sensors or application/system counters.

Relevant Azure service: [Time Series Insights](#)

## 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). Objects in these store types are composed of the stored data, some metadata, and a unique ID for accessing the object. Object stores enables the management of extremely large amounts of unstructured data.

Relevant Azure service: [Blob Storage](#)

## 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.

Relevant Azure service: [File Storage](#)

# Criteria for choosing a data store

6/11/2018 • 8 minutes to read • [Edit Online](#)

Azure supports many types of data storage solutions, each providing different features and capabilities. 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

To start your comparison, gather as much of the following information as you can about your data needs. This information will help you to determine which data storage types will meet your needs.

### 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?

The following sections compare various data store models in terms of workload profile, data types, and example use cases.

## Relational database management systems (RDBMS)

<b>Workload</b>	<ul style="list-style-type: none"><li>● Both the creation of new records and updates to existing data happen regularly.</li><li>● Multiple operations have to be completed in a single transaction.</li><li>● Requires aggregation functions to perform cross-tabulation.</li><li>● Strong integration with reporting tools is required.</li><li>● Relationships are enforced using database constraints.</li><li>● Indexes are used to optimize query performance.</li><li>● Allows access to specific subsets of data.</li></ul>
-----------------	--

<b>Data type</b>	<ul style="list-style-type: none"> <li>• Data is highly normalized.</li> <li>• Database schemas are required and enforced.</li> <li>• Many-to-many relationships between data entities in the database.</li> <li>• Constraints are defined in the schema and imposed on any data in the database.</li> <li>• Data requires high integrity. Indexes and relationships need to be maintained accurately.</li> <li>• Data requires strong consistency. Transactions operate in a way that ensures all data are 100% consistent for all users and processes.</li> <li>• Size of individual data entries is intended to be small to medium-sized.</li> </ul>
<b>Examples</b>	<ul style="list-style-type: none"> <li>• Line of business (human capital management, customer relationship management, enterprise resource planning)</li> <li>• Inventory management</li> <li>• Reporting database</li> <li>• Accounting</li> <li>• Asset management</li> <li>• Fund management</li> <li>• Order management</li> </ul>

## Document databases

<b>Workload</b>	<ul style="list-style-type: none"> <li>• General purpose.</li> <li>• Insert and update operations are common. Both the creation of new records and updates to existing data happen regularly.</li> <li>• No object-relational impedance mismatch. Documents can better match the object structures used in application code.</li> <li>• Optimistic concurrency is more commonly used.</li> <li>• Data must be modified and processed by consuming application.</li> <li>• Data requires index on multiple fields.</li> <li>• Individual documents are retrieved and written as a single block.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>• Data can be managed in de-normalized way.</li> <li>• Size of individual document data is relatively small.</li> <li>• Each document type can use its own schema.</li> <li>• Documents can include optional fields.</li> <li>• Document data is semi-structured, meaning that data types of each field are not strictly defined.</li> <li>• Data aggregation is supported.</li> </ul>

<b>Examples</b>	<ul style="list-style-type: none"> <li>• Product catalog</li> <li>• User accounts</li> <li>• Bill of materials</li> <li>• Personalization</li> <li>• Content management</li> <li>• Operations data</li> <li>• Inventory management</li> <li>• Transaction history data</li> <li>• Materialized view of other NoSQL stores. Replaces file/BLOB indexing.</li> </ul>
-----------------	--

## Key/value stores

<b>Workload</b>	<ul style="list-style-type: none"> <li>• Data is identified and accessed using a single ID key, like a dictionary.</li> <li>• Massively scalable.</li> <li>• No joins, lock, or unions are required.</li> <li>• No aggregation mechanisms are used.</li> <li>• Secondary indexes are generally not used.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>• Data size tends to be large.</li> <li>• Each key is associated with a single value, which is an unmanaged data BLOB.</li> <li>• There is no schema enforcement.</li> <li>• No relationships between entities.</li> </ul>
<b>Examples</b>	<ul style="list-style-type: none"> <li>• Data caching</li> <li>• Session management</li> <li>• User preference and profile management</li> <li>• Product recommendation and ad serving</li> <li>• Dictionaries</li> </ul>

## Graph databases

<b>Workload</b>	<ul style="list-style-type: none"> <li>• The relationships between data items are very complex, involving many hops between related data items.</li> <li>• The relationship between data items are dynamic and change over time.</li> <li>• Relationships between objects are first-class citizens, without requiring foreign-keys and joins to traverse.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>• Data is comprised of nodes and relationships.</li> <li>• Nodes are similar to table rows or JSON documents.</li> <li>• Relationships are just as important as nodes, and are exposed directly in the query language.</li> <li>• Composite objects, such as a person with multiple phone numbers, tend to be broken into separate, smaller nodes, combined with traversable relationships</li> </ul>

<b>Examples</b>	<ul style="list-style-type: none"> <li>• Organization charts</li> <li>• Social graphs</li> <li>• Fraud detection</li> <li>• Analytics</li> <li>• Recommendation engines</li> </ul>
-----------------	--

## Column-family databases

<b>Workload</b>	<ul style="list-style-type: none"> <li>• Most column-family databases perform write operations extremely quickly.</li> <li>• Update and delete operations are rare.</li> <li>• Designed to provide high throughput and low-latency access.</li> <li>• Supports easy query access to a particular set of fields within a much larger record.</li> <li>• Massively scalable.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>• Data is stored in tables consisting of a key column and one or more column families.</li> <li>• Specific columns can vary by individual rows.</li> <li>• Individual cells are accessed via get and put commands</li> <li>• Multiple rows are returned using a scan command.</li> </ul>
<b>Examples</b>	<ul style="list-style-type: none"> <li>• Recommendations</li> <li>• Personalization</li> <li>• Sensor data</li> <li>• Telemetry</li> <li>• Messaging</li> <li>• Social media analytics</li> <li>• Web analytics</li> <li>• Activity monitoring</li> <li>• Weather and other time-series data</li> </ul>

## Search engine databases

<b>Workload</b>	<ul style="list-style-type: none"> <li>• Indexing data from multiple sources and services.</li> <li>• Queries are ad-hoc and can be complex.</li> <li>• Requires aggregation.</li> <li>• Full text search is required.</li> <li>• Ad hoc self-service query is required.</li> <li>• Data analysis with index on all fields is required.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>• Semi-structured or unstructured</li> <li>• Text</li> <li>• Text with reference to structured data</li> </ul>

<b>Examples</b>	<ul style="list-style-type: none"> <li>• Product catalogs</li> <li>• Site search</li> <li>• Logging</li> <li>• Analytics</li> <li>• Shopping sites</li> </ul>
-----------------	---

## Data warehouse

<b>Workload</b>	<ul style="list-style-type: none"> <li>• Data analytics</li> <li>• Enterprise BI</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>• Historical data from multiple sources.</li> <li>• Usually denormalized in a "star" or "snowflake" schema, consisting of fact and dimension tables.</li> <li>• Usually loaded with new data on a scheduled basis.</li> <li>• Dimension tables often include multiple historic versions of an entity, referred to as a <i>slowly changing dimension</i>.</li> </ul>
<b>Examples</b>	An enterprise data warehouse that provides data for analytical models, reports, and dashboards.

## Time series databases

<b>Workload</b>	<ul style="list-style-type: none"> <li>• An overwhelming proportion of operations (95-99%) are writes.</li> <li>• Records are generally appended sequentially in time order.</li> <li>• Updates are rare.</li> <li>• Deletes occur in bulk, and are made to contiguous blocks or records.</li> <li>• Read requests can be larger than available memory.</li> <li>• It's common for multiple reads to occur simultaneously.</li> <li>• Data is read sequentially in either ascending or descending time order.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>• A time stamp that is used as the primary key and sorting mechanism.</li> <li>• Measurements from the entry or descriptions of what the entry represents.</li> <li>• Tags that define additional information about the type, origin, and other information about the entry.</li> </ul>
<b>Examples</b>	<ul style="list-style-type: none"> <li>• Monitoring and event telemetry.</li> <li>• Sensor or other IoT data.</li> </ul>

## Object storage

<b>Workload</b>	<ul style="list-style-type: none"> <li>Identified by key.</li> <li>Objects may be publicly or privately accessible.</li> <li>Content is typically an asset such as a spreadsheet, image, or video file.</li> <li>Content must be durable (persistent), and external to any application tier or virtual machine.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>Data size is large.</li> <li>Blob data.</li> <li>Value is opaque.</li> </ul>
<b>Examples</b>	<ul style="list-style-type: none"> <li>Images, videos, office documents, PDFs</li> <li>CSS, Scripts, CSV</li> <li>Static HTML, JSON</li> <li>Log and audit files</li> <li>Database backups</li> </ul>

## Shared files

<b>Workload</b>	<ul style="list-style-type: none"> <li>Migration from existing apps that interact with the file system.</li> <li>Requires SMB interface.</li> </ul>
<b>Data type</b>	<ul style="list-style-type: none"> <li>Files in a hierarchical set of folders.</li> <li>Accessible with standard I/O libraries.</li> </ul>
<b>Examples</b>	<ul style="list-style-type: none"> <li>Legacy files</li> <li>Shared content accessible among a number of VMs or app instances</li> </ul>

# Ten design principles for Azure applications

8/30/2018 • 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

8/30/2018 • 4 minutes to read • [Edit Online](#)

## Design your application to be self healing when failures occur

In a distributed system, failures 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 [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 [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 resilient applications for Azure](#).

# Make all things redundant

9/28/2018 • 2 minutes to read • [Edit Online](#)

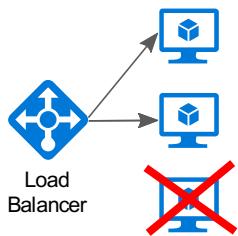
## Build redundancy into your application, to avoid having single points of failure

A resilient application routes around failure. Identify the critical paths in your application. Is there redundancy at each point in the path? If 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 fallback. 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](#).

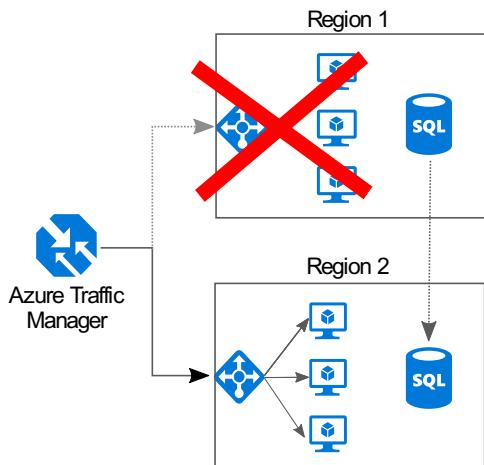


**Replicate databases.** Azure SQL Database and Cosmos DB automatically replicate the data within a region, and you can enable geo-replication across regions. If you are using an IaaS database solution, choose one that supports replication and failover, such as [SQL Server Always On Availability Groups](#).

**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 fallback.** Use Traffic Manager for automatic failover, but not for automatic fallback. Automatic fallback 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 fallback. If the Azure Traffic Manager service fails, change your CNAME records in DNS to point to the other traffic management service.

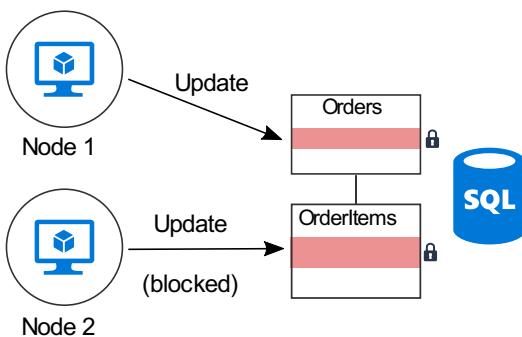
# Minimize coordination

8/30/2018 • 4 minutes to read • [Edit Online](#)

## Minimize coordination between application services to achieve scalability

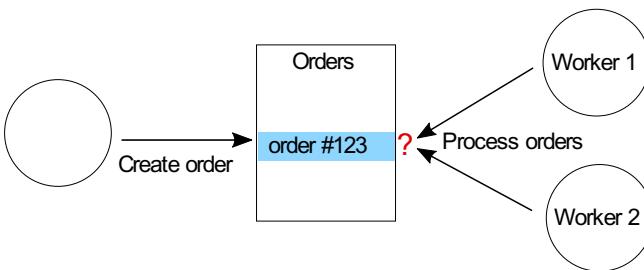
Most cloud applications consist of multiple application services — web front ends, databases, business processes, reporting and analysis, and so on. To achieve scalability and reliability, each of those services should run on multiple instances.

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. For more information, see [CQRS architecture style](#).

**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

8/30/2018 • 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

8/30/2018 • 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 hot spots.** 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

8/30/2018 • 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 managed services

8/30/2018 • 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. Managed services 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 managed services. These include cache, queues, and data storage.

INSTEAD OF RUNNING...	CONSIDER USING...
Active Directory	Azure Active Directory Domain Services
Elasticsearch	Azure Search
Hadoop	HDInsight
IIS	App Service
MongoDB	Cosmos DB
Redis	Azure Redis Cache
SQL Server	Azure SQL Database

# Use the best data store for the job

8/30/2018 • 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 skill set 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

10/8/2018 • 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

8/30/2018 • 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 very 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 (CAPEX). 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. See [Azure pricing](#) for more information. 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.

# Pillars of software quality

8/30/2018 • 10 minutes to read • [Edit Online](#)

A successful cloud application will focus on these five pillars of software quality: Scalability, availability, resiliency, management, and security.

PILLAR	DESCRIPTION
Scalability	The ability of a system to handle increased load.
Availability	The proportion of time that a system is functional and working.
Resiliency	The ability of a system to recover from failures and continue to function.
Management	Operations processes that keep a system running in production.
Security	Protecting applications and data from threats.

## Scalability

Scalability is the ability of a system to handle increased load. There are two main ways that an application can scale. Vertical scaling (*scaling up*) means increasing the capacity of a resource, for example by using a larger VM size. Horizontal scaling (*scaling out*) is adding new instances of a resource, such as VMs or database replicas.

Horizontal scaling has significant advantages over vertical scaling:

- True cloud scale. Applications can be designed to run on hundreds or even thousands of nodes, reaching scales that are not possible on a single node.
- Horizontal scale is elastic. You can add more instances if load increases, or remove them 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 any up any more. 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 be able to 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 auto-scaling 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 simply 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 would then need to consider additional 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 [Scalability checklist](#) to review your design from a scalability standpoint.

## Scalability guidance

- [Design patterns for scalability and performance](#)
- Best practices: [Autoscaling](#), [Background jobs](#), [Caching](#), [CDN](#), [Data partitioning](#)

## Availability

Availability is the proportion of time that the system is functional and working. It is usually measured as a percentage of uptime. Application errors, infrastructure problems, and system load can all reduce availability.

A cloud application should have a service level objective (SLO) that clearly defines the expected availability, and how the availability is measured. When defining availability, look at the critical path. The web front-end might be able to service client requests, but if every transaction fails because it can't connect to the database, the application is not available to users.

Availability is often described in terms of "9s" — for example, "four 9s" means 99.99% uptime. The following table shows the potential cumulative downtime at different availability levels.

% UPTIME	DOWNTIME PER WEEK	DOWNTIME PER MONTH	DOWNTIME PER YEAR
99%	1.68 hours	7.2 hours	3.65 days
99.9%	10 minutes	43.2 minutes	8.76 hours
99.95%	5 minutes	21.6 minutes	4.38 hours
99.99%	1 minute	4.32 minutes	52.56 minutes
99.999%	6 seconds	26 seconds	5.26 minutes

Notice that 99% uptime could translate to an almost 2-hour service outage per week. For many applications, especially consumer-facing applications, that is not an acceptable SLO. On the other hand, five 9s (99.999%) means no more than 5 minutes of downtime in a year. It's challenging enough just detecting an outage that quickly, let alone resolving the issue. To get very high availability (99.99% or higher), you can't rely on manual intervention to recover from failures. The application must be self-diagnosing and self-healing, which is where resiliency becomes crucial.

In Azure, the Service Level Agreement (SLA) describes Microsoft's commitments for uptime and connectivity. If the SLA for a particular service is 99.95%, it means you should expect the service to be available 99.95% of the time.

Applications often depend on multiple services. In general, the probability of either service having downtime is independent. For example, suppose your application depends on two services, each with a 99.9% SLA. The composite SLA for both services is  $99.9\% \times 99.9\% \approx 99.8\%$ , or slightly less than each service by itself.

Use the [Availability checklist](#) to review your design from an availability standpoint.

## Availability guidance

- [Design patterns for availability](#)
- Best practices: [Autoscaling](#), [Background jobs](#)

# Resiliency

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. Resiliency is closely related to availability.

In traditional application development, there has been a focus on reducing mean time between failures (MTBF). Effort was spent trying to prevent the system from failing. In cloud computing, a different mindset is required, due to 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 the use of 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, both within a region and across regions.
- Azure Managed Disks are automatically placed in different storage scale units, to limit the effects of hardware failures.
- 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.

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 big 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 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? This 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?

Use the [Resiliency checklist](#) to review your design from a resiliency standpoint.

## Resiliency guidance

- [Designing resilient applications for Azure](#)
- [Design patterns for resiliency](#)
- Best practices: [Transient fault handling](#), [Retry guidance for specific services](#)

# Management and DevOps

This pillar covers the operations processes that keep an application running in production.

Deployments must be 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.

Monitoring and diagnostics are crucial. Cloud applications run in a remote datacenter where you do not have full

control of the infrastructure or, in some cases, the operating system. In a large application, it's not practical to log into VMs to troubleshoot an issue or sift through log files. With PaaS services, there may not even 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.

Use the [DevOps checklist](#) to review your design from a management and DevOps standpoint.

## Management and DevOps guidance

- [Design patterns for management and monitoring](#)
- Best practices: [Monitoring and diagnostics](#)

# Security

You must think about security throughout the entire lifecycle of an application, from design and implementation to deployment and operations. The Azure platform provides protections against a variety of threats, such as network intrusion and DDoS attacks. But you still need to build security into your application and into your DevOps processes.

Here are some broad security areas to consider.

## 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.

## Protecting your infrastructure

Control access to the Azure resources that you deploy. Every Azure subscription has a [trust relationship](#) with an Azure AD tenant. Use [Role-Based Access Control](#) (RBAC) to grant users within your organization the correct permissions to Azure resources. Grant access by assigning RBAC role 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 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's highly available. 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 SQL Data Warehouse](#), and [Cosmos DB](#).

## Security resources

- [Azure Security Center](#) provides integrated security monitoring and policy management across your Azure subscriptions.
- [Azure Security Documentation](#)
- [Microsoft Trust Center](#)

This guide presents a structured approach for designing data-centric solutions on Microsoft Azure. It is based on proven practices derived from customer engagements.

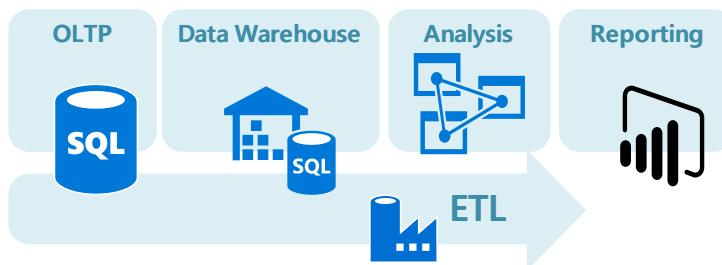
## Introduction

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.

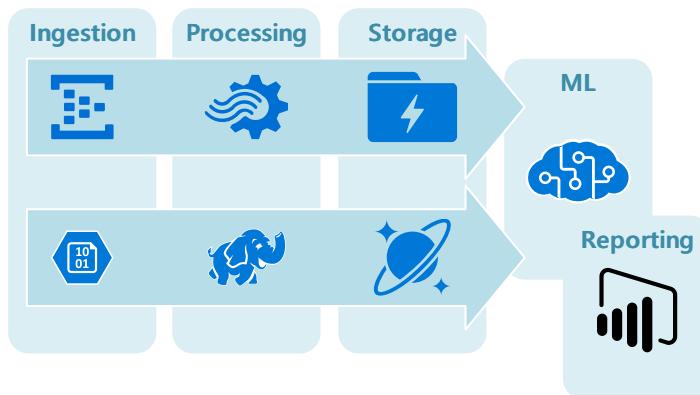
## How this guide is structured

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 pre-defined 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.)



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, th

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.

# Traditional relational database solutions

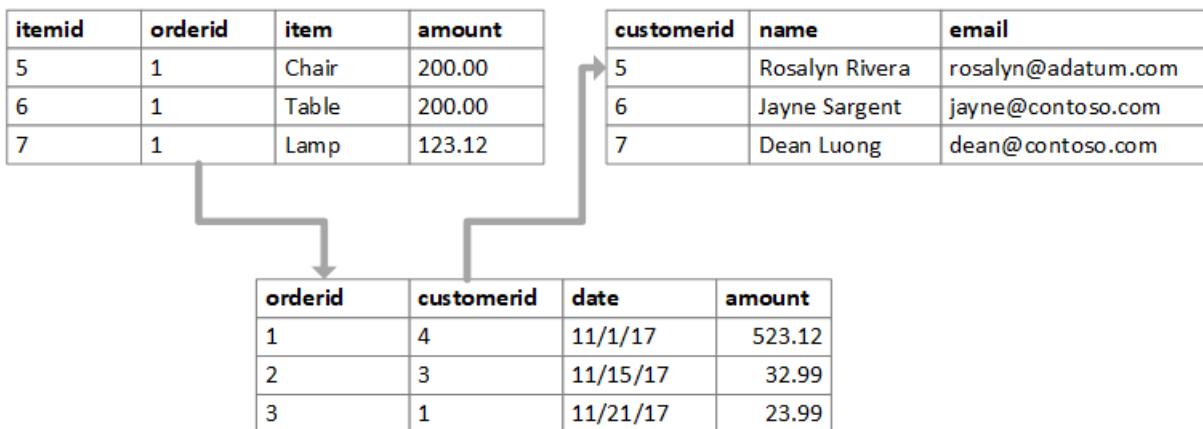
4/11/2018 • 2 minutes to read • [Edit Online](#)

Relational data is data modeled using the relational model. In this model, data is expressed as tuples. A *tuple* is a set of attribute/value pairs. For example, a tuple might be (itemid = 5, orderid = 1, item = "Chair", amount = 200.00). A set of tuples that all share the same attributes is called a *relation*.

Relations are naturally represented as tables, where each tuple is exposed as a row in the table. However, rows have an explicit ordering, unlike tuples. The database schema defines the columns (headings) of each table. Each column is defined with a name and a data type for all values stored in that column across all rows in the table.

itemid	orderid	item	amount
5	1	Chair	200.00
6	1	Table	200.00
7	1	Lamp	123.12

A data store that organizes data using the relational model is referred to as a relational database. Primary keys uniquely identify rows within a table. Foreign key fields are used in one table to refer to a row in another table by referencing the primary key of the other table. Foreign keys are used to maintain referential integrity, ensuring that the referenced rows are not altered or deleted while the referencing row depends on them.



Relational databases support various types of constraints that help to ensure data integrity:

- Unique constraints ensure that all values in a column are unique.
- Foreign key constraints enforce a link between the data in two tables. A foreign key references the primary key or another unique key from another table. A foreign key constraint enforces referential integrity, disallowing changes that cause invalid foreign key values.
- Check constraints, also known as entity integrity constraints, limit the values that can be stored within a single column, or in relationship to values in other columns of the same row.

Most relational databases use the Structured Query Language (SQL) language that enables a declarative approach to querying. The query describes the desired result, but not the steps to execute the query. The engine then decides the best way to execute the query. This differs from a procedural approach, where the query program specifies the processing steps explicitly. However, relational databases can store executable code routines in the form of stored procedures and functions, which enables a mixture of declarative and procedural approaches.

To improve query performance, relational databases use *indexes*. Primary indexes, which are used by the primary key, define the order of the data as it sits on disk. Secondary indexes provide an alternative combination of fields,

so the desired rows can be queried efficiently, without having to re-sort the entire data on disk.

Because relational databases enforce referential integrity, scaling a relational database can become challenging. That's because any query or insert operation might touch any number of tables. You can scale out a relational database by *sharding* the data, but this requires careful design of the schema. For more information, see [Sharding pattern](#).

If data is non-relational or has requirements that are not suited to a relational database, consider a [Non-relational](#) or [NoSQL](#) data store.

# Online transaction processing (OLTP)

4/11/2018 • 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 multi-tenant? 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

	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
Programmability <sup>1</sup>	T-SQL, .NET, R	T-SQL, .NET, R, Python	T-SQL, .NET, R, Python	SQL

[1] Not including client driver support, which allows many programming languages to connect to and use the OLTP data store.

## Scalability capabilities

	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	1 TB	1 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

	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

	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Readable secondaries	Yes	Yes	No	No
Geographic replication	Yes	Yes	No	No
Automatic failover to secondary	Yes	No	No	No
Point-in-time restore	Yes	Yes	Yes	Yes

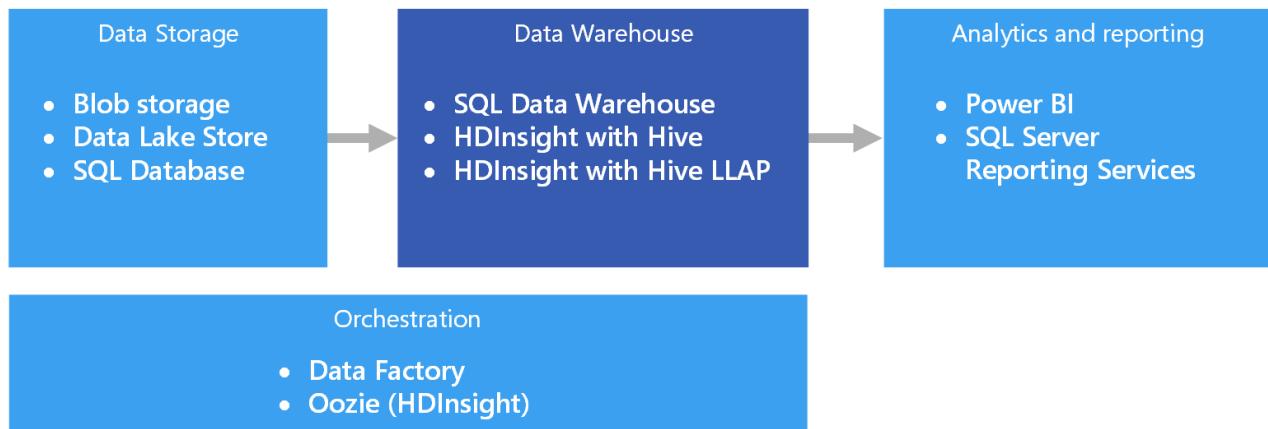
## Security capabilities

	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	No	No
Azure Active Directory authentication	Yes	Yes	No	No
Active Directory authentication	No	Yes	No	No
Multi-factor authentication	Yes	Yes	No	No
Supports <a href="#">Always Encrypted</a>	Yes	Yes	Yes	No
Private IP	No	Yes	Yes	No

# Data warehousing and data marts

6/11/2018 • 12 minutes to read • [Edit Online](#)

A data warehouse is a central, organizational, relational repository of integrated data from one or more disparate sources, across many or all subject areas. Data warehouses store current and historical data and are used for reporting and analysis of the data in different ways.



To move data into a data warehouse, it is extracted on a periodic basis from various sources that contain important business information. As the data is moved, it can be formatted, cleaned, validated, summarized, and reorganized. Alternately, 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 storage space for data used for reporting, analysis, and forming important business decisions using business intelligence (BI) tools.

## Data marts and operational data stores

Managing data at scale is complex, and it is becoming less common to have a single data warehouse that represents all data across the entire enterprise. Instead, organizations create smaller, more focused data warehouses, called *data marts*, that expose the desired data for analytics purposes. An orchestration process populates the data marts from data maintained in an operational data store. The operational data store acts as an intermediary between the source transactional system and the data mart. Data managed by the operational data store is a cleaned version of the data present in the source transactional system, and is typically a subset of the historical data that is maintained by the data warehouse or data mart.

## 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, current, and accurate. Data warehouses do not need to follow the same terse data structure you may be using in your operational/OLTP databases. You can use column names that make sense to business users and analysts, restructure the schema to simplify data relationships, and consolidate several tables into one. These steps help guide users who need to create ad hoc reports, or 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, common keys, common data models, and common access methods.

Data warehouses are optimized for read access, resulting in faster report generation compared to running reports against the source transaction system. In addition, data warehouses provide the following benefits:

- All historical data from multiple sources can be stored and accessed from a data warehouse as the single source of truth.
- You can improve data quality by cleaning up data as it is imported into the data warehouse, providing more accurate data as well as providing consistent codes and descriptions.
- Reporting tools do not compete with the transactional source systems for query processing cycles. A data warehouse allows the transactional system to focus predominantly on handling writes, while the data warehouse satisfies the majority of read requests.
- A data warehouse can help consolidate data from different software.
- Data mining tools can help you find hidden patterns using automatic methodologies against data stored in your warehouse.
- Data warehouses make it easier to provide secure access to authorized users, while restricting access to others. There is no need to grant business users access to the source data, thereby removing a potential attack vector against one or more production transaction systems.
- Data warehouses make it easier to create business intelligence solutions on top of the data, 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. This is an important step, as data warehouses are information driven, where concept mapping drives the rest of the project. This involves standardizing business-related terms and common formats (such as currency and dates), and restructuring 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. Consideration include how to copy data from the source transactional system to the data warehouse, and when to move historical data out of your operational data stores and into the warehouse.
- Maintaining or improving data quality by cleaning the data as it is imported into the warehouse.

## Data warehousing in Azure

In Azure, you may have one or more sources of data, whether from customer transactions, or from various business applications used by various departments. 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 or data mart. In Azure, this analytical store capability can be met with Azure SQL Data Warehouse, or with Azure HDInsight using Hive or Interactive Query. In addition, you will need some level of orchestration to periodically 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 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 are likely to impose a performance penalty with small data sizes, due to the way 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 less than this, 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 Store](#). 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, MPP solutions require a different skillset, due to variances in querying, modeling, partitioning of data, and other factors unique to parallel processing.

When deciding which SMP solution to use, see [A closer look at Azure SQL Database and SQL Server on Azure VMs](#).

Azure SQL Data Warehouse can also be used for small and medium datasets, where the workload is compute and memory intensive. Read more about SQL Data Warehouse patterns and common scenarios:

- [SQL Data Warehouse Patterns and Anti-Patterns](#)
- [SQL Data Warehouse Loading Patterns and Strategies](#)
- [Migrating Data to Azure SQL Data Warehouse](#)
- [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 SQL Data Warehouse or one of the other options. For structured data, SQL Data Warehouse 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 multi-tenancy requirement? If so, SQL Data Warehouse is not ideal for this requirement. For more information, see [SQL Data Warehouse Patterns and Anti-Patterns](#).
- Do you prefer a relational data store? If so, narrow your options to those 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, narrow your options to those that can support 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.
  - SQL Data Warehouse has limits on concurrent queries and concurrent connections. For more information, see [Concurrency and workload management in SQL Data Warehouse](#). Consider using complementary services, such as [Azure Analysis Services](#), to overcome limits in SQL Data Warehouse.
- 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

	AZURE SQL DATABASE	SQL SERVER (VM)	SQL DATA WAREHOUSE	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Is managed service	Yes	No	Yes	Yes <sup>1</sup>	Yes <sup>1</sup>
Requires data orchestration (holds copy of data/historical data)	No	No	Yes	Yes	Yes
Easily integrate multiple data sources	No	No	Yes	Yes	Yes

	AZURE SQL DATABASE	SQL SERVER (VM)	SQL DATA WAREHOUSE	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Supports pausing compute	No	No	Yes	No <sup>2</sup>	No <sup>2</sup>
Relational data store	Yes	Yes	Yes	No	No
Real-time reporting	Yes	Yes	No	No	Yes
Flexible backup restore points	Yes	Yes	No <sup>3</sup>	Yes <sup>4</sup>	Yes <sup>4</sup>
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 SQL Data Warehouse, 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 Store 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

	AZURE SQL DATABASE	SQL SERVER (VM)	SQL DATA WAREHOUSE	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 <sup>1</sup>	No	No
Supports in-memory caching of data	Yes	Yes	No	Yes	Yes

[1] SQL Data Warehouse allows you to scale up or down by adjusting the number of data warehouse units (DWUs). See [Manage compute power in Azure SQL Data Warehouse](#).

## Security capabilities

	AZURE SQL DATABASE	SQL SERVER IN A VIRTUAL MACHINE	SQL DATA WAREHOUSE	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 <sup>1</sup>	local / Azure AD <sup>1</sup>
Authorization	Yes	Yes	Yes	Yes	Yes <sup>1</sup>
Auditing	Yes	Yes	Yes	Yes	Yes <sup>1</sup>
Data encryption at rest	Yes <sup>2</sup>	Yes <sup>2</sup>	Yes <sup>2</sup>	Yes <sup>2</sup>	Yes <sup>1</sup>
Row-level security	Yes	Yes	Yes	No	Yes <sup>1</sup>
Supports firewalls	Yes	Yes	Yes	Yes	Yes <sup>3</sup>
Dynamic data masking	Yes	Yes	Yes	No	Yes <sup>1</sup>

[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:

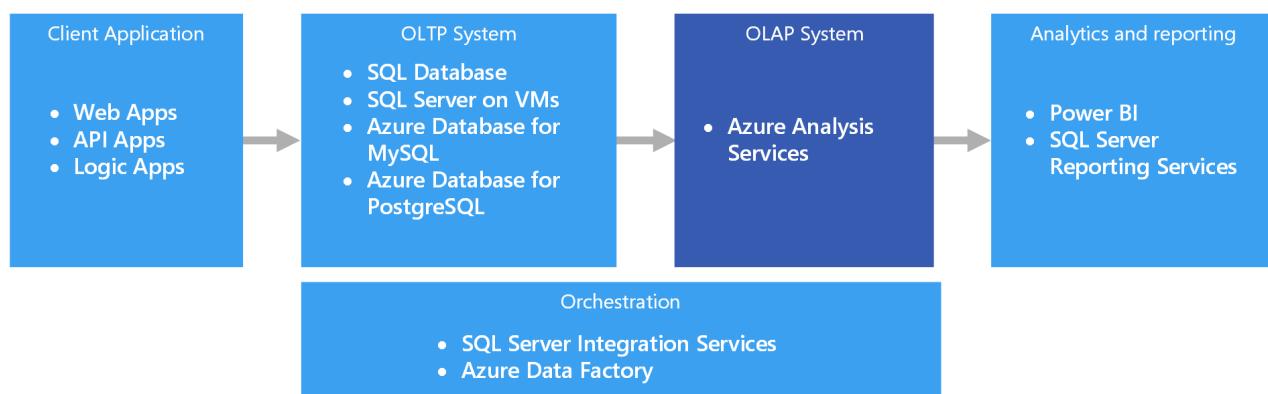
- [Securing your SQL Database](#)
- [Secure a database in SQL Data Warehouse](#)
- [Extend Azure HDInsight using an Azure Virtual Network](#)
- [Enterprise-level Hadoop security with domain-joined HDInsight clusters](#)

# Online analytical processing (OLAP)

4/11/2018 • 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

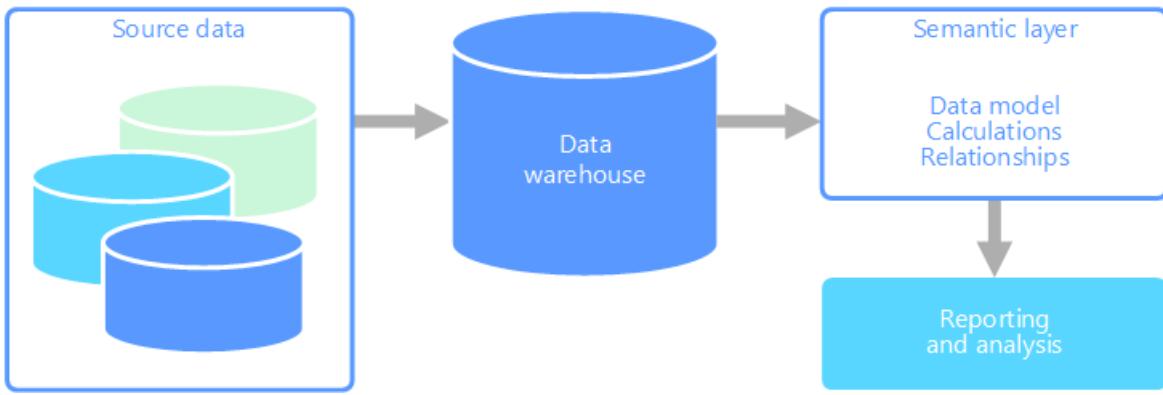
A semantic data model is a conceptual model that describes the meaning of the data elements it contains. Organizations often have their own terms for things, sometimes with synonyms, or even different meanings for the same term. For example, an inventory database might track a piece of equipment with 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)
Workload	Heavy reads, read-only

Requirement	Description
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

	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

	AZURE ANALYSIS SERVICES	SQL SERVER ANALYSIS SERVICES	SQL SERVER WITH COLUMNSTORE INDEXES	AZURE SQL DATABASE WITH COLUMNSTORE INDEXES
Supports multidimensional cubes	No	Yes	No	No
Supports tabular semantic models	Yes	Yes	No	No
Easily integrate multiple data sources	Yes	Yes	No <sup>1</sup>	No <sup>1</sup>
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 <sup>2</sup>	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

	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	No
Dynamic scalability (scale up)	Yes	No	Yes	No

# Extract, transform, and load (ETL)

4/11/2018 • 5 minutes to read • [Edit Online](#)

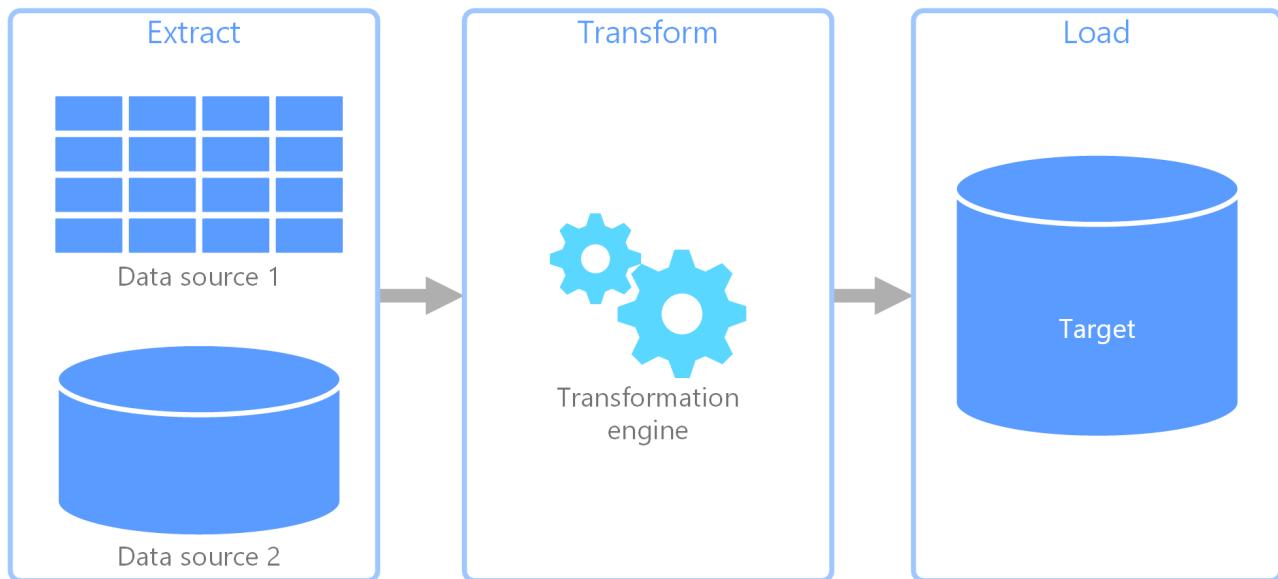
A common problem that organizations face is how to gather data from multiple sources, in multiple formats, and move it to one or more data stores. The destination may not be the same type of data store as the source, and often the format is different, or the data needs to be shaped or cleaned before 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)

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:

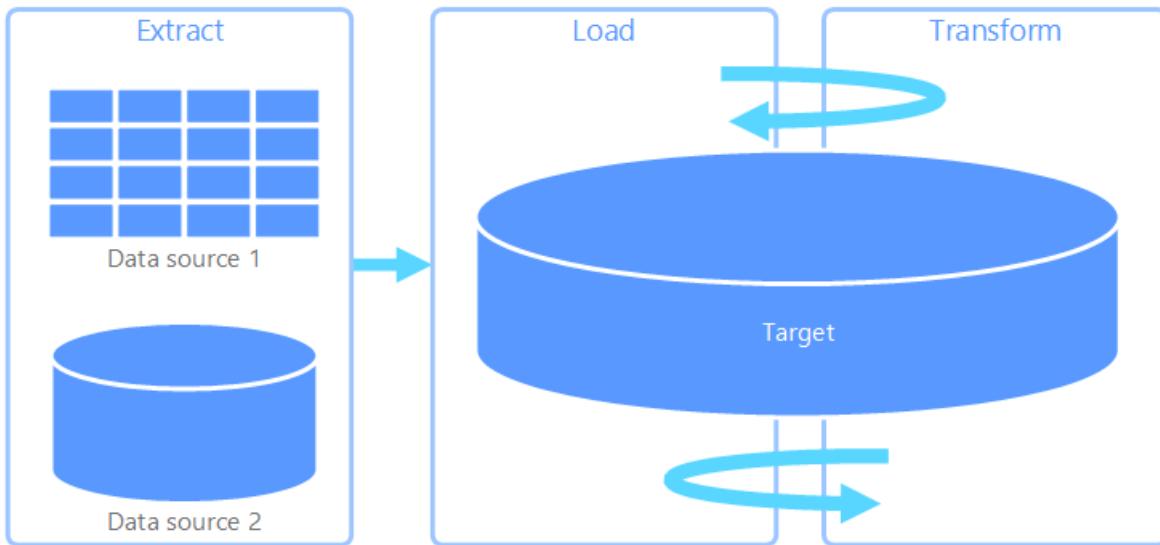
- [Azure Data Factory v2](#)

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 SQL Data Warehouse. 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 SQL Data Warehouse, 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 SQL Data Warehouse](#)
- [HDInsight with Hive](#)
- [Azure Data Factory v2](#)
- [Oozie on HDInsight](#)

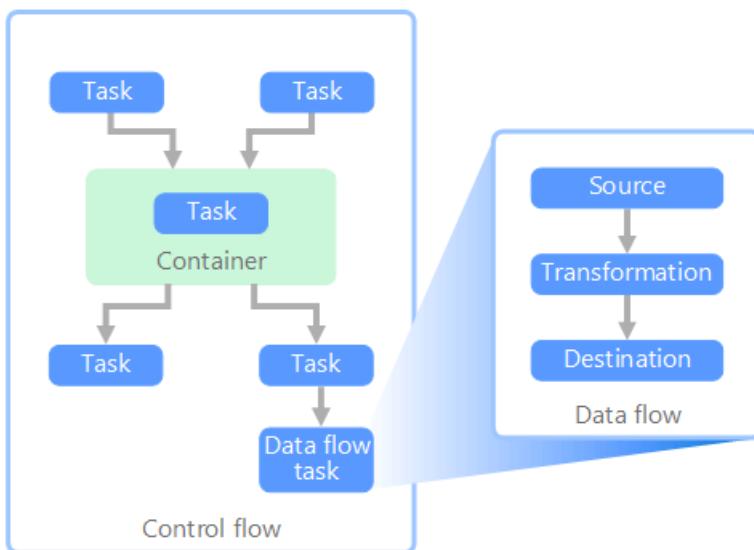
Other tools:

- [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](#)

# Big data architectures

4/11/2018 • 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 data sets 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:

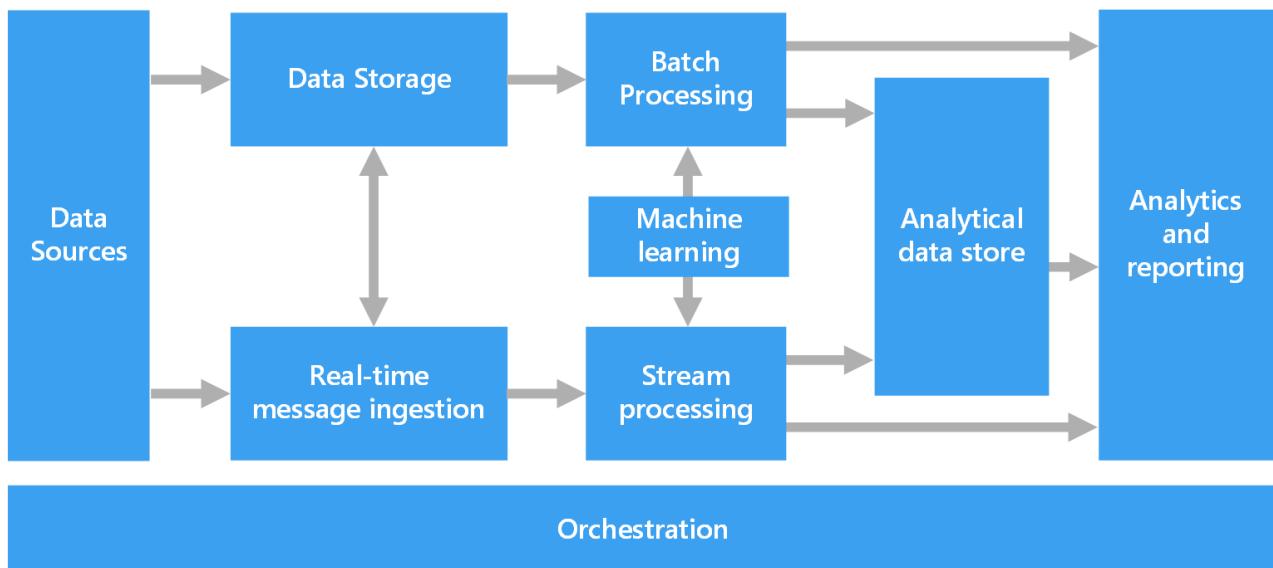
- Batch processing of big data sources at rest.
- Real-time processing of big data in motion.
- Interactive exploration of big data.
- Predictive analytics and machine learning.

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 SQL Data Warehouse 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

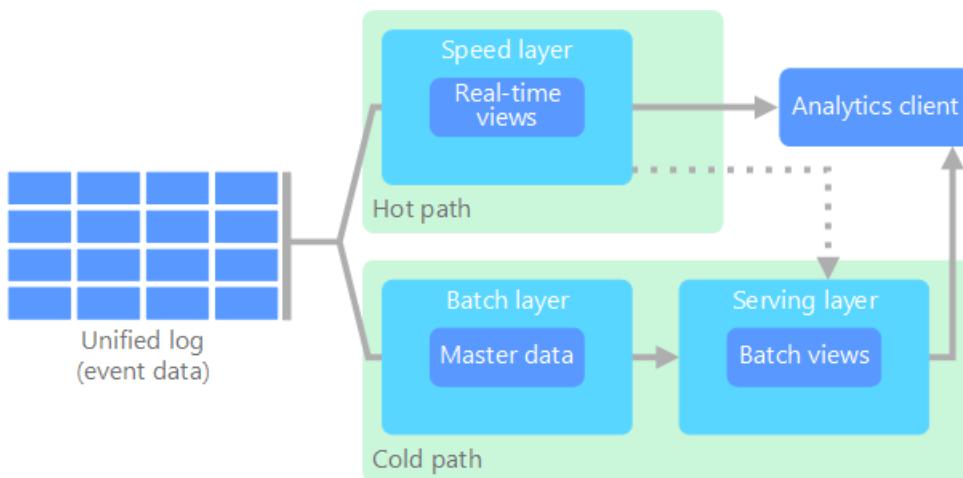
across the entire data set. The results are then stored separately from the raw data and used for querying.

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.

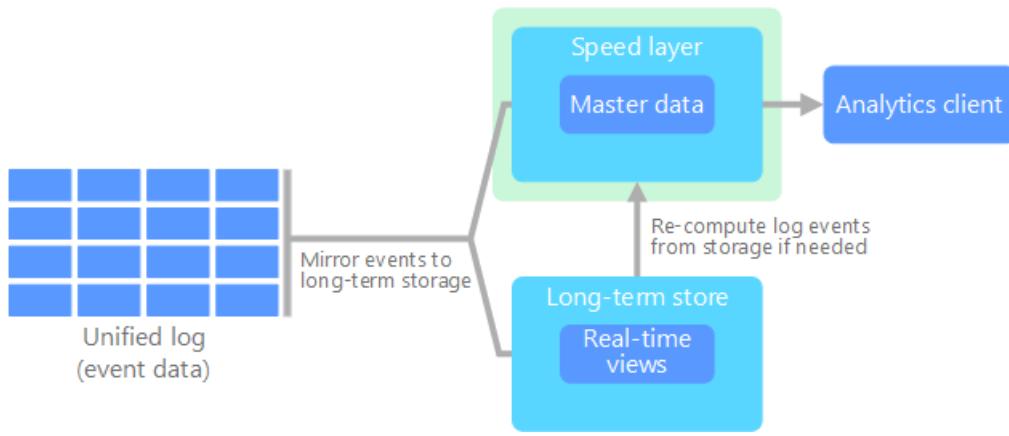
The raw data stored at the batch layer is immutable. Incoming data is always appended to the existing data, and the previous data is never overwritten. Any changes to the value of a particular datum are stored as a new timestamped event record. This allows for recomputation at any point in time across 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.



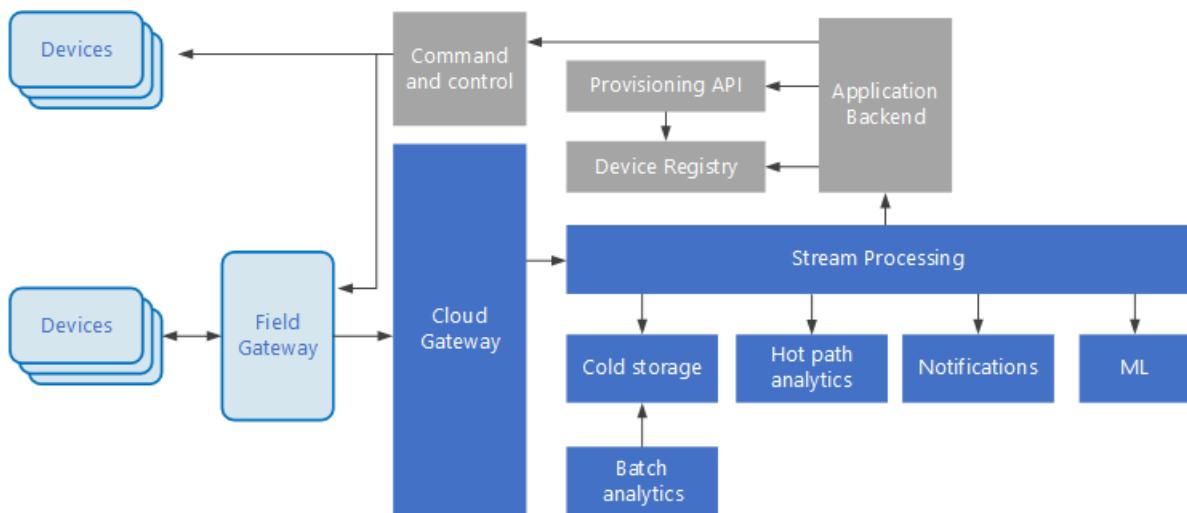
There are some similarities to the lambda architecture's batch layer, in that the event data is immutable and all of it is collected, instead of a subset. The data is ingested as a stream of events into a distributed and fault tolerant unified log. These events are ordered, and the current state of an 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](#)

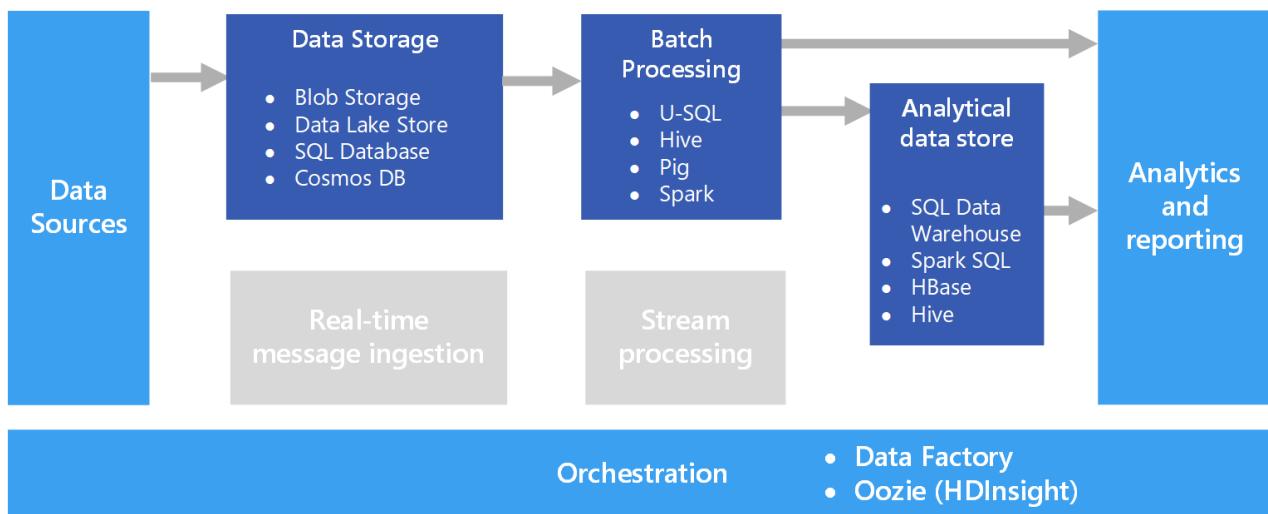
Learn more about IoT on Azure by reading the [Azure IoT reference architecture](#).

# Batch processing

4/11/2018 • 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 make use of 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

- **SQL Data Warehouse.** Azure SQL Data Warehouse 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 multidimensional and 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

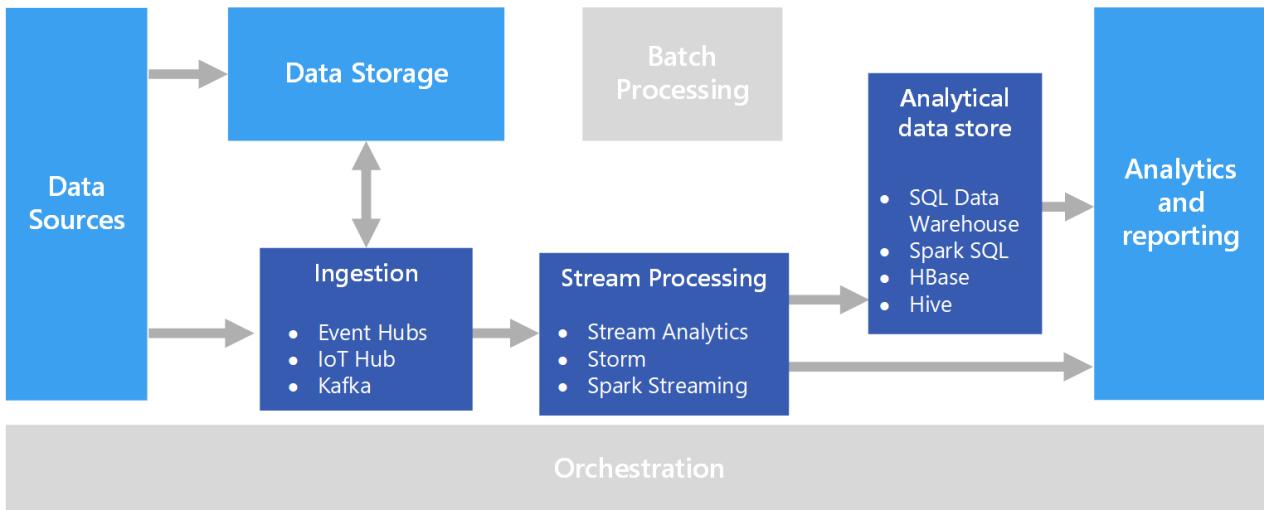
- **Azure Data Factory.** Azure Data Factory pipelines can be used to define a sequence of activities, scheduled for recurring temporal windows. These activities can initiate data copy operations as well as Hive, Pig, MapReduce, or Spark jobs in on-demand HDInsight clusters; U-SQL jobs in Azure Data Lake Analytics; and stored procedures in Azure SQL Data Warehouse 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

6/28/2018 • 4 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 message queuing solution for ingesting millions of event messages per second. The captured event data can be processed by multiple consumers in parallel.
- **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.

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

- **SQL Data Warehouse, HBase, Spark, or Hive.** Processed real-time data can be stored in a relational database such as Azure SQL Data Warehouse, 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.

# Machine learning at scale

5/11/2018 • 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 is operationalizing the learned model in a way that 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.

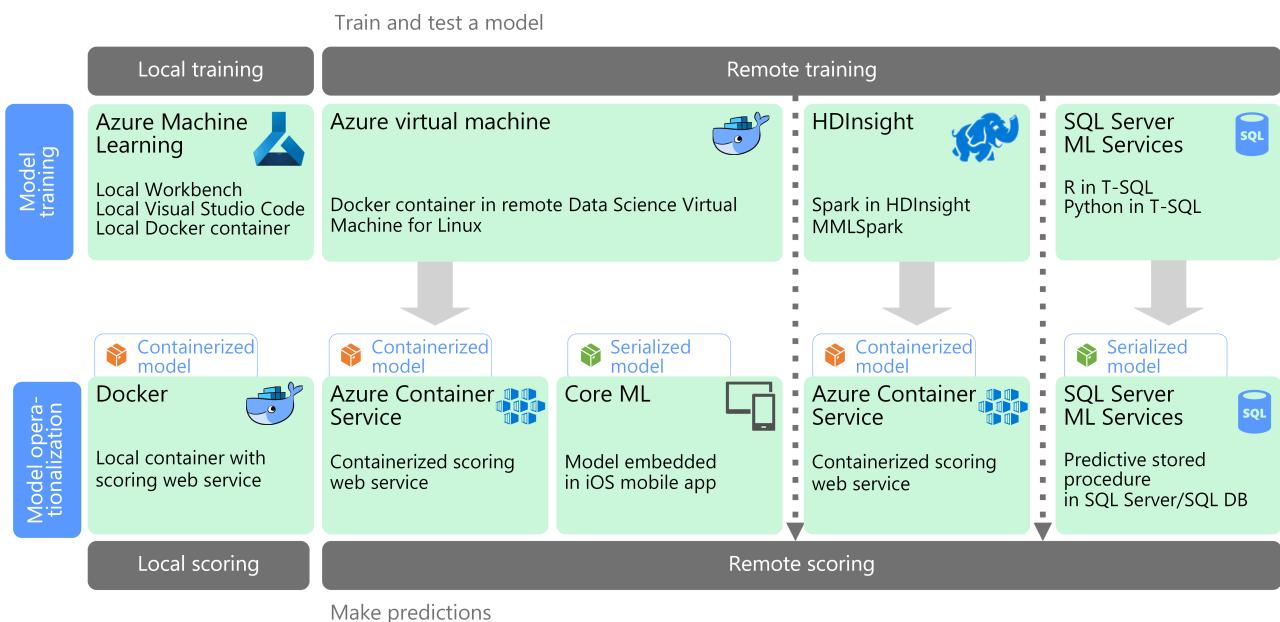
- The model training phase must access the big data stores. It's common to perform the model training using the same big data cluster, such as Spark, that is used for data preparation.
- For scenarios such as deep learning, not only will you need a cluster that can provide you scale out on CPUs, but 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 Microsoft Cognitive Services.
- Use the pretrained neural network models provided by Cognitive Toolkit.
- Embed the serialized models provided by Core ML for an iOS apps.

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, 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 the following topics:

- [Choosing a cognitive services technology](#)
- [Choosing a machine learning technology](#)
- [Choosing a natural language processing technology](#)

# Non-relational data and NoSQL

4/11/2018 • 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, while 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, and instead 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 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 a variety of 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. This 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 usually 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	CustomerID	Column Family: Contact Info
001	First name: Mu Bae Last name: Min	001	Phone number: 555-0100 Email: someone@example.com
002	First name: Francisco Last name: Vila Nova Suffix: Jr.	002	Email: vilanova@contoso.com
003	First name: Lena Last name: Adamczyz Title: Dr.	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 certain 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 usually atomic within a single column family, although some implementations provide atomicity across the entire row, spanning multiple column families.

Relevant Azure service:

- [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
AAAAA	1101001111010100110101111...
AABAB	1001100001011001101011110...
DFA766	0000000000101010110101010...
FABCC4	1110110110101010100101101...

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.

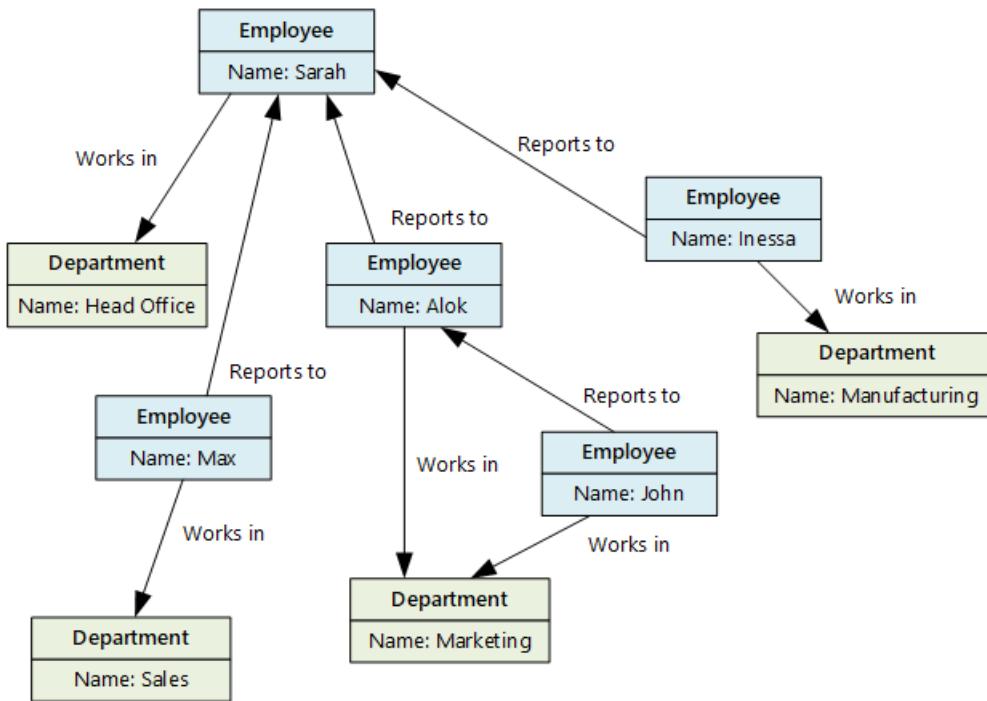
Relevant Azure services:

- [Azure Cosmos DB Table API](#)
- [Azure Redis Cache](#)
- [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 very complex analyses very 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 service:

- [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.csv	0XAABBCCDDEEF...	{created: 2017-06-02}
/delays/2017/06/02/flights.csv	0XAADDCCDDEEF...	{created: 2017-06-03}
/delays/2017/06/03/flights.csv	0XAEBBDEDDEEF...	{created: 2017-06-03}

Some object data stores replicate a given blob across multiple server nodes, which enables fast parallel reads. This 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 service:

- [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.

<b>id</b>	<b>search-document</b>
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 towards 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:

<b>REQUIREMENT</b>	<b>DOCUMENT DATA</b>	<b>COLUMN-FAMILY DATA</b>	<b>KEY/VALUE DATA</b>	<b>GRAPH DATA</b>
Normalization	Denormalized	Denormalized	Denormalized	Normalized
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

Requirement	Document Data	Column-Family Data	Key/Value Data	Graph Data
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
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)

# Advanced analytics

4/11/2018 • 6 minutes to read • [Edit Online](#)

Advanced analytics goes beyond the historical reporting and data aggregation of traditional business intelligence (BI), and uses mathematical, probabilistic, and statistical modeling techniques to enable predictive processing and automated decision making.

Advanced analytics solutions typically involve the following workloads:

- Interactive data exploration and visualization
- Machine Learning model training
- Real-time or batch predictive processing

Most advanced analytics architectures include some or all of the following components:

- **Data storage.** Advanced analytics solutions require data to train machine learning models. Data scientists typically need to explore the data to identify its predictive features and the statistical relationships between them and the values they predict (known as a label). The predicted label can be a quantitative value, like the financial value of something in the future or the duration of a flight delay in minutes. Or it might represent a categorical class, like "true" or "false," "flight delay" or "no flight delay," or categories like "low risk," "medium risk," or "high risk."
- **Batch processing.** To train a machine learning model, you typically need to process a large volume of training data. Training the model can take some time (on the order of minutes to hours). This training can be performed using scripts written in languages such as Python or R, and can be scaled out to reduce training time using distributed processing platforms like Apache Spark hosted in HDInsight or a Docker container.
- **Real-time message ingestion.** In production, many advanced analytics feed real-time data streams to a predictive model that has been published as a web service. The incoming data stream is typically captured in some form of queue and a stream processing engine pulls the data from this queue and applies the prediction to the input data in near real time.
- **Stream processing.** Once you have a trained model, prediction (or scoring) is typically a very fast operation (on the order of milliseconds) for a given set of features. After capturing real-time messages, the relevant feature values can be passed to the predictive service to generate a predicted label.
- **Analytical data store.** In some cases, the predicted label values are written to the analytical data store for reporting and future analysis.
- **Analysis and reporting.** As the name suggests, advanced analytics solutions usually produce some sort of report or analytical feed that includes predicted data values. Often, predicted label values are used to populate real-time dashboards.
- **Orchestration.** Although the initial data exploration and modeling is performed interactively by data scientists, many advanced analytics solutions periodically re-train models with new data — continually refining the accuracy of the models. This retraining can be automated using an orchestrated workflow.

## Machine learning

Machine learning is a mathematical modeling technique used to train a predictive model. The general principle is to apply a statistical algorithm to a large dataset of historical data to uncover relationships between the fields it contains.

Machine learning modeling is usually performed by data scientists, who need to thoroughly explore and prepare the data before training a model. This exploration and preparation typically involves a great deal of interactive data analysis and visualization — usually using languages such as Python and R in interactive tools and environments that are specifically designed for this task.

In some cases, you may be able to use [pretrained models](#) that come with training data obtained and developed by Microsoft. The advantage of pretrained models is that you can score and classify new content right away, even if you don't have the necessary training data, the resources to manage large datasets or to train complex models.

There are two broad categories of machine learning:

- **Supervised learning.** Supervised learning is the most common approach taken by machine learning. In a supervised learning model, the source data consists of a set of *feature* data fields that have a mathematical relationship with one or more *label* data fields. During the training phase of the machine learning process, the data set includes both features and known labels, and an algorithm is applied to fit a function that operates on the features to calculate the corresponding label predictions. Typically, a subset of the training dataset is held back and used to validate the performance of the trained model. Once the model has been trained, it can be deployed into production, and used to predict unknown values.
- **Unsupervised learning.** In an unsupervised learning model, the training data does not include known label values. Instead, the algorithm makes its predictions based on its first exposure to the data. The most common form of unsupervised learning is *clustering*, where the algorithm determines the best way to split the data into a specified number of clusters based on statistical similarities in the features. In clustering, the predicted outcome is the cluster number to which the input features belong. While they can sometimes be used directly to generate useful predictions, such as using clustering to identify groups of users in a database of customers, unsupervised learning approaches are more often used to identify which data is most useful to provide to a supervised learning algorithm in training a model.

Relevant Azure services:

- [Azure Machine Learning](#)
- [Machine Learning Server \(R Server\) on HDInsight](#)

## Deep learning

Machine learning models based on mathematical techniques like linear or logistic regression have been available for some time. More recently, the use of *deep learning* techniques based on neural networks has increased. This is driven partly by the availability of highly scalable processing systems that reduce how long it takes to train complex models. Also, the increased prevalence of big data makes it easier to train deep learning models in a variety of domains.

When designing a cloud architecture for advanced analytics, you should consider the need for large-scale processing of deep learning models. These can be provided through distributed processing platforms like Apache Spark and the latest generation of virtual machines that include access to GPU hardware.

Relevant Azure services:

- [Deep Learning Virtual Machine](#)
- [Apache Spark on HDInsight](#)

## Artificial intelligence

Artificial intelligence (AI) refers to scenarios where a machine mimics the cognitive functions associated with human minds, such as learning and problem solving. Because AI leverages machine learning algorithms, it is viewed as an umbrella term. Most AI solutions rely on a combination of predictive services, often implemented as web services, and natural language interfaces, such as chatbots that interact via text or speech, that are presented

by AI apps running on mobile devices or other clients. In some cases, the machine learning model is embedded with the AI app.

## Model deployment

The predictive services that support AI applications may leverage custom machine learning models, or off-the-shelf cognitive services that provide access to pretrained models. The process of deploying custom models into production is known as operationalization, where the same AI models that are trained and tested within the processing environment are serialized and made available to external applications and services for batch or self-service predictions. To use the predictive capability of the model, it is deserialized and loaded using the same machine learning library that contains the algorithm that was used to train the model in the first place. This library provides predictive functions (often called score or predict) that take the model and features as input and return the prediction. This logic is then wrapped in a function that an application can call directly or can be exposed as a web service.

Relevant Azure services:

- [Azure Machine Learning](#)
- [Machine Learning Server \(R Server\) on HDInsight](#)

## See also

- [Choosing a cognitive services technology](#)
- [Choosing a machine learning technology](#)

# Data lakes

4/11/2018 • 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 hyper-scale, Hadoop-compatible repository.

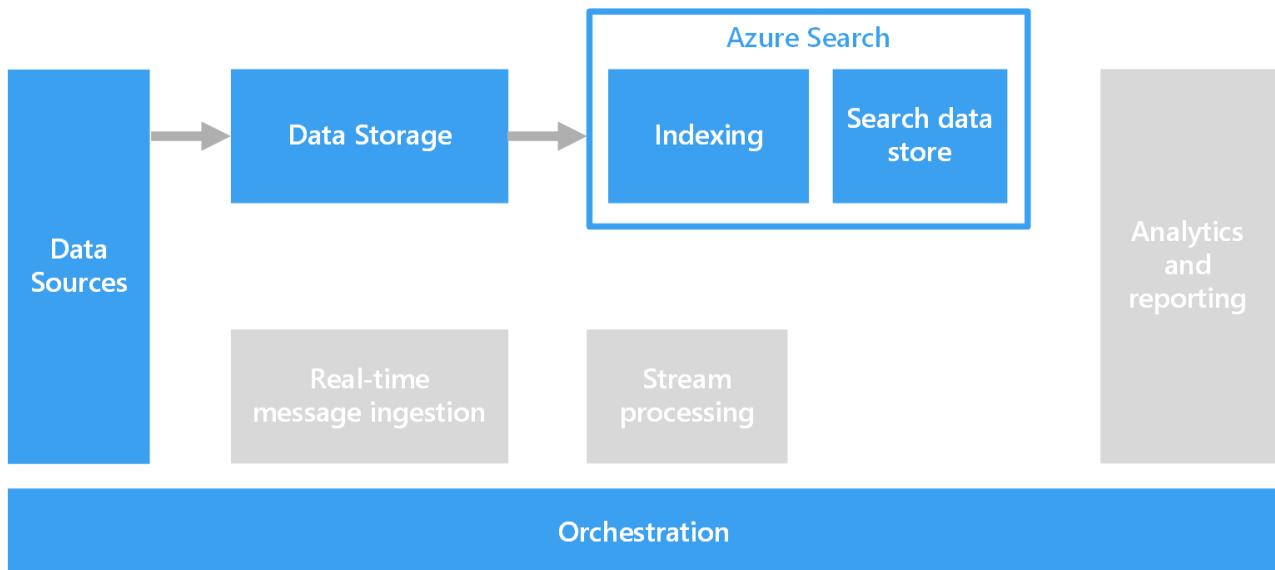
- [Data Lake Analytics](#) is an on-demand analytics job service to simplify big data analytics.

# Processing free-form text for search

2/13/2018 • 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. An exception is using full text search within SQL Server or Azure SQL Database. In this case, the document data is loaded into 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 Search, Elasticsearch, and HDInsight with Solr. Each of these technologies can populate a search index from a collection of documents. Azure Search provides indexers that can automatically populate the index for documents ranging from plain text to Excel and PDF formats. 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.

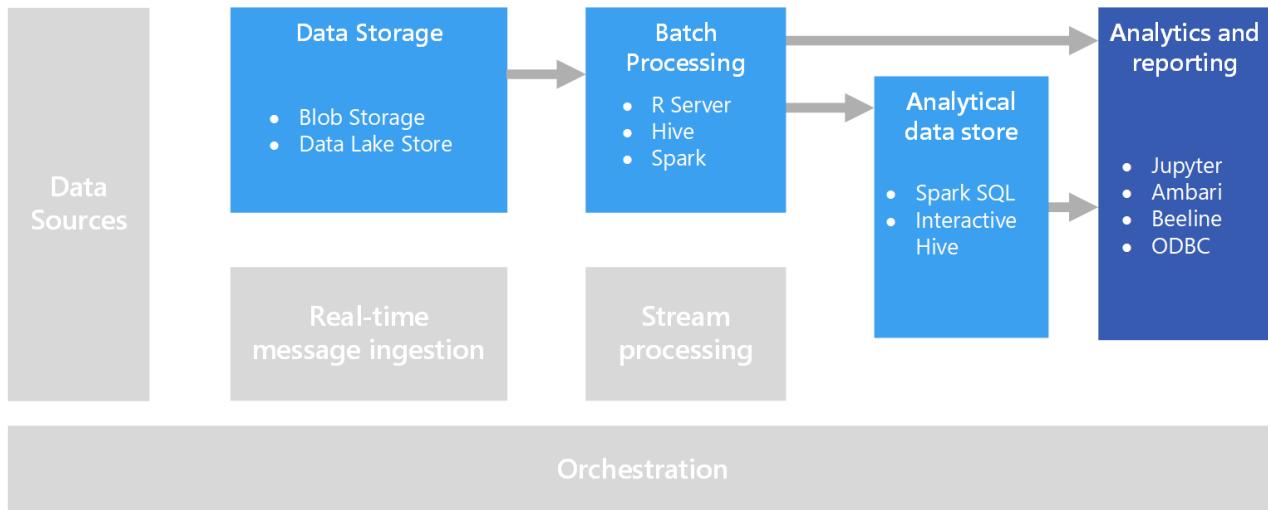
If your text data is stored in SQL Server or Azure SQL Database, you can use the full-text search that is built into the database. The database populates the index from text, binary, or XML data stored within the same database. Clients search by using T-SQL queries.

For more information, see [Search data stores](#).

# Interactive data exploration

8/30/2018 • 4 minutes to read • [Edit Online](#)

In many corporate business intelligence (BI) solutions, reports and semantic models are created by BI specialists and managed centrally. Increasingly, however, organizations want to enable users to make data-driven decisions. Additionally, a growing number of organizations are hiring *data scientists* or *data analysts*, whose job is to explore data interactively and apply statistical models and analytical techniques to find trends and patterns in the data. Interactive data exploration requires tools and platforms that provide low-latency processing for ad-hoc queries and data visualizations.



## Self-service BI

Self-service BI is a name given to a modern approach to business decision making in which users are empowered to find, explore, and share insights from data across the enterprise. To accomplish this, the data solution must support several requirements:

- Discovery of business data sources through a data catalog.
- Master data management to ensure consistency of data entity definitions and values.
- Interactive data modeling and visualization tools for business users.

In a self-service BI solution, business users typically find and consume data sources that are relevant to their particular area of the business, and use intuitive tools and productivity applications to define personal data models and reports that they can share with their colleagues.

Relevant Azure services:

- [Azure Data Catalog](#)
- [Microsoft Power BI](#)

## Data science experimentation

When an organization requires advanced analytics and predictive modeling, the initial preparation work is usually undertaken by specialist data scientists. A data scientist explores the data and applies statistical analytical techniques to find relationships between data *features* and the desired predicted *labels*. Data exploration is typically done using programming languages such as Python or R that natively support statistical modeling and visualization. The scripts used to explore the data are typically hosted in specialized environments such as Jupyter Notebooks. These tools enable data scientists to explore the data programmatically while documenting and

sharing the insights they find.

Relevant Azure services:

- [Azure Notebooks](#)
- [Azure Machine Learning Studio](#)
- [Azure Machine Learning Experimentation Services](#)
- [The Data Science Virtual Machine](#)

## Challenges

- **Data privacy compliance.** You need to be careful about making personal data available to users for self-service analysis and reporting. There are likely to be compliance considerations, due to organizational policies and also regulatory issues.
- **Data volume.** While it may be useful to give users access to the full data source, it can result in very long-running Excel or Power BI operations, or Spark SQL queries that use a lot of cluster resources.
- **User knowledge.** Users create their own queries and aggregations in order to inform business decisions. Are you confident that users have the necessary analytical and querying skills to get accurate results?
- **Sharing results.** There may be security considerations if users can create and share reports or data visualizations.

## Architecture

Although the goal of this scenario is to support interactive data analysis, the data cleansing, sampling, and structuring tasks involved in data science often include long-running processes. That makes a [batch processing](#) architecture appropriate.

## Technology choices

The following technologies are recommended choices for interactive data exploration in Azure.

### Data storage

- **Azure Storage Blob Containers or Azure Data Lake Store.** Data scientists generally work with raw source data, to ensure they have access to all possible features, outliers, and errors in the data. In a big data scenario, this data usually takes the form of files in a data store.

For more information, see [Data storage](#).

### Batch processing

- **R Server or Spark.** Most data scientists use programming languages with strong support for mathematical and statistical packages, such as R or Python. When working with large volumes of data, you can reduce latency by using platforms that enable these languages to use distributed processing. R Server can be used on its own or in conjunction with Spark to scale out R processing functions, and Spark natively supports Python for similar scale-out capabilities in that language.
- **Hive.** Hive is a good choice for transforming data using SQL-like semantics. Users can create and load tables using HiveQL statements, which are semantically similar to SQL.

For more information, see [Batch processing](#).

### Analytical Data Store

- **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. Regardless of whether the data files to be analyzed are raw source files or new files that have been cleaned and prepared by a batch process, users can define Spark SQL tables on them for further

querying an analysis.

- **Hive**. In addition to batch processing raw data by using Hive, you can create a Hive database that contains Hive tables and views based on the folders where the data is stored, enabling interactive queries for analysis and reporting. HDInsight includes an Interactive Hive cluster type that uses in-memory caching to reduce Hive query response times. Users who are comfortable with SQL-like syntax can use Interactive Hive to explore data.

For more information, see [Analytical data stores](#).

## Analytics and reporting

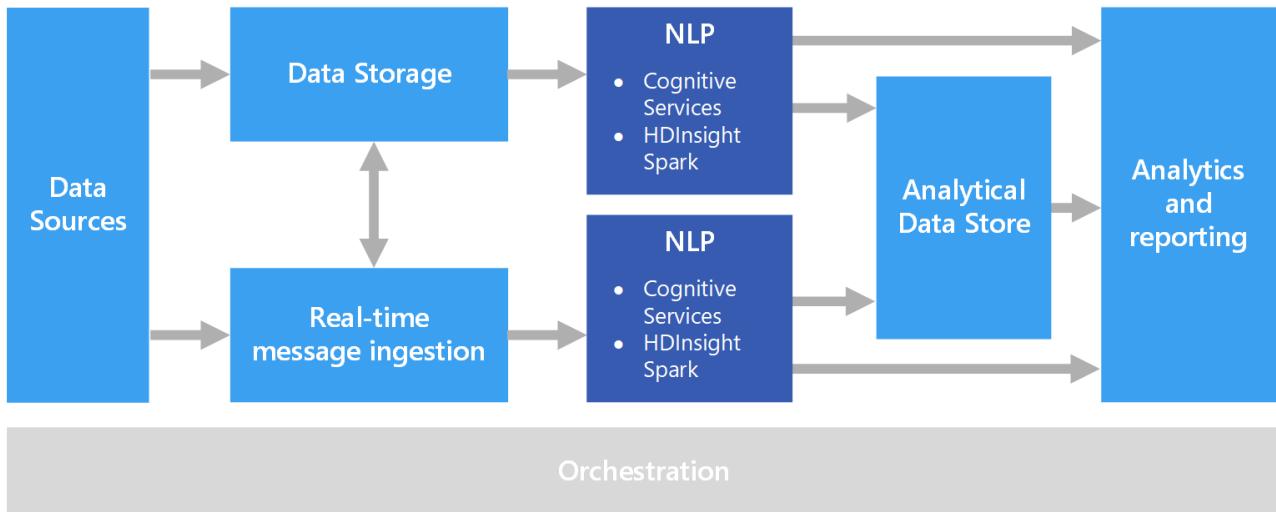
- **Jupyter**. Jupyter Notebooks provides a browser-based interface for running code in languages such as R, Python, or Scala. When using R Server or Spark to batch process data, or when using Spark SQL to define a schema of tables for querying, Jupyter can be a good choice for querying the data. When using Spark, you can use the standard Spark dataframe API or the Spark SQL API as well as embedded SQL statements to query the data and produce visualizations.
- **Drill**. If you want to perform ad hoc data exploration, [Apache Drill](#) is a schema-free SQL query engine. Because it doesn't require a schema, you can query data from a variety of data sources, and the engine will automatically understand the structure of the data.
- **Interactive Hive Clients**. If you use an Interactive Hive cluster to query the data, you can use the Hive view in the Ambari cluster dashboard, the Beeline command line tool, or any ODBC-based tool (using the Hive ODBC driver), such as Microsoft Excel or Power BI.

For more information, see [Data analytics and reporting technology](#).

# Natural language processing

4/11/2018 • 2 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.



## When to use this solution

NLP can be used 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 very 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.

# Architecture

In an NLP solution, free-form text processing is performed against documents containing paragraphs of text. The overall architecture can be a [batch processing](#) or [real-time stream processing](#) architecture.

The actual processing varies based on the desired outcome, but in terms of the pipeline, NLP may be applied in a batch or real-time fashion. For example, sentiment analysis can be used against blocks of text to produce a sentiment score. This could be done by running a batch process against data in storage, or in real time using smaller chunks of data flowing through a messaging service.

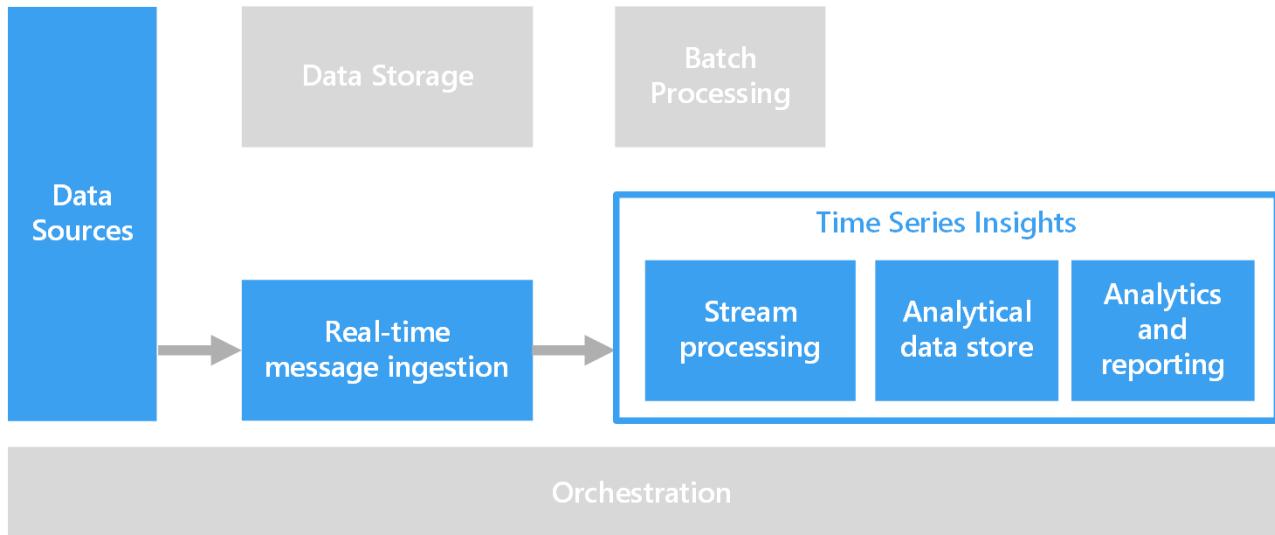
## Technology choices

- [Natural language processing](#)

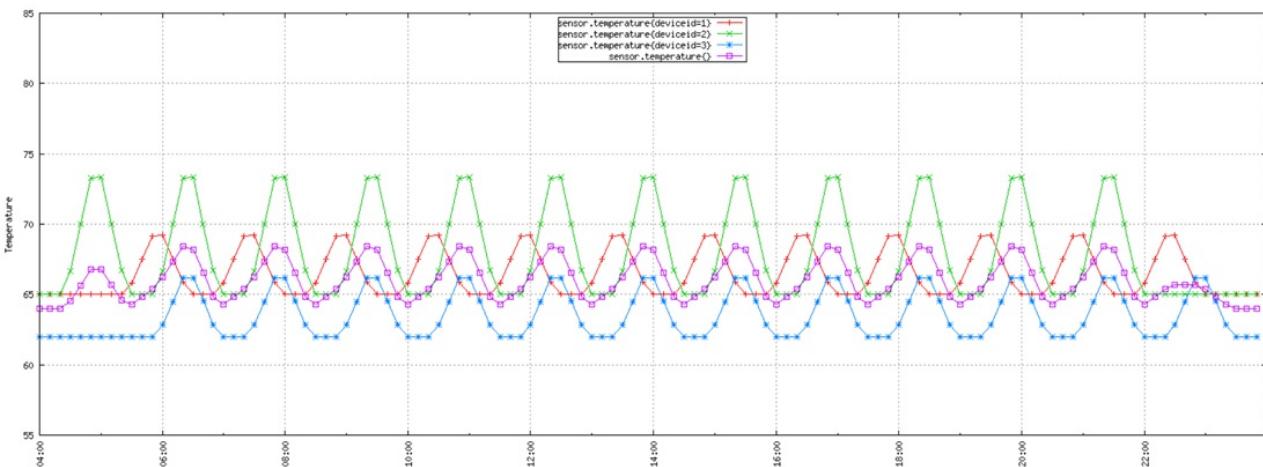
# Time series solutions

4/11/2018 • 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 time stamped 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 [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

4/11/2018 • 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](#).

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.

# Choosing an analytical data store in Azure

9/28/2018 • 4 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:

- [SQL Data Warehouse](#)
- [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 (date 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."

## 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.

## Capability matrix

The following tables summarize the key differences in capabilities.

### General capabilities

	SQL DATABASE	SQL DATA WAREHOUSE	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Is managed service	Yes	Yes	Yes <sup>1</sup>	Yes <sup>1</sup>	Yes	Yes
Primary database model	Relational (columnar format when using columnstore indexes)	Relational tables with columnar storage	Wide column store	Hive/In-Memory	Tabular/MOLAP semantic models	Document store, graph, key-value store, wide column store
SQL language support	Yes	Yes	Yes (using <a href="#">Phoenix</a> JDBC driver)	Yes	No	Yes
Optimized for speed serving layer	Yes <sup>2</sup>	No	Yes	Yes	No	Yes

[1] With manual configuration and scaling.

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

### Scalability capabilities

	SQL DATABASE	SQL DATA WAREHOUSE	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Redundant regional servers for high availability	Yes	Yes	Yes	No	No	Yes

	<b>SQL DATABASE</b>	<b>SQL DATA WAREHOUSE</b>	<b>HBASE/PHOENIX ON HDINSIGHT</b>	<b>HIVE LLAP ON HDINSIGHT</b>	<b>AZURE ANALYSIS SERVICES</b>	<b>COSMOS DB</b>
Supports query scale out	No	Yes	Yes	Yes	Yes	Yes
Dynamic scalability (scale up)	Yes	Yes	No	No	Yes	Yes
Supports in-memory caching of data	Yes	Yes	No	Yes	Yes	No

## Security capabilities

	<b>SQL DATABASE</b>	<b>SQL DATA WAREHOUSE</b>	<b>HBASE/PHOENIX ON HDINSIGHT</b>	<b>HIVE LLAP ON HDINSIGHT</b>	<b>AZURE ANALYSIS SERVICES</b>	<b>COSMOS DB</b>
Authentication	SQL / Azure Active Directory (Azure AD)	SQL / Azure AD	local / Azure AD <sup>1</sup>	local / Azure AD <sup>1</sup>	Azure AD	database users / Azure AD via access control (IAM)
Data encryption at rest	Yes <sup>2</sup>	Yes <sup>2</sup>	Yes <sup>1</sup>	Yes <sup>1</sup>	Yes	Yes
Row-level security	Yes	No	Yes <sup>1</sup>	Yes <sup>1</sup>	Yes (through object-level security in model)	No
Supports firewalls	Yes	Yes	Yes <sup>3</sup>	Yes <sup>3</sup>	Yes	Yes
Dynamic data masking	Yes	No	Yes <sup>1</sup>	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] When used within an Azure Virtual Network. See [Extend Azure HDInsight using an Azure Virtual Network](#).

# Choosing a data analytics technology in Azure

9/28/2018 • 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 as 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

	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 <sup>1</sup>	Yes <sup>1</sup>	Yes
Connect to 100s of data sources	Yes	No	No	No
Offline capabilities	Yes <sup>2</sup>	No	No	No
Embedding capabilities	Yes	No	No	No

	POWER BI	JUPYTER NOTEBOOKS	ZEPELIN NOTEBOOKS	MICROSOFT AZURE NOTEBOOKS
Automatic data refresh	Yes	No	No	No
Access to numerous open source packages	No	Yes <sup>3</sup>	Yes <sup>3</sup>	Yes <sup>4</sup>
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 <a href="#">pricing</a> for hosting options	Free	Free	Free
Multiuser collaboration	Yes	Yes (through sharing or with a multiuser server like <a href="#">JupyterHub</a> )	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](#).

# Choosing a batch processing technology in Azure

4/11/2018 • 2 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.

## What are your options when choosing a batch processing technology?

In Azure, all of the following data stores will meet the core requirements for batch processing:

- [Azure Data Lake Analytics](#)
- [Azure SQL Data Warehouse](#)
- [HDInsight with Spark](#)
- [HDInsight with Hive](#)
- [HDInsight with Hive LLAP](#)

## 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 pause 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

	AZURE DATA LAKE ANALYTICS	AZURE SQL DATA WAREHOUSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP
Is managed service	Yes	Yes	Yes <sup>1</sup>	Yes <sup>1</sup>	Yes <sup>1</sup>
Supports pausing compute	No	Yes	No	No	No
Relational data store	Yes	Yes	No	No	No

	AZURE DATA LAKE ANALYTICS	AZURE SQL DATA WAREHOUSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP
Programmability	U-SQL	T-SQL	Python, Scala, Java, R	HiveQL	HiveQL
Programming paradigm	Mixture of declarative and imperative	Declarative	Mixture of declarative and imperative	Declarative	Declarative
Pricing model	Per batch job	By cluster hour	By cluster hour	By cluster hour	By cluster hour

[1] With manual configuration and scaling.

### Integration capabilities

	AZURE DATA LAKE ANALYTICS	SQL DATA WAREHOUSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP
Access from Azure Data Lake Store	Yes	Yes	Yes	Yes	Yes
Query from Azure Storage	Yes	Yes	Yes	Yes	Yes
Query from external relational stores	Yes	No	Yes	No	No

### Scalability capabilities

	AZURE DATA LAKE ANALYTICS	SQL DATA WAREHOUSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP
Scale-out granularity	Per job	Per cluster	Per cluster	Per cluster	Per cluster
Fast scale out (less than 1 minute)	Yes	Yes	No	No	No
In-memory caching of data	No	Yes	Yes	No	Yes

### Security capabilities

	AZURE DATA LAKE ANALYTICS	SQL DATA WAREHOUSE	HDINSIGHT WITH SPARK	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Authentication	Azure Active Directory (Azure AD)	SQL / Azure AD	No	local / Azure AD <sup>1</sup>	local / Azure AD <sup>1</sup>
Authorization	Yes	Yes	No	Yes <sup>1</sup>	Yes <sup>1</sup>
Auditing	Yes	Yes	No	Yes <sup>1</sup>	Yes <sup>1</sup>

	AZURE DATA LAKE ANALYTICS	SQL DATA WAREHOUSE	HDINSIGHT WITH SPARK	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Data encryption at rest	Yes	Yes <sup>2</sup>	Yes	Yes	Yes
Row-level security	No	Yes	No	Yes <sup>1</sup>	Yes <sup>1</sup>
Supports firewalls	Yes	Yes	Yes	Yes <sup>3</sup>	Yes <sup>3</sup>
Dynamic data masking	No	No	No	Yes <sup>1</sup>	Yes <sup>1</sup>

[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](#).

# Choosing a Microsoft cognitive services technology

4/11/2018 • 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](#)
- [Knowledge](#)
- [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

	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.
Bing Video Search API	Text (web search query)	Search for videos.

	INPUT TYPE	KEY BENEFIT
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

	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.

# Choosing a big data storage technology in Azure

9/28/2018 • 7 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](#)

## 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 SQL Data Warehouse 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](#).
- [Role-Based Access Control \(RBAC\)](#) to control access using Azure Active Directory users and groups.

## Azure Data Lake Store

[Azure Data Lake Store](#) is an enterprise-wide hyper-scale 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 SQL Data Warehouse 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.

## 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 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 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

	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 <a href="#">Azure Active Directory Identities</a>	Based on shared secrets <a href="#">Account Access Keys</a> and <a href="#">Shared Access Signature Keys</a> , and <a href="#">Role-Based Access Control (RBAC)</a>
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.
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 <a href="#">Account Access Keys</a> . For account, container, or blob authorization use <a href="#">Shared Access Signature Keys</a> .
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

	AZURE DATA LAKE STORE	AZURE BLOB STORAGE CONTAINERS
Size limits	No limits on account sizes, file sizes or number of files	Specific limits documented <a href="#">here</a>
Geo-redundancy	Locally-redundant (multiple copies of data in one Azure region)	Locally redundant (LRS), globally redundant (GRS), read-access globally redundant (RA-GRS). See <a href="#">here</a> for more information

## NoSQL database capabilities

	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 <a href="#">Phoenix JDBC driver</a> )
Consistency	Strong, bounded-staleness, session, consistent prefix, eventual	Strong
Native Azure Functions integration	Yes	No
Automatic global distribution	Yes	No <a href="#">HBase cluster replication can be configured</a> 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

# Choosing a machine learning technology in Azure

2/13/2018 • 7 minutes to read • [Edit Online](#)

Data science and machine learning is a workload that is usually undertaken by data scientists. It requires specialist tools, many of which are designed specifically for the type of interactive data exploration and modeling tasks that a data scientist must perform.

Machine learning solutions are built iteratively, and have two distinct phases:

- Data preparation and modeling.
- Deployment and consumption of predictive services.

## Tools and services for data preparation and modeling

Data scientists typically prefer to work with data using custom code written in Python or R. This code is generally run interactively, with the data scientists using it to query and explore the data, generating visualizations and statistics to help determine the relationships with it. There are many interactive environments for R and Python that data scientists can use. A particular favorite is **Jupyter Notebooks** that provides a browser-based shell that enables data scientists to create *notebook* files that contain R or Python code and markdown text. This is an effective way to collaborate by sharing and documenting code and results in a single document.

Other commonly used tools include:

- **Spyder**: The interactive development environment (IDE) for Python provided with the Anaconda Python distribution.
- **R Studio**: An IDE for the R programming language.
- **Visual Studio Code**: A lightweight, cross-platform coding environment that supports Python as well as commonly used frameworks for machine learning and AI development.

In addition to these tools, data scientists can leverage Azure services to simplify code and model management.

### Azure Notebooks

Azure Notebooks is an online Jupyter Notebooks service that enables data scientists to create, run, and share Jupyter Notebooks in cloud-based libraries.

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.

### Data science virtual machine

The data science virtual machine is an Azure virtual machine image that includes the tools and frameworks commonly used by data scientists, including R, Python, Jupyter Notebooks, Visual Studio Code, and libraries for machine learning modeling such as the Microsoft Cognitive Toolkit. These tools can be complex and time

consuming to install, and contain many interdependencies that often lead to version management issues. Having a preinstalled image can reduce the time data scientists spend troubleshooting environment issues, allowing them to focus on the data exploration and modeling tasks they need to perform.

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 Machine Learning

Azure Machine Learning is a cloud-based service for managing machine learning experiments and models. It includes an experimentation service that tracks data preparation and modeling training scripts, maintaining a history of all executions so you can compare model performance across iterations. A cross-platform client tool named Azure Machine Learning Workbench provides a central interface for script management and history, while still enabling data scientists to create scripts in their tool of choice, such as Jupyter Notebooks or Visual Studio Code.

From Azure Machine Learning Workbench, you can use the interactive data preparation tools to simplify common data transformation tasks, and you can configure the script execution environment to run model training scripts locally, in a scalable Docker container, or in Spark.

When you are ready to deploy your model, use the Workbench environment to package the model and deploy it as a web service to a Docker container, Spark on Azure HDInsight, Microsoft Machine Learning Server, or SQL Server. The Azure Machine Learning Model Management service then enables you to track and manage model deployments in the cloud, on edge devices, or across the enterprise.

Key benefits:

- Central management of scripts and run history, making it easy to compare model versions.
- Interactive data transformation through a visual editor.
- Easy deployment and management of models to the cloud or edge devices.

Considerations:

- Requires some familiarity with the model management model and Workbench tool environment.

## Azure Batch AI

Azure Batch AI enables you to run your machine learning experiments in parallel, and perform model training at scale across a cluster of virtual machines with GPUs. Batch AI training enables you to scale out deep learning jobs across clustered GPUs, using frameworks such as Cognitive Toolkit, Caffe, Chainer, and TensorFlow.

Azure Machine Learning Model Management can be used to take models from Batch AI training to deploy, manage, and monitor them.

## Azure Machine Learning Studio

Azure Machine Learning Studio is a cloud-based, visual development environment for creating data experiments, training machine learning models, and publishing them as web services in Azure. Its visual drag-and-drop interface lets data scientists and power users create machine learning solutions quickly, while supporting custom R and Python logic, a wide range of established statistical algorithms and techniques for machine learning modeling tasks, and built-in support for Jupyter Notebooks.

Key benefits:

- Interactive visual interface enables machine learning modeling with minimal code.
- Built-in Jupyter Notebooks for data exploration.
- Direct deployment of trained models as Azure web services.

Considerations:

- Limited scalability. The maximum size of a training dataset is 10 GB.
- Online only. No offline development environment.

## Tools and services for deploying machine learning models

After a data scientist has created a machine learning model, you will typically need to deploy it and consume it from applications or in other data flows. There are a number of potential deployment targets for machine learning models.

### **Spark on Azure HDInsight**

Apache Spark includes Spark MLlib, a framework and library for machine learning models. The Microsoft Machine Learning library for Spark (MMLSpark) also provides deep learning algorithm support for predictive models in Spark.

Key benefits:

- Spark is a distributed platform that offers high scalability for high-volume machine learning processes.
- You can deploy models directly to Spark in HDinsight from Azure Machine Learning Workbench, and manage them using the Azure Machine Learning Model Management service.

Considerations:

- Spark runs in an HDinsight cluster that incurs charges the whole time it is running. If the machine learning service will only be used occasionally, this may result in unnecessary costs.

### **Web service in a container**

You can deploy a machine learning model as a Python web service in a Docker container. You can deploy the model to Azure or to an edge device, where it can be used locally with the data on which it operates.

Key Benefits:

- Containers are a lightweight and generally cost effective way to package and deploy services.
- The ability to deploy to an edge device enables you to move your predictive logic closer to the data.
- You can deploy to a container directly from Azure Machine Learning Workbench.

Considerations:

- This deployment model is based on Docker containers, so you should be familiar with this technology before deploying a web service this way.

### **Microsoft Machine Learning Server**

Machine Learning Server (formerly Microsoft R Server) is a scalable platform for R and Python code, specifically designed for machine learning scenarios.

Key benefits:

- High scalability.
- Direct deployment from Azure Machine Learning Workbench.

Considerations:

- You need to deploy and manage Machine Learning Server in your enterprise.

## **Microsoft SQL Server**

Microsoft SQL Server supports R and Python natively, enabling you to encapsulate machine learning models built in these languages as Transact-SQL functions in a database.

Key benefits:

- Encapsulate predictive logic in a database function, making it easy to include in data-tier logic.

Considerations:

- Assumes a SQL Server database as the data tier for your application.

## **Azure Machine Learning web service**

When you create a machine learning model using Azure Machine Learning Studio, you can deploy it as a web service. This can then be consumed through a REST interface from any client applications capable of communicating by HTTP.

Key benefits:

- Ease of development and deployment.
- Web service management portal with basic monitoring metrics.
- Built-in support for calling Azure Machine Learning web services from Azure Data Lake Analytics, Azure Data Factory, and Azure Stream Analytics.

Considerations:

- Only available for models built using Azure Machine Learning Studio.
- Web-based access only, trained models cannot run on-premises or offline.

# Choosing a natural language processing technology in Azure

2/13/2018 • 2 minutes to read • [Edit Online](#)

Free-form text processing is performed against documents containing paragraphs of text, typically for the purpose of supporting search, but is also used to perform other natural language processing (NLP) tasks such as sentiment analysis, topic detection, language detection, key phrase extraction, and document categorization. This article focuses on the technology choices that act in support of the NLP tasks.

## 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](#)
- [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

	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

	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)
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

	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Entity/intent identification & 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](#)

# Choosing a data pipeline orchestration technology in Azure

2/13/2018 • 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

	AZURE DATA FACTORY	SQL SERVER INTEGRATION SERVICES (SSIS)	OOZIE ON HDINSIGHT
Managed	Yes	No	Yes
Cloud-based	Yes	No (local)	Yes

	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

	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

	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

# Choosing a real-time message ingestion technology in Azure

5/21/2018 • 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.

Feature 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.

	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 <sup>1</sup>	AMQP, HTTPS	<a href="#">Kafka Protocol</a>
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](#).

# Choosing a search data store in Azure

10/8/2018 • 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 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

	AZURE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Is managed service	Yes	No	Yes	Yes
REST API	Yes	Yes	Yes	No

	AZURE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Programmability	.NET	Java	Java	T-SQL
Document indexers for common file types (PDF, DOCX, TXT, and so on)	Yes	No	Yes	No

### Manageability capabilities

	AZURE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Updateable schema	No	Yes	Yes	Yes
Supports scale out	Yes	Yes	Yes	No

### Analytic workload capabilities

	AZURE 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

	AZURE 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	No	Yes	Yes	Yes
Restrict access to allow virtual network access only	No	Yes	Yes	Yes
Active Directory authentication (integrated authentication)	No	No	No	Yes

### See also

Processing free-form text for search

# Choosing a stream processing technology in Azure

10/24/2018 • 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, process the messages, and forward 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 an 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](#)

## 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 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

	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	Scala, Python, Java	Scala, Python, Java, R	Java, C#	C#, F#, Node.js	C#, Node.js, PHP, Java, Python

	AZURE STREAM ANALYTICS	HDINSIGHT WITH SPARK STREAMING	APACHE SPARK IN AZURE DATABRICKS	HDINSIGHT WITH STORM	AZURE FUNCTIONS	AZURE APP SERVICE WEBJOBS
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

	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

	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

	AZURE STREAM ANALYTICS	HDINSIGHT WITH SPARK STREAMING	APACHE SPARK IN AZURE DATABRICKS	HDINSIGHT WITH STORM	AZURE FUNCTIONS	AZURE APP SERVICE WEBJOBS
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](#)

# Transferring data to and from Azure

6/11/2018 • 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 SSDs 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 leverage 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. It is also faster than most other options. 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 SQL Data Warehouse, you can import/export data from Azure Blob Storage and Azure Data Lake Store. Currently, PolyBase is the fastest method of importing data into SQL Data Warehouse.
- **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

	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

	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

	AZURE CLI	AZCOPY	POWERSHELL	ADLCOPY	POLYBASE
Compatible platforms	Linux, OS X, Windows	Linux, Windows	Windows	Linux, OS X, Windows	SQL Server, Azure SQL Data Warehouse
Optimized for big data	No	No	No	Yes <sup>1</sup>	Yes <sup>2</sup>
Copy to relational database	No	No	No	No	Yes
Copy from relational database	No	No	No	No	Yes

	AZURE CLI	AZCOPY	POWERSHELL	ADLCOPY	POLYBASE
Copy to Blob storage	Yes	Yes	Yes	No	Yes
Copy from Blob storage	Yes	Yes	Yes	Yes	Yes
Copy to Data Lake Store	No	No	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

	AZURE STORAGE EXPLORER	AZURE PORTAL *	AZURE DATA FACTORY
Optimized for big data	No	No	Yes
Copy to relational database	No	No	Yes
Copy to 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.

# Extending on-premises data solutions to the cloud

2/13/2018 • 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 data centers, 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 data center 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 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 a 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 off-site 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 the following topics:

- [Choose a solution for connecting an on-premises network to Azure](#)
- [Hybrid network reference architectures](#)

# Securing data solutions

2/13/2018 • 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 (MFA) for users.
- Use role based access control (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 SQL Data Warehouse.** 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 [Role-Based Access Control](#) (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, Office 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](#) (NSGs) 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

[Azure Security Center](#) 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.

# Cloud Design Patterns

8/30/2018 • 6 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

### Availability



Availability is the proportion of time that the system is functional and working, usually measured as a percentage of uptime. It can be affected by system errors, infrastructure problems, malicious attacks, and system load. Cloud applications typically provide users with a service level agreement (SLA), so applications must be designed to maximize availability.

### 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.

### Management and Monitoring



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.

	<p><b>Performance and Scalability</b></p> <p>Performance is an indication of the responsiveness of a system to execute any action within a given time interval, while 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. Cloud applications typically encounter variable workloads and peaks in activity. Predicting these, especially in a multi-tenant scenario, is almost impossible. Instead, applications should be able to scale out within limits to meet peaks in demand, and scale in when demand decreases. Scalability concerns not just compute instances, but other elements such as data storage, messaging infrastructure, and more.</p>
	<p><b>Resiliency</b></p> <p>Resiliency is the ability of a system to gracefully handle and recover from failures. 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. Detecting failures, and recovering quickly and efficiently, is necessary to maintain resiliency.</p>
	<p><b>Security</b></p> <p>Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. Cloud applications are exposed on the Internet outside trusted on-premises boundaries, are often open to the public, and may serve untrusted users. Applications must be designed and deployed in a way that protects them from malicious attacks, restricts access to only approved users, and protects sensitive data.</p>

## Catalog of patterns

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.
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Cache-Aside	Load data on demand into a cache from a data store
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation.
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel.

PATTERN	SUMMARY
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
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.
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.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
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.
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.
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.
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.

PATTERN	SUMMARY
<a href="#">Scheduler Agent Supervisor</a>	Coordinate a set of actions across a distributed set of services and other remote resources.
<a href="#">Sharding</a>	Divide a data store into a set of horizontal partitions or shards.
<a href="#">Sidecar</a>	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
<a href="#">Static Content Hosting</a>	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
<a href="#">Strangler</a>	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.
<a href="#">Throttling</a>	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service.
<a href="#">Valet Key</a>	Use a token or key that provides clients with restricted direct access to a specific resource or service.

# Availability patterns

4/11/2018 • 2 minutes to read • [Edit Online](#)

Availability defines the proportion of time that the system is functional and working. It will be affected by system errors, infrastructure problems, malicious attacks, and system load. It is usually measured as a percentage of uptime. Cloud applications typically provide users with a service level agreement (SLA), which means that applications must be designed and implemented in a way that maximizes availability.

PATTERN	SUMMARY
<a href="#">Health Endpoint Monitoring</a>	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
<a href="#">Queue-Based Load Leveling</a>	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
<a href="#">Throttling</a>	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service.

# Data Management patterns

4/11/2018 • 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.

PATTERN	SUMMARY
<a href="#">Cache-Aside</a>	Load data on demand into a cache from a data store
<a href="#">CQRS</a>	Segregate operations that read data from operations that update data by using separate interfaces.
<a href="#">Event Sourcing</a>	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
<a href="#">Index Table</a>	Create indexes over the fields in data stores that are frequently referenced by queries.
<a href="#">Materialized View</a>	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
<a href="#">Sharding</a>	Divide a data store into a set of horizontal partitions or shards.
<a href="#">Static Content Hosting</a>	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
<a href="#">Valet Key</a>	Use a token or key that provides clients with restricted direct access to a specific resource or service.

# Design and Implementation patterns

4/11/2018 • 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	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

# Messaging patterns

4/11/2018 • 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
<a href="#">Competing Consumers</a>	Enable multiple concurrent consumers to process messages received on the same messaging channel.
<a href="#">Pipes and Filters</a>	Break down a task that performs complex processing into a series of separate elements that can be reused.
<a href="#">Priority Queue</a>	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
<a href="#">Queue-Based Load Leveling</a>	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
<a href="#">Scheduler Agent Supervisor</a>	Coordinate a set of actions across a distributed set of services and other remote resources.

# Management and Monitoring patterns

4/11/2018 • 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
<a href="#">Ambassador</a>	Create helper services that send network requests on behalf of a consumer service or application.
<a href="#">Anti-Corruption Layer</a>	Implement a façade or adapter layer between a modern application and a legacy system.
<a href="#">External Configuration Store</a>	Move configuration information out of the application deployment package to a centralized location.
<a href="#">Gateway Aggregation</a>	Use a gateway to aggregate multiple individual requests into a single request.
<a href="#">Gateway Offloading</a>	Offload shared or specialized service functionality to a gateway proxy.
<a href="#">Gateway Routing</a>	Route requests to multiple services using a single endpoint.
<a href="#">Health Endpoint Monitoring</a>	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
<a href="#">Sidecar</a>	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
<a href="#">Strangler</a>	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

# Performance and Scalability patterns

4/11/2018 • 2 minutes to read • [Edit Online](#)

Performance is an indication of the responsiveness of a system to execute any action within a given time interval, while 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. Cloud applications typically encounter variable workloads and peaks in activity. Predicting these, especially in a multi-tenant scenario, is almost impossible. Instead, applications should be able to scale out within limits to meet peaks in demand, and scale in when demand decreases. Scalability concerns not just compute instances, but other elements such as data storage, messaging infrastructure, and more.

PATTERN	SUMMARY
<a href="#">Cache-Aside</a>	Load data on demand into a cache from a data store
<a href="#">CQRS</a>	Segregate operations that read data from operations that update data by using separate interfaces.
<a href="#">Event Sourcing</a>	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
<a href="#">Index Table</a>	Create indexes over the fields in data stores that are frequently referenced by queries.
<a href="#">Materialized View</a>	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
<a href="#">Priority Queue</a>	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
<a href="#">Queue-Based Load Leveling</a>	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
<a href="#">Sharding</a>	Divide a data store into a set of horizontal partitions or shards.
<a href="#">Static Content Hosting</a>	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
<a href="#">Throttling</a>	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service.

# Resiliency patterns

4/11/2018 • 2 minutes to read • [Edit Online](#)

Resiliency is the ability of a system to gracefully handle and recover from failures. 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. 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.

# Security patterns

4/11/2018 • 2 minutes to read • [Edit Online](#)

Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. Cloud applications are exposed on the Internet outside trusted on-premises boundaries, are often open to the public, and may serve untrusted users. Applications must be designed and deployed in a way that protects them from malicious attacks, restricts access to only approved users, and protects sensitive data.

PATTERN	SUMMARY
<a href="#">Federated Identity</a>	Delegate authentication to an external identity provider.
<a href="#">Gatekeeper</a>	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.
<a href="#">Valet Key</a>	Use a token or key that provides clients with restricted direct access to a specific resource or service.

# Ambassador pattern

7/3/2017 • 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.

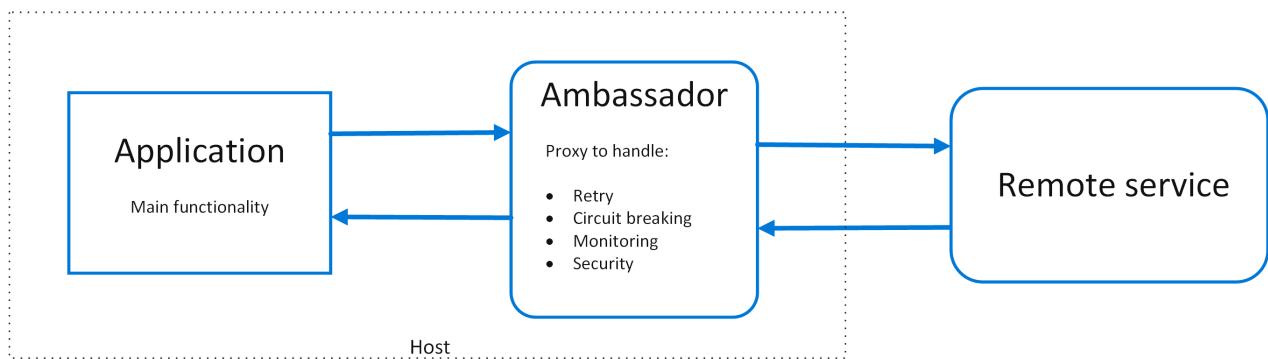
## Context and problem

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:

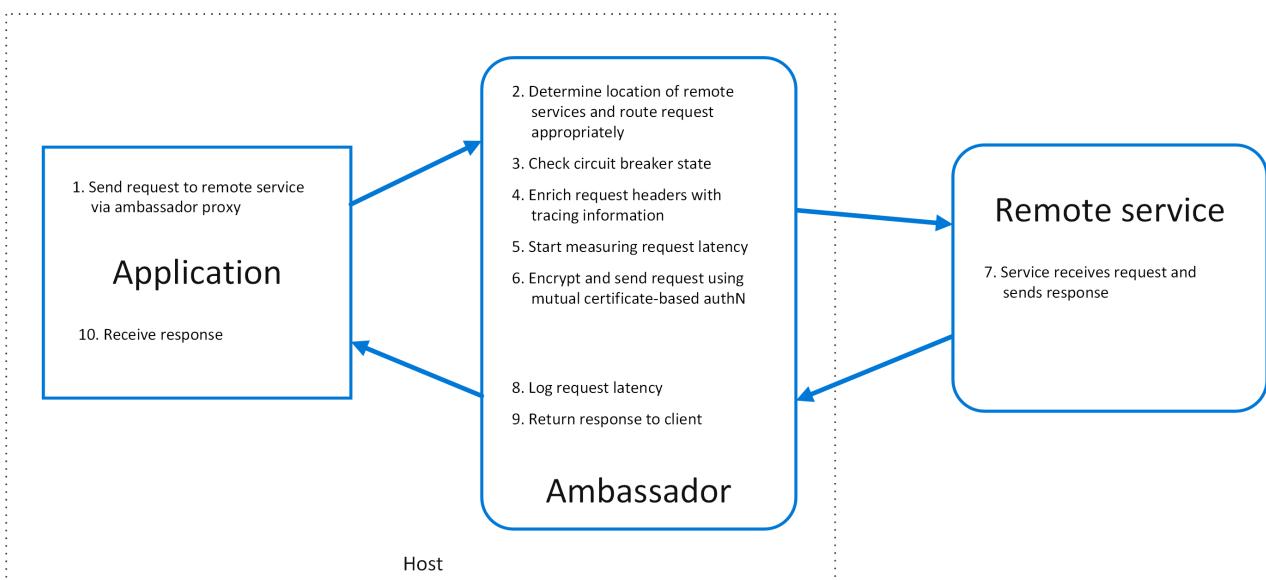
- Need to build a common set of client connectivity features for multiple languages or frameworks.
- Need to offload cross-cutting client connectivity concerns to infrastructure developers or other more specialized teams.
- Need to support cloud or cluster connectivity requirements in a legacy application 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

4/11/2018 • 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.

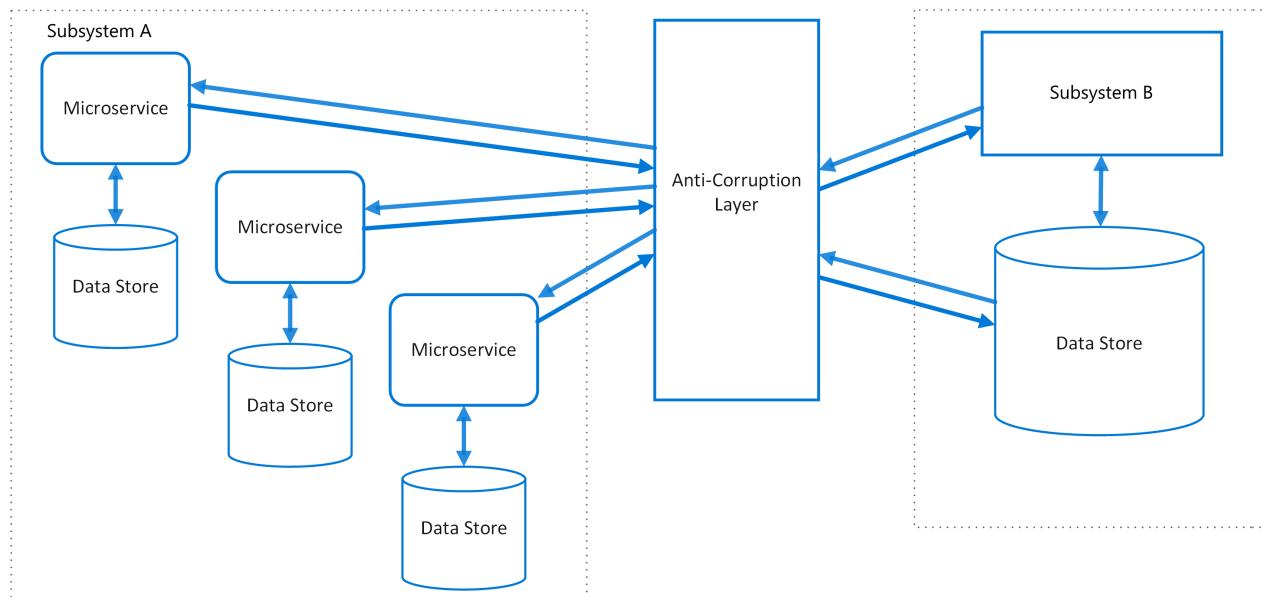
Often these legacy systems suffer from quality issues such as convoluted data schemas or obsolete APIs. The features and technologies used in legacy systems can vary widely from more modern systems. To interoperate with the legacy system, the new application may need to support outdated infrastructure, 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 pattern](#)

# Backends for Frontends pattern

9/28/2018 • 3 minutes to read • [Edit Online](#)

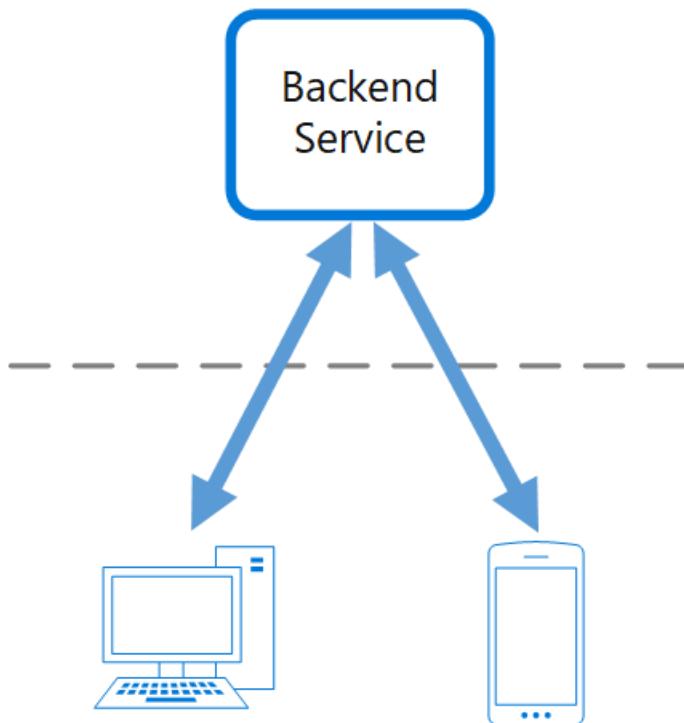
Create separate backend services to be consumed by specific frontend applications or interfaces. This pattern is useful when you want to avoid customizing a single backend for multiple interfaces. This pattern was first described by Sam Newman.

## 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.

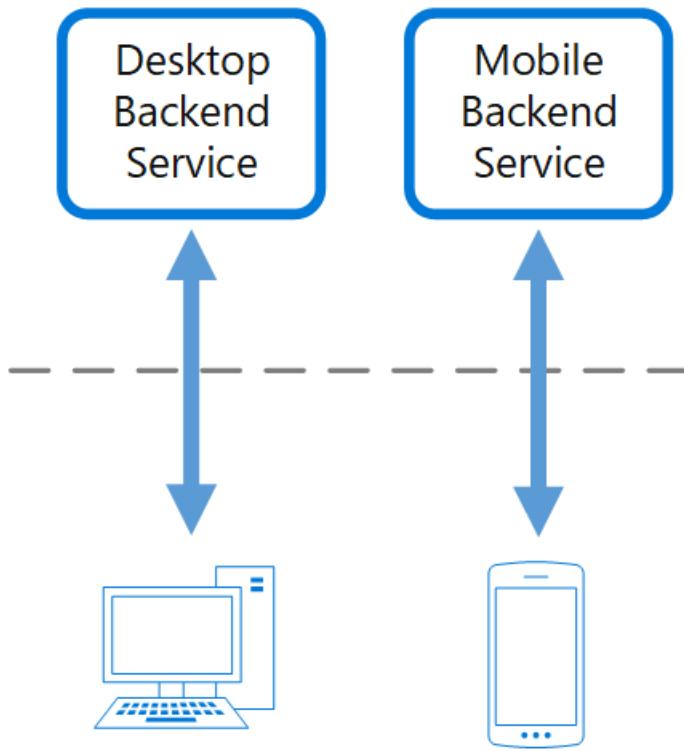
These differences result in competing requirements for the backend. The backend requires regular and significant changes to serve both the desktop web UI and the mobile application. Often, separate interface teams work on each frontend, causing the backend to become a bottleneck in the development process. 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.

## Related guidance

- [Gateway Aggregation pattern](#)
- [Gateway Offloading pattern](#)
- [Gateway Routing pattern](#)

# Bulkhead pattern

4/14/2018 • 4 minutes to read • [Edit Online](#)

Isolate elements of an application into pools so that if one fails, the others will continue to function.

This pattern is named *Bulkhead* because it resembles the sectioned partitions 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 impacted. 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

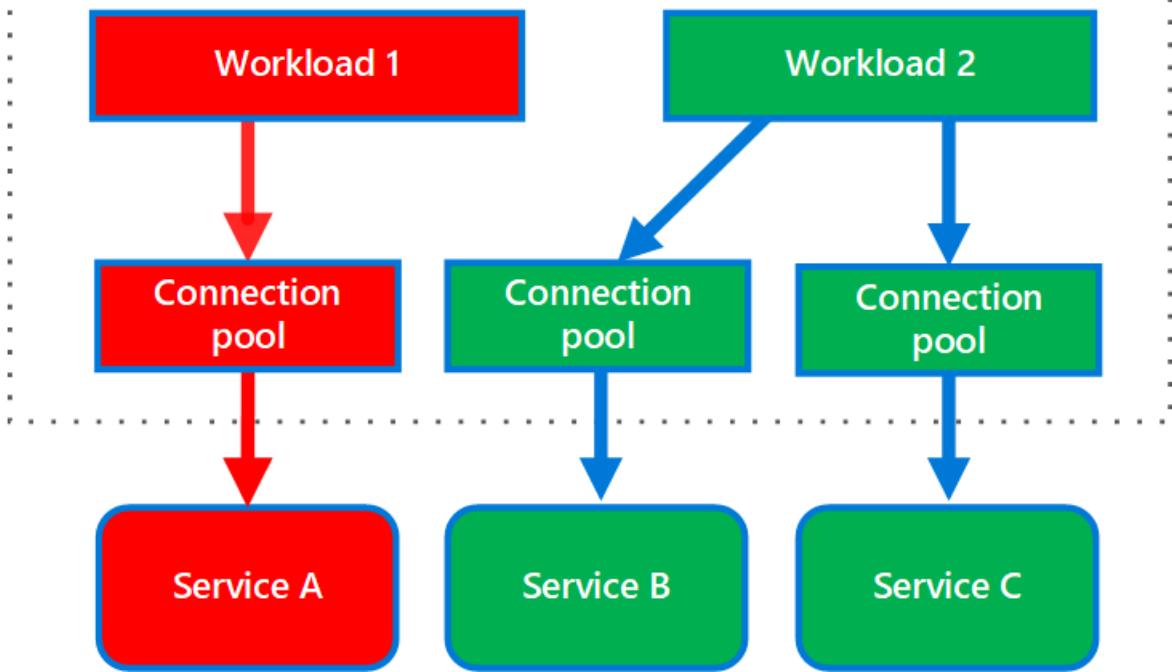
Partition service instances into different groups, based on consumer load and availability requirements. This design helps to isolate failures, and allows you to sustain service functionality for some consumers, even during a failure.

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.

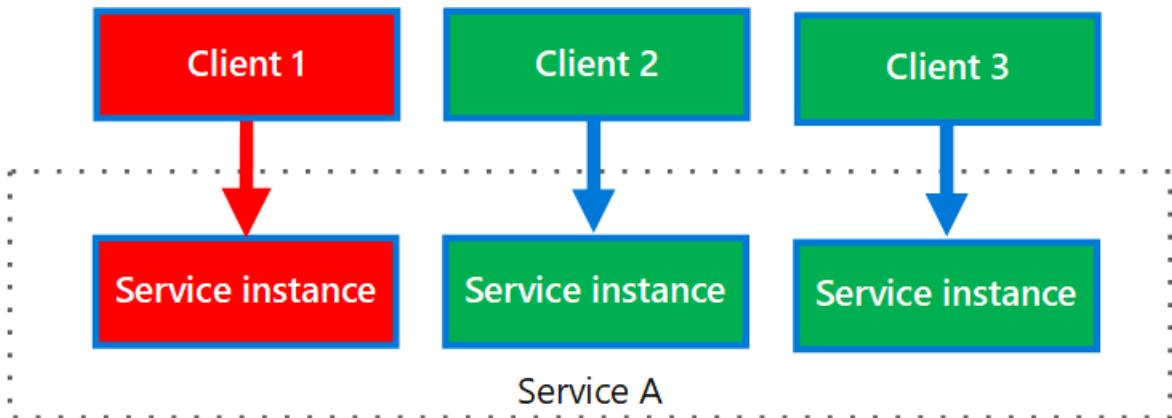
The benefits of this pattern include:

- 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 [Netflix Hystrix](#) 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

- [Circuit Breaker pattern](#)
- [Designing resilient applications for Azure](#)
- [Retry pattern](#)
- [Throttling pattern](#)

# Cache-Aside pattern

4/14/2018 • 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

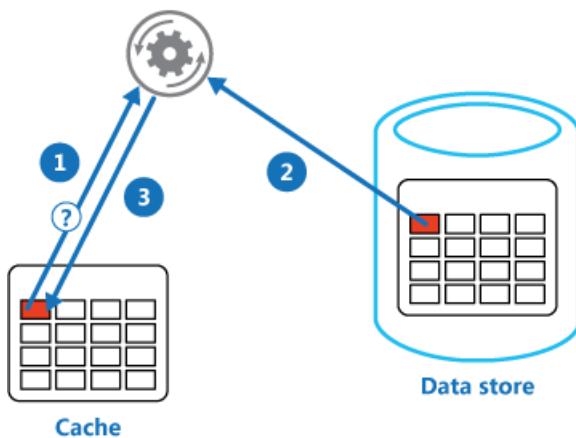
Applications use a cache to improve repeated access to information held in a data store. However, it's impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache 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.



- 1: Determine whether the item is currently held in the cache.
- 2: If the item is not currently in the cache, read the item from the data store.
- 3: Store a copy of the item 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 Redis Cache to create a distributed cache that can be shared by multiple instances of an application.

To connect to an Azure Redis Cache instance, call the static `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 info
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 based on Azure Redis Cache. 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's 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 the Azure Redis Cache API to access the store and retrieve information from the cache. For

more information, see [Using Microsoft Azure Redis Cache](#) and [How to create a Web App with Redis Cache](#)

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.

# Circuit Breaker pattern

6/28/2018 • 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

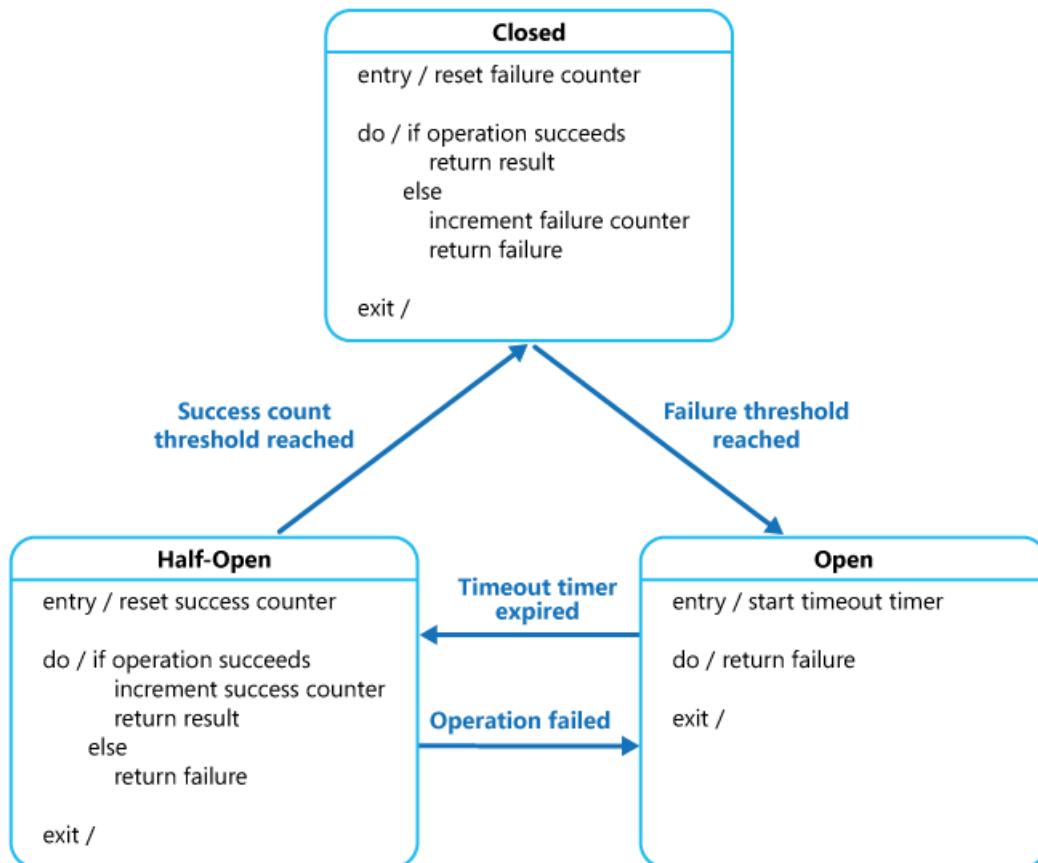
electrical circuit breaker:

- **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 been 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 patterns and 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.

# Command and Query Responsibility Segregation (CQRS) pattern

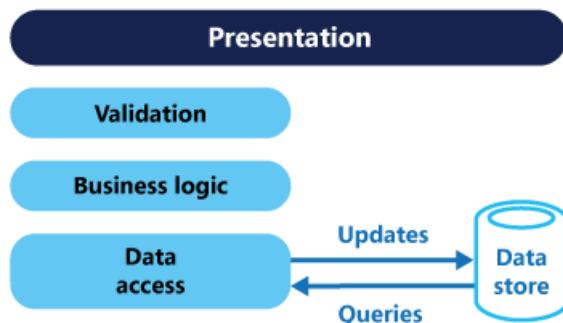
9/28/2018 • 12 minutes to read • [Edit Online](#)

Segregate operations that read data from operations that update data by using separate interfaces. This can maximize performance, scalability, and security. Supports the evolution of the system over time through higher flexibility, and prevents update commands from causing merge conflicts at the domain level.

## Context and problem

In traditional data management systems, both commands (updates to the data) and queries (requests for data) are executed against the same set of entities in a single data repository. These entities can be a subset of the rows in one or more tables in a relational database such as SQL Server.

Typically in these systems, all create, read, update, and delete (CRUD) operations are applied to the same representation of the entity. For example, a data transfer object (DTO) representing a customer is retrieved from the data store by the data access layer (DAL) and displayed on the screen. A user updates some fields of the DTO (perhaps through data binding) and the DTO is then saved back in the data store by the DAL. The same DTO is used for both the read and write operations. The figure illustrates a traditional CRUD architecture.



Traditional CRUD designs work well when only limited business logic is applied to the data operations. Scaffold mechanisms provided by development tools can create data access code very quickly, which can then be customized as required.

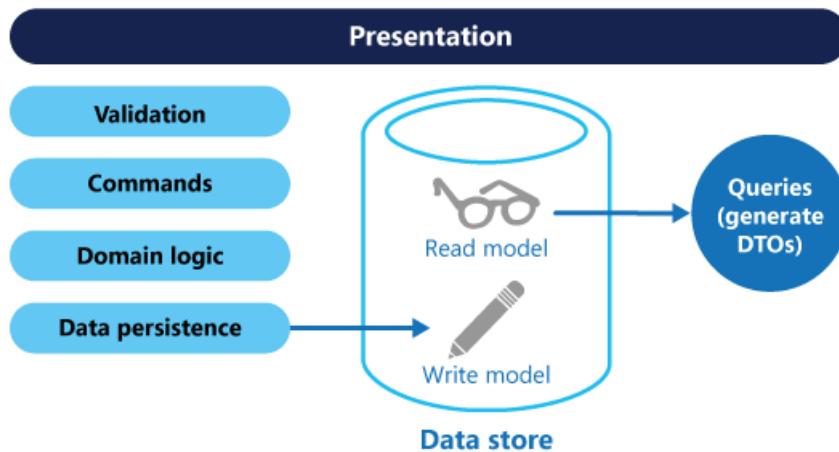
However, the traditional CRUD approach has some disadvantages:

- It often means that there's 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.
- It risks data contention when records are locked in the data store in a collaborative domain, where multiple actors operate in parallel on the same set of data. Or update conflicts caused by concurrent updates when optimistic locking is used. These risks increase as the complexity and throughput of the system grows. In addition, 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.
- It can make managing security and permissions more complex because each entity is subject to both read and write operations, which might expose data in the wrong context.

For a deeper understanding of the limits of the CRUD approach see [CRUD, Only When You Can Afford It.](#)

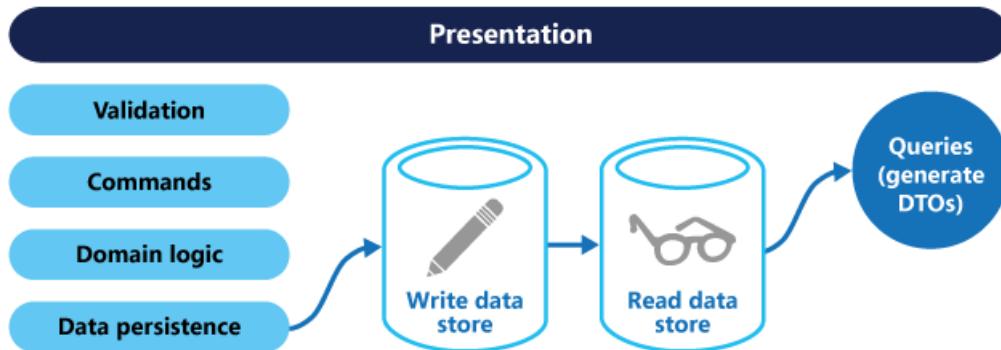
## Solution

Command and Query Responsibility Segregation (CQRS) is a pattern that segregates the operations that read data (queries) from the operations that update data (commands) by using separate interfaces. This means that the data models used for querying and updates are different. The models can then be isolated, as shown in the following figure, although that's not an absolute requirement.



Compared to the single data model used in CRUD-based systems, the use of separate query and update models for the data in CQRS-based systems simplifies design and implementation. However, one disadvantage is that unlike CRUD designs, CQRS code can't automatically be generated using scaffold mechanisms.

The query model for reading data and the update model for writing data can access the same physical store, perhaps by using SQL views or by generating projections on the fly. However, it's common to separate the data into different physical stores to maximize performance, scalability, and security, as shown in the next figure.



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 of the read store can greatly increase query performance and application UI responsiveness, especially in distributed scenarios where read-only replicas are located close to the application instances. Some database systems (SQL Server) provide additional features such as failover replicas to maximize availability.

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.

When the query/read model contains denormalized data (see [Materialized View pattern](#)), performance is maximized when reading data for each of the views in an application or when querying the data in the system.

## Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Dividing the data store into separate physical stores for read and write operations can increase the performance and security of a system, but it can add complexity in terms of resiliency and eventual

consistency. 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, which means that the operation can't be completed.

For a description of eventual consistency see the [Data Consistency Primer](#).

- Consider applying CQRS to limited sections of your system where it will be most valuable.
- A typical approach to deploying eventual consistency is to use event sourcing in conjunction with CQRS so that the write model is an append-only stream of events driven by execution of commands. These events are used to update materialized views that act as the read model. For more information see [Event Sourcing and CQRS](#).

## When to use this pattern

Use this pattern in the following situations:

- Collaborative domains where multiple operations are performed in parallel on the same data. CQRS allows you to define commands with enough granularity to minimize merge conflicts at the domain level (any conflicts that do arise can be merged by the command), even when updating what appears to be the same type of data.
- Task-based user interfaces where users are guided through a complex process as a series of steps or with complex domain models. Also, useful for teams already familiar with domain-driven design (DDD) techniques. The write model has a full command-processing stack with business logic, input validation, and business validation to ensure that everything is always consistent for each of the aggregates (each cluster of associated objects treated as a unit for data changes) in the write model. 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 read/write ratio is very high, and when horizontal scaling is required. For example, in many systems the number of read operations is many times greater than the number of write operations. To accommodate this, consider scaling out the read model, but running the write model on only one or 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 in the following situations:

- Where the domain or the business rules are simple.
- Where a simple CRUD-style user interface and the related data access operations are sufficient.
- For implementation across the whole system. There are specific components of an overall data management scenario where CQRS can be useful, but it can add considerable and unnecessary complexity when it isn't required.

## Event Sourcing and CQRS

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 have occurred, or the current state of an entity.

## Example

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 )
    {
        ...
    }
}

```

The following code shows the `IProductsDomain` interface from the write model.

```

public interface IProductsDomain
{
    void AddNewProduct(int id, string name, string description, decimal price);
    void RateProduct(int userId, int rating);
    void AddToInventory(int productId, int quantity);
    void ConfirmItemsShipped(int productId, int quantity);
    void UpdateStockFromInventoryRecount(int productId, int updatedQuantity);
}

```

Also notice how the `IProductsDomain` interface contains methods that have a meaning in the domain. Typically, in a CRUD environment these methods would have generic names such as `Save` or `Update`, and have a DTO as the only argument. The CQRS approach can be designed to meet the needs of this organization's business and inventory management systems.

## Related patterns and guidance

The following patterns and guidance are useful when implementing this pattern:

- For a comparison of CQRS with other architectural styles, see [Architecture styles](#) and [CQRS architecture style](#).
- [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.
- [Data Partitioning Guidance](#). Describes how the read and write data stores used in the CQRS pattern can be divided into partitions that can be managed and accessed separately to improve scalability, reduce contention, and optimize performance.
- [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.
- 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.
- The post [CQRS by Martin Fowler](#), which explains the basics of the pattern and links to other useful resources.

# Compensating Transaction pattern

9/28/2018 • 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.

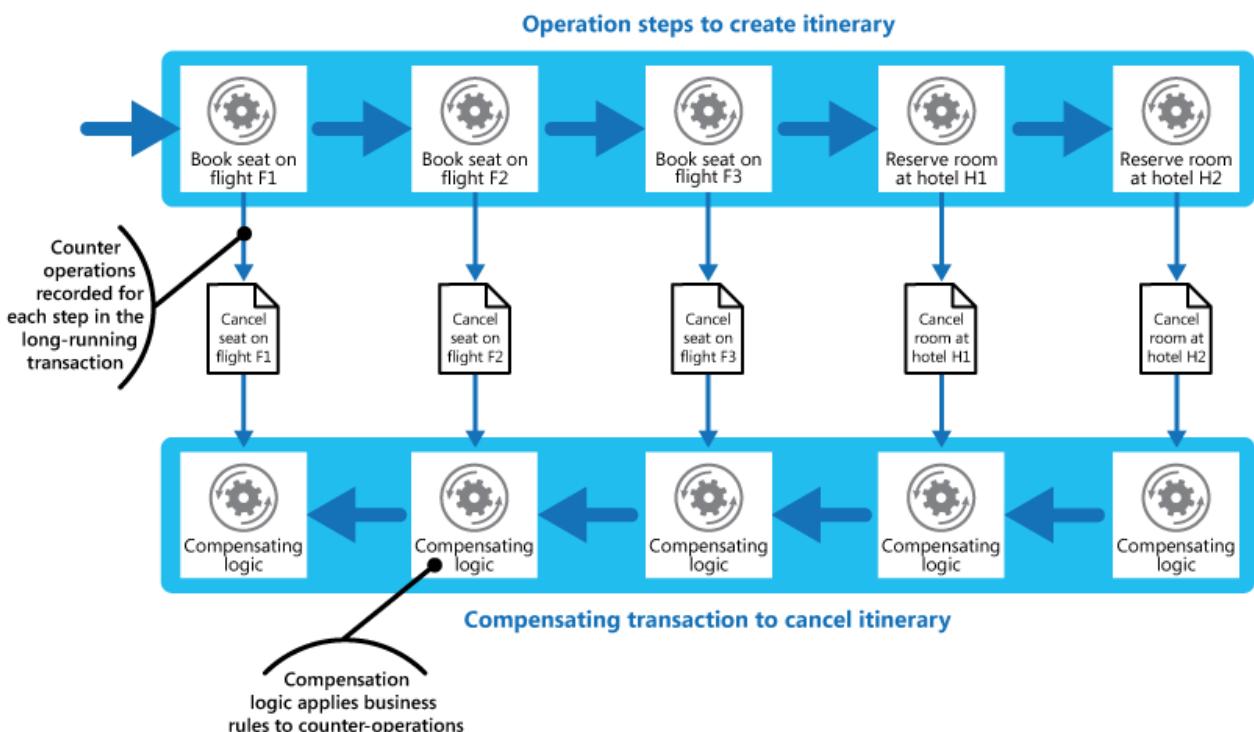
## Example

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.



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 patterns and 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

9/28/2018 • 9 minutes to read • [Edit Online](#)

Enable multiple concurrent consumers to process messages received on the same messaging channel. This enables a system to process multiple messages concurrently to optimize throughput, to improve scalability and availability, and to balance the workload.

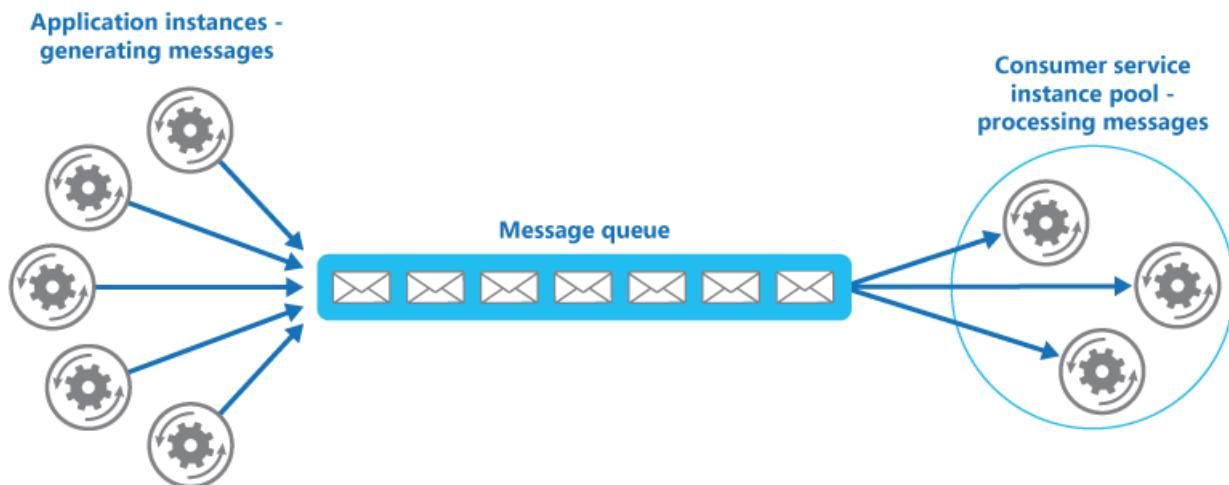
## Context and problem

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. Using a single instance of the consumer service can cause that instance to become flooded with requests, or the messaging system might be overloaded by an influx of messages coming 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 can help to minimize the impact on availability and responsiveness for both the

application and the service instances, as described by the [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 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 storage queues and Service Bus queues that can act as a mechanism for implementing this pattern. The application logic can post messages to a queue, and consumers implemented as tasks in one or more roles can retrieve messages from this queue and process 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 original consumer can delete the message when it's finished processing it. If the consumer fails, the peek lock will time out and the message will become visible again, allowing another consumer to retrieve it.

For detailed information on using Azure Service Bus queues, see [Service Bus queues, topics, and subscriptions](#). For information on using Azure storage queues, see [Get started with Azure Queue storage using .NET](#).

The following code from the `QueueManager` class in CompetingConsumers solution available on [GitHub](#) shows how you can create a queue by using a `QueueClient` instance in the `Start` event handler in a web or worker role.

```

private string queueName = ...;
private string connectionString = ...;
...

public async Task Start()
{
    // Check if the queue already exists.
    var manager = NamespaceManager.CreateFromConnectionString(this.connectionString);
    if (!manager.QueueExists(this.queueName))
    {
        var queueDescription = new QueueDescription(this.queueName);

        // Set the maximum delivery count for messages in the queue. A message
        // is automatically dead-lettered after this number of deliveries. The
        // default value for dead letter count is 10.
        queueDescription.MaxDeliveryCount = 3;

        await manager.CreateQueueAsync(queueDescription);
    }
    ...

    // Create the queue client. By default the PeekLock method is used.
    this.client = QueueClient.CreateFromConnectionString(
        this.connectionString, this.queueName);
}

```

The next code snippet shows how an application can create and send a batch of messages to the queue.

```

public async Task SendMessagesAsync()
{
    // Simulate sending a batch of messages to the queue.
    var messages = new List<BrokeredMessage>();

    for (int i = 0; i < 10; i++)
    {
        var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
        messages.Add(message);
    }
    await this.client.SendBatchAsync(messages);
}

```

The following code shows how a consumer service instance can receive messages from the queue by following an event-driven approach. The `processMessageTask` parameter to the `ReceiveMessages` method is a delegate that references the code to run when a message is received. This code is run asynchronously.

```

private ManualResetEvent pauseProcessingEvent;
...

public void ReceiveMessages(Func<BrokeredMessage, Task> processMessageTask)
{
    // Set up the options for the message pump.
    var options = new OnMessageOptions();

    // When AutoComplete is disabled it's necessary to manually
    // complete or abandon the messages and handle any errors.
    options.AutoComplete = false;
    options.MaxConcurrentCalls = 10;
    options.ExceptionReceived += this.OptionsOnExceptionReceived;

    // Use of the Service Bus OnMessage message pump.
    // The OnMessage method must be called once, otherwise an exception will occur.
    this.client.OnMessageAsync(
        async (msg) =>
    {
        // Will block the current thread if Stop is called.
        this.pauseProcessingEvent.WaitOne();

        // Execute processing task here.
        await processMessageTask(msg);
    },
    options);
}
...

private void OptionsOnExceptionReceived(object sender,
    ExceptionReceivedEventArgs exceptionReceivedEventArgs)
{
    ...
}

```

Note that autoscaling features, such as those available in Azure, can be used to start and stop role instances as the queue length fluctuates. For more information, see [Autoscaling Guidance](#). Also, it's not necessary to maintain a one-to-one correspondence between role instances and worker processes—a single role instance can implement multiple worker processes. For more information, see [Compute Resource Consolidation pattern](#).

## Related patterns and 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.

- This pattern has a [sample application](#) associated with it.

# Compute Resource Consolidation pattern

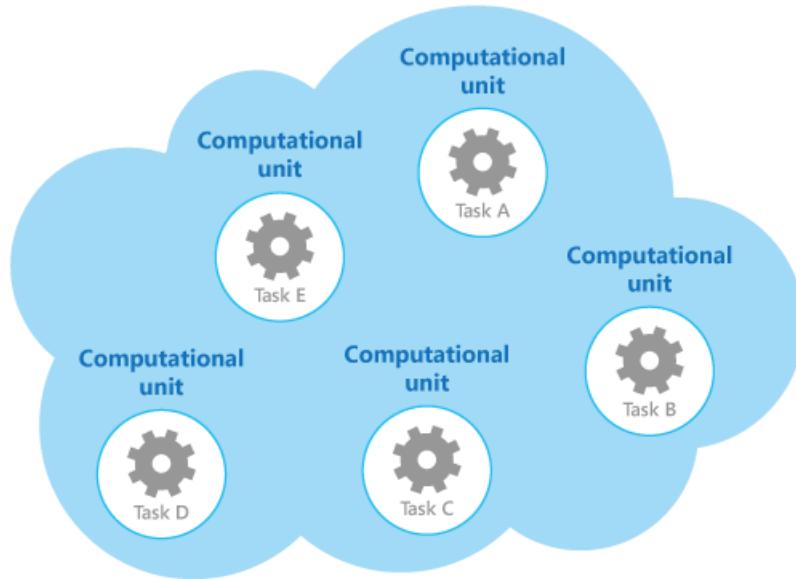
9/28/2018 • 12 minutes to read • [Edit Online](#)

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 utilizes 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.

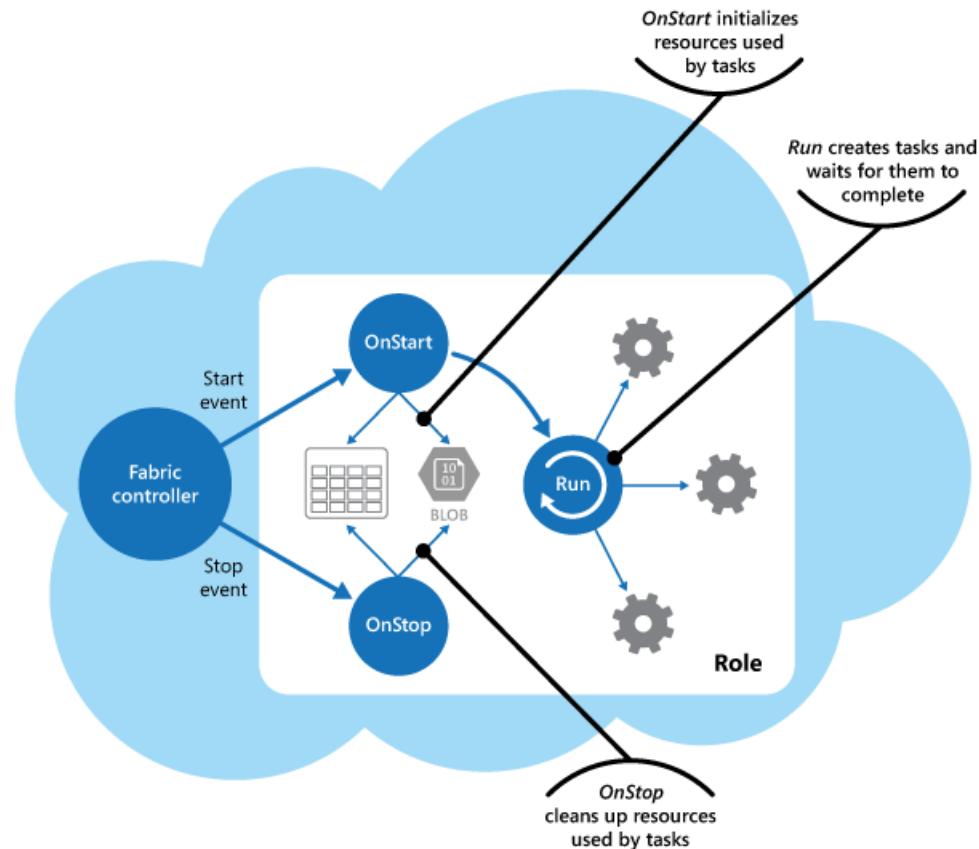
Place the code that actually creates the tasks in the `Run` method. Note that the `Run` method defines the lifetime

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 patterns and 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](#).

# Event Sourcing pattern

9/28/2018 • 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.

For a deeper understanding of the limits of the CRUD approach see [CRUD, Only When You Can Afford It](#).

## Solution

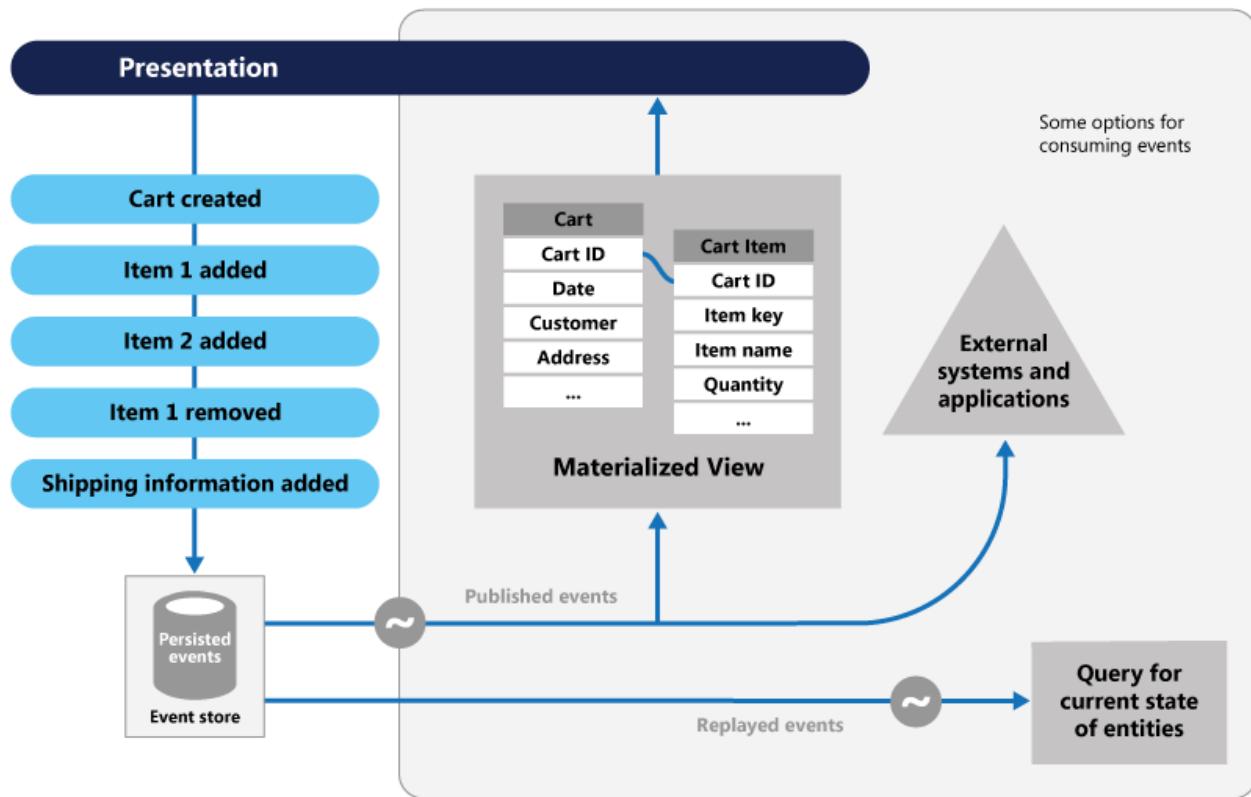
The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where 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 [Snapshot on Martin Fowler's Enterprise Application Architecture website](#) and [Master-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 an 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.

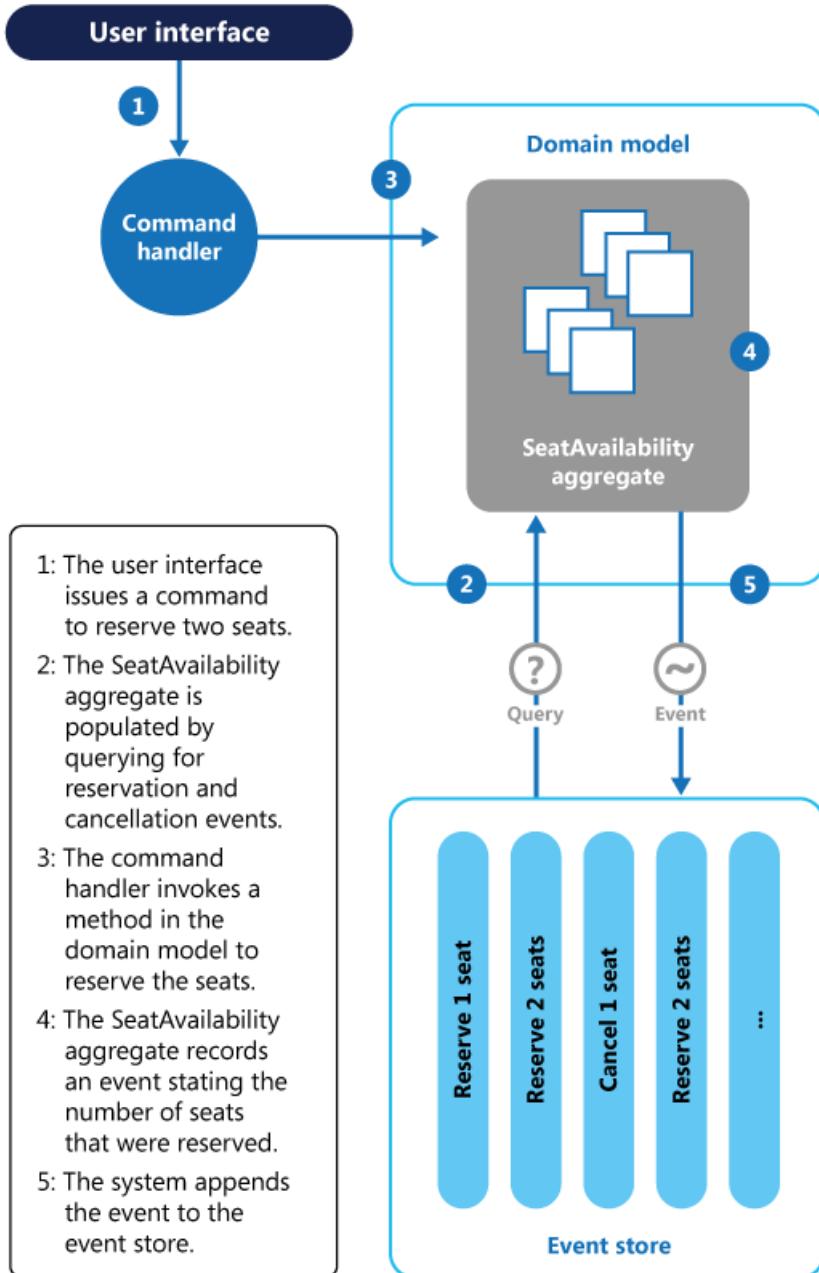
## Example

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](#).

## Related patterns and 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

8/14/2017 • 10 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.

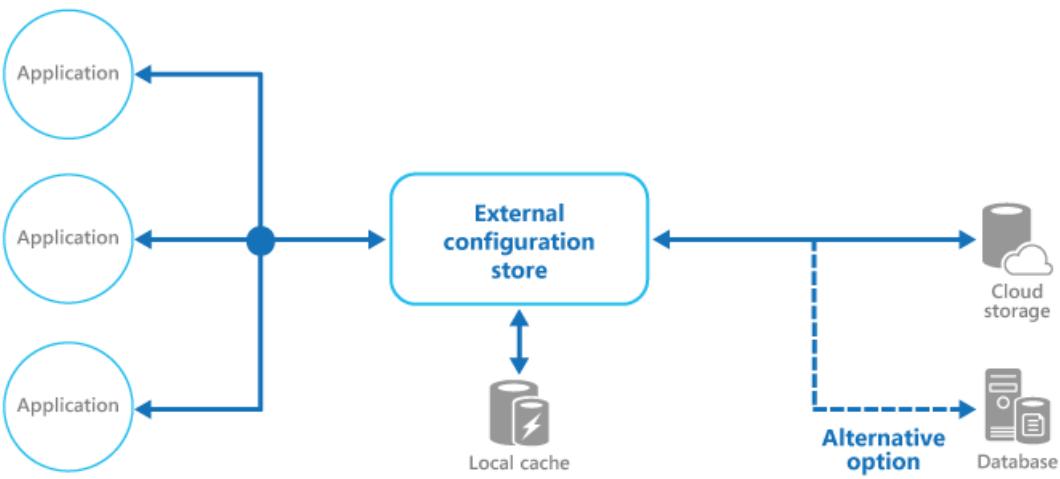
In addition, updates to applications and components might require changes to configuration schemas. Many configuration systems don't support different versions of configuration information.

## 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, but could be a hosted database or other 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 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).

## 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.

## Example

In a Microsoft Azure hosted application, a typical 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.

The following example shows how a configuration store can be implemented over Blob storage to store and expose configuration information. The `BlobSettingsStore` class abstracts Blob storage for holding configuration information, and implements the `ISettingsStore` interface shown in the following code.

This code is provided in the *ExternalConfigurationStore.Cloud* project in the *ExternalConfigurationStore* solution, available from [GitHub](#).

```
public interface ISettingsStore
{
    Task<string> GetVersionAsync();

    Task<Dictionary<string, string>> FindAllAsync();
}
```

This interface defines methods for retrieving and updating 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. The `BlobSettingsStore` class uses the `ETag` property of the blob to implement versioning. The `ETag` property is updated automatically each time the blob is written.

By design, this simple solution exposes all configuration settings as string values rather than typed values.

The `ExternalConfigurationManager` class provides a wrapper around a `BlobSettingsStore` object. An application can use this class to store and retrieve configuration information. This class uses the Microsoft [Reactive Extensions](#) library to expose any changes made to the configuration through an implementation of the `IObservable` interface. If a setting is modified by calling the `SetAppSetting` method, the `Changed` event is raised and all subscribers to

this event will be notified.

Note that all settings are also cached in a `Dictionary` object inside the `ExternalConfigurationManager` class for fast access. The `GetSetting` method used to retrieve a configuration setting reads the data from the cache. If the setting isn't found in the cache, it's retrieved from the `BlobSettingsStore` object instead.

The `GetSettings` method invokes the `CheckForConfigurationChanges` method to detect whether the configuration information in blob storage has changed. It does this by examining the version number and comparing it with the current version number held by the `ExternalConfigurationManager` object. If one or more changes have occurred, the `Changed` event is raised and the configuration settings cached in the `Dictionary` object are refreshed. This is an application of the [Cache-Aside pattern](#).

The following code sample shows how the `Changed` event, the `GetSettings` method, and the `CheckForConfigurationChanges` method are implemented:

```
public class ExternalConfigurationManager : IDisposable
{
    // An abstraction of the configuration store.
    private readonly ISettingsStore settings;
    private readonly ISubject<KeyValuePair<string, string>> changed;
    ...
    private readonly ReaderWriterLockSlim settingsCacheLock = new ReaderWriterLockSlim();
    private readonly SemaphoreSlim syncCacheSemaphore = new SemaphoreSlim(1);
    ...
    private Dictionary<string, string> settingsCache;
    private string currentVersion;
    ...
    public ExternalConfigurationManager(ISettingsStore settings, ...)
    {
        this.settings = settings;
        ...
    }
    ...
    public IObservable<KeyValuePair<string, string>> Changed => this.changed.AsObservable();
    ...

    public string GetAppSetting(string key)
    {
        ...
        // Try to get the value from the settings cache.
        // If there's a cache miss, get the setting from the settings store and refresh the settings cache.

        string value;
        try
        {
            this.settingsCacheLock.EnterReadLock();

            this.settingsCache.TryGetValue(key, out value);
        }
        finally
        {
            this.settingsCacheLock.ExitReadLock();
        }

        return value;
    }
    ...
    private void CheckForConfigurationChanges()
    {
        try
        {
            // It is assumed that updates are infrequent.
            // To avoid race conditions in refreshing the cache, synchronize access to the in-memory cache.
            await this.syncCacheSemaphore.WaitAsync();
        }
    }
}
```

```

        var latestVersion = await this.settings.GetVersionAsync();

        // If the versions are the same, nothing has changed in the configuration.
        if (this.currentVersion == latestVersion) return;

        // Get the latest settings from the settings store and publish changes.
        var latestSettings = await this.settings.FindAllAsync();

        // Refresh the settings cache.
        try
        {
            this.settingsCacheLock.EnterWriteLock();

            if (this.settingsCache != null)
            {
                //Notify settings changed
                latestSettings.Except(this.settingsCache).ToList().ForEach(kv => this.changed.OnNext(kv));
            }
            this.settingsCache = latestSettings;
        }
        finally
        {
            this.settingsCacheLock.ExitWriteLock();
        }

        // Update the current version.
        this.currentVersion = latestVersion;
    }
    catch (Exception ex)
    {
        this.changed.OnError(ex);
    }
    finally
    {
        this.syncCacheSemaphore.Release();
    }
}
}

```

The `ExternalConfigurationManager` class also provides a property named `Environment`. This property supports varying configurations for an application running in different environments, such as staging and production.

An `ExternalConfigurationManager` object can also query the `BlobSettingsStore` object periodically for any changes. In the following code, the `StartMonitor` method calls `CheckForConfigurationChanges` at an interval to detect any changes and raise the `Changed` event, as described earlier.

```

public class ExternalConfigurationManager : IDisposable
{
    ...

    private readonly ISubject<KeyValuePair<string, string>> changed;
    private Dictionary<string, string> settingsCache;
    private readonly CancellationTokenSource cts = new CancellationTokenSource();
    private Task monitoringTask;
    private readonly TimeSpan interval;

    private readonly SemaphoreSlim timerSemaphore = new SemaphoreSlim(1);
    ...

    public ExternalConfigurationManager(string environment) : this(new BlobSettingsStore(environment),
TimeSpan.FromSeconds(15), environment)
    {
    }

    public ExternalConfigurationManager(ISettingsStore settings, TimeSpan interval, string environment)
    {
        this.settings = settings;
    }
}

```

```

        this.settings = settings;
        this.interval = interval;
        this.CheckForConfigurationChangesAsync().Wait();
        this.changed = new Subject<KeyValuePair<string, string>>();
        this.Environment = environment;
    }
    ...
    /// <summary>
    /// Check to see if the current instance is monitoring for changes
    /// </summary>
    public bool IsMonitoring => this.monitoringTask != null && !this.monitoringTask.IsCompleted;

    /// <summary>
    /// Start the background monitoring for configuration changes in the central store
    /// </summary>
    public void StartMonitor()
    {
        if (this.IsMonitoring)
            return;

        try
        {
            this.timerSemaphore.Wait();

            // Check again to make sure we are not already running.
            if (this.IsMonitoring)
                return;

            // Start running our task loop.
            this.monitoringTask = ConfigChangeMonitor();
        }
        finally
        {
            this.timerSemaphore.Release();
        }
    }

    /// <summary>
    /// Loop that monitors for configuration changes
    /// </summary>
    /// <returns></returns>
    public async Task ConfigChangeMonitor()
    {
        while (!cts.Token.IsCancellationRequested)
        {
            await this.CheckForConfigurationChangesAsync();
            await Task.Delay(this.interval, cts.Token);
        }
    }

    /// <summary>
    /// Stop monitoring for configuration changes
    /// </summary>
    public void StopMonitor()
    {
        try
        {
            this.timerSemaphore.Wait();

            // Signal the task to stop.
            this.cts.Cancel();

            // Wait for the loop to stop.
            this.monitoringTask.Wait();

            this.monitoringTask = null;
        }
        finally
        {
    
```

```

        this.timerSemaphore.Release();
    }

    public void Dispose()
    {
        this.cts.Cancel();
    }
    ...
}

```

The `ExternalConfigurationManager` class is instantiated as a singleton instance by the `ExternalConfiguration` class shown below.

```

public static class ExternalConfiguration
{
    private static readonly Lazy<ExternalConfigurationManager> configuredInstance = new
Lazy<ExternalConfigurationManager>(
    () =>
    {
        var environment = Cloud ConfigurationManager.GetSetting("environment");
        return new ExternalConfigurationManager(environment);
    });
}

public static ExternalConfigurationManager Instance => configuredInstance.Value;
}

```

The following code is taken from the `WorkerRole` class in the *ExternalConfigurationStore.Cloud* project. It shows how the application uses the `ExternalConfiguration` class to read a setting.

```

public override void Run()
{
    // Start monitoring configuration changes.
    ExternalConfiguration.Instance.StartMonitor();

    // Get a setting.
    var setting = ExternalConfiguration.Instance.GetAppSetting("setting1");
    Trace.Information("Worker Role: Get setting1, value: " + setting);

    this.completeEvent.WaitOne();
}

```

The following code, also from the `WorkerRole` class, shows how the application subscribes to configuration events.

```

public override bool OnStart()
{
    ...
    // Subscribe to the event.
    ExternalConfiguration.Instance.Changed.Subscribe(
        m => Trace.Information("Configuration has changed. Key:{0} Value:{1}",
            m.Key, m.Value),
        ex => Trace.Error("Error detected: " + ex.Message));
    ...
}

```

## Related patterns and guidance

- A sample that demonstrates this pattern is available on [GitHub](#).

# Federated Identity pattern

8/14/2017 • 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.

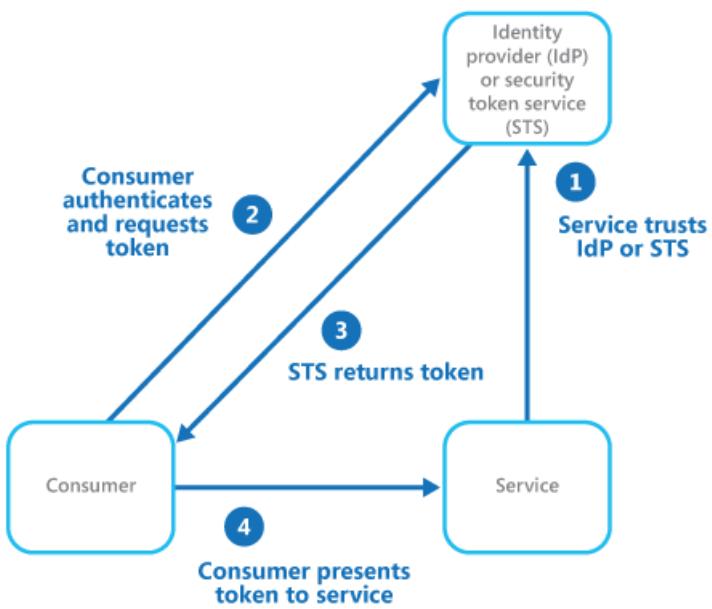
Users typically prefer to use the same credentials for all these applications.

## 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 STSs 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:

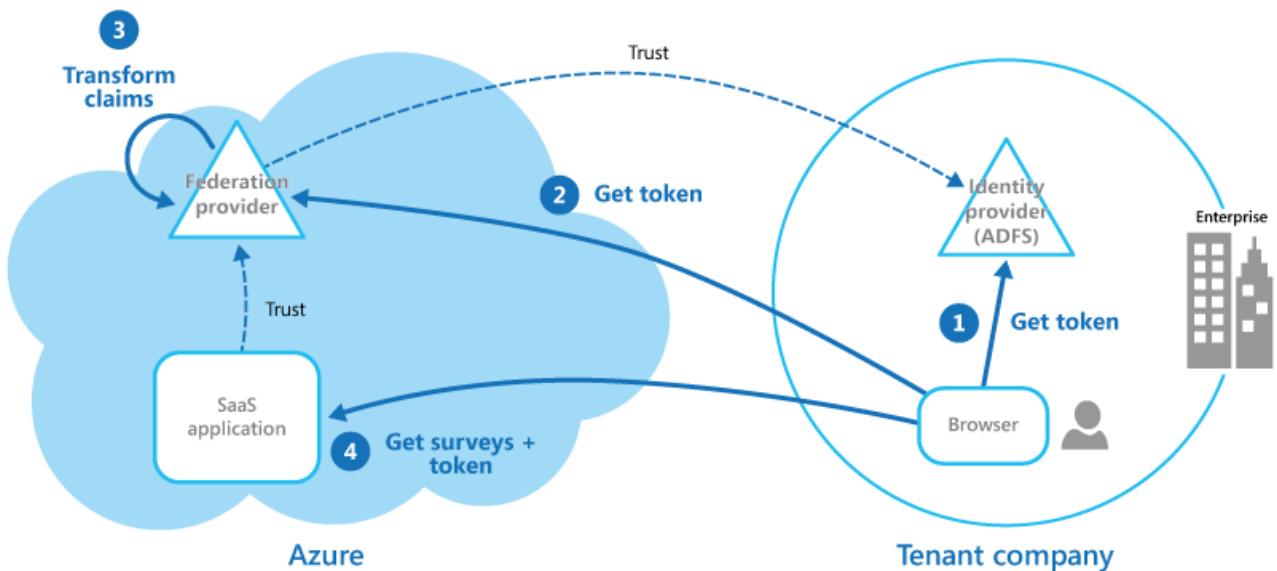
- **Single sign-on in the enterprise.** In this scenario you need to authenticate employees for corporate applications that are hosted in the cloud outside the corporate security boundary, without requiring them to sign in every time they visit an application. The user experience is the same as when 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 (ADFS) 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 ADFS. After successfully authenticating a tenant, ADFS issues a token. The client browser forwards this token to the SaaS application's federation provider, which trusts tokens issued by the tenant's ADFS, 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 ADFS the list of users that can access the application.

## Related guidance

- [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

8/14/2017 • 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.

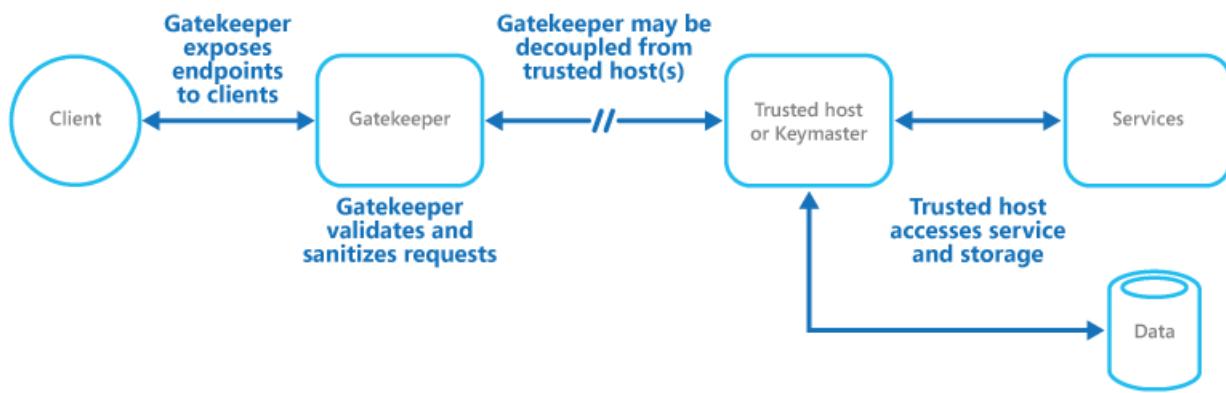
## Context and problem

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 access 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 (sometimes called the keymaster) 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.

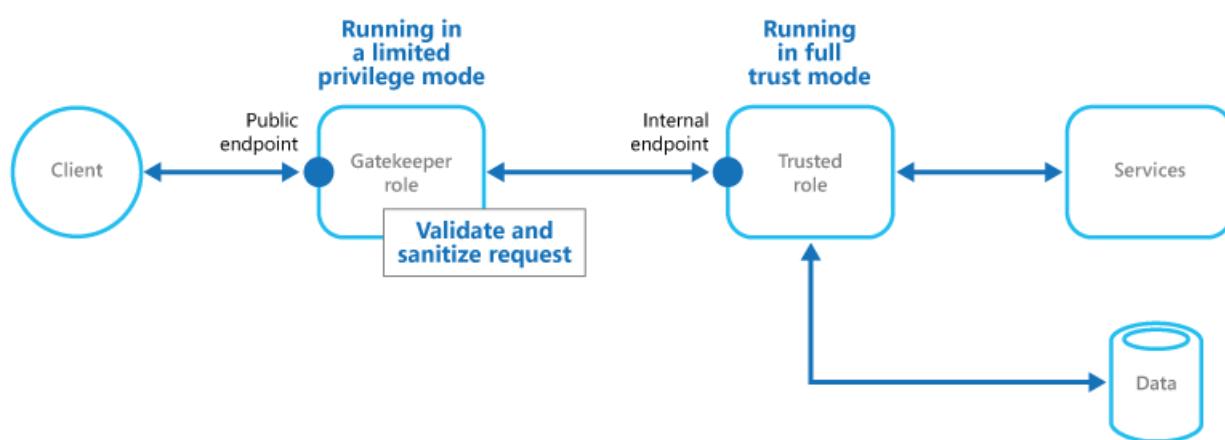
## When to use this pattern

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 patterns

The [Valet Key pattern](#) might also be relevant when implementing the Gatekeeper pattern. When communicating between the Gatekeeper and trusted roles it's good practice to enhance security by using keys or tokens that limit permissions for accessing resources. Describes how to use a token or key that provides clients with restricted

direct access to a specific resource or service.

# Gateway Aggregation pattern

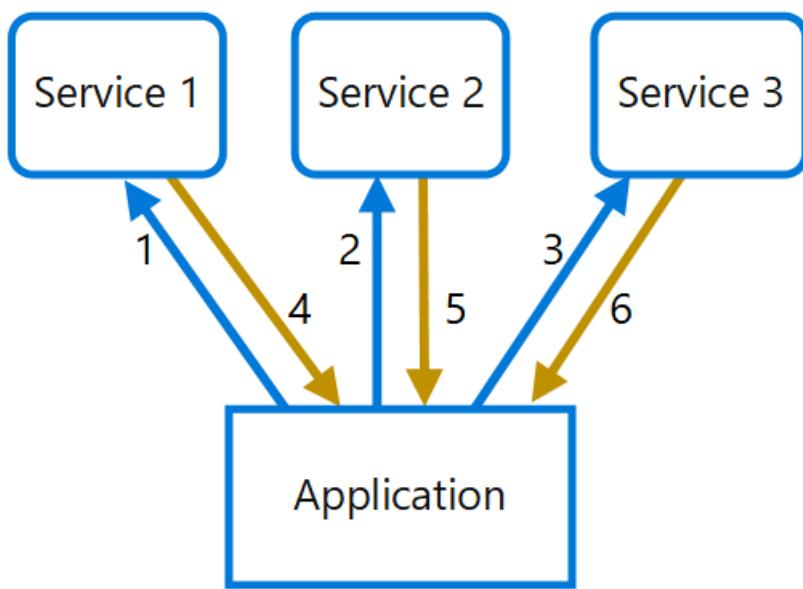
6/26/2017 • 3 minutes to read • [Edit Online](#)

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.

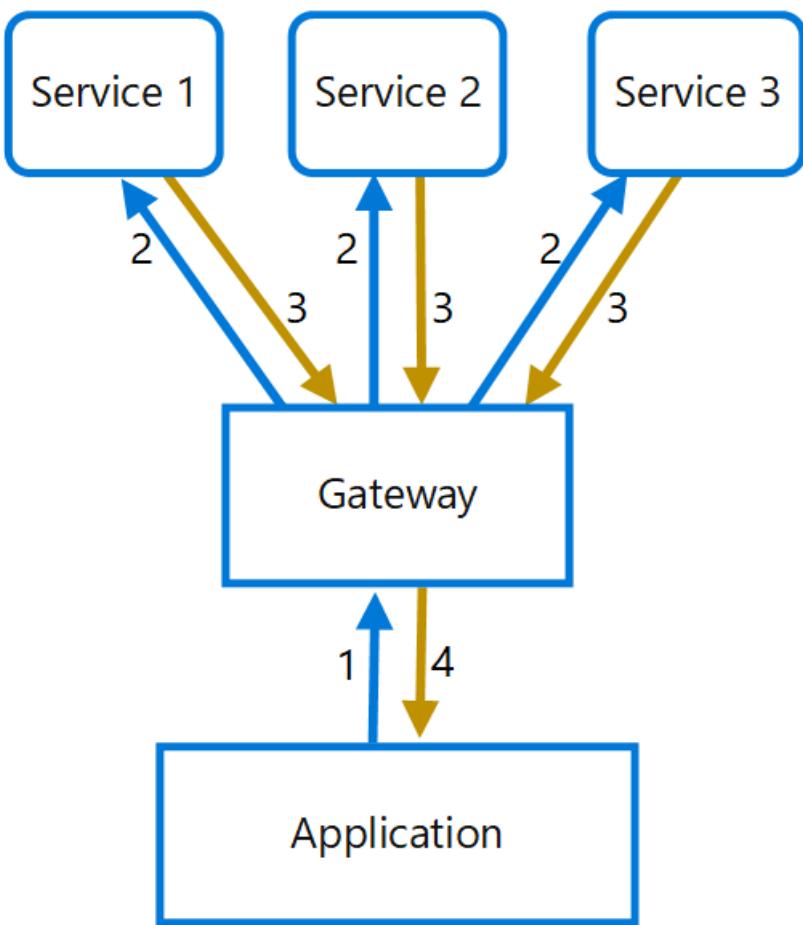


## Solution

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

6/26/2017 • 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.

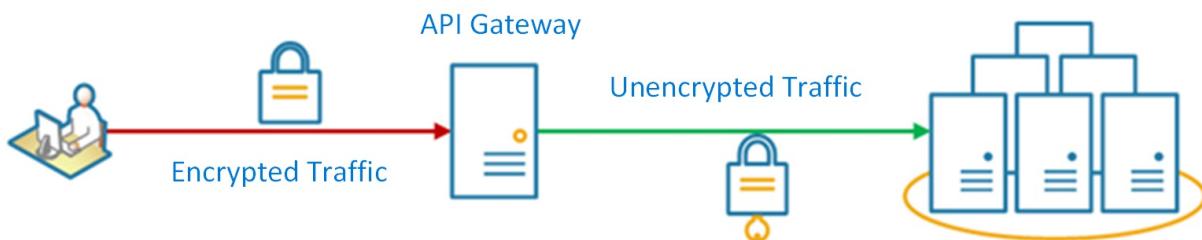
Properly handling security issues (token validation, encryption, SSL certificate management) and other complex tasks can require team members to have highly specialized skills. For example, a certificate needed by an application must be configured and deployed on all application instances. With each 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 an API gateway, particularly cross-cutting concerns such as certificate management, authentication, SSL termination, monitoring, protocol translation, or throttling.

The following diagram shows an API gateway that terminates inbound SSL connections. It requests data on behalf of the original requestor from any HTTP server upstream of the API 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 API gateway is highly available and resilient to failure. Avoid single points of failure by running multiple instances of your API 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 API 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;
    }
}
```

## Related guidance

- [Backends for Frontends pattern](#)
- [Gateway Aggregation pattern](#)

- Gateway Routing pattern

# Gateway Routing pattern

6/11/2018 • 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

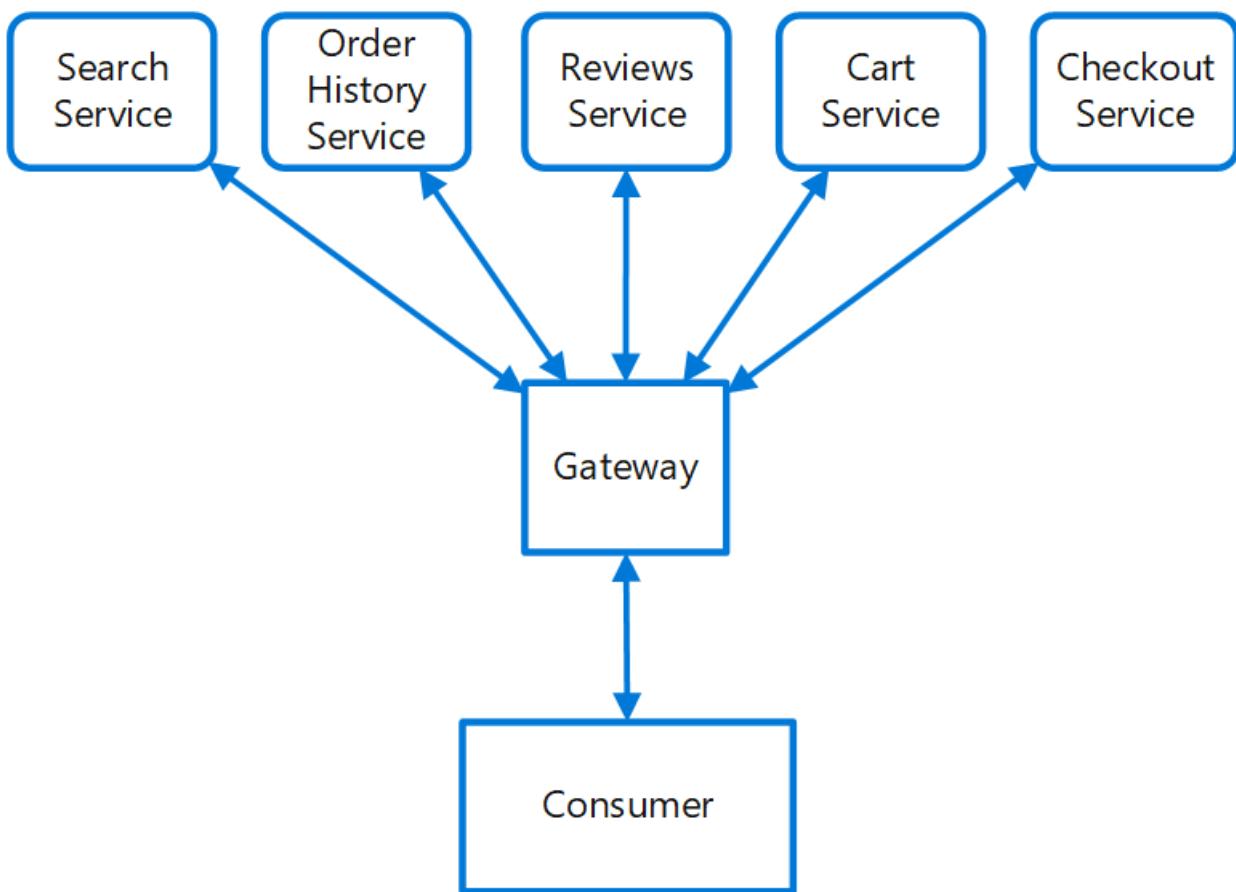
When a client needs to consume multiple services, setting up a separate endpoint for each service and having the client manage each endpoint can be challenging. For example, an e-commerce application might provide services such as search, reviews, cart, checkout, and order history. Each service has a 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;
    }
}
```

## Related guidance

- [Backends for Frontends pattern](#)
- [Gateway Aggregation pattern](#)
- [Gateway Offloading pattern](#)

# Health Endpoint Monitoring pattern

9/28/2018 • 13 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 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).

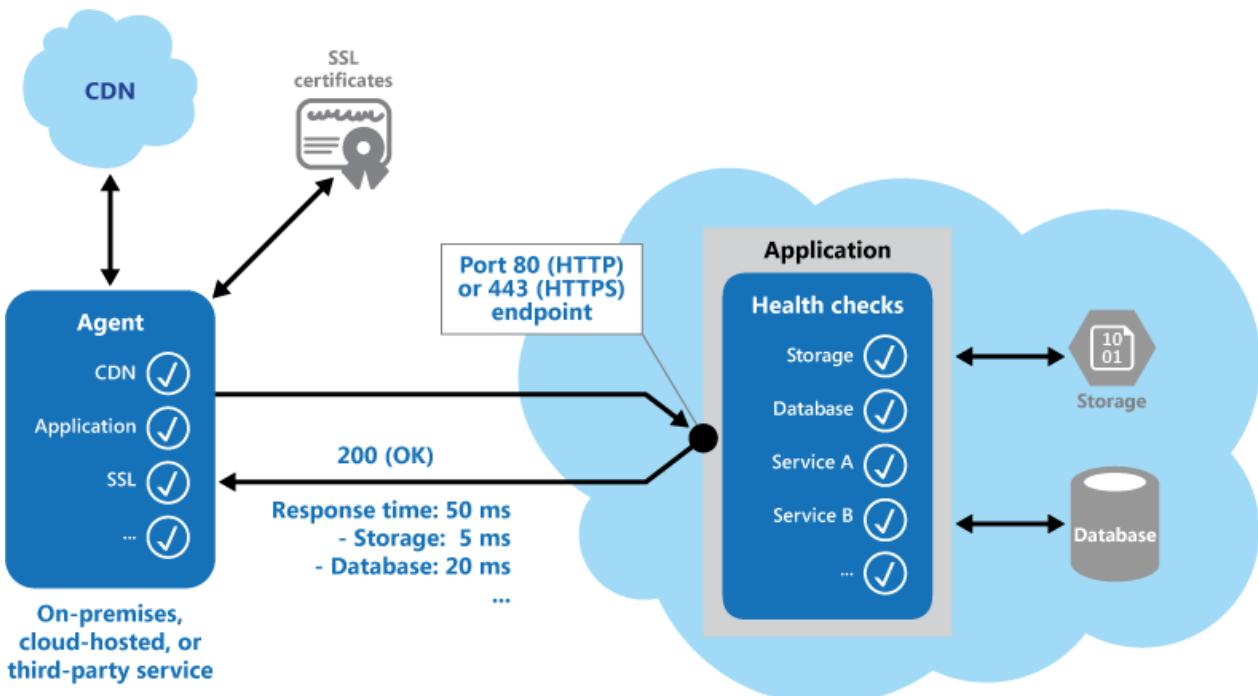
## Solution

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.

Make sure that the application correctly returns a 200 (OK) only when the target resource is found and processed. In some scenarios, such as when using a master page to host the target web page, the server sends back a 200 (OK) status code instead of a 404 (Not Found) code, even when the target content page was not found.

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, /HealthCheck/{GUID}/ 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 from 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. Not all tools and frameworks can be configured to include credentials with the health verification request. For example, Microsoft Azure built-in health verification features can't provide authentication credentials. Some third-party alternatives are [Pingdom](#), [Panopta](#), [NewRelic](#), and [Statuscake](#).
- 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

The following code examples, taken from the `HealthCheckController` class (a sample that demonstrates this pattern is available on [GitHub](#)), demonstrates exposing an endpoint for performing a range of health checks.

The `CoreServices` method, shown below in C#, performs a series of checks on services used in the application. If all of the tests run without error, the method returns a 200 (OK) status code. If any of the tests raises an exception, the method returns a 500 (Internal Error) status code. The method could optionally return additional information when an error occurs, if the monitoring tool or framework is able to make use of it.

```

public ActionResult CoreServices()
{
    try
    {
        // Run a simple check to ensure the database is available.
        DataStore.Instance.CoreHealthCheck();

        // Run a simple check on our external service.
        MyExternalService.Instance.CoreHealthCheck();
    }
    catch (Exception ex)
    {
        Trace.TraceError("Exception in basic health check: {0}", ex.Message);

        // This can optionally return different status codes based on the exception.
        // Optionally it could return more details about the exception.
        // The additional information could be used by administrators who access the
        // endpoint with a browser, or using a ping utility that can display the
        // additional information.
        return new HttpStatusCodeResult((int) HttpStatusCode.InternalServerError);
    }
    return new HttpStatusCodeResult((int) HttpStatusCode.OK);
}

```

The `ObscurePath` method shows how you can read a path from the application configuration and use it as the endpoint for tests. This example, in C#, also shows how you can accept an ID as a parameter and use it to check for valid requests.

```

public ActionResult ObscurePath(string id)
{
    // The id could be used as a simple way to obscure or hide the endpoint.
    // The id to match could be retrieved from configuration and, if matched,
    // perform a specific set of tests and return the result. If not matched it
    // could return a 404 (Not Found) status.

    // The obscure path can be set through configuration to hide the endpoint.
    var hiddenPathKey = CloudConfigurationManager.GetSetting("Test.ObscurePath");

    // If the value passed does not match that in configuration, return 404 (Not Found).
    if (!string.Equals(id, hiddenPathKey))
    {
        return new HttpStatusCodeResult((int) HttpStatusCode.NotFound);
    }

    // Else continue and run the tests...
    // Return results from the core services test.
    return this.CoreServices();
}

```

The `TestResponseFromConfig` method shows how you can expose an endpoint that performs a check for a specified configuration setting value.

```

public ActionResult TestResponseFromConfig()
{
    // Health check that returns a response code set in configuration for testing.
    var returnStatusCodeSetting = CloudConfigurationManager.GetSetting(
        "Test.ReturnStatusCode");

    int returnStatusCode;

    if (!int.TryParse(returnStatusCodeSetting, out returnStatusCode))
    {
        returnStatusCode = (int) HttpStatusCode.OK;
    }

    return new HttpStatusCodeResult(returnStatusCode);
}

```

## 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. Azure Management Services provides a built-in monitoring mechanism for alert rules. The alerts section of the management services page in the Azure portal allows you to configure up to ten alert rules per subscription for your services. These rules specify a condition and a threshold value for a service such as CPU load, or the number of requests or errors per second, and the service can automatically send email notifications to addresses you define in each rule.

The conditions you can monitor vary depending on the hosting mechanism you choose for your application (such as Web Sites, Cloud Services, Virtual Machines, or Mobile Services), 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.](#)

If you host your application in Azure Cloud Services web and worker roles or Virtual Machines, you can take advantage of one of the built-in services in Azure called Traffic Manager. Traffic Manager is a routing and load-balancing service that can distribute requests to specific instances of your Cloud Services hosted 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 ten seconds 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.

Read more information about using [Traffic Manager to monitor your applications](#). Traffic Manager is also discussed in [Multiple Datacenter Deployment Guidance](#).

## Related guidance

The following guidance can be useful when implementing this pattern:

- [Instrumentation and Telemetry Guidance](#). Checking the health of services and components is typically done by probing, but it's also useful to have information in place to monitor application performance and detect events that occur at runtime. This data can be transmitted back to monitoring tools as additional information for health monitoring. Instrumentation and Telemetry Guidance explores gathering remote diagnostics information that's collected by instrumentation in applications.
- [Receiving alert notifications](#).
- This pattern includes a downloadable [sample application](#).

# Index Table pattern

8/14/2017 • 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	Last Name: Smith, Town: Redmond, ...
2	Last Name: Jones, Town: Seattle, ...
3	Last Name: Robinson, Town: Portland, ...
4	Last Name: Brown, Town: Redmond, ...
5	Last Name: Smith, Town: Chicago, ...
6	Last Name: Green, Town: Redmond, ...
7	Last Name: Clarke, Town: Portland, ...
8	Last Name: Smith, Town: Redmond, ...
9	Last Name: Jones, Town: Chicago, ...
...	...
1000	Last Name: 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.

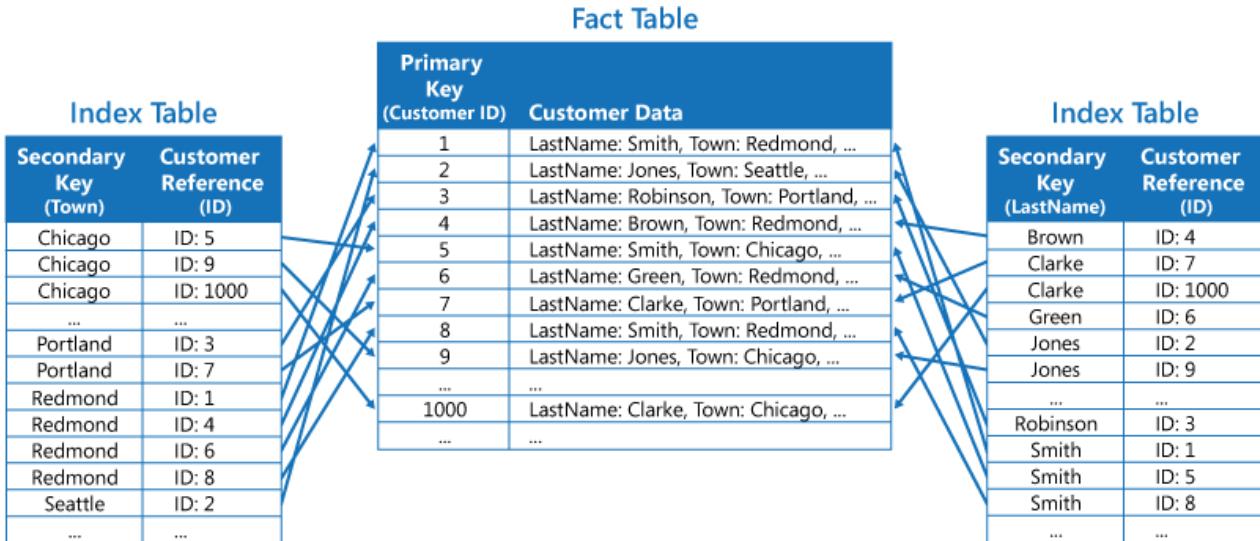
The first strategy is to duplicate the data in each index table but organize it by different keys (complete denormalization). The next figure shows index tables that organize the same customer information by Town and LastName.

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, ...
...	...

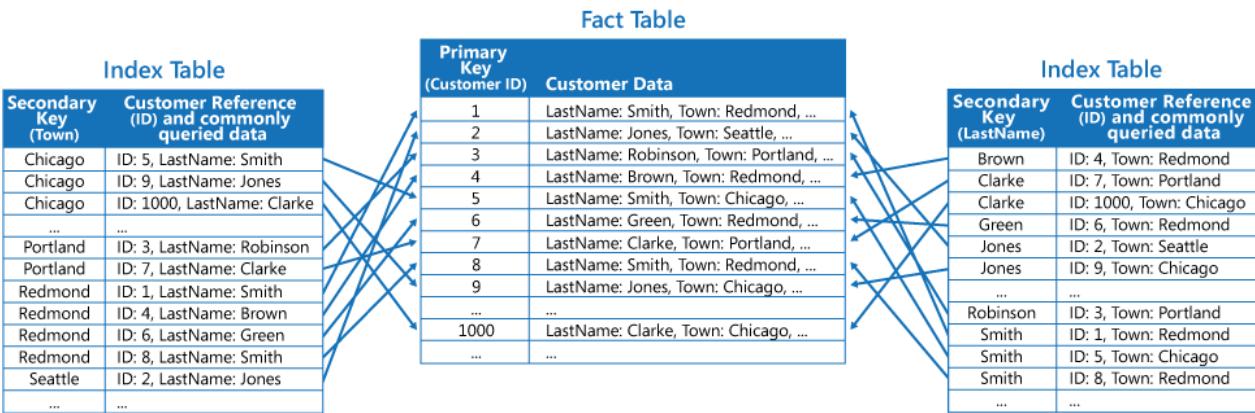
This strategy is appropriate if the data is relatively static compared to the number of times it's queried using each key. If the data is more dynamic, the processing overhead of maintaining each index table becomes too large for this approach to be useful. Also, if the volume of data is very large, 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.



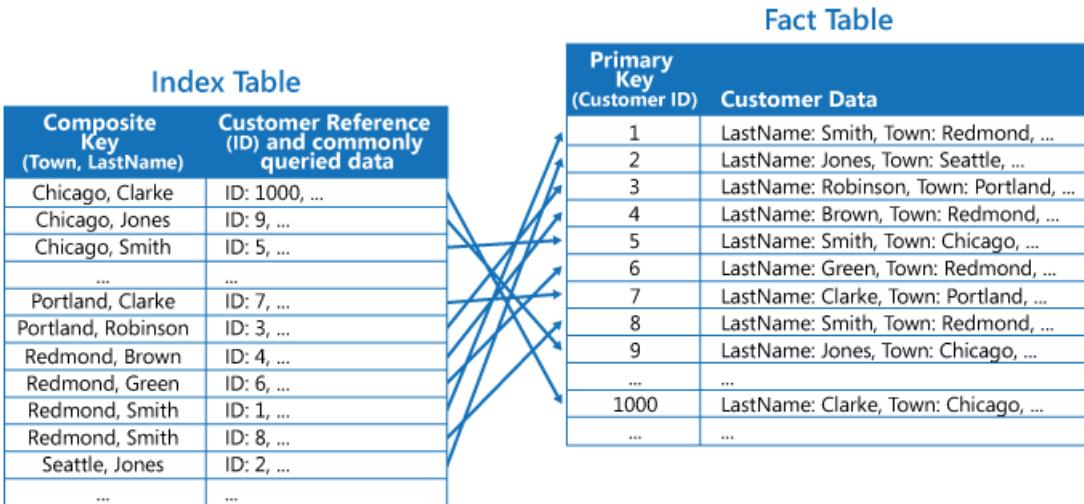
This technique saves space and reduces the overhead of maintaining duplicate data. The disadvantage is that an application has to perform two lookup operations to find data using a secondary key. It has to find the primary key for the data in the index table, and then use the primary key to look up the 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.

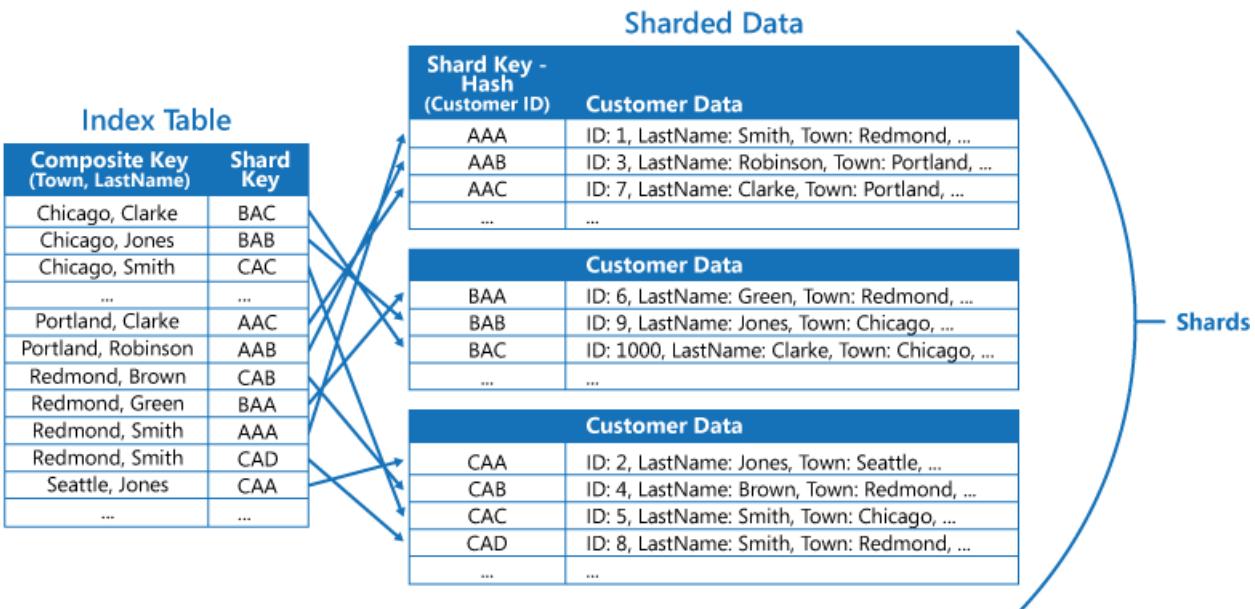


With this strategy, you can strike a balance between the first two approaches. The data for common queries can be retrieved quickly by using a single lookup, while the space and maintenance overhead isn't as significant as duplicating the entire data set.

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.

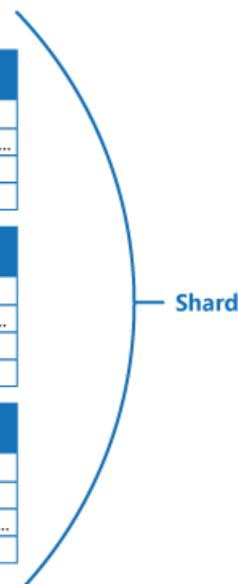
- A field selected as the secondary key for an index table is nondiscriminating and can only have a small set of values (for example, gender).
- The balance of the data values for a field selected as the secondary key for an index table are highly skewed. For example, if 90% of the records contain the 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.



The diagram illustrates the sharding of movie data across three separate Azure storage tables, each representing a different genre. A large blue bracket on the right side of the tables is labeled "Shards", indicating that each table is a shard within the overall system.

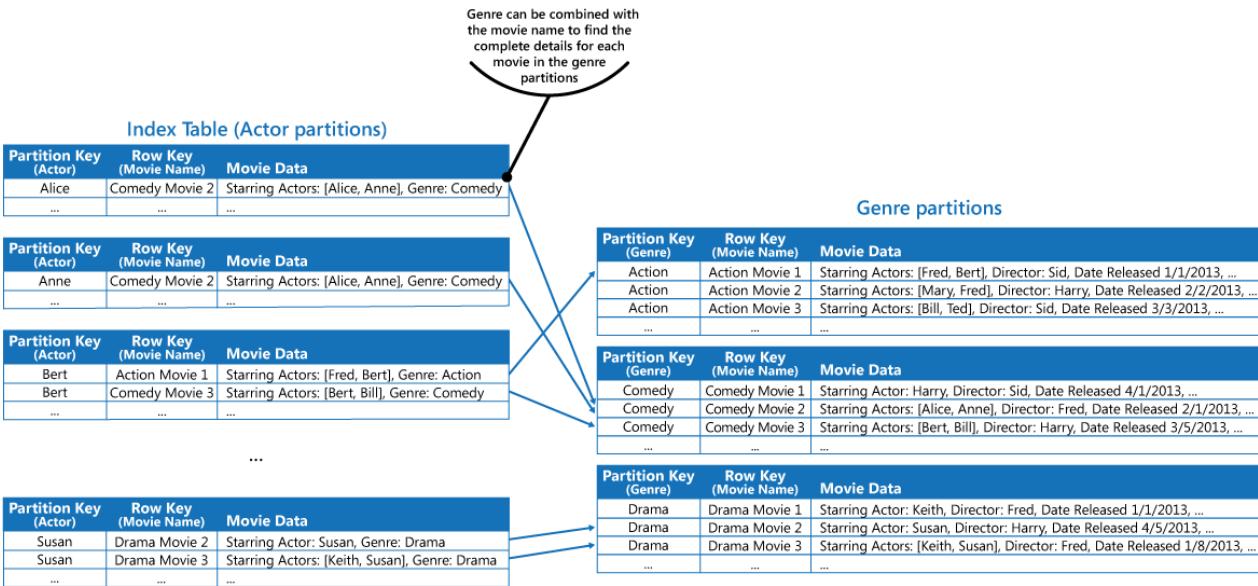
Partition Key (Genre)	Row Key (Movie Name)	Movie Data
Action	Action Movie 1	Starring Actors: [Fred, Bert], Director: Sid, Date Released 1/1/2013, ...
Action	Action Movie 2	Starring Actors: [Mary, Fred], Director: Harry, Date Released 2/2/2013, ...
Action	Action Movie 3	Starring Actors: [Bill, Ted], Director: 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, ...
...	...	...

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 patterns and 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

9/28/2018 • 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 `RunTaskWhenMutexAquired` 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> taskToRunWhenLeaseAquired)
    {
        this.blobSettings = blobSettings;
        this.taskToRunWhenLeaseAquired = taskToRunWhenLeaseAquired;
    }

    public async Task RunTaskWhenMutexAcquired(CancellationToken token)
    {
        var leaseManager = new BlobLeaseManager(blobSettings);
        await this.RunTaskWhenBlobLeaseAcquired(leaseManager, token);
    }
    ...
}
```

The `RunTaskWhenMutexAquired` 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.TryAquireLeaseOrWait(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.taskToRunWhenLeaseAquired.Invoke(leaseCts.Token);
                ...
            }
        }
        ...
    }
}

```

The task started by the leader also runs asynchronously. While this task is running, the `RunTaskWhenBlobLeaseAquired` 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 (...)

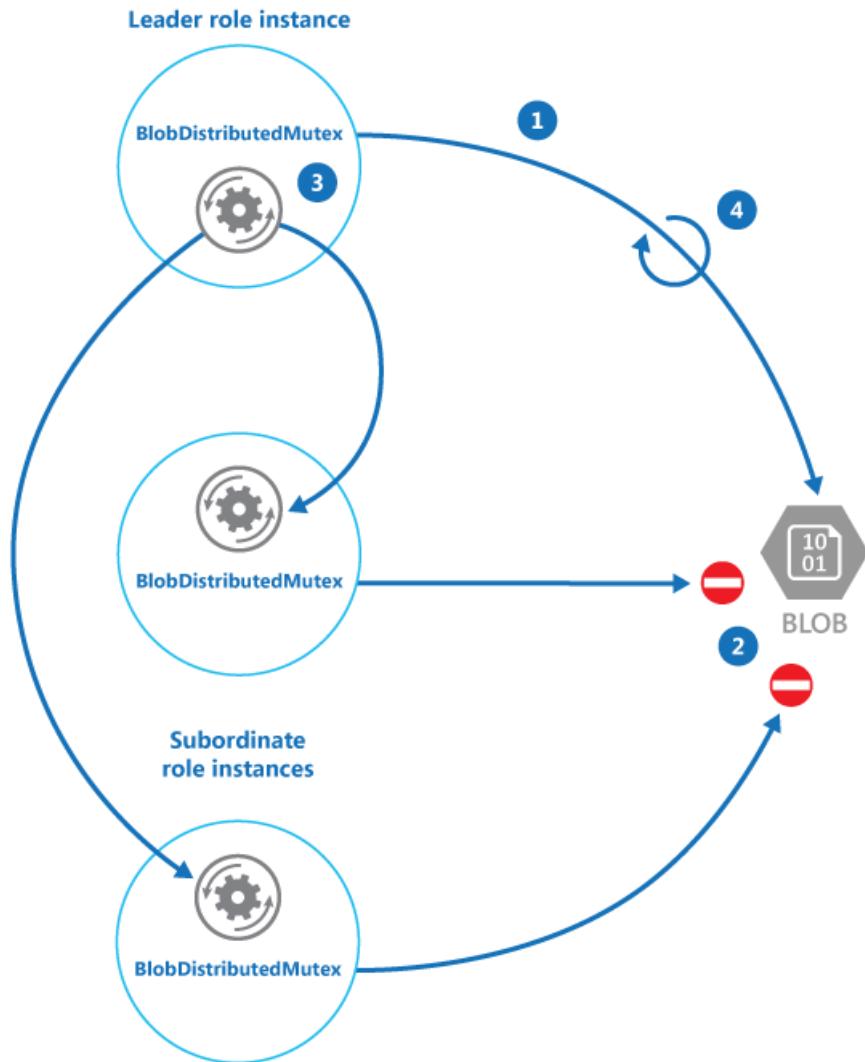
    ...
    if (...)

    ...
    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 `BlobDistributedMutex` 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.

## Related patterns and guidance

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

8/14/2017 • 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.

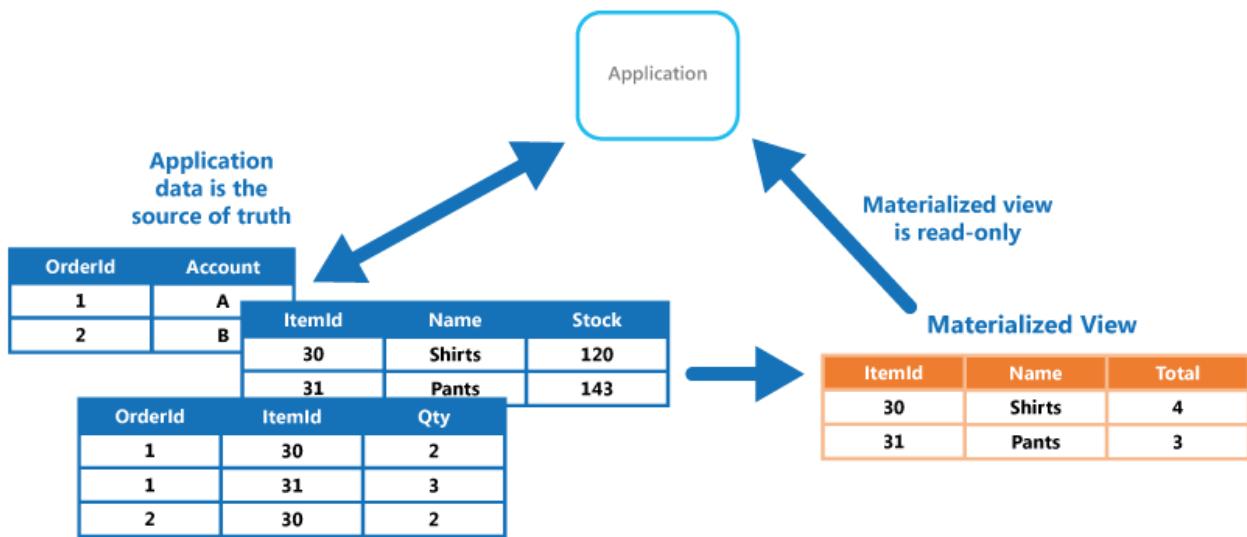
## Solution

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.

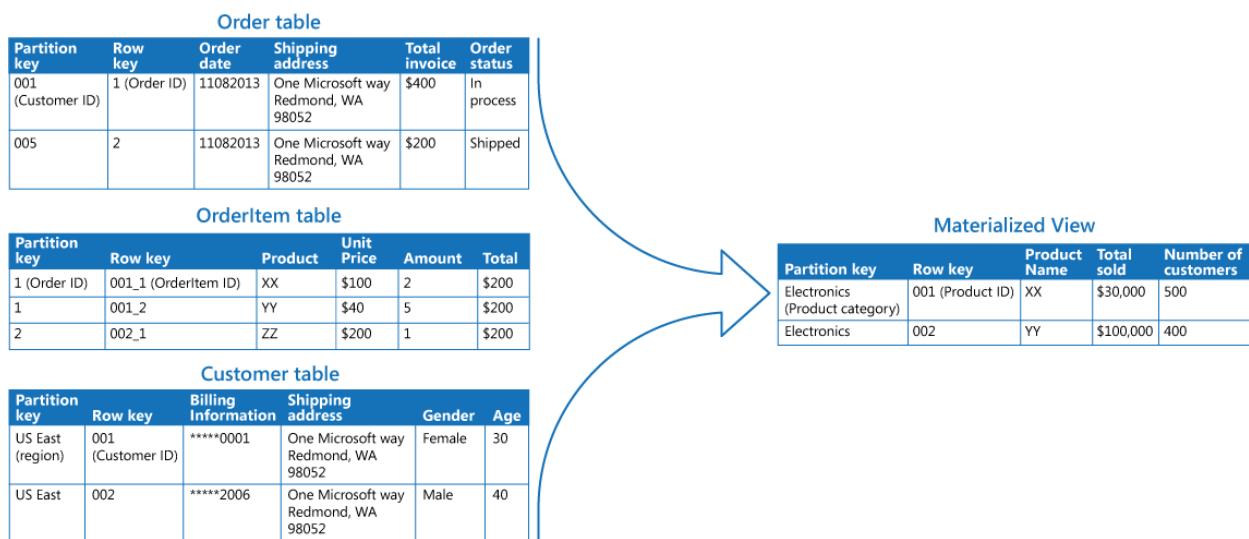
- Supporting occasionally connected or disconnected scenarios where connection to the data store isn't always available. The view can be cached locally in this case.
- Simplifying queries and exposing data for experimentation in a way that doesn't require knowledge of the source data format. For example, 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.

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 utilizes Azure table storage, many relational database management systems also provide native support for materialized views.

## Related patterns and 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

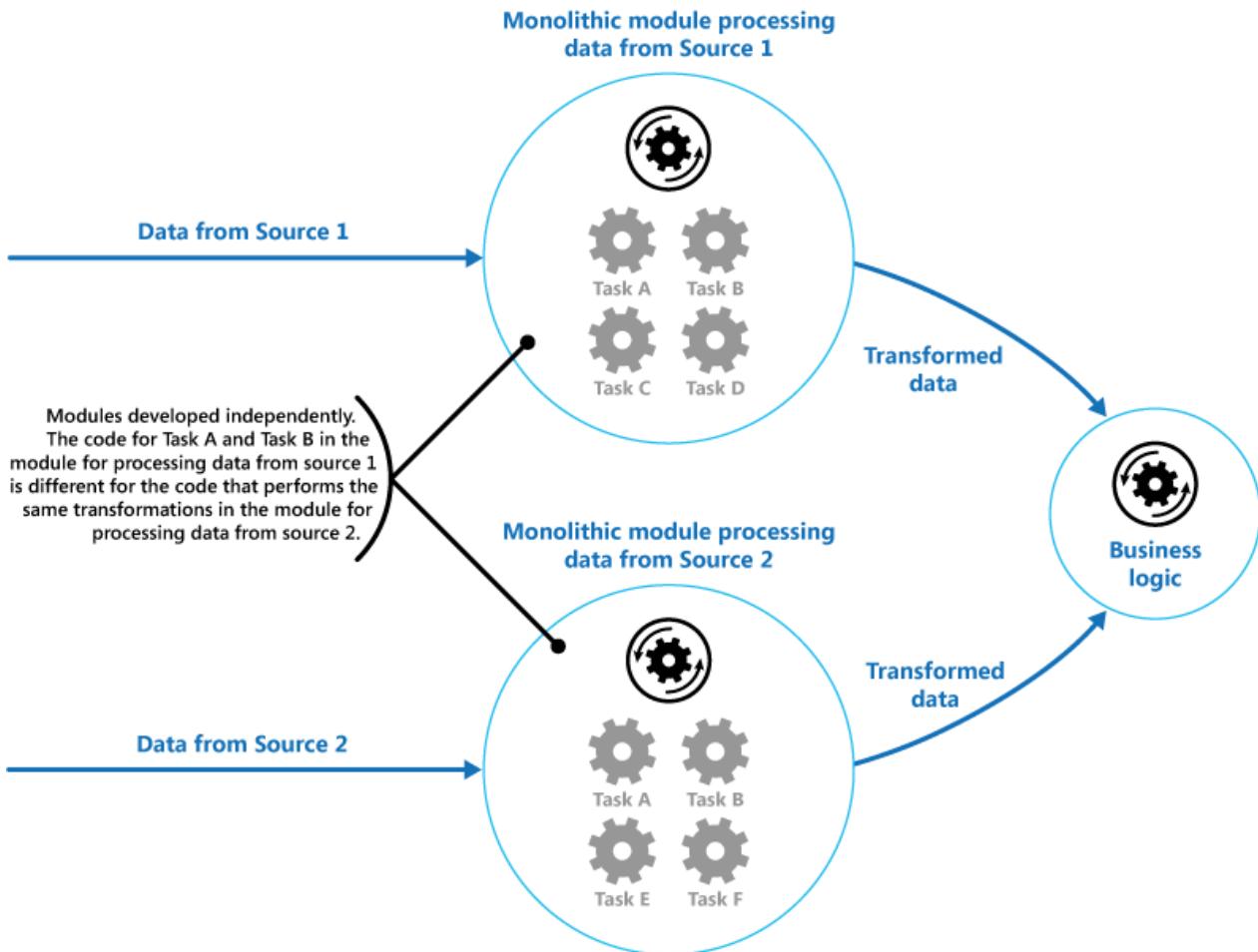
9/28/2018 • 12 minutes to read • [Edit Online](#)

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.



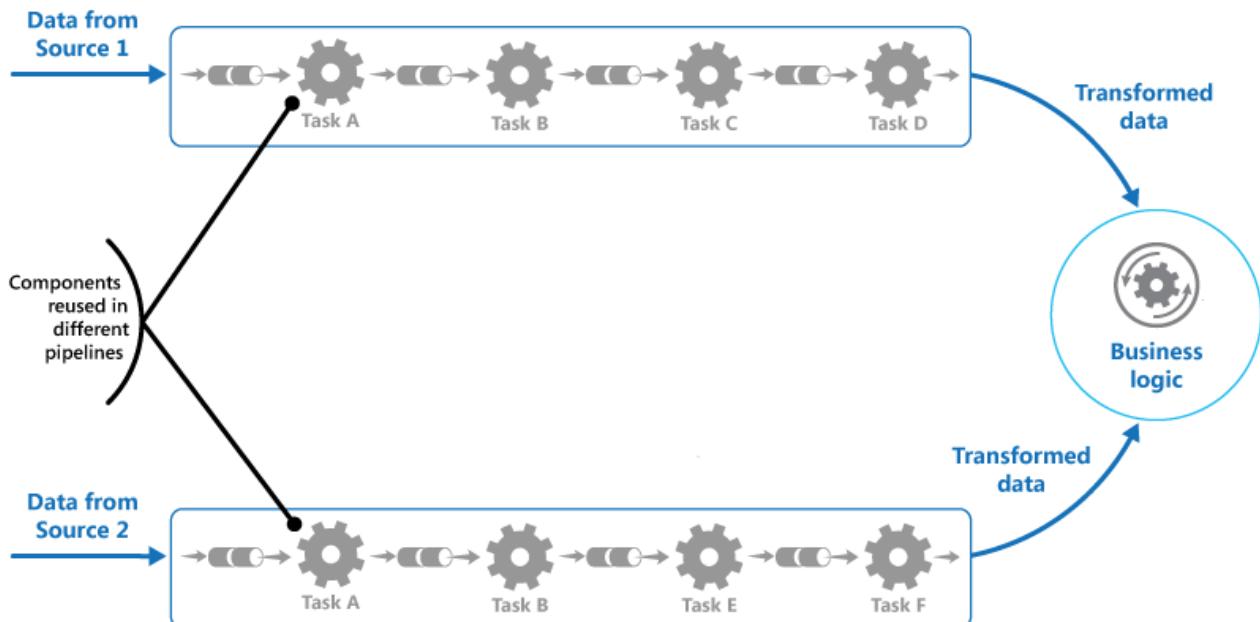
Some of the tasks that the monolithic modules perform are functionally very similar, but the modules have been designed separately. The code that implements the tasks is closely coupled in a module, and has been developed with little or no thought given to reuse or scalability.

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.

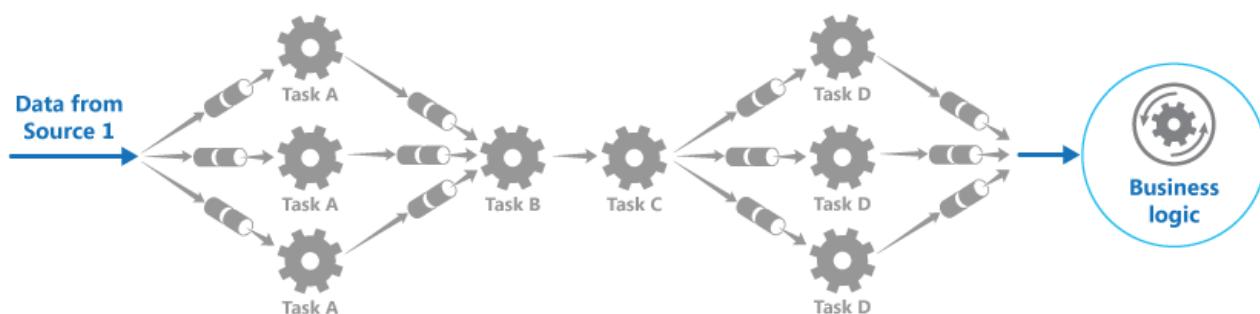
## Solution

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 data center or geographical location, which allows each element in a pipeline to run in an environment that is 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.Information(
                    "Pipeline Message Complete - FilterA:{0} FilterB:{1}",
                    msg.Properties[Constants.FilterAMessageKey],
                    msg.Properties[Constants.FilterBMessageKey]);

                return null;
            });
        ...
    }
    ...
}
```

## Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- A sample that demonstrates this pattern is available on [GitHub](#).
- [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

9/28/2018 • 9 minutes to read • [Edit Online](#)

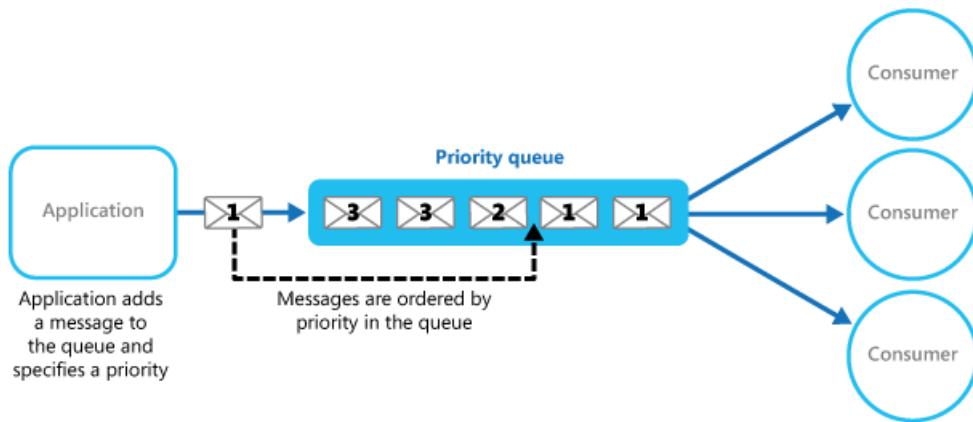
Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority. This pattern is useful in applications that offer different service level guarantees to individual clients.

## 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.

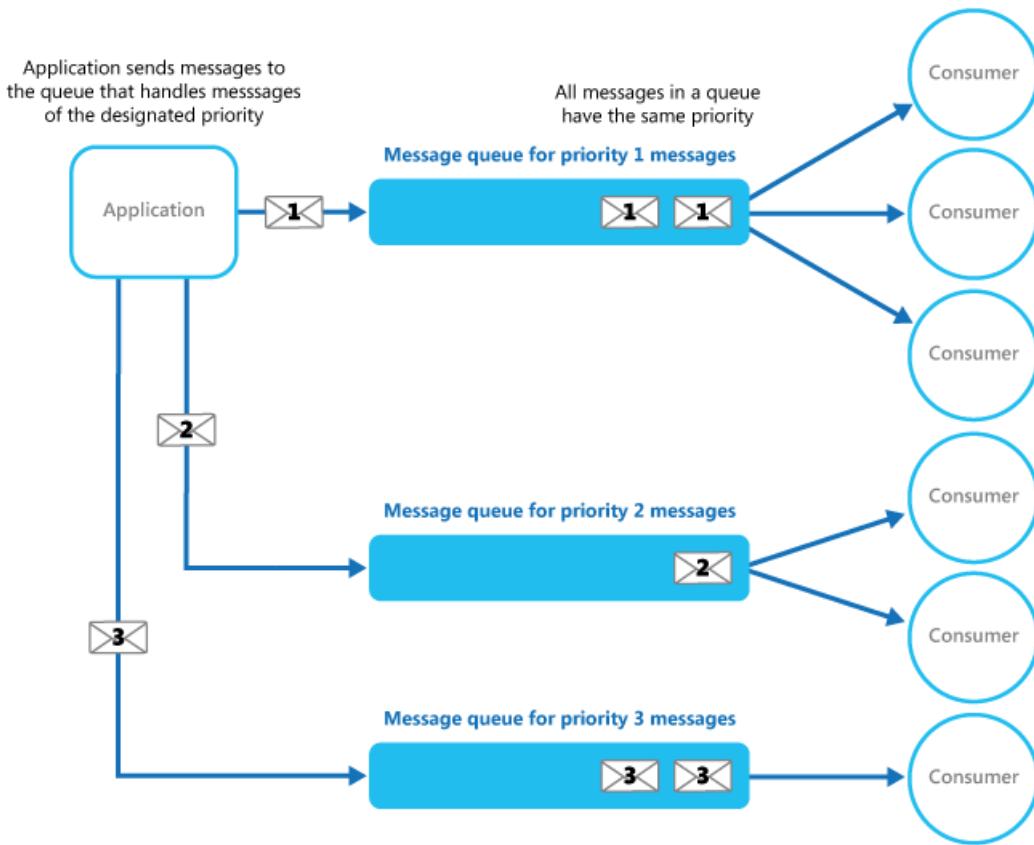
## Solution

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

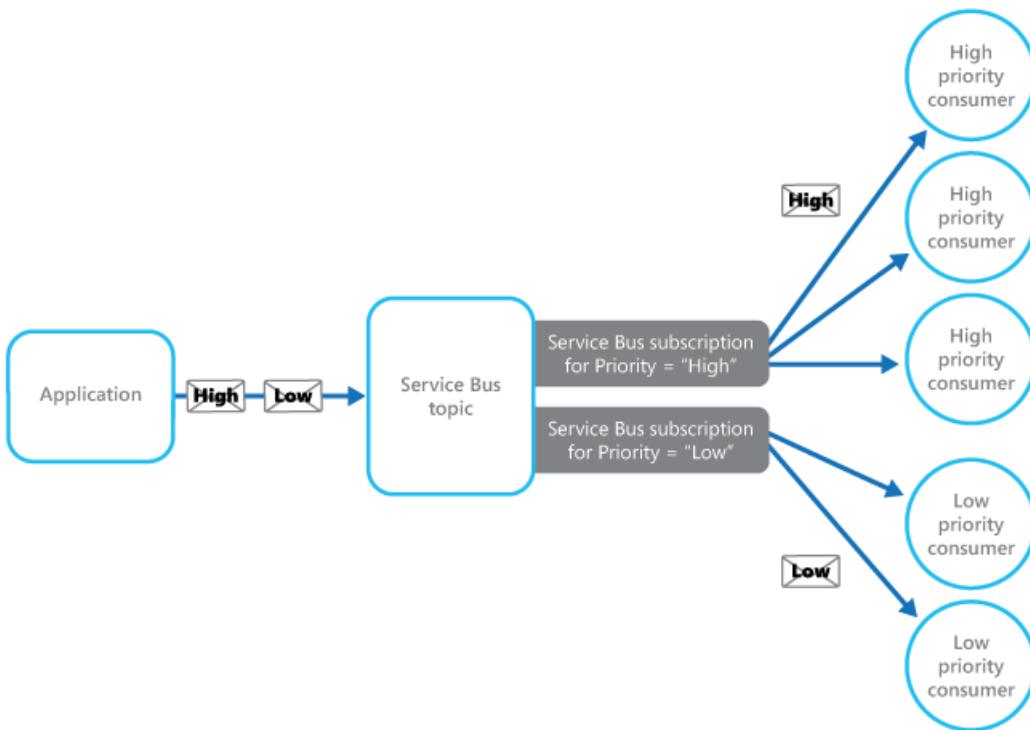
This pattern is useful in scenarios where:

- 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();

```

## Related patterns and guidance

The following patterns and 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.
- [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.
- [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.
- [Enterprise Integration Patterns with Service Bus](#) on Abhishek Lal's blog.

# Queue-Based Load Leveling pattern

12/7/2017 • 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

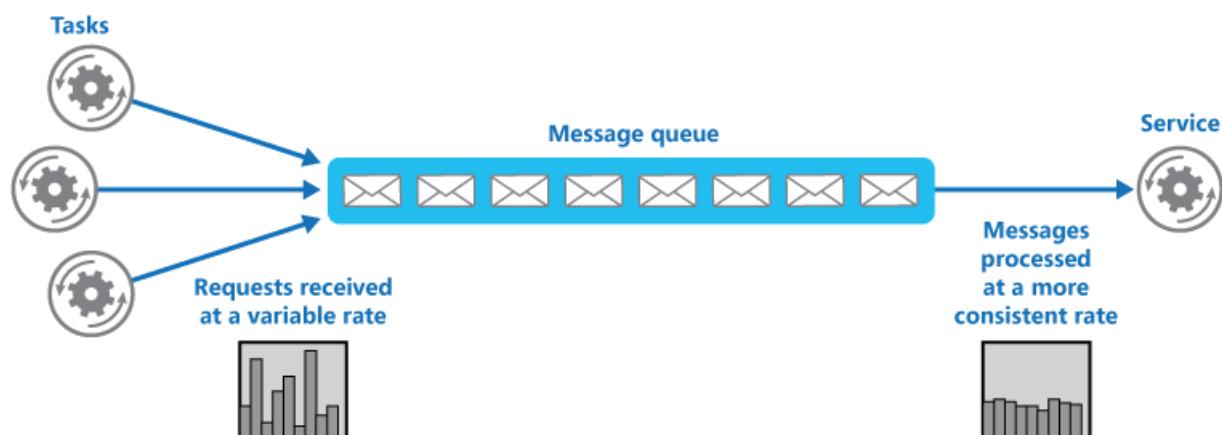
Many solutions in the cloud involve running tasks that invoke services. In this environment, if a service is subjected to intermittent heavy loads, it can cause performance or reliability issues.

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.

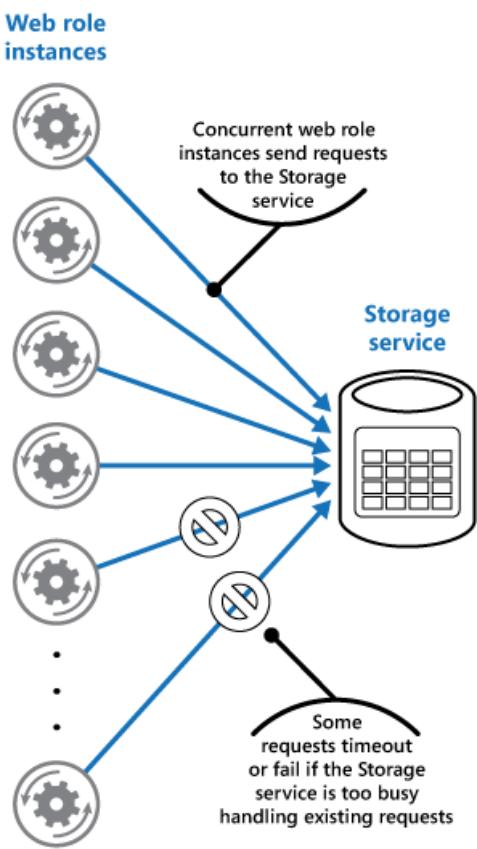
## When to use this pattern

This pattern is useful to any application that uses services that are subject to overloading.

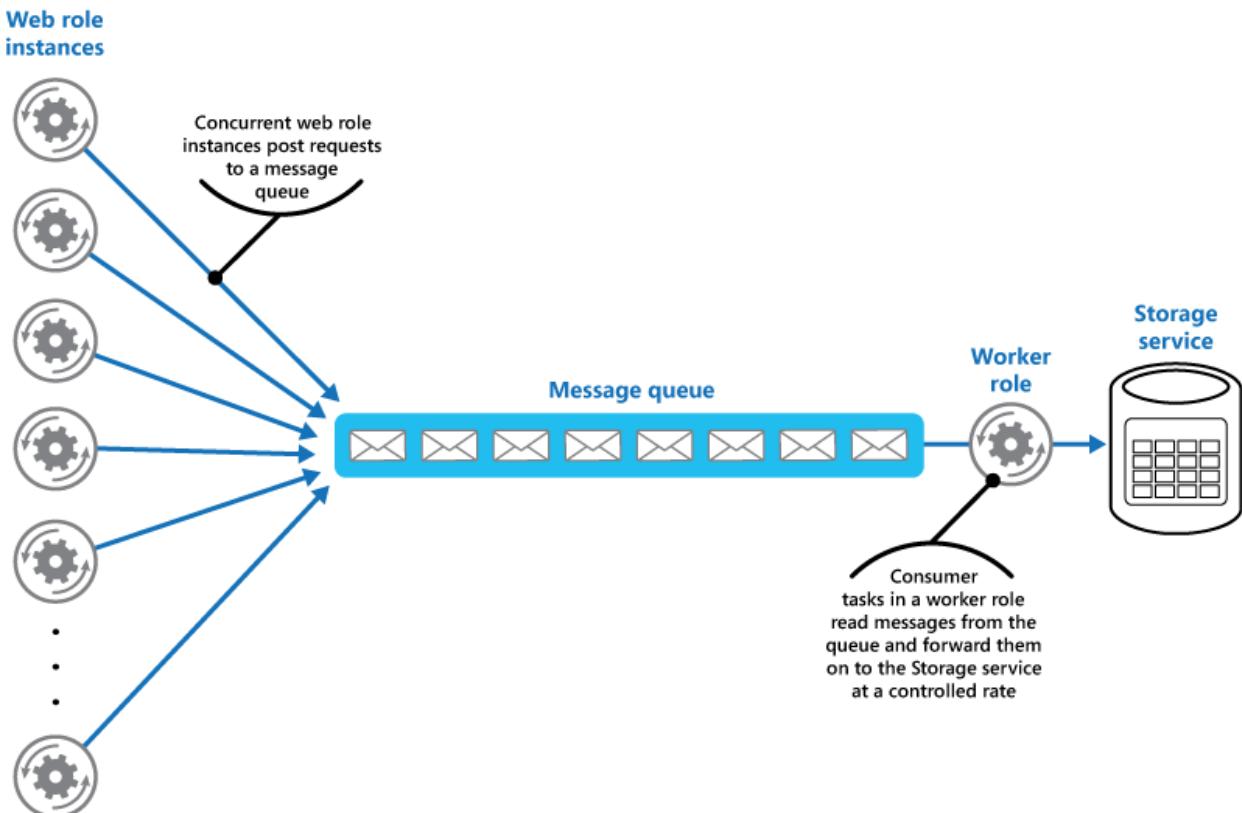
This pattern isn't useful if the application expects a response from the service with minimal latency.

## Example

A Microsoft Azure web role stores data using a separate storage service. If a large number of instances of the web role run concurrently, it's possible that the storage service will be unable to respond to requests quickly enough to prevent these requests from timing out or failing. This figure highlights a service being overwhelmed by a large number of concurrent requests from instances of a web role.



To resolve this, you can use a queue to level the load between the web role instances and the storage service. However, the storage service is designed to accept synchronous requests and can't be easily modified to read messages and manage throughput. You can introduce a worker role to act as a proxy service that receives requests from the queue and forwards them to the storage service. The application logic in the worker role can control the rate at which it passes requests to the storage service to prevent the storage service from being overwhelmed. This figure illustrates using a queue and a worker role to level the load between instances of the web role and the service.



## Related patterns and guidance

The following patterns and 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.
- [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.
- [Queue Service Concepts](#). Information about choosing a messaging and queuing mechanism in Azure applications.

# Retry pattern

2/22/2018 • 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.

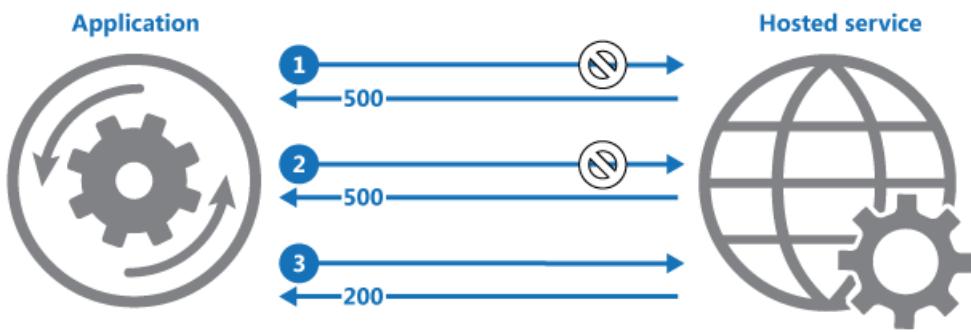
If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies:

- **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. 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 resend 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;
}

```

## Related patterns and guidance

- [Circuit Breaker pattern](#). The Retry pattern is useful for handling transient faults. If a failure is expected to be more long lasting, it might be more appropriate to implement the Circuit Breaker pattern. The Retry pattern can also be used in conjunction with a circuit breaker to provide a comprehensive approach to handling faults.
- [Retry guidance for specific services](#)
- [Connection Resiliency](#)

# Scheduler Agent Supervisor pattern

9/28/2018 • 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.

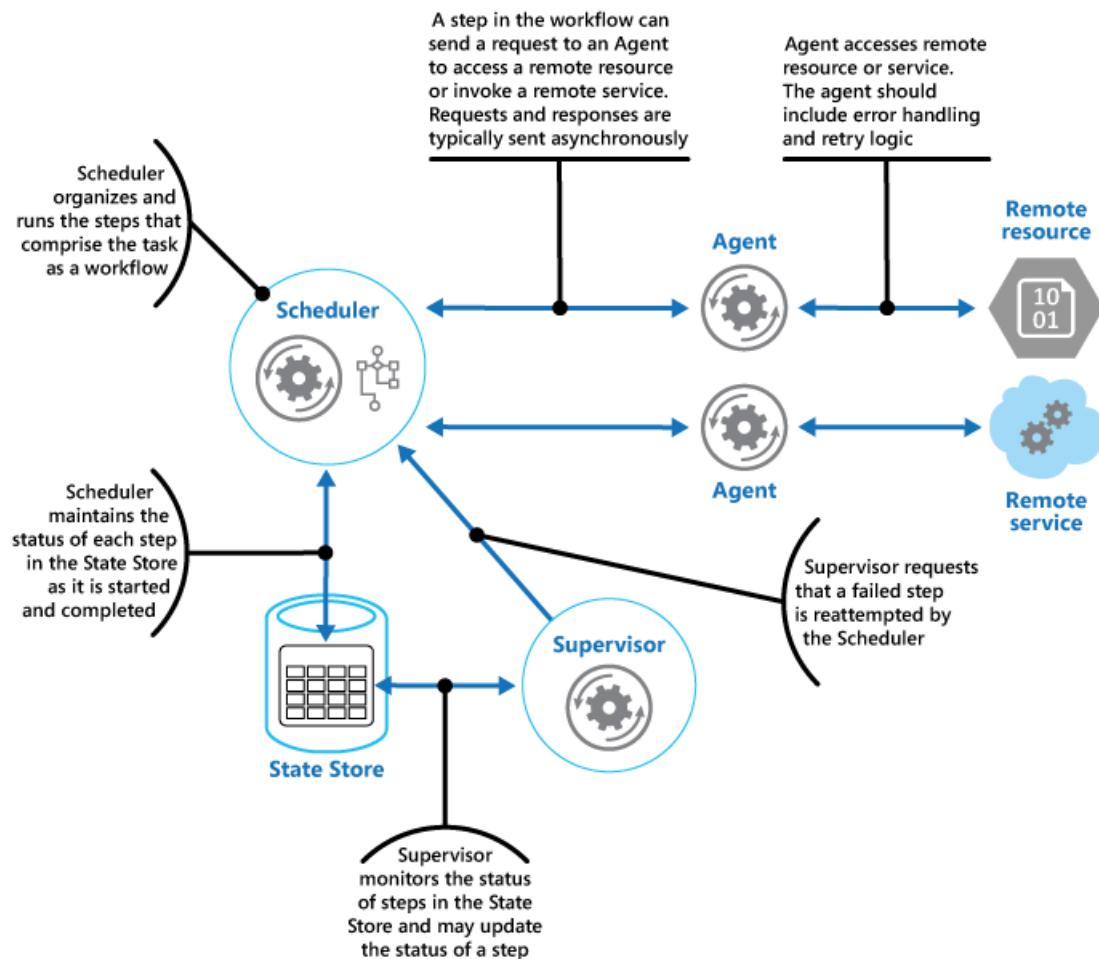
- The **Agent** contains logic that encapsulates a call to a remote service, or access to a remote resource referenced by a step in a task. Each Agent typically wraps calls to a single service or resource, implementing the appropriate error handling and retry logic (subject to a timeout constraint, described 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.



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:

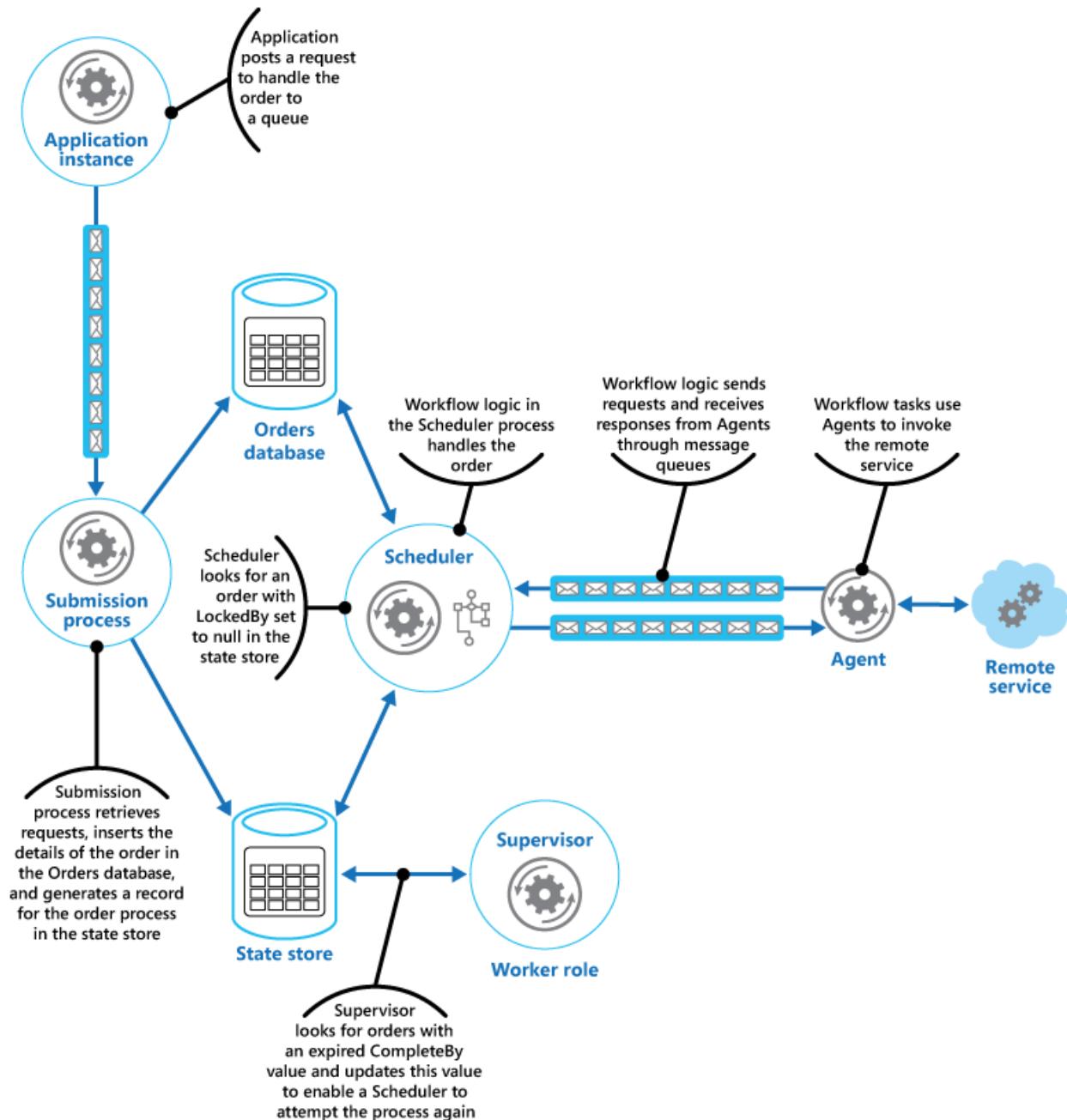
- **OrderId**. The ID of the order in the orders database.
- **LockedBy**. The instance ID of the worker role handling the order. There might be multiple current instances of the worker role running the Scheduler, but each order should only be handled by a single instance.
- **CompleteBy**. The time 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.

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.

## Related patterns and guidance

The following patterns and guidance 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.
- [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.

- [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](#)
- [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](#)

# Sharding pattern

9/28/2018 • 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.

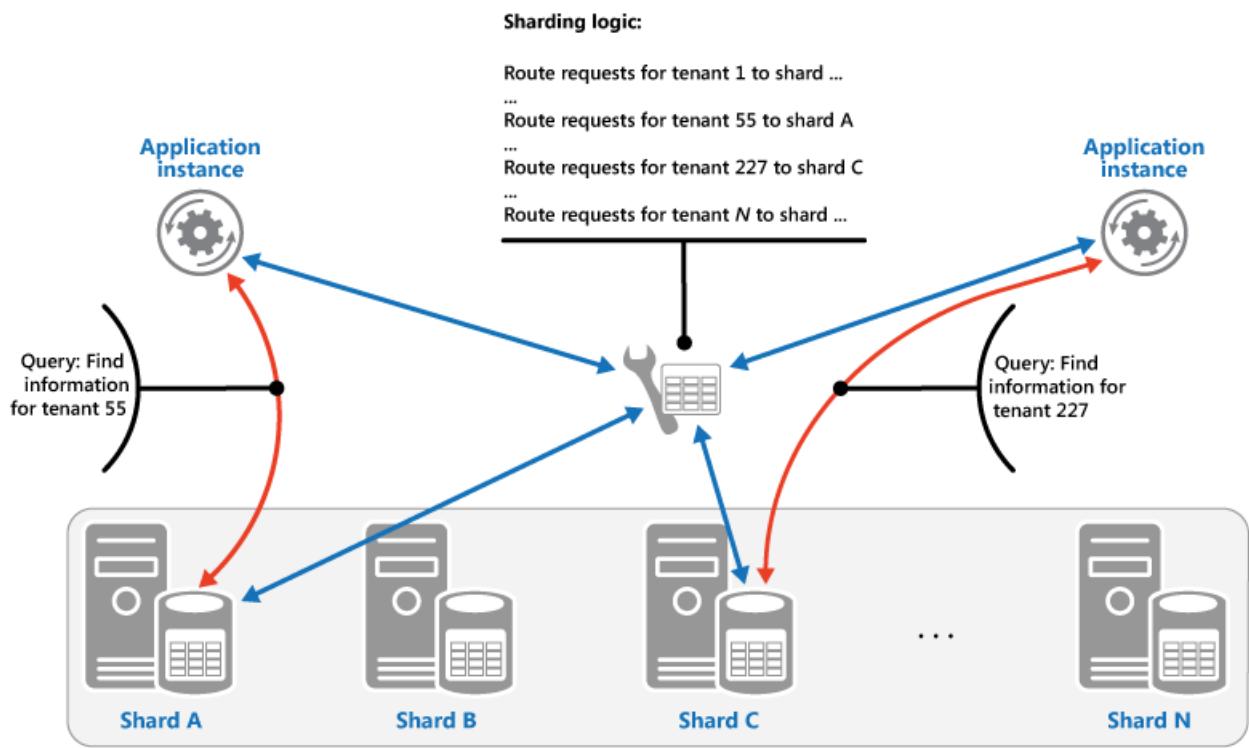
To ensure optimal performance and scalability, it's important to split the data in a way that's appropriate for the types of queries that the application performs. In many cases, it's unlikely that the sharding scheme will exactly match the requirements of every query. For example, in a multi-tenant 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

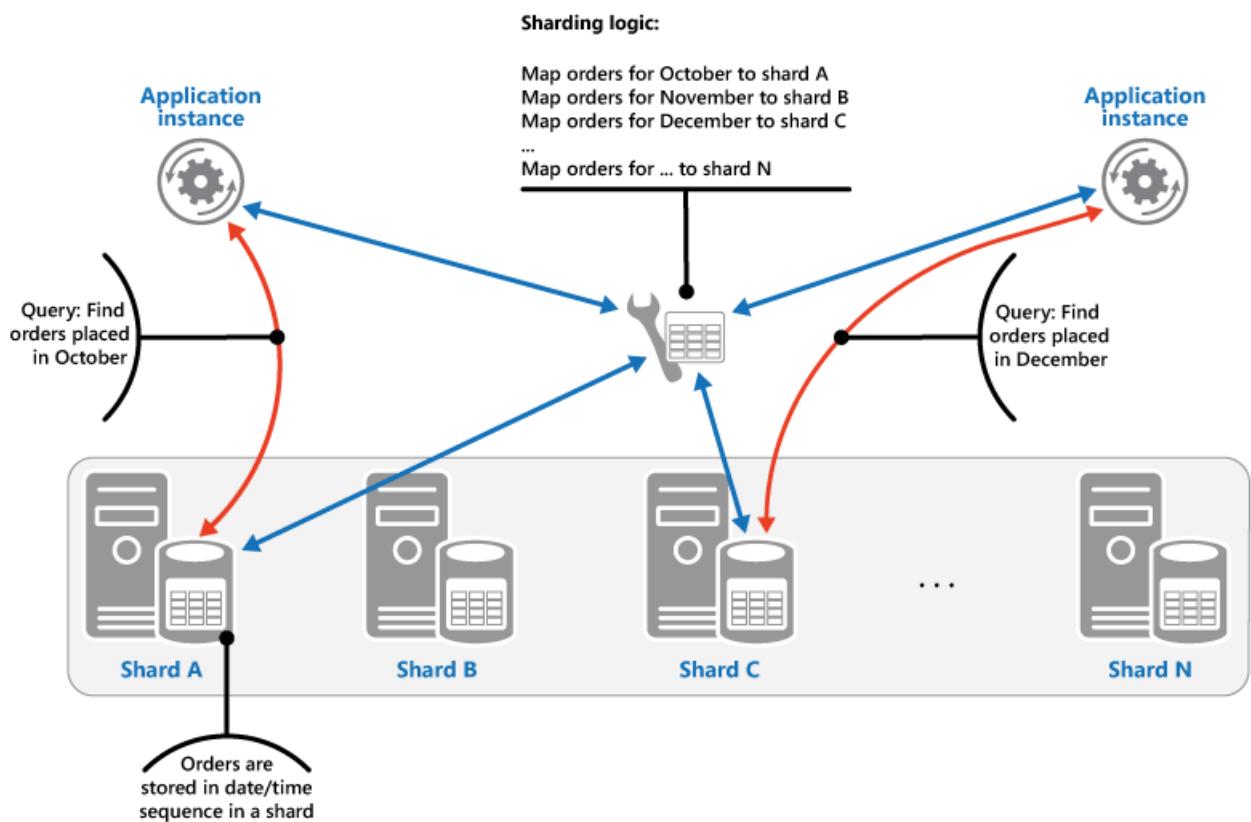
Three strategies are commonly used when selecting the shard key and deciding how to distribute data across shards. Note that there doesn't have to be a one-to-one correspondence between shards and the servers that host them—a single server can host multiple shards. The strategies are:

**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.



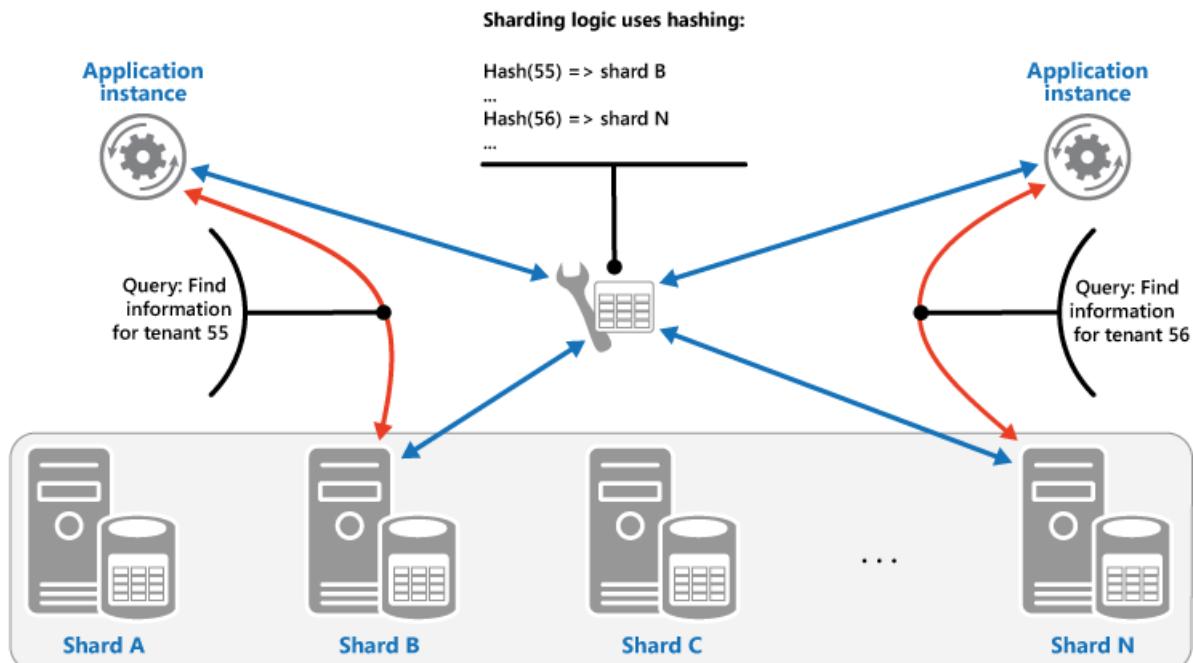
The mapping between the shard key and the physical storage can be based on physical shards where each shard key maps to a physical partition. Alternatively, a more flexible technique for rebalancing shards is virtual partitioning, where shard keys map to the same number of virtual shards, which in turn 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 to 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.

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/master 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);

```

## Related patterns and guidance

The following patterns and 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.
- [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

6/24/2017 • 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.

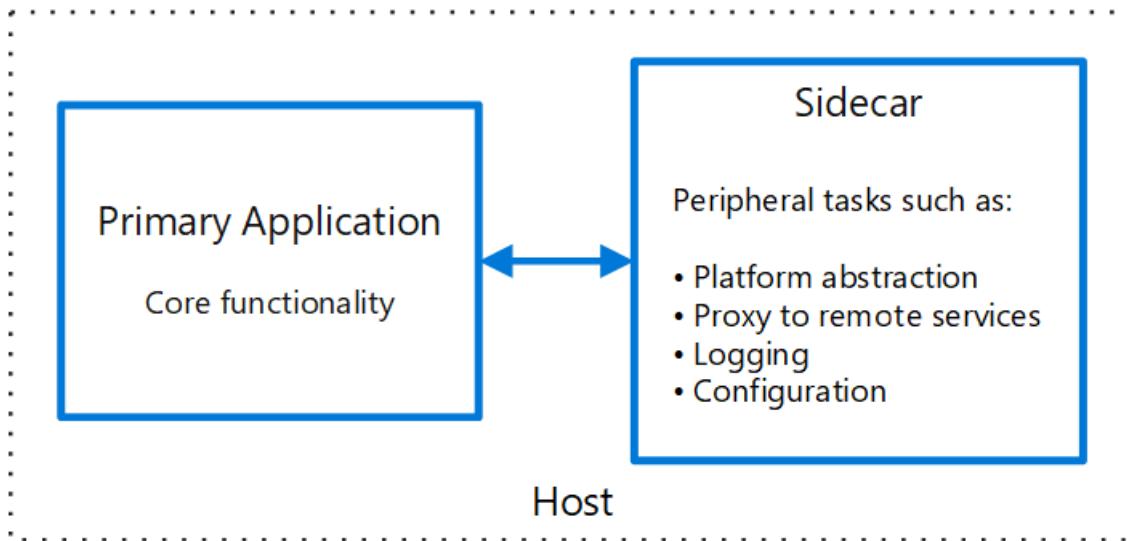
If they are tightly integrated into the application, they can run in the same process as the application, making efficient use of shared resources. However, this also means they are not well isolated, and an outage in one of these components can affect other components or the entire application. Also, 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 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 heterogenous 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

9/28/2018 • 6 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 well tuned to optimize requests through efficient dynamic page code execution 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 it's possible to minimize the need for compute instances (for example, use a smaller instance or fewer instances), by locating some of an application's resources and static pages in a storage service. 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 an application and to update it. 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

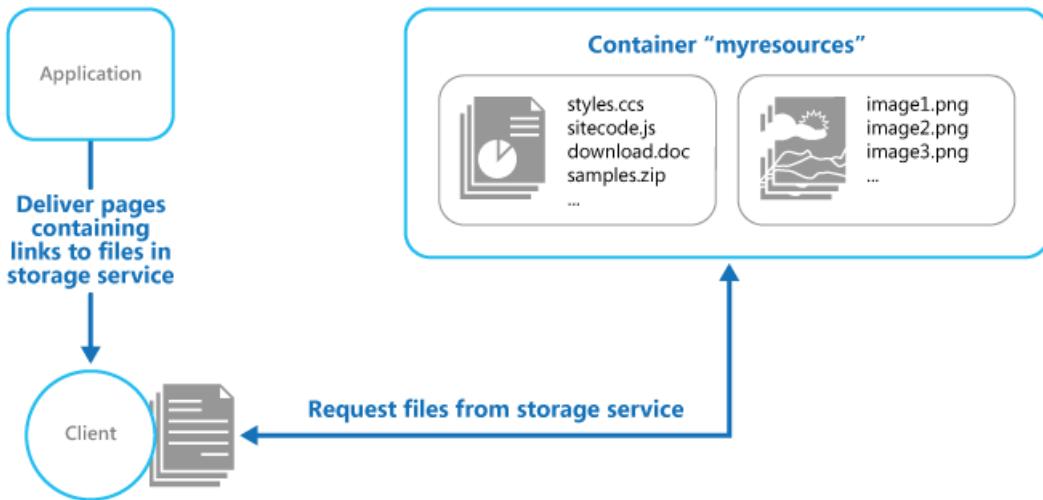
Static content located in Azure Blob storage can be accessed directly by a web browser. Azure provides an HTTP-based interface over storage that can be publicly exposed to clients. For example, content in an Azure Blob storage container is exposed using a URL with the following form:

```
https://[ storage-account-name ].blob.core.windows.net/[ container-name ]/[ file-name ]
```

When uploading the content it's necessary to create one or more blob containers to hold the files and documents. Note that the default permission for a new container is Private, and you must change this to Public to allow clients to access the contents. If it's necessary to protect the content from anonymous access, you can implement the [Valet Key pattern](#) so users must present a valid token to download the resources.

[Blob Service Concepts](#) has information about blob storage, and the ways that you can access and use it.

The links in each page will specify the URL of the resource and the client will access it directly from the storage service. The figure illustrates delivering static parts of an application directly from a storage service.



The links in the pages delivered to the client must specify the full URL of the blob container and resource. For example, a page that contains a link to an image in a public container might contain the following HTML.

```

```

If the resources are protected by using a valet key, such as an Azure shared access signature, this signature must be included in the URLs in the links.

A solution named StaticContentHosting that demonstrates using external storage for static resources is available from [GitHub](#). The StaticContentHosting.Cloud project contains configuration files that 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 account = CloudStorageAccount.Parse(StaticContentStorageConnectionString);

            return string.Format("{0}/{1}", account.BlobEndpoint.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.

```

```

## Related patterns and guidance

- A sample that demonstrates this pattern is available on [GitHub](#).

- [Valet Key pattern](#). If the target resources aren't supposed to be available to anonymous users it's necessary to implement security over the store that holds the static content. Describes how to use a token or key that provides clients with restricted direct access to a specific resource or service such as a cloud-hosted storage service.
- [Blob Service Concepts](#)

# Strangler pattern

10/24/2018 • 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.

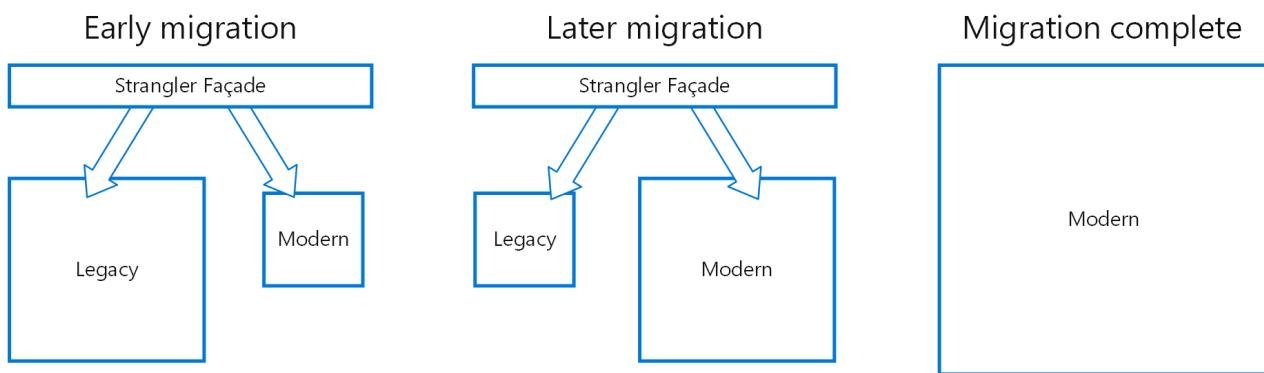
## Context and problem

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 migrations.
- At some point, when the migration is complete, the strangler 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](#)
- [Anti-Corruption Layer pattern](#)
- [Gateway Routing pattern](#)

# Throttling pattern

8/14/2017 • 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

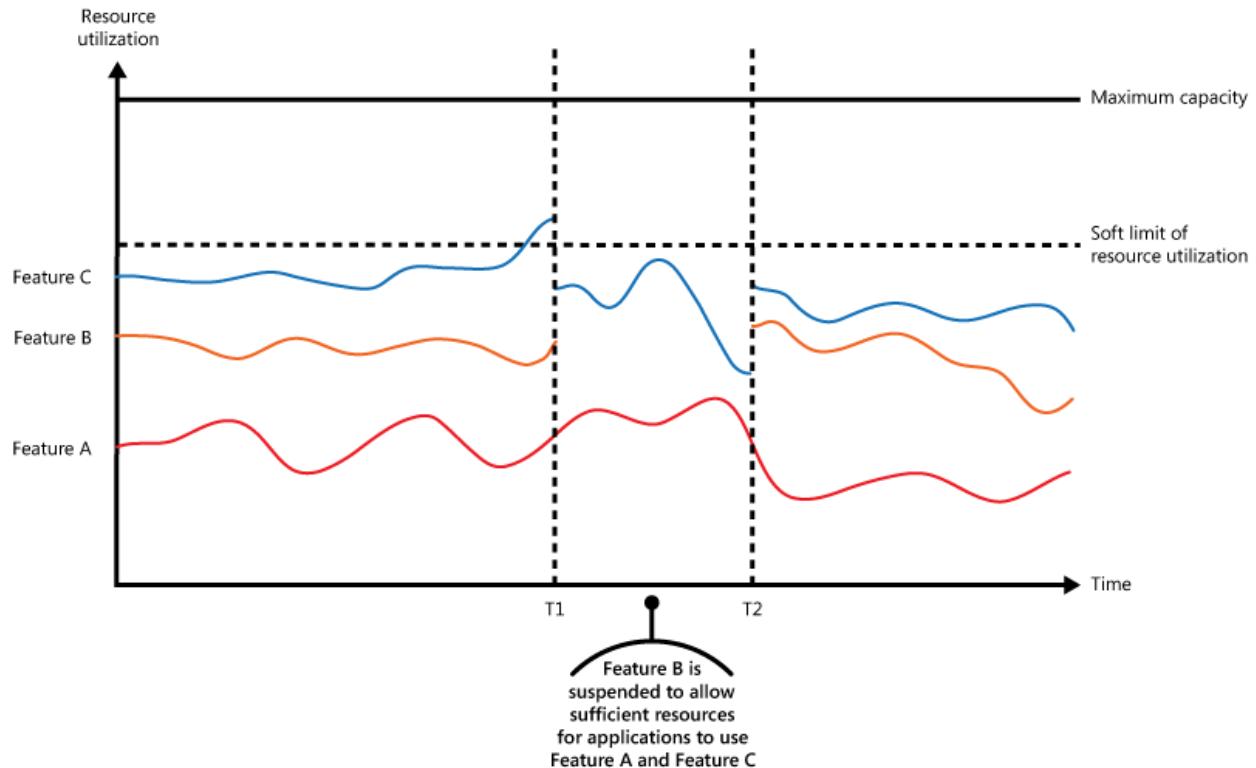
An alternative strategy to autoscaling is to allow applications to use resources only up to a limit, and then throttle them when this limit is reached. The system should monitor how it's using resources so that, when usage exceeds the threshold, it can throttle requests from one or more users. This will 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.

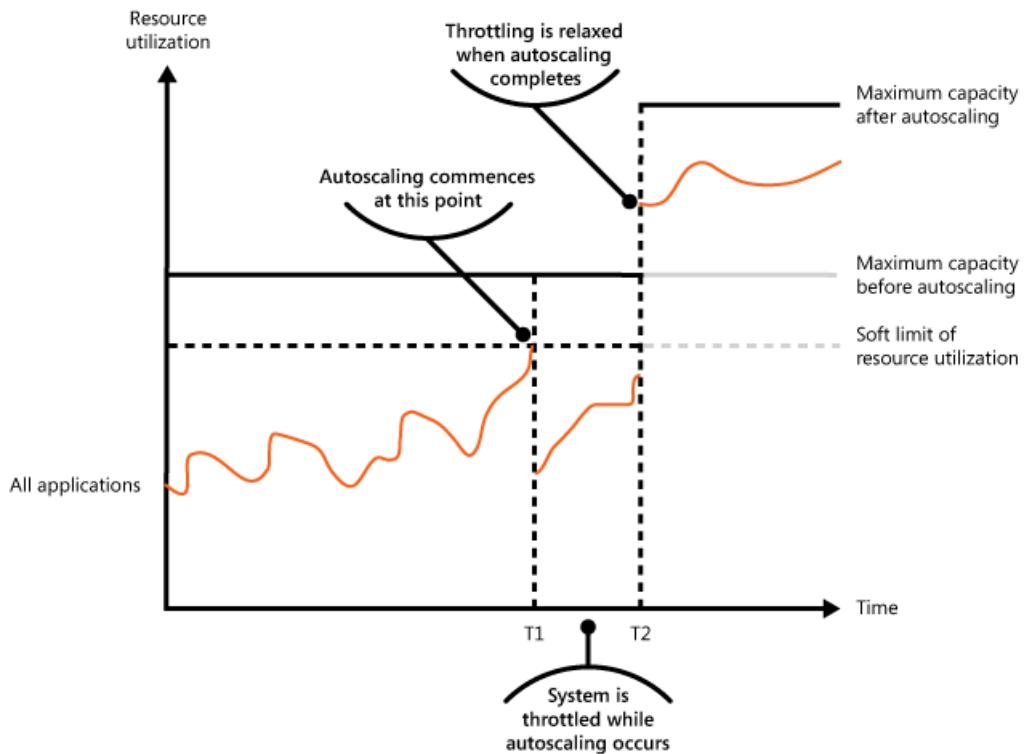


The area immediately below the line for a feature indicates the resources that are used by applications when they invoke this feature. For example, the area below the line for Feature A shows the resources used by applications that are making use of Feature A, and the area between the lines for Feature 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

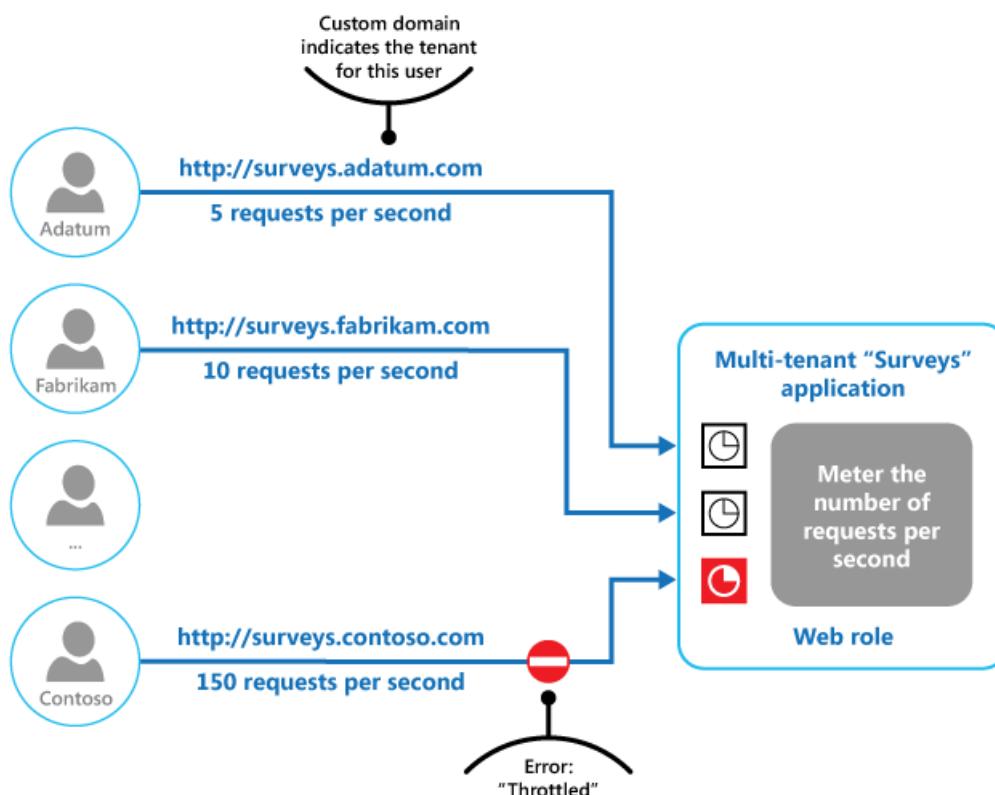
Use this pattern:

- 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.



## Related patterns and guidance

The following patterns and 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.
- [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

8/14/2017 • 13 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.

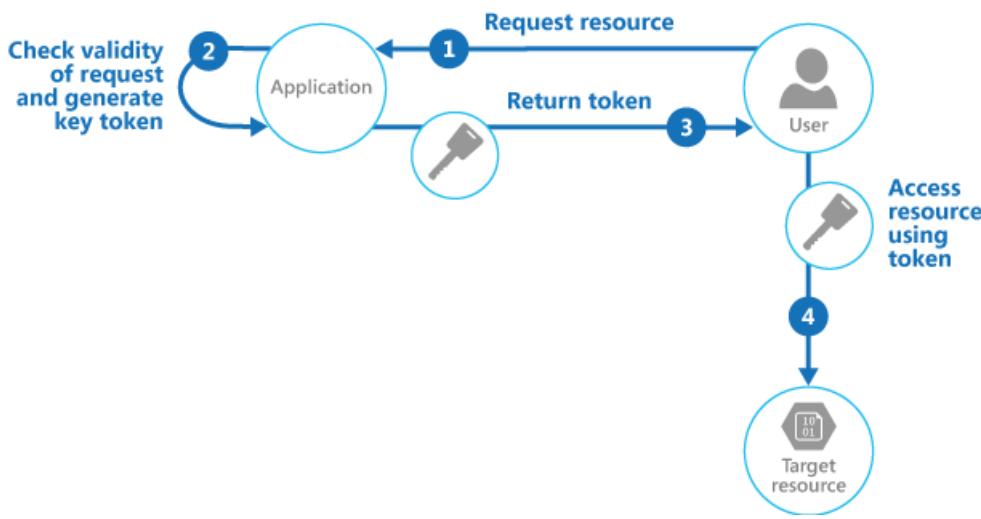
This isn't a realistic approach in distributed systems that need to serve untrusted clients. Instead, applications must be able to securely control access to data in a granular way, but still reduce the load on the server by setting up this connection and then allowing the client to communicate directly 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.

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 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 [Introducing Table SAS \(Shared Access Signature\)](#), [Queue SAS](#) and [update to Blob SAS](#) and [Using Shared Access Signatures](#) on MSDN.

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 CloudStorageAccount account;
    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 blobClient = this.account.CreateCloudBlobClient();
        var container = blobClient.GetContainerReference(this.blobContainer);

        var blob = container.GetBlockBlobReference(blobName);

        var policy = new SharedAccessBlobPolicy
        {
            Permissions = SharedAccessBlobPermissions.Write,
            // Specify a start time five minutes earlier to allow for client clock skew.
            SharedAccessStartTime = DateTime.UtcNow.AddMinutes(-5),
            // Specify a validity period of five minutes starting from now.
            SharedAccessExpiryTime = DateTime.UtcNow.AddMinutes(5)
        };

        // Create the signature.
        var sas = blob.GetSharedAccessSignature(policy);

        return new StorageEntitySas
        {
            BlobUri = blob.Uri,
            Credentials = sas,
            Name = blobName
        };
    }

    public struct StorageEntitySas
    {
        public string Credentials;
        public Uri BlobUri;
        public string Name;
    }
}

```

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 patterns and guidance might also be relevant when implementing this pattern:

- A sample that demonstrates this pattern is available on [GitHub](#).
- [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.

- [Introducing Table SAS \(Shared Access Signature\), Queue SAS and update to Blob SAS](#)
- [Using Shared Access Signatures](#)
- [Shared Access Signature Authentication with Service Bus](#)

# API design

9/28/2018 • 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 utilize it.

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

## Introduction to 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 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. (The article [Data Partitioning](#) describes strategies to scale out a data store.)

- 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 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 does not 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 [Using the HATEOAS Approach to Enable Navigation To Related Resources later](#).

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 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 ecommerce example. Note that 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". "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, 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 awhile 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 operations in REST](#).

## 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 sever 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 standards or specifications that define how to model the HATEOAS principle. The examples shown in this section illustrate one possible 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. Big 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 simply 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 sub-fields 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":"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 have an adverse impact on 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 utilize 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}}
```

Note that 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.

# API implementation

9/28/2018 • 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 focusses 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 [API Design Guidance](#).

## 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 ecommerce 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 URLs; when a client issues an HTTP GET request to obtain a resource, the response should contain URLs 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 include 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 PUT 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 makes use of 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(HttpStatusCode.OK);
        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 URLs 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 URLs 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. For more information, see [ServicePoint Class](#).

### 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 URLs that are this long.

## Maintaining responsiveness, scalability, and availability

The same web API might be utilized 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 an Azure Notification Hub to push asynchronous responses to client applications. For more information, see [Azure Notification Hubs Notify Users](#).
- 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 utilize sockets (or websockets 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 of 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 Management 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 URLs 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 URLs 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 Management 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 Management 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 from the Azure Management portal. You can also create webtests that monitor the health of the web API. A webtest 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 Management 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

9/28/2018 • 13 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.

## Overview

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.

- **Virtual Machines** support autoscaling through the use of [VM Scale Sets](#), which are a way to 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 auto-scaling through VM Scale Sets. Every node type in a Service Fabric cluster is set up as a separate VM scale set. That way, each node type can be scaled in or out independently. See [Scale a Service Fabric cluster in or out using auto-scale 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 VM 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.

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 well 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 most busy. 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 auto-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 be aware that periods of fewer 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.
- 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 that measure specific business processes.
- When autoscaling Service Fabric, the node types in your cluster are made of VM scale sets at the backend, so you need to set up auto-scale rules for each node type. Take into account the number of nodes that you must have before you set up auto-scaling. 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 info, see [scale a Service Fabric cluster in or out using auto-scale 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 shutdown, 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

9/28/2018 • 32 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 [Gatekeeper](#) 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, or a timer service such as Azure Scheduler, 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 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 Container Service.** Azure Container Service provides a container hosting environment on Azure.
- **Azure Cloud Services.** You can write code within a role that executes as a background task.

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 Scheduler 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 WebJobs that may be long running or resource intensive.

#### More information

- [Azure WebJobs recommended resources](#) lists the many useful resources, downloads, and samples for WebJobs.

### Azure Virtual Machines

Background tasks might be implemented in a way that prevents them from being deployed to Azure Web Apps or Cloud Services, 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 Scheduler 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 [Health Endpoint Monitoring Pattern](#).

#### More information

- [Virtual Machines](#) on Azure
- [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](#).

#### More information

- [Run intrinsically parallel workloads with Batch](#)
- [Develop large-scale parallel compute solutions with Batch](#)
- [Batch and HPC solutions for large-scale computing workloads](#)

## Azure Container Service

Azure Container Service lets you configure and manage a cluster of VMs in Azure to run containerized applications. It provides a choice of Docker Swarm, DC/OS, or Kubernetes for orchestration.

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 skillset of your DevOps team, this may or may not be an issue.
- Container Service runs in an IaaS environment. It provisions a cluster of VMs inside a dedicated VNet.

#### More information

- [Introduction to Docker container hosting solutions with Azure Container Service](#)
- [Introduction to private Docker container registries](#)

## Azure Cloud Services

You can execute background tasks within a web role or in a separate worker role. When you are deciding whether to use a worker role, consider scalability and elasticity requirements, task lifetime, release cadence, security, fault tolerance, contention, complexity, and the logical architecture. For more information, see [Compute Resource Consolidation Pattern](#).

There are several ways to implement background tasks within a Cloud Services role:

- Create an implementation of the **RoleEntryPoint** class in the role and use its methods to execute background tasks. The tasks run in the context of WebHost.exe. They can use the **GetSetting** method of the **CloudConfigurationManager** class to load configuration settings. For more information, see [Lifecycle](#).
- Use startup tasks to execute background tasks when the application starts. To force the tasks to continue to run in the background, set the **taskType** property to **background** (if you do not do this, the application startup process will halt and wait for the task to finish). For more information, see [Run startup tasks in Azure](#).
- Use the WebJobs SDK to implement background tasks such as WebJobs that are initiated as a startup task. For more information, see [Create a .NET WebJob in Azure App Service](#).
- Use a startup task to install a Windows service that executes one or more background tasks. You must set the **taskType** property to **background** so that the service executes in the background. For more information, see [Run startup tasks in Azure](#).

The main advantage of running background tasks in the web role is the saving in hosting costs because there is no requirement to deploy additional roles.

Running background tasks in a worker role has several advantages:

- It allows you to manage scaling separately for each type of role. For example, you might need more instances of a web role to support the current load, but fewer instances of the worker role that executes background tasks. By scaling background task compute instances separately from the UI roles, you can reduce hosting costs, while maintaining acceptable performance.
- It offloads the processing overhead for background tasks from the web role. The web role that provides the UI can remain responsive, and it may mean fewer instances are required to support a given volume of requests from users.
- It allows you to implement separation of concerns. Each role type can implement a specific set of clearly defined and related tasks. This makes designing and maintaining the code easier because there is less interdependence of code and functionality between each role.
- It can help to isolate sensitive processes and data. For example, web roles that implement the UI do not need to have access to data that is managed and controlled by a worker role. This can be useful in strengthening security, especially when you use a pattern such as the [Gatekeeper Pattern](#).

## Considerations

Consider the following points when choosing how and where to deploy background tasks when using Cloud Services web and worker roles:

- Hosting background tasks in an existing web role can save the cost of running a separate worker role just for these tasks. However, it is likely to affect the performance and availability of the application if there is contention for processing and other resources. Using a separate worker role protects the web role from the impact of long-running or resource-intensive background tasks.
- If you host background tasks by using the **RoleEntryPoint** class, you can easily move this to another role. For example, if you create the class in a web role and later decide that you need to run the tasks in a worker role, you can move the **RoleEntryPoint** class implementation into the worker role.
- Startup tasks are designed to execute a program or a script. Deploying a background job as an executable program might be more difficult, especially if it also requires deployment of dependent assemblies. It might be easier to deploy and use a script to define a background job when you use startup tasks.
- Exceptions that cause a background task to fail have a different impact, depending on the way that they are

hosted:

- If you use the **RoleEntryPoint** class approach, a failed task will cause the role to restart so that the task automatically restarts. This can affect availability of the application. To prevent this, ensure that you include robust exception handling within the **RoleEntryPoint** class and all the background tasks. Use code to restart tasks that fail where this is appropriate, and throw the exception to restart the role only if you cannot gracefully recover from the failure within your code.
- If you use startup tasks, you are responsible for managing the task execution and checking if it fails.
- Managing and monitoring startup tasks is more difficult than using the **RoleEntryPoint** class approach. However, the Azure WebJobs SDK includes a dashboard to make it easier to manage WebJobs that you initiate through startup tasks.

#### Lifecycle

If you decide to implement background jobs for Cloud Services applications that use web and worker roles by using the **RoleEntryPoint** class, it is important to understand the lifecycle of this class in order to use it correctly.

Web and worker roles go through a set of distinct phases as they start, run, and stop. The **RoleEntryPoint** class exposes a series of events that indicate when these stages are occurring. You use these to initialize, run, and stop your custom background tasks. The complete cycle is:

- Azure loads the role assembly and searches it for a class that derives from **RoleEntryPoint**.
- If it finds this class, it calls **RoleEntryPoint.OnStart()**. You override this method to initialize your background tasks.
- After the **OnStart** method has completed, Azure calls **Application\_Start()** in the application's Global file if this is present (for example, Global.asax in a web role running ASP.NET).
- Azure calls **RoleEntryPoint.Run()** on a new foreground thread that executes in parallel with **OnStart()**. You override this method to start your background tasks.
- When the Run method ends, Azure first calls **Application\_End()** in the application's Global file if this is present, and then calls **RoleEntryPoint.OnStop()**. You override the **OnStop** method to stop your background tasks, clean up resources, dispose of objects, and close connections that the tasks may have used.
- The Azure worker role host process is stopped. At this point, the role will be recycled and will restart.

For more details and an example of using the methods of the **RoleEntryPoint** class, see [Compute Resource Consolidation Pattern](#).

#### Implementation considerations

Consider the following points if you are implementing background tasks in a web or worker role:

- The default **Run** method implementation in the **RoleEntryPoint** class contains a call to **Thread.Sleep(Timeout.Infinite)** that keeps the role alive indefinitely. If you override the **Run** method (which is typically necessary to execute background tasks), you must not allow your code to exit from the method unless you want to recycle the role instance.
- A typical implementation of the **Run** method includes code to start each of the background tasks and a loop construct that periodically checks the state of all the background tasks. It can restart any that fail or monitor for cancellation tokens that indicate that jobs have completed.
- If a background task throws an unhandled exception, that task should be recycled while allowing any other background tasks in the role to continue running. However, if the exception is caused by corruption of objects outside the task, such as shared storage, the exception should be handled by your **RoleEntryPoint** class, all tasks should be cancelled, and the **Run** method should be allowed to end. Azure will then restart the role.
- Use the **OnStop** method to pause or kill background tasks and clean up resources. This might involve stopping long-running or multistep tasks. It is vital to consider how this can be done to avoid data inconsistencies. If a role instance stops for any reason other than a user-initiated shutdown, the code

running in the **OnStop** method must be completed within five minutes before it is forcibly terminated. Ensure that your code can be completed in that time or can tolerate not running to completion.

- The Azure load balancer starts directing traffic to the role instance when the **RoleEntryPoint.OnStart** method returns the value **true**. Therefore, consider putting all your initialization code in the **OnStart** method so that role instances that do not successfully initialize will not receive any traffic.
- You can use startup tasks in addition to the methods of the **RoleEntryPoint** class. You should use startup tasks to initialize any settings that you need to change in the Azure load balancer because these tasks will execute before the role receives any requests. For more information, see [Run startup tasks in Azure](#).
- If there is an error in a startup task, it might force the role to continually restart. This can prevent you from performing a virtual IP (VIP) address swap back to a previously staged version because the swap requires exclusive access to the role. This cannot be obtained while the role is restarting. To resolve this:
  - Add the following code to the beginning of the **OnStart** and **Run** methods in your role:

```
var freeze = CloudConfigurationManager.GetSetting("Freeze");
if (freeze != null)
{
    if (Boolean.Parse(freeze))
    {
        Thread.Sleep(System.Threading.Timeout.Infinite);
    }
}
```

- Add the definition of the **Freeze** setting as a Boolean value to the ServiceDefinition.csdef and ServiceConfiguration.\*.cscfg files for the role and set it to **false**. If the role goes into a repeated restart mode, you can change the setting to **true** to freeze role execution and allow it to be swapped with a previous version.

#### More information

- [Compute Resource Consolidation Pattern](#)
- [Get started with the Azure WebJobs SDK](#)

## Partitioning

If you decide to include background tasks within an existing compute instance (such as a web app, web role, existing worker role, or virtual machine), 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 a fewer number of 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 [Leader Election Pattern](#) and [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 fulfil 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 [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 [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 role or service 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 [Scheduler Agent Supervisor Pattern](#).
- When you use web or worker roles to host multiple background tasks, design your override of the **Run** method to monitor for failed or stalled tasks, and restart them. Where this is not practical, and you are using a worker role, force the worker role to restart by exiting from the **Run** method.
- 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 recycling the role will not block the UI. For more information, see [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, Cloud Services web and worker roles, 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 a Cloud Services application (for example, the UI or components such as the data access layer), hosting the background tasks together in a separate worker role allows the UI and background task roles to scale independently to manage the load. If multiple background tasks have significantly different performance capabilities from each other, consider dividing them into separate worker roles and scaling each role type independently. However, note that this might increase runtime costs compared to combining all the tasks into fewer roles.
- Simply scaling the roles 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

- [Asynchronous Messaging Primer](#)
- [Autoscaling Guidance](#)
- [Compensating Transaction Pattern](#)
- [Competing Consumers Pattern](#)
- [Compute Partitioning Guidance](#)
- [Compute Resource Consolidation Pattern](#)
- [Gatekeeper Pattern](#)
- [Leader Election Pattern](#)
- [Pipes and Filters Pattern](#)
- [Priority Queue Pattern](#)
- [Queue-based Load Leveling Pattern](#)
- [Scheduler Agent Supervisor Pattern](#)

## More information

- [Executing Background Tasks](#)
- [Azure Cloud Services Role Lifecycle](#) (video)
- [What is the Azure WebJobs SDK](#)
- [Run Background tasks with WebJobs](#)
- [Azure Queues and Service Bus Queues - Compared and Contrasted](#)
- [How to Enable Diagnostics in a Cloud Service](#)

# Caching

9/28/2018 • 56 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 which can be accessed by multiple processes and/or machines.

In both cases, caching can be performed client-side and/or 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 very quick to access. It can also provide an extremely effective means for storing modest amounts of static data, since the size of a cache is typically constrained by the volume of memory that's 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 that's 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.

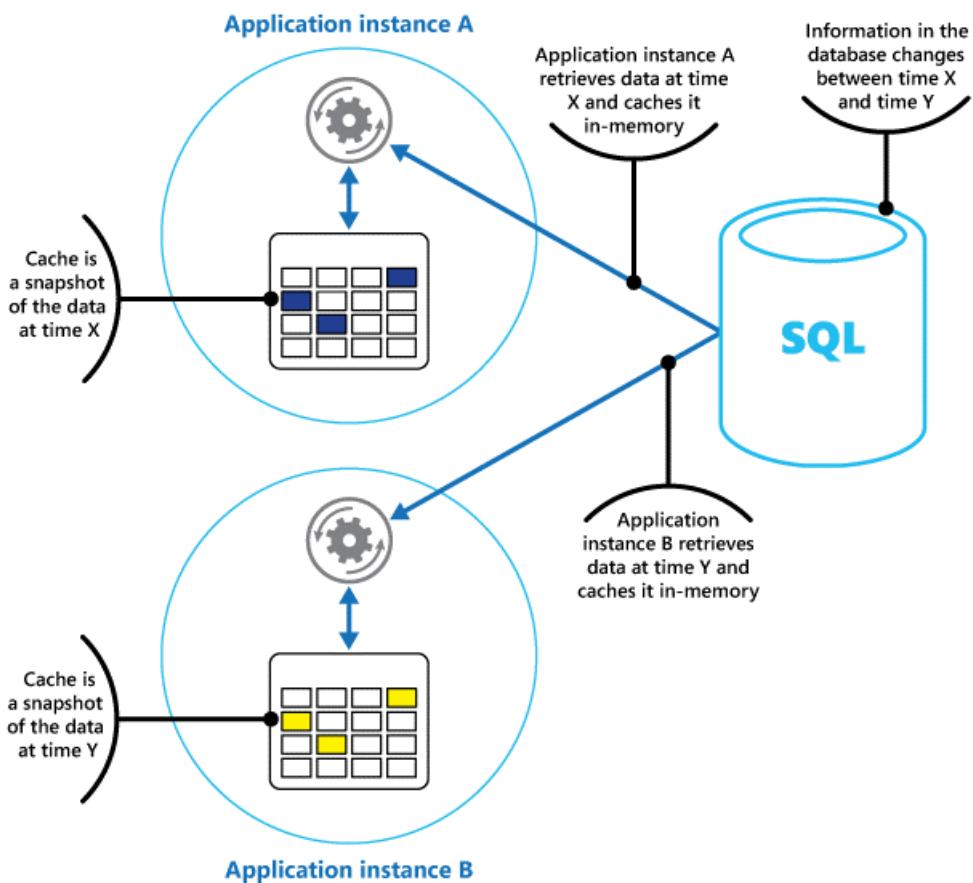


Figure 1: Using an in-memory cache in different instances of an application

### 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.

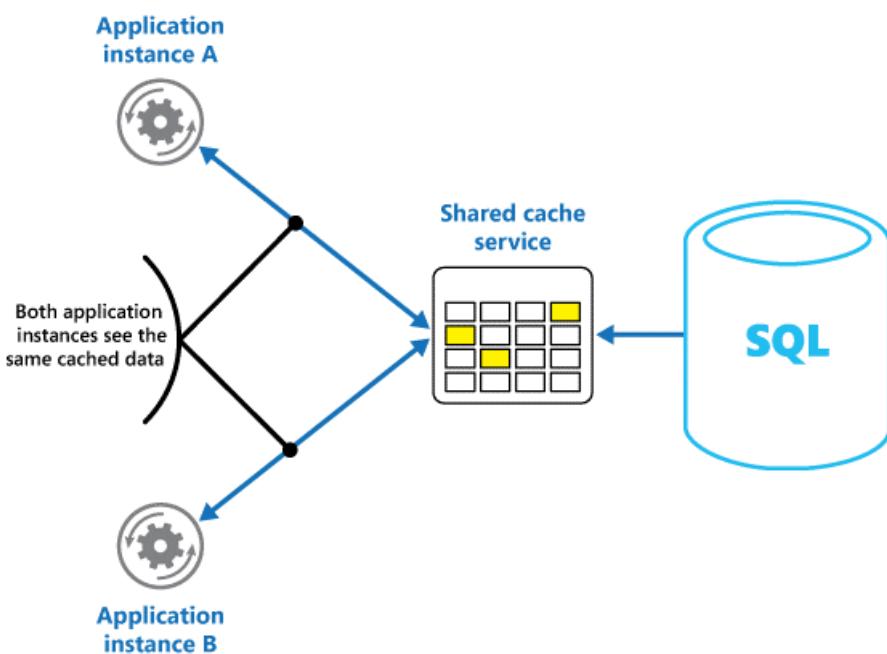


Figure 2: Using a shared cache

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 utilize software that distributes the data across the cluster in a transparent manner. An application instance simply sends a request to the cache service. The underlying

infrastructure is responsible for determining 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 pre-population 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 is 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 non-critical 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 hang 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, there might be a scalability impact on the system 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.

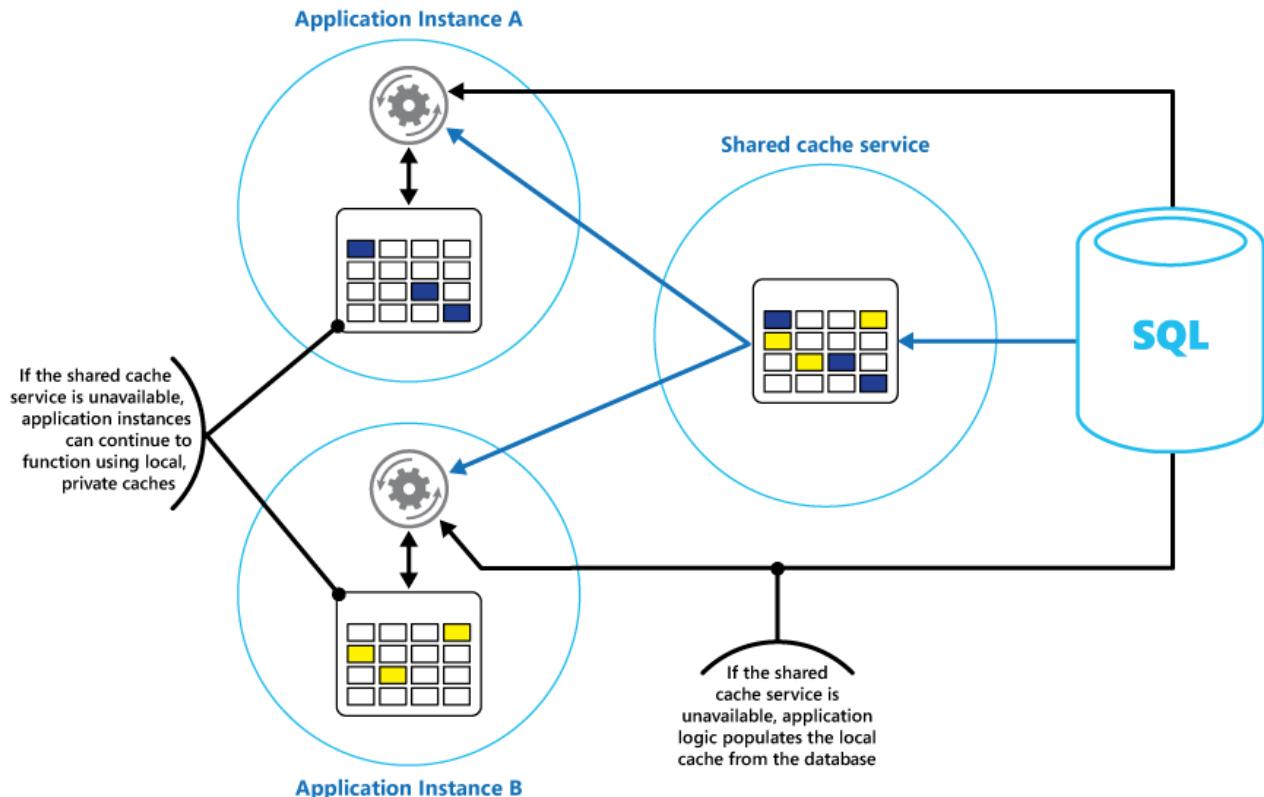


Figure 3: Using a local, private cache with a shared cache

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. The [Data partitioning guidance document](#) on the Microsoft website provides more information about possible partitioning strategies.

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 with Microsoft Azure

[Azure Redis Cache](#) 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 Redis Cache 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 Redis Cache is compatible with many of the various APIs that are used by client applications. If you have existing applications that already use Azure Redis Cache running on-premises, the Azure Redis Cache 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 master/subordinate replication to help ensure availability and maintain throughput. Write operations to a Redis master node are replicated to one or more subordinate nodes. Read operations can be served by the master or any of the subordinates.

In the event of a network partition, subordinates can continue to serve data and then transparently resynchronize with the master 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 master/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 non-transactional 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 Redis Cache provides its own security layer through which clients connect. The underlying Redis servers are not exposed to the public network.

## Azure Redis cache

Azure Redis Cache 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 Redis Cache, visit the page [Lap around Azure Redis Cache](#) 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 Redis Cache. The session state provider for Azure Redis Cache 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 Redis Cache 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 Redis Cache](#).

**NOTE**

Do not use the session state provider for Azure Redis Cache 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 Redis Cache enables you to save the HTTP responses generated by an ASP.NET web application. Using the output cache provider with Azure Redis Cache can improve the response times of applications that render complex HTML output. Application instances that generate similar responses can make use of 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 Redis Cache](#).

## Building a custom Redis cache

Azure Redis Cache acts as a façade to the underlying Redis servers. Currently it supports a fixed set of configurations but does not provide for Redis clustering. 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 master and subordinate nodes if you want to implement replication. Furthermore, if you wish to create a cluster, then you need multiple masters 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 master/subordinate servers (a cluster must contain at least three master nodes).

Each master/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**

Please note that 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 Redis Cache. Create multiple Azure Redis Caches (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 Redis Caches 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 Redis Cache; 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 Redis Cache, 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 Redis Cache, the password is either the primary or secondary key that is generated for Azure Redis Cache by using the Azure Management 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 Redis Cache, see [Azure Redis Cache 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. The next section in this article, Using Redis Caching, provides examples of some of the more advanced techniques that you can apply to data that's held in a Redis cache.

## 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.String.GetAsync` 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 the [Transactions in Redis](#) page on the StackExchange.Redis website.

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 how 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 towards 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 `ISeverer.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 pattern 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

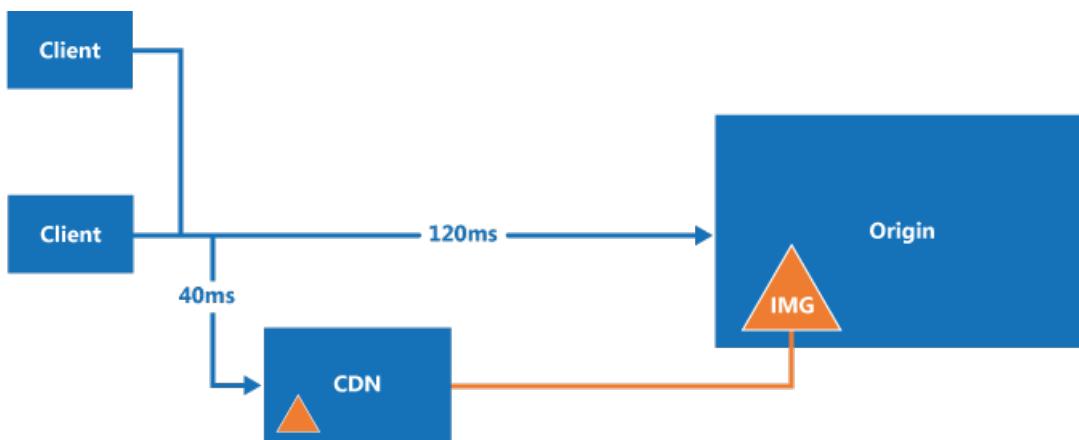
- The [MemoryCache class](#) page on the Microsoft website
- The [Azure Redis Cache documentation](#) page on the Microsoft website
- The [Azure Redis Cache FAQ](#) page on the Microsoft website
- The [Task-based Asynchronous Pattern](#) page on the Microsoft website
- The [Pipelines and multiplexers](#) page on the StackExchange.Redis GitHub repo
- The [Redis persistence](#) page on the Redis website
- The [Replication page](#) on the Redis website
- The [Redis cluster tutorial](#) page on the Redis website
- The [Partitioning: how to split data among multiple Redis instances](#) page on the Redis website
- The [Using Redis as an LRU Cache](#) page on the Redis website
- The [Transactions](#) page on the Redis website
- The [Redis security](#) page on the Redis website
- The [Lap around Azure Redis Cache](#) page on the Azure blog
- The [Running Redis on a CentOS Linux VM in Azure](#) page on the Microsoft website
- The [ASP.NET session state provider for Azure Redis Cache](#) page on the Microsoft website
- The [ASP.NET output cache provider for Azure Redis Cache](#) page on the Microsoft website
- The [An Introduction to Redis data types and abstractions](#) page on the Redis website
- The [Basic usage](#) page on the StackExchange.Redis website
- The [Transactions in Redis](#) page on the StackExchange.Redis repo
- The [Data partitioning guide](#) on the Microsoft website

# Best practices for using content delivery networks (CDNs)

9/28/2018 • 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. To learn more about using Azure CDN, see [CDN Documentation](#).

## 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 data centers. 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 increased 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 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 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.

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. Azure CDN allows you to filter requests based on the country of origin and restrict the content delivered. For more information, see [Restrict access to your content by country](#).

## 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 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.

# Data partitioning

9/28/2018 • 65 minutes to read • [Edit Online](#)

In many large-scale solutions, data is divided into separate partitions that can be managed and accessed separately. The partitioning strategy must be chosen carefully to maximize the benefits while minimizing adverse effects. Partitioning can help improve scalability, reduce contention, and optimize performance. Another benefit of partitioning is that it can provide a mechanism for dividing data by the pattern of use. For example, you can archive older, less active (cold) data in cheaper data storage.

## Why partition data?

Most cloud applications and services store and retrieve data as part of their operations. The design of the data stores that an application uses can have a significant bearing on the performance, throughput, and scalability of a system. One technique that is commonly applied in large-scale systems is to divide the data into separate partitions.

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.

Partitioning data can offer a number of benefits. For example, it can be applied in order to:

- **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 of which is 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. Provided that the data is partitioned in a suitable way, partitioning can make your system more efficient. Operations that affect more than one partition can run in parallel. Each partition can be located near the application that uses it to minimize network latency.
- **Improve availability.** Separating data across multiple servers avoids a single point of failure. If a server fails, or is undergoing planned maintenance, only the data in that partition is unavailable. Operations on other partitions can continue. Increasing the number of partitions reduces the relative impact of a single server failure by reducing the percentage of data that will be unavailable. Replicating each partition can further reduce the chance of a single partition failure affecting operations. It also makes it possible to separate critical data that must be continually and highly available from low-value data that has lower availability requirements (log files, for example).
- **Improve security.** Depending on the nature of the data and how it is partitioned, it might be possible to separate sensitive and non-sensitive data into different partitions, and therefore into different servers or data stores. Security can then be specifically optimized for 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 a blob data store, while more structured data can be held in a document database. For more information, see [Building a polyglot solution](#) in the patterns & practices guide and [Data access for highly-scalable solutions: Using SQL, NoSQL, and polyglot persistence](#) on the Microsoft website.

Some systems do not implement partitioning because it is considered a cost rather than an advantage. Common

reasons for this rationale include:

- Many data storage systems do not support joins across partitions, and it can be difficult to maintain referential integrity in a partitioned system. It is frequently necessary to implement joins and integrity checks in application code (in the partitioning layer), which can result in additional I/O and application complexity.
- Maintaining partitions is not always a trivial task. In a system where the data is volatile, you might need to rebalance partitions periodically to reduce contention and hot spots.
- Some common tools do not work naturally with partitioned data.

## Designing partitions

Data can be partitioned in different ways: horizontally, vertically, or functionally. The strategy you choose depends on the reason for partitioning the data, and the requirements of the applications and services that will use the data.

### NOTE

The partitioning schemes described in this guidance are explained in a way that is independent of the underlying data storage technology. They can be applied to many types of data stores, including relational and NoSQL databases.

### Partitioning strategies

The three typical strategies for partitioning data are:

- **Horizontal partitioning** (often called *sharding*). In this strategy, each partition is a data store in its own right, 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 in an e-commerce application.
- **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 that implements separate business functions for invoicing and managing product inventory might store invoice data in one partition and product inventory data in another.

It's important to note that the three strategies described here can be combined. They are not mutually exclusive, 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. Similarly, the data in a functional partition can be split into shards (which can also be vertically partitioned).

However, the differing requirements of each strategy can raise a number of conflicting issues. You must evaluate and balance all of these when designing a partitioning scheme that meets the overall data processing performance targets for your system. The following sections explore each of the strategies in more detail.

### Horizontal partitioning (sharding)

Figure 1 shows an overview of 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.

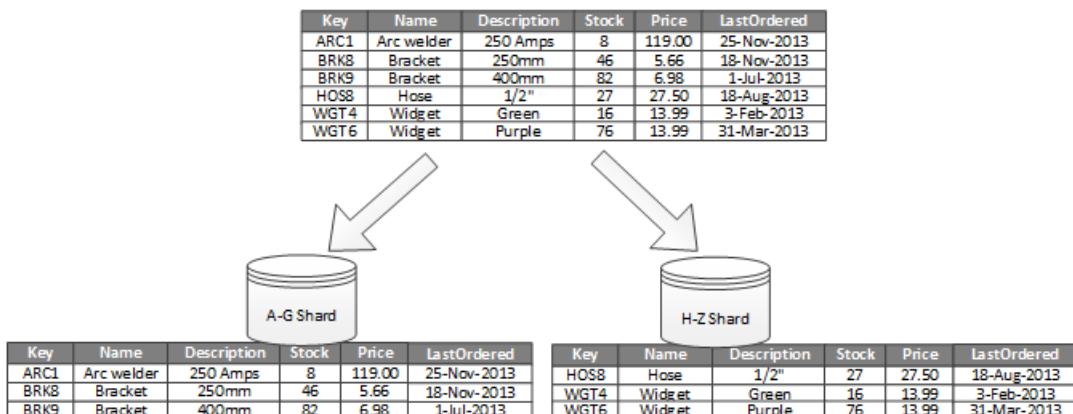


Figure 1. Horizontally partitioning (sharding) data based on a partition key

Sharding helps you spread the load over more computers, which reduces contention and improves performance. You can scale the system out by adding further shards that run on additional servers.

The most important factor when implementing this partitioning strategy is the choice of sharding key. It can be difficult to change the key after the system is in operation. The key must ensure that data is partitioned so that the workload is as even as possible across the shards.

Note that different shards do not have to contain similar volumes of data. Rather, the more important consideration is to balance the number of requests. Some shards might be very large, but each item is the subject of a low number of access operations. Other shards might be smaller, but each item is accessed much more frequently. It is 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 that's being used to host that shard.

If you use a sharding scheme, avoid creating hotspots (or hot partitions) that can affect performance and availability. For example, if you use a hash of a customer identifier instead of the first letter of a customer's name, you prevent the unbalanced distribution that results from common and less common initial letters. This is a typical technique that helps distribute data more evenly across partitions.

Choose a sharding key that minimizes any future requirements to split large shards into smaller pieces, coalesce small shards into larger partitions, or change the schema that describes the data stored in a set of partitions. 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 on the data in these shards while the reconfiguration is taking place. For example, the data in the replicas can be marked as read-only to limit the scope of inconsistencies that might occur while shards are being restructured.

For more detailed information and guidance about many of these considerations, and good practice techniques for designing data stores that implement 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 the items that are accessed most frequently. Figure 2 shows an example of vertical partitioning. In this example, different properties for each data item are held in different partitions. One partition holds data that is accessed more frequently, including the name, description, and price information for products. Another holds the volume in stock and the last ordered date.



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Stock	LastOrdered
ARC1	8	25-Nov-2013
BRK8	46	18-Nov-2013
BRK9	82	1-Jul-2013
HOS8	27	18-Aug-2013
WGT4	16	3-Feb-2013
WGT6	76	31-Mar-2013

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. The stock level and date when the product was last ordered from the manufacturer are held in a separate partition because these two items are commonly used together.

This partitioning scheme has the added advantage that the relatively slow-moving data (product name, description, and price) is separated from the more dynamic data (stock level and last ordered date). An application might find it beneficial to cache the slow-moving data in memory if it is frequently accessed.

Another typical scenario for this partitioning strategy is to maximize the security of sensitive data. For example, you can do this by storing credit card numbers and the corresponding card security verification numbers in separate partitions.

Vertical partitioning can also reduce the amount of concurrent access that's needed to the data.

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

For systems where it is possible to identify a bounded context for each distinct business area or service in the application, functional partitioning provides a technique for improving isolation and data access performance. Another common use of functional partitioning is to separate read-write data from read-only data that's used for reporting purposes. Figure 3 shows an overview of functional partitioning where inventory data is separated from customer data.

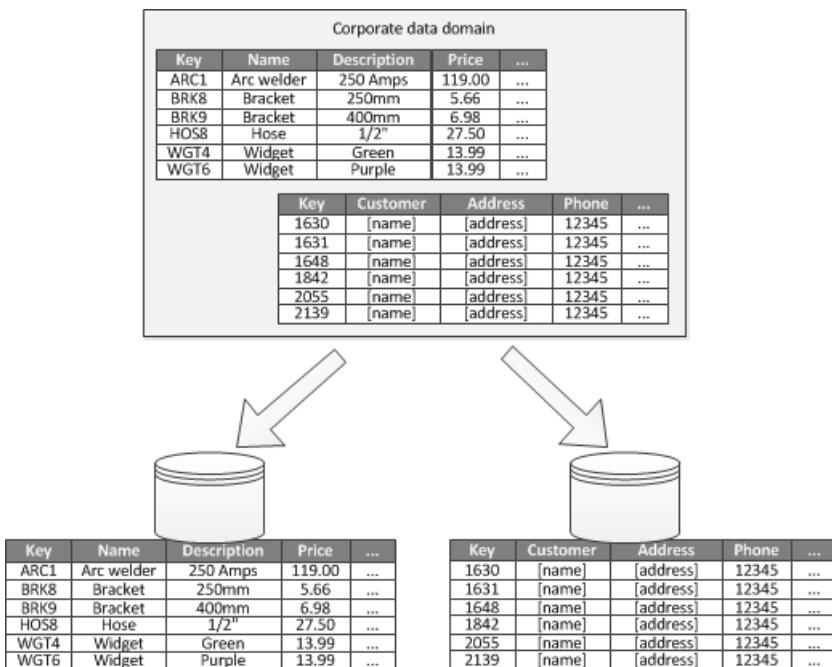


Figure 3. Functionally partitioning data by bounded context or subdomain

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. In the horizontal partitioning strategy, choosing the appropriate shard key is important to make sure distribution is even. For more information, see the [Sharding pattern](#).
3. Make sure that the resources available to each partition are sufficient to handle the scalability requirements in terms of data size and throughput. For example, the node that's hosting a partition might impose a hard limit on the amount of storage space, processing power, or network bandwidth that it provides. If the data storage and processing requirements are likely to exceed these limits, it might be necessary to refine your partitioning strategy or split data out further. For example, one scalability approach might be to separate logging data from the core application features. You do this by using separate data stores to prevent the total data storage requirements from exceeding the scaling limit of the node. If the total number of data stores exceeds the node limit, it might be necessary to use separate storage nodes.
4. Monitor the system under use to verify that the data is distributed as expected and that the partitions can handle the load that is imposed on them. It's possible that the usage does not match the usage that's anticipated by the analysis. In that case, it might be possible to rebalance the partitions. Failing that, it might be necessary to redesign some parts of the system to gain the required balance.

Note that 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, a busy shard might require more resources than are available to a

single partition to handle requests. (There is a limit to the volume of requests that can be handled by a single partition in a particular period of time. See the page [Azure storage scalability and performance targets](#) on the Microsoft website for more details.)

If this is the case, 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, it might be necessary to create additional storage accounts and spread the tables across these accounts. If the number of storage accounts exceeds the number of accounts that are available to a subscription, then it might be necessary to use multiple subscriptions.

## 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 if you are using a relational database.

Follow these steps when designing partitions for query performance:

1. Examine the application requirements and performance:
  - Use the business requirements to determine the critical queries that must always perform quickly.
  - Monitor the system to identify any queries that perform slowly.
  - Establish which queries are performed most frequently. A single instance of each query might have minimal cost, but the cumulative consumption of resources could be significant. It might be beneficial to separate the data that's retrieved by these queries into a distinct partition, or even a cache.
2. Partition the data that is causing slow performance:
  - Limit the size of each partition so that the query response time is within target.
  - Design the shard key so that the application can easily find the partition if you are implementing horizontal partitioning. 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 using asynchronous queries that run 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. Replicating partitions that contain critical data can also improve availability.

When designing and implementing partitions, consider the following factors that affect availability:

- **How critical the data is to business operations.** Some data might include critical business information such as invoice details or bank transactions. Other data might include less critical operational data, such as log files, performance traces, and so on. After identifying each type of data, consider:
  - Storing critical data in highly-available partitions with an appropriate backup plan.
  - Establishing separate management and monitoring mechanisms or procedures for the different criticalities of each dataset. 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 data for bank transactions 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 affecting instances of 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 big 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, although it can also introduce consistency issues. It takes time for changes made to data in a partition to be synchronized with every replica. During this period, different partitions will contain different data values.

## Understanding how partitioning affects design and development

Using 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, when the system starts to suffer performance and scalability issues, the complexity increases because you already have a live system to maintain.

If you update the system to incorporate partitioning in this environment, it necessitates modifying the data access logic. It can also involve migrating large quantities of existing data to distribute it across partitions, often while users expect to be able to continue using the system.

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 in a system that is not expected to scale beyond its initial size, but many commercial systems need to expand as the number of users increases. This expansion is typically accompanied by a growth in the volume of data.

It's also important to understand that partitioning is not always a function of large data stores. 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:

- **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 only within a single partition, but optimizing partitions for one set of queries might adversely affect other sets of queries. When you can't avoid querying 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 it's necessary to obtain a result from one query and use it in the next query.
- **If queries make use of relatively static reference data, such as postal code tables or product lists, consider replicating this data in all of the partitions to reduce the requirement for separate lookup operations in different partitions.** This approach can also reduce the likelihood of the reference data becoming a "hot" dataset that is subject to heavy traffic from across the entire system. However, there is an additional cost associated with synchronizing any changes that might occur to this reference data.
- **Where possible, minimize requirements for referential integrity across vertical and functional partitions.** In these schemes, the application itself is responsible for maintaining referential integrity across partitions when data is updated and consumed. Queries that must join data across multiple partitions run more slowly than queries that join data only within the same partition because the application typically needs to perform consecutive queries based on a key and then on a foreign key. Instead, consider replicating or de-normalizing the relevant data. To minimize the query time where cross-partition joins are necessary, run parallel queries over the partitions and join the data within the application.
- **Consider the effect that the partitioning scheme might have on the data consistency across**

**partitions.** Evaluate whether strong consistency is actually a requirement. Instead, a common approach in the cloud 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. For more information about implementing eventual consistency, see the [Data consistency primer](#).

- **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. Queries that are used with vertical and functional partitioning strategies can naturally specify the partitions. However, horizontal partitioning (sharding) can make locating an item difficult because every shard has the same schema. A typical solution for sharding is to maintain a map that can be used to look up the shard location for specific items of data. This map can be implemented in the sharding logic of the application, or maintained by the data store if it supports transparent sharding.
- **When using a horizontal partitioning strategy, consider periodically rebalancing the shards.** This helps 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.
- **If you replicate each partition, it provides additional protection against failure.** If a single replica fails, queries can be directed towards 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. This means that you might have to create or obtain new tools and utilities.
- **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 (perhaps 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 raise an alert to an operator to correct the problems manually. For example, in an e-commerce application, order information might be held in one partition but the line items that constitute each order might be held in another. The process of placing an order needs to add data to other partitions. If this process fails, there might be line items stored for which there is no corresponding order.

Different data storage technologies typically provide their own features to support partitioning. The following sections summarize the options that are implemented by data stores commonly used by Azure applications. They also describe considerations for designing applications that can best take advantage of these features.

# Partitioning strategies for Azure SQL Database

Azure SQL Database is a relational database-as-a-service that runs in the cloud. It is based on Microsoft SQL Server. A relational database divides information into tables, and each table holds information about entities as a series of rows. Each row contains columns that hold the data for the individual fields of an entity. The page [What is Azure SQL Database?](#) on the Microsoft website provides detailed documentation about creating and using SQL databases.

## Horizontal partitioning with Elastic 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. The Elastic Database feature of SQL Database supports horizontal scaling for a SQL database. Using Elastic Database, 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. Using Elastic Database can also help reduce contention by distributing the load across databases.

### NOTE

Elastic Database is a replacement for the Federations feature of Azure SQL Database. Existing SQL Database Federation installations can be migrated to Elastic Database by using the Federations migration utility. Alternatively, you can implement your own sharding mechanism if your scenario does not lend itself naturally to the features that are provided by Elastic Database.

Each shard is implemented as a SQL database. A shard can hold more than one dataset (referred to as a *shardlet*). Each database maintains metadata that describes the shardlets that it contains. A shardlet can be a single data item, or it can be a group of items that share the same shardlet key. For example, if you are sharding data in a multitenant application, the shardlet key can be the tenant ID, and all data for a given tenant can be held as part of the same shardlet. Data for other tenants would be held in different shardlets.

It is the programmer's responsibility to associate a dataset with a shardlet key. A separate SQL database acts as a global shard map manager. This database contains a list of all the shards and shardlets in the system. A client application that accesses data connects first to the global shard map manager database to obtain a copy of the shard map (listing shards and shardlets), which it then caches locally.

The application then uses this information to route data requests to the appropriate shard. This functionality is hidden behind a series of APIs that are contained in the Azure SQL Database Elastic Database Client Library, which is available as a NuGet package. The page [Elastic Database features overview](#) on the Microsoft website provides a more comprehensive introduction to Elastic Database.

### NOTE

You can replicate the global shard map manager database to reduce latency and improve availability. If you implement the database by using one of the Premium pricing tiers, you can configure active geo-replication to continuously copy data to databases in different regions. Create a copy of the database in each region in which users are based. Then configure your application to connect to this copy to obtain the shard map.

An alternative approach is to use Azure SQL Data Sync or an Azure Data Factory pipeline 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. Additionally, the shard map manager database does not have to be created by using a Premium pricing tier.

Elastic Database provides two schemes for mapping data to shardlets and storing them in shards:

- A **list shard map** describes an association between a single key and 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 privacy and isolation (that is, to prevent one tenant from exhausting the data storage resources available to others), each shardlet can be held within its own shard.

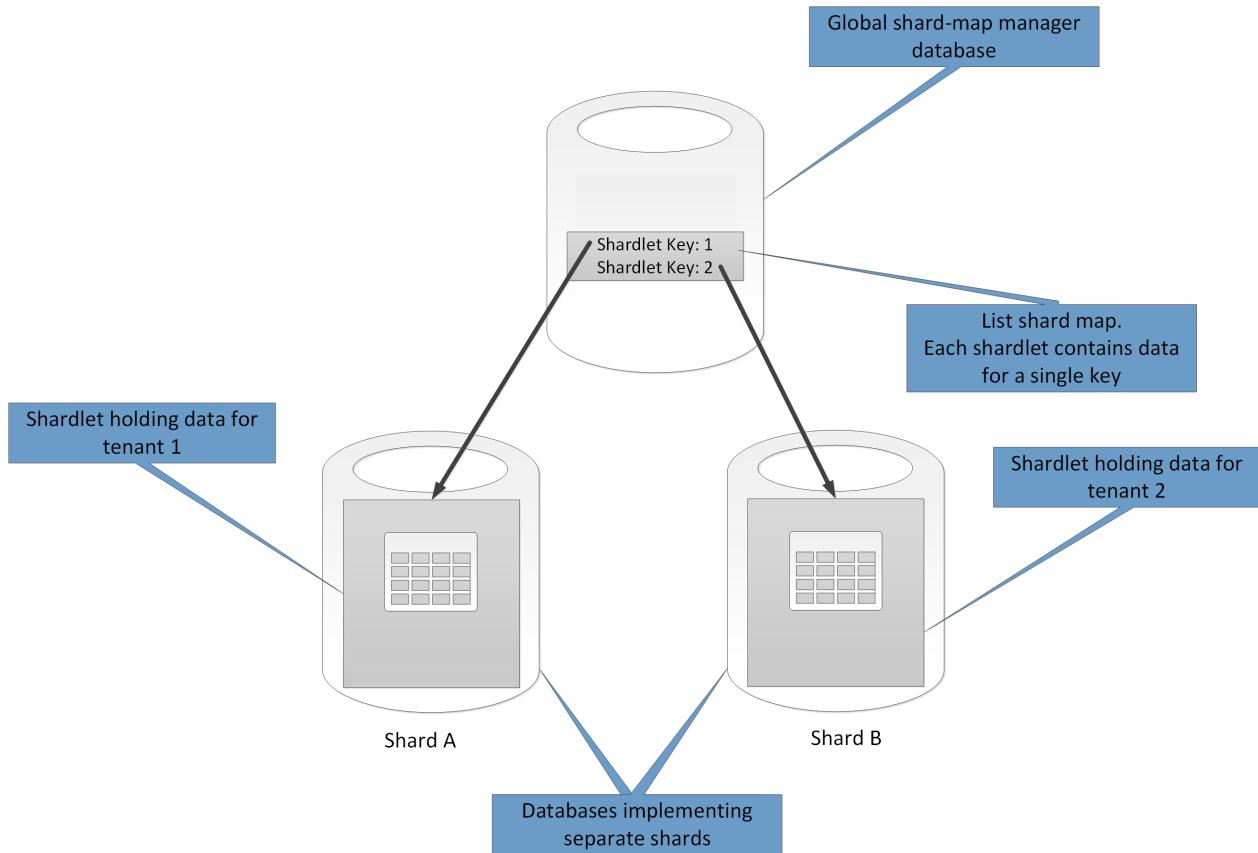


Figure 4. Using a list shard map to store tenant data in separate shards

- A **range shard map** describes an association between a set of contiguous key values and a shardlet. In the multitenant example described previously, as an alternative to implementing dedicated shardlets, 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 resources), but it also creates a risk of reduced data privacy and isolation.

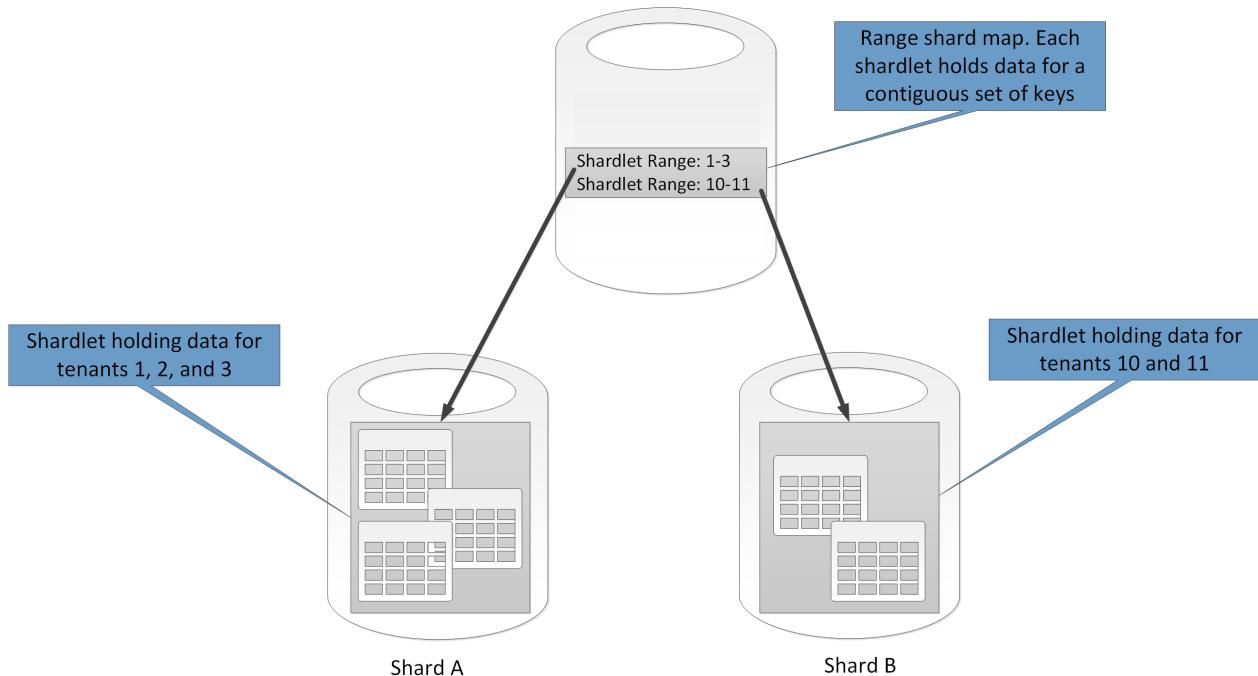
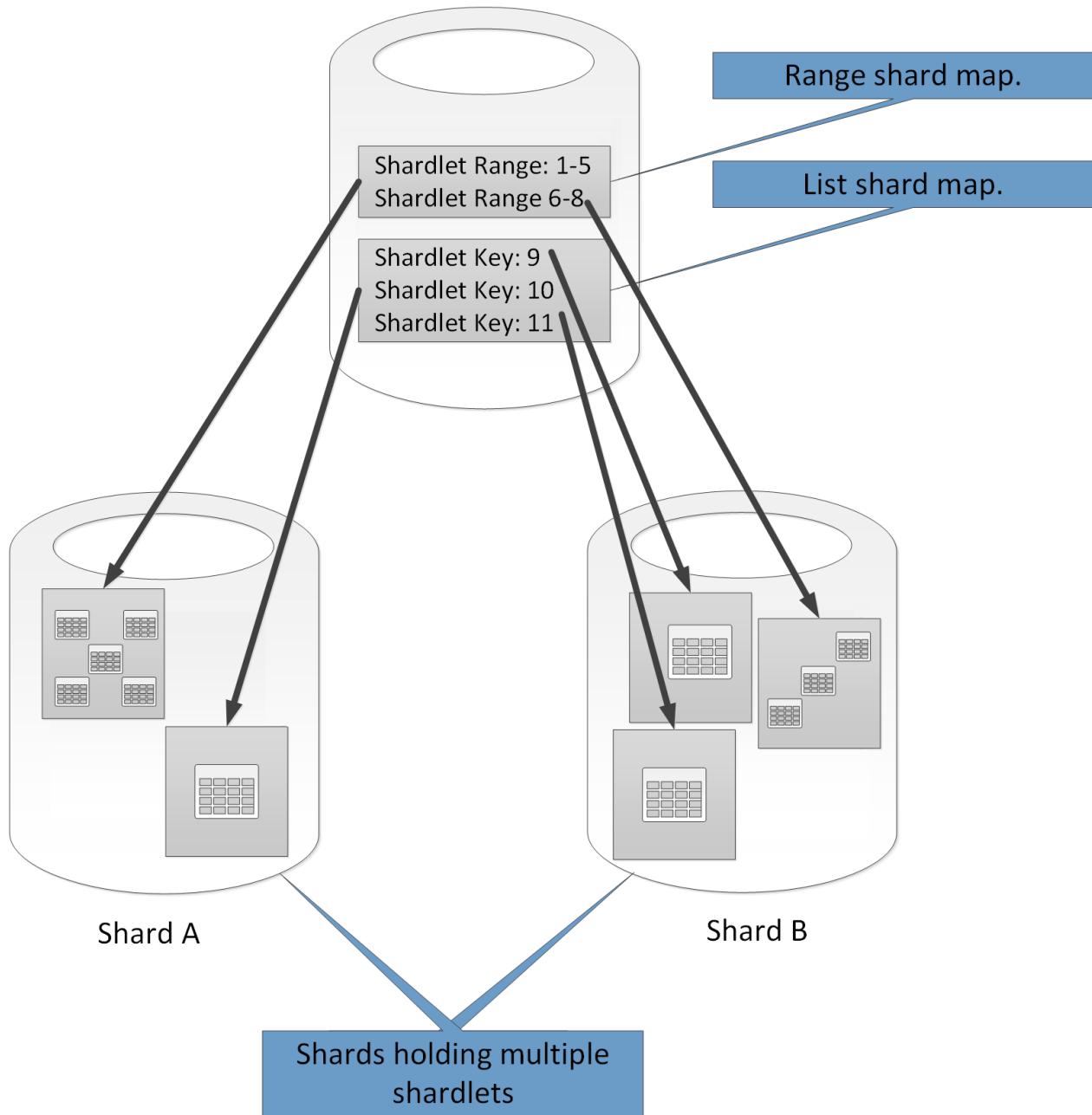


Figure 5. Using a range shard map to store data for a range of tenants in a shard

Note that 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 in the global shard map manager database. (The global shard map manager database can contain multiple shard maps.) Figure 6 depicts this approach.



*Figure 6. Implementing multiple shard maps*

The partitioning scheme that you implement can have a significant bearing on the performance of your system. It can also affect the rate at which shards have to be added or removed, or the rate at which data must be repartitioned across shards. Consider the following points when you use Elastic Database to partition data:

- Group data that is used together in the same shard, and avoid operations that need to access data that's held in multiple shards. Keep in mind that with Elastic Database, a shard is a SQL database in its own right, and Azure SQL Database does not support cross-database joins (which have to be performed on the client side). Remember also that in Azure SQL Database, referential integrity constraints, triggers, and stored procedures in one database cannot reference objects in another. Therefore, don't design a system that has dependencies between shards. A SQL database can, however, contain tables that hold copies of reference data frequently used by queries and other operations. These tables do not have to belong to any specific shardlet. Replicating this data across shards can help remove the need to join data that spans

databases. Ideally, such data should be static or slow-moving to minimize the replication effort and reduce the chances of it becoming stale.

**NOTE**

Although SQL Database does not support cross-database joins, you can perform cross-shard queries with the Elastic Database API. These queries can transparently iterate through the data held in all the shardlets that are referenced by a shard map. The Elastic Database API breaks cross-shard queries down into a series of individual queries (one for each database) and then merges the results. For more information, see the page [Multi-shard querying](#) on the Microsoft website.

- The data stored in shardlets that belong to the same shard map should have the same schema. For example, don't create a list shard map that points to some shardlets containing tenant data and other shardlets containing product information. This rule is not enforced by Elastic Database, but data management and querying becomes very complex if each shardlet has a different schema. In the example just cited, a good solution is to create two list shard maps: one that references tenant data and another that points to product information. Remember that the data belonging to different shardlets can be stored in the same shard.

**NOTE**

The cross-shard query functionality of the Elastic Database API depends on each shardlet in the shard map containing the same schema.

- Transactional operations are only supported for data that's held within the same 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 affected data in the same shard or implement eventual consistency. For more information, see the [Data consistency primer](#).
- Place shards close to the users that access the data in those shards (in other words, geo-locate the shards). This strategy helps reduce latency.
- Avoid having a mixture of highly active (hotspots) and relatively inactive shards. Try to spread the load evenly across shards. This might require hashing the shardlet 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.
- Currently, only a limited set of SQL data types are supported as shardlet keys; *int*, *bigint*, *varbinary*, and *uniqueidentifier*. The SQL *int* and *bigint* types correspond to the *int* and *long* data types in C#, and have the same ranges. The SQL *varbinary* type can be handled by using a *Byte* array in C#, and the SQL *uniqueidentier* type corresponds to the *Guid* class in the .NET Framework.

As the name implies, Elastic Database makes it possible for a system to add and remove shards as the volume of data shrinks and grows. The APIs in the Azure SQL Database Elastic Database client library enable an application to 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, Elastic Database provides a separate split-merge service. This service runs in a cloud-hosted service (which must be created by the developer) and migrates data safely between shards. For more information, see the topic [Scaling using the Elastic Database split-merge tool](#) on the Microsoft website.

## Partitioning strategies for Azure Storage

Azure storage provides four abstractions for managing data:

- Blob Storage stores unstructured object data. A blob can be any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as Object storage.
- Table Storage stores structured datasets. Table storage is a NoSQL key-attribute data store, which allows for rapid development and fast access to large quantities of data.
- Queue Storage provides reliable messaging for workflow processing and for communication between components of cloud services.
- File Storage offers shared storage for legacy applications using the standard SMB protocol. Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File service REST API.

Table storage and blob storage are essentially key-value stores that are optimized to hold structured and unstructured data respectively. Storage queues provide a mechanism for building loosely coupled, scalable applications. Table storage, file storage, blob storage, and storage queues are created within the context of an Azure storage account. Storage accounts support three forms of redundancy:

- **Locally redundant storage**, which maintains three copies of data within a single datacenter. This form of redundancy protects against hardware failure but not against a disaster that encompasses the entire datacenter.
- **Zone-redundant storage**, which maintains three copies of data spread across different datacenters within the same region (or across two geographically close regions). This form of redundancy can protect against disasters that occur within a single datacenter, but cannot protect against large-scale network disconnects that affect an entire region. Note that zone-redundant storage is currently only available for block blobs.
- **Geo-redundant storage**, which maintains six copies of data: three copies in one region (your local region), and another three copies in a remote region. This form of redundancy provides the highest level of disaster protection.

Microsoft has published scalability targets for Azure Storage. For more information, see the page [Azure Storage scalability and performance targets](#) on the Microsoft website. Currently, the total storage account capacity cannot exceed 500 TB. (This includes the size of data that's held in table storage, file storage and blob storage, as well as outstanding messages that are held in storage queue).

The maximum request rate for a storage account (assuming a 1-KB entity, blob, or message size) is 20,000 requests per second. A storage account has a maximum of 1000 IOPS (8 KB in size) per file share. If your system is likely to exceed these limits, consider partitioning the load across multiple storage accounts. A single Azure subscription can create up to 200 storage accounts. However, note that these limits might change over time.

## 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 that's stored in a table must provide a two-part key that includes:

- **The partition key**. This is a string value that determines in which partition Azure table storage will place the entity. All entities with the same partition key will be stored in the same partition.
- **The row key**. This is another 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.

The remainder of the data for an entity consists of application-defined fields. No particular schemas are enforced, and each row can contain a different set of application-defined fields. The only limitation is that the maximum size of an entity (including the partition and row keys) is currently 1 MB. The maximum size of a table is 200 TB, although these figures might change in the future. (Check the page [Azure Storage scalability and performance](#)

targets on the Microsoft website for the most recent information about these limits.)

If you are attempting to store entities that exceed this capacity, then consider splitting them into multiple tables. Use vertical partitioning to divide the fields into the groups that are most likely to be accessed together.

Figure 7 shows the logical structure of an example storage account (Contoso Data) for a fictitious e-commerce application. 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 in which the customer is located, and the row key contains the customer ID. In the Product Info table, the products are partitioned by product category, and the row key contains the product number. In the Order Info table, the orders are partitioned by the date on which they were placed, and the row key specifies the time the order was received. Note that all data is ordered by the row key in each partition.

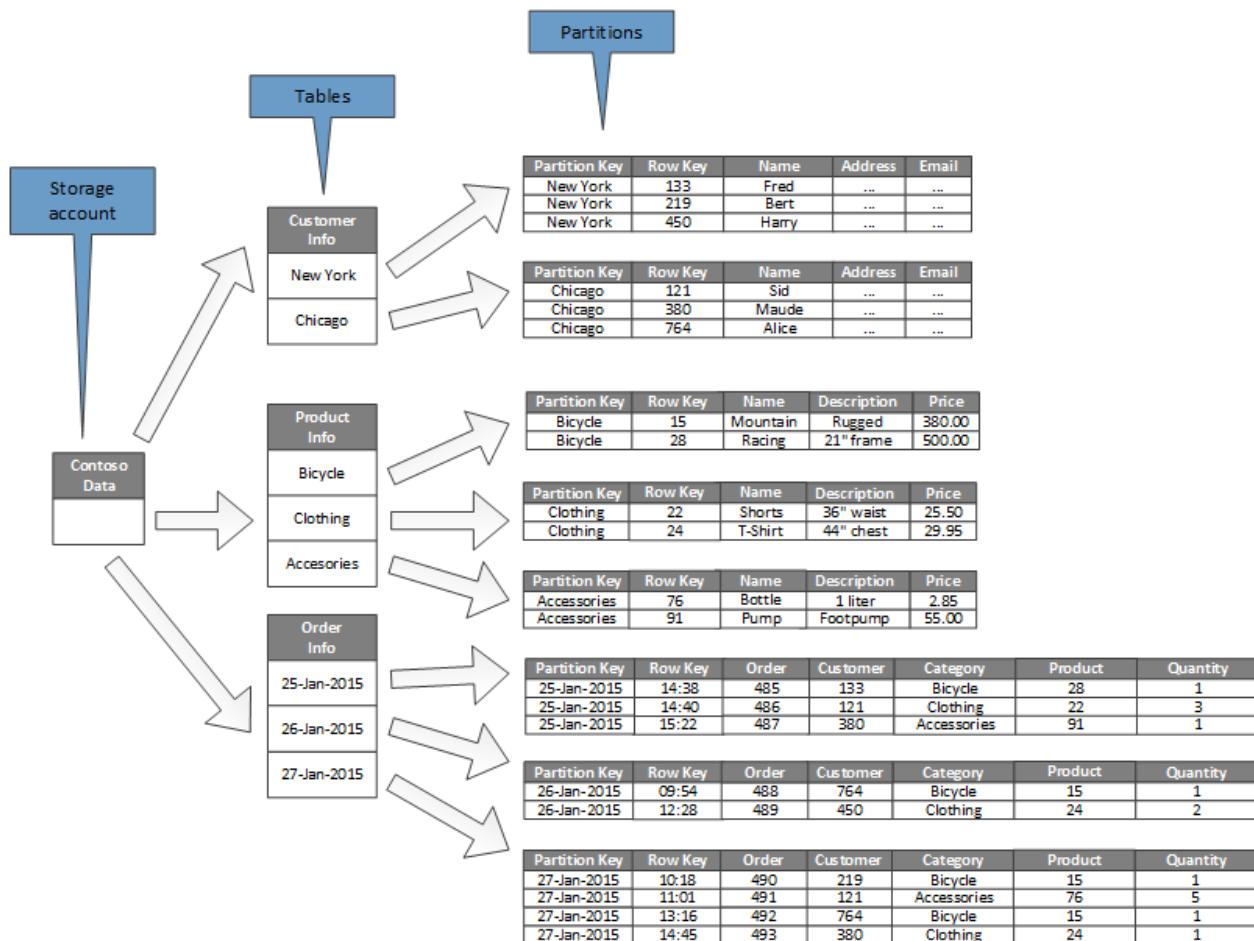


Figure 7. The tables and partitions in an example storage account

#### NOTE

Azure table storage also adds a timestamp field to each entity. The timestamp field is maintained by table storage and is updated each time the entity is modified and written back to a partition. The table storage service uses this field to implement optimistic concurrency. (Each time an application writes an entity back to table storage, the table storage service compares the value of the timestamp in the entity that's being written with the value that's held in table storage. If the values are different, it means that another application must have modified the entity since it was last retrieved, and the write operation fails. Don't modify this field in your own code, and don't specify a value for this field when you create a new entity.)

Azure table storage uses the partition key to determine how to store the data. 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 a single server in an Azure datacenter to help ensure that queries that retrieve data from a single partition run quickly. However, different partitions can be distributed across multiple servers. Additionally, a single server can host multiple partitions if these partitions are limited in size.

Consider the following points when you design your entities for Azure table storage:

- The selection of partition key and row key values should be driven by the way in which 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, the partition key might require Azure table storage to scan every partition for your data.

**TIP**

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 comprising 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](#).
- If you generate partition keys by using a monotonic increasing or decreasing sequence (such as "0001", "0002", "0003", and so on) and each partition only contains a limited amount of data, then Azure table storage can physically group these partitions together on the same server. This mechanism 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 focused on a single server because all insertions of new entities are likely to be concentrated at one end or the other of the contiguous ranges. It can also reduce scalability. To spread the load more evenly across servers, consider hashing the partition key to make the sequence more random.
- Azure table storage supports transactional operations for entities that belong to the same partition. This means that 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 as described by the [Data consistency primer](#). For more information about table storage and transactions, go to the page [Performing entity group transactions](#) on the Microsoft website.
- Give careful attention to the granularity of the partition key because of the following reasons:
  - Using the same partition key for every entity causes the table storage service to create a single large partition that's held on one server. This prevents it from scaling out and instead focuses the load on a single server. As a result, this approach is only suitable for systems that manage a small number of entities. However, this approach 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 (depending on the size of the entities). 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 to generate the partition keys might help to optimize these queries.

- Sharing the partition key across a subset of entities makes it possible for you 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 additional information about partitioning data in Azure table storage, see the article [Azure storage table design guide](#) on the Microsoft website.

## Partitioning Azure blob storage

Azure blob storage makes it possible to hold large binary objects--currently up to 5 TB in size for block blobs or 1 TB for page blobs. (For the most recent information, go to the page [Azure Storage scalability and performance targets](#) on the Microsoft website.) Use block blobs in scenarios such as streaming where 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. This means each blob can have its own partition if load on the blob demands it. 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.

The actions of writing a single block (block blob) or page (page blob) 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.

Azure blob storage targets transfer rates of up to 60 MB per second or 500 requests per second for each blob. If you anticipate surpassing these limits, and the blob data is relatively static, then consider replicating blobs by using the Azure Content Delivery Network. For more information, see the page [Azure Content Delivery Network](#) on the Microsoft website. For additional guidance and considerations, see [Using Azure Content Delivery Network](#).

## 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 hotspot. 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 impact 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 strategies for 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 on the Microsoft website on the page [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 strategies for 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 strategies for 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. A full description of these capabilities is available on the page [What is Azure Search?](#) on the Microsoft website.

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 the page [Service limits in Azure Search](#) on the Microsoft website.

#### 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 details, 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 towards 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 strategies for Azure Redis Cache

Azure Redis Cache provides a shared caching service in the cloud that's based on the Redis key-value data store. As its name implies, Azure Redis Cache is intended as a caching solution. Use it only for holding transient data and not as a permanent data store. Applications that utilize Azure Redis Cache should be able to continue functioning if the cache is unavailable. Azure Redis Cache 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, go to the page [Azure Redis Cache](#) on the Microsoft website.

Partitioning a Redis data store involves splitting the data across instances of the Redis service. Each instance constitutes a single partition. Azure Redis Cache abstracts the Redis services behind a façade and does not 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 very simple, but if the partitioning scheme changes (for example, if additional Azure Redis Cache instances are created), client applications might need to be reconfigured.

Native Redis (not Azure Redis Cache) 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 re-partitioned) without requiring that you reconfigure the clients.

## IMPORTANT

Azure Redis Cache does not currently support Redis clustering. If you want to implement this approach with Azure, then you must implement your own Redis servers by installing Redis on a set of Azure virtual machines and configuring them manually. The page [Running Redis on a CentOS Linux VM in Azure](#) on the Microsoft website walks through an example that shows you how to build and configure a Redis node running as an Azure VM.

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 Redis Cache:

- Azure Redis Cache 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 Redis Cache 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.

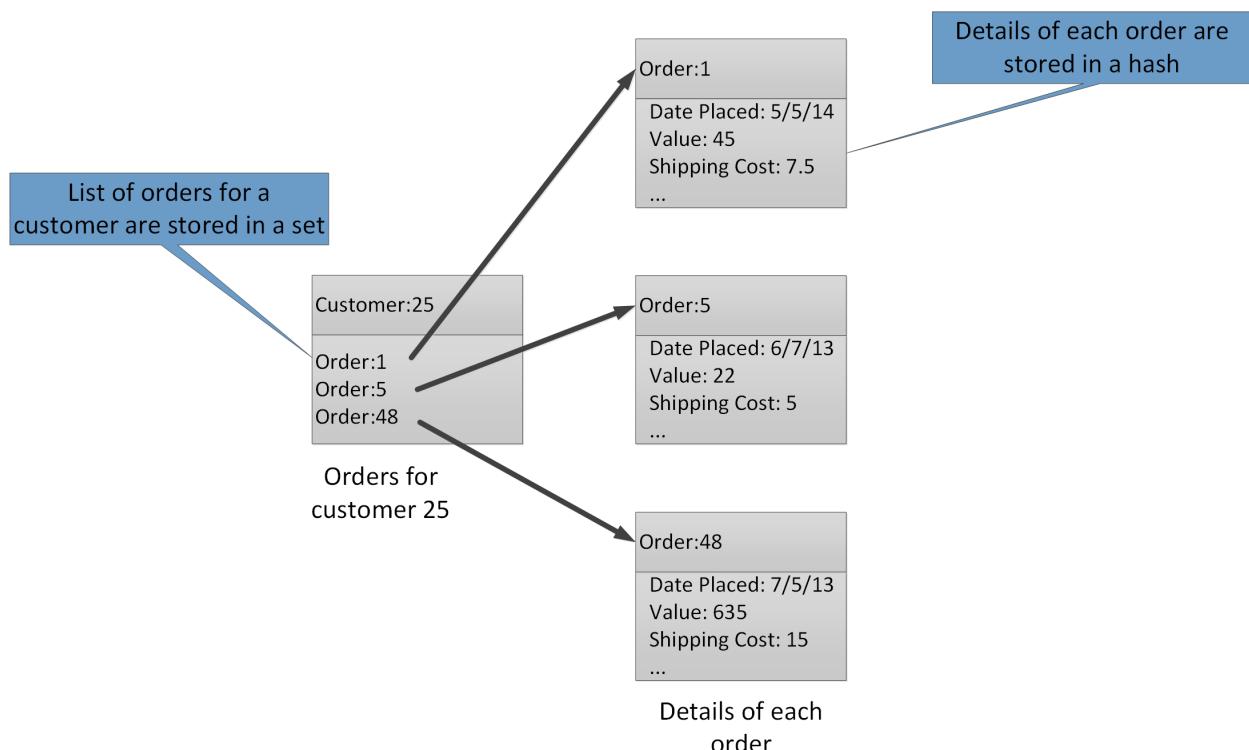


Figure 8. Suggested structure in Redis storage for recording customer orders and their details

**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 Redis Cache. 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 Redis Cache 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 Redis Cache to remove data if space is at a premium.

**NOTE**

When you use Azure Redis cache, you specify the maximum size of the cache (from 250 MB to 53 GB) by selecting the appropriate pricing tier. However, after an Azure Redis Cache 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 Strategies for 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 strategies for 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](#).

## Rebalancing partitions

As a system matures and you understand the usage patterns better, you might have to adjust the partitioning scheme. For example, individual partitions might start attracting a disproportionate volume of traffic and become hot, leading to excessive contention. Additionally, you might have underestimated the volume of data in some partitions, causing you to approach the limits of the storage capacity in these partitions. Whatever the cause, it is sometimes necessary to rebalance partitions to spread the load more evenly.

In some cases, data storage systems that don't publicly expose how data is allocated to servers can automatically rebalance partitions within the limits of the resources available. In other situations, rebalancing is an administrative task that consists of two stages:

1. Determining the new partitioning strategy to ascertain:
  - Which partitions might need to be split (or possibly combined).
  - How to allocate data to these new partitions by designing new partition keys.
2. Migrating the affected data from the old partitioning scheme to the new set of partitions.

### NOTE

The mapping of database collections to servers is transparent, but you can still reach the storage capacity and throughput limits of a Cosmos DB account. If this happens, you might need to redesign your partitioning scheme and migrate the data.

Depending on the data storage technology and the design of your data storage system, you might be able to migrate data between partitions while they are in use (online migration). If this isn't possible, you might need to make the affected partitions temporarily unavailable while the data is relocated (offline migration).

## Offline migration

Offline migration is arguably the simplest approach because it reduces the chances of contention occurring. Don't make any changes to the data while it is being moved and restructured.

Conceptually, this process includes the following steps:

1. Mark the shard offline.
2. Split-merge and move the data to the new shards.
3. Verify the data.
4. Bring the new shards online.
5. Remove the old shard.

To retain some availability, you can mark the original shard as read-only in step 1 rather than making it unavailable. This allows applications to read the data while it is being moved but not to change it.

## Online migration

Online migration is more complex to perform but less disruptive to users because data remains available during the entire procedure. The process is similar to that used by offline migration, except that the original shard is not marked offline (step 1). Depending on the granularity of the migration process (for example, whether it's done item by item or 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 shard and the new shard).

For an example of a solution that supports online migration, see the article [Scaling using the Elastic Database split-merge tool](#) on the Microsoft website.

## Related patterns and guidance

When considering strategies for implementing data consistency, the following patterns might also be relevant to your scenario:

- The [Data consistency primer](#) page on the Microsoft website describes strategies for maintaining consistency in a distributed environment such as the cloud.
- The [Data partitioning guidance](#) page on the Microsoft website provides a general overview of how to design partitions to meet various criteria in a distributed solution.
- The [sharding pattern](#) as described on the Microsoft website summarizes some common strategies for sharding data.
- The [index table pattern](#) as described on the Microsoft website illustrates 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](#) as described on the Microsoft website describes how to generate pre-populated 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.
- The [Using Azure Content Delivery Network](#) article on the Microsoft website provides additional guidance on configuring and using Content Delivery Network with Azure.

## More information

- The page [What is Azure SQL Database?](#) on the Microsoft website provides detailed documentation that describes how to create and use SQL databases.
- The page [Elastic Database features overview](#) on the Microsoft website provides a comprehensive introduction to Elastic Database.
- The page [Scaling using the Elastic Database split-merge tool](#) on the Microsoft website contains information about using the split-merge service to manage Elastic Database shards.

- The page [Azure storage scalability and performance targets](#) on the Microsoft website documents the current sizing and throughput limits of Azure Storage.
- The page [Performing entity group transactions](#) on the Microsoft website provides detailed information about implementing transactional operations over entities that are stored in Azure table storage.
- The article [Azure Storage table design guide](#) on the Microsoft website contains detailed information about partitioning data in Azure table storage.
- The page [Using Azure Content Delivery Network](#) on the Microsoft website describes how to replicate data that's held in Azure blob storage by using the Azure Content Delivery Network.
- The page [What is Azure Search?](#) on the Microsoft website provides a full description of the capabilities that are available in Azure Search.
- The page [Service limits in Azure Search](#) on the Microsoft website contains information about the capacity of each instance of Azure Search.
- The page [Supported data types \(Azure Search\)](#) on the Microsoft website summarizes the data types that you can use in searchable documents and indexes.
- The page [Azure Redis Cache](#) on the Microsoft website provides an introduction to Azure Redis Cache.
- The [Partitioning: how to split data among multiple Redis instances](#) page on the Redis website provides information about how to implement partitioning with Redis.
- The page [Running Redis on a CentOS Linux VM in Azure](#) on the Microsoft website walks through an example that shows you how to build and configure a Redis node running as an Azure VM.
- The [Data types](#) page on the Redis website describes the data types that are available with Redis and Azure Redis Cache.

# Monitoring and diagnostics

9/28/2018 • 69 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 utilize 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 time-stamped.

### 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:

$$\text{Availability} = ((\text{Total Time} - \text{Total Downtime}) / \text{Total Time}) * 100$$

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.
- If one authenticated account repeatedly tries to access a prohibited resource during a specified period.
- If 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, time-stamped 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 5 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 time-stamped.

### **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. Non-repudiation is an important factor in many e-business systems to help maintain trust 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 utilize. 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 impact 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.

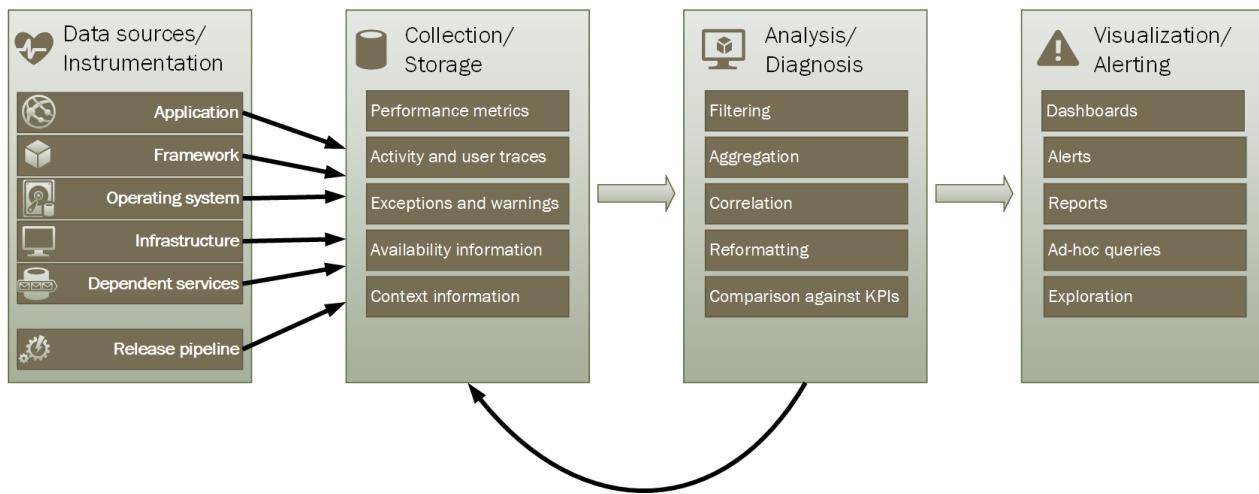


Figure 1. The stages in the monitoring and diagnostics pipeline

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 make use of 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 time-stamped 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 time stamps 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 time stamps. 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.

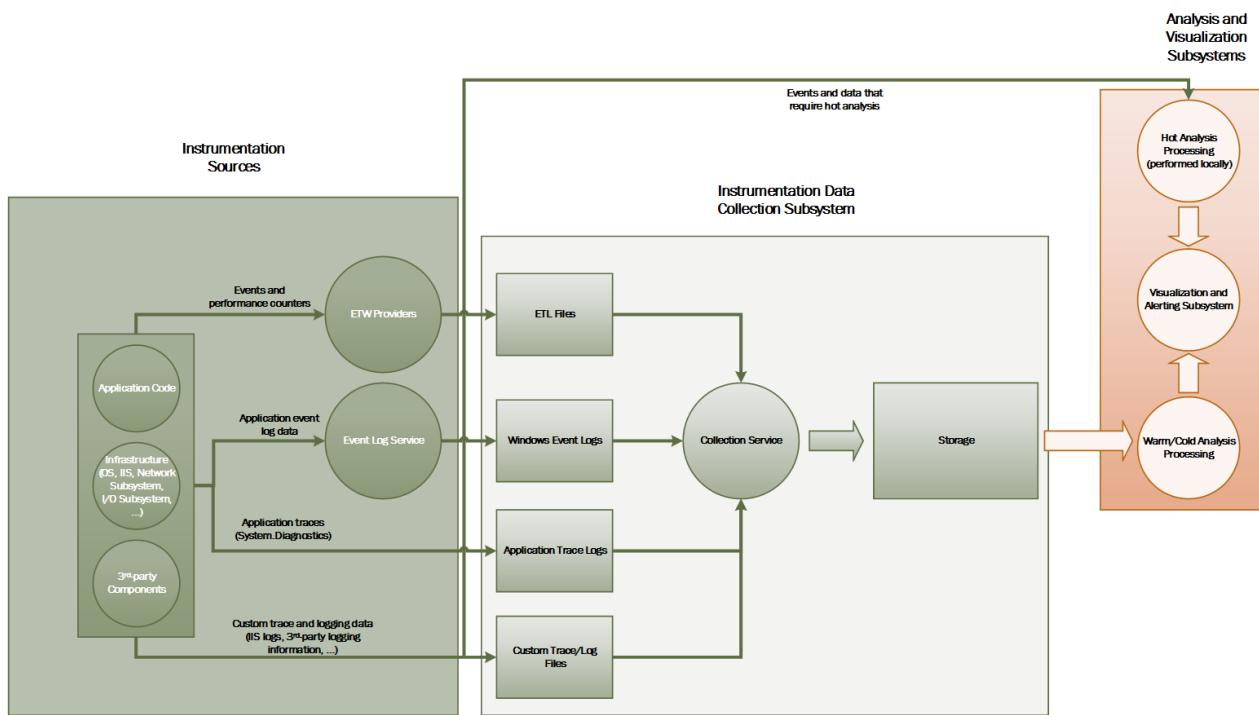


Figure 2. Collecting instrumentation data

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.

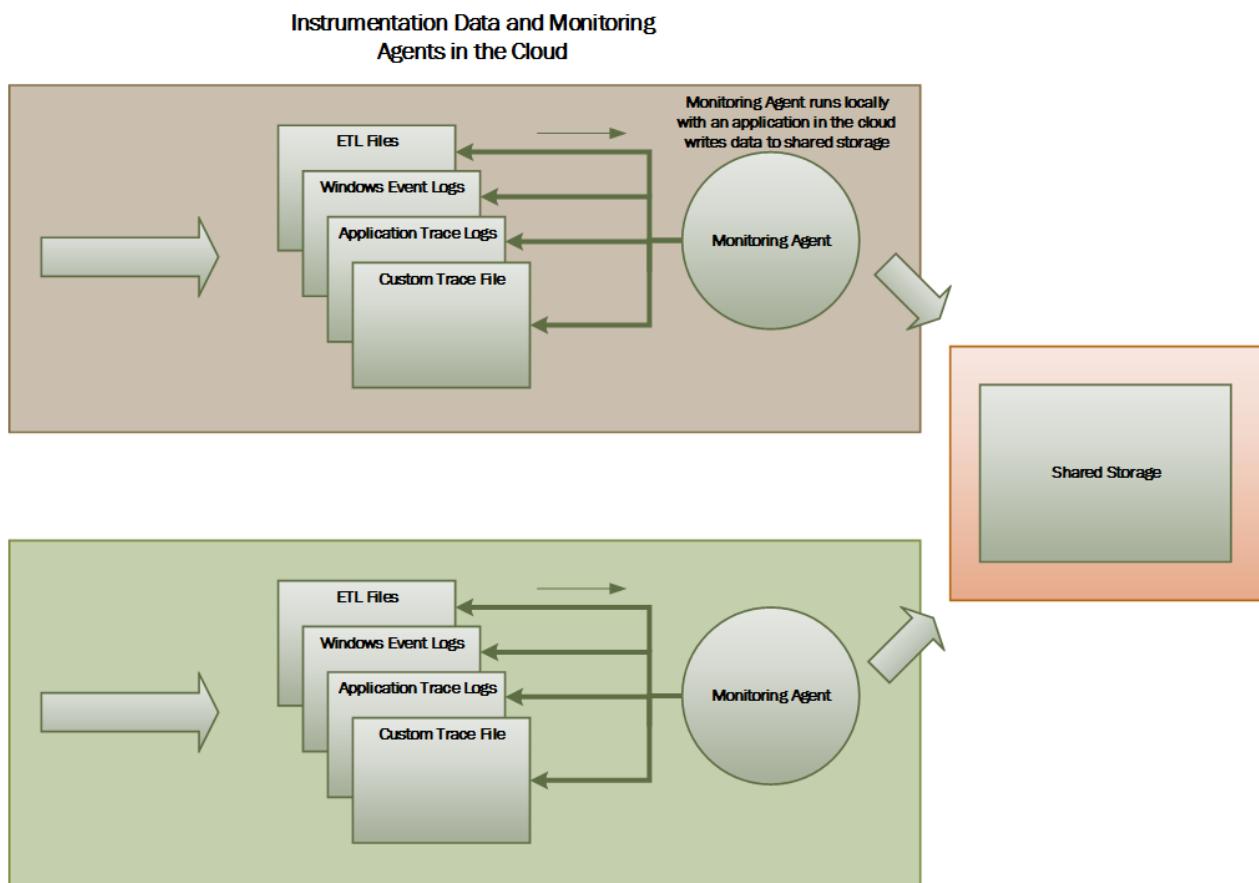


Figure 3. Using a monitoring agent to pull information and write to shared storage

**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.

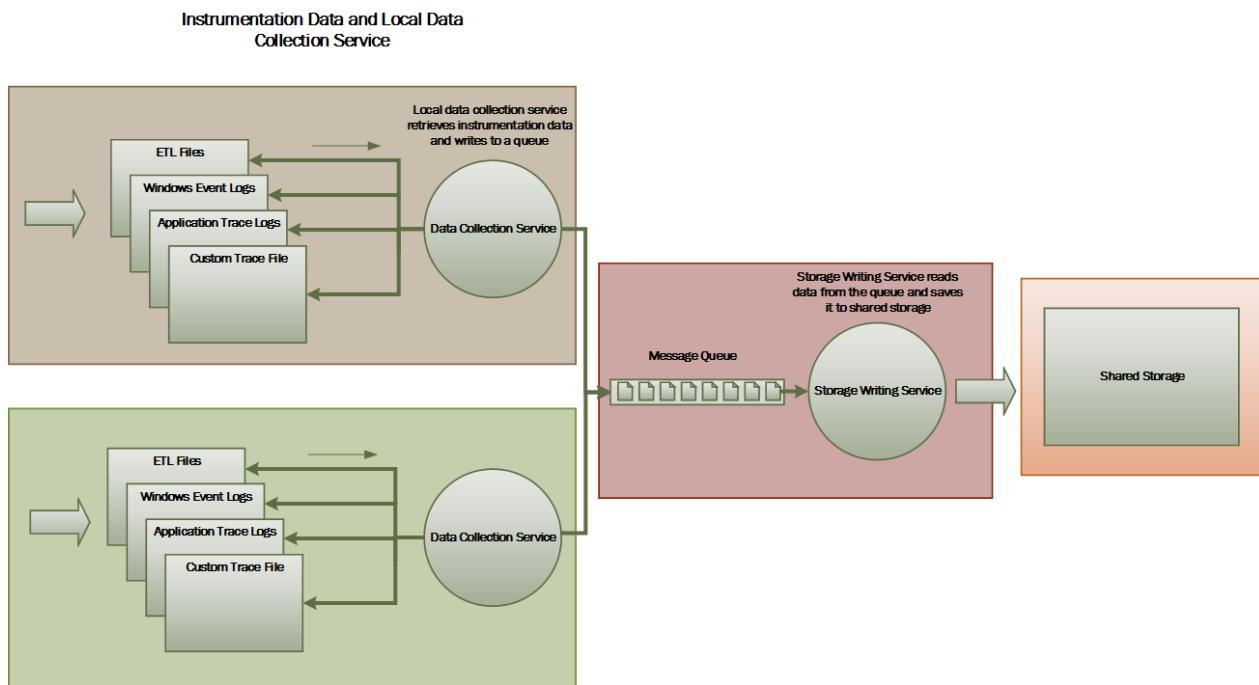


Figure 4. Using a queue to buffer instrumentation data

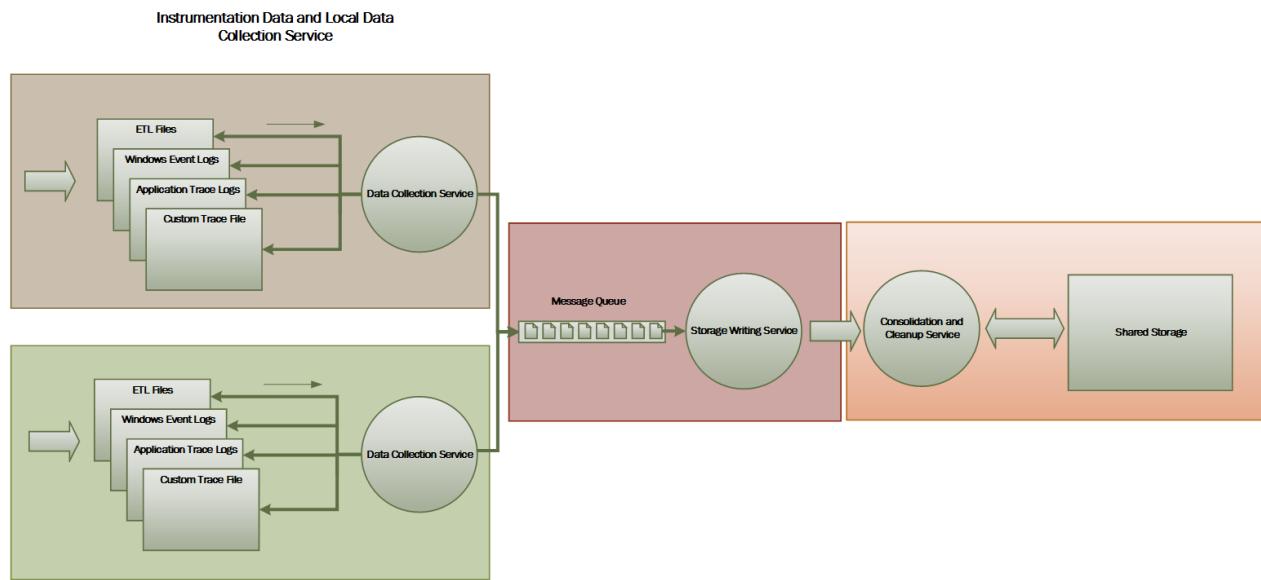
The local data-collection service can add data to a queue immediately after it's received. The queue acts as a buffer, and the storage writing service can retrieve and write the data at its own pace. By default, a queue operates on a first-in, first-out basis. But you can prioritize messages to accelerate 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.

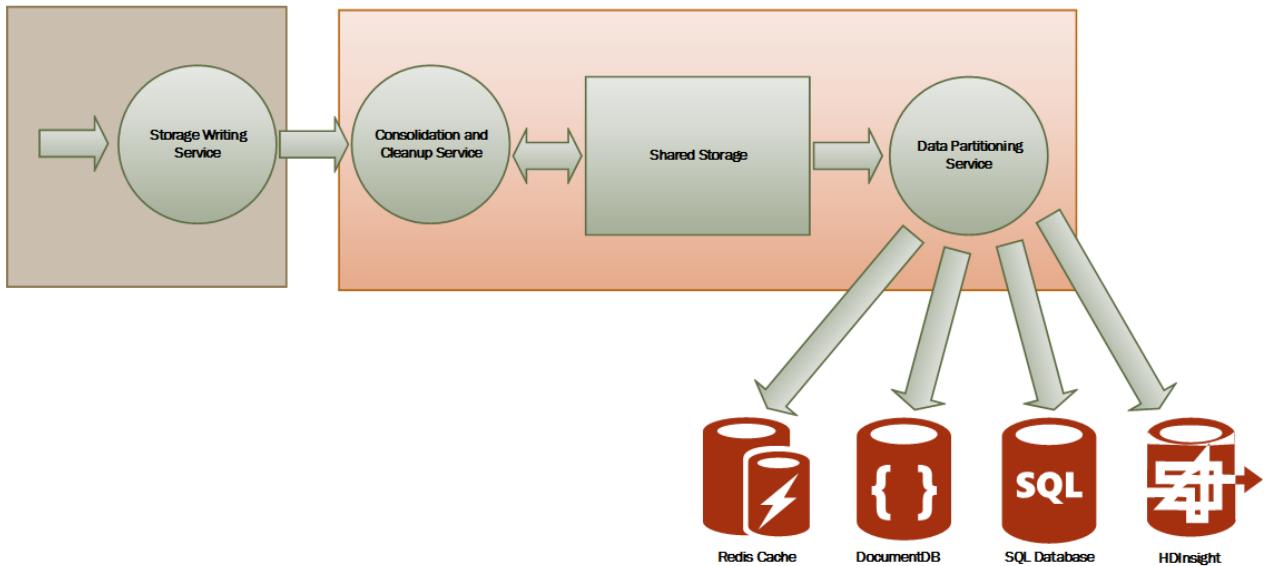


Figure 6. Partitioning data according to analytical and storage requirements

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 pre-processing 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.

## More information

- [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#)
- [Azure: Telemetry Basics and Troubleshooting](#)
- [Enabling Diagnostics in Azure Cloud Services and Virtual Machines](#)
- [Azure Redis Cache, Azure Cosmos DB, and HDInsight](#)
- [How to use Service Bus queues](#)
- [SQL Server business intelligence in Azure Virtual Machines](#)
- [Receive alert notifications](#) and [Track service health](#)
- [Application Insights](#)

# Naming conventions

10/24/2018 • 9 minutes to read • [Edit Online](#)

This article is a summary of the naming rules and restrictions for Azure resources and a baseline set of recommendations for naming conventions. You can use these recommendations as a starting point for your own conventions specific to your needs.

The choice of a name for any resource in Microsoft Azure is important because:

- It is difficult to change a name later.
- Names must meet the requirements of their specific resource type.

Consistent naming conventions make resources easier to locate. They can also indicate the role of a resource in a solution.

The key to success with naming conventions is establishing and following them across your applications and organizations.

## Naming subscriptions

When naming Azure subscriptions, verbose names make understanding the context and purpose of each subscription clear. When working in an environment with many subscriptions, following a shared naming convention can improve clarity.

A recommended pattern for naming subscriptions is:

```
<Company> <Department (optional)> <Product Line (optional)> <Environment>
```

- Company would usually be the same for each subscription. However, some companies may have child companies within the organizational structure. These companies may be managed by a central IT group. In these cases, they could be differentiated by having both the parent company name (*Contoso*) and child company name (*Northwind*).
- Department is a name within the organization that contains a group of individuals. This item within the namespace is optional.
- Product line is a specific name for a product or function that is performed from within the department. This is generally optional for internal-facing services and applications. However, it is highly recommended to use for public-facing services that require easy separation and identification (such as for clear separation of billing records).
- Environment is the name that describes the deployment lifecycle of the applications or services, such as Dev, QA, or Prod.

COMPANY	DEPARTMENT	PRODUCT LINE OR SERVICE	ENVIRONMENT	FULL NAME
Contoso	SocialGaming	AwesomeService	Production	Contoso SocialGaming AwesomeService Production
Contoso	SocialGaming	AwesomeService	Dev	Contoso SocialGaming AwesomeService Dev

COMPANY	DEPARTMENT	PRODUCT LINE OR SERVICE	ENVIRONMENT	FULL NAME
Contoso	IT	InternalApps	Production	Contoso IT InternalApps Production
Contoso	IT	InternalApps	Dev	Contoso IT InternalApps Dev

For more information on how to organize subscriptions for larger enterprises, see [Azure enterprise scaffold - prescriptive subscription governance](#).

## Use affixes to avoid ambiguity

When naming resources in Azure, it is recommended to use common prefixes or suffixes to identify the type and context of the resource. While all the information about type, metadata, context, is available programmatically, applying common affixes simplifies visual identification. When incorporating affixes into your naming convention, it is important to clearly specify whether the affix is at the beginning of the name (prefix) or at the end (suffix).

For instance, here are two possible names for a service hosting a calculation engine:

- SvcCalculationEngine (prefix)
- CalculationEngineSvc (suffix)

Affixes can refer to different aspects that describe the particular resources. The following table shows some examples typically used.

ASPECT	EXAMPLE	NOTES
Environment	dev, prod, QA	Identifies the environment for the resource
Location	uw (US West), ue (US East)	Identifies the region into which the resource is deployed
Instance	01, 02	For resources that have more than one named instance (web servers, etc.).
Product or Service	service	Identifies the product, application, or service that the resource supports
Role	sql, web, messaging	Identifies the role of the associated resource

When developing a specific naming convention for your company or projects, it is important to choose a common set of affixes and their position (suffix or prefix).

## Naming rules and restrictions

Each resource or service type in Azure enforces a set of naming restrictions and scope; any naming convention or pattern must adhere to the requisite naming rules and scope. For example, while the name of a VM maps to a DNS name (and is thus required to be unique across all of Azure), the name of a VNET is scoped to the Resource Group that it is created within.

In general, avoid having any special characters (- or \_) as the first or last character in any name. These

characters will cause most validation rules to fail.

## General

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Resource Group	Subscription	1-90	Case insensitive	Alphanumeric, underscore, parentheses, hyphen, period (except at end), and Unicode characters that match the regex documented <a href="#">here</a> .	<service short name>-<environment>-rg	profx-prod-rg
Availability Set	Resource Group	1-80	Case insensitive	Alphanumeric, underscore, and hyphen	<service-short-name>-<context>-as	profx-sql-as
Tag	Associated Entity	512 (name), 256 (value)	Case insensitive	Alphanumeric	"key" : "value"	"department" : "Central IT"

## Compute

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Virtual Machine	Resource Group	1-15 (Windows), 1-64 (Linux)	Case insensitive	Alphanumeric and hyphen	<name>-<role>-vm<number>	profx-sql-vm1
Function App	Global	1-60	Case insensitive	Alphanumeric and hyphen	<name>-func	calcprofit-func

### NOTE

Virtual machines in Azure have two distinct names: virtual machine name, and host name. When you create a VM in the portal, the same name is used for both the host name, and the virtual machine resource name. The restrictions above are for the host name. The actual resource name can have up to 64 characters.

## Storage

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Storage account name (data)	Global	3-24	Lowercase	Alphanumeric	<globally unique name> <number> (use a function to calculate a unique guid for naming storage accounts)	profodata001
Storage account name (disks)	Global	3-24	Lowercase	Alphanumeric	<vm name without profxsql001st0 hyphens>st<number>	
Container name	Storage account	3-63	Lowercase	Alphanumeric and hyphen	<context>	logs
Blob name	Container	1-1024	Case sensitive	Any URL characters	<variable based on blob usage>	<variable based on blob usage>
Queue name	Storage account	3-63	Lowercase	Alphanumeric and hyphen	<service short name>-<context>-<num>	awesomeservice-messages-001
Table name	Storage account	3-63	Case insensitive	Alphanumeric	<service short name> <context>	awesomeservicelogs
File name	Storage account	3-63	Lowercase	Alphanumeric	<variable based on blob usage>	<variable based on blob usage>
Data Lake Store	Global	3-24	Lowercase	Alphanumeric	<name>dls	telemetrydls

## Networking

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Virtual Network (VNet)	Resource Group	2-64	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service short name>-vnet	profx-vnet
Subnet	Parent VNet	2-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<descriptive context>	web

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Network Interface	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<vmname>-nic<num>	profx-sql1-vm1-nic1
Network Security Group	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service short name>-<context>-nsg	profx-app-nsg
Network Security Group Rule	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<descriptive context>	sql-allow
Public IP Address	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<vm or service name>-pip	profx-sql1-vm1-pip
Load Balancer	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service or role>-lb	profx-lb
Load Balanced Rules Config	Load Balancer	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<descriptive context>	http
Azure Application Gateway	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service or role>-agw	profx-agw
Traffic Manager Profile	Resource Group	1-63	Case insensitive	Alphanumeric, hyphen, and period	<descriptive context>	app1

## Containers

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Container Registry	Global	5-50	Case insensitive	Alphanumeric	<service short name>registry	app1registry

## Organize resources with tags

The Azure Resource Manager supports tagging entities with arbitrary text strings to identify context and streamline automation. For example, the tag "sqlVersion"="sql2014ee" could identify VMs running SQL Server 2014 Enterprise Edition. Tags should be used to augment and enhance context along side of the naming conventions chosen.

## TIP

One other advantage of tags is that tags span resource groups, allowing you to link and correlate entities across disparate deployments.

Each resource or resource group can have a maximum of **15** tags. The tag name is limited to 512 characters, and the tag value is limited to 256 characters.

For more information on resource tagging, refer to [Using tags to organize your Azure resources](#).

Some of the common tagging use cases are:

- **Billing**; Grouping resources and associating them with billing or charge back codes.
- **Service Context Identification**; Identify groups of resources across Resource Groups for common operations and grouping
- **Access Control and Security Context**; Administrative role identification based on portfolio, system, service, app, instance, etc.

## TIP

Tag early - tag often. Better to have a baseline tagging scheme in place and adjust over time rather than having to retrofit after the fact.

An example of some common tagging approaches:

TAG NAME	KEY	EXAMPLE	COMMENT
Bill To / Internal Chargeback ID	billTo	IT-Chargeback-1234	An internal I/O or billing code
Operator or Directly Responsible Individual (DRI)	managedBy	joe@contoso.com	Alias or email address
Project Name	projectName	myproject	Name of the project or product line
Project Version	projectVersion	3.4	Version of the project or product line
Environment	environment	<Production, Staging, QA >	Environmental identifier
Tier	tier	Front End, Back End, Data	Tier or role/context identification
Data Profile	dataProfile	Public, Confidential, Restricted, Internal	Sensitivity of data stored in the resource

## Tips and tricks

Some types of resources may require additional care on naming and conventions.

### Virtual machines

Especially in larger topologies, carefully naming virtual machines streamlines identifying the role and purpose of

each machine, and enabling more predictable scripting.

## Storage accounts and storage entities

There are two primary use cases for storage accounts - backing disks for VMs, and storing data in blobs, queues and tables. Storage accounts used for VM disks should follow the naming convention of associating them with the parent VM name (and with the potential need for multiple storage accounts for high-end VM SKUs, also apply a number suffix).

### TIP

Storage accounts - whether for data or disks - should follow a naming convention that allows for multiple storage accounts to be leveraged (i.e. always using a numeric suffix).

It's possible to configure a custom domain name for accessing blob data in your Azure Storage account. The default endpoint for the Blob service is <https://<name>.blob.core.windows.net>.

But if you map a custom domain (such as [www.contoso.com](http://www.contoso.com)) to the blob endpoint for your storage account, you can also access blob data in your storage account by using that domain. For example, with a custom domain name,

<https://mystorage.blob.core.windows.net/mycontainer/myblob> could be accessed as

<https://www.contoso.com/mycontainer/myblob>.

For more information about configuring this feature, refer to [Configure a custom domain name for your Blob storage endpoint](#).

For more information on naming blobs, containers and tables, refer to the following list:

- [Naming and Referencing Containers, Blobs, and Metadata](#)
- [Naming Queues and Metadata](#)
- [Naming Tables](#)

A blob name can contain any combination of characters, but reserved URL characters must be properly escaped.

Avoid blob names that end with a period (.), a forward slash (/), or a sequence or combination of the two. By convention, the forward slash is the **virtual** directory separator. Do not use a backward slash (\) in a blob name.

The client APIs may allow it, but then fail to hash properly, and the signatures will not match.

It is not possible to modify the name of a storage account or container after it has been created. If you want to use a new name, you must delete it and create a new one.

### TIP

We recommend that you establish a naming convention for all storage accounts and types before embarking on the development of a new service or application.

# Transient fault handling

9/28/2018 • 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 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, performance and availability of the application and its components is typically maintained through expensive and often under-used hardware redundancy, and components and resources are located close to each other. While this 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 a higher overall availability by using shared resources, redundancy, automatic failover, and dynamic resource allocation across a huge number of 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, very heavy traffic loads may slow communication and cause intermittent connection failures.

## Challenges

Transient faults can have a huge impact 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 handing 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 one is available unless you have specific and well-understood requirements that mean a different retry behavior is 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 cancelling 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 times 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 times 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 extremely short, perhaps caused by 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 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 very large 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 that defined in the SLA
- Over-aggressive retry strategies, which have too short intervals or too many retries, 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 of 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 in 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 non-transient 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 (you should not, of course, 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 non-transient 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 impacting 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 very 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 multi-step 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 multi-step 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](#)

# Retry guidance for specific services

9/28/2018 • 40 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 ADAL library	Embedded into ADAL 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
Redis Cache	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 Active Directory Authentication Library (ADAL). To avoid unexpected lockouts, we recommend that third party libraries and application code do **not** retry failed connections, but allow ADAL to handle retries.

### Retry usage guidance

Consider the following guidelines when using Azure Active Directory:

- When possible, use the ADAL 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.
- An exponential back-off policy is recommended for use in batch scenarios with Azure Active Directory.

Consider starting with the following settings for retrying operations. These are general purpose settings, 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	60 sec	ExponentialBackoff	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

## More information

- [Azure Active Directory Authentication Libraries](#)

## Cosmos DB

Cosmos DB is a fully-managed multi-model database that supports schema-less JSON data. It offers configurable and reliable performance, native JavaScript transactional processing, and is built for the cloud with elastic scale.

### Retry mechanism

The `DocumentClient` class automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. 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 `DocumentClientException`.

### Policy configuration

The following table shows the default settings for the `RetryOptions` class.

SETTING	DEFAULT VALUE	DESCRIPTION
MaxRetryAttemptsOnThrottledRequests	9	The maximum number of retries if the request fails because Cosmos DB applied rate limiting on the client.
MaxRetryWaitTimeInSeconds	30	The maximum retry time in seconds.

### Example

```
DocumentClient client = new DocumentClient(new Uri(endpoint), authKey);  
var options = client.ConnectionPolicy.RetryOptions;  
options.MaxRetryAttemptsOnThrottledRequests = 5;  
options.MaxRetryWaitTimeInSeconds = 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 hyper-scale 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`.

### 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 Redis Cache

Azure Redis Cache 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).Milliseconds;
var maxDeltaBackOffInMilliseconds = TimeSpan.FromSeconds(20).Milliseconds;
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. Note that 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 Redis Cache](#). 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 Redis Cache:

- 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).Milliseconds; // 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 [RetryPolicy](#) base class. All of the Service Bus clients expose a **RetryPolicy** property that can be set to one of the implementations of the **RetryPolicy** base class. The built-in implementations are:

- The [RetryExponential Class](#). This exposes properties that control the back-off interval, the retry count, and the **TerminationTimeBuffer** property that is used to limit the total time for the operation to complete.
- The [NoRetry Class](#). This is used when retries at the Service Bus API level are not required, such as when retries are managed by another process as part of a batch or multiple step operation.

Service Bus actions can return a range of exceptions, as listed in [Service Bus messaging exceptions](#). The list provides information about which if these indicate that retrying the operation is appropriate. For example, a **ServerBusyException** indicates that the client should wait for a period of time, then retry the operation. The occurrence of a **ServerBusyException** also causes Service Bus to switch to a different mode, in which an extra 10-second delay is added to the computed retry delays. This mode is reset after a short period.

The exceptions returned from Service Bus expose the **IsTransient** property that indicates if the client should retry the operation. The built-in **RetryExponential** policy relies on the **IsTransient** property in the **MessagingException** class, which is the base class for all Service Bus exceptions. If you create custom implementations of the **RetryPolicy** base class you 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.

### Policy configuration

Retry policies are set programmatically, and can be set as a default policy for a **NamespaceManager** and for a **MessagingFactory**, or individually for each messaging client. To set the default retry policy for a messaging session you set the **RetryPolicy** of the **NamespaceManager**.

```
namespaceManager.Settings.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                                               maxBackoff: TimeSpan.FromSeconds(30),
                                                               maxRetryCount: 3);
```

To set the default retry policy for all clients created from a messaging factory, you set the **RetryPolicy** of the **MessagingFactory**.

```
messagingFactory.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                                               maxBackoff: TimeSpan.FromSeconds(30),
                                                               maxRetryCount: 3);
```

To set the retry policy for a messaging client, or to override its default policy, you set its **RetryPolicy** property using an instance of the required policy class:

```
client.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                                               maxBackoff: TimeSpan.FromSeconds(30),
                                                               maxRetryCount: 3);
```

The retry policy cannot be set at the individual operation level. It applies to all operations for the messaging client. The following table shows the default settings for the built-in retry policy.

SETTING	DEFAULT VALUE	MEANING
Policy	Exponential	Exponential back-off.
MinimalBackoff	0	Minimum back-off interval. This is added to the retry interval computed from deltaBackoff.

SETTING	DEFAULT VALUE	MEANING
MaximumBackoff	30 seconds	Maximum back-off interval. MaximumBackoff is used if the computed retry interval is greater than MaxBackoff.
DeltaBackoff	3 seconds	Back-off interval between retries. Multiples of this timespan will be used for subsequent retry attempts.
TimeBuffer	5 seconds	The termination time buffer associated with the retry. Retry attempts will be abandoned if the remaining time is less than TimeBuffer.
MaxRetryCount	10	The maximum number of retries.
ServerBusyBaseSleepTime	10 seconds	If the last exception encountered was <b>ServerBusyException</b> , this value will be added to the computed retry interval. This value cannot be changed.

### 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, which 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 following settings for retrying operations. These are general purpose settings, 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 msec. Attempt 4: Delay ~13 msec.

\* 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.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 following settings for retrying operations. These are general purpose settings, 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	ExponentialBackoff	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 200ms delay.
        onRetry: (exception, attempt) =>
    {
        // Capture some info for logging/telemetry.
        logger.LogWarning($"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 are general purpose settings, 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 table and blob storage, files, and storage queues.

### 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](#).

There are different implementations of the interface. Storage clients can choose from policies specifically designed for accessing tables, blobs, and queues. Each implementation uses a different retry strategy that essentially defines the retry interval and other details.

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 details, 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 hang.

SETTING	DEFAULT VALUE	MEANING
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 <b>PrimaryThenSecondary</b> 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.WindowsAzure.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 following settings for retrying operations. These are general purpose settings, 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 auto-flushing 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.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.RetryPolicies;
using Microsoft.WindowsAzure.Storage.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 non-standard 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.
- 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](#)

# Performance antipatterns for cloud applications

6/23/2017 • 2 minutes to read • [Edit Online](#)

A *performance antipattern* is a common practice that is likely to cause scalability problems when an application is under pressure.

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 very difficult in a test environment to simulate real users, 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 tests are 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 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.

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.
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.

ANTIPATTERN	DESCRIPTION
No Caching	Failing to cache data.
Synchronous I/O	Blocking the calling thread while I/O completes.

# Busy Database antipattern

9/28/2018 • 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.

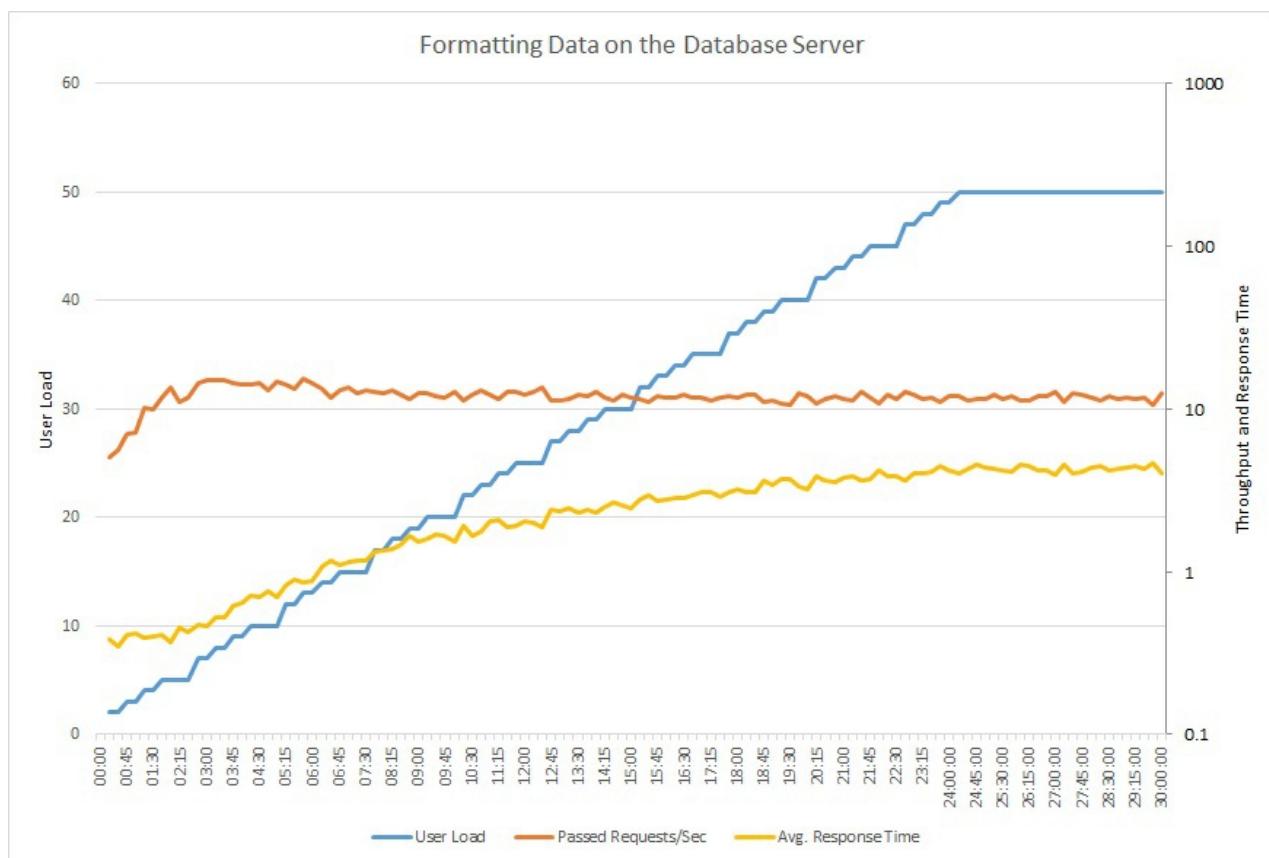
If the volume of database activity is low or response times are relatively fast, then a busy database is unlikely to be a performance problem.

## Example diagnosis

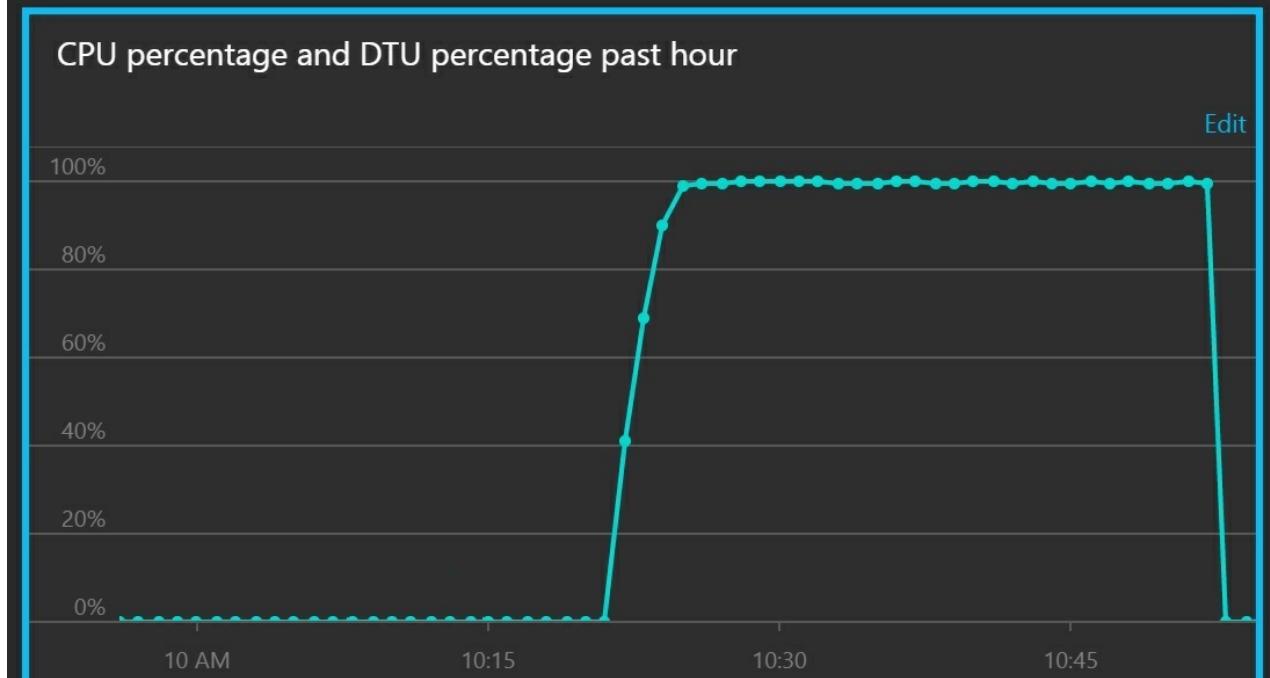
The following sections apply these steps to the sample application described earlier.

### 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. Note that a logarithmic scale is used for those two metrics.



The next graph shows CPU utilization and DTUs as a percentage of service quota. DTUs provides a measure of how much processing the database performs. The graph shows that CPU and DTU utilization both quickly reached 100%.



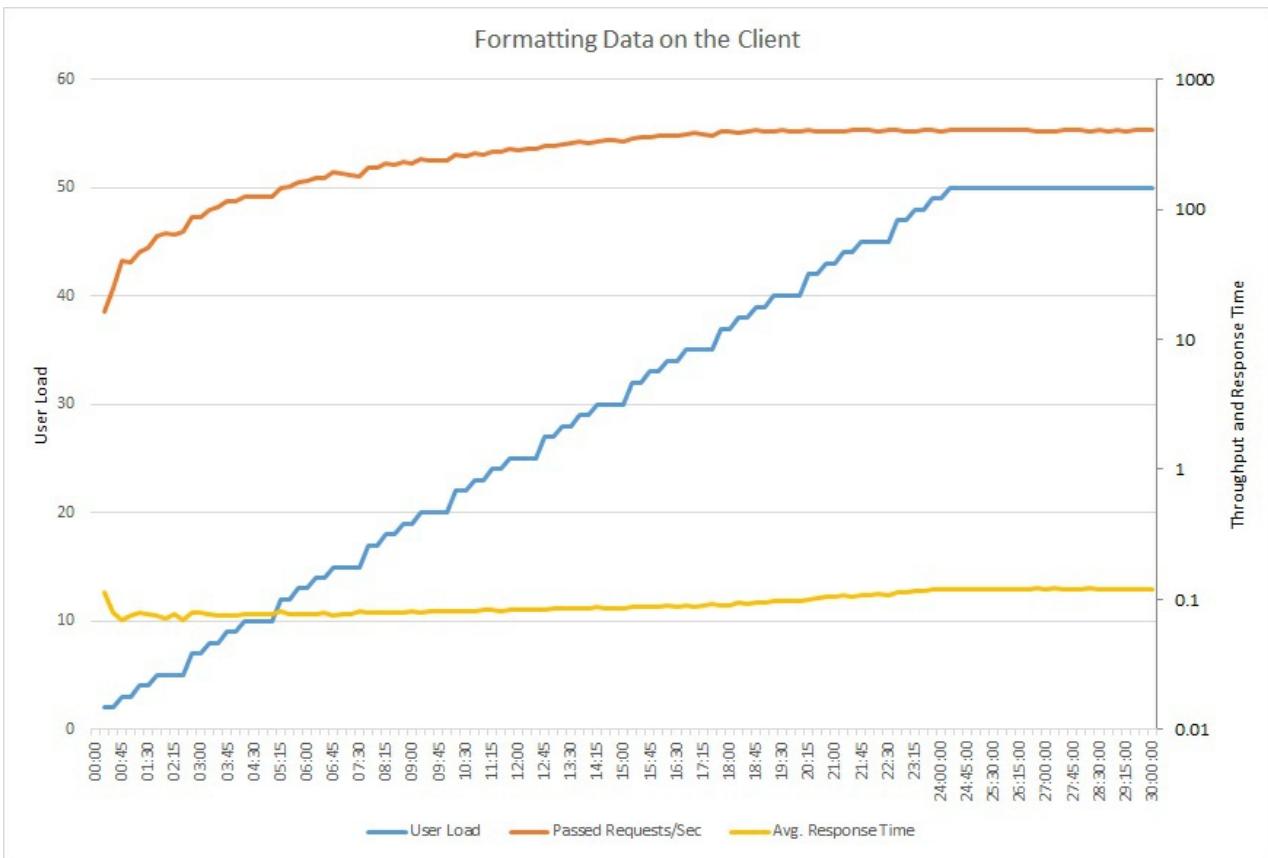
#### Examine the work performed by the database

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.

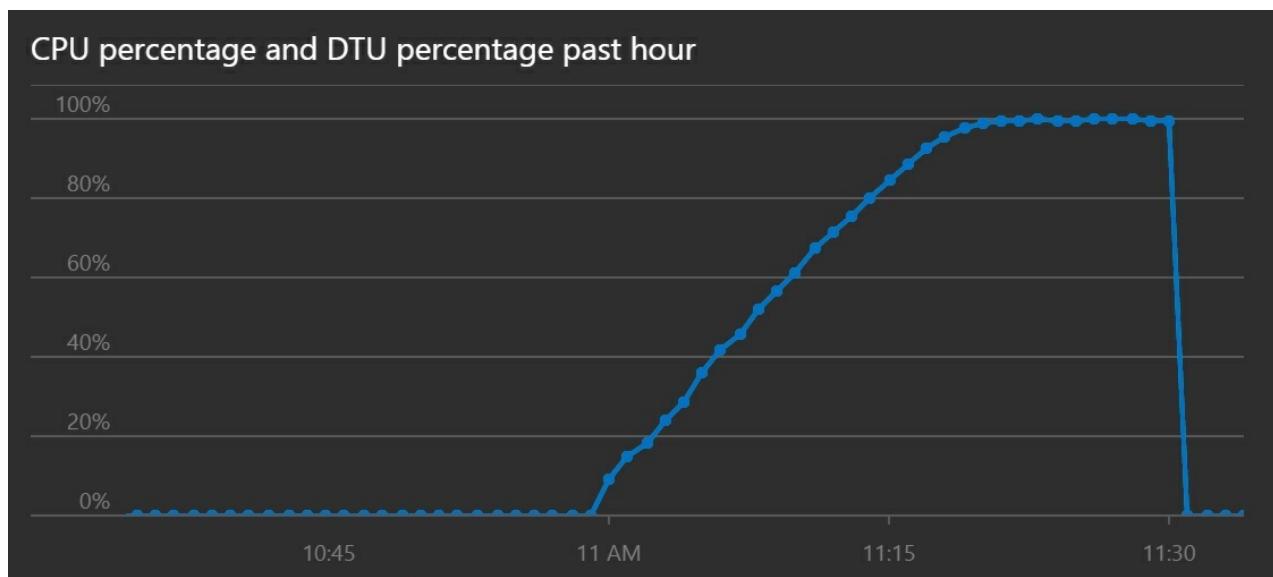
If the database operations are purely data access operations, without a lot of processing, then the problem might be [Extraneous Fetching](#).

#### 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.



CPU and DTU utilization shows that the system took longer to reach saturation, despite the increased throughput.



## Related resources

- [Extraneous Fetching antipattern](#)

# Busy Front End antipattern

9/28/2018 • 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.Sleep(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.Sleep(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

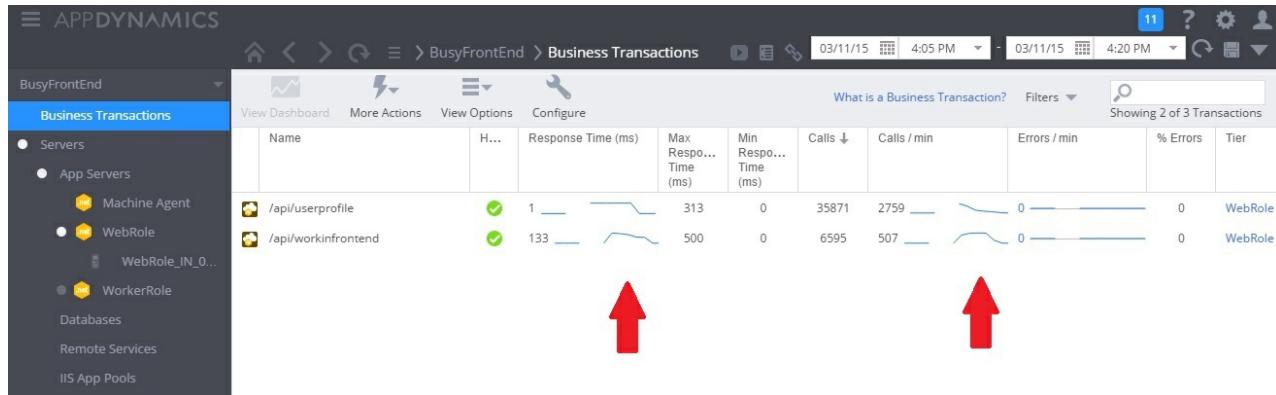
The following sections apply these steps to the sample application described earlier.

### 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

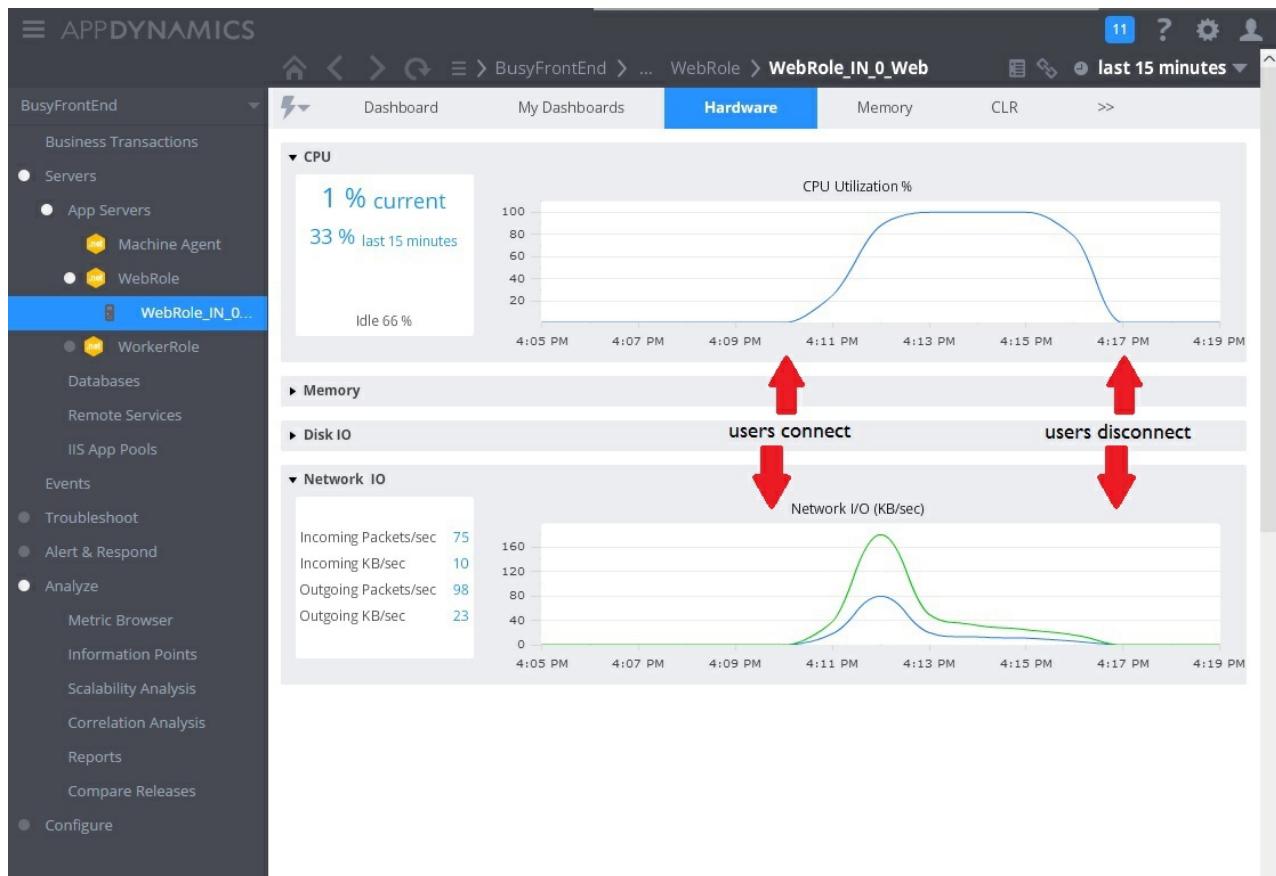
periods of stress, slow-running resource-hungry requests will likely impact other operations, and this behavior can be observed by monitoring the system and noting the drop off in performance.

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.

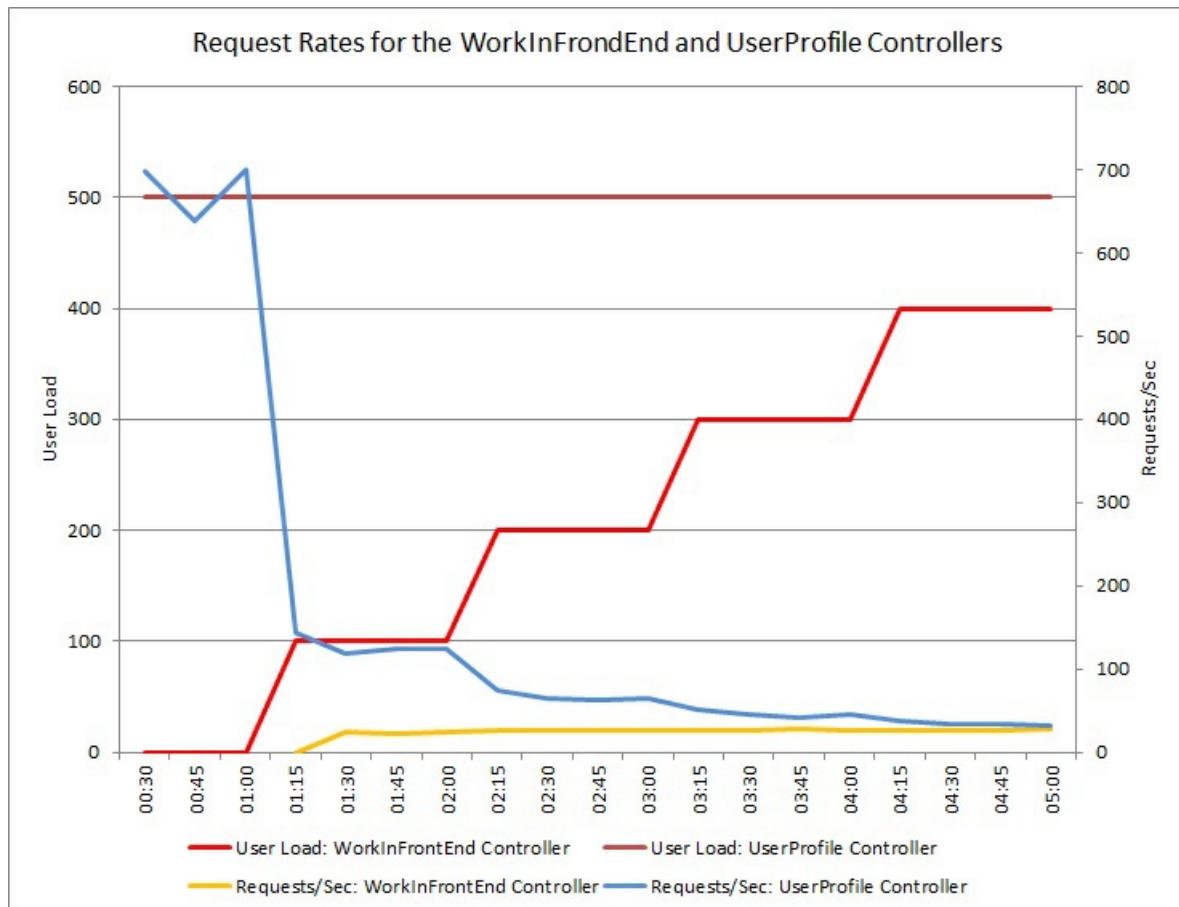


At this point, it appears the `Post` method in the `WorkInFrontEnd` controller is a prime candidate for closer examination. Further work in a controlled environment is needed to confirm the hypothesis.

### 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 towards 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.

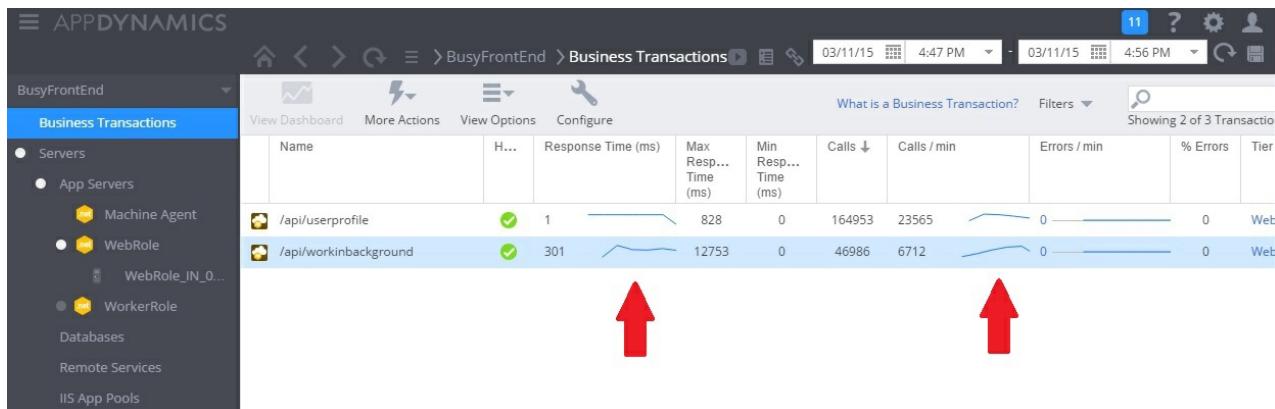
However, the work performed by this method still consumes CPU, memory, and other resources. Enabling this process to run asynchronously might actually damage performance, as users can trigger a large number of these operations simultaneously, in an uncontrolled manner. There is a limit to the number 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.

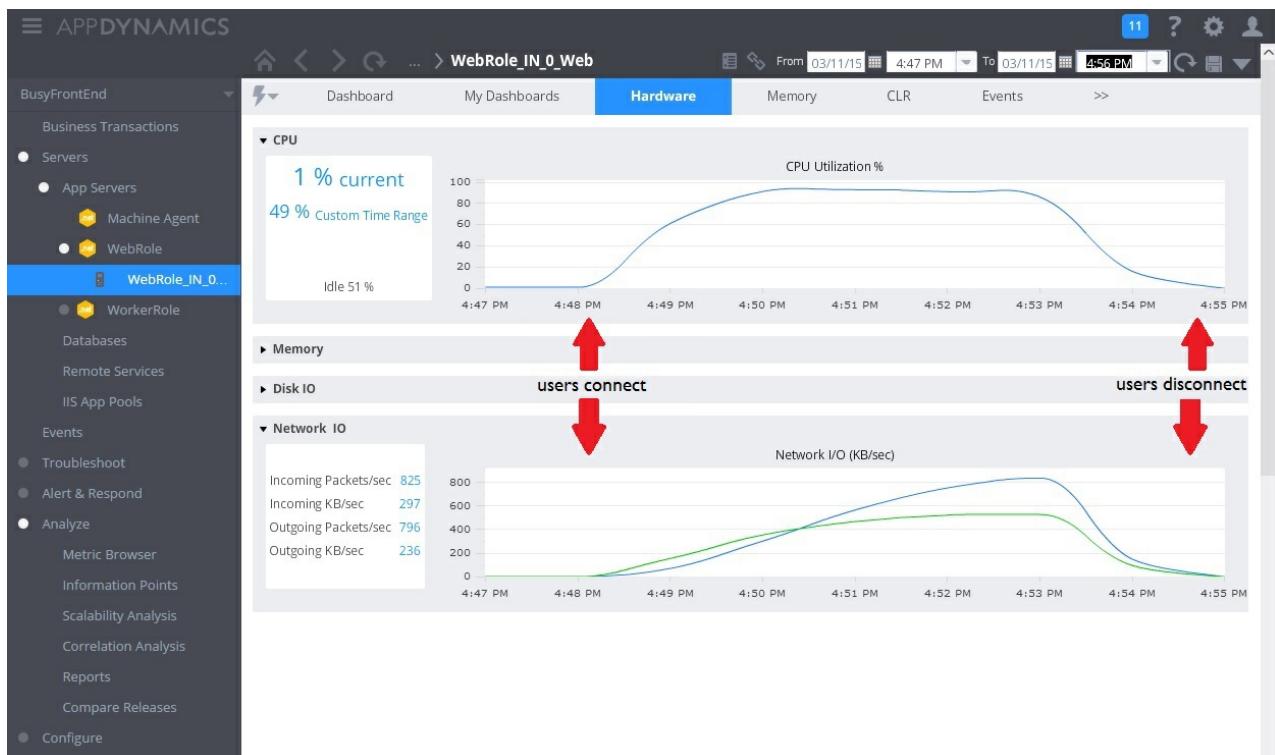
## Implement the solution and verify the result

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.

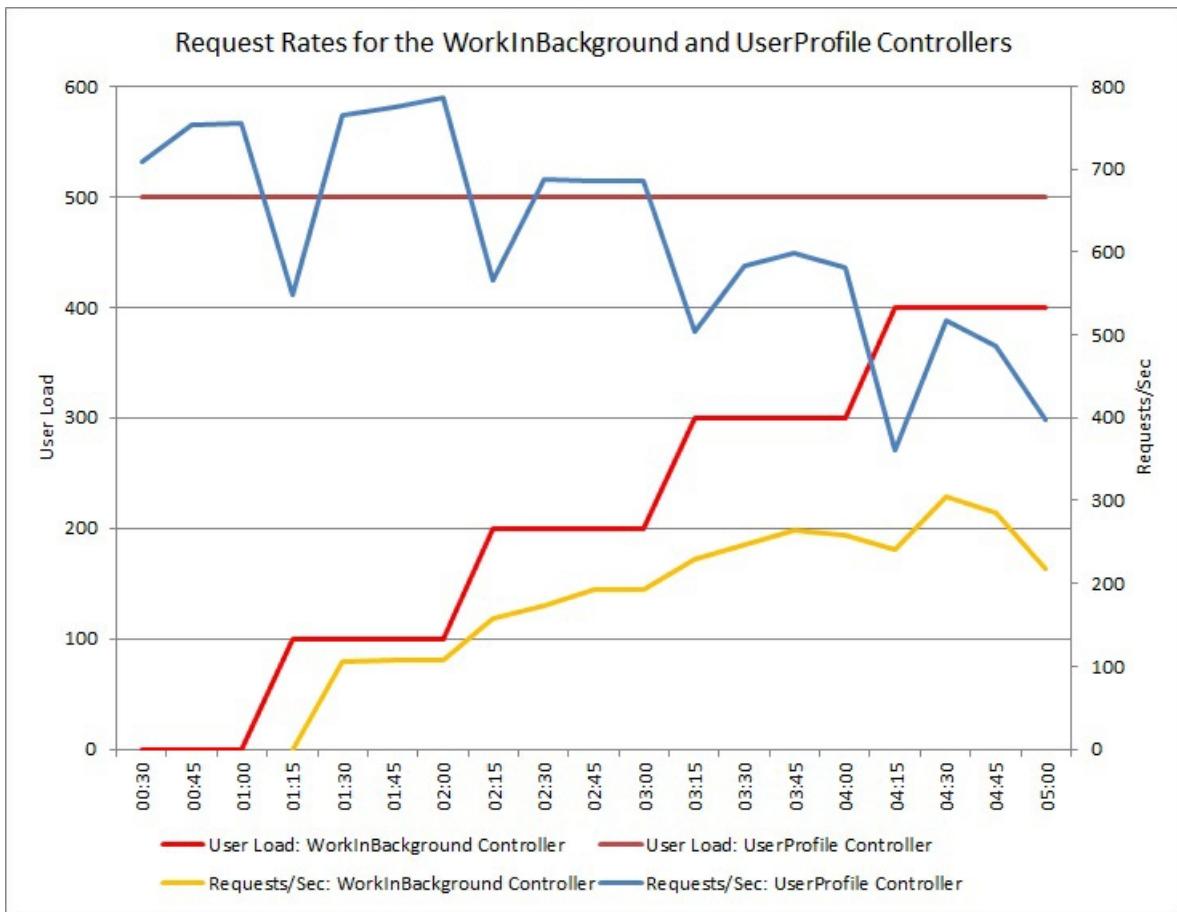


Note that the `WorkInBackground` controller also handled a much larger volume of requests. However, you can't make a direct comparison in this case, because the work being performed in this controller is very different from the original code. The new version simply queues a request, rather than performing 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 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

9/28/2018 • 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 cust)
{
    using (Stream fileStream = new FileStream(CustomersFileName, FileMode.Append))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        byte [] data = null;
        using (MemoryStream memStream = new MemoryStream())
        {
            formatter.Serialize(memStream, cust);
            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 cust in customers)
        {
            byte[] data = null;
            using (MemoryStream memStream = new MemoryStream())
            {
                formatter.Serialize(memStream, cust);
                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 cust = new Customer(...);
customers.Add(cust);

// 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:

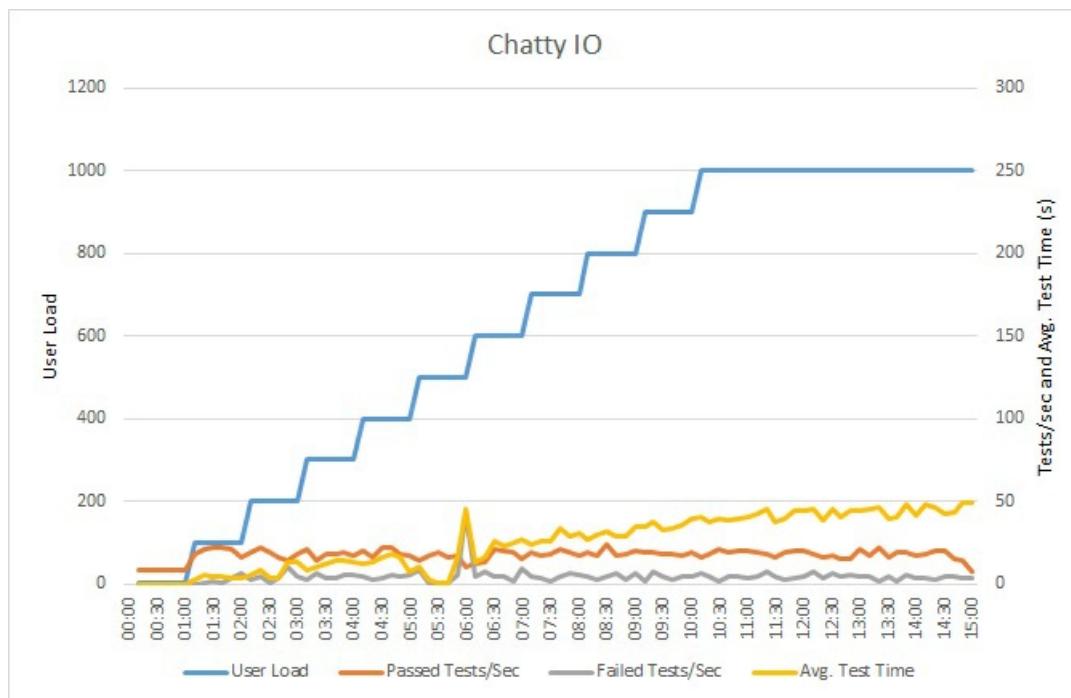
- A large number of small I/O requests made to the same file.
- A large number of small network requests made by an application instance to the same service.
- A large number of small requests made by an application instance to the same data store.
- Applications and services becoming I/O bound.

## 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 10s 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

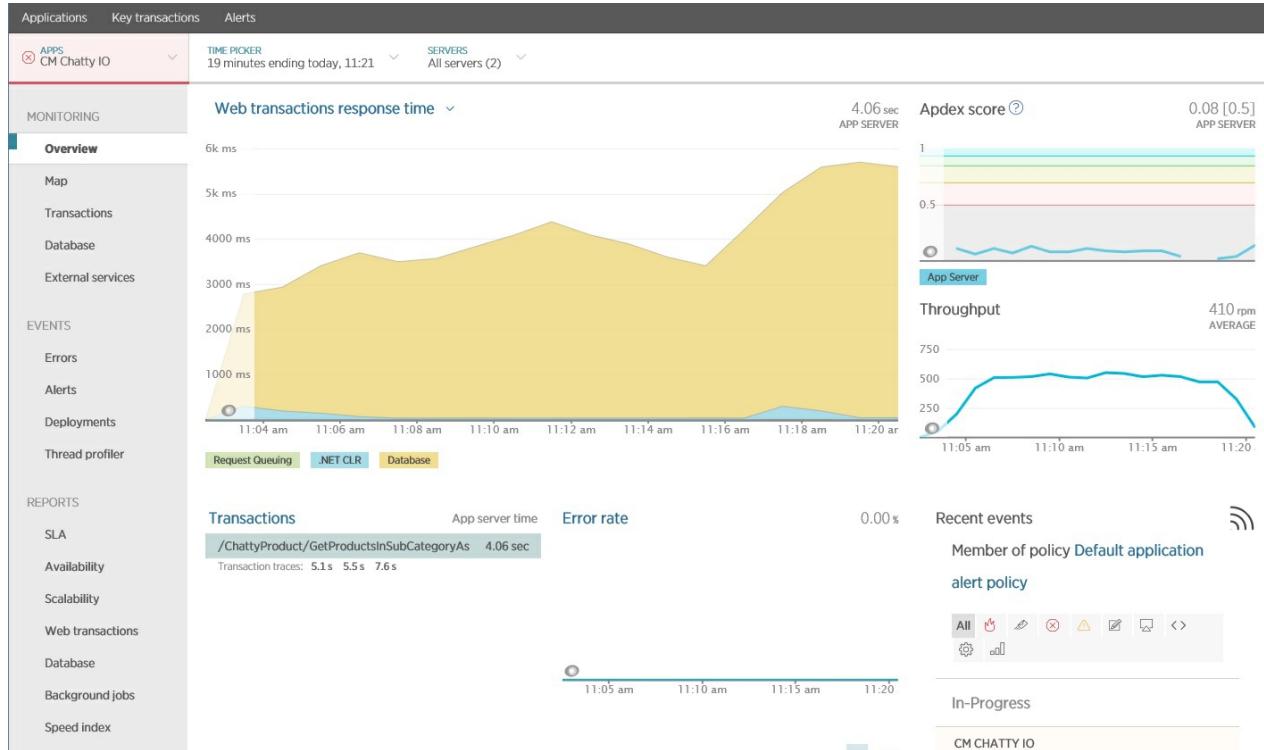
The application was deployed as an Azure App Service web app, using Azure SQL Database. The load test used a simulated step workload of up to 1000 concurrent users. The database was configured with a connection pool supporting up to 1000 concurrent connections, to reduce the chance that contention for 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.

The screenshot shows the New Relic APM interface for the 'CM Chatty IO' application. On the left, a sidebar lists monitoring, events, reports, and settings. The main area displays a chart titled 'Top 5 database operations by wall clock time' with three stacked areas: production.productlistpricehistory - SELECT (yellow), production.product - SELECT (red), and production.productssubcategory - SELECT (blue). Below this are two side-by-side charts: 'Database throughput' (20k cpm) and 'Database response time' (200 ms). A table at the bottom lists 'Slow query traces' with columns for SQL query, response time, and sample count.

It turns out that the `GetProductsInSubCategoryAsync` method, shown earlier, performs 45 SELECT queries. Each query causes the application to open a new SQL connection.

The screenshot shows a transaction trace for the endpoint `/ChattyProduct/GetProductsInSubCategoryAsync`. The trace summary indicates a total duration of 7,570 ms. The 'Slowest components' table lists the following details:

Component	Count	Duration	%
production.productlistpricehistory - SELECT	43	7,290 ms	96%
production.product - SELECT	1	133 ms	2%
production.productssubcategory - SELECT	1	86.8 ms	1%
ChattyProduct.GetProductsInSubCategoryAsync	1	53.6 ms	1%
System.Data.SqlClient.SqlConnection.Open()	45	1.24 ms	0%
HttpHandler: HttpControllerHandler	1	0.905 ms	0%

## 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.

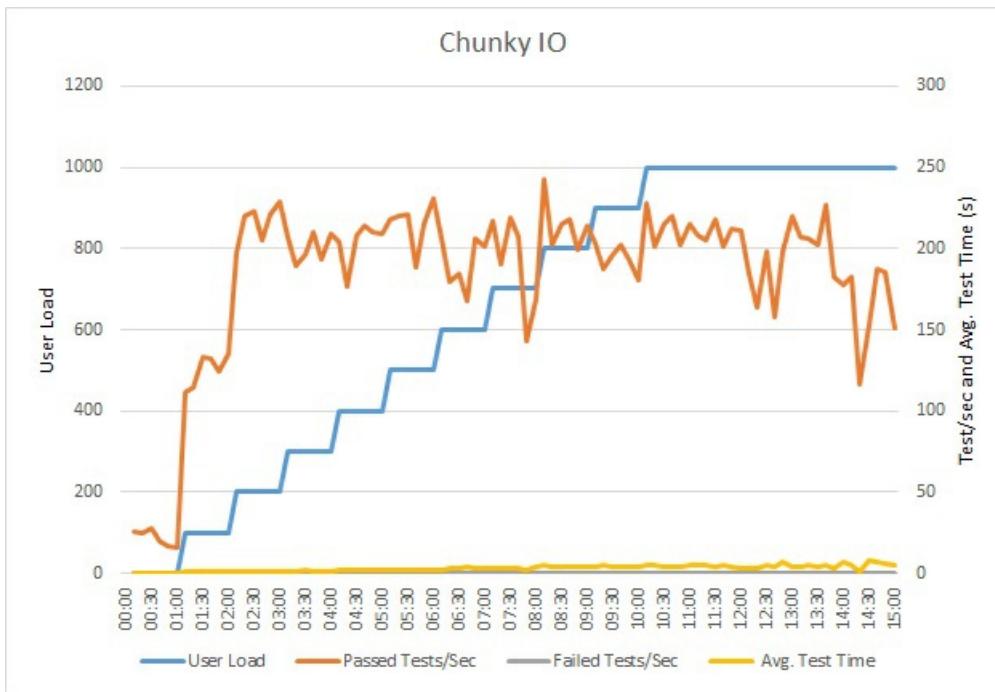
The screenshot shows the Application Insights interface. On the left, the 'Transactions' section is selected, showing a list of monitoring categories like MONITORING, EVENTS, and REPORTS. A search bar at the top right filters results by 'Most time consuming'. Below it, a specific trace is highlighted: '/ChattyProduct/GetProductsInSubCategoryAsync' with a duration of 100%. The main pane displays a 'Transaction trace' for the trace ID RD000D3A105220. It shows a single trace entry from 2015-03-18 11:06:31 with a total duration of 7,570 ms. The 'SQL statements' tab is selected, showing three queries:

Total duration	Call count	SQL
7,290 ms	43	SELECT [Extent1].[ProductID] AS [ProductID], [Extent1].[StartDate] AS [Start Date], [Extent1].[EndDate] AS [End Date], [Extent1].[ListPrice] AS [List Price] FROM [Production].[ProductListPriceHistory] AS [Extent1] WHERE [Extent1].[ProductID] = @p__linq_0
86 ms	1	SELECT TOP (?) [Extent1].[ProductSubcategoryId] AS [ProductSubcategoryId], [Extent1].[ProductCategoryId] AS [ProductCategoryId], [Extent1].[Name] AS [Name] FROM [Production].[ProductSubcategory] AS [Extent1] WHERE [Extent1].[ProductSubcategoryId] = @p__linq_0
133 ms	1	SELECT [Extent1].[ProductID] AS [ProductID], [Extent1].[Name] AS [Name], [Extent1].[ProductNumber] AS [ProductNumber], [Extent1].[ListPrice] AS [ListPrice], [Extent1].[ProductSubcategoryId] AS [ProductSubcategoryId] FROM [Production].[Product] AS [Extent1] WHERE @p__linq_0 = [Extent1].[ProductSubcategoryId]

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.

The screenshot shows the Application Insights Metrics blade. On the left, there's a sidebar with sections like Applications, Key transactions, Alerts, External services, EVENTS, Errors, Alerts, Deployments, Thread profiler, REPORTS, SLA, Availability, Scalability, Web transactions, Database, Background jobs, Speed index, SETTINGS, Application, Availability monitoring, Environment, and ALERTS. The main area displays a trace detail for a request on 2015-03-18 at 11:40:24, which took 11,700 ms. The trace details tab is selected. Below it, the SQL statements tab is shown. A large SQL query is displayed, starting with:

```

SELECT [Project1].[ProductSubcategoryID] AS [ProductSubcategoryID], [Proj
ect1].[ProductCategoryID] AS [ProductCategoryID], [Project1].[Name] AS [N
ame], [Project1].[C2] AS [C1], [Project1].[ProductID] AS [ProductID], [Pr
ject1].[Name1] AS [Name1], [Project1].[ProductNumber] AS [ProductNumb
er], [Project1].[ListPrice] AS [ListPrice], [Project1].[ProductSubcategory
ID1] AS [ProductSubcategoryID1], [Project1].[C1] AS [C2], [Project1].[Pro
ductID1] AS [ProductID1], [Project1].[StartDate] AS [StartDate], [Proj
ect1].[EndDate] AS [EndDate], [Project1].[ListPrice1] AS [ListPrice1] FROM
( SELECT [Limit1].[ProductSubcategoryID] AS [ProductSubcategoryID], [Lim
it1].[ProductCategoryID] AS [ProductCategoryID], [Limit1].[Name] AS [Nam
e], [Join1].[ProductID1] AS [ProductID], [Join1].[Name] AS [Name1], [Join
1].[ProductNumber] AS [ProductNumber], [Join1].[ListPrice1] AS [ListPric
e], [Join1].[ProductSubcategoryID1] AS [ProductSubcategoryID1], [Join1].[P
roductID2] AS [ProductID1], [Join1].[StartDate] AS [StartDate], [Join1].
[EndDate] AS [EndDate], [Join1].[ListPrice2] AS [ListPrice1], CASE WHEN
([Join1].[ProductID1] IS NULL) THEN CAST(NULL AS int) WHEN ([Join1].[Prod
uctID2] IS NULL) THEN CAST(NULL AS int) ELSE ? END AS [C1], CASE WHEN ((J
oin1).[ProductID1] IS NULL) THEN CAST(NULL AS int) ELSE ? END AS [C2] FRO
M (SELECT TOP (?) [Extent1].[ProductSubcategoryID] AS [ProductSubcategory
ID], [Extent1].[ProductCategoryID] AS [ProductCategoryID], [Extent1].[Nam
e] AS [Name] FROM [Production].[ProductSubcategory] AS [Extent1] WHERE [E
xtent1].[ProductSubcategoryID] = @__pling_0 ) AS [Limit1] LEFT OUTER JOI
N (SELECT [Extent2].[ProductID] AS [ProductID1], [Extent2].[Name] AS [Nam
e], [Extent2].[ProductNumber] AS [ProductNumber], [Extent2].[ListPrice] A
S [ListPrice1], [Extent2].[ProductSubcategoryID] AS [ProductSubcategoryI
d], [Extent3].[ProductID] AS [ProductID2], [Extent3].[StartDate] AS [Star
tDate], [Extent3].[EndDate]... (more)

```

## Related resources

- [API Design best practices](#)
- [Caching best practices](#)
- [Data Consistency Primer](#)
- [Extraneous Fetching antipattern](#)
- [No Caching antipattern](#)

# Extraneous Fetching antipattern

4/11/2018 • 10 minutes to read • [Edit Online](#)

Retrieving more data than needed for a business operation can result in unnecessary I/O overhead and reduce responsiveness.

## Problem description

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 the problem

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 actually 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 actually 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 the problem

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 are likely to 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:

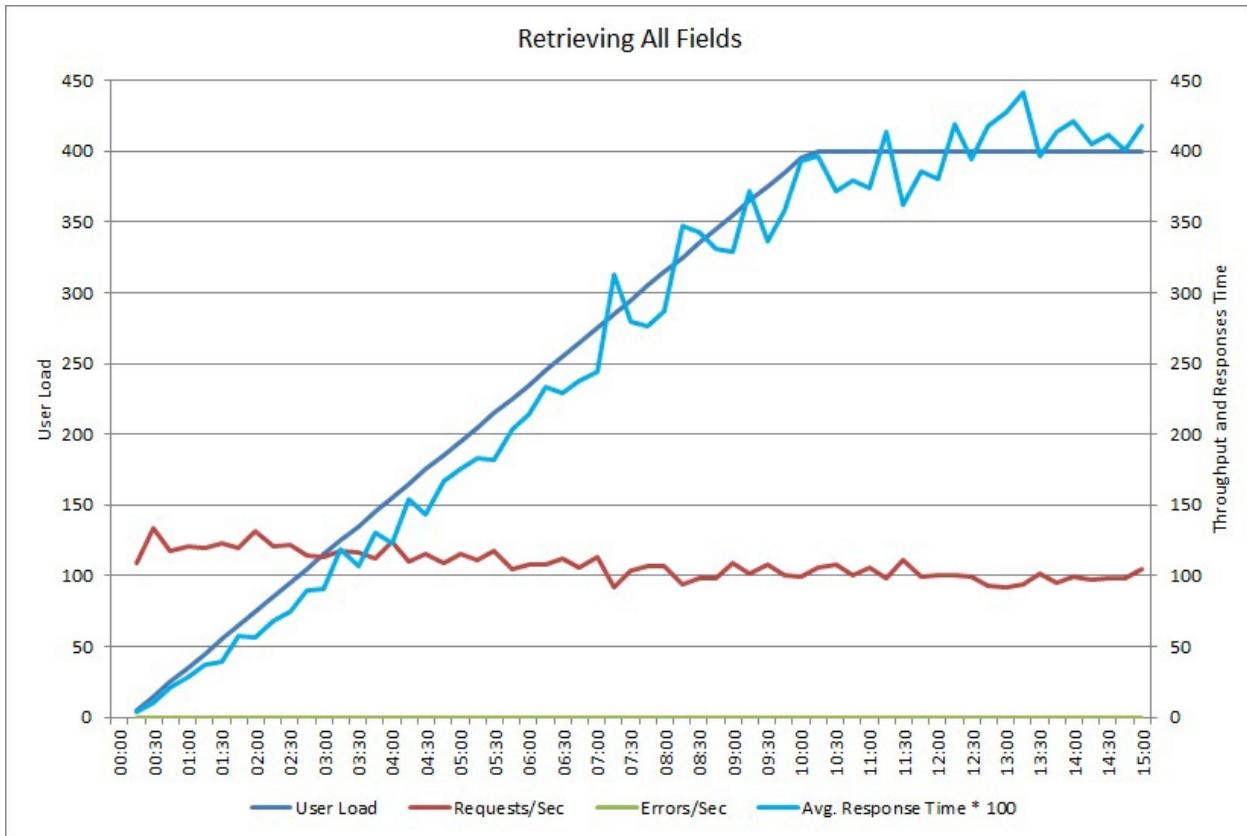
- Frequent, large I/O requests made to the same resource or data store.
- Contention in a shared resource or data store.
- An operation that frequently receives large volumes of data over the network.
- Applications and services spending significant time waiting for I/O to complete.

## Example diagnosis

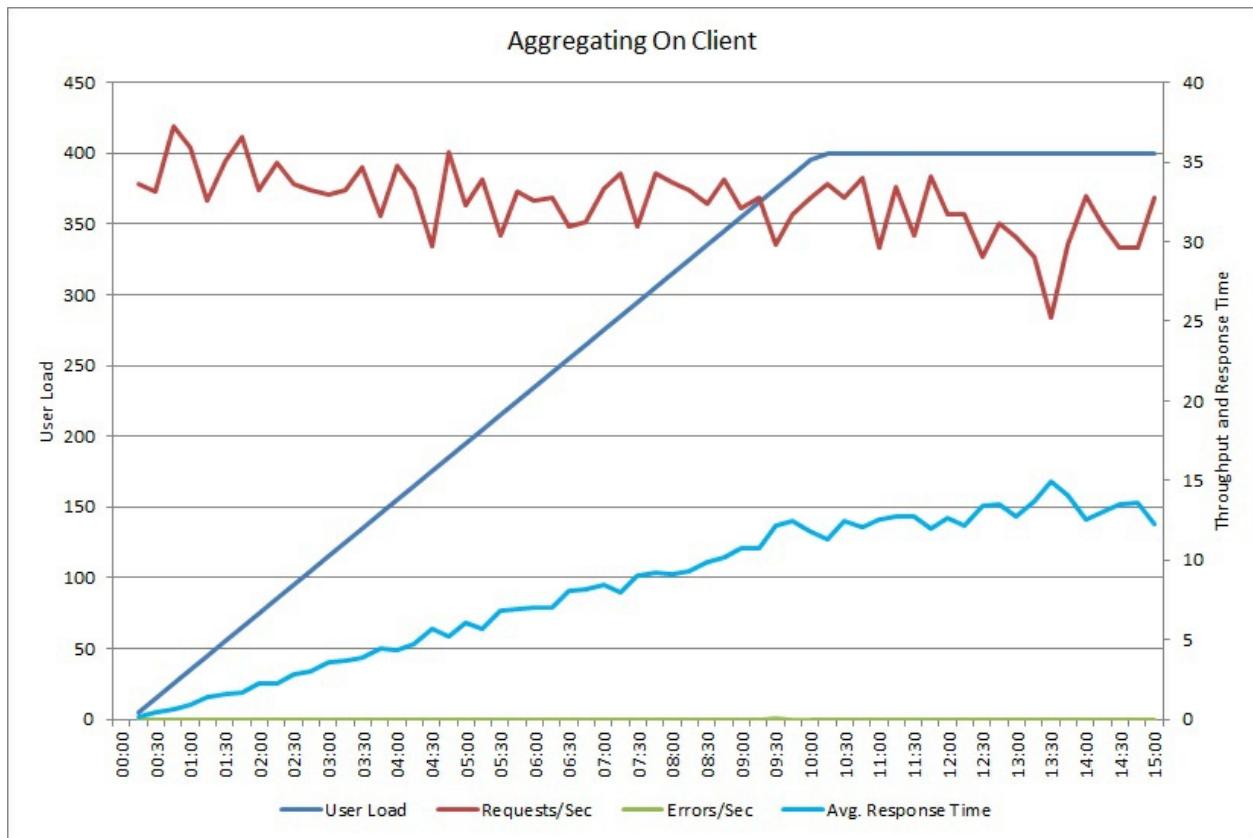
The following sections apply these steps to the previous examples.

### 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.

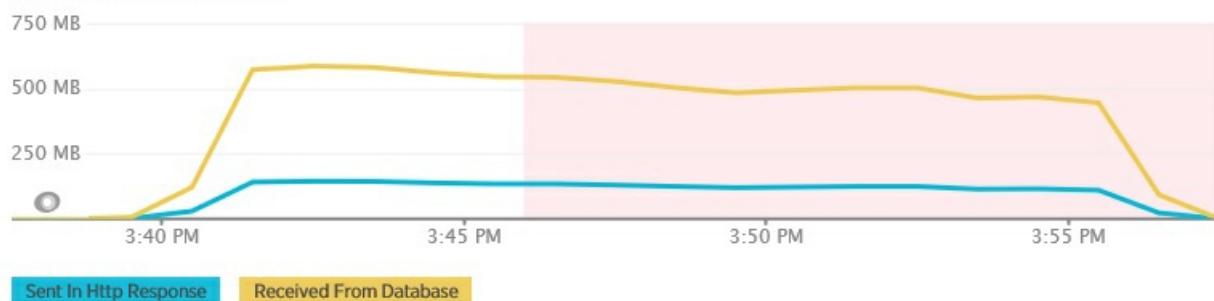
Compare this information against the volume of data being returned by the application to the client. Track the ratio of the volume of data returned by the data store against the volume of data returned to the client. If there is any large disparity, investigate to determine whether the application is 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,

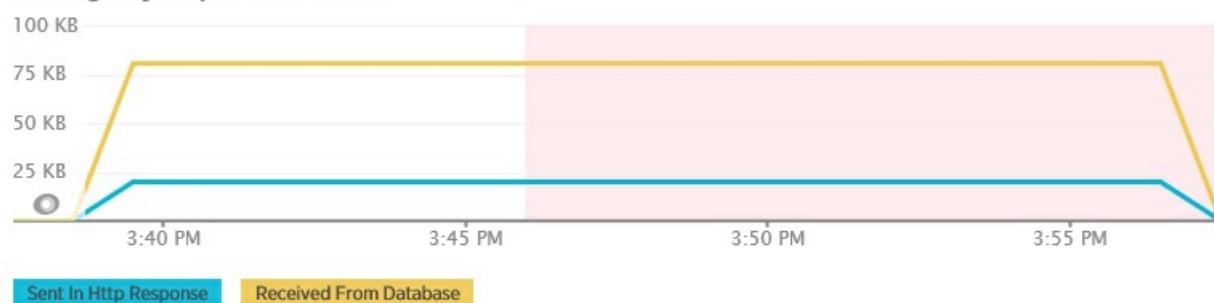
or you can model a series of synthetic workloads and run them against a test system.

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.

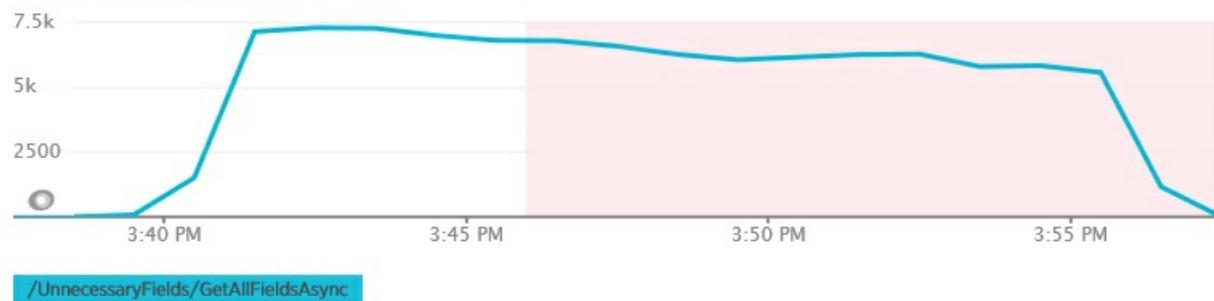
### Total Bytes per Minute



### Average Bytes per Transaction



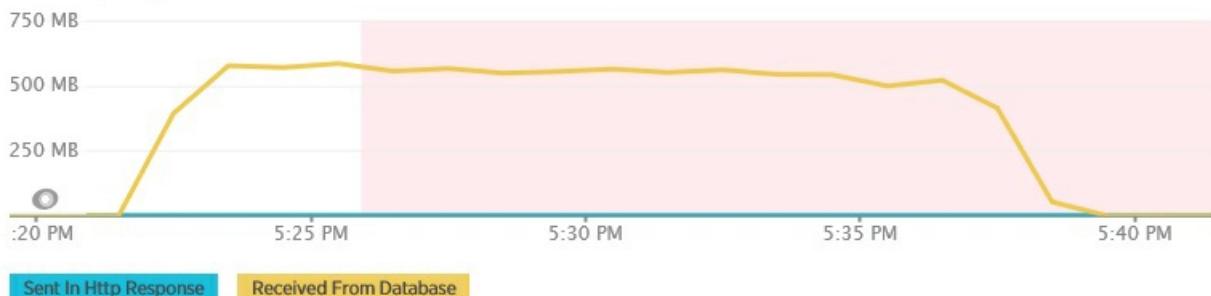
### Requests per Minute



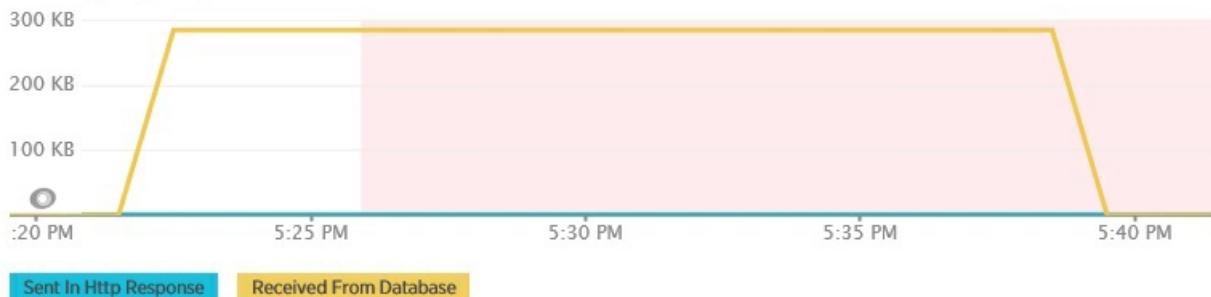
For each request, the database returned 80,503 bytes, but the response to the client only contained 19,855 bytes, about 25% of the size of the database response. The size of the data returned to the client can vary depending on the format. For this load test, the client requested JSON data. Separate 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 280Kb 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.

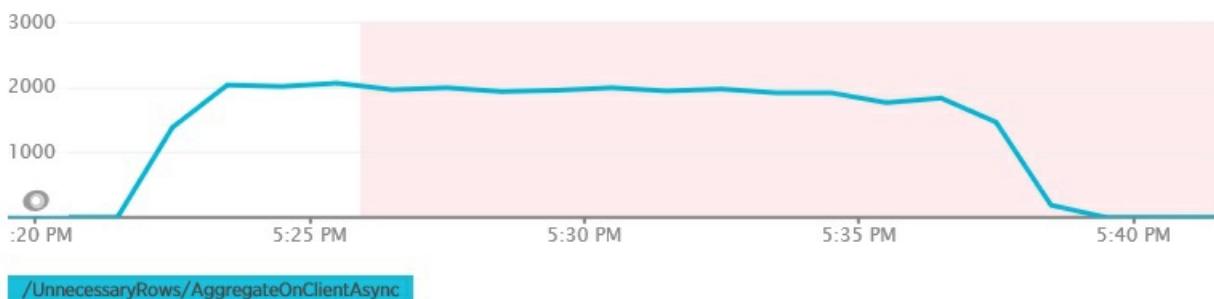
### Total Bytes per Minute



### Average Bytes per Transaction



### Requests per Minute



### 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

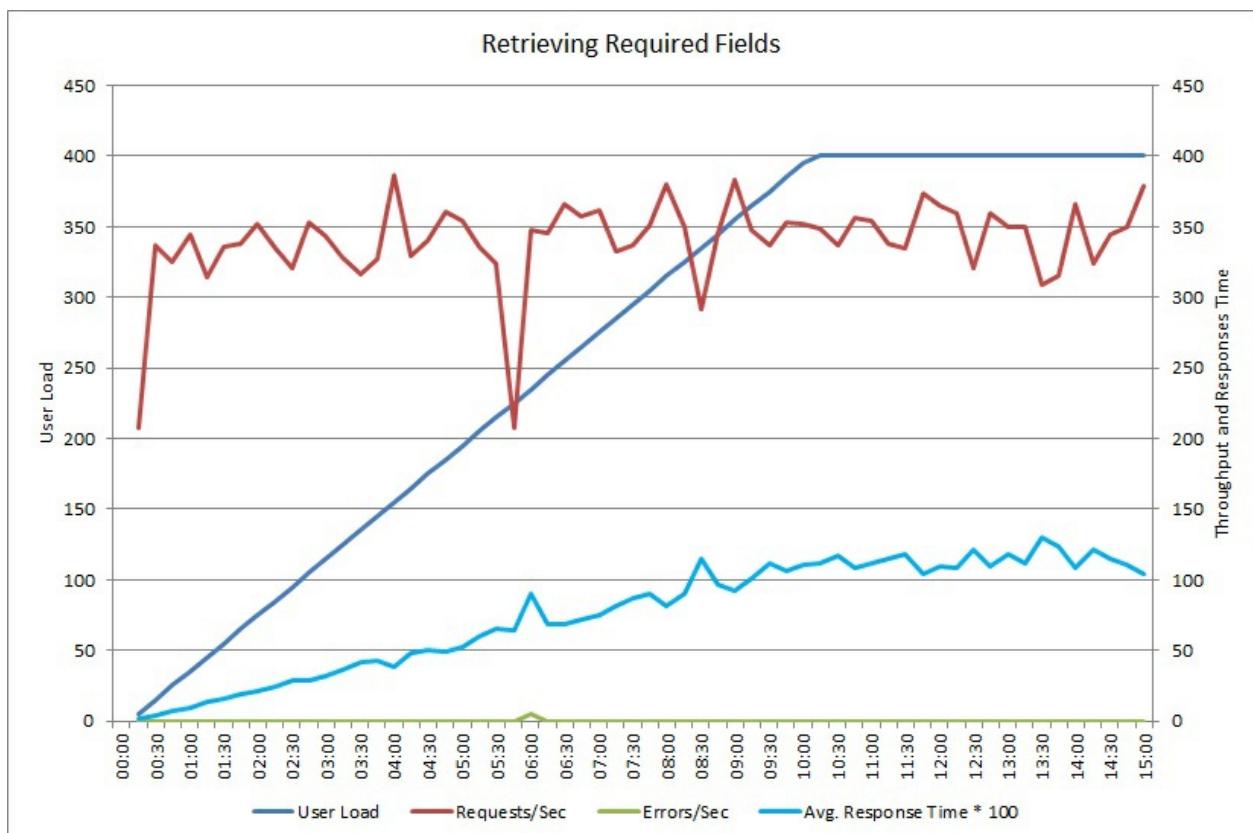
## Plan Information

## Advanced Information

Run Count	867	Plan Handle	0x0600E0004D00CD02A0EDD87D550000000100000000000000000000000000000000
Last Run Time	03/03/2015 15:32:25	SQL Handle	0x020000004D00CD02B85696A75DA7BEBF79E79259988F30C300000000
Plan Generation Count	1	Query Hash	0x05ACF4F9056ACADC
Time Plan Cached	03/03/2015 15:26:32	Query Plan Hash	0x0DA11AA10A268A7B

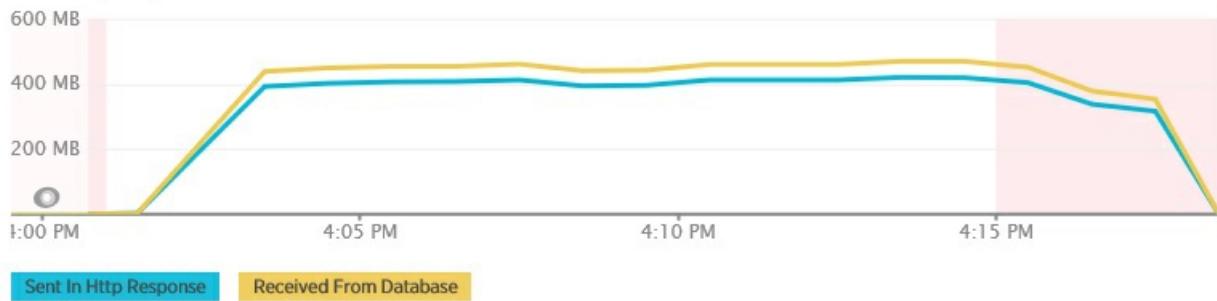
## Implement the solution and verify the result

After changing the `GetRequiredFieldsAsync` method to use a SELECT statement on the database side, load testing showed the following results.

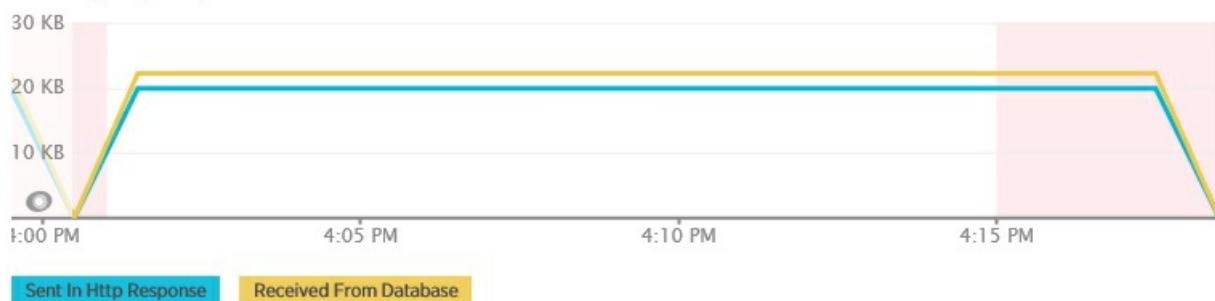


This load test used the same deployment and the same simulated workload of 400 concurrent users as before. The graph shows much lower latency. Response time rises with load to approximately 1.3 seconds, compared to 4 seconds in the previous case. The throughput is also higher at 350 requests per second compared to 100 earlier. The volume of data retrieved from the database now closely matches the size of the HTTP response messages.

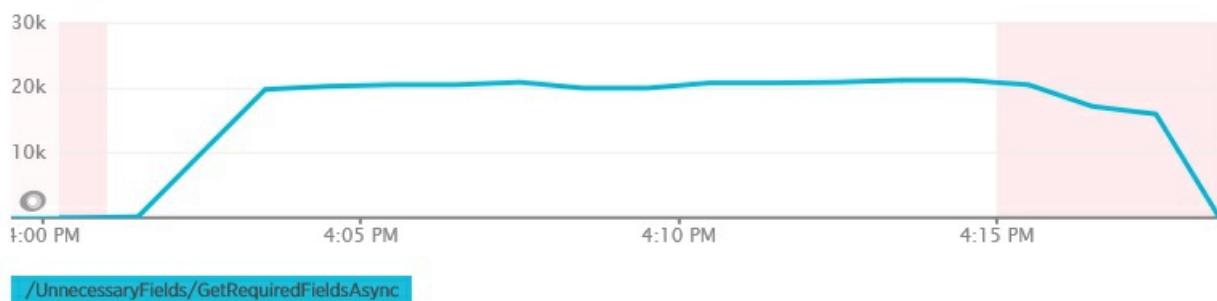
#### Total Bytes per Minute



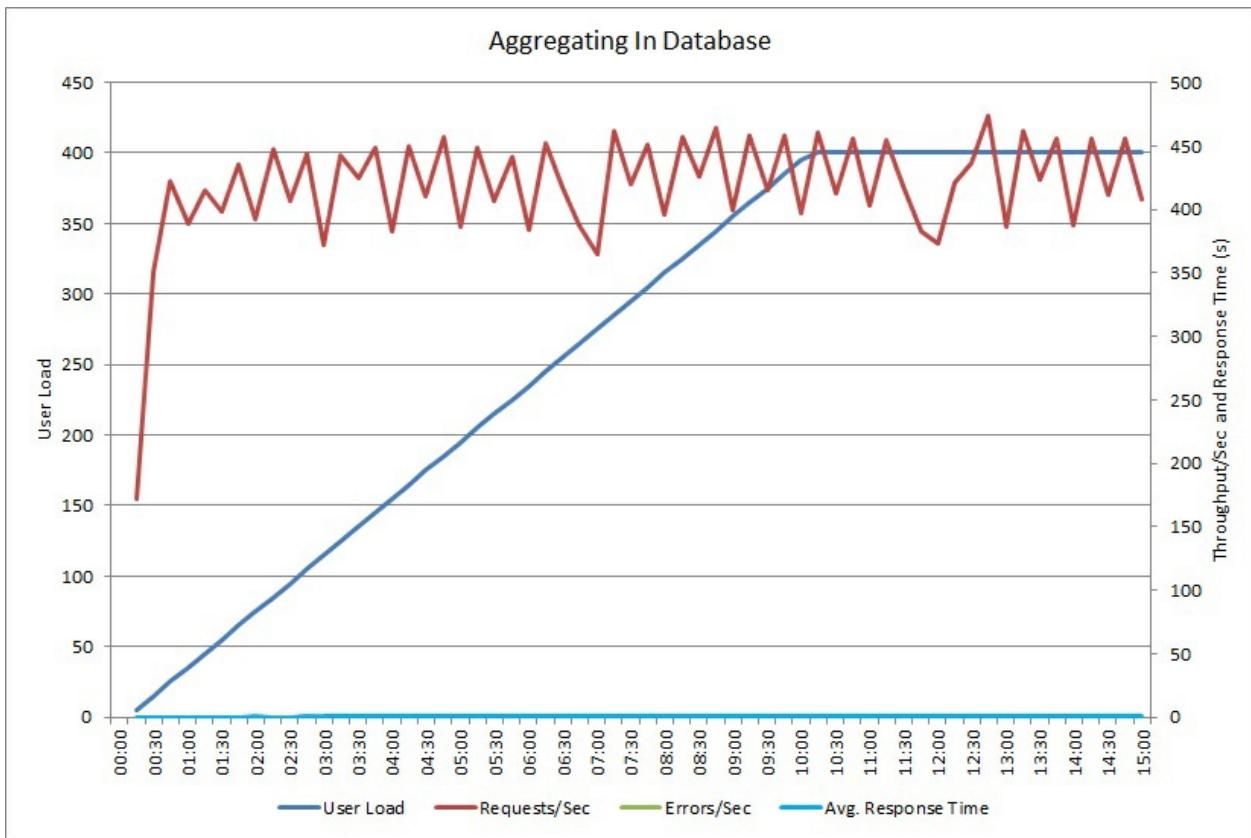
#### Average Bytes per Transaction



#### Requests per Minute



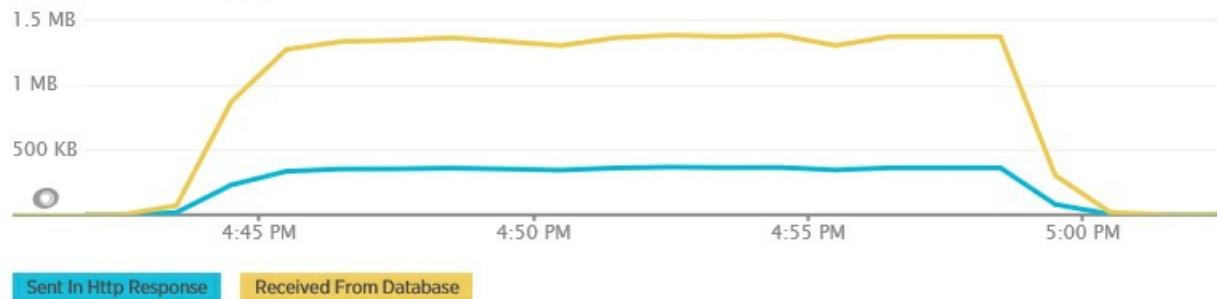
Load testing using the `AggregateOnDatabaseAsync` method generates the following results:



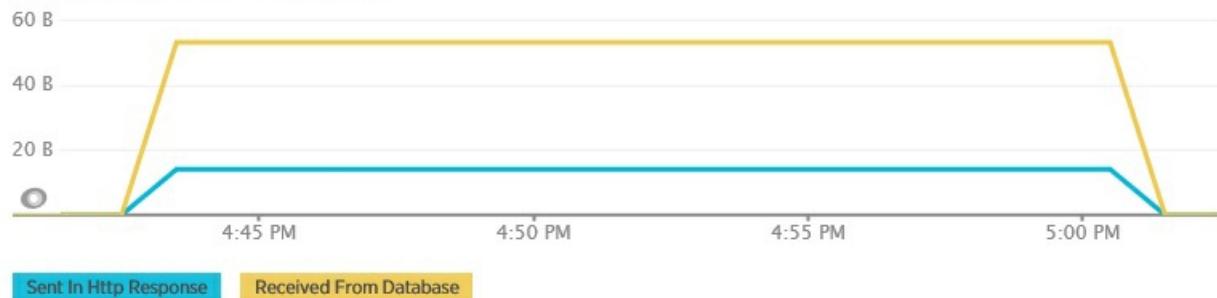
The average response time is now minimal. This is an order of magnitude improvement in performance, caused primarily by the large reduction in I/O from the database.

Here is the corresponding telemetry for the `AggregateOnDatabaseAsync` method. The amount of data retrieved from the database was vastly reduced, from over 280Kb 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.

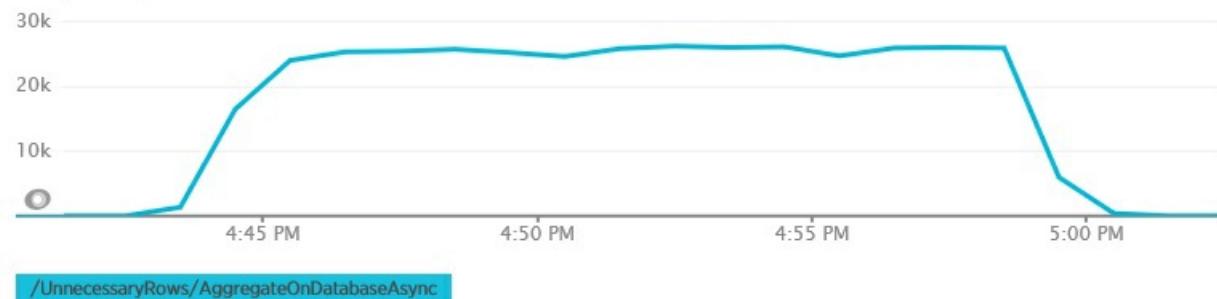
### Total Bytes per Minute



### Average Bytes per Transaction



### Requests per Minute



## Related resources

- [Busy Database antipattern](#)
- [Chatty I/O antipattern](#)
- [Data partitioning best practices](#)

# Improper Instantiation antipattern

2/22/2018 • 6 minutes to read • [Edit Online](#)

It can hurt performance to continually create new instances of an object that is meant to be created once and then shared.

## 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 Redis Cache.

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 instances 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.Sleep(Int32.MaxValue / 100);
    }
    ...
}

```

## How to fix the problem

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.
- 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 factories objects are intended to be saved and reused, rather than disposed and recreated. 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 the problem

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.

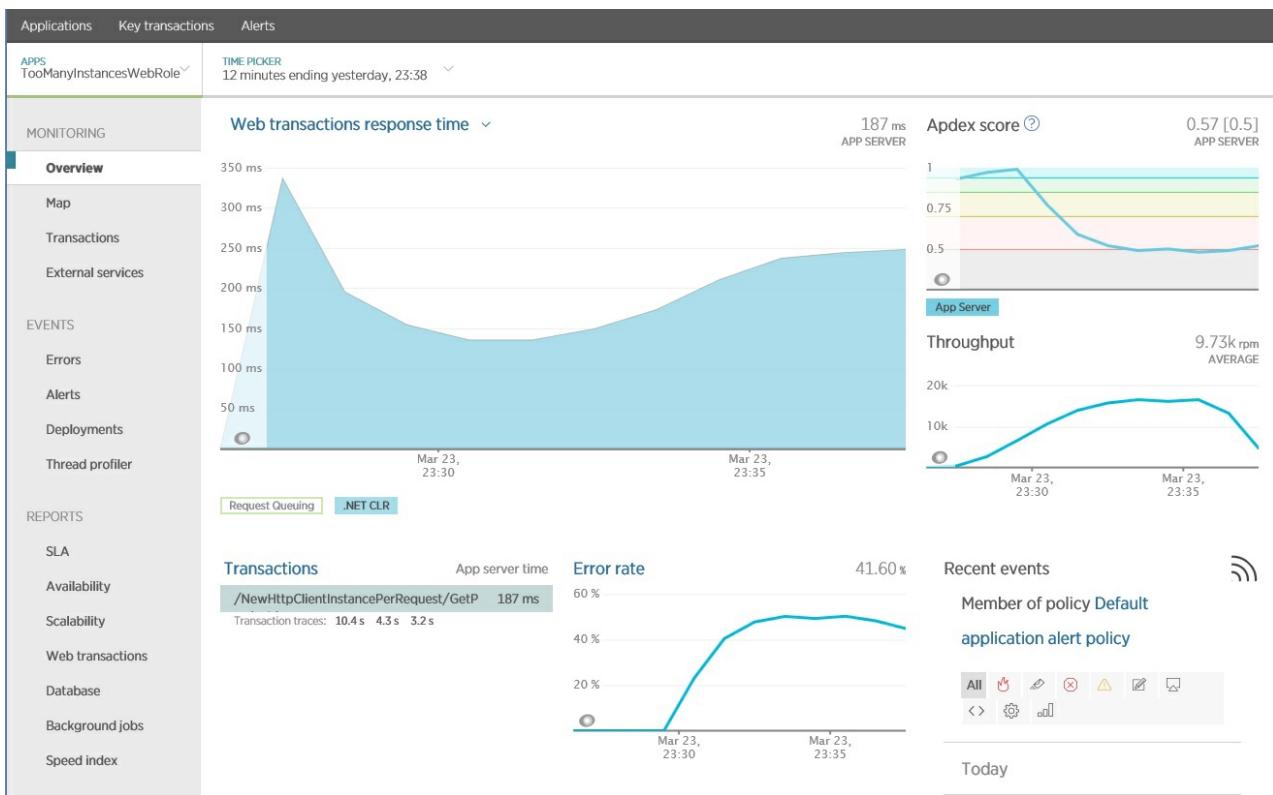
Look at stack traces for operations that are slow-running or that generate exceptions when the system is under load. This information can help to identify how these operations are utilizing resources. Exceptions can help to determine whether errors are caused by shared resources being exhausted.

## Example diagnosis

The following sections apply these steps to the sample application described earlier.

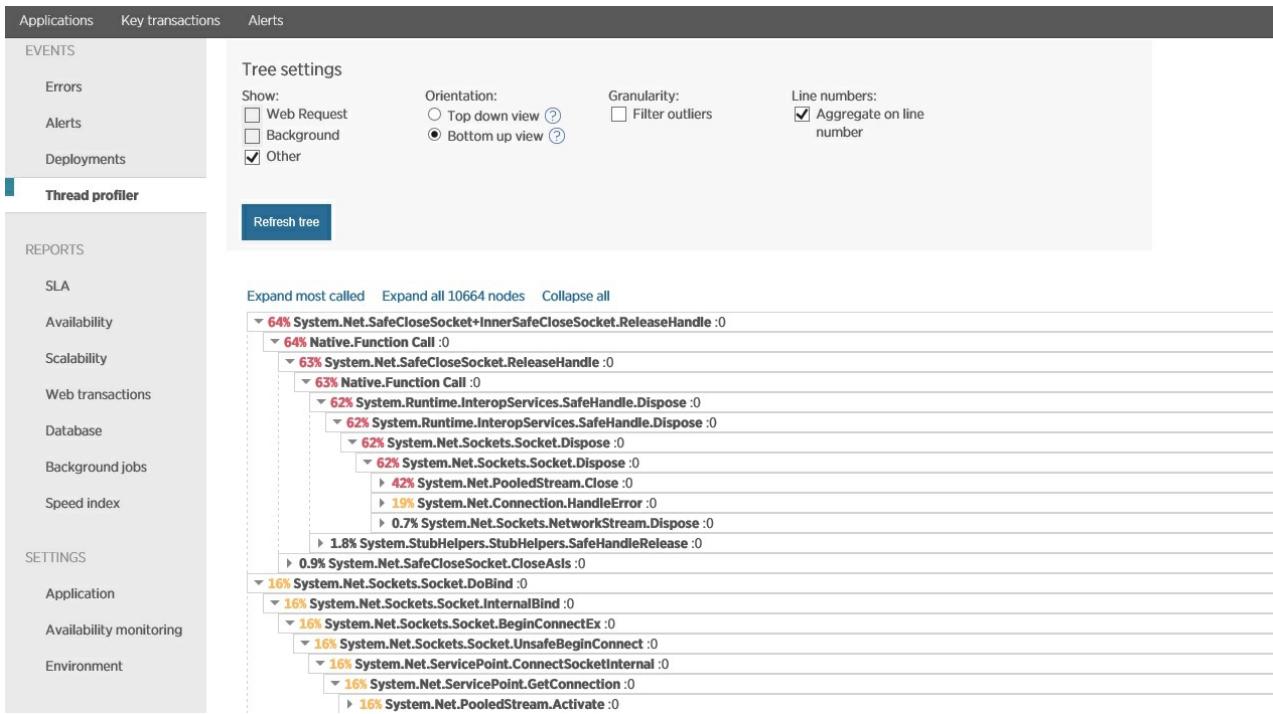
### Identify points of slow down or failure

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

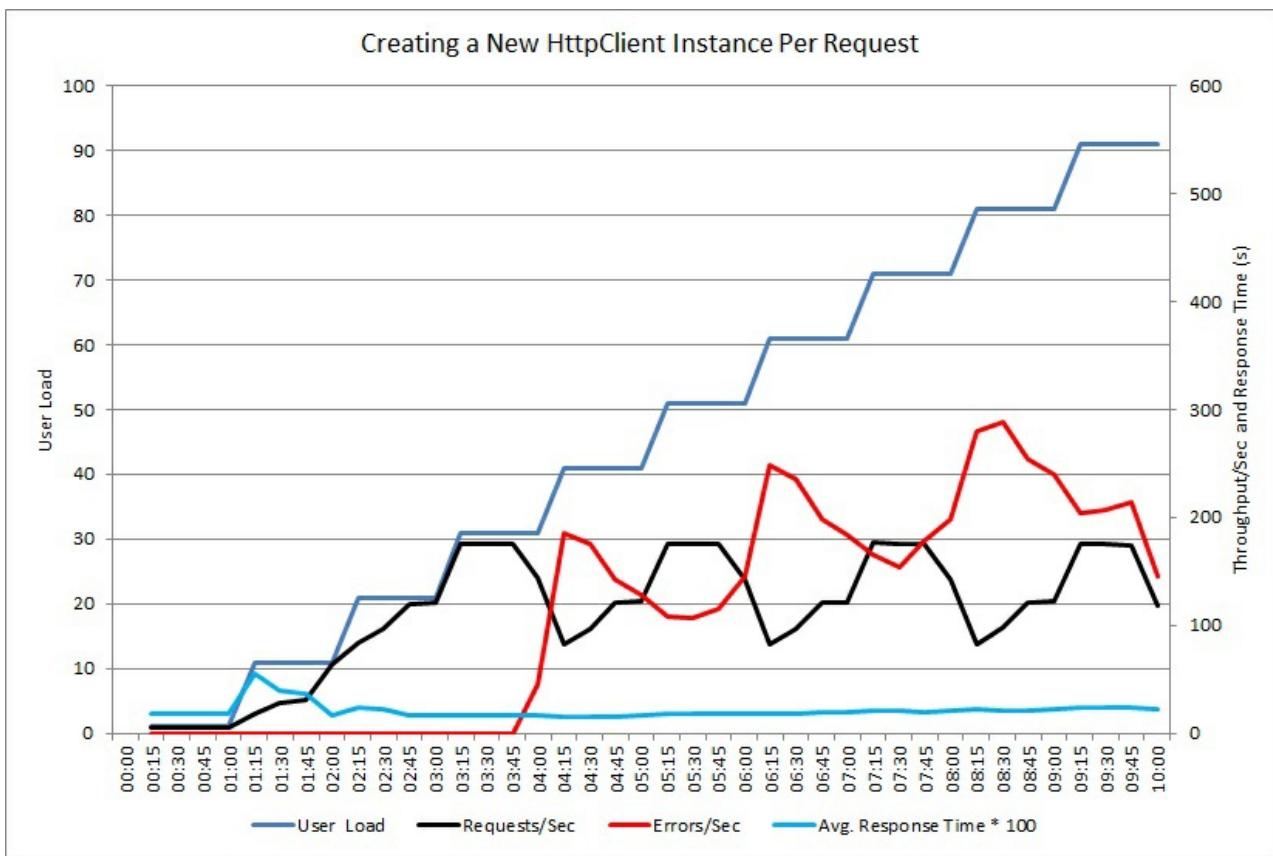
The next image shows data captured using thread profiling, over the same period corresponding as the previous image. The system spends a significant time opening socket connections, and even more time closing them and handling socket exceptions.



## 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

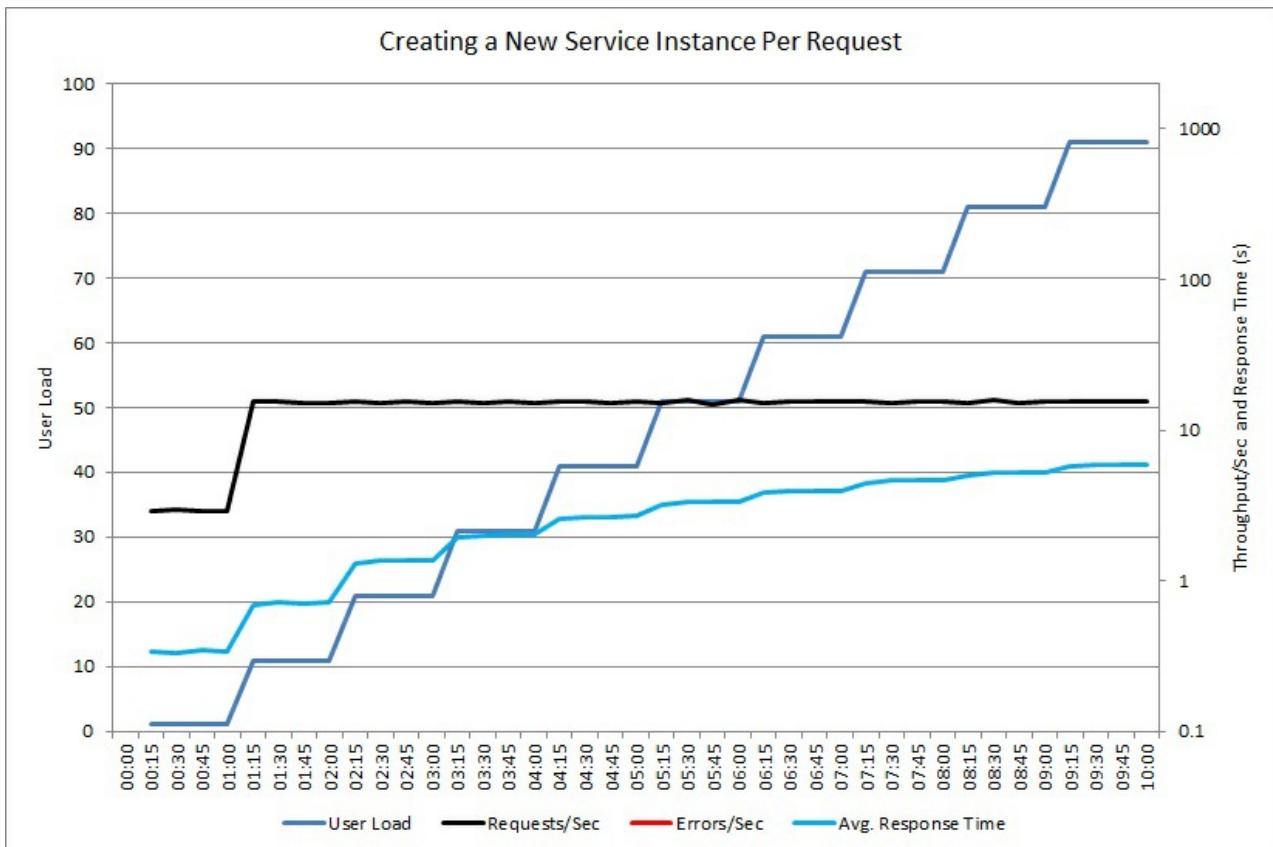
NewHttpClientInstancePerRequest controller as the user load increases to 100 concurrent users.



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.

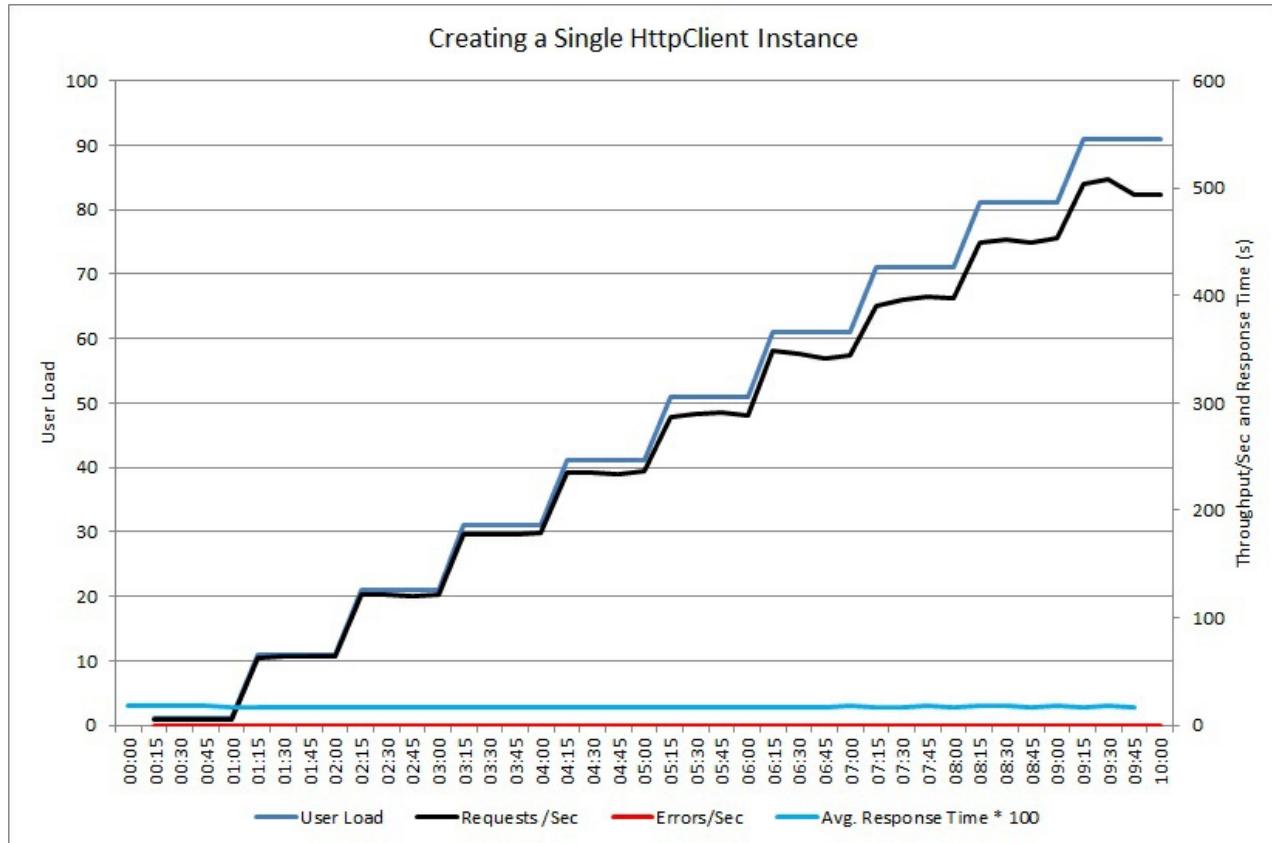
The next graph shows a similar test for a controller that creates the custom `ExpensiveToCreateService` object.



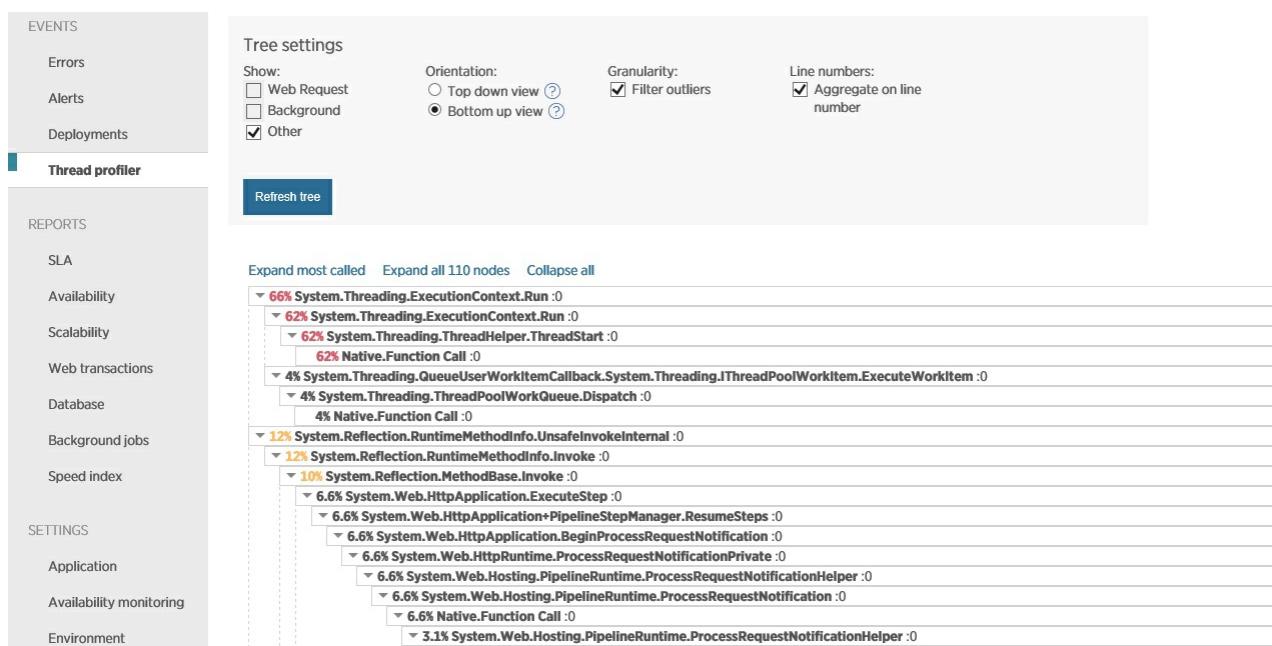
This time, the controller does not generate any exceptions, but throughput still reaches a plateau, while the average response time increases by a factor of 20. (The graph uses a logarithmic scale for response time and throughput.) Telemetry showed that creating new instances of the `ExpensiveToCreateService` was the main cause of the problem.

## Implement the solution and verify the result

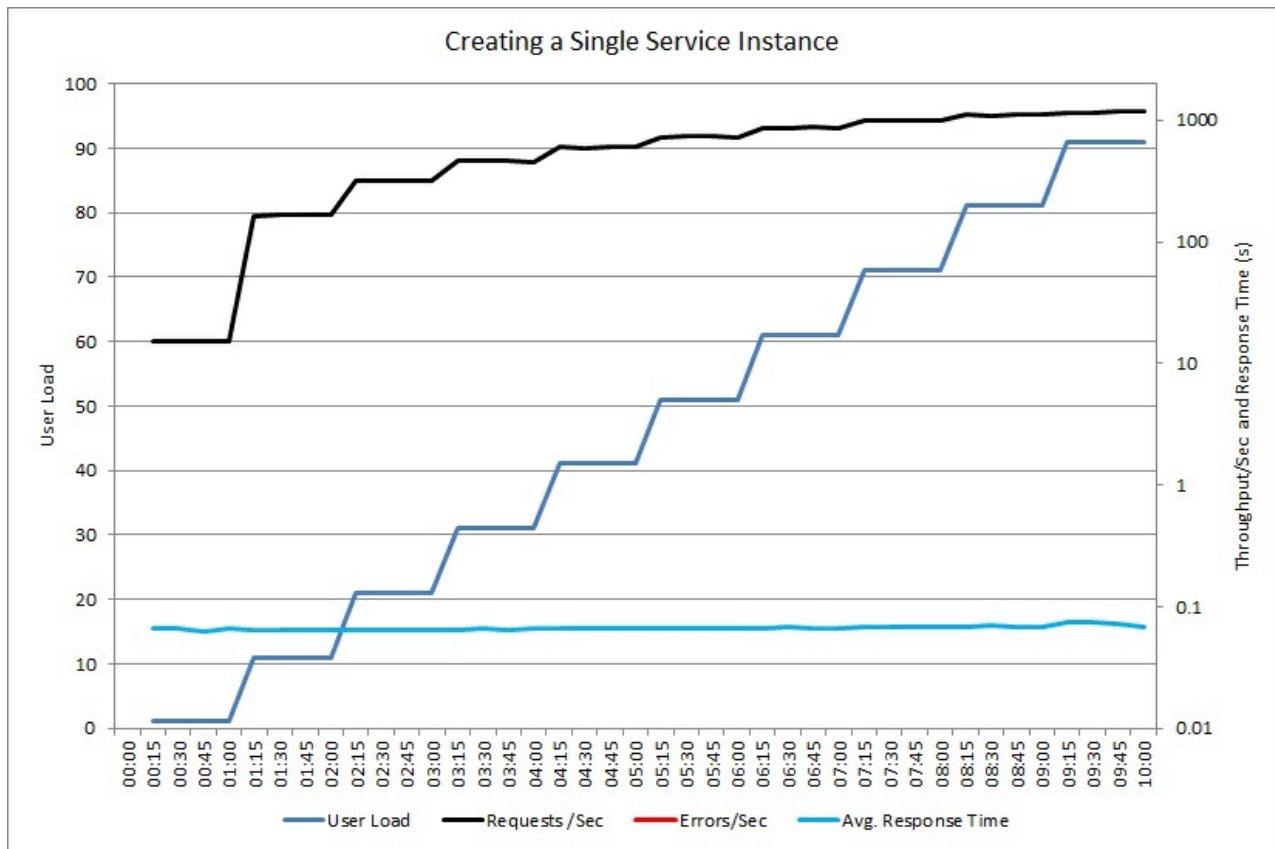
After switching the `GetProductAsync` method to share a single `HttpClient` instance, a second load test showed improved performance. No errors were reported, and the system was able to handle an increasing load of up to 500 requests per second. The average response time was cut in half, compared with 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.



# Monolithic Persistence antipattern

9/28/2018 • 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 heterogenous 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 Redis Cache](#) 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.

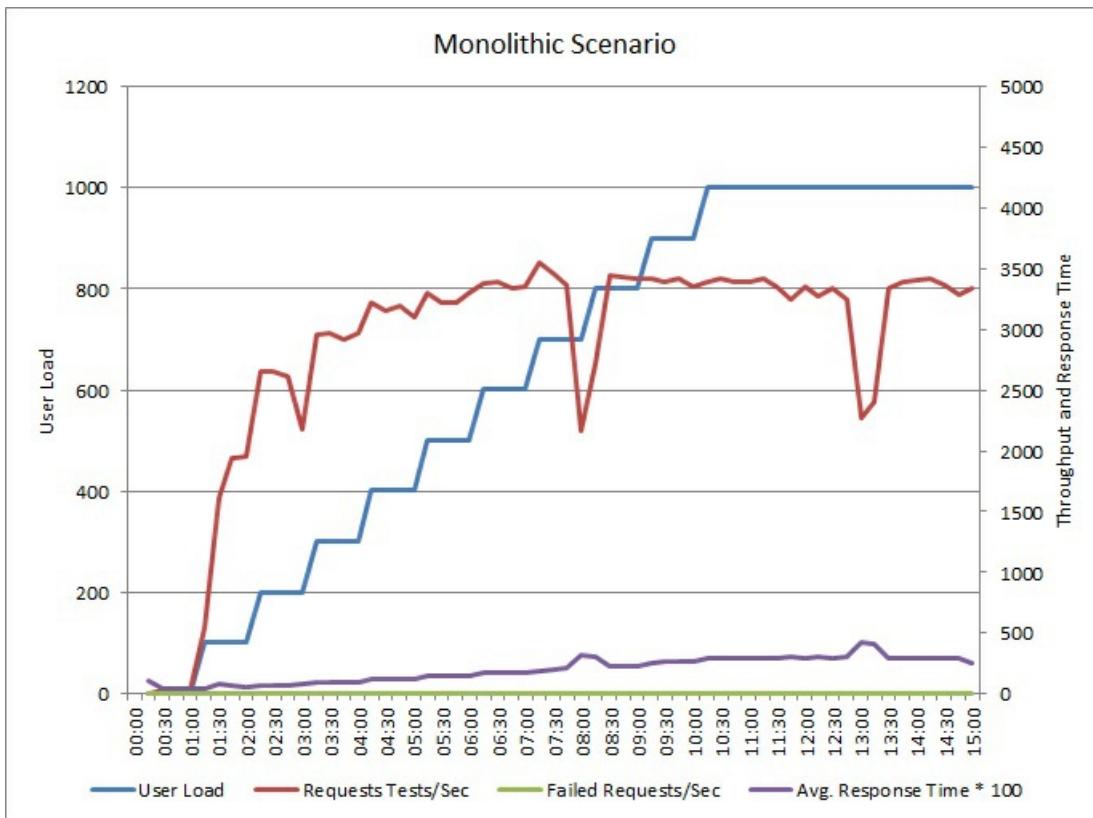
1. Instrument the system to record the key performance statistics. Capture timing information for each operation, as well as the points where the application reads and writes data.
2. If possible, monitor the system running for a few days in a production environment to get a real-world view of how the 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

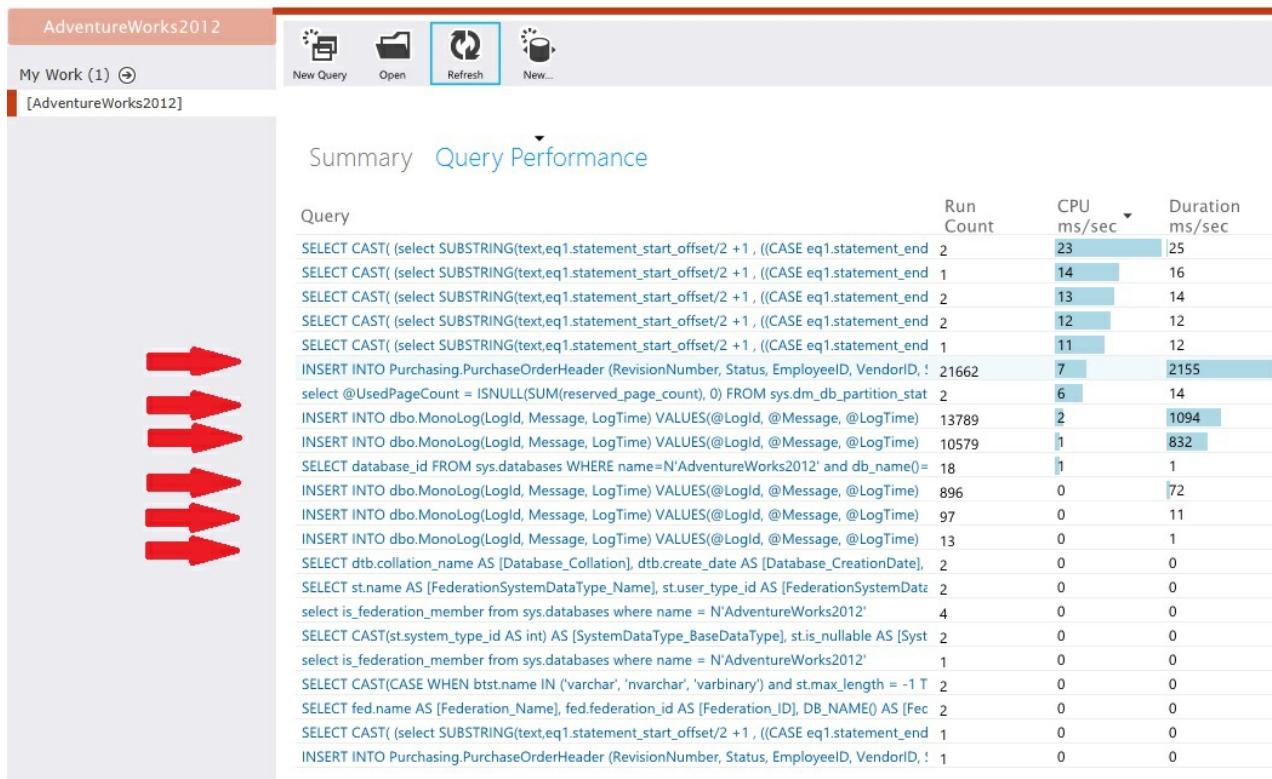
The next graph shows the utilization of database throughput units (DTU) during the load test. (A DTU is a measure of available capacity, and is a combination of CPU utilization, memory allocation, I/O rate.) Utilization of DTUs quickly reached 100%. This is roughly the point where throughput peaked in 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.

## Monitoring



## 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.



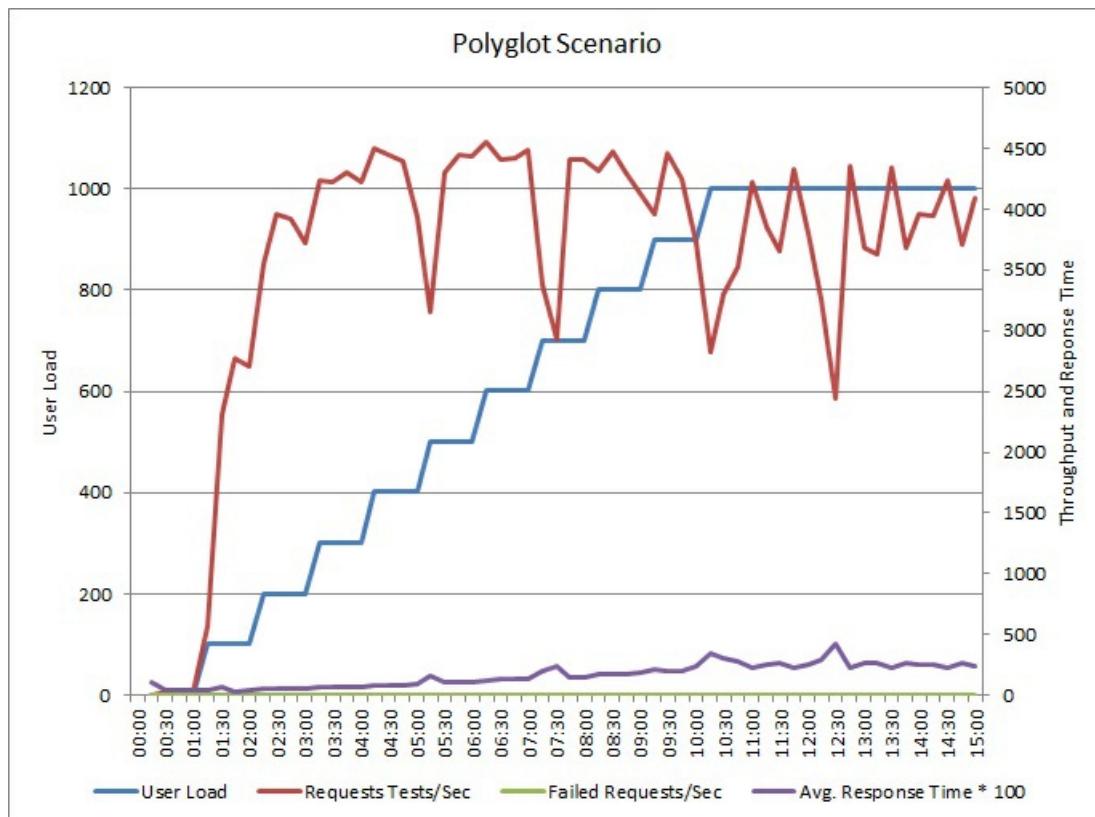
## Identify resource contention

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.

## Monitoring



## 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

9/28/2018 • 8 minutes to read • [Edit Online](#)

In a cloud application that handles many concurrent requests, repeatedly fetching the same data can reduce performance and scalability.

## Problem description

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.

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 problem

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][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 Redis Cache. If the value isn't found in Redis 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

- 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 nonstatic 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 the problem

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.

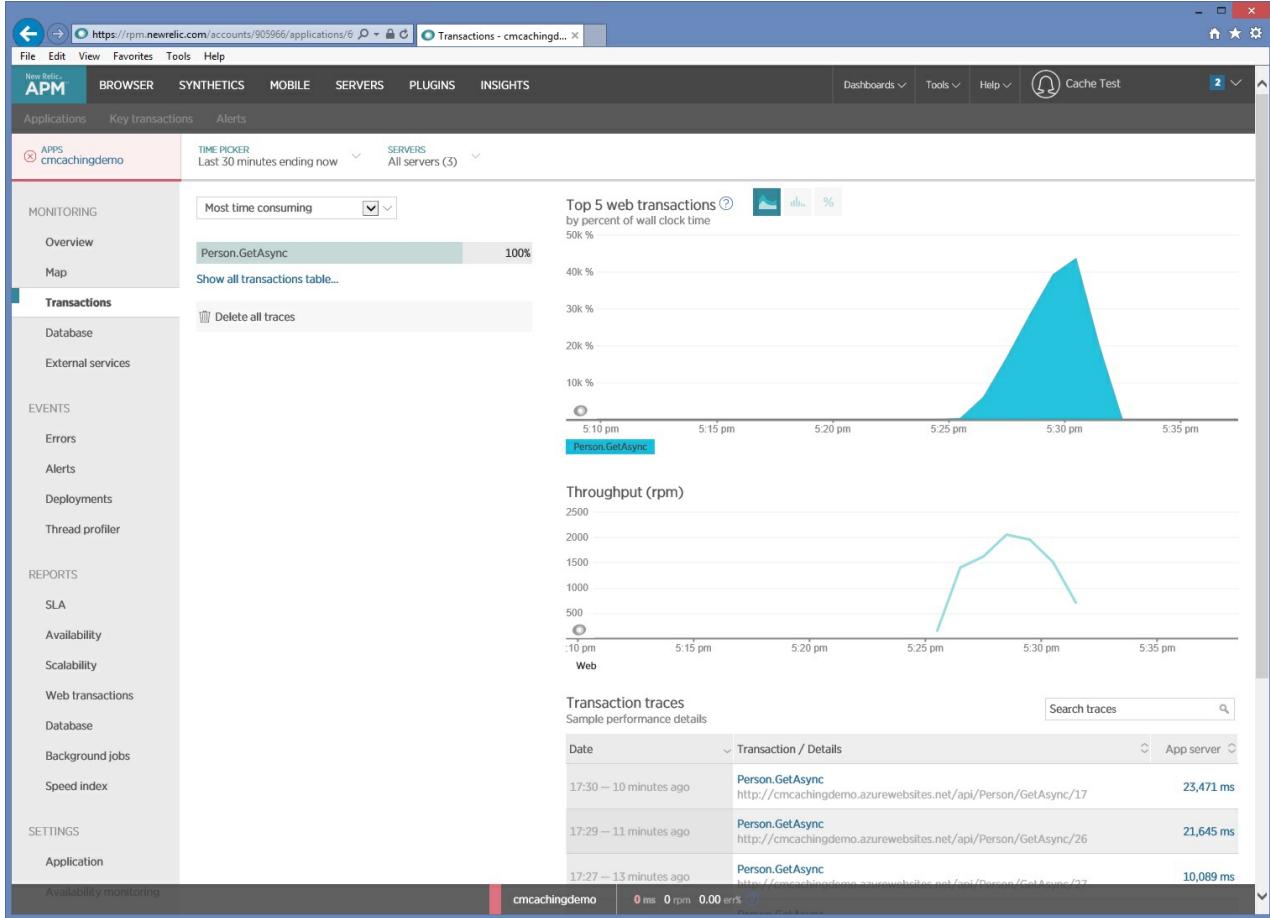
# Example diagnosis

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.

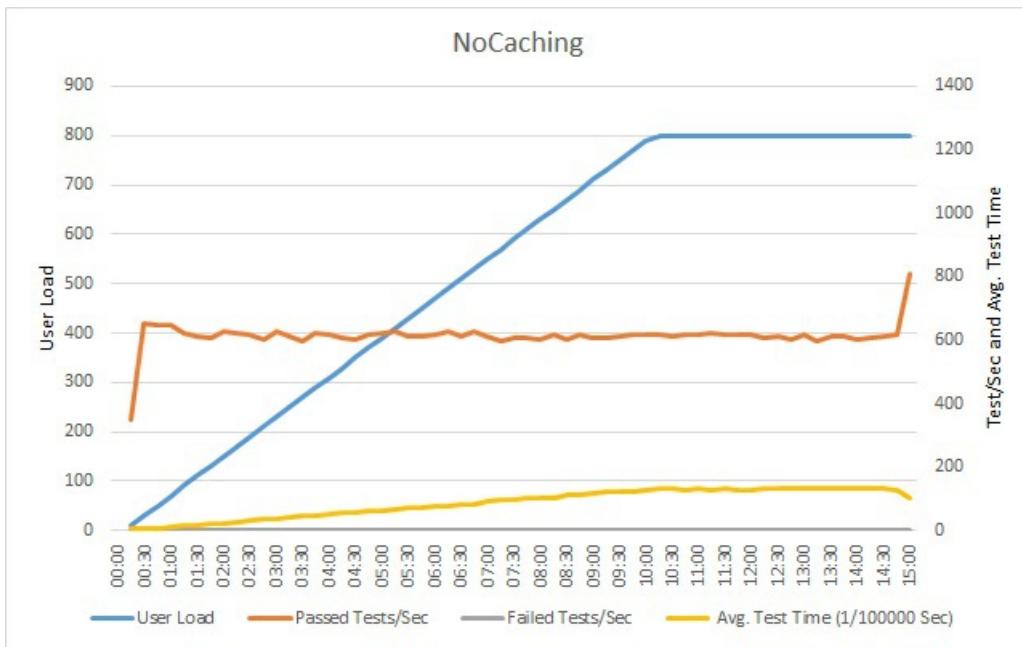
The following image shows monitoring data captured by [New Relic](#) during a load test. In this case, the only HTTP GET operation performed is `Person.GetAsync`. But in a live production environment, knowing the relative frequency that each request is performed can give you insight into which resources 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	SELECT UseCounts, Text, Query_Plan FROM sys.dm_exec_cached_plans WHERE query_plan IS NOT NULL	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	select is_federation_member from sys.databases where database_id = 1	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
256049	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1] WHERE [Extent1].[BusinessEntityId] = @p__linq_0	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
2	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
2	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
3	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
10	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
11	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
12	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
13	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
14	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
15	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
16	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
17	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
18	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
19	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
20	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
21	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
22	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
23	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
24	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">

Here is the SQL query that is causing so many database requests:

```

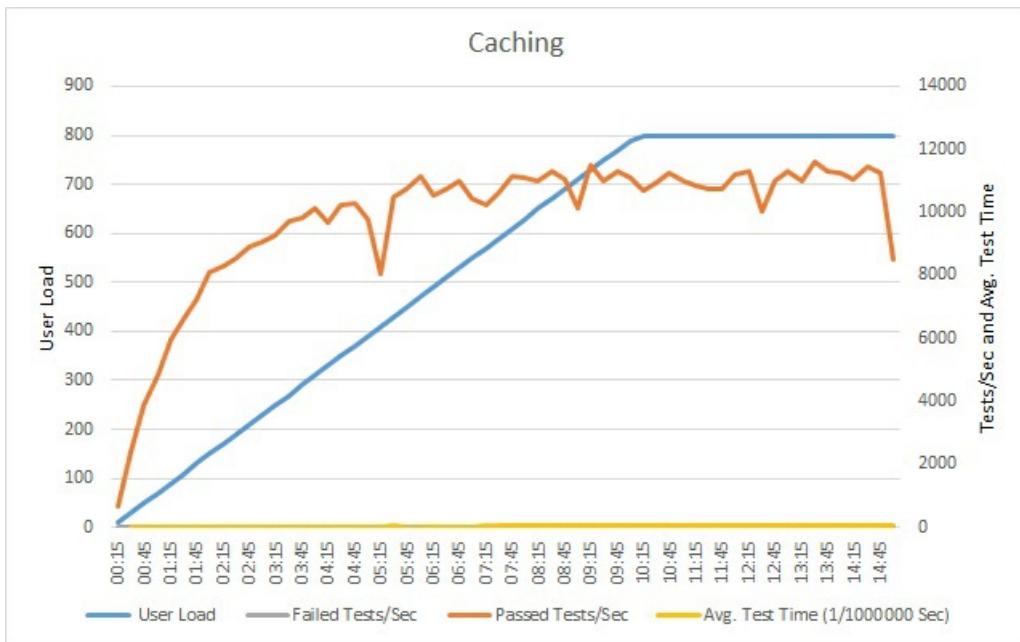
(@p__linq_0 int)SELECT TOP (2)
[Extent1].[BusinessEntityId] AS [BusinessEntityId],
[Extent1].[FirstName] AS [FirstName],
[Extent1].[LastName] AS [LastName]
FROM [Person].[Person] AS [Extent1]
WHERE [Extent1].[BusinessEntityId] = @p__linq_0

```

This is the query that Entity Framework generates in  `GetByIdAsync` method shown earlier.

### Implement the 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 1ms earlier — a 20× improvement.

## Related resources

- [API implementation best practices](#)
- [Cache-Aside Pattern](#)
- [Caching best practices](#)
- [Circuit Breaker pattern](#)

# Synchronous I/O antipattern

9/28/2018 • 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(() => LibraryIOperation());
}

```

Now the calling code can await on the wrapper:

```

// Invoke the asynchronous wrapper using a task
await LibraryIOperationAsync();

```

## 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 or appear to hang 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

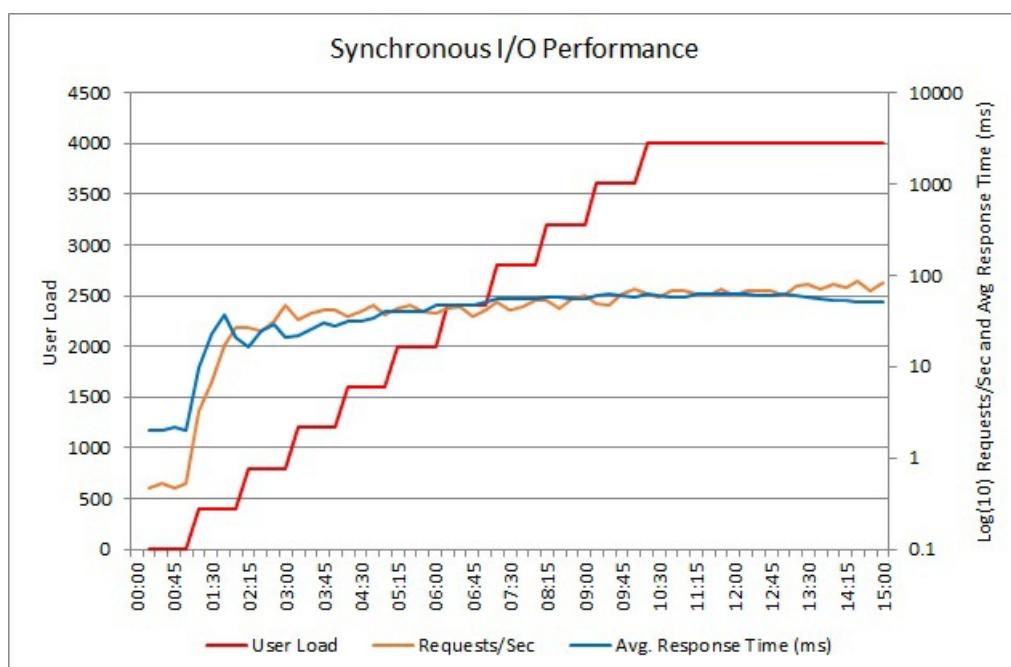
For Azure web applications and web roles, it's worth monitoring the performance of the IIS web server. In particular, pay attention to the request queue length to establish whether requests are being blocked waiting for available threads during periods of high activity. You can gather this information 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

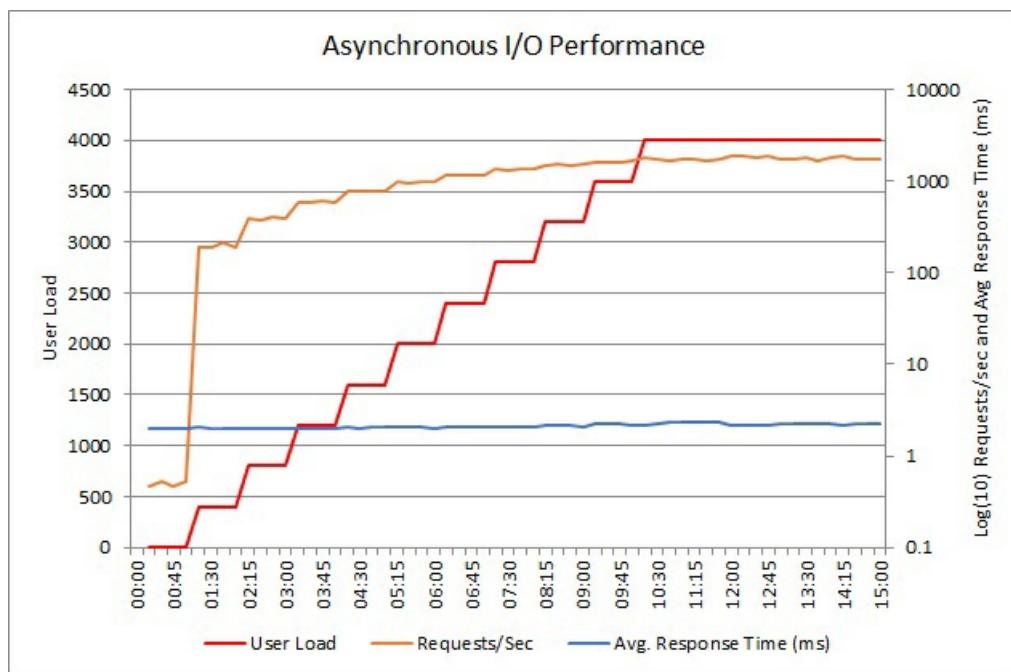
response time is slightly over 2 seconds. When the load reaches approximately 2500 concurrent users, the average response time reaches a plateau, although the volume of requests per second continues to increase. Note that the scale for these two measures is logarithmic. The number of requests per second 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.

# Azure for AWS Professionals

9/28/2018 • 15 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.

## Overview

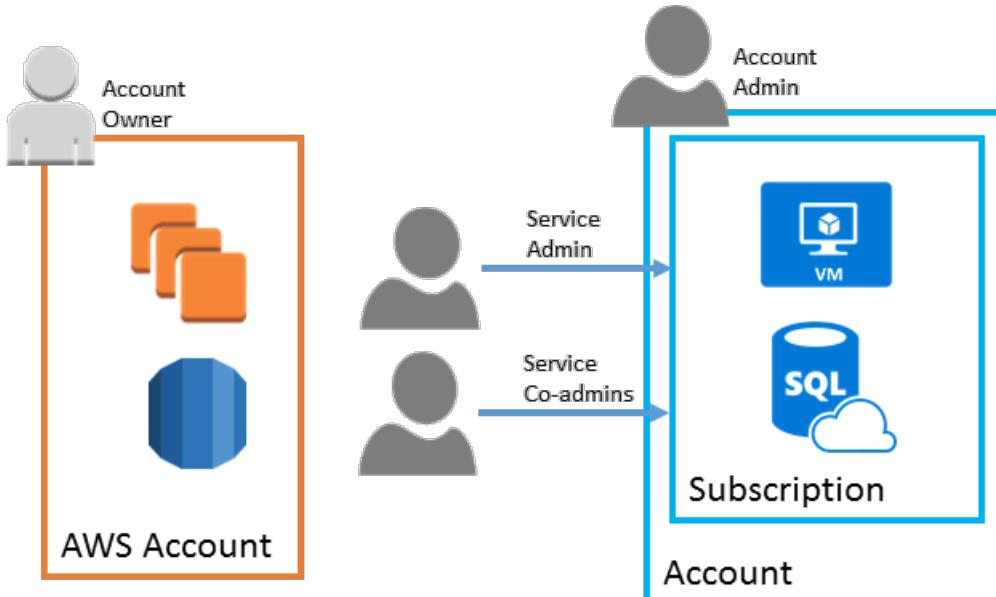
Like AWS, Microsoft Azure is built around a core set of compute, storage, database, and networking services. In many cases, both platforms offer a basic equivalence between the products and services they offer. Both AWS and Azure allow you to build highly available solutions based on Windows or Linux hosts. So, if 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. There are also cases where a particular service might be offered on one platform, but not the other. See [charts of comparable Azure and AWS services](#).

## Accounts and subscriptions

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*

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. There is only one service administrator 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.

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**

- [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 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 2.0 tool provides a command-line interface capable of creating and managing Azure resources. 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](#)

## Regions and zones (high availability)

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 data center, such as loss of power in a data center. 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, there is a tradeoff between greater redundancy and reliability versus higher cost and complexity.

In AWS, 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 a number of features to make an application redundant at every level of 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

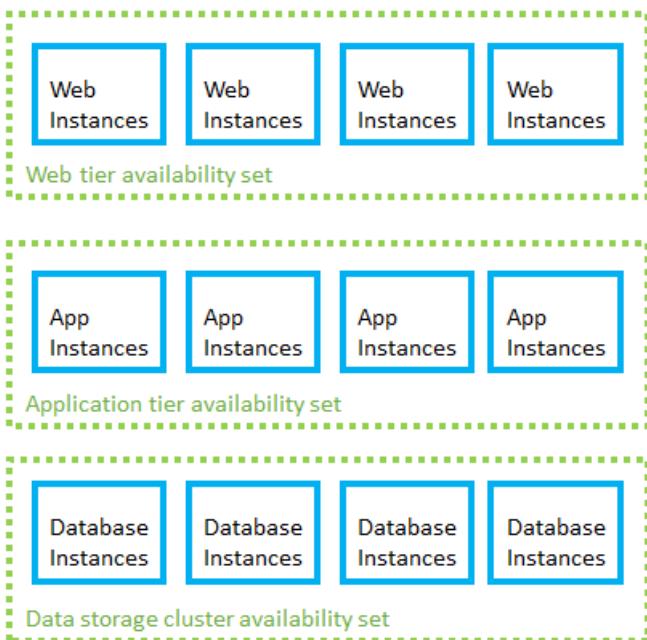
#### Availability sets

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 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 usually separated by at least 300 miles. This is intended to ensure larger scale disasters only impact 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 and availability for virtual machines in Azure](#)
- [High availability for Azure applications](#)
- [Disaster recovery for Azure applications](#)

- [Planned maintenance for Linux virtual machines in Azure](#)

## Services

For a listing of how services map between 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 details. You can find the uptime guarantees and downtime credit policies for each Azure product or service on the [Service Level Agreements](#) page.

The following sections provide a brief explanation of how commonly used features and services differ between the AWS and Azure platforms.

### Compute services

#### EC2 Instances and Azure virtual machines

Although AWS instance types and Azure virtual machine sizes breakdown in a similar way, there are differences in the RAM, CPU, and storage capabilities.

- [Amazon EC2 Instance Types](#)
- [Sizes for virtual machines in Azure \(Windows\)](#)
- [Sizes for virtual machines in Azure \(Linux\)](#)

Unlike AWS' per second billing, Azure on-demand VMs are billed by the minute.

#### 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 the same low-latency temporary read-write storage as EC2 Instance Storage (also called ephemeral storage).

Higher performance disk IO is supported using [Azure premium storage](#). This is similar to the Provisioned IOPS storage options provided by 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, Lambda functionality also overlaps with other Azure services:

- [WebJobs](#) - allow you to create scheduled or continuously running background tasks.
- [Logic Apps](#) - provides communications, integration, and business rule management services.

#### Autoscaling, Azure VM scaling, and Azure App Service Autoscale

Autoscaling in Azure is handled by two services:

- [VM 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 Container Service](#) supports Docker containers managed through Docker Swarm, Kubernetes, or DC/OS.

#### Other compute services

Azure offers several compute services that do not have direct equivalents in AWS:

- [Azure Batch](#) - allows you to manage compute-intensive work across a scalable collection of virtual machines.
- [Service Fabric](#) - platform for developing and hosting scalable [microservice](#) solutions.

## 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](#)

## Storage

### 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. 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.

## See also

- [Microsoft Azure Storage Performance and Scalability Checklist](#)
- [Azure Storage security guide](#)
- [Patterns & Practices: Content Delivery Network \(CDN\) guidance](#)

## Networking

### Elastic Load Balancing, Azure Load Balancer, and Azure Application Gateway

The Azure equivalents of the two Elastic Load Balancing services are:

- [Load Balancer](#) - provides the same capabilities as the AWS Classic Load Balancer, allowing you to distribute traffic for multiple VMs at the network level. It also provides 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.

#### **See also**

- [Create a virtual network using the Azure portal](#)
- [Plan and design Azure Virtual Networks](#)
- [Azure Network Security Best Practices](#)

## **Database services**

### **RDS and Azure relational database services**

Azure provides several different relational database services that are the equivalent of AWS' Relational Database Service (RDS).

- [SQL Database](#)
- [Azure Database for MySQL](#)
- [Azure Database for PostgreSQL](#)

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](#)
- [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 Nodejs](#)

## **Security and identity**

### **Directory service and Azure Active Directory**

Azure splits up directory services into the following offerings:

- [Azure Active Directory](#) - cloud based directory and identity management service.
- [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.

## **Web application firewall**

In addition to the [Application Gateway Web Application Firewall](#), you can also use [web application firewalls](#) from third-party vendors like [Barracuda Networks](#).

### **See also**

- [Getting started with Microsoft Azure security](#)
- [Azure Identity Management and access control security best practices](#)

## **Application and messaging services**

### **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.

### **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](#) - a cloud messaging service that allows communication between application components within the Azure platform.
- [Service Bus](#) - a more robust messaging system for connecting applications, services, and devices. Using the related [Service Bus relay](#), Service Bus can also connect to remotely hosted applications and services.

### **Device Farm**

The AWS Device Farm provides cross-device testing services. In Azure, [Xamarin Test Cloud](#) 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.

### **See also**

- [How to use Queue storage from Node.js](#)
- [How to use Service Bus queues](#)

## **Analytics and big data**

The [Cortana Intelligence Suite](#) is Azure's package of products and services designed to capture, organize, analyze, and visualize large amounts of data. The Cortana suite consists of the following services:

- [HDInsight](#) - managed Apache distribution that includes Hadoop, Spark, Storm, or HBase.
- [Data Factory](#) - provides data orchestration and data pipeline functionality.
- [SQL Data Warehouse](#) - large-scale relational data storage.
- [Data Lake Store](#) - large-scale storage optimized for big data analytics workloads.
- [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
- [PowerBI](#) - used to power data visualization.

### **See also**

- [Cortana Intelligence Gallery](#)
- [Understanding Microsoft big data solutions](#)

- [Azure Data Lake & Azure HDInsight Blog](#)

## Internet of Things

### See also

- [Get started with Azure IoT Hub](#)
- [Comparison of IoT Hub and Event Hubs](#)

## Mobile services

### Notifications

Notification Hubs do not support sending SMS or email messages, so third-party services are needed for those delivery types.

### See also

- [Create an Android app](#)
- [Authentication and Authorization in Azure Mobile Apps](#)
- [Sending push notifications with Azure Notification Hubs](#)

## Management and monitoring

### See also

- [Monitoring and diagnostics guidance](#)
- [Best practices for creating Azure Resource Manager templates](#)
- [Azure Resource Manager Quickstart templates](#)

## Next steps

- [Get started with Azure](#)
- [Azure solution architectures](#)
- [Azure Reference Architectures](#)

# AWS to Azure services comparison

10/26/2018 • 16 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.

In the following tables, there are multiple Azure services listed for some AWS services. The Azure services are similar to one another, but depth and breadth of capabilities vary.

[Download a PDF of the Azure & AWS Cloud Service Map](#)

## Azure and AWS for multicloud solutions

As the leading public cloud platforms, Azure and AWS each offer businesses 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

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Marketplace	AWS Marketplace	<a href="#">Azure Marketplace</a>	Easy-to-deploy and automatically configured third-party applications, including single virtual machine or multiple virtual machine solutions.

## Compute

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Virtual servers	Elastic Compute Cloud (EC2) Instances	<a href="#">Azure Virtual Machines</a>	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.
	Amazon Lightsail	<a href="#">Azure Virtual Machines &amp; Images</a>	Collection of virtual machine templates to select from when building out your virtual machine.

Area	AWS Service	Azure Service	Description
Container instances	EC2 Container Service (ECS)	<a href="#">Azure Container Service</a>	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.
	EC2 Container Registry	<a href="#">Azure Container Registry</a>	Allows customers to store Docker formatted images. Used to create all types of container deployments on Azure.
Microservices / container orchestrators	Elastic Container Service for Kubernetes (EKS)	<a href="#">Azure Kubernetes Service (AKS)</a>	Deploy orchestrated containerized applications with Kubernetes. Simplify monitoring and cluster management through auto upgrades and a built-in operations console.
		<a href="#">Service Fabric</a>	A compute service that orchestrates and manages the execution, lifetime, and resilience of complex, inter-related code components that can be either stateless or stateful.
		<a href="#">Service Fabric Mesh</a>	Fully managed service that enables developers to deploy microservices applications without managing virtual machines, storage, or networking.
		<a href="#">Azure Container Service (ACS)</a>	Quickly deploy a production ready Kubernetes, DC/OS, or Docker Swarm cluster
Serverless	Lambda	<a href="#">Azure Functions</a> <a href="#">Azure Event Grid</a>	Integrate systems and run backend processes in response to events or schedules without provisioning or managing servers.
Backend process logic		<a href="#">Web Jobs</a>	Provides an easy way to run background processes in an application context.
Batch computing	AWS Batch	<a href="#">Azure Batch</a>	Run large-scale parallel and high-performance computing applications efficiently in the cloud.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Scalability	AWS Auto Scaling	<a href="#">Virtual Machine Scale Sets</a> <a href="#">Azure App Service Scale Capability (PaaS)</a> <a href="#">Azure AutoScaling</a>	Lets you automatically change the number of instances providing a particular compute workload. You set defined metric and thresholds that determine if the platform adds or removes instances.

## Storage

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Object storage	Simple Storage Services (S3)	<a href="#">Azure Storage—Block Blob (for content logs, files) (Standard—Hot)</a>	Object storage service, for use cases including cloud applications, content distribution, backup, archiving, disaster recovery, and big data analytics.
Virtual Server disk infrastructure	Elastic Block Store (EBS)	<a href="#">Azure Storage Disk—Page Blobs (for VHDs or other random-write type data)</a> <a href="#">Azure Storage Disks—Premium Storage</a>	SSD storage optimized for I/O intensive read/write operations. For use as high performance Azure virtual machine storage.
Shared file storage	Elastic File System	<a href="#">Azure Files (file share between VMs)</a>	Provides a simple interface to create and configure file systems quickly, and share common files. It's shared file storage without the need for a supporting virtual machine, and can be used with traditional protocols that access files over a network.
Archiving—cool storage	S3 Infrequent Access (IA)	<a href="#">Azure Storage—Standard Cool</a>	Cool storage is a lower cost tier for storing data that is infrequently accessed and long-lived.
Archiving—cold storage	S3 Glacier	<a href="#">Azure Storage—Standard Archive</a>	Archive storage has the lowest storage cost and higher data retrieval costs compared to hot and cool storage.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Backup	None	<a href="#">Azure Backup</a>	Backup and archival solutions allow files and folders to be backed up and recovered from the cloud, and provide off-site protection against data loss. There are two components of backup—the software service that orchestrates backup/retrieval and the underlying backup storage infrastructure.
Hybrid storage	Storage Gateway	<a href="#">StorSimple</a>	Integrates on-premises IT environments with cloud storage. Automates data management and storage, plus supports disaster recovery.
Bulk data transfer	AWS Import/Export Disk	<a href="#">Import/Export</a>	A data transport solution that uses secure disks and appliances to transfer large amounts of data. Also offers data protection during transit.
	AWS Import/Export Snowball  AWS Snowball Edge  AWS Snowmobile	<a href="#">Azure Data Box</a>	Petabyte- to Exabyte-scale data transport solution that uses secure data storage devices to transfer large amounts of data into and out of the AWS cloud, at lower cost than Internet-based transfers.
Disaster recovery	None	<a href="#">Site Recovery</a>	Automates protection and replication of virtual machines. Offers health monitoring, recovery plans, and recovery plan testing.

## Networking and content delivery

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Cloud virtual networking	Virtual Private Cloud (VPC)	<a href="#">Virtual Network</a>	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.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Cross-premises connectivity	AWS VPN Gateway	<a href="#">Azure VPN Gateway</a>	Azure VPN Gateways connect Azure virtual networks to other Azure virtual networks, or customer on-premises networks (Site To Site). It also allows end users to connect to Azure services through VPN tunneling (Point To Site).
Domain name system management	Route 53	<a href="#">Azure DNS</a>	Manage your DNS records using the same credentials and billing and support contract as your other Azure services
	Route 53	<a href="#">Traffic Manager</a>	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.
Content delivery network	CloudFront	<a href="#">Azure Content Delivery Network</a>	A global content delivery network that delivers audio, video, applications, images, and other files.
Dedicated network	Direct Connect	<a href="#">ExpressRoute</a>	Establishes a dedicated, private network connection from a location to the cloud provider (not over the Internet).
Load balancing	Classic Load Balancer Network Load Balancer Application Load Balancer	<a href="#">Load Balancer</a> <a href="#">Application Gateway</a>	Automatically distributes incoming application traffic to add scale, handle failover, and route to a collection of resources.

## Database

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Relational database	RDS	<a href="#">SQL Database</a> <a href="#">Azure Database for MySQL</a> <a href="#">Azure Database for PostgreSQL</a>	Relational database-as-a-service (DBaaS) where the database resilience, scale, and maintenance are primarily handled by the platform.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
NoSQL—document storage	DynamoDB	<a href="#">Azure Cosmos DB</a>	A globally distributed, multi-model database that natively supports multiple data models: key-value, documents, graphs, and columnar.
NoSQL—key/value storage	DynamoDB and SimpleDB	<a href="#">Table Storage</a>	A nonrelational data store for semi-structured data. Developers store and query data items via web services requests.
Caching	ElastiCache	<a href="#">Azure Redis Cache</a>	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	<a href="#">Azure Database Migration Service</a>	Typically is focused on the migration of database schema and data from one database format to a specific database technology in the cloud.

## Analytics and big data

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic data warehouse	Redshift	<a href="#">SQL Data Warehouse</a>	A fully managed data warehouse that analyzes data using business intelligence tools. It can transact SQL queries across relational and nonrelational data.
Big data processing	Elastic MapReduce (EMR)	<a href="#">HDInsight</a>	Supports technologies that break up large data processing tasks into multiple jobs, and then combine the results to enable massive parallelism.
Data orchestration	Data Pipeline	<a href="#">Data Factory</a>	Processes and moves data between different compute and storage services, as well as on-premises data sources at specified intervals. Users can create, schedule, orchestrate, and manage data pipelines.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
	AWS Glue	Data Factory Data Catalog	Cloud-based ETL/data integration service that orchestrates and automates the movement and transformation of data from various sources.
Analytics	Kinesis Analytics	Stream Analytics 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.
Visualization	QuickSight	PowerBI	Business intelligence tools that build visualizations, perform ad hoc analysis, and develop business insights from data.
	None	Power BI Embedded	Allows visualization and data analysis tools to be embedded in applications.
Search	Elasticsearch Service	Marketplace—Elasticsearch	A scalable search server based on Apache Lucene.
	CloudSearch	Azure Search	Delivers full-text search and related search analytics and capabilities.
Machine learning	Machine Learning	Azure Machine Learning Studio Azure Machine Learning Service	Produces an end-to-end workflow to create, process, refine, and publish predictive models that can be used to understand what might happen from complex data sets.
Data discovery	None	Data Catalog	Provides the ability to better register, enrich, discover, understand, and consume data sources.
	Amazon Athena	Azure Data Lake Analytics	Provides a serverless interactive query service that uses standard SQL for analyzing databases.

## Intelligence

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
------	-------------	---------------	-------------

Area	AWS Service	Azure Service	Description
Conversational user interfaces virtual personal assistant	Alexa Skills Kits	<a href="#">Cortana Intelligence Suite — Cortana Integration</a>	Services cover intelligence cognitive services, machine learning, analytics, information management, big data and dashboards and visualizations.
		<a href="#">Microsoft Bot Framework + Azure Bot Service</a>	Builds and connects intelligent bots that interact with your users using text/SMS, Skype, Teams, Slack, Office 365 mail, Twitter, and other popular services.
Speech recognition	Amazon Lex	<a href="#">Bing Speech API</a>	API capable of converting speech to text, understanding intent, and converting text back to speech for natural responsiveness.
		<a href="#">Language Understanding Intelligent Service (LUIS)</a>	Allows your applications to understand user commands contextually.
		<a href="#">Speaker Recognition API</a>	Gives your app the ability to recognize individual speakers.
		<a href="#">Custom Recognition Intelligent Service (CRIS)</a>	Fine-tunes speech recognition to eliminate barriers such as speaking style, background noise, and vocabulary.
Text to Speech	Amazon Polly	<a href="#">Bing Speech API</a>	Enables both Speech to Text, and Text into Speech capabilities.
Visual recognition	Amazon Rekognition	<a href="#">Computer Vision API</a>	Distills actionable information from images, generates captions and identifies objects in images.
		<a href="#">Face API</a>	Detects, identifies, analyzes, organizes, and tags faces in photos.
		<a href="#">Emotions API</a>	Recognizes emotions in images.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
		<a href="#">Video API</a>	Intelligent video processing produces stable video output, detects motion, creates intelligent thumbnails, detects and tracks faces.

## Internet of things (IoT)

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Internet of Things	AWS IoT Other Services (Kinesis, Machine Learning, EMR, Data Pipeline, SNS, QuickSight)	<a href="#">Azure IoT Suite (IoT Hub, Machine Learning, Stream Analytics, Notification Hubs, PowerBI)</a>	Provides a preconfigured solution for monitoring, maintaining, and deploying common IoT scenarios.
	AWS IoT	<a href="#">Azure IoT Hub</a>	A cloud gateway for managing bidirectional communication with billions of IoT devices, securely and at scale.
Edge compute for IoT	AWS Greengrass	<a href="#">Azure IoT Edge</a>	Managed service that deploys cloud intelligence directly on IoT devices to run in on-prem scenarios.
Streaming data	Kinesis Firehose Kinesis Streams	<a href="#">Event Hubs</a>	Services that allow the mass ingestion of small data inputs, typically from devices and sensors, to process and route the data.

## Management and monitoring

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Cloud advisor	Trusted Advisor	<a href="#">Azure Advisor</a>	Provides analysis of cloud resource configuration and security so subscribers can ensure they're making use of best practices and optimum configurations.
Deployment orchestration (DevOps)	OpsWorks (Chef-based)	<a href="#">Azure Automation</a>	Configures and operates applications of all shapes and sizes, and provides templates to create and manage a collection of resources.

Area	AWS Service	Azure Service	Description
	CloudFormation	Azure Resource Manager VM extensions Azure Automation	Provides a way for users to automate the manual, long-running, error-prone, and frequently repeated IT tasks.
Management & monitoring (DevOps)	CloudWatch	Azure portal Azure Monitor	A unified console that simplifies building, deploying, and managing your cloud resources.
	CloudWatch	Azure Application Insights + Azure Monitor	An extensible analytics service that helps you understand the performance and usage of your live web application. It's designed for developers, to help you continuously improve the performance and usability of your app.
	AWS X-Ray	Azure Application Insights + Azure Monitor	An extensible application performance management service for web developers on multiple platforms. You can use it to monitor your live web application, detect performance anomalies, and diagnose issues with your app.
	AWS Usage and Billing Report	Azure Billing API	Services to help generate, monitor, forecast, and share billing data for resource usage by time, organization, or product resources.
	AWS Management Console	Azure portal	A unified management console that simplifies building, deploying, and operating your cloud resources.
Administration	AWS Application Discovery Service	Azure Log Analytics in Operations Management Suite	Provides deeper insights into your application and workloads by collecting, correlating and visualizing all your machine data, such as event logs, network logs, performance data, and much more, from both on-premises and cloud assets.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
	Amazon EC2 Systems Manager	<a href="#">Microsoft Operations Management Suite—Automation and Control functionalities</a>	Enables continuous IT services and compliance through process automation and configuration management. You can transform complex and repetitive tasks with IT automation.
	AWS Personal Health Dashboard	<a href="#">Azure Resource Health</a>	Provides detailed information about the health of resources as well as recommended actions for maintaining resource health.
	Third Party	<a href="#">Azure Storage Explorer</a>	Standalone app from Microsoft that allows you to easily work with Azure Storage data on Windows, Mac OS, and Linux.

## Mobile services

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Pro app development	Mobile Hub	<a href="#">Mobile Apps</a> <a href="#">Xamarin Apps</a>	Provides backend mobile services for rapid development of mobile solutions, identity management, data synchronization, and storage and notifications across devices.
	Mobile SDK	<a href="#">Mobile Apps</a>	Provides the technology to rapidly build cross-platform and native apps for mobile devices.
	Cognito	<a href="#">Mobile Apps</a>	Provides authentication capabilities for mobile applications.
App testing	AWS Device Farm	<a href="#">Xamarin Test Cloud (front end)</a>	Provides services to support testing mobile applications.
Analytics	Mobile Analytics	<a href="#">HockeyApp</a> <a href="#">Application Insights</a>	Supports monitoring, and feedback collection for the debugging and analysis of a mobile application service quality.

Area	AWS Service	Azure Service	Description
Enterprise mobility management	None	Intune	Provides mobile device management, mobile application management, and PC management capabilities from the cloud.

## Security, identity, and access

Area	AWS Service	Azure Service	Description
Authentication and authorization	Identity and Access Management (IAM)	Azure Active Directory Azure Active Directory Premium	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 Organizations	Azure Subscription and Service Management + Azure RBAC	Security policy and role management for working with multiple accounts.
	Multi-Factor Authentication	Multi-Factor Authentication	Helps safeguard access to data and applications while meeting user demand for a simple sign-in process. It delivers strong authentication with a range of verification options, allowing users to choose the method they prefer.
Information protection	None	Azure Information Protection	Service to help control and secure email, documents, and sensitive data that you share outside your company walls.
Encryption	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.
	Key Management Service 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).

Area	AWS Service	Azure Service	Description
Firewall	Web Application Firewall	<a href="#">Application Gateway Web Application Firewall</a>	A firewall that protects web applications from common web exploits. Users can define customizable web security rules.
Security	Inspector	<a href="#">Security Center</a>	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	<a href="#">App Service Certificates available on the Portal</a>	Service that allows customers to create, manage and consume certificates seamlessly in the cloud.
	GuardDuty	Azure AD, Operations Management Suite (OMS), Security Center	Azure offers built-in advanced threat detection functionality, which can be configured and customized to meet your requirements. For more information, see <a href="#">Azure advanced threat detection</a> .
Directory services	AWS Directory Service + Windows Server Active Directory on AWS	<a href="#">Azure Active Directory Domain Services + Windows Server Active Directory on Azure IaaS</a>	Comprehensive identity and access management cloud solution that provides a robust set of capabilities to manage users and groups. It helps secure access to on-premises and cloud applications, including Microsoft online services like Office 365 and many non-Microsoft SaaS applications.
	Cognito	<a href="#">Azure Active Directory B2C</a>	A highly available, global, identity management service for consumer-facing applications that scales to hundreds of millions of identities.
	AWS Directory Service	<a href="#">Windows Server Active Directory</a>	Services for supporting Microsoft Active Directory in the cloud.
Compliance	AWS Artifact	<a href="#">Service Trust Platform</a>	Provides access to audit reports, compliance guides, and trust documents from across cloud services.

Area	AWS Service	Azure Service	Description
Security	AWS Shield	Azure DDoS Protection Service	Provides cloud services with protection from distributed denial of services (DDoS) attacks.

## Developer tools

Area	AWS Service	Azure Service	Description
Media transcoding	Elastic Transcoder	Media Services	Services that offer broadcast-quality video streaming services, including various transcoding technologies.
Email	Simple Email Service (SES)	Marketplace—Email	Services for integrating email functionality into applications.
Messaging	Simple Queue Service (SQS)	Azure Queue Storage	Provides a managed message queueing service for communicating between decoupled application components.
Messaging	Simple Queue Service (SQS)	Service Bus Queues, Topics, Relays	Supports a set of cloud-based, message-oriented middleware technologies including reliable message queuing and durable publish/subscribe messaging.
Workflow	Simple Workflow Service (SWF)	Logic Apps	Serverless technology for connecting apps, data and devices anywhere—on-premises or in the cloud for large ecosystems of SaaS and cloud based connectors.
API management	API Gateway	API Management	A turnkey solution for publishing APIs to external and internal consumers.
	Elastic Beanstalk	Web Apps (App Service) Cloud Services API Apps (App Service)	Managed hosting platforms providing easy to use services for deploying and scaling web applications and services.
	CodeDeploy CodeCommit CodePipeline	Azure DevOps	A cloud service for collaborating on code development.

Area	AWS Service	Azure Service	Description
	AWS Developer Tools	<a href="#">Azure Developer Tools</a>	Collection of tools for building, debugging, deploying, diagnosing, and managing multi-platform, scalable apps and services.
		<a href="#">Power Apps</a>	Technology to rapidly build business solutions, connecting to existing services and data sources such as Excel, SharePoint, Dynamics 365, and more using a visual designer.
App testing	None	<a href="#">Azure DevTest Labs (backend)</a>	Testing technology to build out heterogeneous solutions for testing cross-platform functionality to your dev/test environment. Integrates to a full DevOps Continuous Integration/Deployment with Visual Studio Online service and 3rd parties such as Jenkins, Chef, Puppet, CloudTest Lite, Octopus Deploy, and others.
App customer payment service	Amazon Flexible Payment Service and Amazon Dev Pay	None	Cloud service that provides developers a payment service for their cloud based applications.
Game development (cloud-based tools)	GameLift	None	AWS managed service for hosting dedicated game servers.
	Lumberyard	None	Game engine integrated with AWS and Twitch.
DevOps	AWS CodeBuild	<a href="#">Visual Studio Team Services</a>	Fully managed build service that supports continuous integration and deployment.
Backend process logic	AWS Step Functions	<a href="#">Logic Apps</a>	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.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Programmatic access	Command Line Interface	<a href="#">Azure Command Line Interface (CLI)</a> <a href="#">Azure PowerShell</a>	Built on top of the native REST API across all cloud services, various programming language-specific wrappers provide easier ways to create solutions.
Predefined templates	AWS Quick Start	<a href="#">Azure Quickstart templates</a>	Community-led templates for creating and deploying virtual machine-based solutions.

## Enterprise integration

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Enterprise app integration	None	<a href="#">Logic Apps</a>	Provides out-of-the box line-of-business application integration for SAP, Oracle, SQL Server, and Websphere MQ. Connect apps, data, and devices on-premises or in the cloud with our large ecosystem of SaaS and cloud-based connectors, including Salesforce, Office 365, Twitter, Dropbox, Google Services, and more.
Enterprise application services	None	<a href="#">Dynamics 365</a>	Dynamics 365 delivers the full spectrum of CRM through five individual apps — Sales, Customer Service, Field Service, Project Service Automation, and Marketing —that work seamlessly together.
	Amazon WorkMail Amazon WorkDocs	<a href="#">Office 365</a>	Fully integrated Cloud service providing communications, email, document management in the cloud and available on a wide variety of devices.
Content management in the cloud	None	<a href="#">SharePoint Online</a>	Provides a collaborative way for individuals, teams, and organizations to intelligently discover, share, and collaborate on content from anywhere and on any device.
Commercial PaaS-IaaS-DBaaS framework	None	<a href="#">Azure Stack</a>	A hybrid cloud platform that lets you deliver Azure services from your organization's datacenter.



Example scenarios walk through the process of solving specific architectural and business problems on Azure. Each scenario is based on a real customer example and is intended to provide fast, easy to read guidance to help accelerate your own implementation.

Within each scenario you can find an architecture diagram, data flows, and details of all of the components. You can also learn about alternatives technologies that may better fit your business need, rough pricing of the solution, and considerations when running the solution in production. Many scenarios will also contain a method to easily deploy the solution in your own Azure account.

Jump to: [AI scenarios](#) | [Application scenarios](#) | [Data scenarios](#) | [Infrastructure scenarios](#)

## New Scenarios

[Decentralized trust between banks on Azure](#) [Highly scalable and secure WordPress website](#) [Data warehousing & analytics for sales and marketing](#)

## Top Scenarios

[Image classification for insurance claims](#) [Conversational chatbot for hotel reservations](#) [Secure Windows web application for regulated industries](#)

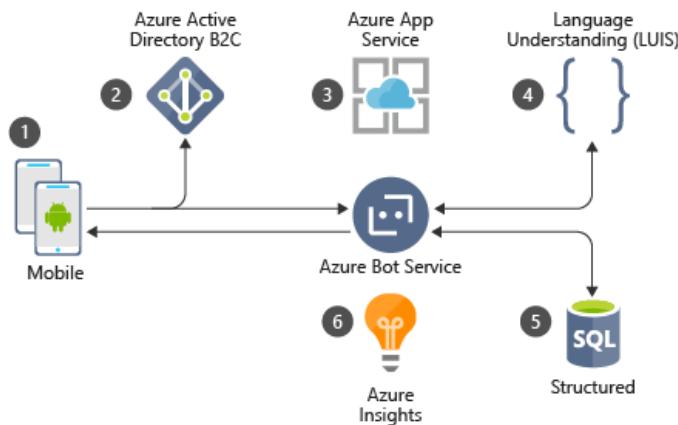
## Submit your idea for a new scenario

Do you have a scenario that you'd like us to create?

Would you like to build one yourself?

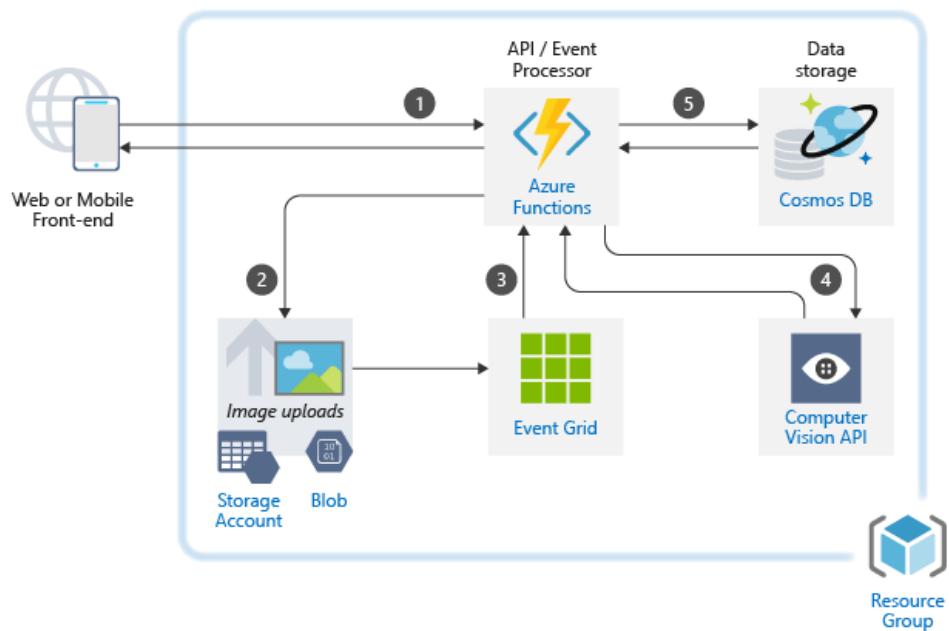
[Submit your idea here!](#)

## AI Scenarios



## Conversational chatbot for hotel reservations on Azure

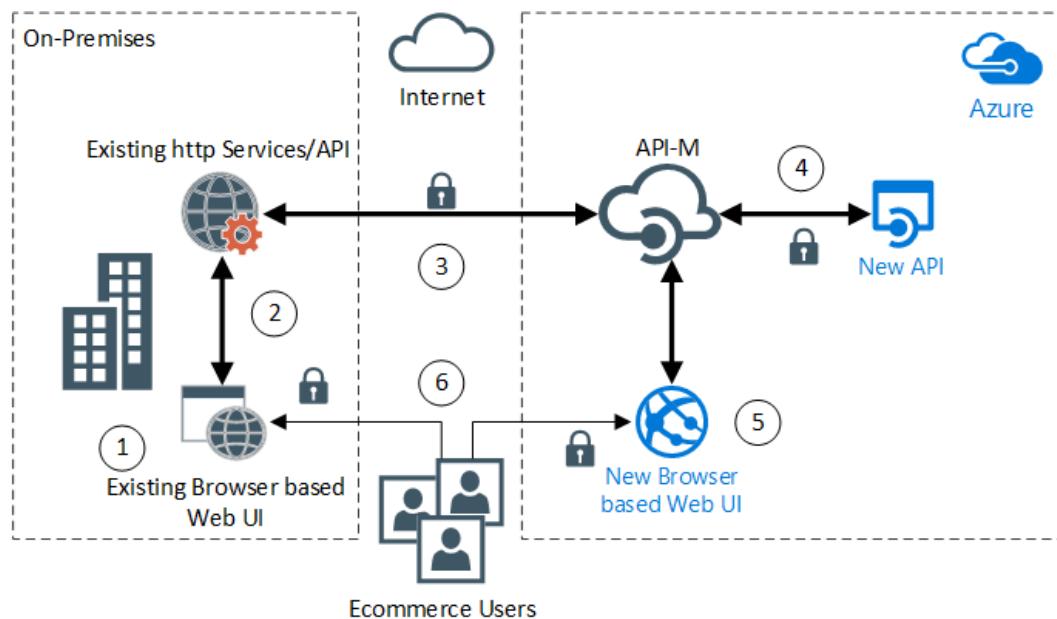
Build a conversational chatbot for commerce applications with Azure Bot Service.



## Image classification for insurance claims on Azure

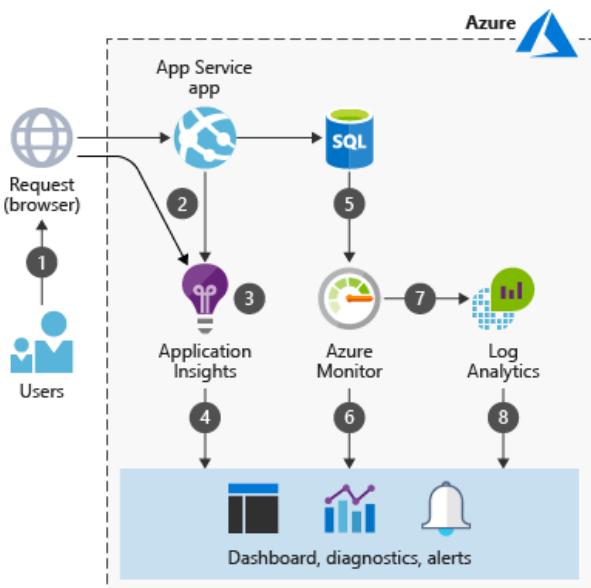
Build image processing into your Azure applications.

## Application Scenarios



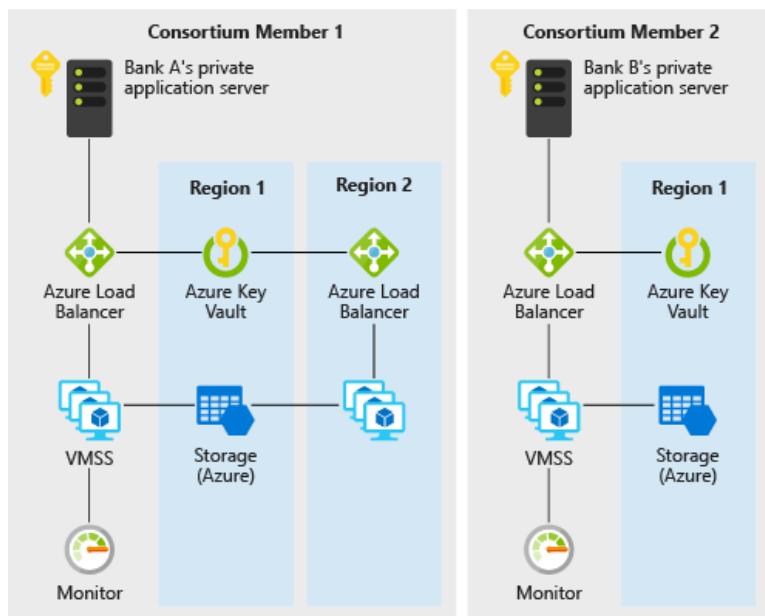
## Migrating a legacy web application to an API-based architecture on Azure

Use Azure API Management to modernize a legacy web application.



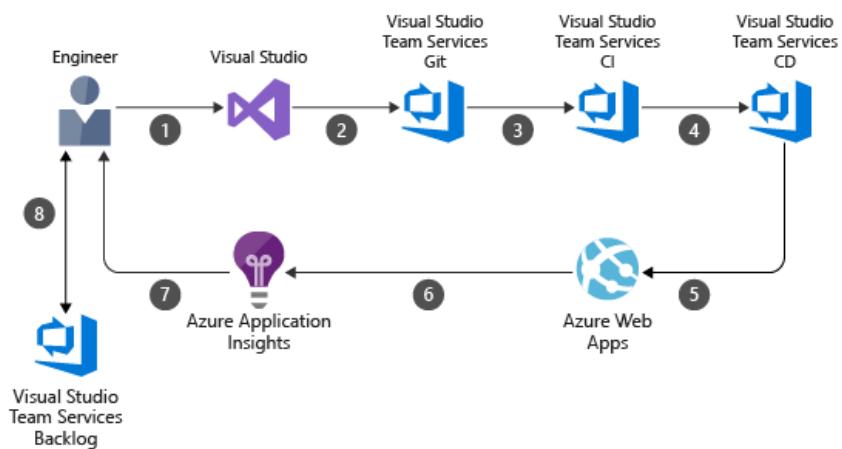
## Web application monitoring on Azure

Monitor a web application hosted in Azure App Service.



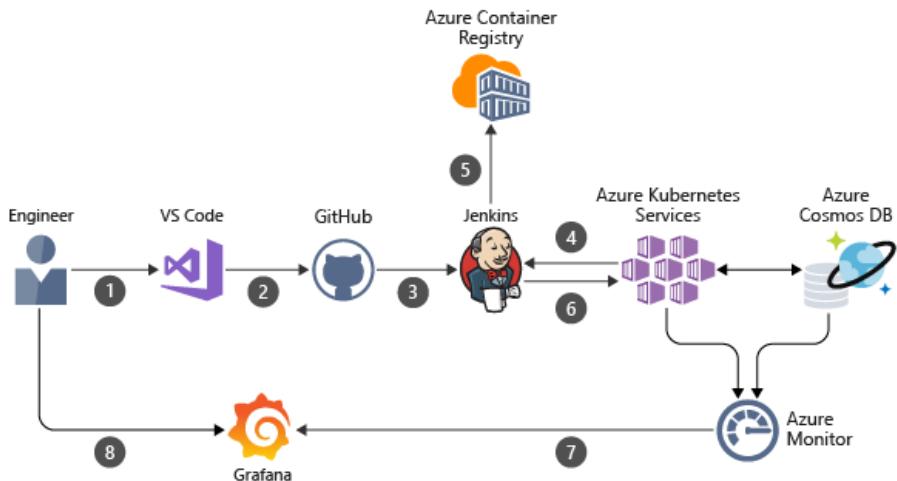
## Decentralized trust between banks on Azure

Establish a trusted environment for communication and information sharing without resorting to a centralized database.



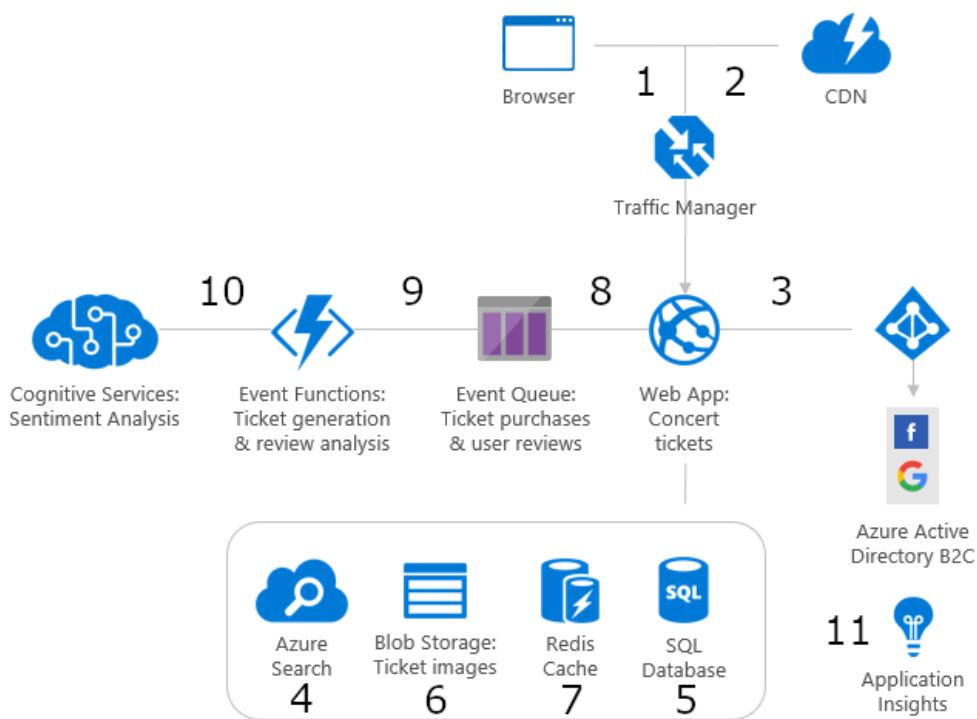
## CI/CD pipeline with Azure DevOps

Build and release a .NET app to Azure Web Apps using Azure DevOps.



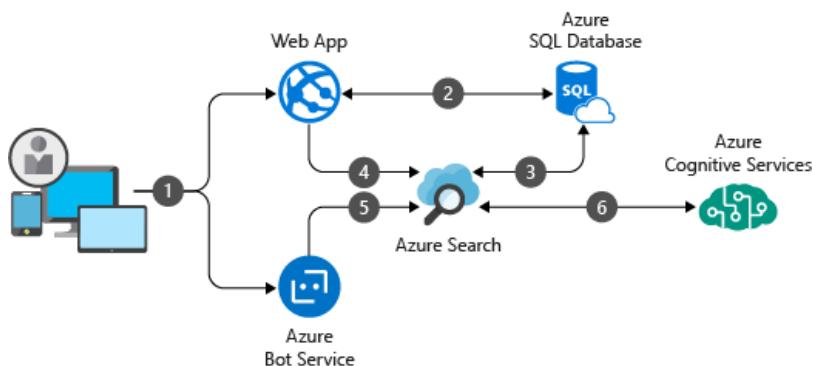
## CI/CD pipeline for container-based workloads

Build a DevOps pipeline for a Node.js web app with Jenkins, Azure Container Registry, Azure Kubernetes Service, Cosmos DB, and Grafana.



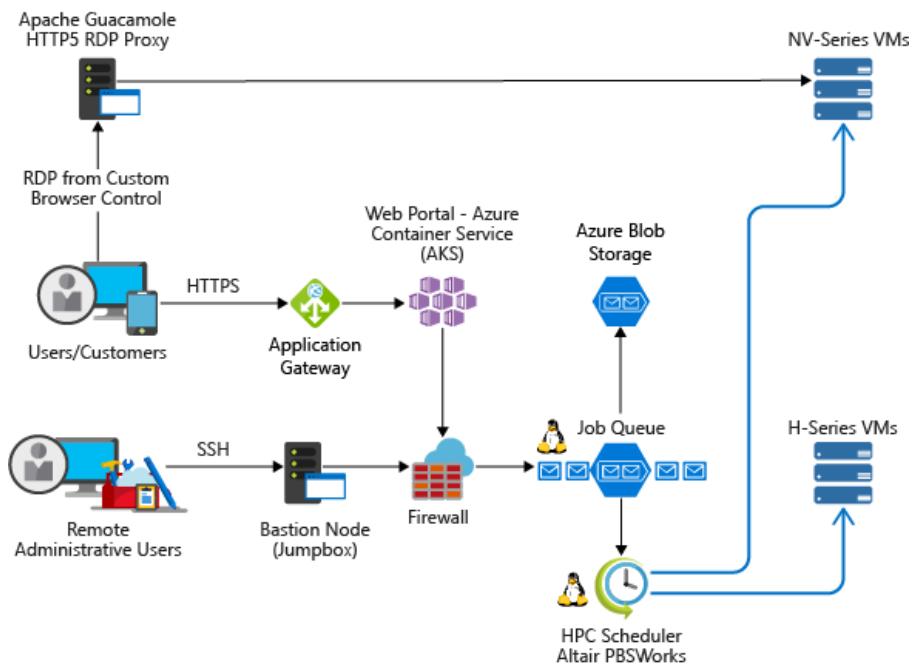
## E-commerce front end on Azure

Host an e-commerce site on Azure.



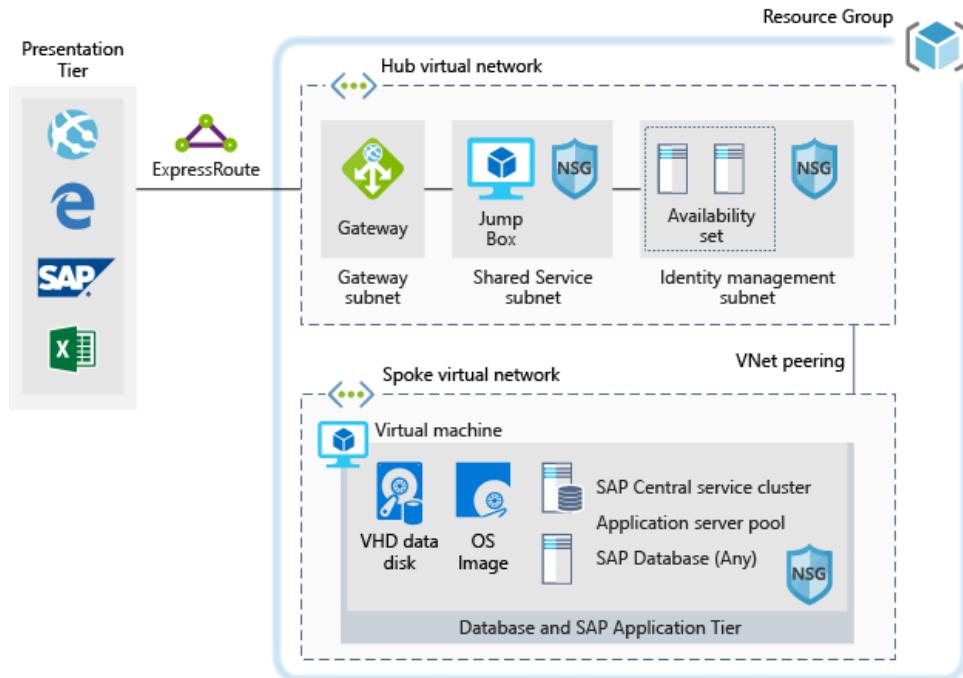
## Intelligent product search engine for e-commerce

Provide a world-class search experience in an e-commerce application.



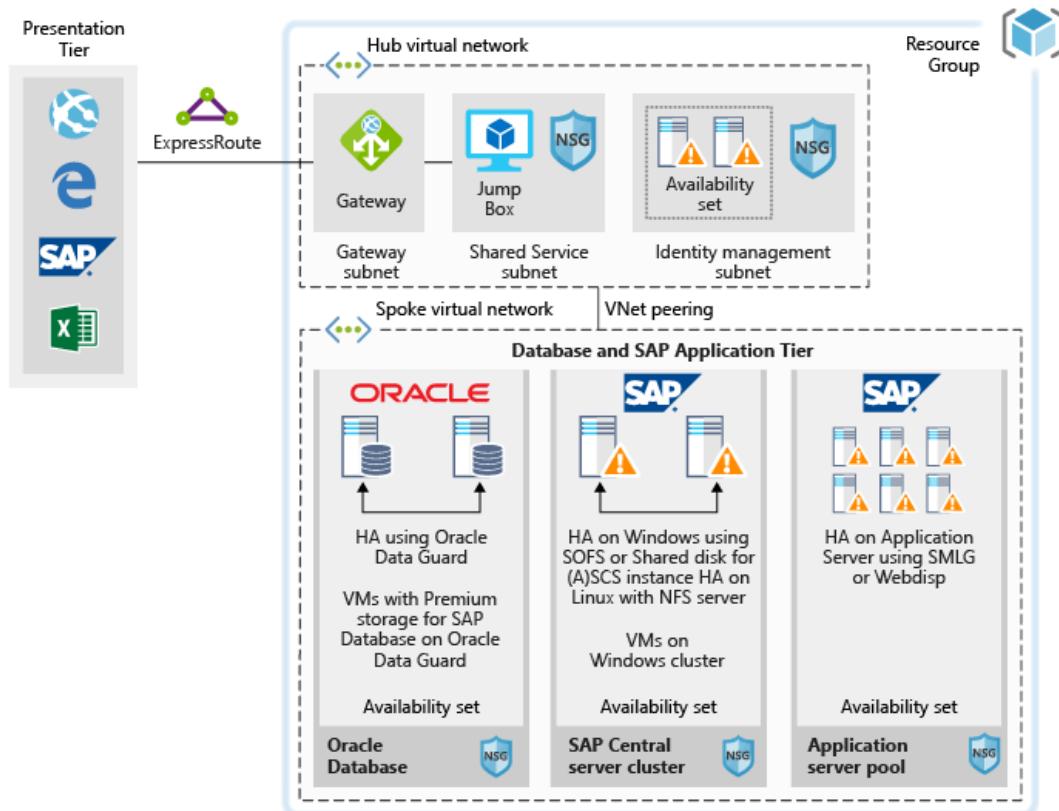
## A computer-aided engineering service on Azure

Provide a software-as-a-service (SaaS) platform for computer-aided engineering (CAE) on Azure.



## Dev/test environments for SAP workloads on Azure

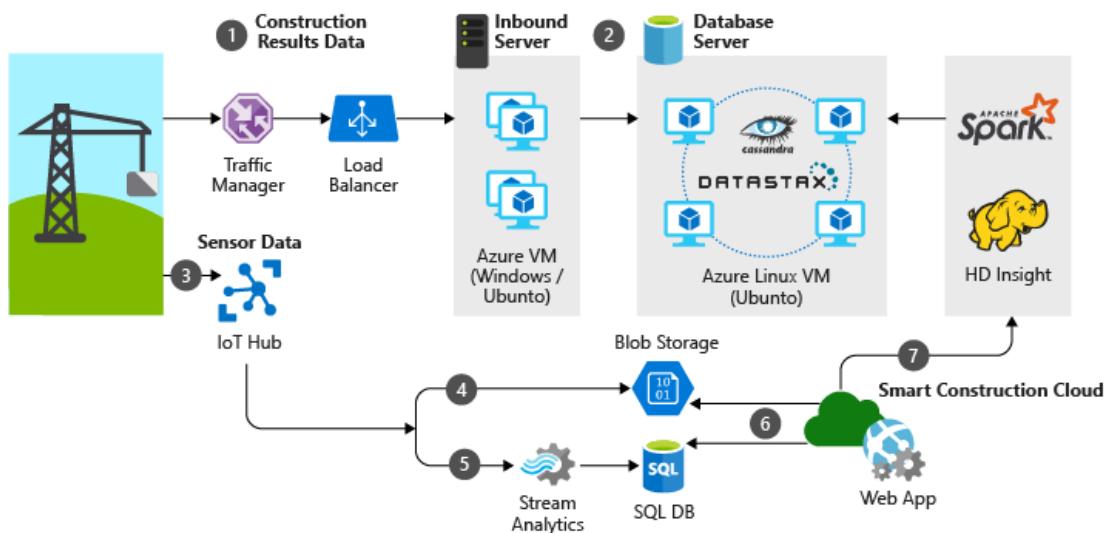
Build a dev/test environment for SAP workloads.



## Running SAP production workloads using an Oracle database on Azure

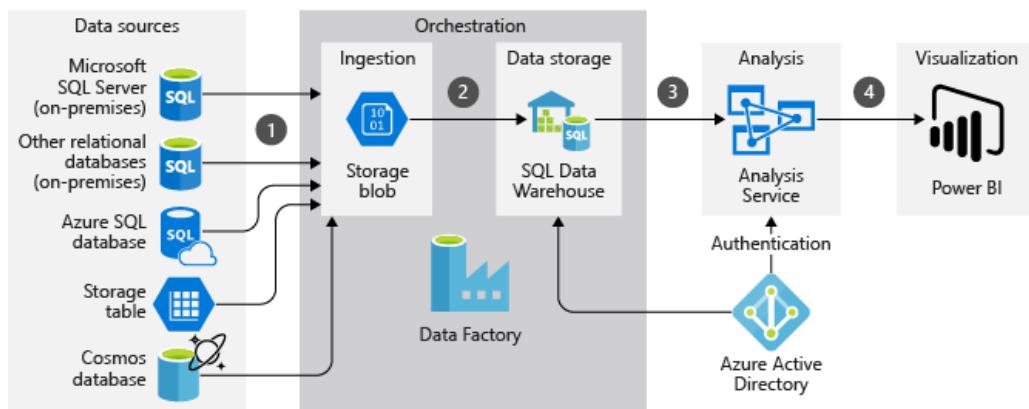
Run an SAP production deployment in Azure using an Oracle database.

## Data Scenarios



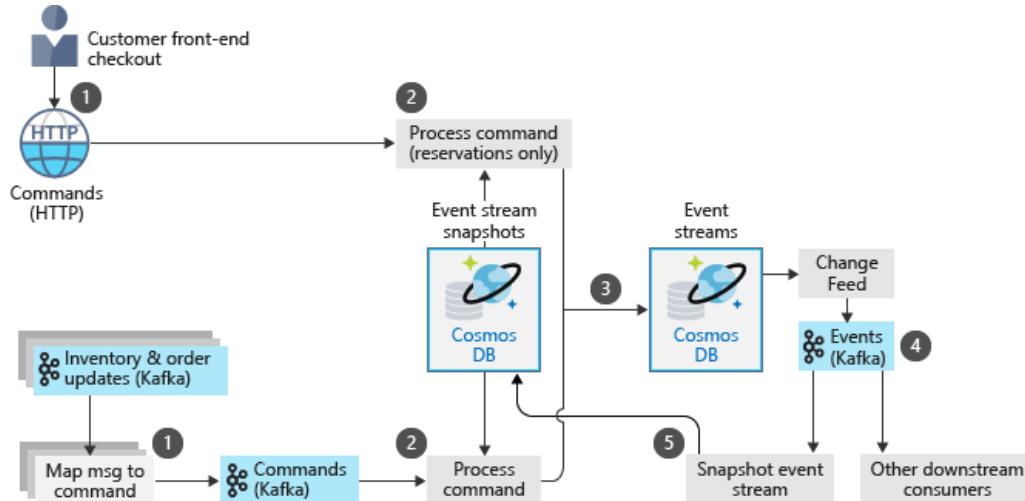
## IoT and data analytics in the construction industry

Use IoT devices and data analytics to provide comprehensive management and operation of construction projects.



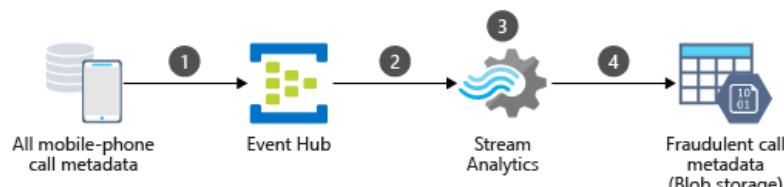
## Data warehousing and analytics for sales and marketing

Consolidate data from multiple sources and optimize data analytics.



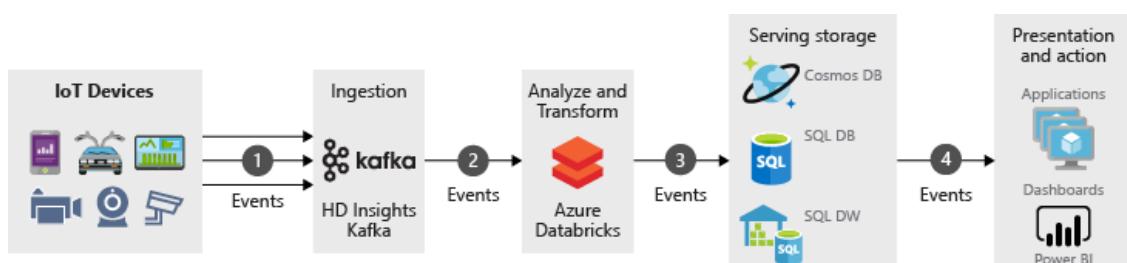
## Scalable order processing on Azure

Build a highly scalable order processing pipeline using Azure Cosmos DB.



## Real-time fraud detection on Azure

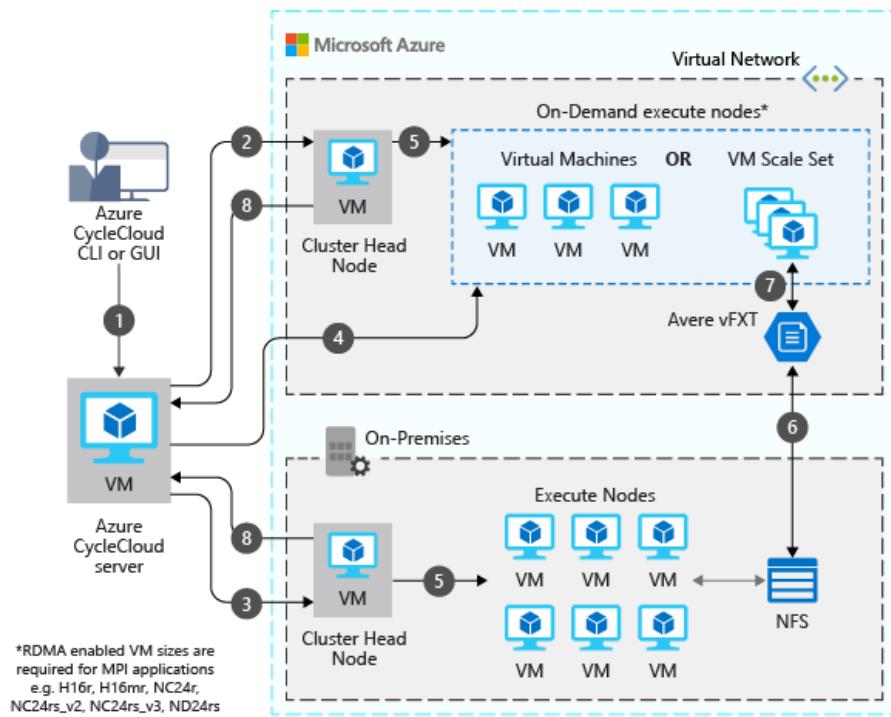
Detect fraudulent activity in real-time using Azure Event Hubs and Stream Analytics.



## Ingestion and processing of real-time automotive IoT data

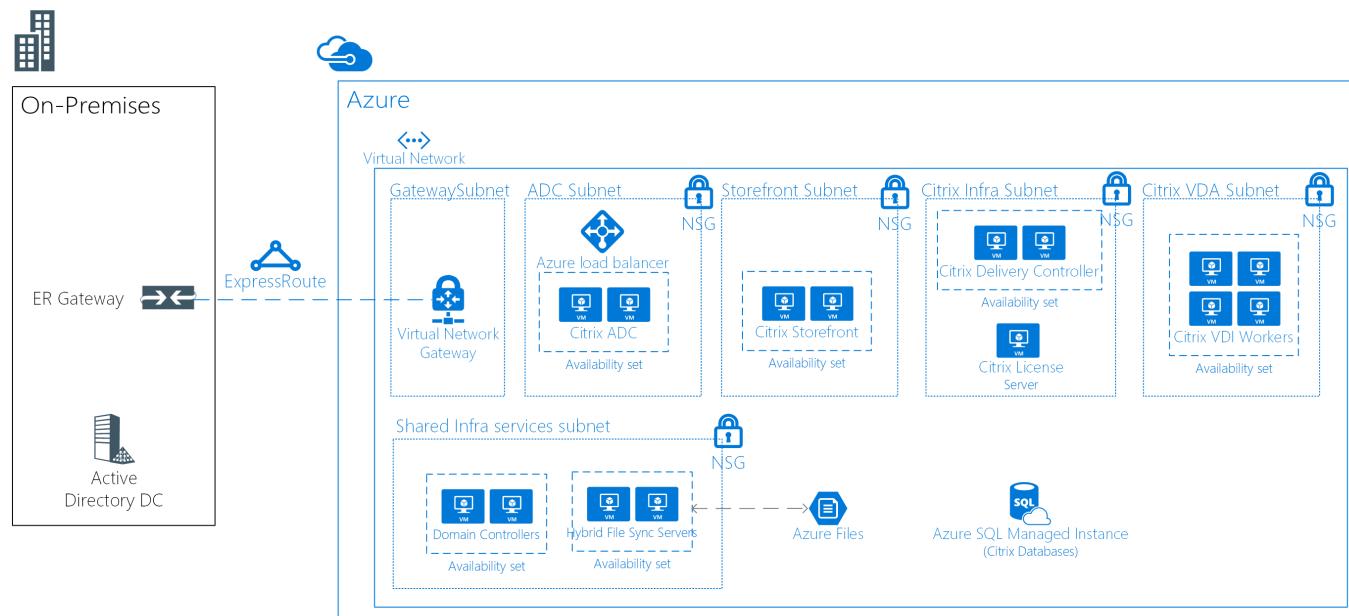
Ingest and process real-time vehicle data using IoT.

## Infrastructure Scenarios



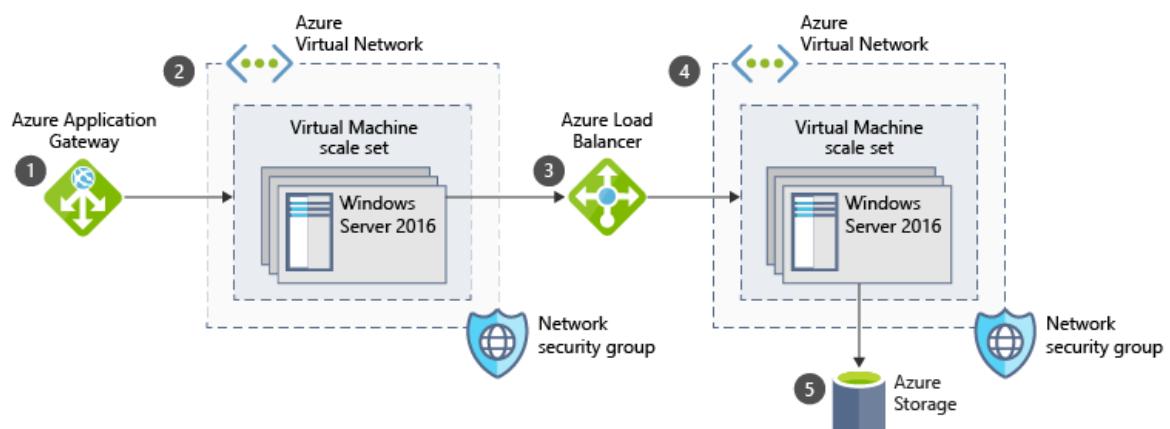
## Running computational fluid dynamics (CFD) simulations on Azure

Execute computational fluid dynamics (CFD) simulations on Azure.



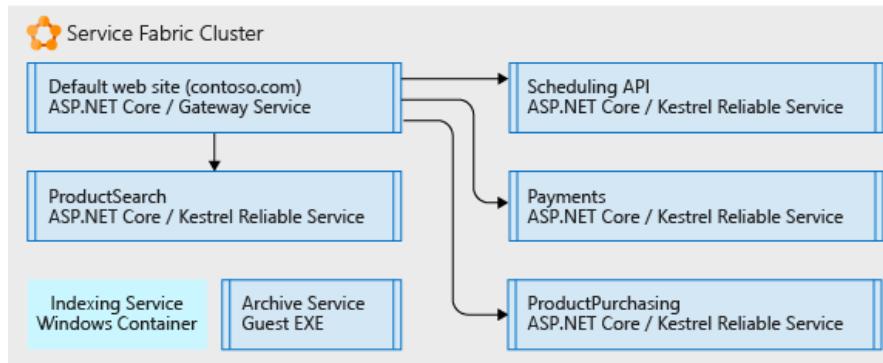
## Linux virtual desktops with Citrix

Build a VDI environment for Linux Desktops using Citrix on Azure.



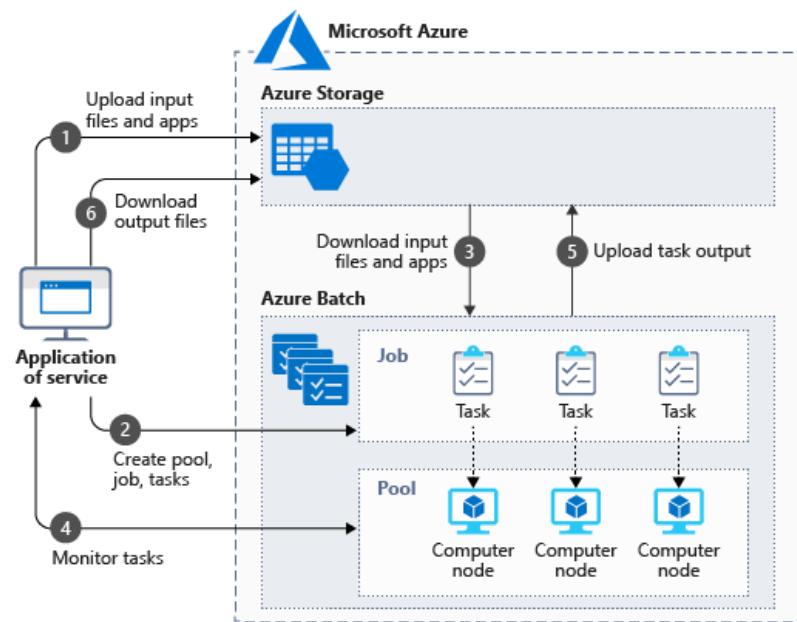
## Secure Windows web application for regulated industries

Build a secure, multi-tier web application with Windows Server on Azure using scale sets, Application Gateway, and load balancers.



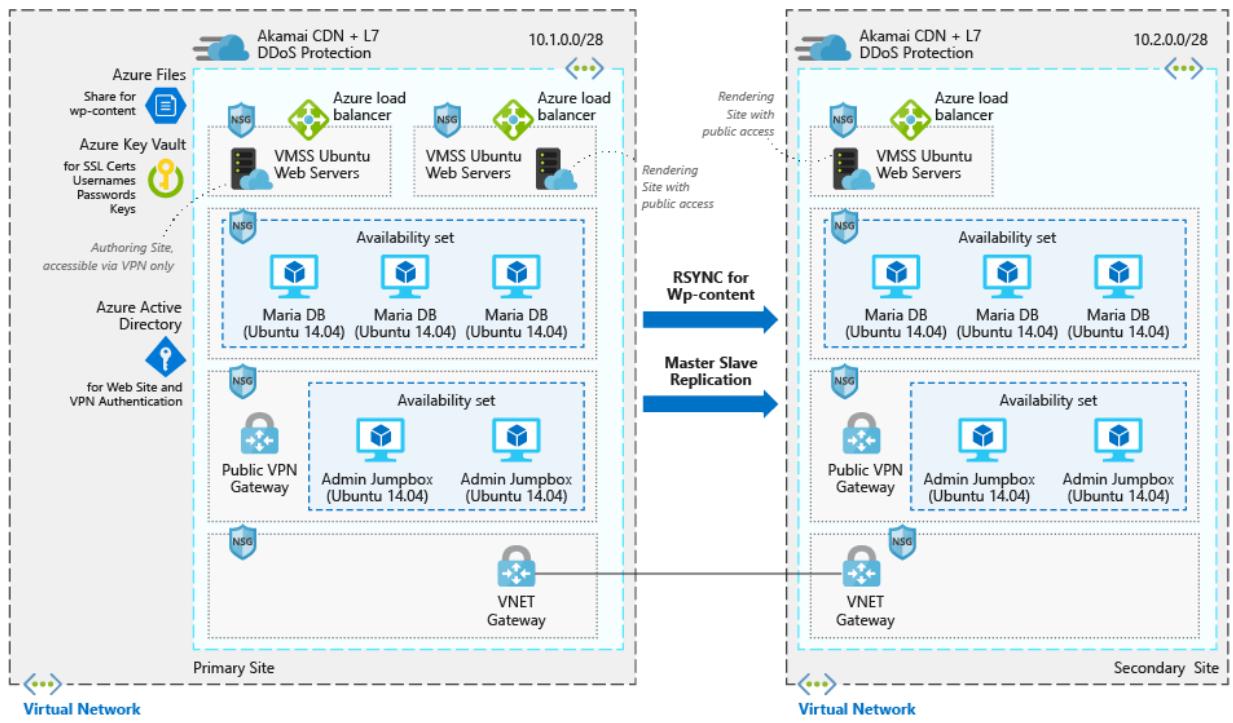
## Using Service Fabric to decompose monolithic applications

Decompose a large monolithic application into microservices.



## 3D video rendering on Azure

Run native HPC workloads in Azure using the Azure Batch service.



## Highly scalable and secure WordPress websites on Azure

Build a highly scalable and secure WordPress website for media events.

# Conversational chatbot for hotel reservations on Azure

10/5/2018 • 6 minutes to read • [Edit Online](#)

This example scenario is applicable to businesses that need to integrate a conversational chatbot into applications. In this scenario, a C# chatbot is used for a hotel chain that allows customers to check availability and book accommodation through a web or mobile application.

Potential uses include providing a way for customers to view hotel availability and book rooms, review a restaurant take-out menu and place a food order, or search for and order prints of photographs. Traditionally, businesses would need to hire and train customer service agents to respond to these customer requests, and customers would have to wait until a representative is available to provide assistance.

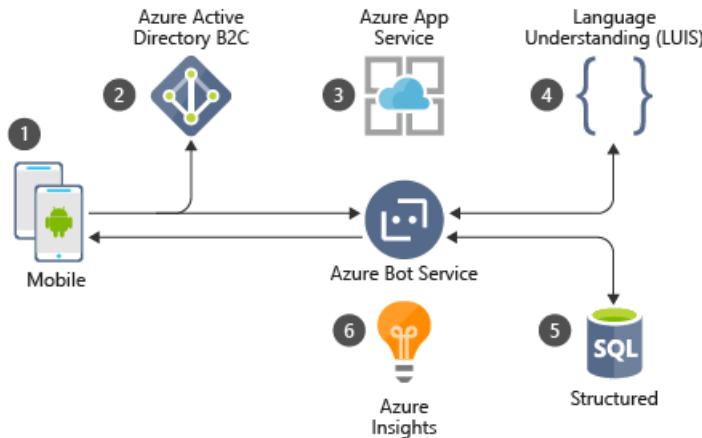
By using Azure services such as the Bot Service and Language Understanding or Speech API services, companies can assist customers and process orders or reservations with automated, scalable bots.

## Relevant use cases

Consider this scenario for the following use cases:

- View restaurant take-out menu and order food
- Check hotel availability and reserve a room
- Search available photos and order prints

## Architecture



This scenario covers a conversational bot that functions as a concierge for a hotel. The data flows through the scenario as follows:

1. The customer accesses the chatbot with a mobile or web app.
2. Using Azure Active Directory B2C (Business 2 Customer), the user is authenticated.
3. Interacting with the Bot Service, the user requests information about hotel availability.
4. Cognitive Services processes the natural language request to understand the customer communication.
5. After the user is happy with the results, the bot adds or updates the customer's reservation in a SQL Database.
6. Application Insights gathers runtime telemetry throughout the process to help the DevOps team with bot performance and usage.

## Components

- [Azure Active Directory](#) is Microsoft's multi-tenant cloud-based directory and identity management service. Azure AD supports a B2C connector allowing you to identify individuals using external IDs such as Google, Facebook, or a Microsoft Account.
- [App Service](#) enables you to build and host web applications in the programming language of your choice without managing infrastructure.
- [Bot Service](#) provides tools to build, test, deploy, and manage intelligent bots.
- [Cognitive Services](#) lets you use intelligent algorithms to see, hear, speak, understand, and interpret your user needs through natural methods of communication.
- [SQL Database](#) is a fully managed relational cloud database service that provides SQL Server engine compatibility.
- [Application Insights](#) is an extensible Application Performance Management (APM) service that lets you monitor the performance of applications, such as your chatbot.

## Alternatives

- [Microsoft Speech API](#) can be used to change how customers interface with your bot.
- [QnA Maker](#) can be used as to quickly add knowledge to your bot from semi-structured content like an FAQ.
- [Translator Text](#) is a service that you might consider to easily add multi-lingual support to your bot.

# Considerations

## Availability

This scenario uses Azure SQL Database for storing customer reservations. SQL Database includes zone redundant databases, failover groups, and geo-replication. For more information, see [Azure SQL Database availability capabilities](#).

For other availability topics, see the [availability checklist](#) in the Azure Architecture Center.

## Scalability

This scenario uses Azure App Service. With App Service, you can automatically scale the number of instances that run your bot. This functionality lets you keep up with customer demand for your web application and chatbot. For more information on autoscale, see [Autoscaling best practices](#) in the Azure Architecture Center.

For other scalability topics, see the [scalability checklist](#) in the Azure Architecture Center.

## Security

This scenario uses Azure Active Directory B2C (Business 2 Consumer) to authenticate users. With AAD B2C, your chatbot doesn't store any sensitive customer account information or credentials. For more information, see [Azure Active Directory B2C overview](#).

Information stored in Azure SQL Database is encrypted at rest with transparent data encryption (TDE). SQL Database also offers Always Encrypted which encrypts data during querying and processing. For more information on SQL Database security, see [Azure SQL Database security and compliance](#).

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

## Resiliency

This scenario uses Azure SQL Database for storing customer reservations. SQL Database includes zone redundant databases, failover groups, geo-replication, and automatic backups. These features allow your application to continue running if there is a maintenance event or outage. For more information, see [Azure SQL Database availability capabilities](#).

To monitor the health of your application, this scenario uses Application Insights. With Application Insights, you can generate alerts and respond to performance issues that would impact the customer experience and availability of the chatbot. For more information, see [What is Application Insights?](#)

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Deploy the scenario

This scenario is divided into three components for you to explore areas that you are most focused on:

- [Infrastructure components](#). Use an Azure Resource Manager template to deploy the core infrastructure components of an App Service, Web App, Application Insights, Storage account, and SQL Server and database.
- [Web App Chatbot](#). Use the Azure CLI to deploy a bot with the Bot Service and Language Understanding and Intelligent Services (LUIS) app.
- [Sample C# chatbot application](#). Use Visual Studio to review the sample hotel reservation C# application code and deploy to a bot in Azure.

**Prerequisites.** You must have an existing Azure account. If you don't have an Azure subscription, create a [free account](#) before you begin.

### Deploy infrastructure components

To deploy the infrastructure components with an Azure Resource Manager template, perform the following steps.

1. Click the **Deploy to Azure** button:



2. Wait for the template deployment to open in the Azure portal, then complete the following steps:

- Choose to **Create new** resource group, then provide a name such as *myCommerceChatBotInfrastructure* in the text box.
- Select a region from the **Location** drop-down box.
- Provide a username and secure password for the SQL Server administrator account.
- Review the terms and conditions, then check **I agree to the terms and conditions stated above**.
- Select the **Purchase** button.

It takes a few minutes for the deployment to complete.

### Deploy Web App chatbot

To create the chatbot, use the Azure CLI. The following example installs the CLI extension for Bot Service, creates a resource group, then deploys a bot that uses Application Insights. When prompted, authenticate your Microsoft account and allow the bot to register itself with the Bot Service and Language Understanding and Intelligent Services (LUIS) app.

```
# Install the Azure CLI extension for the Bot Service
az extension add --name botservice --yes

# Create a resource group
az group create --name myCommerceChatbot --location eastus

# Create a Web App Chatbot that uses Application Insights
az bot create \
    --resource-group myCommerceChatbot \
    --name commerceChatbot \
    --location eastus \
    --kind webapp \
    --sku S1 \
    --insights eastus
```

### Deploy chatbot C# application code

A sample C# application is available on GitHub:

- [Commerce Bot C# sample](#)

The sample application includes the Azure Active Directory authentication components and integration with the Language Understanding and Intelligent Services (LUIS) component of Cognitive Services. The application requires Visual Studio to build and deploy the scenario. Additional information on configuring AAD B2C and the LUIS app can be found in the GitHub repo documentation.

## Pricing

To explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic.

We have provided three sample cost profiles based on the number of messages you expect your chatbot to process:

- **Small**: this pricing example correlates to processing < 10,000 messages per month.
- **Medium**: this pricing example correlates to processing < 500,000 messages per month.
- **Large**: this pricing example correlates to processing < 10 million messages per month.

## Related resources

For a set of guided tutorials on leveraging the Azure Bot Service, see the [tutorial node](#) of the documentation.

# Image classification for insurance claims on Azure

10/5/2018 • 4 minutes to read • [Edit Online](#)

This scenario is relevant for businesses that need to process images.

Potential applications include classifying images for a fashion website, analyzing text and images for insurance claims, or understanding telemetry data from game screenshots. Traditionally, companies would need to develop expertise in machine learning models, train the models, and finally run the images through their custom process to get the data out of the images.

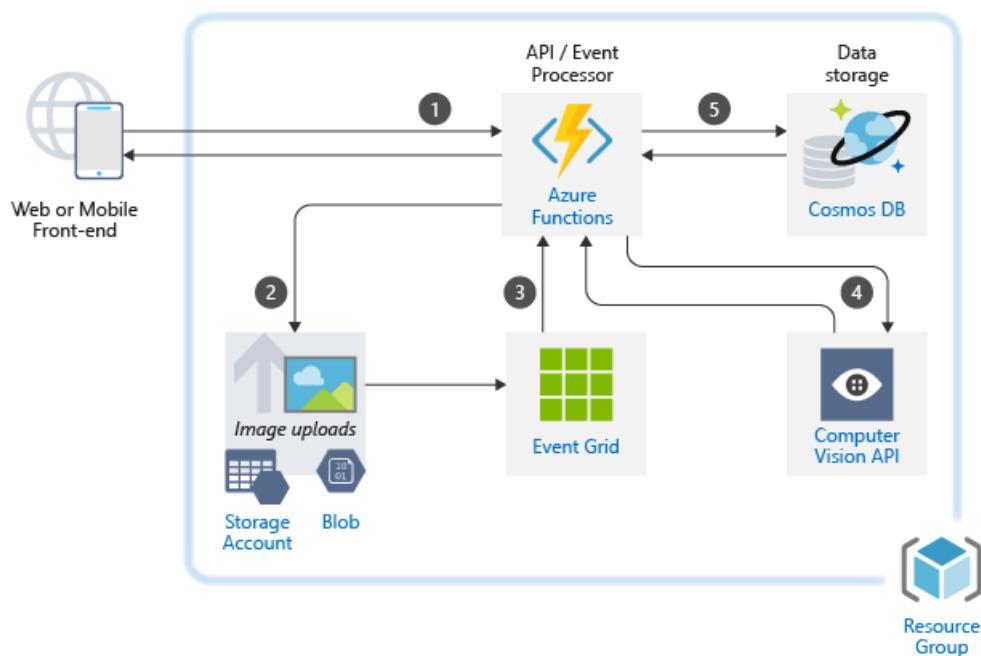
By using Azure services such as the Computer Vision API and Azure Functions, companies can eliminate the need to manage individual servers, while reducing costs and leveraging the expertise that Microsoft has already developed around processing images with Cognitive Services. This example scenario specifically addresses an image-processing use case. If you have different AI needs, consider the full suite of [Cognitive Services](#).

## Relevant use cases

Consider this scenario for the following use cases:

- Classify images on a fashion website.
- Classify telemetry data from screenshots of games.

## Architecture



This scenario covers the back-end components of a web or mobile application. Data flows through the scenario as follows:

1. The API layer is built using Azure Functions. These APIs enable the application to upload images and retrieve data from Cosmos DB.
2. When an image is uploaded via an API call, it's stored in Blob storage.
3. Adding new files to Blob storage triggers an Event Grid notification to be sent to an Azure Function.
4. Azure Functions sends a link to the newly uploaded file to the Computer Vision API to analyze.
5. Once the data has been returned from the Computer Vision API, Azure Functions makes an entry in Cosmos

DB to persist the results of the analysis along with the image metadata.

## Components

- [Computer Vision API](#) is part of the Cognitive Services suite and is used to retrieve information about each image.
- [Azure Functions](#) provides the back-end API for the web application, as well as the event processing for uploaded images.
- [Event Grid](#) triggers an event when a new image is uploaded to blob storage. The image is then processed with Azure functions.
- [Blob storage](#) stores all of the image files that are uploaded into the web application, as well any static files that the web application consumes.
- [Cosmos DB](#) stores metadata about each image that is uploaded, including the results of the processing from Computer Vision API.

## Alternatives

- [Custom Vision Service](#). The Computer Vision API returns a set of [taxonomy-based categories](#). If you need to process information that isn't returned by the Computer Vision API, consider the Custom Vision Service, which lets you build custom image classifiers.
- [Azure Search](#). If your use case involves querying the metadata to find images that meet specific criteria, consider using Azure Search. Currently in preview, [Cognitive search](#) seamlessly integrates this workflow.

## Considerations

### Scalability

The majority of the components used in this example scenario are managed services that will automatically scale. A couple notable exceptions: Azure Functions has a limit of a maximum of 200 instances. If you need to scale beyond this limit, consider multiple regions or app plans.

Cosmos DB doesn't auto-scale in terms of provisioned request units (RUs). For guidance on estimating your requirements see [request units](#) in our documentation. To fully take advantage of the scaling in Cosmos DB, understand how [partition keys](#) work in CosmosDB.

NoSQL databases frequently trade consistency (in the sense of the CAP theorem) for availability, scalability, and partitioning. In this example scenario, a key-value data model is used and transaction consistency is rarely needed as most operations are by definition atomic. Additional guidance to [Choose the right data store](#) is available in the Azure Architecture Center. If your implementation requires high consistency, you can [choose your consistency level](#) in CosmosDB.

For general guidance on designing scalable solutions, see the [scalability checklist](#) in the Azure Architecture Center.

### Security

[Managed identities for Azure resources](#) are used to provide access to other resources internal to your account and then assigned to your Azure Functions. Only allow access to the requisite resources in those identities to ensure that nothing extra is exposed to your functions (and potentially to your customers).

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

### Resiliency

All of the components in this scenario are managed, so at a regional level they are all resilient automatically.

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Pricing

To explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic.

We have provided three sample cost profiles based on amount of traffic (we assume all images are 100 kb in size):

- [Small](#): this pricing example correlates to processing < 5000 images a month.
- [Medium](#): this pricing example correlates to processing 500,000 images a month.
- [Large](#): this pricing example correlates to processing 50 million images a month.

## Related resources

For a guided learning path of this scenario, see [Build a serverless web app in Azure](#).

Before deploying this example scenario in a production environment, review the Azure Functions [best practices](#).

# A computer-aided engineering service on Azure

10/5/2018 • 5 minutes to read • [Edit Online](#)

This example scenario demonstrates delivery of a software-as-a-service (SaaS) platform built on the high-performance computing (HPC) capabilities of Azure. This scenario is based on an engineering software solution. However, the architecture is relevant to other industries requiring HPC resources such as image rendering, complex modeling, and financial risk calculation.

This example demonstrates an engineering software provider that delivers computer-aided engineering (CAE) applications to engineering firms and manufacturing enterprises. CAE solutions enable innovation, reduce development times, and lower costs throughout the lifetime of a product's design. These solutions require substantial compute resources and often process high data volumes. The high costs of an on-premises HPC appliance or high-end workstations often put these technologies out of reach for small engineering firms, entrepreneurs, and students.

The company wants to expand the market for its applications by building a SaaS platform backed by cloud-based HPC technologies. Their customers should be able to pay for compute resources as needed and access massive computing power that would be unaffordable otherwise.

The company's goals include:

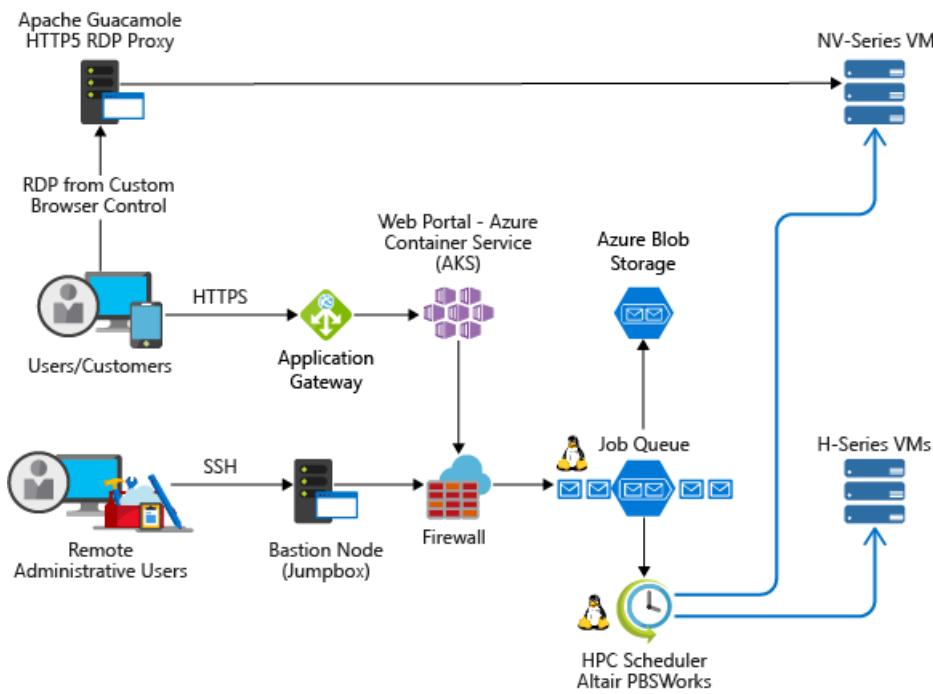
- Taking advantage of HPC capabilities in Azure to accelerate the product design and testing process.
- Using the latest hardware innovations to run complex simulations, while minimizing the costs for simpler simulations.
- Enabling true-to-life visualization and rendering in a web browser, without requiring a high-end engineering workstation.

## Relevant use cases

Other scenarios using this architecture might include:

- Genomics research
- Weather simulation
- Computational chemistry applications

## Architecture



- Users can access NV-series virtual machines (VMs) via a browser with an HTML5-based RDP connection using the [Apache Guacamole service](#). These VM instances provide powerful GPUs for rendering and collaborative tasks. Users can edit their designs and view their results without needing access to high-end mobile computing devices or laptops. The scheduler spins up additional VMs based on user-defined heuristics.
- From a desktop CAD session, users can submit workloads for execution on available HPC cluster nodes. These workloads perform tasks such as stress analysis or computational fluid dynamics calculations, eliminating the need for dedicated on-premises compute clusters. These cluster nodes can be configured to auto-scale based on load or queue depth based on active user demand for compute resources.
- Azure Kubernetes Service (AKS) is used to host the web resources available to end users.

## Components

- [H-series virtual machines](#) are used to run compute-intensive simulations such as molecular modeling and computational fluid dynamics. The solution also takes advantage of technologies like remote direct memory access (RDMA) connectivity and InfiniBand networking.
- [NV-series virtual machines](#) give engineers high-end workstation functionality from a standard web browser. These virtual machines have NVIDIA Tesla M60 GPUs that support advanced rendering and can run single precision workloads.
- [General purpose virtual machines](#) running CentOS handle more traditional workloads such as web applications.
- [Application Gateway](#) load balances the requests coming into the web servers.
- [Azure Kubernetes Service \(AKS\)](#) is used to run scalable workloads at a lower cost for simulations that don't require the high end capabilities of HPC or GPU virtual machines.
- [Altair PBS Works Suite](#) orchestrates the HPC workflow, ensuring that enough virtual machine instances are available to handle the current load. It also deallocates virtual machines when demand is lower to reduce costs.
- [Blob storage](#) stores files that support the scheduled jobs.

## Alternatives

- [Azure CycleCloud](#) simplifies creating, managing, operating, and optimizing HPC clusters. It offers advanced policy and governance features. CycleCloud supports any job scheduler or software stack.
- [HPC Pack](#) can create and manage an Azure HPC cluster for Windows Server-based workloads. HPC Pack isn't an option for Linux-based workloads.
- [Azure Automation State Configuration](#) provides an infrastructure-as-code approach to defining the virtual machines and software to be deployed. Virtual machines can be deployed as part of a virtual machine scale set,

with auto-scaling rules for compute nodes based on the number of jobs submitted to the job queue. When a new virtual machine is needed, it is provisioned using the latest patched image from the Azure image gallery, and then the required software is installed and configured via a PowerShell DSC configuration script.

- [Azure Functions](#)

## Considerations

- While using an infrastructure-as-code approach is a great way to manage virtual machine build definitions, it can take a long time to provision a new virtual machine using a script. This solution found a good middle ground by using the DSC script to periodically create a golden image, which can then be used to provision a new virtual machine faster than completely building a VM on demand using DSC. Azure DevOps Services or other CI/CD tooling can periodically refresh golden images using DSC scripts.
- Balancing overall solution costs with fast availability of compute resources is a key consideration. Provisioning a pool of N-series virtual machine instances and putting them in a deallocated state lowers the operating costs. When an additional virtual machine is needed, reallocating an existing instance will involve powering up the virtual machine on a different host, but the PCI bus detection time required by the OS to identify and install drivers for the GPU is eliminated because a virtual machine that is deprovisioned and then reprovisioned will retain the same PCI bus for the GPU upon restart.
- The original architecture relied entirely on Azure virtual machines for running simulations. In order to reduce costs for workloads that didn't require all the capabilities of a virtual machine, these workloads were containerized and deployed to Azure Kubernetes Service (AKS).
- The company's workforce had existing skills in open-source technologies. They can take advantage of these skills by building on technologies like Linux and Kubernetes.

## Pricing

To help you explore the cost of running this scenario, many of the required services are pre-configured in a [cost calculator example](#). The costs of your solution are dependent on the number and scale of services needed to meet your requirements.

The following considerations will drive a substantial portion of the costs for this solution:

- Azure virtual machine costs increase linearly as additional instances are provisioned. Virtual machines that are deallocated will only incur storage costs, and not compute costs. These deallocated machines can then be reallocated when demand is high.
- Azure Kubernetes Services costs are based on the VM type chosen to support the workload. The costs will increase linearly based on the number of VMs in the cluster.

## Next Steps

- Read the [Altair customer story](#). This example scenario is based on a version of their architecture.
- Review other [Big Compute solutions](#) available in Azure.

# Decentralized trust between banks on Azure

10/9/2018 • 6 minutes to read • [Edit Online](#)

This example scenario is useful for banks or any other institutions that want to establish a trusted environment for information sharing without resorting to a centralized database. For the purpose of this example, we will describe the scenario in the context of maintaining credit score information between banks, but the architecture can be applied to any scenario where a consortium of organizations want to share validated information with one another without resorting to the use of a central system ran by one single party.

Traditionally, banks within a financial system rely on centralized sources such as credit bureaus for information on an individual's credit score and history. A centralized approach presents a concentration of operational risk and sometimes an unnecessary third party.

With DLTs (distributed ledger technology), a consortium of banks can establish a decentralized system that can be more efficient, less susceptible to attack, and serve as a new platform where innovative structures can be implemented to solve traditional challenges with privacy, speed, and cost.

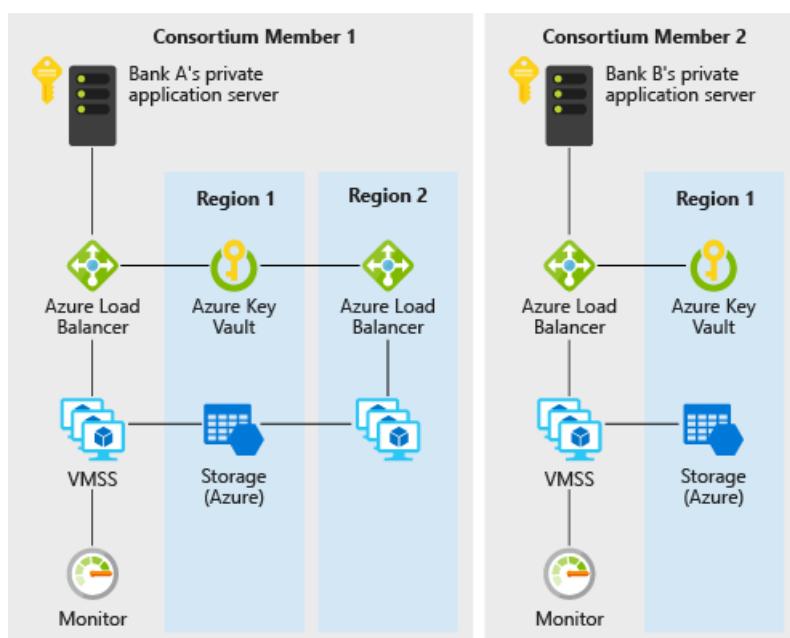
This example will show you how Azure services such as virtual machine scale set, Virtual Network, Key Vault, Storage, Load Balancer, and Monitor can be quickly provisioned for the deployment of an efficient private Ethereum PoA blockchain where member banks can establish their own nodes.

## Relevant use cases

These other uses cases have similar design patterns:

- Movement of allocated budgets between different business units of a multinational corporation
- Cross-border payments
- Trade finance scenarios
- Loyalty systems involving different companies
- Supply chain ecosystems and many more

## Architecture



This scenario covers the back-end components that are necessary to create a scalable, secure, and monitored private, enterprise blockchain network within a consortium of two or more members. Details of how these components are provisioned (that is, within different subscriptions and resource groups) as well as the connectivity requirements (that is, VPN or ExpressRoute) are left for your consideration based on your organization's policy requirements. Here's how data flows:

1. Bank A creates/updates an individual's credit record by sending a transaction to the blockchain network via JSON-RPC.
2. Data flows from Bank A's private application server to the Azure load balancer and subsequently to a validating node VM on the virtual machine scale set.
3. The Ethereum PoA network creates a block at a preset time (2 seconds for this scenario).
4. The transaction is bundled into the created block and validated across the blockchain network.
5. Bank B can read the credit record created by bank A by communicating with its own node similarly via JSON-RPC.

## Components

- Virtual Machines within Virtual Machine Scale Sets provides the on-demand compute facility to host the validator processes for the blockchain
- Key Vault is used as the secure storage facility for the private keys of each validator
- Load Balancer spreads the RPC, peering, and Governance DApp requests
- Storage hosting persistent network information and coordinating leasing
- Operations Management Suite (a bundling of a few Azure services) provides insight into available nodes, transactions per minute and consortium members

## Alternatives

The Ethereum PoA approach is chosen for this example because it is a good entry point for a consortium of organizations that want to create an environment where information can be exchanged and shared with one another easily in a trusted, decentralized, and easy to understand way. The available Azure solution templates also provide a fast and convenient way not just for a consortium leader to start an Ethereum PoA blockchain, but also for member organizations in the consortium to spin up their own Azure resources within their own resource group and subscription to join an existing network.

For other extended or different scenarios, concerns such as transaction privacy may arise. For example, in a securities transfer scenario, members in a consortium may not want their transactions to be visible even to other members. Other alternatives to Ethereum PoA exist that addresses these concerns in their own way:

- Corda
- Quorum
- Hyperledger

## Considerations

### Availability

[Azure Monitor](#) is used to continuously monitor the blockchain network for issues to ensure availability. A link to a custom monitoring dashboard based on Azure Monitor will be sent to you upon successful deployment of the blockchain solution template used in this scenario. The dashboard shows nodes that are reporting heartbeats in the past 30 minutes as well as other useful statistics.

For other availability topics, see the [availability checklist](#) in the Azure Architecture Center.

### Scalability

A popular concern for blockchain is the number of transactions that a blockchain can include within a preset amount of time. This scenario uses Proof-of-Authority where such scalability can be better managed than Proof-

of-Work. In Proof-of-Authority based networks, consensus participants are known and managed, making it more suitable for private blockchain for a consortium of organization that knows one another. Parameters such as average block time, transactions per minute and compute resource consumption can be easily monitored via the custom dashboard. Resources can then be adjusted accordingly based on scale requirements.

For general guidance on designing scalable scenario, see the [scalability checklist](#) in the Azure Architecture Center.

## Security

[Azure Key Vault](#) is used to easily store and manage the private keys of validators. The default deployment in this example creates a blockchain network that is accessible via the internet. For production scenario where a private network is desired, members can be connected to each other via VNet-to-VNet VPN gateway connections. The steps for configuring a VPN are included in the related resources section below.

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

## Resiliency

The Ethereum PoA blockchain can itself provide some degree of resilience as the validator nodes can be deployed in different regions. Azure has options for deployments in over 54 regions worldwide. A blockchain such as the one in this scenario provides unique and refreshing possibilities of cooperation to increase resilience. The resilience of the network is not just provided for by a single centralized party but all members of the consortium. A Proof-of-Authority based blockchain allows network resilience to be even more planned and deliberate.

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Pricing

To explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected performance and availability requirements.

We have provided three sample cost profiles based on the number of scale set VM instances that run your applications (the instances can reside in different regions).

- [Small](#): this pricing example correlates to 2 VMs per month with monitoring turned off
- [Medium](#): this pricing example correlates to 7 VMs per month with monitoring turned on
- [Large](#): this pricing example correlates to 15 VMs per month with monitoring turned on

The above pricing is for one consortium member to start or join a blockchain network. Typically in a consortium where there are multiple companies or organizations involved, each member will get their own Azure subscription.

## Next Steps

To see an example of this scenario, deploy the [Ethereum PoA blockchain demo application](#) on Azure, then go through the [README of the scenario source code](#).

## Related resources

For more information on using the Ethereum Proof-of-Authority solution template for Azure, review this [usage guide](#).

# Web application monitoring on Azure

10/13/2018 • 7 minutes to read • [Edit Online](#)

Platform as a service (PaaS) offerings on Azure manage compute resources for you and in some ways change how you monitor deployments. Azure includes multiple monitoring services, each of which performs a specific role. Together, these services deliver a comprehensive solution for collecting, analyzing, and acting on telemetry from your applications and the Azure resources they use.

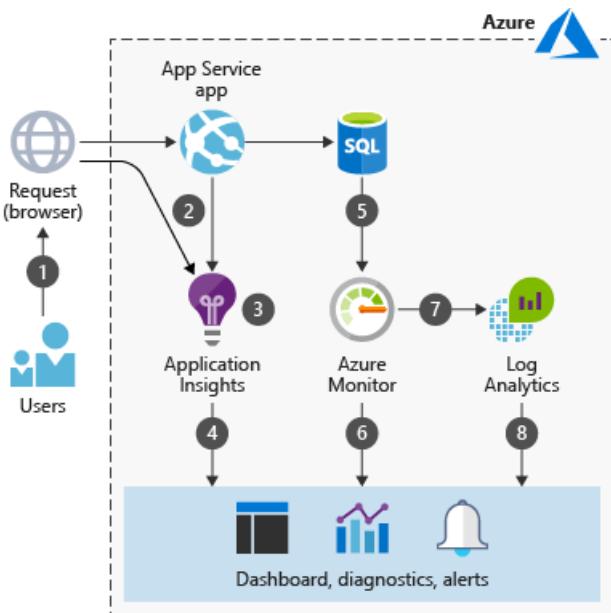
This scenario addresses the monitoring services you can use and describes a dataflow model for use with multiple data sources. When it comes to monitoring, many tools and services work with Azure deployments. In this scenario, we choose readily available services precisely because they are easy to consume. Other monitoring options are discussed later in this article.

## Relevant use cases

Consider this scenario for the following use cases:

- Instrumenting a web application for monitoring telemetry.
- Collecting front-end and back-end telemetry for an application deployed on Azure.
- Monitoring metrics and quotas associated with services on Azure.

## Architecture



This scenario uses a managed Azure environment to host an application and data tier. The data flows through the scenario as follows:

1. A user interacts with the application.
2. The browser and app service emit telemetry.
3. Application Insights collects and analyzes application health, performance, and usage data.
4. Developers and administrators can review health, performance, and usage information.
5. Azure SQL Database emits telemetry.
6. Azure Monitor collects and analyzes infrastructure metrics and quotas.
7. Log Analytics collects and analyzes logs and metrics.

8. Developers and administrators can review health, performance, and usage information.

## Components

- [Azure App Service](#) is a PaaS service for building and hosting apps in managed virtual machines. The underlying compute infrastructures on which your apps run is managed for you. App Service provides monitoring of resource usage quotas and app metrics, logging of diagnostic information, and alerts based on metrics. Even better, you can use Application Insights to create [availability tests](#) for testing your application from different regions.
- [Application Insights](#) is an extensible Application Performance Management (APM) service for developers and supports multiple platforms. It monitors the application, detects application anomalies such as poor performance and failures, and sends telemetry to the Azure portal. Application Insights can also be used for logging, distributed tracing, and custom application metrics.
- [Azure Monitor](#) provides base-level infrastructure [metrics and logs](#) for most services in Azure. You can interact with the metrics in several ways, including charting them in Azure portal, accessing them through the REST API, or querying them using PowerShell or CLI. Azure Monitor also offers its data directly into [Log Analytics and other services](#), where you can query and combine it with data from other sources on premises or in the cloud.
- [Log Analytics](#) helps correlate the usage and performance data collected by Application Insights with configuration and performance data across the Azure resources that support the app. This scenario uses the [Azure Log Analytics agent](#) to push SQL Server audit logs into Log Analytics. You can write queries and view data in the Log Analytics blade of the Azure portal.

## Considerations

A best practice is to add Application Insights to your code at development using the [Application Insights SDKs](#), and customize per application. These open source SDKs are available for most application frameworks. To enrich and control the data you collect, incorporate the use of the SDKs both for testing and production deployments into your development process. The main requirement is for the app to have a direct or indirect line of sight to the Applications Insights ingestion endpoint hosted with an Internet-facing address. You can then add telemetry or enrich an existing telemetry collection.

Runtime monitoring is another easy way to get started. The telemetry that is collected must be controlled through configuration files. For example, you can include runtime methods that enable tools such as [Application Insights Status Monitor](#) to deploy the SDKs into the correct folder and add the right configurations to begin monitoring.

Like Application Insights, Log Analytics provides tools for [analyzing data across sources](#), creating complex queries, and [sending proactive alerts](#) on specified conditions. You can also view telemetry in [the Azure portal](#). Log Analytics adds value to existing monitoring services such as [Azure Monitor][Azure Monitor] and can also monitor on-premises environments.

Both Application Insights and Log Analytics use [Azure Log Analytics Query Language](#). You can also use [cross-resource queries](#) to analyze the telemetry gathered by Application Insights and Log Analytics in a single query.

Azure Monitor, Application Insights, and Log Analytics all send [alerts](#). For example, Azure Monitor alerts on platform-level metrics such as CPU utilization, while Application Insights alerts on application-level metrics such as server response time. Azure Monitor alerts on new events in the Azure Activity Log, while Log Analytics can issue alerts about metrics or event data for the services configured to use it. [Unified alerts in Azure Monitor](#) is a new, unified alerting experience in Azure that uses a different taxonomy.

## Alternatives

This article describes conveniently available monitoring options with popular features, but you have many choices, including the option to create your own logging mechanisms. A best practice is to add monitoring services as you build out tiers in a solution. Here are some possible extensions and alternatives:

- Consolidate Azure Monitor and Application Insights metrics in Grafana using the [Azure Monitor Data Source For Grafana](#).
- [Data Dog](#) features a connector for Azure Monitor
- Automate monitoring functions using [Azure Automation](#).
- Add communication with [ITSM solutions](#).
- Extend Log Analytics with a [management solution](#).

## Scalability and availability

This scenario focuses on PaaS solutions for monitoring in large part because they conveniently handle availability and scalability for you and are backed by service-level agreements (SLAs). For example, App Services provides a guaranteed [SLA](#) for its availability.

Application Insights has [limits](#) on how many requests can be processed per second. If you exceed the request limit, you may experience message throttling. To prevent this, implement [filtering](#) or [sampling](#) to reduce the data rate.

High availability considerations for the app you run, however, are the developer's responsibility. For information about scale, for example, see the [Scalability considerations](#) section in the basic web application reference architecture. After an app is deployed, you can set up tests to [monitor its availability](#) using Application Insights.

## Security

Sensitive information and compliance requirements affect data collection, retention, and storage. Learn more about how [Application Insights](#) and [Log Analytics](#) handle telemetry.

The following security considerations may also apply:

- Develop a plan to handle personal information if developers are allowed to collect their own data or enrich existing telemetry.
- Consider data retention. For example, Application Insights retains telemetry data for 90 days. Archive data you want access to for longer periods using Microsoft Power BI, Continuous Export, or the REST API. Storage rates apply.
- Limit access to Azure resources to control access to data and who can view telemetry from a specific application. To help lock down access to monitoring telemetry, see [Resources, roles, and access control in Application Insights](#).
- Consider whether to control read/write access in application code to prevent users from adding version or tag markers that limit data ingestion from the application. With Application Insights, there is no control over individual data items once they are sent to a resource, so if a user has access to any data, they have access to all data in an individual resource.
- Add [governance](#) mechanisms to enforce policy or cost controls over Azure resources if needed. For example, use Log Analytics for security-related monitoring such as policies and role based access control, or use [Azure Policy](#) to create, assign and, manage policy definitions.
- To monitor potential security issues and get a central view of the security state of your Azure resources, consider using [Azure Security Center](#).

## Pricing

Monitoring charges can add up quickly, so consider pricing up front, understand what you are monitoring, and check the associated fees for each service. Azure Monitor provides [basic metrics](#) at no cost, while monitoring costs for [Application Insights](#) and [Log Analytics](#) are based on the amount of data ingested and the number of tests you run.

To help you get started, use the [pricing calculator](#) to estimate costs. To see how the pricing would change for your particular use case, change the various options to match your expected deployment.

Telemetry from Application Insights is sent to the Azure portal during debugging and after you have published

your app. For testing purposes and to avoid charges, a limited volume of telemetry is instrumented. To add more indicators, you can raise the telemetry limit. For more granular control, see [Sampling in Application Insights](#).

After deployment, you can watch a [Live Metrics Stream](#) of performance indicators. This data is not stored---you are viewing real-time metrics---but the telemetry can be collected and analyzed later. There is no charge for Live Stream data.

Log Analytics is billed per gigabyte (GB) of data ingested into the service. The first 5 GB of data ingested to the Azure Log Analytics service every month is offered free, and the data is retained at no charge for first 31 days in your Log Analytics workspace.

## Next steps

Check out these resources designed to help you get started with your own monitoring solution:

[Basic web application reference architecture](#)

[Start monitoring your ASP.NET Web Application](#)

[Collect data about Azure Virtual Machines](#)

## Related resources

[Monitoring Azure applications and resources](#)

[Find and diagnose run-time exceptions with Azure Application Insights](#)

# CI/CD pipeline for container-based workloads

10/5/2018 • 7 minutes to read • [Edit Online](#)

This example scenario is applicable to businesses that want to modernize application development by using containers and DevOps workflows. In this scenario, a Node.js web app is built and deployed by Jenkins into an Azure Container Registry and Azure Kubernetes Service. For a globally distributed database tier, Azure Cosmos DB is used. To monitor and troubleshoot application performance, Azure Monitor integrates with a Grafana instance and dashboard.

Example application scenarios include providing an automated development environment, validating new code commits, and pushing new deployments into staging or production environments. Traditionally, businesses had to manually build and compile applications and updates, and maintain a large, monolithic code base. With a modern approach to application development that uses continuous integration (CI) and continuous deployment (CD), you can more quickly build, test, and deploy services. This modern approach lets you release applications and updates to your customers faster, and respond to changing business demands in a more agile manner.

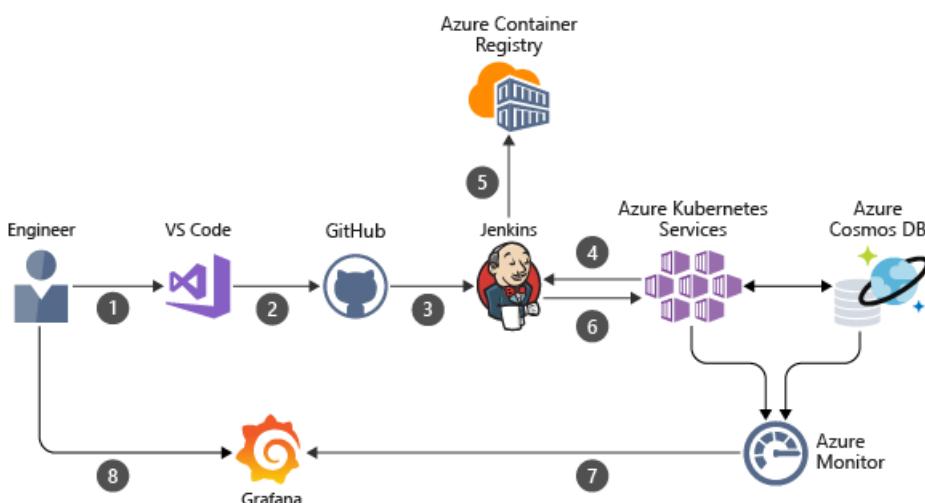
By using Azure services such as Azure Kubernetes Service, Container Registry, and Cosmos DB, companies can use the latest in application development techniques and tools to simplify the process of implementing high availability.

## Relevant use cases

Consider this scenario for the following use cases:

- Modernizing application development practices to a microservice, container-based approach.
- Speeding up application development and deployment lifecycles.
- Automating deployments to test or acceptance environments for validation.

## Architecture



This scenario covers a DevOps pipeline for a Node.js web application and database back end. The data flows through the scenario as follows:

1. A developer makes changes to the Node.js web application source code.
2. The code change is committed to a source control repository, such as GitHub.
3. To start the continuous integration (CI) process, a GitHub webhook triggers a Jenkins project build.
4. The Jenkins build job uses a dynamic build agent in Azure Kubernetes Service to perform a container build

process.

5. A container image is created from the code in source control, and is then pushed to an Azure Container Registry.
6. Through continuous deployment (CD), Jenkins deploys this updated container image to the Kubernetes cluster.
7. The Node.js web application uses Cosmos DB as its back end. Both Cosmos DB and Azure Kubernetes Service report metrics to Azure Monitor.
8. A Grafana instance provides visual dashboards of the application performance based on the data from Azure Monitor.

## Components

- [Jenkins](#) is an open-source automation server that can integrate with Azure services to enable continuous integration (CI) and continuous deployment (CD). In this scenario, Jenkins orchestrates the creation of new container images based on commits to source control, pushes those images to Azure Container Registry, then updates application instances in Azure Kubernetes Service.
- [Azure Linux Virtual Machines](#) is the IaaS platform used to run the Jenkins and Grafana instances.
- [Azure Container Registry](#) stores and manages container images that are used by the Azure Kubernetes Service cluster. Images are securely stored, and can be replicated to other regions by the Azure platform to speed up deployment times.
- [Azure Kubernetes Service](#) is a managed Kubernetes platform that lets you deploy and manage containerized applications without container orchestration expertise. As a hosted Kubernetes service, Azure handles critical tasks like health monitoring and maintenance for you.
- [Azure Cosmos DB](#) is a globally distributed, multi-model database that allows you to choose from various database and consistency models to suit your needs. With Cosmos DB, your data can be globally replicated, and there is no cluster management or replication components to deploy and configure.
- [Azure Monitor](#) helps you track performance, maintain security, and identify trends. Metrics obtained by Monitor can be used by other resources and tools, such as Grafana.
- [Grafana](#) is an open-source solution to query, visualize, alert, and understand metrics. A data source plugin for Azure Monitor allows Grafana to create visual dashboards to monitor the performance of your applications running in Azure Kubernetes Service and using Cosmos DB.

## Alternatives

- [Azure Pipelines](#) help you implement a continuous integration (CI), test, and deployment (CD) pipeline for any app.
- [Kubernetes](#) can be run directly on Azure VMs instead of via a managed service if you would like more control over the cluster.
- [Service Fabric](#) is another alternate container orchestrator that can replace AKS.

## Considerations

### Availability

To monitor your application performance and report on issues, this scenario combines Azure Monitor with Grafana for visual dashboards. These tools let you monitor and troubleshoot performance issues that may require code updates, which can all then be deployed with the CI/CD pipeline.

As part of the Azure Kubernetes Service cluster, a load balancer distributes application traffic to one or more containers (pods) that run your application. This approach to running containerized applications in Kubernetes provides a highly available infrastructure for your customers.

For other availability topics, see the [availability checklist](#) available in the Azure Architecture Center.

### Scalability

Azure Kubernetes Service lets you scale the number of cluster nodes to meet the demands of your applications. As

your application increases, you can scale out the number of Kubernetes nodes that run your service.

Application data is stored in Azure Cosmos DB, a globally distributed, multi-model database that can scale globally. Cosmos DB abstracts the need to scale your infrastructure as with traditional database components, and you can choose to replicate your Cosmos DB globally to meet the demands of your customers.

For other scalability topics, see the [scalability checklist](#) available in the Azure Architecture Center.

## Security

To minimize the attack footprint, this scenario does not expose the Jenkins VM instance over HTTP. For any management tasks that require you to interact with Jenkins, you create a secure remote connection using an SSH tunnel from your local machine. Only SSH public key authentication is allowed for the Jenkins and Grafana VM instances. Password-based logins are disabled. For more information, see [Run a Jenkins server on Azure](#).

For separation of credentials and permissions, this scenario uses a dedicated Azure Active Directory (AD) service principal. The credentials for this service principal are stored as a secure credential object in Jenkins so that they are not directly exposed and visible within scripts or the build pipeline.

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

## Resiliency

This scenario uses Azure Kubernetes Service for your application. Built into Kubernetes are resiliency components that monitor and restart the containers (pods) if there is an issue. Combined with running multiple Kubernetes nodes, your application can tolerate a pod or node being unavailable.

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

# Deploy the scenario

## Prerequisites.

- You must have an existing Azure account. If you don't have an Azure subscription, create a [free account](#) before you begin.
- You need an SSH public key pair. For steps on how to create a public key pair, see [Create and use an SSH key pair for Linux VMs](#).
- You need an Azure Active Directory (AD) service principal for the authentication of service and resources. If needed, you can create a service principal with `az ad sp create-for-rbac`

```
az ad sp create-for-rbac --name myDevOpsScenario
```

Make a note of the *appId* and *password* in the output from this command. You provide these values to the template when you deploy the scenario.

To deploy this scenario with an Azure Resource Manager template, perform the following steps.

1. Click the **Deploy to Azure** button:



2. Wait for the template deployment to open in the Azure portal, then complete the following steps:

- Choose to **Create new** resource group, then provide a name such as *myAKSDevOpsScenario* in the text box.
- Select a region from the **Location** drop-down box.
- Enter your service principal app ID and password from the `az ad sp create-for-rbac` command.
- Provide a username and secure password for the Jenkins instance and Grafana console.

- Provide an SSH key to secure logins to the Linux VMs.
- Review the terms and conditions, then check **I agree to the terms and conditions stated above.**
- Select the **Purchase** button.

It can take 15-20 minutes for the deployment to complete.

## Pricing

To explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic.

We have provided three sample cost profiles based on the number of container images to store and Kubernetes nodes to run your applications.

- [Small](#): this pricing example correlates to 1000 container builds per month.
- [Medium](#): this pricing example correlates to 100,000 container builds per month.
- [Large](#): this pricing example correlates to 1,000,000 container builds per month.

## Related resources

This scenario used Azure Container Registry and Azure Kubernetes Service to store and run a container-based application. Azure Container Instances can also be used to run container-based applications, without having to provision any orchestration components. For more information, see [Azure Container Instances overview](#).

# CI/CD pipeline with Azure DevOps

10/5/2018 • 8 minutes to read • [Edit Online](#)

DevOps is the integration of development, quality assurance, and IT operations. DevOps requires both unified culture and a strong set of processes for delivering software.

This example scenario demonstrates how development teams can use Azure DevOps to deploy a .NET two-tier web application to Azure App Service. The Web Application depends on downstream Azure platform as a service (PaaS) services. This document also points out some considerations that you should make when designing such a scenario using Azure PaaS.

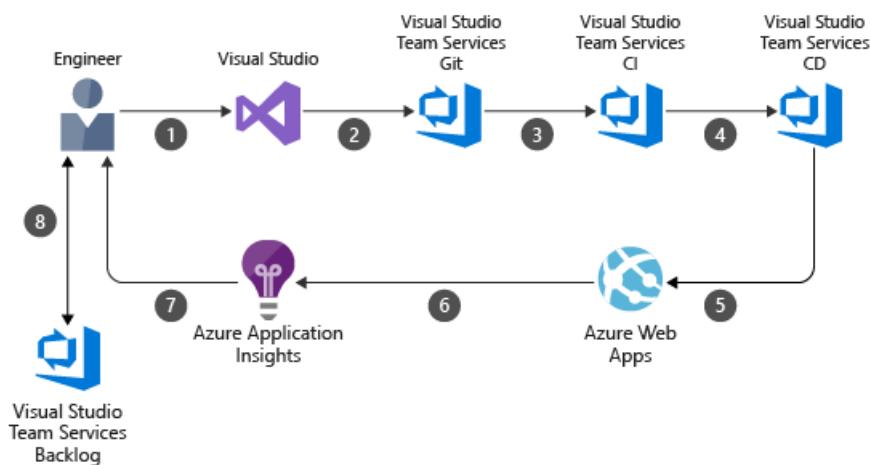
Adopting a modern approach to application development using Continuous Integration and Continuous Deployment (CI/CD), helps you to accelerate the delivery of value to your users through a robust build, test, deployment, and monitoring service. By using a platform such as Azure DevOps along with Azure services such as App Service, organizations can focus on the development of their scenario rather than the management of the supporting infrastructure.

## Relevant use cases

Consider DevOps for the following use cases:

- Accelerating application development and development life cycles
- Building quality and consistency into an automated build and release process

## Architecture



This scenario covers a CI/CD pipeline for a .NET web application using Azure DevOps. The data flows through the scenario as follows:

1. Change application source code.
2. Commit application code and Web Apps web.config file.
3. Continuous integration triggers application build and unit tests.
4. Continuous deployment trigger orchestrates deployment of application artifacts *with environment-specific parameterized configuration values*.
5. Deployment to Azure App Service.
6. Azure Application Insights collects and analyzes health, performance, and usage data.
7. Review health, performance, and usage information.

## Components

- [Azure DevOps](#) is a service for managing your development life cycle end-to-end — from planning and project management, to code management, and continuing to build and release.
- [Azure Web Apps](#) is a PaaS service for hosting web applications, REST APIs, and mobile back ends. While this article focuses on .NET, there are several additional development platform options supported.
- [Application Insights](#) is a first-party, extensible Application Performance Management (APM) service for web developers on multiple platforms.

## Alternative DevOps tooling options

While this article focuses on Azure DevOps, [Team Foundation Server](#) could be used as on-premises substitute. Alternatively, you could also use a set of technologies for an open source development pipeline using [Jenkins](#).

From an infrastructure-as-code perspective, [Azure Resource Manager Templates](#) are included as part of the Azure DevOps project, but you could consider [Terraform](#) or [Chef](#). If you prefer an infrastructure-as-a-service (IaaS)-based deployment and require configuration management, you could consider either [Azure Automation State Configuration](#), [Ansible](#), or [Chef](#).

## Alternatives to Azure Web Apps

You could consider these alternatives to hosting in Azure Web Apps:

- [Azure Virtual Machines](#) — For workloads that require a high degree of control, or depend on OS components and services that are not possible with Web Apps (for example, the Windows GAC, or COM).
- [Service Fabric](#) — a good option if the workload architecture is focused around distributed components that benefit from being deployed and run across a cluster with a high degree of control. Service Fabric can also be used to host containers.
- [Azure Functions](#) - an effective serverless approach if the workload architecture is centered around fine grained distributed components, requiring minimal dependencies, where individual components are only required to run on demand (not continuously) and orchestration of components is not required.

This [decision tree](#) may help when choosing the right path to take for a migration.

## DevOps

[Continuous Integration \(CI\)](#) maintains a stable build, with multiple developers regularly committing small, frequent changes to the shared codebase. As part of your continuous integration pipeline, you should:

- Frequently commit smaller code changes. Avoid batching up larger or more complex changes that may be more difficult to merge successfully.
- Conduct unit testing of your application components with sufficient code coverage, including testing the unhappy paths.
- Ensure the build is run against the shared master (or trunk) branch. This branch should be stable and maintained as "deployment ready". Incomplete or work-in-progress changes should be isolated in a separate branch with frequent "forward integration" merges to avoid conflicts later.

[Continuous Delivery \(CD\)](#) demonstrates not just a stable build but a stable deployment. This makes realizing CD a little more difficult, requiring environment-specific configuration and a mechanism for setting those values correctly. Other CD considerations include the following:

- Sufficient integration testing coverage is required to validate that the various components are configured and working correctly end-to-end.
- CD may also require setting up and resetting environment-specific data and managing database schema versions.
- Continuous delivery should also extend to load testing and user acceptance testing environments.
- Continuous delivery benefits from continuous monitoring, ideally across all environments.
- The consistency and reliability of deployments and integration testing across environments is made easier by

scripting the creation and configuration of the hosting infrastructure. This is considerably easier for cloud-based workloads. For more information, see [Infrastructure as Code](#).

- Begin continuous delivery as early as possible in the project lifecycle. The later you begin, the more difficult it will be to incorporate.
- Integration and unit tests should be given the same priority as application features.
- Use environment-agnostic deployment packages and manage environment-specific configuration via the release process.
- Protect sensitive configuration using the release management tooling, or by calling out to a Hardware-security-module (HSM) or [Azure Key Vault](#) during the release process. Do not store sensitive configuration within source control.

**Continuous Learning.** The most effective monitoring of a CD environment is provided by application performance monitoring (APM) tools such as [Application Insights](#). Sufficient depth of monitoring for an application workload is critical to understand bugs or performance under load. Application Insights can be integrated into VSTS to enable [continuous monitoring of the CD pipeline](#). This could be used to enable automatic progression to the next stage, without human intervention, or rollback if an alert is detected.

## Considerations

### Availability

Consider leveraging the [typical design patterns for availability](#) when building your cloud application.

Review the availability considerations in the appropriate [App Service web application reference architecture](#)

For other availability topics, see the [availability checklist](#) in the Azure Architecture Center.

### Scalability

When building a cloud application be aware of the [typical design patterns for scalability](#).

Review the scalability considerations in the appropriate [App Service web application reference architecture](#)

For other scalability topics, see the [scalability checklist](#) in the Azure Architecture Center.

### Security

Consider leveraging the [typical design patterns for security](#) where appropriate.

Review the security considerations in the appropriate [App Service web application reference architecture](#).

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

### Resiliency

Review the [typical design patterns for resiliency](#) and consider implementing these where appropriate.

You can find a number of [recommended practices for App Service](#) in the Azure Architecture Center.

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Deploy the scenario

### Prerequisites

- You must have an existing Azure account. If you don't have an Azure subscription, create a [free account](#) before you begin.
- You must sign up for an Azure DevOps organization. For more information, see [Quickstart: Create your organization](#).

### Walk-through

In this scenario, you'll use the Azure DevOps project to create your CI/CD pipeline.

The Azure DevOps project will deploy an App Service Plan, App Service, and an App Insights resource for you, as well as configure the Azure DevOps project for you.

Once you've deployed the Azure DevOps project and the build is completed, review the associated code changes, work items, and test results. You will notice that no test results are displayed, because the code does not contain any tests to run.

Review the release definitions. Notice that a release pipeline has been set up, releasing our application into the Dev environment. Observe that there is a **continuous deployment trigger** set from the **Drop** build artifact, with automatic releases into the Dev environment. As part of a continuous deployment process, you may see releases that span multiple environments. A release can span both infrastructure (using techniques such as infrastructure-as-code), and can also deploy the application packages required along with any post-configuration tasks.

## Additional considerations

- Consider leveraging one of the [tokenization tasks](#) available in the VSTS marketplace.
- Consider using the [Deploy: Azure Key Vault](#) VSTS task to download secrets from an Azure Key Vault into your release. You can then use those secrets as variables in your release definition, so you can avoid storing them in source control.
- Consider using [release variables](#) in your release definitions to drive configuration changes of your environments. Release variables can be scoped to an entire release or a given environment. When using variables for secret information, ensure that you select the padlock icon.
- Consider using [deployment gates](#) in your release pipeline. This lets you leverage monitoring data in association with external systems (for example, incident management or additional bespoke systems) to determine whether a release should be promoted.
- Where manual intervention in a release pipeline is required, consider using the [approvals](#) functionality.
- Consider using [Application Insights](#) and additional monitoring tools as early as possible in your release pipeline. Many organizations only begin monitoring in their production environment; by monitoring your other environments, you can identify bugs earlier in the development process and avoid issues in your production environment.

## Pricing

Azure DevOps costs depend on the number of users in your organization that require access, along with other factors like the number of concurrent build/releases required and number of test users. For more information, see [Azure DevOps pricing](#).

- [Azure DevOps](#) is a service that enables you to manage your development life cycle. It is paid for on a per-user per-month basis. There may be additional charges dependent upon concurrent pipelines needed, in addition to any additional test users or user basic licenses.

## Related resources

- [What is DevOps?](#)
- [DevOps at Microsoft - How we work with Azure DevOps](#)
- [Step-by-step Tutorials: DevOps with Azure DevOps](#)
- [Devops Checklist](#)
- [Create a CI/CD pipeline for .NET with the Azure DevOps project](#)

# Dev/test environments for SAP workloads on Azure

10/9/2018 • 3 minutes to read • [Edit Online](#)

This example shows how to establish a dev/test environment for SAP NetWeaver in a Windows or Linux environment on Azure. The database used is AnyDB, the SAP term for any supported DBMS (that isn't SAP HANA). Because this architecture is designed for non-production environments, it's deployed with just a single virtual machine (VM) and its size can be changed to accommodate your organization's needs.

For production use cases review the SAP reference architectures available below:

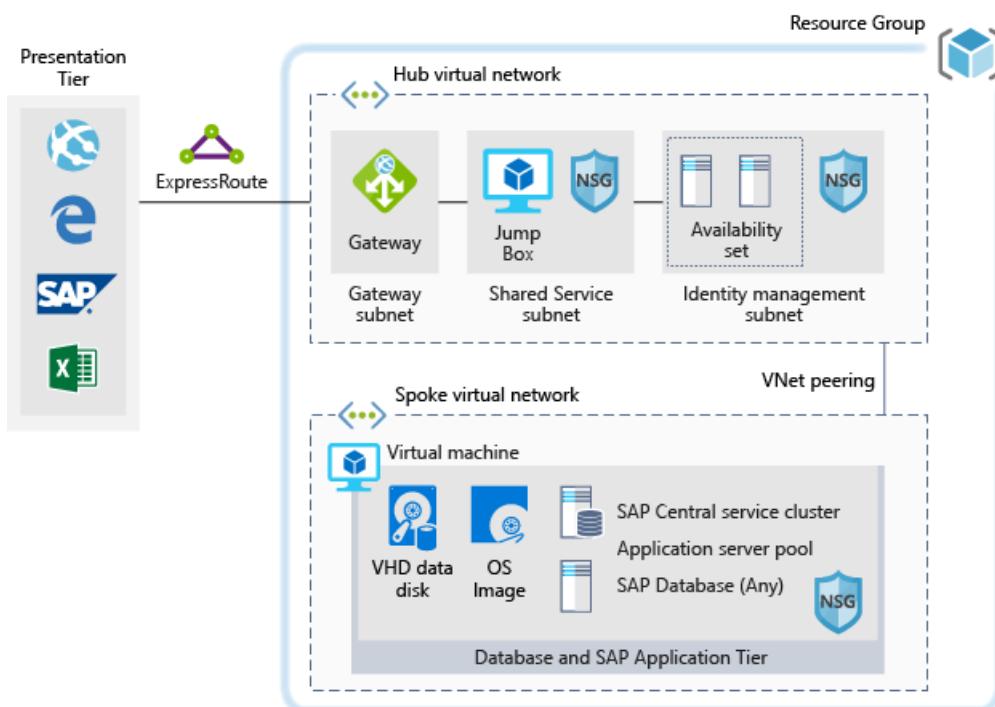
- [SAP NetWeaver for AnyDB](#)
- [SAP S/4HANA](#)
- [SAP on Azure large instances](#)

## Relevant use cases

Consider this scenario for the following use cases:

- Non-critical SAP non-productive workloads (sandbox, development, test, quality assurance)
- Non-critical SAP business workloads

## Architecture



This scenario demonstrates provisioning a single SAP system database and SAP application server on a single virtual machine. The data flows through the scenario as follows:

1. Customers use the SAP user interface or other client tools (Excel, a web browser, or other web application) to access the Azure-based SAP system.
2. Connectivity is provided through the use of an established ExpressRoute. The ExpressRoute connection is terminated in Azure at the ExpressRoute gateway. Network traffic routes through the ExpressRoute gateway to the gateway subnet, and from the gateway subnet to the application-tier spoke subnet (see the [hub-spoke](#) pattern) and via a Network Security Gateway to the SAP application virtual machine.

3. The identity management servers provide authentication services.
4. The jump box provides local management capabilities.

## Components

- [Virtual Networks](#) are the basis of network communication within Azure.
- [Virtual Machine](#) Azure Virtual Machines provides on-demand, high-scale, secure, virtualized infrastructure using Windows or Linux Server.
- [ExpressRoute](#) lets you extend your on-premises networks into the Microsoft cloud over a private connection facilitated by a connectivity provider.
- [Network Security Group](#) lets you limit network traffic to resources in a virtual network. 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.
- [Resource Groups](#) act as logical containers for Azure resources.

## Considerations

### Availability

Microsoft offers a service level agreement (SLA) for single VM instances. For more information on Microsoft Azure Service Level Agreement for Virtual Machines [SLA For Virtual Machines](#)

### Scalability

For general guidance on designing scalable solutions, see the [scalability checklist](#) in the Azure Architecture Center.

### Security

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

### Resiliency

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Pricing

To help you explore the cost of running this scenario, all of the services are pre-configured in the cost calculator examples below. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic.

We have provided four sample cost profiles based on amount of traffic you expect to receive:

SIZE	SAPS	VM TYPE	STORAGE	AZURE PRICING CALCULATOR
Small	8000	D8s_v3	2xP20, 1xP10	<a href="#">Small</a>
Medium	16000	D16s_v3	3xP20, 1xP10	<a href="#">Medium</a>
Large	32000	E32s_v3	3xP20, 1xP10	<a href="#">Large</a>
Extra Large	64000	M64s	4xP20, 1xP10	<a href="#">Extra Large</a>

### NOTE

This pricing is a guide that only indicates the VMs and storage costs. It excludes networking, backup storage, and data ingress/egress charges.

- **Small:** A small system consists of VM type D8s\_v3 with 8x vCPUs, 32 GB RAM and 200 GB temp storage, additionally two 512 GB and one 128 GB premium storage disks.
- **Medium:** A medium system consists of VM type D16s\_v3 with 16x vCPUs, 64 GB RAM and 400 GB temp storage, additionally three 512 GB and one 128 GB premium storage disks.
- **Large:** A large system consists of VM type E32s\_v3 with 32x vCPUs, 256 GB RAM and 512 GB temp storage, additionally three 512GB and one 128GB premium storage disks.
- **Extra Large:** An extra large system consists of a VM type M64s with 64x vCPUs, 1024 GB RAM and 2000 GB temp storage, additionally four 512 GB and one 128 GB premium storage disks.

## Deployment

Click here to deploy the underlying infrastructure for this scenario.



### NOTE

SAP and Oracle are not installed during this deployment. You will need to deploy these components separately.

# Running SAP production workloads using an Oracle Database on Azure

10/5/2018 • 5 minutes to read • [Edit Online](#)

SAP systems are used to run mission-critical business applications. Any outage disrupts key processes and can cause increased expenses or lost revenue. Avoiding these outcomes requires an SAP infrastructure that is highly available and resilient when failures occur.

Building a highly available SAP environment requires eliminating single points of failure in your system architecture and processes. Single points of failure can be caused by site failures, errors in system components, or even human error.

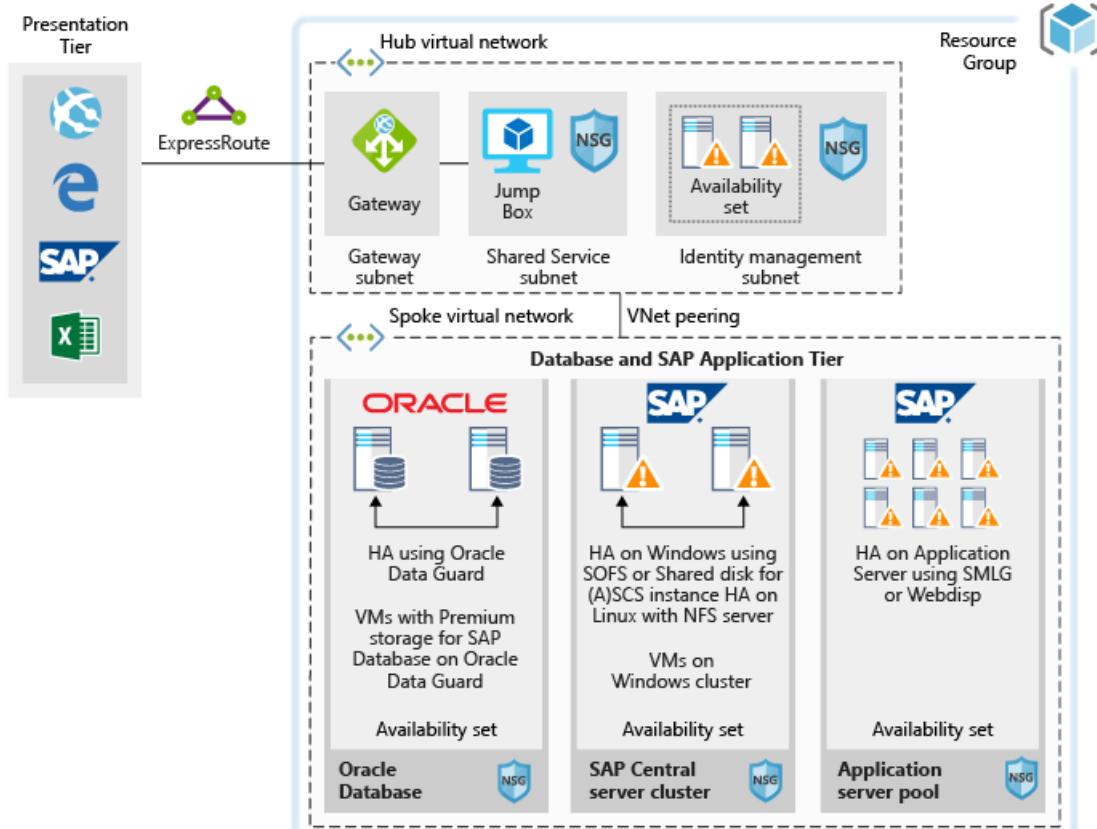
This example scenario demonstrates an SAP deployment on Windows or Linux virtual machines (VMs) on Azure, along with a High Availability (HA) Oracle database. For your SAP deployment, you can use VMs of different sizes based on your requirements.

## Relevant use cases

Consider this example for the following scenarios:

- Mission-critical workloads running on SAP.
- Non-critical SAP workloads.
- Test environments for SAP that simulate a high-availability environment.

## Architecture



This example includes a high availability configuration for an Oracle database, SAP central services, and multiple

SAP application servers running on different virtual machines. The Azure network uses a [hub-and-spoke topology](#) for security purposes. The data flows through the solution as follows:

1. Users access the SAP system via the SAP user interface, a web browser, or other client tools like Microsoft Excel. An ExpressRoute connection provides access from the organization's on-premises network to resources running in Azure.
2. The ExpressRoute terminates in Azure at the ExpressRoute virtual network (VNet) gateway. Network traffic is routed to a gateway subnet through the ExpressRoute gateway created in the hub VNet.
3. The hub VNet is peered to a spoke VNet. The application tier subnet hosts the virtual machines running SAP in an availability set.
4. The identity management servers provide authentication services for the solution.
5. The jump box is used by system administrators to securely manage resources deployed in Azure.

## Components

- [Virtual Networks](#) are used in this scenario to create a virtual hub-and-spoke topology in Azure.
- [Virtual Machines](#) provide the compute resources for each tier of the solution. Each cluster of virtual machines is configured as an [availability set](#).
- [ExpressRoute](#) extends your on-premises network into the Microsoft cloud through a private connection established by a connectivity provider.
- [Network Security Groups \(NSG\)](#) limit network access to the resources in a virtual network. An NSG contains a list of security rules that allow or deny network traffic based on source or destination IP address, port, and protocol.
- [Resource Groups](#) act as logical containers for Azure resources.

## Alternatives

SAP provides flexible options for different combinations of operating system, database management system, and VM types in an Azure environment. For more information, see [SAP note 1928533](#), "SAP Applications on Azure: Supported Products and Azure VM Types".

## Considerations

Recommended practices are defined for building highly available SAP environments in Azure. For more information, see [High-availability architecture and scenarios for SAP NetWeaver](#). For more information, see [High availability of SAP applications on Azure VMs](#).

- Oracle databases also have recommended practices for Azure. For more information, see [Designing and implementing an Oracle database in Azure](#).
- Oracle Data Guard is used to eliminate single points of failure for mission critical Oracle databases. For more information, see [Implementing Oracle Data Guard on a Linux virtual machine in Azure](#).

Microsoft Azure offers infrastructure services that can be used to deploy SAP products with an Oracle database. For more information, see [Deploying an Oracle DBMS on Azure for an SAP workload](#).

## Pricing

To help you explore the cost of running this scenario, all of the services are pre-configured in the cost calculator examples below. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic.

We have provided four sample cost profiles based on amount of traffic you expect to receive:

SIZE	SAPS	DB VM TYPE	DB STORAGE	(A)SCS VM	(A)SCS STORAGE	APP VM TYPE	APP STORAGE	AZURE PRICING CALCULATOR
Small	30000	DS13_v2	4xP20, 1xP20	DS11_v2	1x P10	DS13_v2	1x P10	Small
Medium	70000	DS14_v2	6xP20, 1xP20	DS11_v2	1x P10	4x DS13_v2	1x P10	Medium
Large	180000	E32s_v3	5xP30, 1xP20	DS11_v2	1x P10	6x DS14_v2	1x P10	Large
Extra Large	250000	M64s	6xP30, 1xP30	DS11_v2	1x P10	10x DS14_v2	1x P10	Extra Large

#### NOTE

This pricing is a guide and only indicates the VMs and storage costs. It excludes networking, backup storage, and data ingress/egress charges.

- **Small:** A small system consists of VM type DS13\_v2 for the database server with 8x vCPUs, 56-GB RAM, and 112-GB temp storage, additionally five 512-GB premium storage disks. An SAP Central Instance server using a DS11\_v2 VM types with 2x vCPUs 14-GB RAM and 28-GB temp storage. A single VM type DS13\_v2 for the SAP application server with 8x vCPUs, 56-GB RAM, and 400-GB temp storage, additionally one 128-GB premium storage disk.
- **Medium:** A medium system consists of VM type DS14\_v2 for the database server with 16x vCPUs, 112 GB RAM, and 800-GB temp storage, additionally seven 512-GB premium storage disks. An SAP Central Instance server using a DS11\_v2 VM types with 2x vCPUs 14-GB RAM and 28-GB temp storage. Four VM type DS13\_v2 for the SAP application server with 8x vCPUs, 56-GB RAM, and 400-GB temp storage, additionally one 128-GB premium storage disk.
- **Large:** A large system consists of VM type E32s\_v3 for the database server with 32x vCPUs, 256-GB RAM and 800-GB temp storage, additionally three 512 GB and one 128-GB premium storage disks. An SAP Central Instance server using a DS11\_v2 VM types with 2x vCPUs 14-GB RAM and 28-GB temp storage. Six VM type DS14\_v2 for the SAP application servers with 16x vCPUs, 112 GB RAM, and 224 GB temp storage, additionally six 128-GB premium storage disk.
- **Extra Large:** An extra large system consists of the M64s VM type for the database server with 64x vCPUs, 1024 GB RAM, and 2000 GB temp storage, additionally seven 1024-GB premium storage disks. An SAP Central Instance server using a DS11\_v2 VM types with 2x vCPUs 14-GB RAM and 28-GB temp storage. 10 VM type DS14\_v2 for the SAP application servers with 16x vCPUs, 112 GB RAM, and 224 GB temp storage, additionally ten 128-GB premium storage disk.

## Deployment

Use the following link to deploy the underlying infrastructure for this scenario.



**NOTE**

SAP and Oracle are not installed during this deployment. You will need to deploy these components separately.

## Related resources

For other information about running SAP production workloads in Azure, review the following reference architectures:

- [SAP NetWeaver for AnyDB](#)
- [SAP S/4HANA](#)
- [SAP HANA large instances](#)

# An e-commerce front end on Azure

10/5/2018 • 6 minutes to read • [Edit Online](#)

This example scenario walks you through an implementation of an e-commerce front end using Azure platform as a service (PaaS) tools. Many e-commerce websites face seasonality and traffic variability over time. When demand for your products or services takes off, whether predictably or unpredictably, using PaaS tools will allow you to handle more customers and more transactions automatically. Additionally, this scenario takes advantage of cloud economics by paying only for the capacity you use.

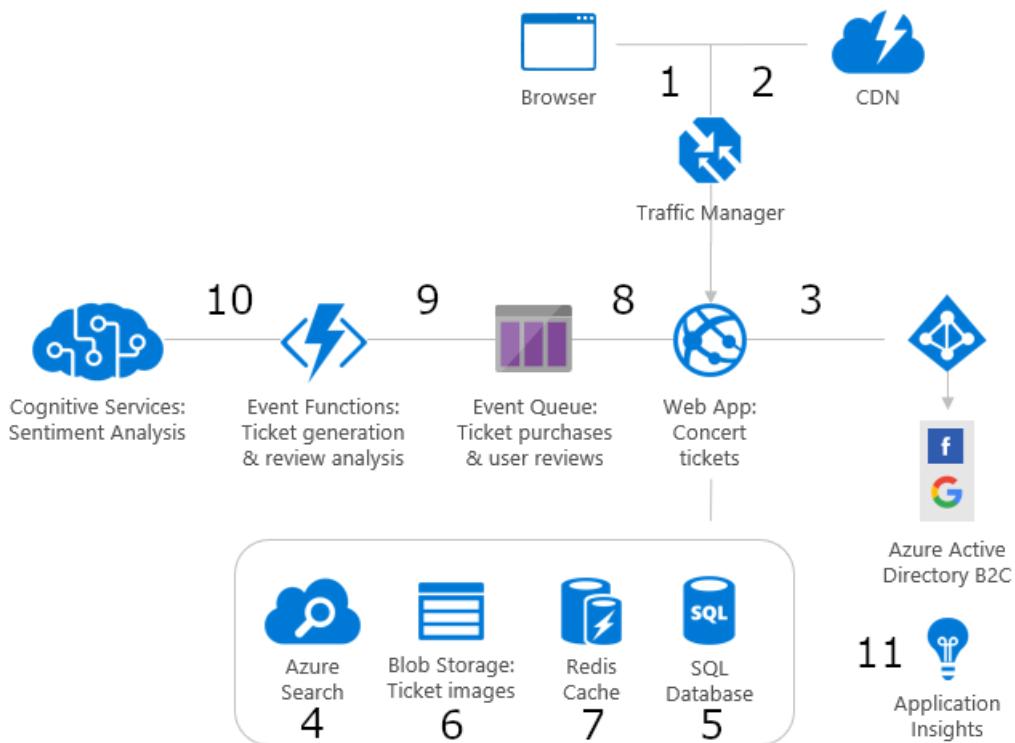
This document will help you learn about various Azure PaaS components and considerations used to bring together to deploy a sample e-commerce application, *Relecloud Concerts*, an online concert ticketing platform.

## Relevant use cases

Consider this scenario for the following use cases:

- Building an application that needs elastic scale to handle bursts of users at different times.
- Building an application that is designed to operate at high availability in different Azure regions around the world.

## Architecture



This scenario covers purchasing tickets from an e-commerce site, the data flows through the scenario as follows:

1. Azure Traffic Manager routes a user's request to the e-commerce site hosted in Azure App Service.
2. Azure CDN serves static images and content to the user.
3. User signs in to the application through an Azure Active Directory B2C tenant.
4. User searches for concerts using Azure Search.
5. Web site pulls concert details from Azure SQL Database.
6. Web site refers to purchased ticket images in Blob Storage.

7. Database query results are cached in Azure Redis Cache for better performance.
8. User submits ticket orders and concert reviews, which are placed in the queue.
9. Azure Functions processes order payment and concert reviews.
10. Cognitive services provide an analysis of the concert review to determine the sentiment (positive or negative).
11. Application Insights provides performance metrics for monitoring the health of the web application.

## Components

- [Azure CDN](#) delivers static, cached content from locations close to users to reduce latency.
- [Azure Traffic Manager](#) controls the distribution of user traffic for service endpoints in different Azure regions.
- [App Services - Web Apps](#) hosts web applications allowing auto-scale and high availability without having to manage infrastructure.
- [Azure Active Directory - B2C](#) is an identity management service that enables customization and control over how customers sign up, sign in, and manage their profiles in an application.
- [Storage Queues](#) stores large numbers of queue messages that can be accessed by an application.
- [Functions](#) are serverless compute options that allow applications to run on-demand without having to manage infrastructure.
- [Cognitive Services - Sentiment Analysis](#) uses machine learning APIs and enables developers to easily add intelligent features – such as emotion and video detection; facial, speech, and vision recognition; and speech and language understanding – into applications.
- [Azure Search](#) is a search-as-a-service cloud solution that provides a rich search experience over private, heterogenous content in web, mobile, and enterprise applications.
- [Storage Blobs](#) are optimized to store large amounts of unstructured data, such as text or binary data.
- [Redis Cache](#) improves the performance and scalability of systems that rely heavily on back-end data stores by temporarily copying frequently accessed data to fast storage located close to the application.
- [SQL Database](#) is a general-purpose relational database managed service in Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML.
- [Application Insights](#) is designed to help you continuously improve performance and usability by automatically detecting performance anomalies through built-in analytics tools to help understand what users do with an app.

## Alternatives

Many other technologies are available for building a customer facing application focused on e-commerce at scale. These cover both the front end of the application as well as the data tier.

Other options for the web tier and functions include:

- [Service Fabric](#) - A platform focused around building distributed components that benefit from being deployed and run across a cluster with a high degree of control. Service Fabric can also be used to host containers.
- [Azure Kubernetes Service](#) - A platform for building and deploying container-based solutions that can be used as one implementation of a microservices architecture. This allows for agility of different components of the application to be able to scale independently on demand.
- [Azure Container Instances](#) - A way of quickly deploying and running containers with a short lifecycle. Containers here are deployed to run a quick processing job such as processing a message or performing a calculation and then deprovisioned as soon as they are complete.
- [Service Bus][service-bus] could be used in place of Storage Queue's.

Other options for the data tier include:

- [Cosmos DB](#): Microsoft's globally distributed, multi-model database. This service provides a platform to run other data models such as Mongo DB, Cassandra, Graph data, or simple table storage.

## Considerations

## Availability

- Consider leveraging the [typical design patterns for availability](#) when building your cloud application.
- Review the availability considerations in the appropriate [App Service web application reference architecture](#)
- For additional considerations concerning availability, see the [availability checklist](#) in the Azure Architecture Center.

## Scalability

- When building a cloud application be aware of the [typical design patterns for scalability](#).
- Review the scalability considerations in the appropriate [App Service web application reference architecture](#)
- For other scalability topics, see the [scalability checklist](#) available in the Azure Architecture Center.

## Security

- Consider leveraging the [typical design patterns for security](#) where appropriate.
- Review the security considerations in the appropriate [App Service web application reference architecture](#).
- Consider following a [secure development lifecycle](#) process to help developers build more secure software and address security compliance requirements while reducing development cost.
- Review the blueprint architecture for [Azure PCI DSS compliance](#).

## Resiliency

- Consider leveraging the [circuit breaker pattern](#) to provide graceful error handling should one part of the application not be available.
- Review the [typical design patterns for resiliency](#) and consider implementing these where appropriate.
- You can find a number of [recommended practices for App Service](#) in the Azure Architecture Center.
- Consider using active [geo-replication](#) for the data tier and [geo-redundant](#) storage for images and queues.
- For a deeper discussion on [resiliency](#), see the relevant article in the Azure Architecture Center.

## Deploy the scenario

To deploy this scenario, you can follow this [step-by-step tutorial](#) demonstrating how to manually deploy each component. This tutorial also provides a .NET sample application that runs a simple ticket purchasing application. Additionally, there is a Resource Manager template to automate the deployment of most of the Azure resources.

## Pricing

Explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case change the appropriate variables to match your expected traffic.

We have provided three sample cost profiles based on amount of traffic you expect to get:

- **Small:** This pricing example represents the components necessary to build the out for a minimum production level instance. Here we are assuming a small number of users, numbering only in a few thousand per month. The app is using a single instance of a standard web app that will be enough to enable autoscaling. Each of the other components are scaled to a basic tier that will allow for a minimum amount of cost but still ensure that there is SLA support and enough capacity to handle a production level workload.
- **Medium:** This pricing example represents the components indicative of a moderate size deployment. Here we estimate approximately 100,000 users using the system over the course of a month. The expected traffic is handled in a single app service instance with a moderate standard tier. Additionally, moderate tiers of cognitive and search services are added to the calculator.
- **Large:** This pricing example represents an application meant for high scale, at the order of millions of users per month moving terabytes of data. At this level of usage high performance, premium tier web apps deployed in multiple regions fronted by traffic manager is required. Data consists of the following: storage, databases, and CDN, are configured for terabytes of data.

## Related resources

- [Reference Architecture for Multi-Region Web Application](#)
- [eShop on Containers Reference Example](#)

# Migrating a legacy web application to an API-based architecture on Azure

10/9/2018 • 4 minutes to read • [Edit Online](#)

An e-commerce company in the travel industry is modernizing their legacy browser-based software stack. While their existing stack is mostly monolithic, some [SOAP-based HTTP services](#) exist from a recent project. They are considering the creation of additional revenue streams to monetize some of the internal intellectual property that's been developed.

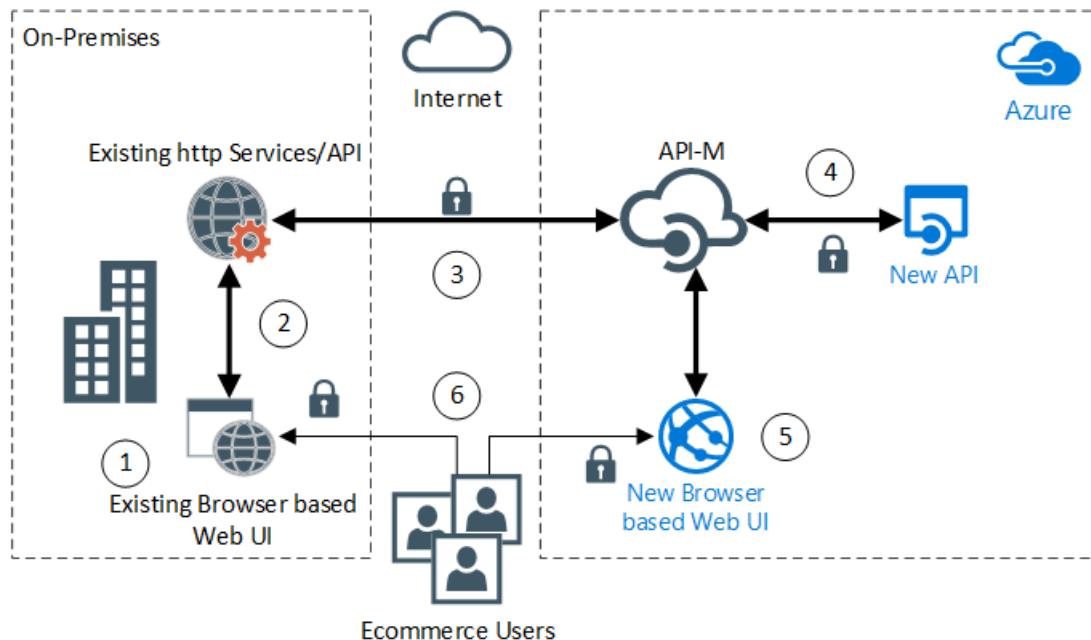
Goals for the project include addressing technical debt, improving ongoing maintenance, and accelerating feature development with fewer regression bugs. The project will use an iterative process to avoid risk, with some steps performed in parallel:

- The development team will modernize the application back end, which is composed of relational databases hosted on VMs.
- The in-house development team will write new business functionality, which will be exposed over new HTTP APIs.
- A contract development team will build a new browser-based UI, which will be hosted in Azure.

New application features will be delivered in stages. They will *gradually replace* the existing browser-based client-server UI functionality (hosted on-premises) that powers their e-commerce business today.

The management team does not want to modernize unnecessarily. They also want to maintain control of scope and costs. To do this, they have decided to preserve their existing SOAP HTTP services. They also intend to minimize changes to the existing UI. [Azure API Management \(APIM\)](#) can be utilized to address many of the project's requirements and constraints.

## Architecture



The new UI will be hosted as a platform as a service (PaaS) application on Azure, and will depend on both existing and new HTTP APIs. These APIs will ship with a better-designed set of interfaces enabling better performance, easier integration, and future extensibility.

## Components and Security

1. The existing on-premises web application will continue to directly consume the existing on-premises web services.
2. Calls from the existing web app to the existing HTTP services will remain unchanged. These calls are internal to the corporate network.
3. Inbound calls are made from Azure to the existing internal services:
  - The security team allows traffic from the APIM instance to pass through the corporate firewall to the existing on-premises services [using secure transport \(HTTPS/SSL\)](#).
  - The operations team will allow inbound calls to the services only from the APIM instance. This requirement is met by [white-listing the IP address of the APIM instance](#) within the corporate network perimeter.
  - A new module is configured into the on-premises HTTP services request pipeline (to act upon **only** those connections originating externally), which will validate [a certificate which APIM will provide](#).
4. The new API:
  - Is surfaced only through the APIM instance, which will provide the API facade. The new API won't be accessed directly.
  - Is developed and published as an [Azure PaaS Web API App](#).
  - Is white-listed (via [Web App settings](#)) to accept only the [APIM VIP](#).
  - Is hosted in Azure Web Apps with Secure Transport/SSL turned on.
  - Has authorization enabled, [provided by the Azure App Service](#) using Azure Active Directory and OAuth2.
5. The new browser-based web application will depend on the Azure API Management instance for **both** the existing HTTP API and the new API.

The APIM instance will be configured to map the legacy HTTP services to a new API contract. By doing this, the new Web UI is unaware of the integration with a set of legacy services/APIs and new APIs. In the future, the project team will gradually port functionality to the new APIs and retire the original services. These changes will be handled within APIM configuration, leaving the front-end UI unaffected and avoiding redevelopment work.

## Alternatives

- If the organization was planning to move their infrastructure entirely to Azure, including the VMs hosting the legacy applications, then APIM would still be a great option since it can act as a facade for any addressable HTTP endpoint.
- If the customer had decided to keep the existing endpoints private and not expose them publicly, their API Management instance could be linked to an [Azure Virtual Network \(VNet\)](#):
  - In an [Azure lift and shift scenario](#) linked to their deployed Azure Virtual Network, the customer could directly address the back-end service through private IP addresses.
  - In the on-premises scenario, the API Management instance could reach back to the internal service privately via an [Azure VPN gateway and site-to-site IPSec VPN connection](#) or [ExpressRoute](#) making this a [hybrid Azure and on-premises scenario](#).
- The API Management instance can be kept private by deploying the API Management instance in Internal mode. The deployment could then be used with an [Azure Application Gateway](#) to enable public access for some APIs while others remain internal. For more information, see [Connecting APIM in internal mode to a VNET](#).

### NOTE

For general information on connecting API Management to a VNET, [see here](#).

## Availability and scalability

- Azure API Management can be [scaled out](#) by choosing a pricing tier and then adding units.
- Scaling also happens [automatically with auto scaling](#).

- Deploying across multiple regions will enable fail over options and can be done in the [Premium tier](#).
- Consider [Integrating with Azure Application Insights](#), which also surfaces metrics through [Azure Monitor](#) for monitoring.

## Deployment

To get started, [create an Azure API Management instance in the portal](#).

Alternatively, you can choose from an existing Azure Resource Manager [quickstart template](#) that aligns to your specific use case.

## Pricing

API Management is offered in four tiers: developer, basic, standard, and premium. You can find detailed guidance on the difference in these tiers at the [Azure API Management pricing guidance here](#).

Customers can scale API Management by adding and removing units. Each unit has capacity that depends on its tier.

### NOTE

The Developer tier can be used for evaluation of the API Management features. The Developer tier should not be used for production.

To view projected costs and customize to your deployment needs, you can modify the number of scale units and App Service instances in the [Azure Pricing Calculator](#).

## Related resources

Check out the extensive Azure API Management [documentation](#) and [reference articles](#).

# Intelligent product search engine for e-commerce

10/5/2018 • 6 minutes to read • [Edit Online](#)

This example scenario shows how using a dedicated search service can dramatically increase the relevance of search results for your e-commerce customers.

Search is the primary mechanism through which customers find and ultimately purchase products, making it essential that search results are relevant to the *intent* of the search query, and that the end-to-end search experience matches that of search giants by providing near-instant results, linguistic analysis, geo-location matching, filtering, faceting, auto-complete, hit highlighting etc.

Imagine a typical e-commerce web application with product data stored in a relational database like SQL Server or Azure SQL Database. Search queries are often handled inside the database using `LIKE` queries or [Full-Text Search](#) features. By using [Azure Search](#) instead, you free up your operational database from the query processing and you can easily start taking advantage of those hard-to-implement features that provide your customers with the best possible search experience. Also, because Azure Search is a platform as a service (PaaS) component, you don't have to worry about managing infrastructure or becoming a search expert.

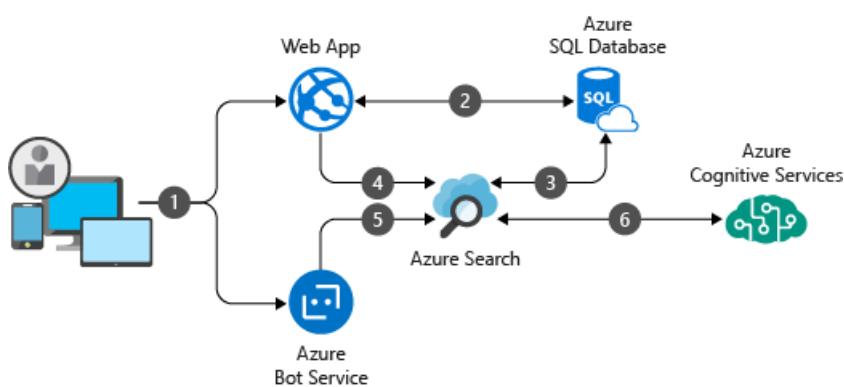
## Relevant use cases

These other uses cases have similar design patterns:

- Finding real estate listings or stores near the user's physical location.
- Searching for articles in a news site or looking for sports results, with a higher preference for more *recent* information.
- Searching through large repositories for *document-centric* organizations like policy makers and notaries.

Ultimately *any* application that has some form of search functionality can benefit from having a dedicated search service.

## Architecture



This scenario covers an e-commerce solution where customers can search through a product catalog.

1. Customers navigate to the **e-commerce web application** from any device.
2. The product catalog is maintained in an **Azure SQL Database** for transactional processing.
3. Azure Search uses a **search indexer** to automatically keep its search index up-to-date through integrated change tracking.
4. Customer's search queries are offloaded to the **Azure Search** service, which processes the query and returns the most relevant results.

5. As an alternative to a web-based search experience, customers can also use a **conversational bot** in social media or straight from digital assistants to search for products and incrementally refine their search query and results.
6. Optionally, the **Cognitive Search** feature can be used to apply artificial intelligence for even smarter processing.

## Components

- [App Services - Web Apps](#) hosts web applications allowing auto-scale and high availability without having to manage infrastructure.
- [SQL Database](#) is a general-purpose relational database-managed service in Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML.
- [Azure Search](#) is a search-as-a-service cloud solution that provides a rich search experience over private, heterogenous content in web, mobile, and enterprise applications.
- [Bot Service](#) provides tools to build, test, deploy, and manage intelligent bots.
- [Cognitive Services](#) lets you use intelligent algorithms to see, hear, speak, understand, and interpret your user needs through natural methods of communication.

## Alternatives

- You could use **in-database search** capabilities, for example, through SQL Server full-text search, but then your transactional store also processes queries (increasing the need for processing power) and the search capabilities inside the database are more limited.
- You could host the open-source [Apache Lucene](#) (on which Azure Search is built upon) on Azure Virtual Machines, but then you are back to managing Infrastructure-as-a-Service (IaaS) and don't benefit from the many features that Azure Search provides on top of Lucene.
- You could also consider deploying [Elastic Search](#) from the Azure Marketplace, which is an alternative and capable search product from a third-party vendor, but also in this case you are running an IaaS workload.

Other options for the data tier include:

- [Cosmos DB](#) - Microsoft's globally distributed, multi-model database. Costmos DB provides a platform to run other data models such as Mongo DB, Cassandra, Graph data, or simple table storage. Azure Search also supports indexing the data from Cosmos DB directly.

## Considerations

### Scalability

The [pricing tier](#) of the Azure Search service doesn't determine the available features but is used mainly for [capacity planning](#) as it defines the maximum storage you get and how many partitions and replicas you can provision.

**Partitions** allow you to index more documents and get higher write throughputs, whereas **replicas** provide more Queries-Per-Second (QPS) and High Availability.

You can dynamically change the number of partitions and replicas but it's not possible to change the pricing tier, so you should carefully consider the right tier for your target workload. If you need to change the tier anyway, you will need to provision a new service side by side and reload your indexes there, at which point you can point your applications at the new service.

### Availability

Azure Search provides a [99.9% availability SLA](#) for *reads* (that is, querying) if you have at least two replicas, and for *updates* (that is, updating the search indexes) if you have at least three replicas. Therefore you should provision at least two replicas if you want your customers to be able to *search* reliably, and 3 if actual *changes to the index* should also be considered high availability operations.

If there is a need to make breaking changes to the index without downtime (for example, changing data types,

deleting or renaming fields), the index will need to be rebuilt. Similar to changing service tier, this means creating a new index, repopulating it with the data, and then updating your applications to point at the new index.

## Security

Azure Search is compliant with many [security and data privacy standards](#), which makes it possible to be used in most industries.

For securing access to the service, Azure Search uses two types of keys: **admin keys**, which allow you to perform *any* task against the service, and **query keys**, which can only be used for read-only operations like querying.

Typically, the application that performs the search does not update the index, so it should only be configured with a query key and not an admin key (especially if the search is performed from an end-user device like script running in a web browser).

## Search Relevance

How successful your e-commerce application is depends largely on the relevance of the search results to your customers. Carefully tuning your search service to provide optimal results based on user research, or relying on built-in features such as [search traffic analysis](#) to understand your customer's search patterns allows you to make decisions based on data.

Typical ways to tune your search service include:

- Using [scoring profiles](#) to influence the relevance of search results, for example, based on which field matched the query, how recent the data is, the geographical distance to the user, ...
- Using [Microsoft provided language analyzers](#) that use an advanced Natural Language Processing (NLP) stack to better interpret queries
- Using [custom analyzers](#) to ensure your products are found correctly, especially if you want to search on non-language based information like a product's make and model.

## Deploy this scenario

To deploy a more complete e-commerce version of this scenario, you can follow this [step-by-step tutorial](#) that provides a .NET sample application that runs a simple ticket purchasing application. It also includes Azure Search and uses many of the features discussed. Additionally, there is a Resource Manager template to automate the deployment of most of the Azure resources.

## Pricing

To explore the cost of running this scenario, all the services mentioned above are pre-configured in the cost calculator. To see how the pricing would change for your particular use case change the appropriate variables to match your expected usage.

We have provided three sample cost profiles based on amount of traffic you expect to get:

- **Small:** In this profile, we're using a single `Standard S1` Web App to host the website, the free tier of the Azure Bot service, a single `Basic` Azure Search service, and a `Standard S2` SQL Database.
- **Medium:** Here we are scaling up the Web App to two instances of the `Standard S3` tier, upgrading the Search Service to a `Standard S1` tier, and using a `Standard S6` SQL Database.
- **Large:** In the largest profile, we use four instances of a `Premium P2V2` Web App, upgrade the Azure Bot service to the `Standard S1` tier (with 1.000.000 messages in Premium channels), use 2 units of the `Standard S3` Azure Search service, and a `Premium P6` SQL Database.

## Related resources

To learn more about Azure Search, visit the [documentation center](#), check out the [samples](#), or see a full fledged

[demo site](#) in action.

# IoT and data analytics in the construction industry

10/5/2018 • 5 minutes to read • [Edit Online](#)

This example scenario is relevant to organizations building solutions that integrate data from many IoT devices into a comprehensive data analysis architecture to improve and automate decision making. Potential applications include construction, mining, manufacturing, or other industry solutions involving large volumes of data from many IoT-based data inputs.

In this scenario, a construction equipment manufacturer builds vehicles, meters, and drones that use IoT and GPS technologies to emit telemetry data. The company wants to modernize their data architecture to better monitor operating conditions and equipment health. Replacing the company's legacy solution using on-premises infrastructure would be both time and labor intensive, and would not be able to scale sufficiently to handle the anticipated data volume.

The company wants to build a cloud-based "smart construction" solution. It should gather a comprehensive set of data for a construction site and automate the operation and maintenance of the various elements of the site. The company's goals include:

- Integrating and analyzing all construction site equipment and data to minimize equipment downtime and reduce theft.
- Remotely and automatically controlling construction equipment to mitigate the effects of a labor shortage, ultimately requiring fewer workers and enabling lower-skilled workers to succeed.
- Minimizing the operating costs and labor requirements for the supporting infrastructure, while increasing productivity and safety.
- Easily scaling the infrastructure to support increases in telemetry data.
- Complying with all relevant legal requirements by provisioning resources in-country without compromising system availability.
- Using open-source software to maximize the investment in workers' current skills.

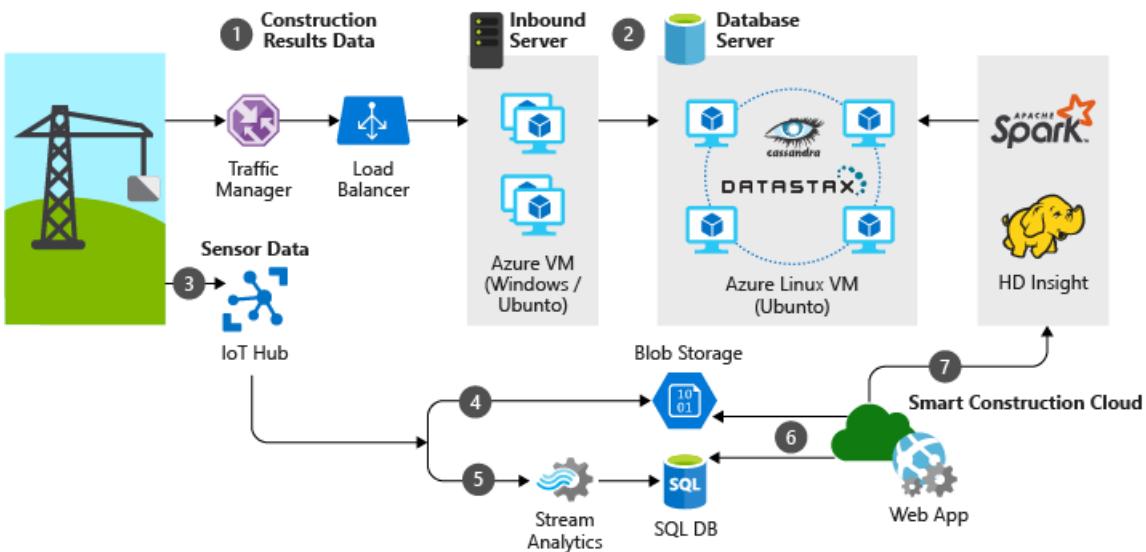
Using managed Azure services such as IoT Hub and HDInsight will allow the customer to rapidly build and deploy a comprehensive solution with a lower operating cost. If you have additional data analytics needs, you should review the list of available [fully managed data analytics services in Azure](#).

## Relevant use cases

Consider this solution for the following use cases:

- Construction, mining, or equipment manufacturing scenarios
- Large-scale collection of device data for storage and analysis
- Ingestion and analysis of large datasets

## Architecture



The data flows through the solution as follows:

1. Construction equipment collects sensor data and sends the construction results data at regular intervals to load balanced web services hosted on a cluster of Azure virtual machines.
2. The custom web services ingest the construction results data and store it in an Apache Cassandra cluster also running on Azure virtual machines.
3. Another dataset is gathered by IoT sensors on various construction equipment and sent to IoT Hub.
4. Raw data collected is sent directly from IoT Hub to Azure blob storage and is immediately available for viewing and analysis.
5. Data collected via IoT Hub is processed in near real time by an Azure Stream Analytics job and stored in an Azure SQL database.
6. The Smart Construction Cloud web application is available to analysts and end users to view and analyze sensor data and imagery.
7. Batch jobs are initiated on demand by users of the web application. The batch job runs in Apache Spark on HDInsight and analyzes new data stored in the Cassandra cluster.

## Components

- **IoT Hub** acts as a central message hub for secure bi-directional communication with per-device identity between the cloud platform and the construction equipment and other site elements. IoT Hub can rapidly collect data for each device for ingestion into the data analytics pipeline.
- **Azure Stream Analytics** is an event-processing engine that can analyze high volumes of data streaming from devices and other data sources. It also supports extracting information from data streams to identify patterns and relationships. In this scenario, Stream Analytics ingests and analyzes data from IoT devices and stores the results in Azure SQL Database.
- **Azure SQL Database** contains the results of analyzed data from IoT devices and meters, which can be viewed by analysts and users via an Azure-based Web application.
- **Blob storage** stores image data gathered from the IoT hub devices. The image data can be viewed via the web application.
- **Traffic Manager** controls the distribution of user traffic for service endpoints in different Azure regions.
- **Load Balancer** distributes data submissions from construction equipment devices across the VM-based web services to provide high availability.
- **Azure Virtual Machines** host the web services that receive and ingest the construction results data into the Apache Cassandra database.
- **Apache Cassandra** is a distributed NoSQL database used to store construction data for later processing by Apache Spark.
- **Web Apps** hosts the end-user web application, which can be used to query and view source data and images.

Users can also initiate batch jobs in Apache Spark via the application.

- [Apache Spark on HDInsight](#) supports in-memory processing to boost the performance of big-data analytic applications. In this scenario, Spark is used to run complex algorithms over the data stored in Apache Cassandra.

## Alternatives

- [Cosmos DB](#) is an alternative NoSQL database technology. Cosmos DB provides [multi-master support at global scale](#) with [multiple well-defined consistency levels](#) to meet various customer requirements. It also supports the [Cassandra API](#).
- [Azure Databricks](#) is an Apache Spark-based analytics platform optimized for Azure. It is integrated with Azure to provide one-click setup, streamlined workflows, and an interactive collaborative workspace.
- [Data Lake Storage](#) is an alternative to Blob storage. For this scenario, Data Lake Storage was not available in the targeted region.
- [Web Apps](#) could also be used to host the web services for ingesting construction results data.
- Many technology options are available for real-time message ingestion, data storage, stream processing, storage of analytical data, and analytics and reporting. For an overview of these options, their capabilities, and key selection criteria, see [Big data architectures: Real-time processing](#) in the [Azure Data Architecture Guide](#).

## Considerations

The broad availability of Azure regions is an important factor for this scenario. Having more than one region in a single country can provide disaster recovery while also enabling compliance with contractual obligations and law enforcement requirements. Azure's high-speed communication between regions is also an important factor in this scenario.

Azure support for open-source technologies allowed the customer to take advantage of their existing workforce skills. The customer can also accelerate the adoption of new technologies with lower costs and operating workloads compared to an on-premises solution.

## Pricing

The following considerations will drive a substantial portion of the costs for this solution.

- Azure virtual machine costs will increase linearly as additional instances are provisioned. Virtual machines that are deallocated will only incur storage costs, and not compute costs. These deallocated machines can then be reallocated when demand is high.
- [IoT Hub](#) costs are driven by the number of IoT units provisioned as well as the service tier chosen, which determines the number of messages per day per unit allowed.
- [Stream Analytics](#) is priced by the number of streaming units required to process the data into the service.

## Related resources

To see an implementation of a similar architecture, read the [Komatsu customer story](#).

Guidance for big data architectures is available in the [Azure Data Architecture Guide](#).

# Data warehousing and analytics for sales and marketing

10/5/2018 • 5 minutes to read • [Edit Online](#)

This example scenario demonstrates a data pipeline that integrates large amounts of data from multiple sources into a unified analytics platform in Azure. This specific scenario is based on a sales and marketing solution, but the design patterns are relevant for many industries requiring advanced analytics of large datasets such as e-commerce, retail, and healthcare.

This example demonstrates a sales and marketing company that creates incentive programs. These programs reward customers, suppliers, salespeople, and employees. Data is fundamental to these programs, and the company wants to improve the insights gained through data analytics using Azure.

The company needs a modern approach to analysis data, so that decisions are made using the right data at the right time. The company's goals include:

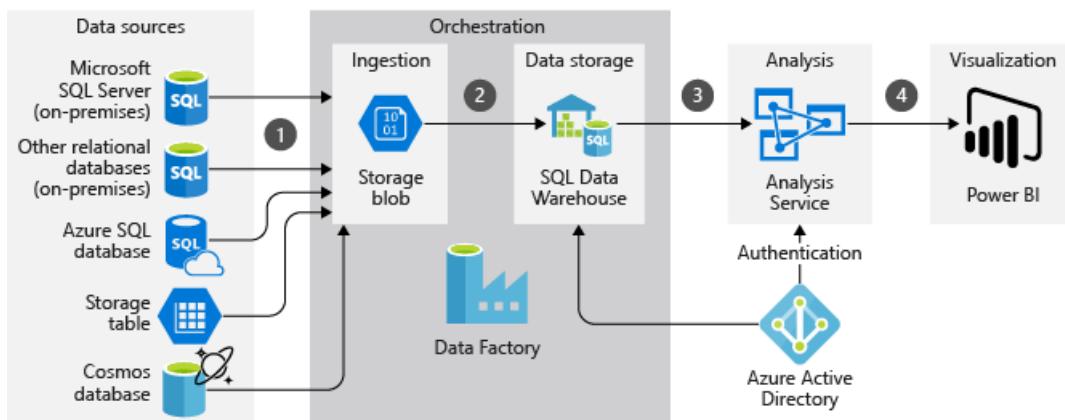
- Combining different kinds of data sources into a cloud-scale platform.
- Transforming source data into a common taxonomy and structure, to make the data consistent and easily compared.
- Loading data using a highly parallelized approach that can support thousands of incentive programs, without the high costs of deploying and maintaining on-premises infrastructure.
- Greatly reducing the time needed to gather and transform data, so you can focus on analyzing the data.

## Relevant use cases

You could also use this approach to accomplish the following:

- Establish a data warehouse to be a single source of truth for your data.
- Integrate relational data sources with other unstructured datasets.
- Use semantic modeling and powerful visualization tools for simpler data analysis.

## Architecture



The data flows through the solution as follows:

1. For each data source, any updates are exported periodically into a staging area in Azure Blob storage.
2. Data Factory incrementally loads the data from Blob storage into staging tables in SQL Data Warehouse. The data is cleansed and transformed during this process. Polybase can parallelize the process for large datasets.

3. After loading a new batch of data into the warehouse, a previously created Analysis Services tabular model is refreshed. This semantic model simplifies the analysis of business data and relationships.
4. Business analysts use Microsoft Power BI to analyze warehoused data via the Analysis Services semantic model.

## Components

The company has data sources on many different platforms:

- SQL Server on-premises
- Oracle on-premises
- Azure SQL Database
- Azure table storage
- Cosmos DB

Data is loaded from these different data sources using several Azure components:

- [Blob storage](#) is used to stage source data before it's loaded into SQL Data Warehouse.
- [Data Factory](#) orchestrates the transformation of staged data into a common structure in SQL Data Warehouse. Data Factory [uses Polybase when loading data into SQL Data Warehouse](#) to maximize throughput.
- [SQL Data Warehouse](#) is a distributed system for storing and analyzing large datasets. Its use of massive parallel processing (MPP) makes it suitable for running high-performance analytics. SQL Data Warehouse can use [PolyBase](#) to rapidly load data from Blob storage.
- [Analysis Services](#) provides a semantic model for your data. It can also increase system performance when analyzing your data.
- [Power BI](#) is a suite of business analytics tools to analyze data and share insights. Power BI can query a semantic model stored in Analysis Services, or it can query SQL Data Warehouse directly.
- [Azure Active Directory \(Azure AD\)](#) authenticates users who connect to the Analysis Services server through Power BI. Data Factory can also use Azure AD to authenticate to SQL Data Warehouse via a service principal or [Managed identity for Azure resources](#).

## Alternatives

- The example pipeline includes several different kinds of data sources. This architecture can handle a wide variety of relational and non-relational data sources.
- Data Factory orchestrates the workflows for your data pipeline. If you want to load data only one time or on demand, you could use tools like SQL Server bulk copy (bcp) and AzCopy to copy data into Blob storage. You can then load the data directly into SQL Data Warehouse using Polybase.
- If you have very large datasets, consider using [Data Lake Storage](#), which provides limitless storage for analytics data.
- An on-premises [SQL Server Parallel Data Warehouse](#) appliance can also be used for big data processing. However, operating costs are often much lower with a managed cloud-based solution like SQL Data Warehouse.
- SQL Data Warehouse is not a good fit for OLTP workloads or data sets smaller than 250GB. For those cases you should use Azure SQL Database or SQL Server.
- For comparisons of other alternatives, see:
  - [Choosing a data pipeline orchestration technology in Azure](#)
  - [Choosing a batch processing technology in Azure](#)
  - [Choosing an analytical data store in Azure](#)
  - [Choosing a data analytics technology in Azure](#)

## Considerations

The technologies in this architecture were chosen because they met the company's requirements for scalability and

availability, while helping them control costs.

- The [massively parallel processing architecture](#) of SQL Data Warehouse provides scalability and high performance.
- SQL Data Warehouse has [guaranteed SLAs](#) and [recommended practices for achieving high availability](#).
- When analysis activity is low, the company can [scale SQL Data Warehouse on demand](#), reducing or even pausing compute to lower costs.
- Azure Analysis Services can be [scaled out](#) to reduce response times during high query workloads. You can also separate processing from the query pool, so that client queries aren't slowed down by processing operations.
- Azure Analysis Services also has [guaranteed SLAs](#) and [recommended practices for achieving high availability](#).
- The [SQL Data Warehouse security model](#) provides connection security, [authentication](#) and [authorization](#) via Azure AD or SQL Server authentication, and encryption. [Azure Analysis Services](#) uses Azure AD for identity management and user authentication.

## Pricing

Review a [pricing sample for a data warehousing scenario](#) via the Azure pricing calculator. Adjust the values to see how your requirements affect your costs.

- [SQL Data Warehouse](#) allows you to scale your compute and storage levels independently. Compute resources are charged per hour, and you can scale or pause these resources on demand. Storage resources are billed per terabyte, so your costs will increase as you ingest more data.
- [Data Factory](#) costs are based on the number of read/write operations, monitoring operations, and orchestration activities performed in a workload. Your Data Factory costs will increase with each additional data stream and the amount of data processed by each one.
- [Analysis Services](#) is available in developer, basic, and standard tiers. Instances are priced based on query processing units (QPUs) and available memory. To keep your costs lower, minimize the number of queries you run, how much data they process, and how often they run.
- [Power BI](#) has different product options for different requirements. [Power BI Embedded](#) provides an Azure-based option for embedding Power BI functionality inside your applications. A Power BI Embedded instance is included in the pricing sample above.

## Next Steps

- Review the [Azure reference architecture for automated enterprise BI](#), which includes instructions for deploying an instance of this architecture in Azure.
- Read the [Maritz Motivation Solutions customer story](#). That story describes a similar approach to managing customer data.
- Find comprehensive architectural guidance on data pipelines, data warehousing, online analytical processing (OLAP), and big data in the [Azure Data Architecture Guide](#).

# Ingestion and processing of real-time automotive IoT data

10/9/2018 • 7 minutes to read • [Edit Online](#)

This example scenario builds a real time data ingestion and processing pipeline to ingest and process messages from IoT devices (in general sensors) into a big data analytic platform in Azure. Vehicle telematics ingestion and processing platforms are the key to create connected car solutions. This specific scenario is motivated by the car telematics ingestion and processing systems. However, the design patterns are relevant for many industries using sensors to manage and monitor complex systems in industries such as smart buildings, communications, manufacturing, retail, and healthcare.

This example demonstrates a real time data ingestion and processing pipeline for messages from IoT devices installed in vehicles. Thousands and millions of messages (or events) are generated by the IoT devices and sensors. By capturing and analyzing these messages, we can decipher valuable insights and take appropriate actions. For example, with cars equipped telematics devices, if we can capture the device (IoT) messages in real time, we would be able to monitor the live location of vehicles, plan optimized routes, provide assistance to drivers, and support telematics-related industries such as auto insurance.

For this example demonstration, imagine a car manufacturing company that wants to create a real time system to ingest and process messages from telematics devices. The company's goals include:

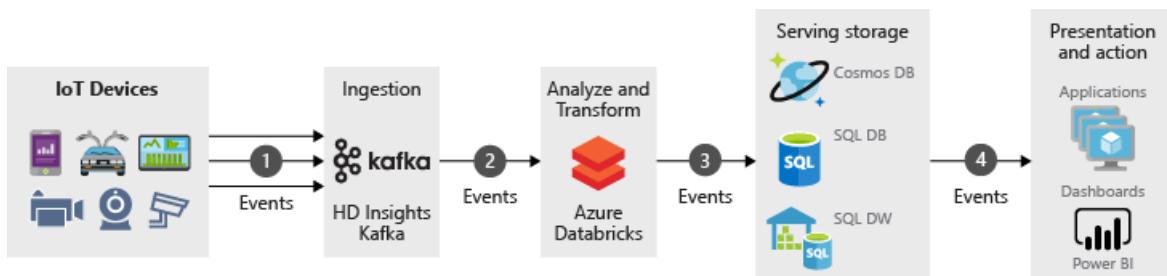
- Ingest and store data in real time from vehicles sensors and devices.
- Analyze the messages to understand vehicle location, and other information emitted through different types of sensors (such as engine-related sensors and environment-related sensors).
- Store the data after analysis for other downstream processing to provide actionable insights (For example, in accident scenarios, insurance agencies may be interested to know what happened during an accident etc.)

## Relevant use cases

Consider this scenario for the following use cases along with the above goals, when creating telematics ingestion and processing system:

- Vehicle maintenance reminders and alerting.
- Location-based services for the vehicle passengers (that is, SOS).
- Autonomous (self-driving) vehicles.

## Architecture



In a typical big data processing pipeline implementation, the data flows from left to right. In this real time big data processing pipeline, the data flows through the solution as follows:

1. Events generated from the IoT data sources are sent to the stream ingestion layer through Azure HDInsight

Kafka as a stream of messages. HDInsight Kafka stores streams of data in topics for a configurable of time.

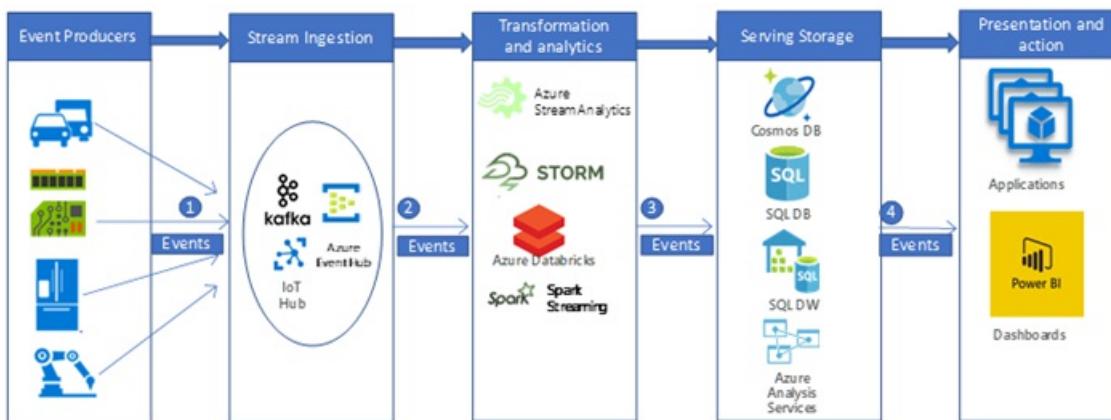
2. Kafka consumer, Azure Databricks, picks up the message in real time from the Kafka topic, to process the data based on the business logic and can then send to Serving layer for storage.
3. Downstream storage services, like Azure Cosmos DB, Azure SQL Data warehouse, or Azure SQL DB, will then be a data source for presentation and action layer.
4. Business analysts can use Microsoft Power BI to analyze warehoused data. Other applications can be built upon the serving layer as well. For example, we can expose APIs based on the service layer data for third party uses.

## Components

IoT device-generated events (data or messages) are ingested, processed, and then stored for further analysis, presentation, and action, using the following Azure components:

- [Apache Kafka on HDInsight](#) is in the ingestion layer. The data is written into the Kafka topic using a Kafka producer API.
- [Azure Databricks](#) is located in the transformation and analytics layer. Databricks notebooks implement a Kafka consumer API to read the data from the Kafka topic.
- [Azure Cosmos DB](#), [Azure SQL Database](#), and Azure SQL Data Warehouse are in the Serving storage layer, where Azure Databricks can write the data via data connectors.
- [Azure SQL Data Warehouse](#) is a distributed system for storing and analyzing large datasets. Its use of massive parallel processing (MPP) makes it suitable for running high-performance analytics.
- [Power BI](#) is a suite of business analytics tools to analyze data and share insights. Power BI can query a semantic model stored in Analysis Services, or it can query SQL Data Warehouse directly.
- [Azure Active Directory \(Azure AD\)](#) authenticates users, when connecting to [Azure Databricks](#). If we would build a cube in [Analysis Services](#) based on the model based on Azure SQL Data Warehouse data, we could use AAD to connect to the Analysis Services server through Power BI. Data Factory can also use Azure AD to authenticate to SQL Data Warehouse via a service principal or Managed Service Identity (MSI).
- [Azure App Services](#), in particular [API App](#) can be used to expose data to third parties, based on the data stored in the Serving Layer.

## Alternatives



A more generalized big data pipeline could be implemented using other Azure components.

- In the stream ingestion layer, we could use [IoT Hub](#) or [Event Hub](#), instead of [HDInsight Kafka](#) to ingest data.
- In the transformation and analytics layer, we could use [HDInsight Storm](#), [HDInsight Spark](#), or [Azure Stream Analytics](#).
- [Analysis Services](#) provides a semantic model for your data. It can also increase system performance when analyzing your data. You can build the model based on Azure DW data.

## Considerations

The technologies in this architecture were chosen based on the scale needed to process events, the SLA of the services, the cost management and ease of management of the components.

- Managed [HDInsight Kafka](#) comes with a 99.9% SLA is integrated with Azure Managed Disks
- [Azure Databricks](#) is optimized from the ground up for performance and cost-efficiency in the cloud. The Databricks Runtime adds several key capabilities to Apache Spark workloads that can increase performance and reduce costs by as much as 10-100x when running on Azure, including:
  - Auto-scaling and auto-termination for Spark clusters to automatically minimize costs.
  - Performance optimizations including caching, indexing, and advanced query optimization, which can improve performance by as much as 10-100x over traditional Apache Spark deployments in cloud or on-premises environments.
  - Integration with Azure Active Directory enables you to run complete Azure-based solutions using Azure Databricks.
  - Role-based access in Azure Databricks enables fine-grained user permissions for notebooks, clusters, jobs, and data.
  - Comes with Enterprise-grade SLAs.
- Azure Cosmos DB is Microsoft's globally distributed, multi-model database. Azure Cosmos DB was built from the ground up with global distribution and horizontal scale at its core. It offers turnkey global distribution across any number of Azure regions by transparently scaling and replicating your data wherever your users are. You can elastically scale throughput and storage worldwide, and pay only for the throughput and storage you need.
- The massively parallel processing architecture of SQL Data Warehouse provides scalability and high performance.
- Azure SQL Data Warehouse has guaranteed SLAs and recommended practices for achieving high availability.
- When analysis activity is low, the company can scale Azure SQL Data Warehouse on demand, reducing or even pausing compute to lower costs.
- The Azure SQL Data Warehouse security model provides connection security, authentication, and authorization via Azure AD or SQL Server authentication, and encryption.

## Pricing

Review [Azure Databricks pricing](#), [Azure HDInsight pricing](#), [pricing sample for a data warehousing scenario](#) via the Azure pricing calculator. Adjust the values to see how your requirements affect your costs.

- [Azure HDInsight](#) is a fully-managed cloud service that makes it easy, fast, and cost-effective to process massive amounts of data
- [Azure Databricks](#) offers two distinct workloads on several [VM Instances](#) tailored for your data analytics workflow—the Data Engineering workload makes it easy for data engineers to build and execute jobs, and the Data Analytics workload makes it easy for data scientists to explore, visualize, manipulate, and share data and insights interactively.
- [Azure 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—all backed by industry leading comprehensive [service level agreements](#) (SLAs).
- [Azure SQL Data Warehouse](#) allows you to scale your compute and storage levels independently. Compute resources are charged per hour, and you can scale or pause these resources on demand. Storage resources are billed per terabyte, so your costs will increase as you ingest more data.
- [Analysis Services](#) is available in developer, basic, and standard tiers. Instances are priced based on query processing units (QPs) and available memory. To keep your costs lower, minimize the number of queries you run, how much data they process, and how often they run.

- [Power BI](#) has different product options for different requirements. [Power BI Embedded](#) provides an Azure-based option for embedding Power BI functionality inside your applications. A Power BI Embedded instance is included in the pricing sample above.

## Next Steps

- Review the [Real-time analytics](#) reference architecture that includes big data pipeline flow.
- Review the [Advanced analytics on big data](#) reference architecture to get a peek on how different azure components can help build a big data pipeline.
- Read the [Real time processing](#) Azure documentation to get a quick view of how different Azure components help in processing streams of data in real time.
- Find comprehensive architectural guidance on data pipelines, data warehousing, online analytical processing (OLAP), and big data in the [Azure Data Architecture Guide](#).

# Real-time fraud detection on Azure

10/5/2018 • 4 minutes to read • [Edit Online](#)

This example scenario is relevant to organizations that need to analyze data in real time to detect fraudulent transactions or other anomalous activity.

Potential applications include identifying fraudulent credit card activity or mobile phone calls. Traditional online analytical systems might take hours to transform and analyze the data to identify anomalous activity.

By using fully managed Azure services such as Event Hubs and Stream Analytics, companies can eliminate the need to manage individual servers, while reducing costs and leveraging Microsoft's expertise in cloud-scale data ingestion and real-time analytics. This scenario specifically addresses the detection of fraudulent activity. If you have other needs for data analytics, you should review the list of available [Azure Analytics services](#).

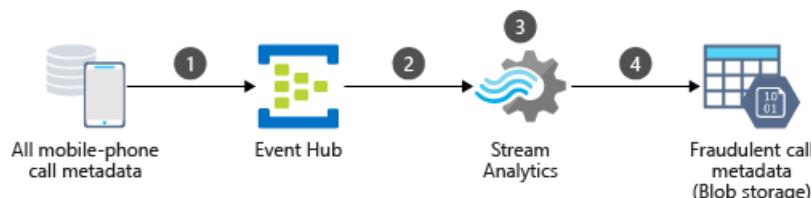
This sample represents one part of a broader data processing architecture and strategy. Other options for this aspect of an overall architecture are discussed later in this article.

## Relevant use cases

Consider this scenario for the following use cases:

- Detecting fraudulent mobile-phone calls in telecommunications scenarios.
- Identifying fraudulent credit card transactions for banking institutions.
- Identifying fraudulent purchases in retail or e-commerce scenarios.

## Architecture



This scenario covers the back-end components of a real-time analytics pipeline. Data flows through the scenario as follows:

1. Mobile phone call metadata is sent from the source system to an Azure Event Hubs instance.
2. A Stream Analytics job is started, which receives data via the event hub source.
3. The Stream Analytics job runs a predefined query to transform the input stream and analyze it based on a fraudulent-transaction algorithm. This query uses a tumbling window to segment the stream into distinct temporal units.
4. The Stream Analytics job writes the transformed stream representing detected fraudulent calls to an output sink in Azure Blob storage.

## Components

- [Azure Event Hubs](#) is a real-time 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. In this scenario, Event Hubs receives all phone call metadata to be analyzed for fraudulent activity.
- [Azure Stream Analytics](#) is an event-processing engine that can analyze high volumes of data streaming from devices and other data sources. It also supports extracting information from data streams to identify patterns

and relationships. These patterns can trigger other downstream actions. In this scenario, Stream Analytics transforms the input stream from Event Hubs to identify fraudulent calls.

- [Blob storage](#) is used in this scenario to store the results of the Stream Analytics job.

## Considerations

### Alternatives

Many technology choices are available for real-time message ingestion, data storage, stream processing, storage of analytical data, and analytics and reporting. For an overview of these options, their capabilities, and key selection criteria, see [Big data architectures: Real-time processing](#) in the Azure Data Architecture Guide.

Additionally, more complex algorithms for fraud detection can be produced by various machine learning services in Azure. For an overview of these options, see [Technology choices for machine learning](#) in the [Azure Data Architecture Guide](#).

### Availability

Azure Monitor provides unified user interfaces for monitoring across various Azure services. For more information, see [Monitoring in Microsoft Azure](#). Event Hubs and Stream Analytics are both integrated with Azure Monitor.

For other availability considerations, see the [availability checklist](#) in the Azure Architecture Center.

### Scalability

The components of this scenario are designed for hyper-scale ingestion and massively parallel real-time analytics. Azure Event Hubs is highly scalable, capable of receiving and processing millions of events per second with low latency. Event Hubs can [automatically scale up](#) the number of throughput units to meet usage needs. Azure Stream Analytics is capable of analyzing high volumes of streaming data from many sources. You can scale up Stream Analytics by increasing the number of [streaming units](#) allocated to execute your streaming job.

For general guidance on designing scalable scenario, see the [scalability checklist](#) in the Azure Architecture Center.

### Security

Azure Event Hubs secures data through an [authentication and security model](#) based on a combination of Shared Access Signature (SAS) tokens and event publishers. An event publisher defines a virtual endpoint for an event hub. The publisher can only be used to send messages to an event hub. It is not possible to receive messages from a publisher.

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

### Resiliency

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Deploy the scenario

To deploy this scenario, you can follow this [step-by-step tutorial](#) demonstrating how to manually deploy each component of the scenario. This tutorial also provides a .NET client application to generate sample phone call metadata and send that data to an event hub instance.

## Pricing

To explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected data volume.

We have provided three sample cost profiles based on amount of traffic you expect to get:

- **Small**: process one million events through one standard streaming unit per month.
- **Medium**: process 100M events through five standard streaming units per month.
- **Large**: process 999 million events through 20 standard streaming units per month.

## Related resources

More complex fraud detection scenarios can benefit from a machine learning model. For scenarios built using Machine Learning Server, see [Fraud detection using Machine Learning Server](#). For other solution templates using Machine Learning Server, see [Data science scenarios and solution templates](#). For an example solution using Azure Data Lake Analytics, see [Using Azure Data Lake and R for Fraud Detection](#).

# Scalable order processing on Azure

10/5/2018 • 6 minutes to read • [Edit Online](#)

This example scenario is relevant to organizations that need a highly scalable and resilient architecture for online order processing. Potential applications include e-commerce and retail point-of-sale, order fulfillment, and inventory reservation and tracking.

This scenario takes an event sourcing approach, using a functional programming model implemented via microservices. Each microservice is treated as a stream processor, and all business logic is implemented via microservices. This approach enables high availability and resiliency, geo-replication, and fast performance.

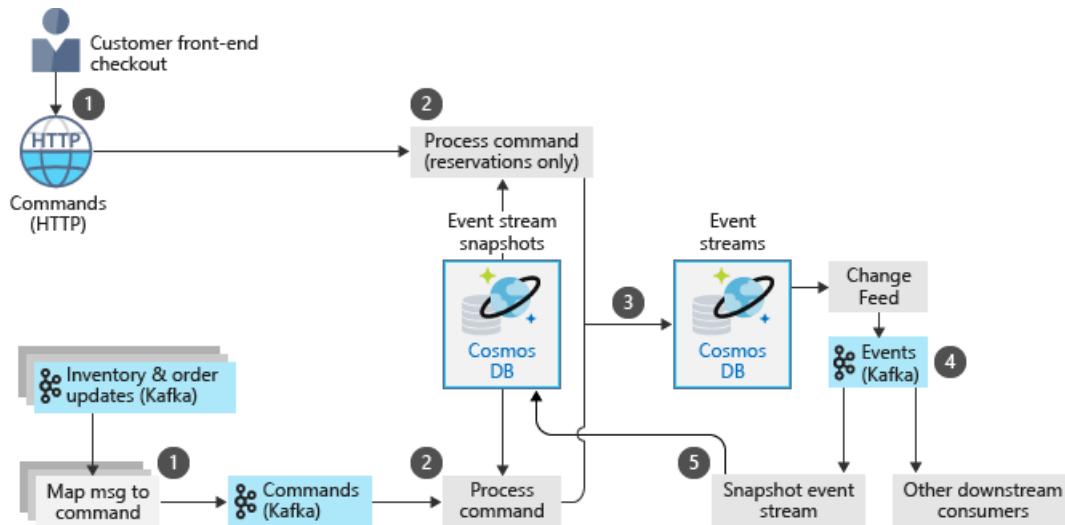
Using managed Azure services such as Cosmos DB and HDInsight can help reduce costs by leveraging Microsoft's expertise in globally distributed cloud-scale data storage and retrieval. This scenario specifically addresses an e-commerce or retail scenario; if you have other needs for data services, you should review the list of available [fully managed intelligent database services in Azure](#).

## Relevant use cases

Consider this scenario for the following use cases:

- E-commerce or retail point-of-sale back-end systems.
- Inventory management systems.
- Order fulfillment systems.
- Other integration scenarios relevant to an order processing pipeline.

## Architecture



This architecture details key components of an order processing pipeline. The data flows through the scenario as follows:

1. Event messages enter the system via customer-facing applications (synchronously over HTTP) and various back-end systems (asynchronously via Apache Kafka). These messages are passed into a command processing pipeline.
2. Each event message is ingested and mapped to one of a defined set of commands by a command processor microservice. The command processor retrieves any current state relevant to executing the command from an event stream snapshot database. The command is then executed, and the output of the command is emitted as

- a new event.
3. Each event emitted as the output of a command is committed to an event stream database using Cosmos DB.
  4. For each database insert or update committed to the event stream database, an event is raised by the Cosmos DB Change Feed. Downstream systems can subscribe to any event topics that are relevant to that system.
  5. All events from the Cosmos DB Change Feed are also sent to a snapshot event stream microservice, which calculates any state changes caused by events that have occurred. The new state is then committed to the event stream snapshot database stored in Cosmos DB. The snapshot database provides a globally distributed, low latency data source for the current state of all data elements. The event stream database provides a complete record of all event messages that have passed through the architecture, which enables robust testing, troubleshooting, and disaster recovery scenarios.

## Components

- [Cosmos DB](#) is Microsoft's globally distributed, multi-model database that enables your solutions to elastically and independently scale throughput and storage across any number of geographic regions. It offers throughput, latency, availability, and consistency guarantees with comprehensive service level agreements (SLAs). This scenario uses Cosmos DB for event stream storage and snapshot storage, and leverages [Cosmos DB's Change Feed](#) features to provide data consistency and fault recovery.
- [Apache Kafka on HDInsight](#) is a managed service implementation of Apache Kafka, an open-source distributed streaming platform for building real-time streaming data pipelines and applications. Kafka also provides message broker functionality similar to a message queue, for publishing and subscribing to named data streams. This scenario uses Kafka to process incoming as well as downstream events in the order processing pipeline.

## Considerations

Many technology options are available for real-time message ingestion, data storage, stream processing, storage of analytical data, and analytics and reporting. For an overview of these options, their capabilities, and key selection criteria, see [Big data architectures: Real-time processing](#) in the [Azure Data Architecture Guide](#).

Microservices have become a popular architectural style for building cloud applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. Microservices require a different approach to designing and building applications. Tools such as Docker, Kubernetes, Azure Service Fabric, and Nomad enable the development of microservices-based architectures. For guidance on building and running a microservices-based architecture, see [Designing microservices on Azure](/azure/architecture/microservices) in the Azure Architecture Center.

## Availability

This scenario's event sourcing approach allows system components to be loosely coupled and deployed independently of one another. Cosmos DB offers [high availability](#) and helps organization manage the tradeoffs associated with consistency, availability, and performance, all with [corresponding guarantees](#). Apache Kafka on HDInsight is also designed for [high availability](#).

Azure Monitor provides unified user interfaces for monitoring across various Azure services. For more information, see [Monitoring in Microsoft Azure](#). Event Hubs and Stream Analytics are both integrated with Azure Monitor.

For other availability considerations, see the [availability checklist](#).

## Scalability

Kafka on HDInsight allows [configuration of storage and scalability](#) for Kafka clusters. Cosmos DB provides fast, predictable performance and [scales seamlessly](#) as your application grows. The event sourcing microservices-based architecture of this scenario also makes it easier to scale your system and expand its functionality.

For other scalability considerations, see the [scalability checklist](#) available in the Azure Architecture Center.

## Security

The [Cosmos DB security model](#) authenticates users and provides access to its data and resources. For more information, see [Cosmos DB database security](#).

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

## Resiliency

The event sourcing architecture and associated technologies in this example scenario make this scenario highly resilient when failures occur. For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Pricing

To examine the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how pricing would change for your particular scenario, change the appropriate variables to match your expected data volume. For this scenario, the example pricing includes only Cosmos DB and a Kafka cluster for processing events raised from the Cosmos DB Change Feed. Event processors and microservices for originating systems and other downstream systems are not included, and their cost is highly dependent on the quantity and scale of these services as well as the technologies chosen for implementing them.

The currency of Azure Cosmos DB is the request unit (RU). With request units, you don't need to reserve read/write capacities or provision CPU, memory, and IOPS. Azure Cosmos DB supports various APIs that have different operations, ranging from simple reads and writes to complex graph queries. Because not all requests are equal, requests are assigned a normalized quantity of request units based on the amount of computation required to serve the request. The number of request units required by your solution is dependent on data element size and the number of database read and write operations per second. For more information, see [Request units in Azure Cosmos DB](#). These estimated prices are based on Cosmos DB running in two Azure regions.

We have provided three sample cost profiles based on amount of activity you expect:

- **Small**: this pricing example correlates to 5 RUs reserved with a 1 TB data store in Cosmos DB and a small (D3 v2) Kafka cluster.
- **Medium**: this pricing example correlates to 50 RUs reserved with a 10 TB data store in Cosmos DB and a midsized (D4 v2) Kafka cluster.
- **Large**: this pricing example correlates to 500 RUs reserved with a 30 TB data store in Cosmos DB and a large (D5 v2) Kafka cluster.

## Related resources

This example scenario is based on a more extensive version of this architecture built by [Jet.com](#) for its end-to-end order processing pipeline. For more information, see the [jet.com technical customer profile](#) and [jet.com's presentation at Build 2018](#).

Other related resources include:

- [Designing Data-Intensive Applications](#) by Martin Kleppmann (O'Reilly Media, 2017).
- [Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#](#) by Scott Wlaschin (Pragmatic Programmers LLC, 2018).
- Other [Cosmos DB use cases](#)
- [Real time processing architecture](#) in the [Azure Data Architecture Guide](#).

# Running computational fluid dynamics (CFD) simulations on Azure

10/5/2018 • 5 minutes to read • [Edit Online](#)

Computational Fluid Dynamics (CFD) simulations require significant compute time along with specialized hardware. As cluster usage increases, simulation times and overall grid use grow, leading to issues with spare capacity and long queue times. Adding physical hardware can be expensive, and may not align to the usage peaks and valleys that a business goes through. By taking advantage of Azure, many of these challenges can be overcome with no capital expenditure.

Azure provides the hardware you need to run your CFD jobs on both GPU and CPU virtual machines. RDMA (Remote Direct Memory Access) enabled VM sizes have FDR InfiniBand-based networking which allows for low latency MPI (Message Passing Interface) communication. Combined with the Avere vFXT, which provides an enterprise-scale clustered file system, customers can ensure maximum throughput for read operations in Azure.

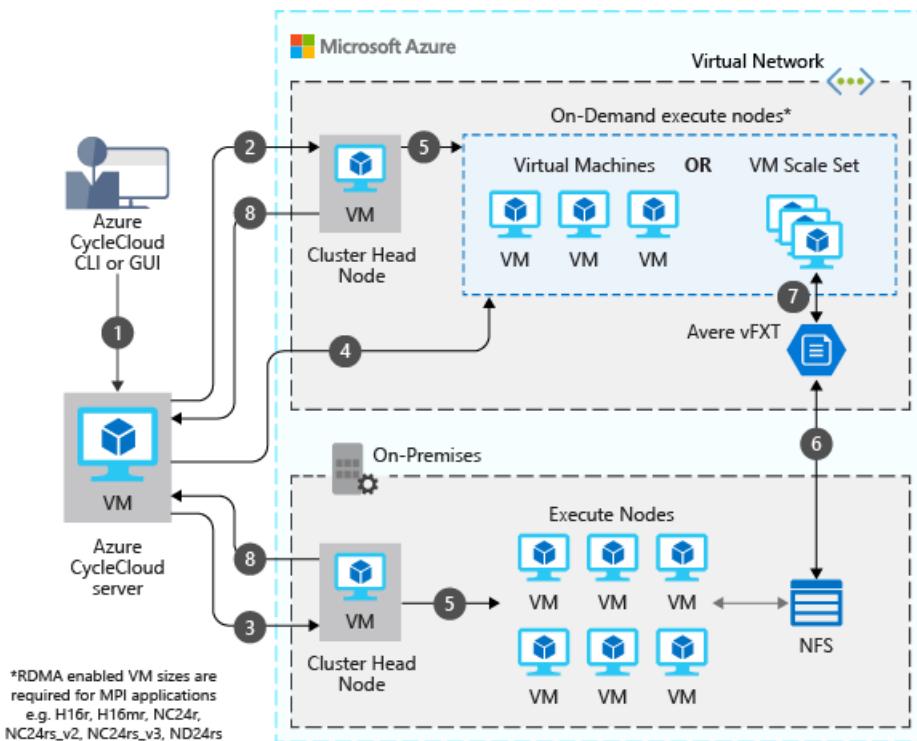
To simplify the creation, management, and optimization of HPC clusters, Azure CycleCloud can be used to provision clusters and orchestrate data in both hybrid and cloud scenarios. By monitoring the pending jobs, CycleCloud will automatically launch on-demand compute, where you only pay for what you use, connected to the workload scheduler of your choice.

## Relevant use cases

Consider this scenario for these industries where CFD applications could be used:

- Aeronautics
- Automotive
- Building HVAC
- Oil and gas
- Life sciences

## Architecture



This diagram shows a high-level overview of a typical hybrid design providing job monitoring of the on-demand nodes in Azure:

1. Connect to the Azure CycleCloud server to configure the cluster.
2. Configure and create the cluster head node, using RDMA enabled machines for MPI.
3. Add and configure the on-premises head node.
4. If there are insufficient resources, Azure CycleCloud will scale up (or down) compute resources in Azure. A predetermined limit can be defined to prevent over allocation.
5. Tasks allocated to the execute nodes.
6. Data cached in Azure from on-premises NFS server.
7. Data read in from the Avere vFXT for Azure cache.
8. Job and task information relayed to the Azure CycleCloud server.

## Components

- [Azure CycleCloud](#) a tool for creating, managing, operating, and optimizing HPC and Big Compute clusters in Azure.
- [Avere vFXT on Azure](#) is used to provide an enterprise-scale clustered file system built for the cloud.
- [Azure Virtual Machines \(VMs\)](#) are used to create a static set of compute instances.
- [Virtual Machine Scale Sets \(virtual machine scale set\)](#) provide a group of identical VMs capable of being scaled up or down by Azure CycleCloud.
- [Azure Storage accounts](#) are used for synchronization and data retention.
- [Virtual Networks](#) enable many types of Azure resources, such as Azure Virtual Machines (VMs), to securely communicate with each other, the internet, and on-premises networks.

## Alternatives

Customers can also use Azure CycleCloud to create a grid entirely in Azure. In this setup, the Azure CycleCloud server is run within your Azure subscription.

For a modern application approach where management of a workload scheduler is not needed, [Azure Batch](#) can help. Azure Batch can run large-scale parallel and high-performance computing (HPC) applications efficiently in the cloud. Azure Batch allows you to define the Azure compute resources to execute your applications in parallel or at scale without manually configuring or managing infrastructure. Azure Batch schedules compute-intensive tasks and dynamically adds and removes compute resources based on your requirements.

## Scalability, and Security

Scaling the execute nodes on Azure CycleCloud can be accomplished either manually or using autoscaling. For more information, see [CycleCloud Autoscaling](#).

For general guidance on designing secure solutions, see the [Azure security documentation](#).

## Deploy this scenario

Before deploying in Azure, some pre-requisites are required. Follow these steps before deploying the Resource Manager template:

1. Create a [service principal](#) for retrieving the appId, displayName, name, password, and tenant.
2. Generate an [SSH key pair](#) to sign in securely to the CycleCloud server.



3. [Log into the CycleCloud server](#) to configure and create a new cluster.
4. [Create a cluster](#).

The Avere Cache is an optional solution that can drastically increase read throughput for the application job data. Avere vFXT for Azure solves the problem of running these enterprise HPC applications in the cloud while leveraging data stored on-premises or in Azure Blob storage.

For organizations that are planning for a hybrid infrastructure with both on-premises storage and cloud computing, HPC applications can “burst” into Azure using data stored in NAS devices and spin up virtual CPUs as needed. The data set is never moved completely into the cloud. The requested bytes are temporarily cached using an Avere cluster during processing.

To set up and configure an Avere vFXT installation, follow the [Avere Setup and Configuration guide](#).

## Pricing

The cost of running an HPC implementation using CycleCloud server will vary depending on a number of factors. For example, CycleCloud is charged by the amount of compute time that is used, with the Master and CycleCloud server typically being constantly allocated and running. The cost of running the Execute nodes will depend on how long these are up and running as well as what size is used. The normal Azure charges for storage and networking also apply.

This scenario shows how CFD applications can be run in Azure, so the machines will require RDMA functionality, which is only available on specific VM sizes. The following are examples of costs that could be incurred for a scale set that is allocated continuously for eight hours per day for one month, with data egress of 1 TB. It also includes pricing for the Azure CycleCloud server and the Avere vFXT for Azure install:

- Region: North Europe
- Azure CycleCloud Server: 1 x Standard D3 (4 x CPUs, 14 GB Memory, Standard HDD 32 GB)
- Azure CycleCloud Master Server: 1 x Standard D12 v (4 x CPUs, 28 GB Memory, Standard HDD 32 GB)
- Azure CycleCloud Node Array: 10 x Standard H16r (16 x CPUs, 112 GB Memory)
- Avere vFXT on Azure Cluster: 3 x D16s v3 (200 GB OS, Premium SSD 1-TB data disk)
- Data Egress: 1 TB

Review this [price estimate](#) for the hardware listed above.

## Next Steps

Once you've deployed the sample, learn more about [Azure CycleCloud](#).

## Related resources

- [RDMA Capable Machine Instances](#)
- [Customizing an RDMA Instance VM](#)

# Linux Virtual Desktops with Citrix

10/5/2018 • 5 minutes to read • [Edit Online](#)

This example scenario is applicable to any industry that needs a Virtual Desktop Infrastructure (VDI) for Linux Desktops. VDI refers to the process of running a user desktop inside a virtual machine that lives on a server in the datacenter. The customer in this scenario chose to use a Citrix-based solution for their VDI needs.

Organizations often have heterogeneous environments with multiple devices and operating systems being used by employees. It can be challenging to provide consistent access to applications while maintaining a secure environment. A VDI solution for Linux desktops will allow your organization to provide access irrespective of the device or OS used by the end user.

Some benefits for this sample solution include the following:

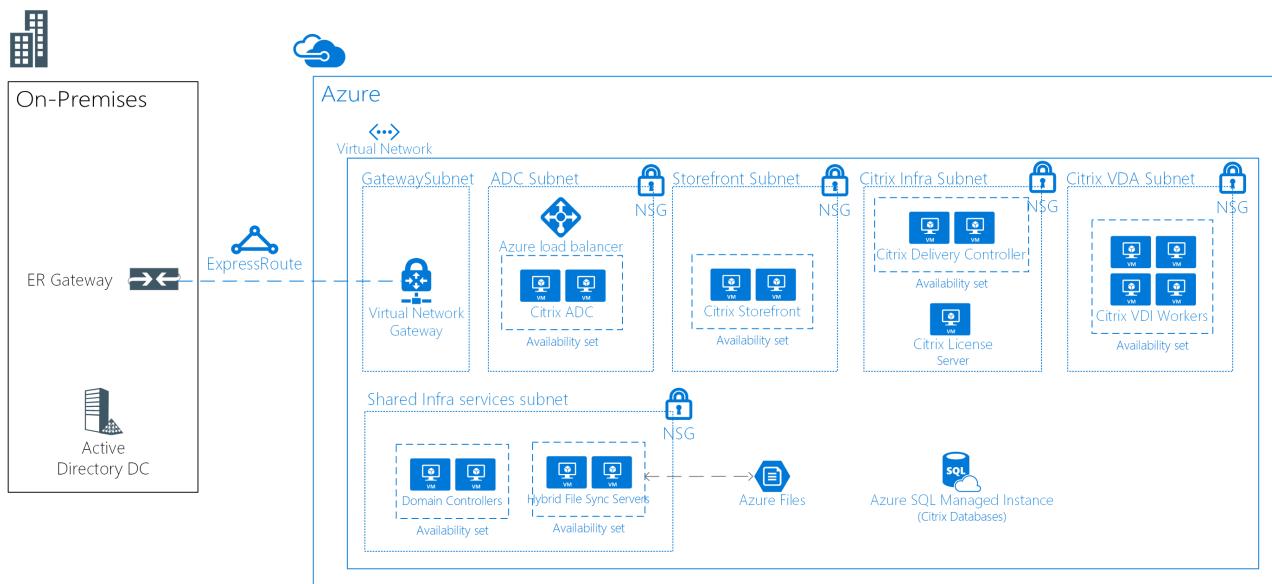
- Return on investment will be higher with shared Linux virtual desktops by giving more users access to the same infrastructure. By consolidating resources on a centralized VDI environment, the end user devices don't need to be as powerful.
- Performance will be consistent regardless of the end user device.
- Users can access Linux applications from any device (including non-Linux devices).
- Sensitive data can be secured in the Azure data center for all distributed employees.

## Relevant use cases

Consider this scenario for the following use case:

- Providing secure access to mission-critical, specialized Linux VDI desktops from Linux or non-Linux devices

## Architecture



This example scenario demonstrates allowing the corporate network to access the Linux Virtual Desktops:

- An ExpressRoute is established between the on-premises environment and Azure, for fast and reliable connectivity to the cloud.
- Citrix XenDesktop solution deployed for VDI.
- The CitrixVDA run on Ubuntu (or another supported distro).

- Azure Network Security Groups will apply the correct network ACLs.
- Citrix ADC (NetScaler) will publish and load balance all the Citrix services.
- Active Directory Domain Services will be used to domain join the Citrix servers. VDA servers will not be domain joined.
- Azure Hybrid File Sync will enable shared storage across the solution. For example, it can be used in remote/home solutions.

For this scenario, the following SKUs are used:

- Citrix ADC (NetScaler): 2 x D4sv3 with [NetScaler 12.0 VPX Standard Edition 200 MBPS PAYG image](#)
- Citrix License Server: 1 x D2s v3
- Citrix VDA: 4 x D8s v3
- Citrix Storefront: 2 x D2s v3
- Citrix Delivery Controller: 2 x D2s v3
- Domain Controllers: 2 x D2sv3
- Azure File Servers: 2 x D2sv3

#### **NOTE**

All the licenses (other than NetScaler) are bring-your-own-license (BYOL)

## **Components**

- [Azure Virtual Network](#) allows resources such as VMs to securely communicate with each other, the internet, and on-premises networks. Virtual networks provide isolation and segmentation, filter and route traffic, and allow connection between locations. One virtual network will be used for all resources in this scenario.
- [Azure network security groups](#) contain a list of security rules that allow or deny inbound or outbound network traffic based on source or destination IP address, port, and protocol. The virtual networks in this scenario are secured with network security group rules that restrict the flow of traffic between the application components.
- [Azure load balancer](#) distributes inbound traffic according to rules and health probes. A load balancer provides low latency and high throughput, and scales up to millions of flows for all TCP and UDP applications. An internal load balancer is used in this scenario to distribute traffic on the Citrix NetScaler.
- [Azure Hybrid File Sync](#) will be used for all shared storage. The storage will replicate to two file servers using Hybrid File Sync.
- [Azure SQL Database](#) is a relational database-as-a-service (DBaaS) based on the latest stable version of Microsoft SQL Server Database Engine. It will be used for hosting Citrix databases.
- [ExpressRoute](#) lets you extend your on-premises networks into the Microsoft cloud over a private connection facilitated by a connectivity provider.
- [Active Directory Domain Services is used for Directory Services and user authentication]
- [Azure Availability Sets](#) will ensure that the VMs you deploy on Azure are distributed across multiple isolated hardware nodes in a cluster. Doing this ensures that if a hardware or software failure within Azure happens, only a subset of your VMs are impacted and that your overall solution remains available and operational.
- [Citrix ADC \(NetScaler\)](#) is an application delivery controller that performs application-specific traffic analysis to intelligently distribute, optimize, and secure Layer 4-Layer 7 (L4-L7) network traffic for web applications.
- [Citrix Storefront](#) is an enterprise app store that improves security and simplifies deployments, delivering a modern, unmatched near-native user experience across Citrix Receiver on any platform. StoreFront makes it easy to manage multi-site and multi-version Citrix Virtual Apps and Desktops environments.
- [Citrix License Server](#) will manage the licenses for Citrix products.
- [Citrix XenDesktops VDA](#) enables connections to applications and desktops. The VDA is installed on the machine that runs the applications or virtual desktops for the user. It enables the machines to register with Delivery Controllers and manage the High Definition eXperience (HDX) connection to a user device.

- [Citrix Delivery Controller](#) is the server-side component responsible for managing user access, plus brokering and optimizing connections. Controllers also provide the Machine Creation Services that create desktop and server images.

## Alternatives

- There are multiple partners with VDI solutions that supported in Azure such as VMware, Workspot, and others. This specific sample architecture is based on a deployed project that used Citrix.
- Citrix provides a cloud service that abstracts part of this architecture. It could be an alternative for this solution. For more information, see [Citrix Cloud](#).

## Considerations

- Check the [Citrix Linux Requirements](#).
- Latency can have impact on the overall solution. For a production environment, test accordingly.
- Depending on the scenario, the solution may need VMs with GPUs for VDA. For this solution, it is assumed that GPU is not a requirement.

## Availability, Scalability, and Security

- This sample solution is designed for high availability for all roles other than the licensing server. Because the environment continues to function during a 30-day grace period if the license server is offline, no additional redundancy is required on that server.
- All servers providing similar roles should be deployed in [Availability Sets](#).
- This sample solution does not include Disaster Recovery capabilities. [Azure Site Recovery](#) could be a good add-on to this design.
- For a production deployment management solution should be implemented such as [backup](#), [monitoring](#) and [update management](#).
- This sample solution should work for about 250 concurrent (about 50-60 per VDA server) users with a mixed usage. But that will greatly depended on the type of applications being used. For production use, rigorous load testing should be performed.

## Deploy this scenario

For deployment information see the official [Citrix documentation](#).

## Pricing

- The Citrix XenDesktop licenses are not included in Azure service charges.
- The Citrix NetScaler license is included in a pay-as-you-go model.
- Using reserved instances will greatly reduce the compute cost for the solution.
- The ExpressRoute cost is not included.

## Next Steps

- Check Citrix documentation for planning and deployment [here](#).
- To deploy Citrix ADC (NetScaler) in Azure, review the Resource Manager templates provided by Citrix [here](#).

# Using Service Fabric to decompose monolithic applications

10/5/2018 • 6 minutes to read • [Edit Online](#)

In this example scenario, we walk through an approach using [Service Fabric](#) as a platform for decomposing an unwieldy monolithic application. Here we consider an iterative approach to decomposing an IIS/ASP.NET web site into an application composed of multiple, manageable microservices.

Moving from a monolithic architecture to a microservice architecture provides the following benefits:

- You can change one small, understandable unit of code and deploy only that unit.
- Each code unit requires just a few minutes or less to deploy.
- If there is an error in that small unit, only that unit stops working, not the whole application.
- Small units of code can be distributed easily and discretely among multiple development teams.
- New developers can quickly and easily grasp the discrete functionality of each unit.

A large IIS application on a server farm is used in this example, but the concepts of iterative decomposition and hosting can be used for any type of large application. While this solution uses Windows, Service Fabric can also run on Linux. It can be run on-premises, in Azure, or on VM nodes in the cloud provider of your choice.

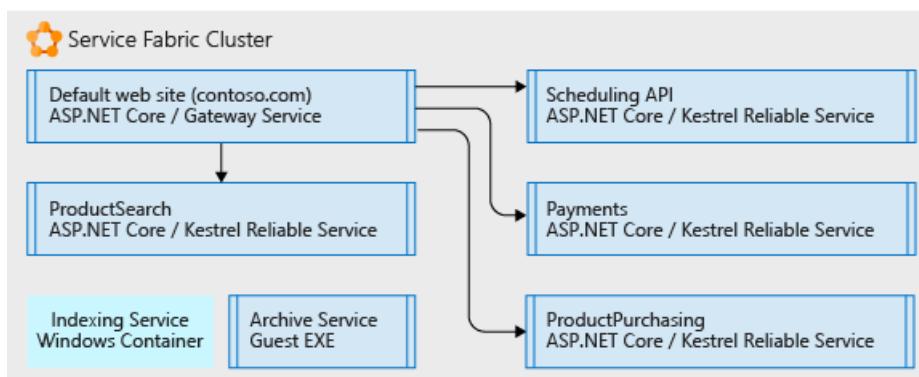
## Relevant use cases

This scenario is relevant to organizations with large monolithic Web applications that are experiencing:

- Errors in small code changes that break the entire website.
- Releases taking multiple days due to the need to release update the entire website.
- Long ramp-up times when onboarding new developers or teams due to the complex code base, requiring a single individual to know more than is feasible.

## Architecture

Using Service Fabric as the hosting platform, we can convert a large IIS web site into a collection of microservices as shown below:



In the picture above, we decomposed all the parts of a large IIS application into:

- A routing or gateway service that accepts incoming browser requests, parses them to determine what service should handle them, and forwards the request to that service.
- Four ASP.NET Core applications that were formally virtual directories under the single IIS site running as ASP.NET applications. The applications were separated into their own independent microservices. The effect is

that they can be changed, versioned, and upgraded separately. In this example, we rewrote each application using .Net Core and ASP.NET Core. These were written as [Reliable Services](#) so they can natively access the full Service Fabric platform capabilities and benefits (communication services, health reports, notifications, etc.).

- A Windows service called *Indexing Service*, placed in a Windows container so that it no longer makes direct changes to registry of the underlying server, but can run self-contained and be deployed with all its dependencies as a single unit.
- An Archive service, which is just an executable that runs according to a schedule and performs some tasks for the sites. It is hosted directly as a stand-alone executable because we determined it does what it needs to do without modification and it is not worth the investment to change.

## Considerations

The first challenge is to begin to identify smaller bits of code that can be factored out from the monolith into microservices that the monolith can call. Iteratively over time, the monolith is broken up into a collection of these microservices that developers can easily understand, change, and quickly deploy at low risk.

Service Fabric was chosen because it is capable of supporting running all the microservices in their various forms. For example you may have a mix of stand-alone executables, new small web sites, new small APIs, and containerized services, etc. Service Fabric can combine all these service types onto a single cluster.

To get to this final, decomposed application, we used an iterative approach. We started with a large IIS/ASP.NET web site on a server farm. A single node of the server farm is pictured below. It contains the original web site with several virtual directories, an additional Windows Service the site calls, and an executable that does some periodic site archive maintenance.



On the first development iteration, the IIS site and its virtual directories placed in a [Windows Container](#). Doing this allows the site to remain operational, but not tightly bound to the underlying server node OS. The container is run and orchestrated by the underlying Service Fabric node, but the node does not have to have any state that the site is dependent on (registry entries, files, etc.). All of those items are in the container. We have also placed the Indexing service in a Windows Container for the same reasons. The containers can be deployed, versioned, and scaled independently. Finally, we hosted the Archive Service a simple [stand-alone executable file](#) since it is a self-contained .exe with no special requirements.

The picture below shows how our large web site is now partially decomposed into independent units and ready to be decomposed more as time allows.



Further development focuses on separating the single large Default Web site container pictured above. Each of the virtual directory ASP.NET apps is removed from the container one at a time and ported to ASP.NET Core [reliable services](#).

Once each of the virtual directories has been factored out, the Default Web site is written as an ASP.NET Core reliable service, which accepts incoming browser requests and routes them to the correct ASP.NET application.

## Availability, Scalability, and Security

Service Fabric is [capable of supporting various forms of microservices](#) while keeping calls between them on the same cluster fast and simple. Service Fabric is a [fault tolerant](#), self-healing cluster that can run containers, executables, and even has a native API for writing microservices directly to it (the 'Reliable Services' referred to above). The platform facilitates rolling upgrades and versioning of each microservice. You can tell the platform to run more or fewer of any given microservice distributed across the Service Fabric cluster in order to [scale](#) in or out only the microservices you need.

Service Fabric is a cluster built on an infrastructure of virtual (or physical) nodes, which have networking, storage, and an operating system. As such, it has a set of administrative, maintenance, and monitoring tasks.

You'll also want to consider governance and control of the cluster. Just as you would not want people arbitrarily deploying databases to your production database server, neither would you want people deploying applications to the Service Fabric cluster without some oversight.

Service Fabric is capable of hosting many different [application scenarios](#), take some time to see which ones apply to your scenario.

## Pricing

For a Service Fabric cluster hosted in Azure, the largest part of the cost is the number and size of the nodes in your cluster. Azure allows quick and simple creation of a cluster composed of the underlying node size you specify, but the compute charges are based on the node size multiplied by the number of nodes.

Other less costly components of cost are the storage charges for each node's virtual disks and network IO egress charges from Azure (for example network traffic out of Azure to a user's browser).

To get an idea of cost, we have created an example using some default values for cluster size, networking, and storage: Take a look at the [pricing calculator](#). Feel free to update the values in this default calculator to those relevant to your situation.

## Next Steps

Take some time to familiarize yourself with the platform by going through the [documentation](#) and reviewing the many different [application scenarios](#) for Service Fabric. The documentation will tell you what a cluster consists of, what it can run on, software architecture, and maintenance for it.

To see a demonstration of Service Fabric for an existing .NET application, deploy the Service Fabric [quickstart](#).

From the standpoint of your current application, begin to think about its different functions. Choose one of them and think through how you can separate only that function from the whole. Take it one discrete, understandable, piece at a time.

## Related resources

- [Building Microservices on Azure](#)
- [Service Fabric Overview](#)
- [Service Fabric Programming Model](#)
- [Service Fabric Availability](#)
- [Scaling Service Fabric](#)
- [Hosting Containers in Service Fabric](#)
- [Hosting Stand-Alone Executables in Service Fabric](#)

- Service Fabric Native Reliable Services
- Service Fabric Application Scenarios

# Highly scalable and secure WordPress website

10/5/2018 • 5 minutes to read • [Edit Online](#)

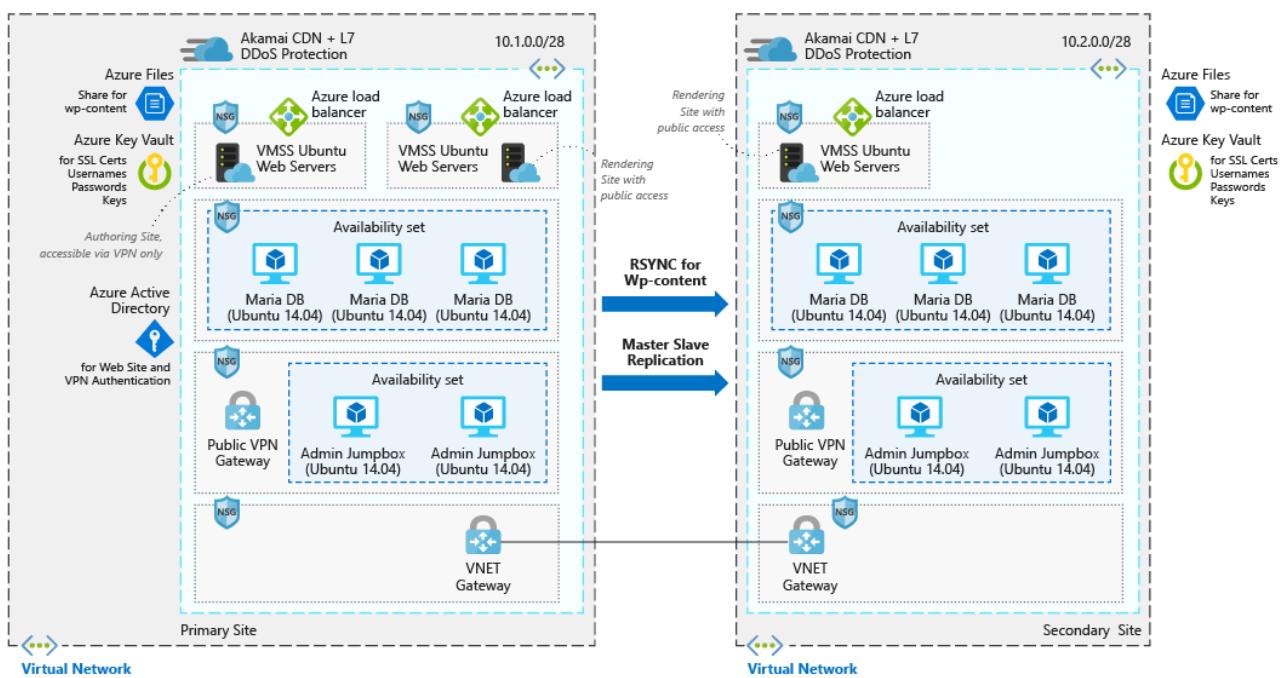
This example scenario is applicable to companies that need a highly scalable and secure installation of WordPress. This scenario is based on a deployment that was used for a large convention and was successfully able to scale to meet the spike traffic that sessions drove to the site.

## Relevant use cases

Consider this scenario for the following use cases:

- Media events that cause traffic surges.
- Blogs that use WordPress as their content management system.
- Business or e-commerce websites that use WordPress.
- Web sites built using other content management systems.

## Architecture



This scenario covers a scalable and secure installation of WordPress that uses Ubuntu web servers and MariaDB. There are two distinct data flows in this scenario the first is users access the website:

1. Users access the front-end website through a CDN.
2. The CDN uses an Azure load balancer as the origin, and pulls any data that isn't cached from there.
3. The Azure load balancer distributes requests to the virtual machine scale sets of web servers.
4. The WordPress application pulls any dynamic information out of the Maria DB clusters, all static content is hosted in Azure Files.
5. SSL keys are stored Azure Key Vault.

The second workflow is how authors contribute new content:

1. Authors connect securely to the public VPN gateway.
2. VPN authentication information is stored in Azure Active Directory.

3. A connection is then established to the Admin jump boxes.
4. From the admin jump box, the author is then able to connect to the Azure load balancer for the authoring cluster.
5. The Azure load balancer distributes traffic to the virtual machine scale sets of web servers that have write access to the Maria DB cluster.
6. New static content is uploaded to Azure files and dynamic content is written into the Maria DB cluster.
7. These changes are then replicated to the alternate region via rsync or master/slave replication.

## Components

- [Azure Content Delivery Network \(CDN\)](#) is a distributed network of servers that efficiently delivers web content to users. CDNs minimize latency by storing cached content on edge servers in point-of-presence locations near to end users.
- [Virtual networks](#) allow resources such as VMs to securely communicate with each other, the Internet, and on-premises networks. Virtual networks provide isolation and segmentation, filter and route traffic, and allow connection between locations. The two networks are connected via Vnet peering.
- [Network security groups](#) contain a list of security rules that allow or deny inbound or outbound network traffic based on source or destination IP address, port, and protocol. The virtual networks in this scenario are secured with network security group rules that restrict the flow of traffic between the application components.
- [Load balancers](#) distribute inbound traffic according to rules and health probes. A load balancer provides low latency and high throughput, and scales up to millions of flows for all TCP and UDP applications. A load balancer is used in this scenario to distribute traffic from the content deliver network to the front-end web servers.
- [Virtual machine scale sets](#) let you create and manage a group of identical load-balanced VMs. The number of VM instances can automatically increase or decrease in response to demand or a defined schedule. Two separate virtual machine scale sets are used in this scenario - one for the front-end web-servers serving content, and one for the front-end webservers used to author new content.
- [Azure Files](#) provides a fully-managed file share in the cloud that hosts all of the WordPress content in this scenario, so that all of the VMs have access to the data.
- [Azure Key Vault](#) is used to store and tightly control access to passwords, certificates, and keys.
- [Azure Active Directory \(Azure AD\)](#) is a multi-tenant, cloud-based directory and identity management service. In this scenario, Azure AD provides authentication services for the website and the VPN tunnels.

## Alternatives

- [SQL Server for Linux](#) can replace the MariaDB data store.
- [Azure database for MySQL](#) can replace the MariaDB data store if you prefer a fully managed solution.

## Considerations

### Availability

The VM instances in this scenario are deployed across multiple regions, with the data replicated between the two via RSYNC for the WordPress content and master slave replication for the MariaDB clusters.

For other availability topics, see the [availability checklist](#) in the Azure Architecture Center.

### Scalability

This scenario uses virtual machine scale sets for the two front-end web server clusters in each region. With scale sets, the number of VM instances that run the front-end application tier can automatically scale in response to customer demand, or based on a defined schedule. For more information, see [Overview of autoscale with virtual machine scale sets](#).

The back end is a MariaDB cluster in an availability set. For more information, see the [MariaDB cluster tutorial](#).

For other scalability topics, see the [scalability checklist](#) in the Azure Architecture Center.

## Security

All the virtual network traffic into the front-end application tier is protected by network security groups. Rules limit the flow of traffic so that only the front-end application tier VM instances can access the back-end database tier. No outbound Internet traffic is allowed from the database tier. To reduce the attack footprint, no direct remote management ports are open. For more information, see [Azure network security groups](#).

For general guidance on designing secure scenarios, see the [Azure Security Documentation](#).

## Resiliency

In combination with the use of multiple regions, data replication and virtual machine scale sets, this scenario uses Azure load balancers. These networking components distribute traffic to the connected VM instances, and include health probes that ensure traffic is only distributed to healthy VMs. All of these networking components are fronted via a CDN. This makes the networking resources and application resilient to issues that would otherwise disrupt traffic and impact end-user access.

For general guidance on designing resilient scenarios, see [Designing resilient applications for Azure](#).

## Pricing

To explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic.

We have provided a pre-configured [cost profile](#) based on the architecture diagram provided above. To configure the pricing calculator for your use case, there are a couple main things to consider:

- How much traffic are you expecting in terms of GB/month? The amount of traffic will have the biggest impact on your cost, as it will impact the number of VMs that are required to surface the data in the virtual machine scale set. Additionally, it will directly correlate with the amount of data that is surfaced via the CDN.
- How much new data are you going to be writing to your website? New data written to your website correlates with how much data is mirrored across the regions.
- How much of your content is dynamic? How much is static? The variance around dynamic and static content influences how much data has to be retrieved from the database tier versus how much will be cached in the CDN.

# Secure Windows web application for regulated industries

10/5/2018 • 7 minutes to read • [Edit Online](#)

This example scenario is applicable to regulated industries that have a need to secure multi-tier applications. In this scenario, a front-end ASP.NET application securely connects to a protected back-end Microsoft SQL Server cluster.

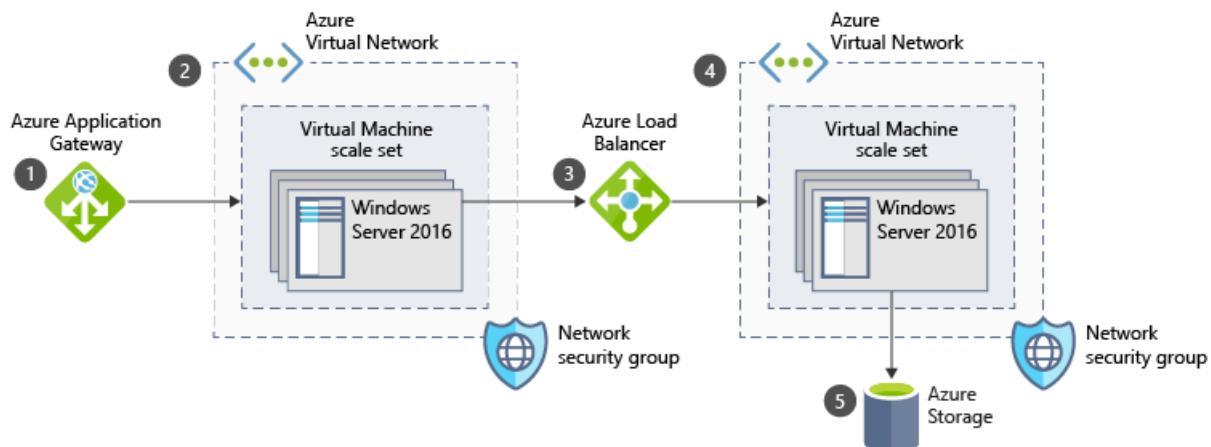
Example application scenarios include running operating room applications, patient appointments and records keeping, or prescription refills and ordering. Traditionally, organizations had to maintain legacy on-premises applications and services for these scenarios. With a secure way and scalable way to deploy these Windows Server applications in Azure, organizations can modernize their deployments and reduce their on-premises operating costs and management overhead.

## Relevant use cases

Consider this scenario for the following use cases:

- Modernizing application deployments in a secure cloud environment.
- Reducing legacy on-premises application and service management.
- Improving patient healthcare and experience with new application platforms.

## Architecture



This scenario covers a multi-tier regulated industries application that uses ASP.NET and Microsoft SQL Server. The data flows through the scenario as follows:

1. Users access the front-end ASP.NET regulated industries application through an Azure Application Gateway.
2. The Application Gateway distributes traffic to VM instances within an Azure virtual machine scale set.
3. The ASP.NET application connects to Microsoft SQL Server cluster in a back-end tier via an Azure load balancer. These back-end SQL Server instances are in a separate Azure virtual network, secured by network security group rules that limit traffic flow.
4. The load balancer distributes SQL Server traffic to VM instances in another virtual machine scale set.
5. Azure Blob Storage acts as a Cloud Witness for the SQL Server cluster in the back-end tier. The connection from within the VNet is enabled with a VNet Service Endpoint for Azure Storage.

## Components

- **Azure Application Gateway** is a layer 7 web traffic load balancer that is application-aware and can distribute

traffic based on specific routing rules. App Gateway can also handle SSL offloading for improved web server performance.

- [Azure Virtual Network](#) allows resources such as VMs to securely communicate with each other, the Internet, and on-premises networks. Virtual networks provide isolation and segmentation, filter and route traffic, and allow connection between locations. Two virtual networks combined with the appropriate NSGs are used in this scenario to provide a [demilitarized zone](#) (DMZ) and isolation of the application components. Virtual network peering connects the two networks together.
- [Azure virtual machine scale set](#) lets you create and manage a group of identical, load balanced, VMs. The number of VM instances can automatically increase or decrease in response to demand or a defined schedule. Two separate virtual machine scale sets are used in this scenario - one for the front-end ASP.NET application instances, and one for the back-end SQL Server cluster VM instances. PowerShell desired state configuration (DSC) or the Azure custom script extension can be used to provision the VM instances with the required software and configuration settings.
- [Azure network security groups](#) contain a list of security rules that allow or deny inbound or outbound network traffic based on source or destination IP address, port, and protocol. The virtual networks in this scenario are secured with network security group rules that restrict the flow of traffic between the application components.
- [Azure load balancer](#) distributes inbound traffic according to rules and health probes. A load balancer provides low latency and high throughput, and scales up to millions of flows for all TCP and UDP applications. An internal load balancer is used in this scenario to distribute traffic from the front-end application tier to the back-end SQL Server cluster.
- [Azure Blob Storage](#) acts a Cloud Witness location for the SQL Server cluster. This witness is used for cluster operations and decisions that require an additional vote to decide quorum. Using Cloud Witness removes the need for an additional VM to act as a traditional File Share Witness.

## Alternatives

- \*nix, windows can easily be replaced by a variety of other operating systems as nothing in the infrastructure depends on the operating system.
- [SQL Server for Linux](#) can replace the back-end data store.
- [Cosmos DB](#) is another alternative for the data store.

## Considerations

### Availability

The VM instances in this scenario are deployed across Availability Zones. Each zone is made up of one or more datacenters equipped with independent power, cooling, and networking. A minimum of three zones are available in all enabled regions. This distribution of VM instances across zones provides high availability to the application tiers. For more information, see [what are Availability Zones in Azure?](#)

The database tier can be configured to use Always On availability groups. With this SQL Server configuration, one primary database within a cluster is configured with up to eight secondary databases. If an issue occurs with the primary database, the cluster fails over to one of the secondary databases, which allows the application to continue to be available. For more information, see [Overview of Always On availability groups for SQL Server](#).

For other availability topics, see the [availability checklist](#) in the Azure Architecture Center.

### Scalability

This scenario uses virtual machine scale sets for the front-end and back-end components. With scale sets, the number of VM instances that run the front-end application tier can automatically scale in response to customer demand, or based on a defined schedule. For more information, see [Overview of autoscale with virtual machine scale sets](#).

For other scalability topics, see the [scalability checklist](#) in the Azure Architecture Center.

## Security

All the virtual network traffic into the front-end application tier is protected by network security groups. Rules limit the flow of traffic so that only the front-end application tier VM instances can access the back-end database tier. No outbound Internet traffic is allowed from the database tier. To reduce the attack footprint, no direct remote management ports are open. For more information, see [Azure network security groups](#).

To view guidance on deploying Payment Card Industry Data Security Standards (PCI DSS 3.2) [compliant infrastructure](#). For general guidance on designing secure scenarios, see the [Azure Security Documentation](#).

## Resiliency

In combination with the use of Availability Zones and virtual machine scale sets, this scenario uses Azure Application Gateway and load balancer. These two networking components distribute traffic to the connected VM instances, and include health probes that ensure traffic is only distributed to healthy VMs. Two Application Gateway instances are configured in an active-passive configuration, and a zone-redundant load balancer is used. This configuration makes the networking resources and application resilient to issues that would otherwise disrupt traffic and impact end-user access.

For general guidance on designing resilient scenarios, see [Designing resilient applications for Azure](#).

## Deploy the scenario

### Prerequisites.

- You must have an existing Azure account. If you don't have an Azure subscription, create a [free account](#) before you begin.
- To deploy a SQL Server cluster into the back-end scale set, you would need a domain in Azure Active Directory (AD) Domain Services.

To deploy the core infrastructure for this scenario with an Azure Resource Manager template, perform the following steps.

1. Select the **Deploy to Azure** button:



2. Wait for the template deployment to open in the Azure portal, then complete the following steps:

- Choose to **Create new** resource group, then provide a name such as *myWindowsscenario* in the text box.
- Select a region from the **Location** drop-down box.
- Provide a username and secure password for the virtual machine scale set instances.
- Review the terms and conditions, then check **I agree to the terms and conditions stated above**.
- Select the **Purchase** button.

It can take 15-20 minutes for the deployment to complete.

## Pricing

To explore the cost of running this scenario, all of the services are pre-configured in the cost calculator. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic.

We have provided three sample cost profiles based on the number of scale set VM instances that run your applications.

- **Small**: this pricing example correlates to two front-end and two back-end VM instances.
- **Medium**: this pricing example correlates to 20 front-end and 5 back-end VM instances.

- **Large**: this pricing example correlates to 100 front-end and 10 back-end VM instances.

## Related resources

This scenario used a back-end virtual machine scale set that runs a Microsoft SQL Server cluster. Cosmos DB could also be used as a scalable and secure database tier for the application data. An [Azure virtual network service endpoint](#) allows you to secure your critical Azure service resources to only your virtual networks. In this scenario, VNet endpoints allow you to secure traffic between the front-end application tier and Cosmos DB. For more information, see the [Azure Cosmos DB overview][docs-cosmos-db](#).

You can also view a detailed [reference architecture for a generic N-tier application using SQL Server](#).

# 3D video rendering on Azure

10/5/2018 • 5 minutes to read • [Edit Online](#)

3D video rendering is a time consuming process that requires a significant amount of CPU time to complete. On a single machine, the process of generating a video file from static assets can take hours or even days depending on the length and complexity of the video you are producing. Many companies will purchase either expensive high end desktop computers to perform these tasks, or invest in large render farms that they can submit jobs to. However, by taking advantage of Azure Batch, that power is available to you when you need it and shuts itself down when you don't, all without any capital investment.

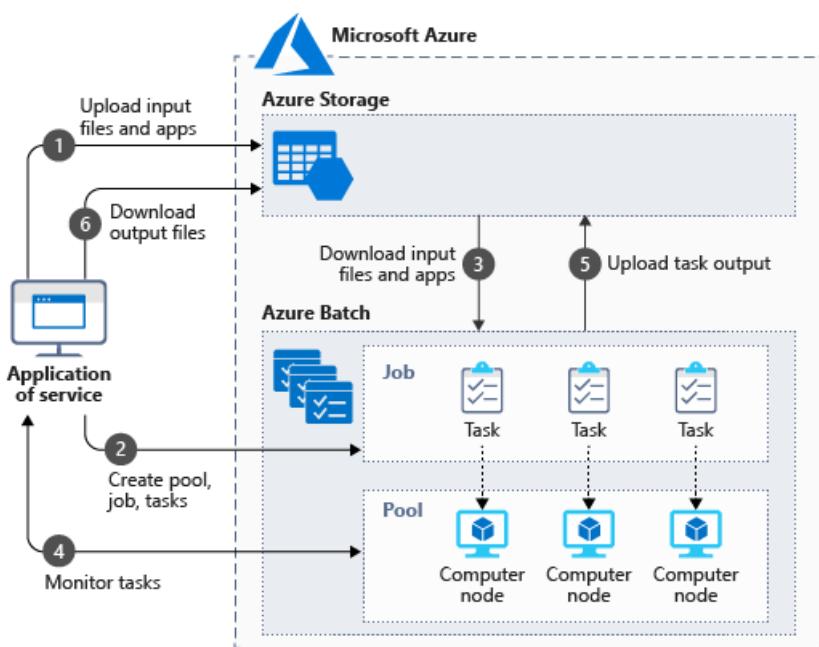
Batch gives you a consistent management experience and job scheduling, whether you select Windows Server or Linux compute nodes. With Batch, you can use your existing Windows or Linux applications, including AutoDesl Maya and Blender, to run large-scale render jobs in Azure.

## Relevant use cases

Consider this scenario for these similar use cases:

- 3D modeling
- Visual FX (VFX) rendering
- Video transcoding
- Image processing, color correction, and resizing

## Architecture



This scenario shows a workflow that uses Azure Batch. The data flows as follows:

1. Upload input files and the applications to process those files to your Azure Storage account.
2. Create a Batch pool of compute nodes in your Batch account, a job to run the workload on the pool, and tasks in the job.
3. Download input files and the applications to Batch.
4. Monitor task execution.
5. Upload task output.

## 6. Download output files.

To simplify this process, you could also use the [Batch Plugins for Maya and 3ds Max](#)

### Components

Azure Batch builds upon the following Azure technologies:

- [Virtual Networks](#) are used for both the head node and the compute resources.
- [Azure Storage accounts](#) are used for synchronization and data retention.
- [Virtual Machine Scale Sets](#) are used by CycleCloud for compute resources.

## Considerations

### Machine Sizes available for Azure Batch

While most rendering customers will choose resources with high CPU power, other workloads using virtual machine scale sets may choose VMs differently and will depend on a number of factors:

- Is the application being run memory bound?
- Does the application need to use GPUs?
- Are the job types embarrassingly parallel or require infiniband connectivity for tightly coupled jobs?
- Require fast I/O to access storage on the compute Nodes.

Azure has a wide range of VM sizes that can address each and every one of the above application requirements, some are specific to HPC, but even the smallest sizes can be utilized to provide an effective grid implementation:

- [HPC VM sizes](#) Due to the CPU bound nature of rendering, Microsoft typically suggests the Azure H-Series VMs. This type of VM is built specifically for high end computational needs, they have 8 and 16 core vCPU sizes available, and features DDR4 memory, SSD temporary storage, and Haswell E5 Intel technology.
- [GPU VM sizes](#) GPU optimized VM sizes are specialized virtual machines available with single or multiple NVIDIA GPUs. These sizes are designed for compute-intensive, graphics-intensive, and visualization workloads.
- NC, NCv2, NCv3, and ND sizes are optimized for compute-intensive and network-intensive applications and algorithms, including CUDA and OpenCL-based applications and simulations, AI, and Deep Learning. NV sizes are optimized and designed for remote visualization, streaming, gaming, encoding, and VDI scenarios utilizing frameworks such as OpenGL and DirectX.
- [Memory optimized VM sizes](#) When more memory is required, the memory optimized VM sizes offer a higher memory-to-CPU ratio.
- [General purposes VM sizes](#) General-purpose VM sizes are also available and provide balanced CPU-to-memory ratio.

### Alternatives

If you require more control over your rendering environment in Azure or need a hybrid implementation, then CycleCloud computing can help orchestrate an IaaS grid in the cloud. Using the same underlying Azure technologies as Azure Batch, it makes building and maintaining an IaaS grid an efficient process. To find out more and learn about the design principles use the following link:

For a complete overview of all the HPC solutions that are available to you in Azure, see the article [HPC, Batch, and Big Compute solutions using Azure VMs](#)

### Availability

Monitoring of the Azure Batch components is available through a range of services, tools, and APIs. Monitoring is discussed further in the [Monitor Batch solutions](#) article.

### Scalability

Pools within an Azure Batch account can either scale through manual intervention or, by using a formula based on

Azure Batch metrics, be scaled automatically. For more information on scalability, see the article [Create an automatic scaling formula for scaling nodes in a Batch pool](#).

## Security

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

## Resiliency

While there is currently no failover capability in Azure Batch, we recommend using the following steps to ensure availability if there is an unplanned outage:

- Create an Azure Batch account in an alternate Azure location with an alternate Storage Account
- Create the same node pools with the same name, with zero nodes allocated
- Ensure Applications are created and updated to the alternate Storage Account
- Upload input files and submit jobs to the alternate Azure Batch account

## Deploy this scenario

### Creating an Azure Batch account and pools manually

This scenario demonstrates how Azure Batch works while showcasing Azure Batch Labs as an example SaaS solution that can be developed for your own customers:

[Azure Batch Masterclass](#)

### Deploying the example scenario using an Azure Resource Manager template

The template will deploy:

- A new Azure Batch account
- A storage account
- A node pool associated with the batch account
- The node pool will be configured to use A2 v2 VMs with Canonical Ubuntu images
- The node pool will contain zero VMs initially and will require you to manually scale to add VMs



[Learn more about Resource Manager templates](#)

## Pricing

The cost of using Azure Batch will depend on the VM sizes that are used for the pools and how long these VMs are allocated and running, there is no cost associated with an Azure Batch account creation. Storage and data egress should be taken into account as these will apply additional costs.

The following are examples of costs that could be incurred for a job that completes in 8 hours using a different number of servers:

- 100 High-Performance CPU VMs: [Cost Estimate](#)

100 x H16m (16 cores, 225 GB RAM, Premium Storage 512 GB), 2 TB Blob Storage, 1-TB egress

- 50 High-Performance CPU VMs: [Cost Estimate](#)

50 x H16m (16 cores, 225 GB RAM, Premium Storage 512 GB), 2 TB Blob Storage, 1-TB egress

- 10 High-Performance CPU VMs: [Cost Estimate](#)

10 x H16m (16 cores, 225 GB RAM, Premium Storage 512 GB), 2 TB Blob Storage, 1-TB egress

## Pricing for low-priority VMs

Azure Batch also supports the use of low-priority VMs in the node pools, which can potentially provide a substantial cost saving. For more information, including a price comparison between standard VMs and low-priority VMs, see [Azure Batch Pricing](#).

### NOTE

Low-priority VMs are only suitable for certain applications and workloads.

## Related resources

[Azure Batch Overview](#)

[Azure Batch Documentation](#)

[Using containers on Azure Batch](#)

Our reference architectures are arranged by scenario, with related architectures grouped together. Each architecture includes recommended practices, along with considerations for scalability, availability, manageability, and security. Most also include a deployable solution.

Jump to: [AI](#) | [Big data](#) | [N-tier applications](#) | [Virtual networks](#) | [Active Directory](#) | [VM workloads](#) | [Web applications](#)

## AI and machine learning



### **Batch scoring for deep learning models**

Automate running batch jobs that apply neural style transfer to a video.

## Big data solutions

### **Enterprise BI with SQL Data Warehouse**

ELT (extract-load-transform) pipeline to move data from an on-premises database into SQL Data Warehouse.

### **Automated enterprise BI with Azure Data Factory**

Automate an ELT pipeline to perform incremental loading from an on-premises database.



### **Stream processing with Azure Stream Analytics**

End-to-end stream processing pipeline that correlates records from two data streams to calculate a rolling average.

## N-tier applications



### **N-tier application with SQL Server**

Virtual machines configured for an N-tier application using SQL Server on Windows.



### **Multi-region N-tier application**

N-tier application in two regions for high availability, using SQL Server Always On Availability Groups.



### **N-tier application with Cassandra**

Virtual machines configured for an N-tier application using Apache Cassandra on Linux.

# Virtual networks



## Hybrid network using a virtual private network (VPN)

Connect an on-premises network to an Azure virtual network.



## Hybrid network using ExpressRoute

Use a private, dedicated connection to extend an on-premises network to Azure.



## Hybrid network using ExpressRoute with VPN failover

Use ExpressRoute with a VPN as a failover connection for high availability.

## Hub-spoke network topology

Create a central point of connectivity to your on-premises network, while isolating workloads.

## Hub-spoke topology with shared services

Extend a hub-spoke topology by including shared services such as Active Directory.



## DMZ between Azure and on-premises

Use network virtual appliances to create a secure hybrid network.



## DMZ between Azure and the Internet

Use network virtual appliances to create a secure network that accepts Internet traffic.

# Extending on-premises Active Directory to Azure



## Integrate with Azure Active Directory

Integrate on-premises AD domains with Azure Active Directory.

## Extend an on-premises Active Directory domain to Azure

Deploy Active Directory Domain Services (AD DS) in Azure to extend your on-premises domain.

## Create an AD DS forest in Azure

Create a separate AD domain in Azure that is trusted by your on-premises AD forest.

## Extend Active Directory Federation Services (AD FS) to Azure

Use AD FS for federated authentication and authorization for components running in Azure.

## VM workloads



### Jenkins build server

Scalable, enterprise-grade Jenkins server on Azure.

### SharePoint Server 2016 farm

Highly available SharePoint Server 2016 farm on Azure with SQL Server Always On Availability Groups.



### SAP NetWeaver

SAP NetWeaver on Windows, in a high availability environment that supports disaster recovery.



### SAP S/4HANA

SAP S/4HANA on Linux, in a high availability environment that supports disaster recovery.



### SAP HANA on Azure Large Instances

HANA Large Instances are deployed on physical servers in Azure regions.

# Web applications



## Basic web application

Web application with Azure App Service and Azure SQL Database.



## Highly scalable web application

Proven practices for improving scalability in a web application.



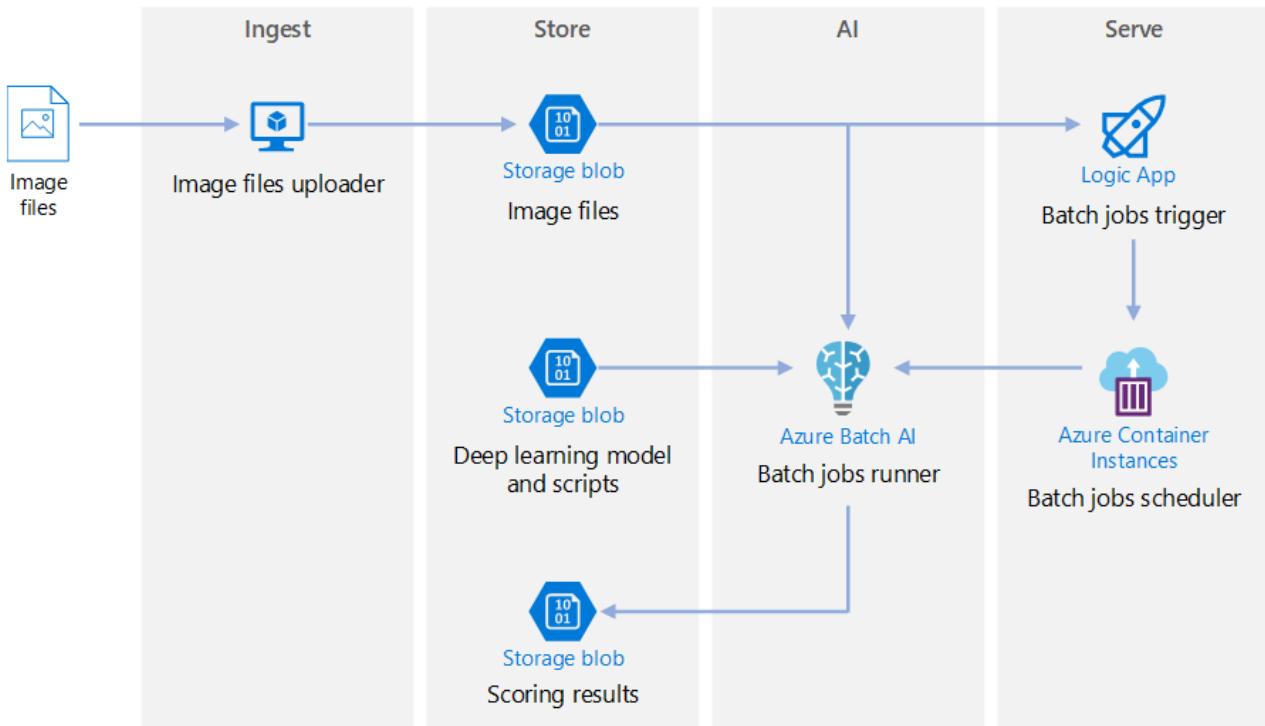
## Highly available web application

Run an App Service web app in multiple regions to achieve high availability.

# Batch scoring on Azure for deep learning models

10/3/2018 • 10 minutes to read • [Edit Online](#)

This reference architecture shows how to apply neural style transfer to a video, using Azure Batch AI. *Style transfer* is a deep learning technique that composes an existing image in the style of another image. This architecture can be generalized for any scenario that uses batch scoring with deep learning. [Deploy this solution](#).



**Scenario:** A media organization has a video whose style they want to change to look like a specific painting. The organization wants to be able to apply this style to all frames of the video in a timely manner and in an automated fashion. For more background about neural style transfer algorithms, see [Image Style Transfer Using Convolutional Neural Networks \(PDF\)](#).

STYLE IMAGE:	INPUT/CONTENT VIDEO:	OUTPUT VIDEO:
	 <a href="#">click to view video</a>	 <a href="#">click to view video</a>

This reference architecture is designed for workloads that are triggered by the presence of new media in Azure storage. Processing involves the following steps:

1. Upload a selected style image (like a Van Gogh painting) and a style transfer script to Blob Storage.

2. Create an autoscaling Batch AI cluster that is ready to start taking work.
3. Split the video file into individual frames and upload those frames into Blob Storage.
4. Once all frames are uploaded, upload a trigger file to Blob Storage.
5. This file triggers a Logic App that creates a container running in Azure Container Instances.
6. The container runs a script that creates the Batch AI jobs. Each job applies the neural style transfer in parallel across the nodes of the Batch AI cluster.
7. Once the images are generated, they are saved back to Blob Storage.
8. Download the generated frames, and stitch back the images into a video.

## Architecture

This architecture consists of the following components.

### Compute

**Azure Batch AI** is used to run the neural style transfer algorithm. Batch AI supports deep learning workloads by providing containerized environments that are pre-configured for deep learning frameworks, on GPU-enabled VMs. Batch AI can also connect the compute cluster to Blob storage.

### Storage

**Blob storage** is used to store all images (input images, style images, and output images) as well as all logs produced from Batch AI. Blob storage integrates with Batch AI via [blobfuse](#), an open-source virtual filesystem that is backed by Blob storage. Blob storage is also very cost-effective for the performance that this workload requires.

### Trigger / scheduling

**Azure Logic Apps** is used to trigger the workflow. When the Logic App detects that a blob has been added to the container, it triggers the Batch AI process. Logic Apps is a good fit for this reference architecture because it's an easy way to detect changes to blob storage and provides an easy process for changing the trigger.

**Azure Container Instances** is used to run the Python scripts that create the Batch AI jobs. Running these scripts inside a Docker container is a convenient way to run them on demand. For this architecture, we use Container Instances because there is a pre-built Logic App connector for it, which allows the Logic App to trigger the Batch AI job. Container Instances can spin up stateless processes quickly.

**DockerHub** is used to store the Docker image that Container Instances uses to execute the job creation process. DockerHub was chosen for this architecture because it's easy to use and is the default image repository for Docker users. [Azure Container Registry](#) can also be used.

### Data preparation

This reference architecture uses video footage of an orangutan in a tree. You can download the footage from [here](#) and process it for the workflow by following these steps:

1. Use [AzCopy](#) to download the video from the public blob.
2. Use [FFmpeg](#) to extract the audio file, so that the audio file can be stitched back into the output video later.
3. Use FFmpeg to break the video into individual frames. The frames will be processed independently, in parallel.
4. Use AzCopy to copy the individual frames into your blob container.

At this stage, the video footage is in a form that can be used for neural style transfer.

## Performance considerations

### GPU vs CPU

For deep learning workloads, GPUs will generally out-perform CPUs by a considerable amount, to the extent that a sizeable cluster of CPUs is usually needed to get comparable performance. While it's an option to use only CPUs in this architecture, GPUs will provide a much better cost/performance profile. We recommend using the latest

[NCv3 series]vm-sizes-gpu of GPU optimized VMs.

GPUs are not enabled by default in all regions. Make sure to select a region with GPUs enabled. In addition, subscriptions have a default quota of zero cores for GPU-optimized VMs. You can raise this quota by opening a support request. Make sure that your subscription has enough quota to run your workload.

### Parallelizing across VMs vs cores

When running a style transfer process as a batch job, the jobs that run primarily on GPUs will have to be parallelized across VMs. Two approaches are possible: You can create a larger cluster using VMs that have a single GPU, or create a smaller cluster using VMs with many GPUs.

For this workload, these two options will have comparable performance. Using fewer VMs with more GPUs per VM can help to reduce data movement. However, the data volume per job for this workload is not very big, so you won't observe much throttling by blob storage.

### Images batch size per Batch AI job

Another parameter that must be configured is the number of images to process per Batch AI job. On the one hand, you want to ensure that work is spread broadly across the nodes and that if a job fails, you don't have to retry too many images. That points to having many Batch AI jobs and thus a low number of images to process per job. On the other hand, if too few images are processed per job, the setup/startup time becomes disproportionately large. You can set the number of jobs to equal the maximum number of nodes in the cluster. This will be the most performant assuming that no jobs fail, because it minimizes the amount of setup/startup cost. However, if a job fails, a large number of images might need to be reprocessed.

### File servers

When using Batch AI, you can choose multiple storage options depending on the throughput needed for your scenario. For workloads with low throughput requirements, using blob storage (via blobfuse) should be enough. Alternatively, Batch AI also supports a Batch AI File Server, a managed single-node NFS, which can be automatically mounted on cluster nodes to provide a centrally accessible storage location for jobs. For most cases, only one file server is needed in a workspace, and you can separate data for your training jobs into different directories. If a single-node NFS isn't appropriate for your workloads, Batch AI supports other storage options, including Azure Files or custom solutions such as a Gluster or Lustre file system.

## Security considerations

### Restricting access to Azure blob storage

In this reference architecture, Azure blob storage is the main storage component that needs to be protected. The baseline deployment shown in the GitHub repo uses storage account keys to access the blob storage. For further control and protection, consider using a shared access signature (SAS) instead. This grants limited access to objects in storage, without needing to hard code the account keys or save them in plaintext. This approach is especially useful because account keys are visible in plaintext inside of Logic App's designer interface. Using an SAS also helps to ensure that the storage account has proper governance, and that access is granted only to the people intended to have it.

For scenarios with more sensitive data, make sure that all of your storage keys are protected, because these keys grant full access to all input and output data from the workload.

### Data encryption and data movement

This reference architecture uses style transfer as an example of a batch scoring process. For more data-sensitive scenarios, the data in storage should be encrypted at rest. Each time data is moved from one location to the next, use SSL to secure the data transfer. For more information, see [Azure Storage security guide](#).

### Securing data in a virtual network

When deploying your Batch AI cluster, you can configure your cluster to be provisioned inside a subnet of a virtual

network. This allows the compute nodes in the cluster to communicate securely with other virtual machines, or even with an on-premises network. You can also use [service endpoints](#) with blob storage to grant access from a virtual network or use a single-node NFS inside the VNET with Batch AI to ensure that the data is always protected.

### Protecting against malicious activity

In scenarios where there are multiple users, make sure that sensitive data is protected against malicious activity. If other users are given access to this deployment to customize the input data, take note of the following precautions and considerations:

- Use RBAC to limit users' access to only the resources they need.
- Provision two separate storage accounts. Store input and output data in the first account. External users can be given access to this account. Store executable scripts and output log files in the other account. External users should not have access to this account. This will ensure that external users cannot modify any executable files (to inject malicious code), and don't have access to logfiles, which could hold sensitive information.
- Malicious users can DDOS the job queue or inject malformed poison messages in the job queue, causing the system to lock up or causing dequeuing errors.

## Monitoring and logging

### Monitoring Batch AI jobs

While running your job, it's important to monitor the progress and make sure that things are working as expected. However, it can be a challenge to monitor across a cluster of active nodes.

To get a sense of the overall state of the cluster, go to the Batch AI blade of the Azure Portal to inspect the state of the nodes in the cluster. If a node is inactive or a job has failed, the error logs are saved to blob storage, and are also accessible in the Jobs blade in the Azure Portal.

Monitoring can be further enriched by connecting logs to Application Insights or by running separate processes to poll for the state of the Batch AI cluster and its jobs.

### Logging in Batch AI

Batch AI will automatically log all stdout/stderr to the associate blob storage account. Using a storage navigation tool such as Storage Explorer will provide a much easier experience for navigating log files.

The deployment steps for this reference architecture also shows how to set up a more simple logging system, such that all the logs across the different jobs are saved to the same directory in your blob container, as shown below. Use these logs to monitor how long it takes for each job and each image to process. This will give you a better sense of how to optimize the process even further.

## Cost considerations

Compared to the storage and scheduling components, the compute resources used in this reference architecture by far dominate in terms of costs. One of the main challenges is effectively parallelizing the work across a cluster of GPU-enabled machines.

The Batch AI cluster size can automatically scale up and down depending on the jobs in the queue. You can enable auto-scale with Batch AI in one of two ways. You can do so programmatically, which can be configured in the `.env` file that is part of the [deployment steps](#), or you can change the scale formula directly in the portal after the cluster is created.

For work that doesn't require immediate processing, configure the auto-scale formula so the default state (minimum) is a cluster of zero nodes. With this configuration, the cluster starts with zero nodes and only scales up when it detects jobs in the queue. If the batch scoring process only happens a few times a day or less, this setting enables significant cost savings.

Auto-scaling may not be appropriate for batch jobs that happen too close to each other. The time that it takes for a cluster to spin up and spin down also incur a cost, so if a batch workload begins only a few minutes after the previous job ends, it might be more cost effective to keep the cluster running between jobs.

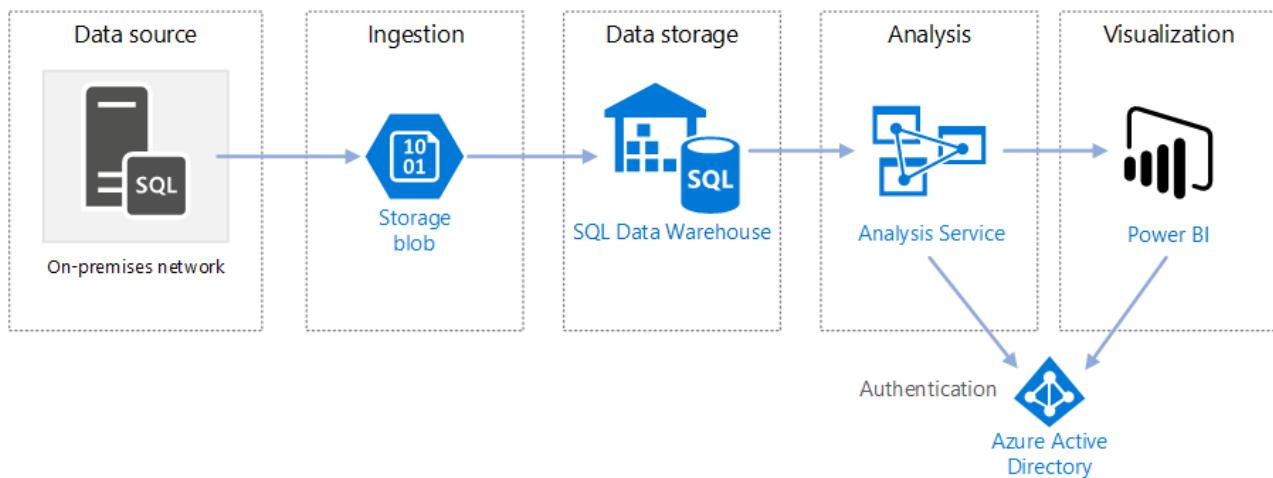
## Deploy the solution

To deploy this reference architecture, follow the steps described in the [GitHub repo](#).

# Enterprise BI with SQL Data Warehouse

7/2/2018 • 18 minutes to read • [Edit Online](#)

This reference architecture implements an [ELT](#) (extract-load-transform) pipeline that moves data from an on-premises SQL Server database into SQL Data Warehouse and transforms the data for analysis. [Deploy this solution.](#)



**Scenario:** An organization has a large OLTP data set stored in a SQL Server database on premises. The organization wants to use SQL Data Warehouse to perform analysis using Power BI.

This reference architecture is designed for one-time or on-demand jobs. If you need to move data on a continuing basis (hourly or daily), we recommend using Azure Data Factory to define an automated workflow. For a reference architecture that uses Data Factory, see [Automated enterprise BI with SQL Data Warehouse and Azure Data Factory](#).

## Architecture

The architecture consists of the following components.

### Data source

**SQL Server.** The source data is located in a SQL Server database on premises. To simulate the on-premises environment, the deployment scripts for this architecture provision a VM in Azure with SQL Server installed. The [Wide World Importers OLTP sample database](#) is used as the source data.

### Ingestion and data storage

**Blob Storage.** Blob storage is used as a staging area to copy the data before loading it into SQL Data Warehouse.

**Azure SQL Data Warehouse.** [SQL Data Warehouse](#) 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.

### Analysis and reporting

**Azure Analysis Services.** [Analysis Services](#) is a fully managed service that provides data modeling capabilities. Use Analysis Services to create a semantic model that users can query. Analysis Services is especially useful in a BI dashboard scenario. In this architecture, Analysis Services reads data from the data warehouse to process the semantic model, and efficiently serves dashboard queries. It also supports elastic concurrency, by scaling out replicas for faster query processing.

Currently, Azure Analysis Services supports tabular models but not multidimensional models. Tabular models use relational modeling constructs (tables and columns), whereas multidimensional models use OLAP modeling constructs (cubes, dimensions, and measures). If you require multidimensional models, use SQL Server Analysis Services (SSAS). For more information, see [Comparing tabular and multidimensional solutions](#).

**Power BI.** Power BI is a suite of business analytics tools to analyze data for business insights. In this architecture, it queries the semantic model stored in Analysis Services.

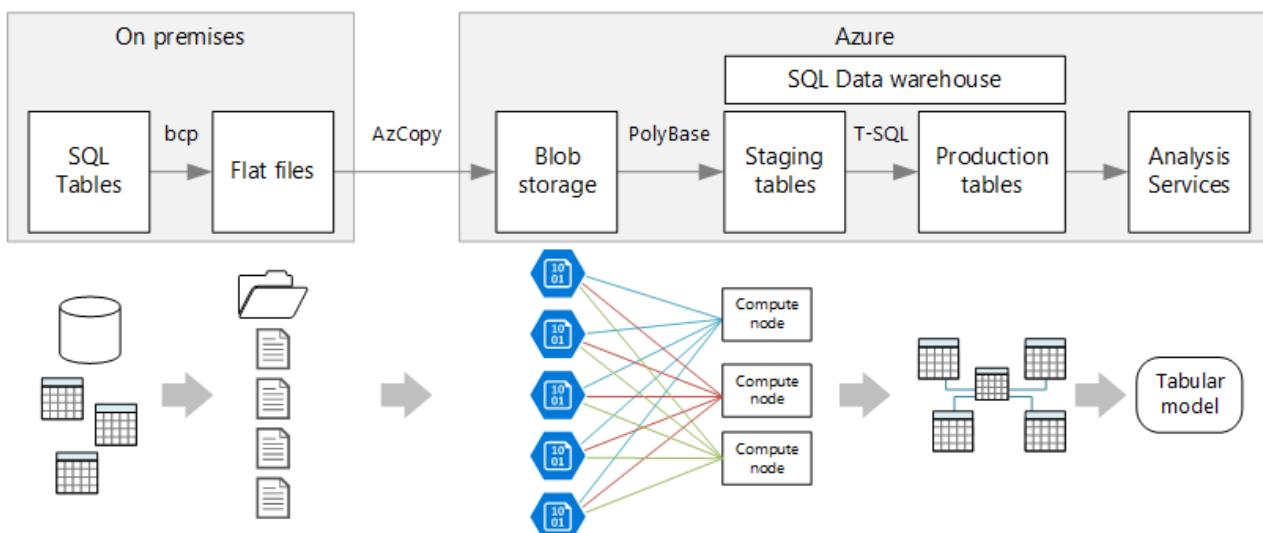
## Authentication

**Azure Active Directory** (Azure AD) authenticates users who connect to the Analysis Services server through Power BI.

## Data pipeline

This reference architecture uses the [WorldWideImporters](#) sample database as a data source. The data pipeline has the following stages:

1. Export the data from SQL Server to flat files (bcp utility).
2. Copy the flat files to Azure Blob Storage (AzCopy).
3. Load the data into SQL Data Warehouse (PolyBase).
4. Transform the data into a star schema (T-SQL).
5. Load a semantic model into Analysis Services (SQL Server Data Tools).



### NOTE

For steps 1 – 3, consider using Redgate Data Platform Studio. Data Platform Studio applies the most appropriate compatibility fixes and optimizations, so it's the quickest way to get started with SQL Data Warehouse. For more information, see [Load data with Redgate Data Platform Studio](#).

The next sections describe these stages in more detail.

### Export data from SQL Server

The [bcp](#) (bulk copy program) utility is a fast way to create flat text files from SQL tables. In this step, you select the columns that you want to export, but don't transform the data. Any data transformations should happen in SQL Data Warehouse.

### Recommendations

If possible, schedule data extraction during off-peak hours, to minimize resource contention in the production environment.

Avoid running bcp on the database server. Instead, run it from another machine. Write the files to a local drive. Ensure that you have sufficient I/O resources to handle the concurrent writes. For best performance, export the files to dedicated fast storage drives.

You can speed up the network transfer by saving the exported data in Gzip compressed format. However, loading compressed files into the warehouse is slower than loading uncompressed files, so there is a tradeoff between faster network transfer versus faster loading. If you decide to use Gzip compression, don't create a single Gzip file. Instead, split the data into multiple compressed files.

### **Copy flat files into blob storage**

The [AzCopy](#) utility is designed for high-performance copying of data into Azure blob storage.

#### **Recommendations**

Create the storage account in a region near the location of the source data. Deploy the storage account and the SQL Data Warehouse instance in the same region.

Don't run AzCopy on the same machine that runs your production workloads, because the CPU and I/O consumption can interfere with the production workload.

Test the upload first to see what the upload speed is like. You can use the /NC option in AzCopy to specify the number of concurrent copy operations. Start with the default value, then experiment with this setting to tune the performance. In a low-bandwidth environment, too many concurrent operations can overwhelm the network connection and prevent the operations from completing successfully.

AzCopy moves data to storage over the public internet. If this isn't fast enough, consider setting up an [ExpressRoute](#) circuit. ExpressRoute is a service that routes your data through a dedicated private connection to Azure. Another option, if your network connection is too slow, is to physically ship the data on disk to an Azure datacenter. For more information, see [Transferring data to and from Azure](#).

During a copy operation, AzCopy creates a temporary journal file, which enables AzCopy to restart the operation if it gets interrupted (for example, due to a network error). Make sure there is enough disk space to store the journal files. You can use the /Z option to specify where the journal files are written.

### **Load data into SQL Data Warehouse**

Use [PolyBase](#) to load the files from blob storage into the data warehouse. PolyBase is designed to leverage the MPP (Massively Parallel Processing) architecture of SQL Data Warehouse, which makes it the fastest way to load data into SQL Data Warehouse.

Loading the data is a two-step process:

1. Create a set of external tables for the data. An external table is a table definition that points to data stored outside of the warehouse — in this case, the flat files in blob storage. This step does not move any data into the warehouse.
2. Create staging tables, and load the data into the staging tables. This step copies the data into the warehouse.

#### **Recommendations**

Consider SQL Data Warehouse when you have large amounts of data (more than 1 TB) and are running an analytics workload that will benefit from parallelism. SQL Data Warehouse is not a good fit for OLTP workloads or smaller data sets (< 250GB). For data sets less than 250GB, consider Azure SQL Database or SQL Server. For more information, see [Data warehousing](#).

Create the staging tables as heap tables, which are not indexed. The queries that create the production tables will result in a full table scan, so there is no reason to index the staging tables.

PolyBase automatically takes advantage of parallelism in the warehouse. The load performance scales as you increase DWUs. For best performance, use a single load operation. There is no performance benefit to breaking

the input data into chunks and running multiple concurrent loads.

PolyBase can read Gzip compressed files. However, only a single reader is used per compressed file, because uncompressing the file is a single-threaded operation. Therefore, avoid loading a single large compressed file. Instead, split the data into multiple compressed files, in order to take advantage of parallelism.

Be aware of the following limitations:

- PolyBase supports a maximum column size of `varchar(8000)`, `nvarchar(4000)`, or `varbinary(8000)`. If you have data that exceeds these limits, one option is to break the data up into chunks when you export it, and then reassemble the chunks after import.
- PolyBase uses a fixed row terminator of `\n` or newline. This can cause problems if newline characters appear in the source data.
- Your source data schema might contain data types that are not supported in SQL Data Warehouse.

To work around these limitations, you can create a stored procedure that performs the necessary conversions. Reference this stored procedure when you run bcp. Alternatively, [Redgate Data Platform Studio](#) automatically converts data types that aren't supported in SQL Data Warehouse.

For more information, see the following articles:

- [Best practices for loading data into Azure SQL Data Warehouse](#).
- [Migrate your schemas to SQL Data Warehouse](#)
- [Guidance for defining data types for tables in SQL Data Warehouse](#)

## Transform the data

Transform the data and move it into production tables. In this step, the data is transformed into a star schema with dimension tables and fact tables, suitable for semantic modeling.

Create the production tables with clustered columnstore indexes, which offer the best overall query performance. Columnstore indexes are optimized for queries that scan many records. Columnstore indexes don't perform as well for singleton lookups (that is, looking up a single row). If you need to perform frequent singleton lookups, you can add a non-clustered index to a table. Singleton lookups can run significantly faster using a non-clustered index. However, singleton lookups are typically less common in data warehouse scenarios than OLTP workloads. For more information, see [Indexing tables in SQL Data Warehouse](#).

### NOTE

Clustered columnstore tables do not support `varchar(max)`, `nvarchar(max)`, or `varbinary(max)` data types. In that case, consider a heap or clustered index. You might put those columns into a separate table.

Because the sample database is not very large, we created replicated tables with no partitions. For production workloads, using distributed tables is likely to improve query performance. See [Guidance for designing distributed tables in Azure SQL Data Warehouse](#). Our example scripts run the queries using a static resource class.

## Load the semantic model

Load the data into a tabular model in Azure Analysis Services. In this step, you create a semantic data model by using SQL Server Data Tools (SSDT). You can also create a model by importing it from a Power BI Desktop file. Because SQL Data Warehouse does not support foreign keys, you must add the relationships to the semantic model, so that you can join across tables.

## Use Power BI to visualize the data

Power BI supports two options for connecting to Azure Analysis Services:

- Import. The data is imported into the Power BI model.

- Live Connection. Data is pulled directly from Analysis Services.

We recommend Live Connection because it doesn't require copying data into the Power BI model. Also, using DirectQuery ensures that results are always consistent with the latest source data. For more information, see [Connect with Power BI](#).

## Recommendations

Avoid running BI dashboard queries directly against the data warehouse. BI dashboards require very low response times, which direct queries against the warehouse may be unable to satisfy. Also, refreshing the dashboard will count against the number of concurrent queries, which could impact performance.

Azure Analysis Services is designed to handle the query requirements of a BI dashboard, so the recommended practice is to query Analysis Services from Power BI.

## Scalability considerations

### SQL Data Warehouse

With SQL Data Warehouse, you can scale out your compute resources on demand. The query engine optimizes queries for parallel processing based on the number of compute nodes, and moves data between nodes as necessary. For more information, see [Manage compute in Azure SQL Data Warehouse](#).

### Analysis Services

For production workloads, we recommend the Standard Tier for Azure Analysis Services, because it supports partitioning and DirectQuery. Within a tier, the instance size determines the memory and processing power. Processing power is measured in Query Processing Units (QPU). Monitor your QPU usage to select the appropriate size. For more information, see [Monitor server metrics](#).

Under high load, query performance can become degraded due to query concurrency. You can scale out Analysis Services by creating a pool of replicas to process queries, so that more queries can be performed concurrently. The work of processing the data model always happens on the primary server. By default, the primary server also handles queries. Optionally, you can designate the primary server to run processing exclusively, so that the query pool handles all queries. If you have high processing requirements, you should separate the processing from the query pool. If you have high query loads, and relatively light processing, you can include the primary server in the query pool. For more information, see [Azure Analysis Services scale-out](#).

To reduce the amount of unnecessary processing, consider using partitions to divide the tabular model into logical parts. Each partition can be processed separately. For more information, see [Partitions](#).

## Security considerations

### IP whitelisting of Analysis Services clients

Consider using the Analysis Services firewall feature to whitelist client IP addresses. If enabled, the firewall blocks all client connections other than those specified in the firewall rules. The default rules whitelist the Power BI service, but you can disable this rule if desired. For more information, see [Hardening Azure Analysis Services with the new firewall capability](#).

### Authorization

Azure Analysis Services uses Azure Active Directory (Azure AD) to authenticate users who connect to an Analysis Services server. You can restrict what data a particular user is able to view, by creating roles and then assigning Azure AD users or groups to those roles. For each role, you can:

- Protect tables or individual columns.
- Protect individual rows based on filter expressions.

For more information, see [Manage database roles and users](#).

## Deploy the solution

A deployment for this reference architecture is available on [GitHub](#). It deploys the following:

- A Windows VM to simulate an on-premises database server. It includes SQL Server 2017 and related tools, along with Power BI Desktop.
- An Azure storage account that provides Blob storage to hold data exported from the SQL Server database.
- An Azure SQL Data Warehouse instance.
- An Azure Analysis Services instance.

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Deploy the simulated on-premises server

First you'll deploy a VM as a simulated on-premises server, which includes SQL Server 2017 and related tools. This step also loads the [Wide World Importers OLTP database](#) into SQL Server.

1. Navigate to the `data\enterprise_bi_sqldw\onprem\templates` folder of the repository.
2. In the `onprem.parameters.json` file, replace the values for `adminUsername` and `adminPassword`. Also change the values in the `SqlUserCredentials` section to match the user name and password. Note the `.\\` prefix in the `userName` property.

```
"SqlUserCredentials": {  
    "userName": ".\\username",  
    "password": "password"  
}
```

3. Run `azbb` as shown below to deploy the on-premises server.

```
azbb -s <subscription_id> -g <resource_group_name> -l <region> -p onprem.parameters.json --deploy
```

Specify a region that supports SQL Data Warehouse and Azure Analysis Services. See [Azure Products by Region](#)

4. The deployment may take 20 to 30 minutes to complete, which includes running the `DSC` script to install the tools and restore the database. Verify the deployment in the Azure portal by reviewing the resources in the resource group. You should see the `sql-vm1` virtual machine and its associated resources.

### Deploy the Azure resources

This step provisions SQL Data Warehouse and Azure Analysis Services, along with a Storage account. If you want, you can run this step in parallel with the previous step.

1. Navigate to the `data\enterprise_bi_sqldw\azure\templates` folder of the repository.
2. Run the following Azure CLI command to create a resource group. You can deploy to a different resource group than the previous step, but choose the same region.

```
az group create --name <resource_group_name> --location <region>
```

3. Run the following Azure CLI command to deploy the Azure resources. Replace the parameter values shown in angle brackets.

```
az group deployment create --resource-group <resource_group_name> \
--template-file azure-resources-deploy.json \
--parameters "dwServerName"<server_name>" \
"dwAdminLogin"<admin_username>" "dwAdminPassword"<password>" \
"storageAccountName"<storage_account_name>" \
"analysisServerName"<analysis_server_name>" \
"analysisServerAdmin"<user@contoso.com>"
```

- The `storageAccountName` parameter must follow the [naming rules](#) for Storage accounts.
  - For the `analysisServerAdmin` parameter, use your Azure Active Directory user principal name (UPN).
4. Verify the deployment in the Azure portal by reviewing the resources in the resource group. You should see a storage account, Azure SQL Data Warehouse instance, and Analysis Services instance.
  5. Use the Azure portal to get the access key for the storage account. Select the storage account to open it. Under **Settings**, select **Access keys**. Copy the primary key value. You will use it in the next step.

### Export the source data to Azure Blob storage

In this step, you will run a PowerShell script that uses bcp to export the SQL database to flat files on the VM, and then uses AzCopy to copy those files into Azure Blob Storage.

1. Use Remote Desktop to connect to the simulated on-premises VM.
2. While logged into the VM, run the following commands from a PowerShell window.

```
cd 'C:\SampleDataFiles\reference-architectures\data\enterprise_bi_sqldw\onprem'
.\Load_SourceData_To_Blob.ps1 -File .\sql_scripts\db_objects.txt -Destination
'https://<storage_account_name>.blob.core.windows.net/wwi' -StorageAccountKey '<storage_account_key>'
```

For the `Destination` parameter, replace `<storage_account_name>` with the name the Storage account that you created previously. For the `storageAccountKey` parameter, use the access key for that Storage account.

3. In the Azure portal, verify that the source data was copied to Blob storage by navigating to the storage account, selecting the Blob service, and opening the `wwi` container. You should see a list of tables prefaced with `WorldWideImporters_Application_*`.

### Run the data warehouse scripts

1. From your Remote Desktop session, launch SQL Server Management Studio (SSMS).

2. Connect to SQL Data Warehouse

- Server type: Database Engine
- Server name: `<dwServerName>.database.windows.net`, where `<dwServerName>` is the name that you specified when you deployed the Azure resources. You can get this name from the Azure portal.

- Authentication: SQL Server Authentication. Use the credentials that you specified when you deployed the Azure resources, in the `dwAdminLogin` and `dwAdminPassword` parameters.
3. Navigate to the `C:\SampleDataFiles\reference-architectures\data\enterprise_bi_sqldw\azure\sqldw_scripts` folder on the VM. You will execute the scripts in this folder in numerical order, `STEP_1` through `STEP_7`.
  4. Select the `master` database in SSMS and open the `STEP_1` script. Change the value of the password in the following line, then execute the script.

```
CREATE LOGIN LoaderRC20 WITH PASSWORD = '<change this value>';
```

5. Select the `wwi` database in SSMS. Open the `STEP_2` script and execute the script. If you get an error, make sure you are running the script against the `wwi` database and not `master`.
6. Open a new connection to SQL Data Warehouse, using the `LoaderRC20` user and the password indicated in the `STEP_1` script.
7. Using this connection, open the `STEP_3` script. Set the following values in the script:

- SECRET: Use the access key for your storage account.
- LOCATION: Use the name of the storage account as follows:

```
wasbs://wwi@<storage_account_name>.blob.core.windows.net .
```

8. Using the same connection, execute scripts `STEP_4` through `STEP_7` sequentially. Verify that each script completes successfully before running the next.

In SSMS, you should see a set of `prd.*` tables in the `wwi` database. To verify that the data was generated, run the following query:

```
SELECT TOP 10 * FROM prd.CityDimensions
```

## Build the Analysis Services model

In this step, you will create a tabular model that imports data from the data warehouse. Then you will deploy the model to Azure Analysis Services.

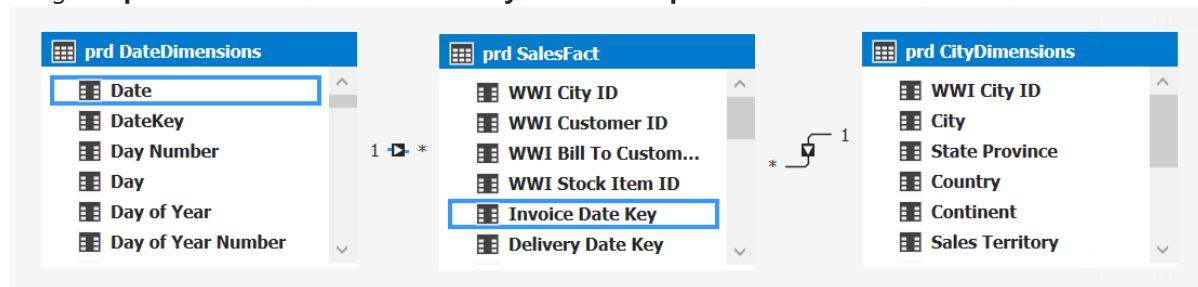
1. From your Remote Desktop session, launch SQL Server Data Tools 2015.
2. Select **File > New > Project**.
3. In the **New Project** dialog, under **Templates**, select **Business Intelligence > Analysis Services > Analysis Services Tabular Project**.
4. Name the project and click **OK**.
5. In the **Tabular model designer** dialog, select **Integrated workspace** and set **Compatibility level** to `SQL Server 2017 / Azure Analysis Services (1400)`. Click **OK**.
6. In the **Tabular Model Explorer** window, right-click the project and select **Import from Data Source**.
7. Select **Azure SQL Data Warehouse** and click **Connect**.
8. For **Server**, enter the fully qualified name of your Azure SQL Data Warehouse server. For **Database**, enter `wwi`. Click **OK**.
9. In the next dialog, choose **Database** authentication and enter your Azure SQL Data Warehouse user name and password, and click **OK**.

10. In the **Navigator** dialog, select the checkboxes for **prd.CityDimensions**, **prd.DateDimensions**, and **prd.SalesFact**.

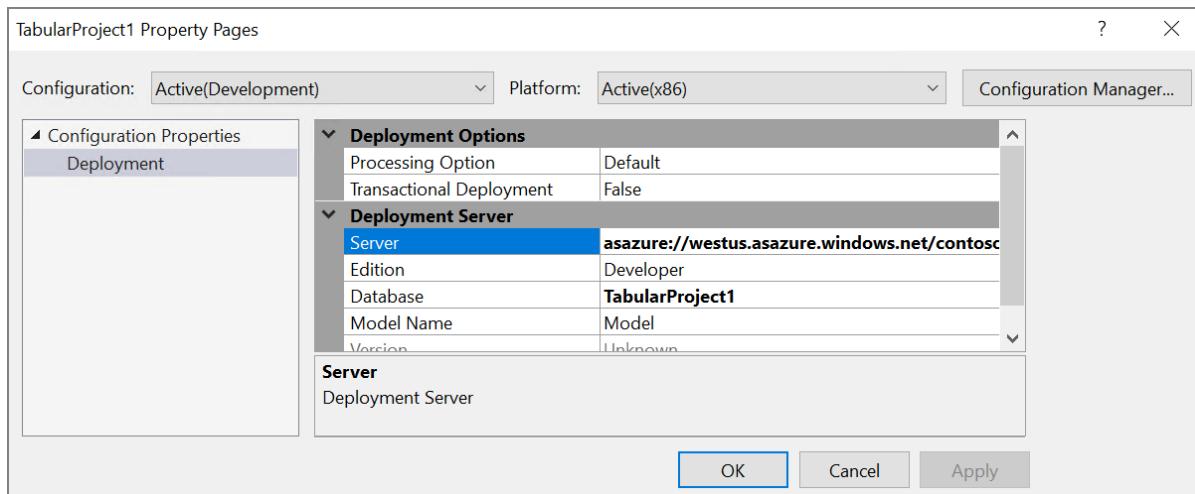
The screenshot shows the 'Navigator' dialog box. On the left, there is a list of tables under 'Display Options'. Several checkboxes are checked: 'prd.CityDimensions', 'prd.DateDimensions', and 'prd.SalesFact'. The 'prd.SalesFact' checkbox is highlighted with a yellow background. On the right, there is a preview of the 'prd.SalesFact' table with columns 'WWI City ID', 'WWI Customer ID', and 'WWI Bill To Customer ID'. The data preview shows approximately 20 rows of sales facts.

WWI City ID	WWI Customer ID	WWI Bill To Customer ID
37988	1049	
24590	977	
20412	466	
31266	552	
15689	1003	
9064	522	
9932	170	
17306	103	
372	892	
27648	457	
32083	184	
11491	536	
34285	585	
34422	964	
32688	119	
18772	144	
36513	569	

11. Click **Load**. When processing is complete, click **Close**. You should now see a tabular view of the data.
12. In the **Tabular Model Explorer** window, right-click the project and select **Model View > Diagram View**.
13. Drag the **[prd.SalesFact].[WWI City ID]** field to the **[prd.CityDimensions].[WWI City ID]** field to create a relationship.
14. Drag the **[prd.SalesFact].[Invoice Date Key]** field to the **[prd.DateDimensions].[Date]** field.



15. From the **File** menu, choose **Save All**.
16. In **Solution Explorer**, right-click the project and select **Properties**.
17. Under **Server**, enter the URL of your Azure Analysis Services instance. You can get this value from the Azure Portal. In the portal, select the Analysis Services resource, click the Overview pane, and look for the **Server Name** property. It will be similar to `asazure://westus.asazure.windows.net/contoso`. Click **OK**.



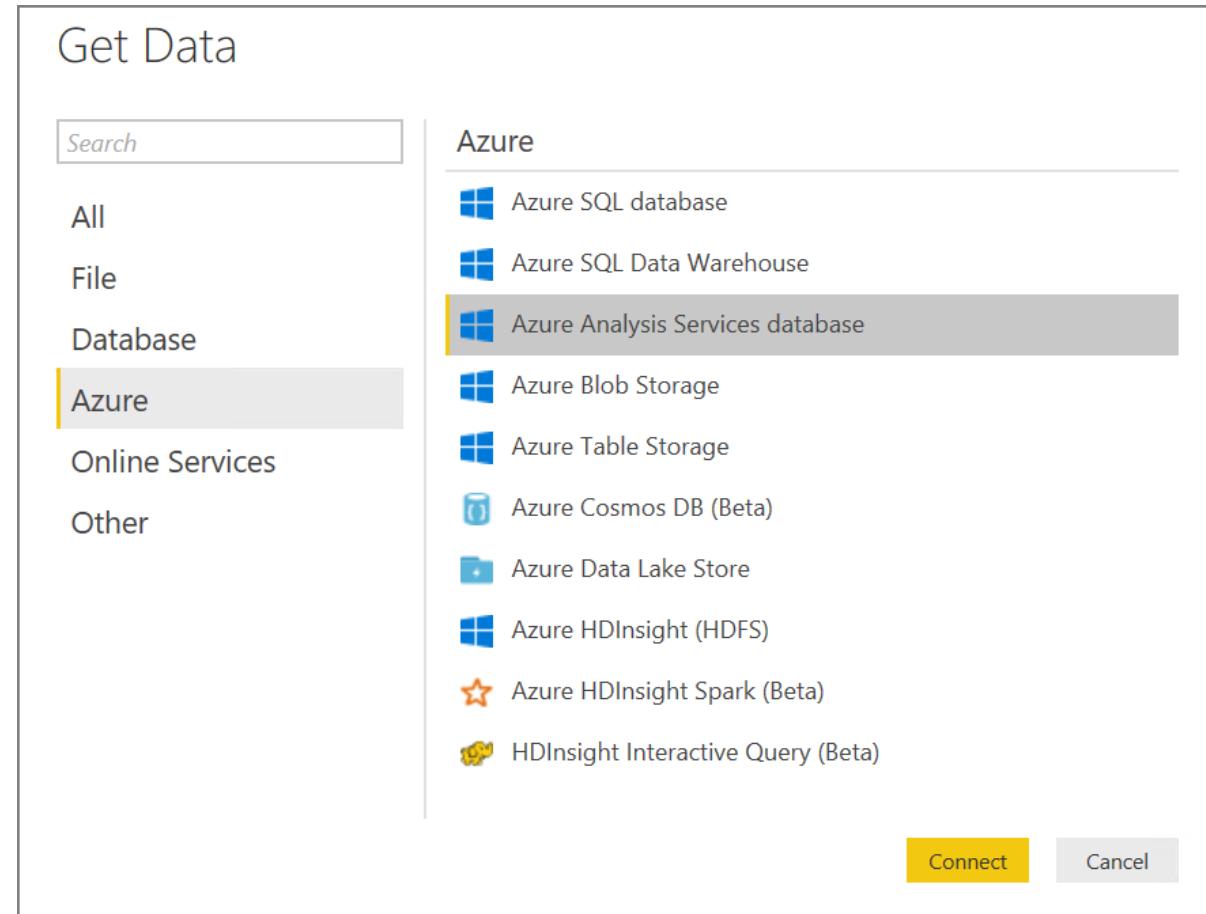
18. In **Solution Explorer**, right-click the project and select **Deploy**. Sign into Azure if prompted. When processing is complete, click **Close**.
19. In the Azure portal, view the details for your Azure Analysis Services instance. Verify that your model appears in the list of models.

Models on Analysis Services Server		
NAME	COMPATIBILITY	DATE MODIFIED
TabularProject1	1400	3/24/2018, 8:59 PM

## Analyze the data in Power BI Desktop

In this step, you will use Power BI to create a report from the data in Analysis Services.

1. From your Remote Desktop session, launch Power BI Desktop.
2. In the Welcome Screen, click **Get Data**.
3. Select **Azure > Azure Analysis Services database**. Click **Connect**



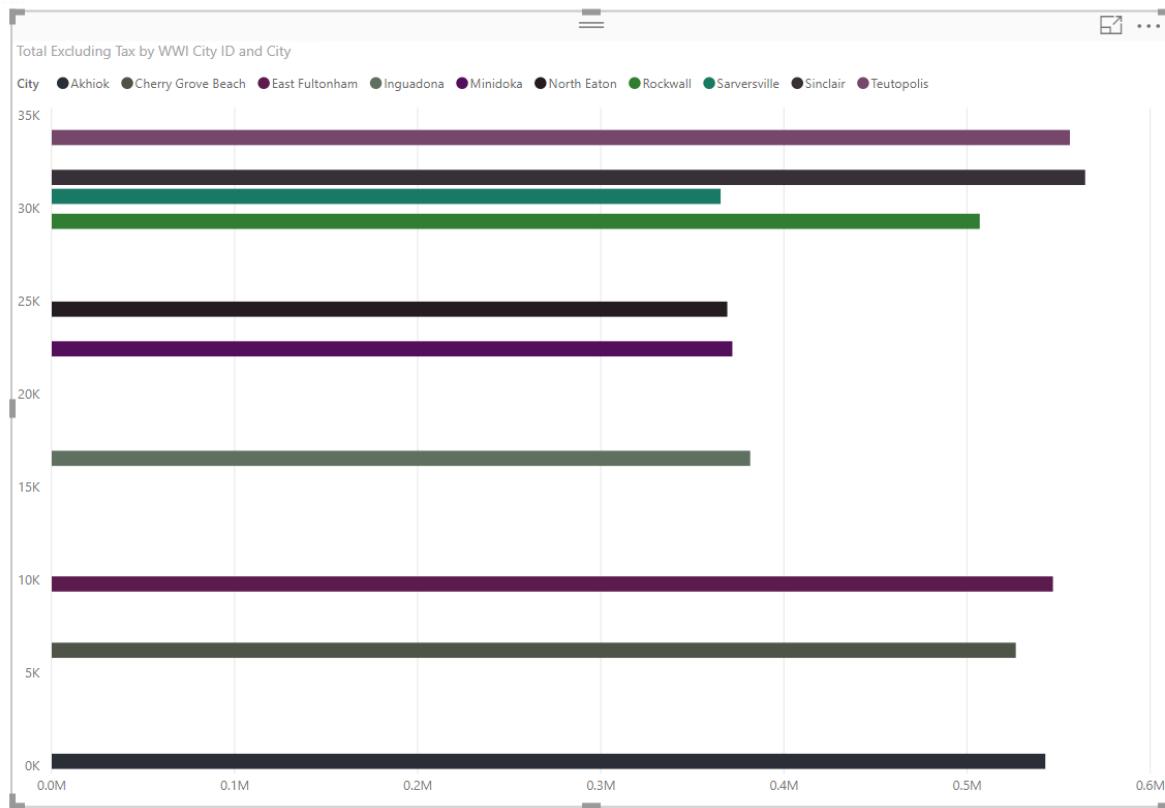
4. Enter the URL of your Analysis Services instance, then click **OK**. Sign into Azure if prompted.
5. In the **Navigator** dialog, expand the tabular project that you deployed, select the model that you created, and click **OK**.
6. In the **Visualizations** pane, select the **Stacked Bar Chart** icon. In the Report view, resize the visualization to make it larger.
7. In the **Fields** pane, expand **prd.CityDimensions**.
8. Drag **prd.CityDimensions > WWI City ID** to the **Axis** well.
9. Drag **prd.CityDimensions > City** to the **Legend** well.
10. In the **Fields** pane, expand **prd.SalesFact**.
11. Drag **prd.SalesFact > Total Excluding Tax** to the **Value** well.

The screenshot shows the Power BI Fields pane. At the top, there's a search bar labeled "Search". Below it, a tree view shows "prd CityDimensions" expanded, containing fields: City (checked), Continent, Country, Latest Recorded Pop..., Region, Sales Territory, State Province, Subregion, Valid From, Valid To, and WWI City ID (checked). Below the dimensions, "prd DateDimensions" is shown. On the left side of the Fields pane, there's a "Visualizations" section with various chart icons, an "Axis" section with "WWI City ID" selected, a "Legend" section with "City" selected, and a "Value" section with "Total Excluding Tax" selected.

12. Under **Visual Level Filters**, select **WWI City ID**.
13. Set the **Filter Type** to **Top N**, and set **Show Items** to **Top 10**.
14. Drag **prd.SalesFact > Total Excluding Tax** to the **By Value** well

The screenshot shows the Power BI Visual Level Filters pane. It includes sections for "FILTERS", "Visual level filters" (containing "City (All)", "Total Excluding Tax (All)", and "WWI City ID (All)" which has an upward arrow icon), "Filter Type" (set to "Top N"), "Show items:" (set to "Top 10"), and "By value" (containing "Total Excluding Tax"). To the right, a list of available measures is shown, with "Total Excluding Tax" checked. The list includes: Package, Profit, Quantity, Tax Amount, Tax Rate, Total Chiller Items, Total Dry Items, Total Excluding Tax (checked), Total Including Tax, Unit Price, WWI Bill To Custom..., WWI City ID, WWI Customer ID, WWI Invoice ID, WWI Saleperson ID, and WWI Stock Item ID.

15. Click **Apply Filter**. The visualization shows the top 10 total sales by city.



To learn more about Power BI Desktop, see [Getting started with Power BI Desktop](#).

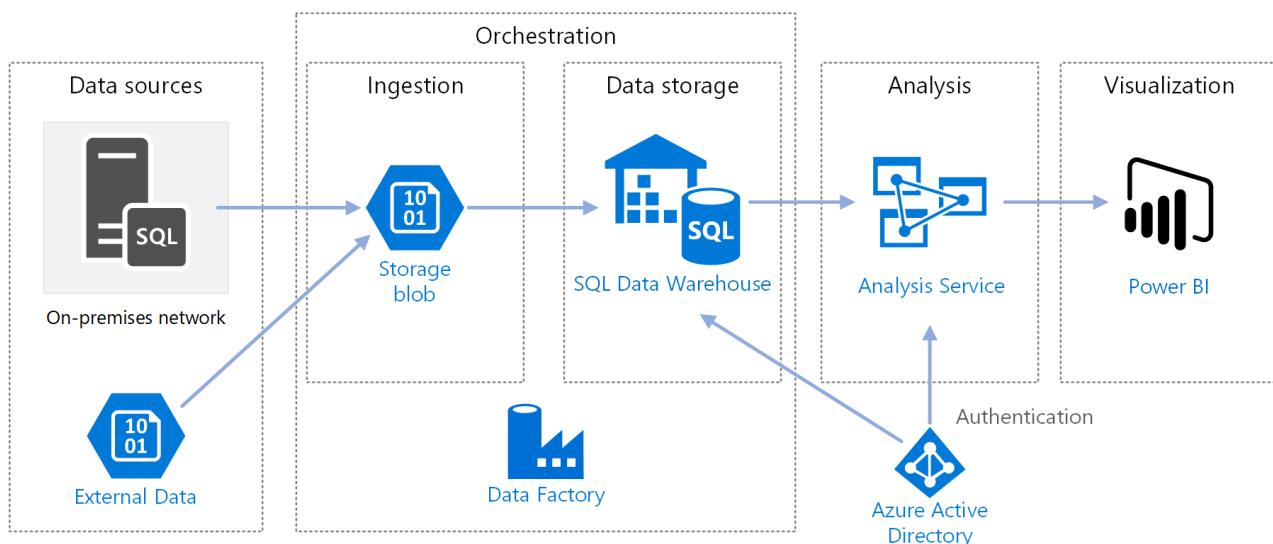
## Next steps

- For more information about this reference architecture, visit our [GitHub repository](#).
- Learn about the [Azure Building Blocks](#).

# Automated enterprise BI with SQL Data Warehouse and Azure Data Factory

10/9/2018 • 18 minutes to read • [Edit Online](#)

This reference architecture shows how to perform incremental loading in an **ELT** (extract-load-transform) pipeline. It uses Azure Data Factory to automate the ELT pipeline. The pipeline incrementally moves the latest OLTP data from an on-premises SQL Server database into SQL Data Warehouse. Transactional data is transformed into a tabular model for analysis. [Deploy this solution.](#)



This architecture builds on the one shown in [Enterprise BI with SQL Data Warehouse](#), but adds some features that are important for enterprise data warehousing scenarios.

- Automation of the pipeline using Data Factory.
- Incremental loading.
- Integrating multiple data sources.
- Loading binary data such as geospatial data and images.

## Architecture

The architecture consists of the following components.

### Data sources

**On-premises SQL Server.** The source data is located in a SQL Server database on premises. To simulate the on-premises environment, the deployment scripts for this architecture provision a virtual machine in Azure with SQL Server installed. The [Wide World Importers OLTP sample database](#) is used as the source database.

**External data.** A common scenario for data warehouses is to integrate multiple data sources. This reference architecture loads an external data set that contains city populations by year, and integrates it with the data from the OLTP database. You can use this data for insights such as: "Does sales growth in each region match or exceed population growth?"

### Ingestion and data storage

**Blob Storage.** Blob storage is used as a staging area for the source data before loading it into SQL Data Warehouse.

**Azure SQL Data Warehouse.** [SQL Data Warehouse](#) 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.

**Azure Data Factory.** [Data Factory](#) is a managed service that orchestrates and automates data movement and data transformation. In this architecture, it coordinates the various stages of the ELT process.

## Analysis and reporting

**Azure Analysis Services.** [Analysis Services](#) is a fully managed service that provides data modeling capabilities. The semantic model is loaded into Analysis Services.

**Power BI.** Power BI is a suite of business analytics tools to analyze data for business insights. In this architecture, it queries the semantic model stored in Analysis Services.

## Authentication

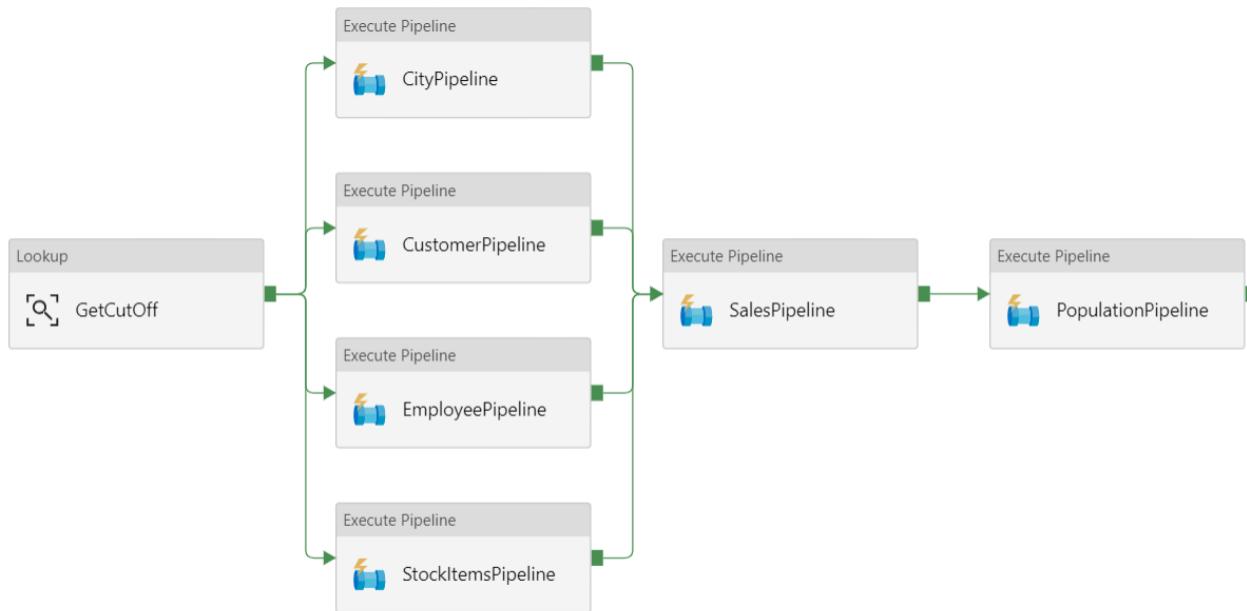
**Azure Active Directory** (Azure AD) authenticates users who connect to the Analysis Services server through Power BI.

Data Factory can use also use Azure AD to authenticate to SQL Data Warehouse, by using a service principal or Managed Service Identity (MSI). For simplicity, the example deployment uses SQL Server authentication.

## Data pipeline

In [Azure Data Factory](#), a pipeline is a logical grouping of activities used to coordinate a task — in this case, loading and transforming data into SQL Data Warehouse.

This reference architecture defines a master pipeline that runs a sequence of child pipelines. Each child pipeline loads data into one or more data warehouse tables.



## Incremental loading

When you run an automated ETL or ELT process, it's most efficient to load only the data that changed since the previous run. This is called an *incremental load*, as opposed to a full load that loads all of the data. To perform an incremental load, you need a way to identify which data has changed. The most common approach is to use a *high water mark* value, which means tracking the latest value of some column in the source table, either a datetime column or a unique integer column.

Starting with SQL Server 2016, you can use [temporal tables](#). These are system-versioned tables that keep a full history of data changes. The database engine automatically records the history of every change in a separate

history table. You can query the historical data by adding a FOR SYSTEM\_TIME clause to a query. Internally, the database engine queries the history table, but this is transparent to the application.

#### NOTE

For earlier versions of SQL Server, you can use [Change Data Capture](#) (CDC). This approach is less convenient than temporal tables, because you have to query a separate change table, and changes are tracked by a log sequence number, rather than a timestamp.

Temporal tables are useful for dimension data, which can change over time. Fact tables usually represent an immutable transaction such as a sale, in which case keeping the system version history doesn't make sense. Instead, transactions usually have a column that represents the transaction date, which can be used as the watermark value. For example, in the Wide World Importers OLTP database, the Sales.Invoices and Sales.InvoiceLines tables have a `LastEditedWhen` field that defaults to `sysdatetime()`.

Here is the general flow for the ELT pipeline:

1. For each table in the source database, track the cutoff time when the last ELT job ran. Store this information in the data warehouse. (On initial setup, all times are set to '1-1-1900'.)
2. During the data export step, the cutoff time is passed as a parameter to a set of stored procedures in the source database. These stored procedures query for any records that were changed or created after the cutoff time. For the Sales fact table, the `LastEditedWhen` column is used. For the dimension data, system-versioned temporal tables are used.
3. When the data migration is complete, update the table that stores the cutoff times.

It's also useful to record a *lineage* for each ELT run. For a given record, the lineage associates that record with the ELT run that produced the data. For each ETL run, a new lineage record is created for every table, showing the starting and ending load times. The lineage keys for each record are stored in the dimension and fact tables.

City Key	WWI City ID	City	State Province	Valid From	Valid To	Lineage Key
1	1091	Arden	New York	2013-01-01 00:02:00.0000000	2013-07-01 16:00:00.0000000	47
2	20189	Lucasville	Ohio	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
3	20731	Manhattan	New York	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
4	36176	Waxhaw	North Carolina	2014-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
5	29413	Rolling Fields	Kentucky	2013-01-01 00:05:00.0000000	2013-07-01 16:00:00.0000000	47
6	3359	Blue Ridge	Alabama	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
7	7780	Coupeville	Washington	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
8	2404	Belen	New Mexico	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
9	29733	Royal Palm Beach	Florida	2013-01-01 00:01:00.0000000	2013-07-01 16:00:00.0000000	47
10	17000	Jenkins	Minnesota	2013-01-01 00:02:00.0000000	2014-07-01 16:00:00.0000000	47

After a new batch of data is loaded into the warehouse, refresh the Analysis Services tabular model. See [Asynchronous refresh with the REST API](#).

## Data cleansing

Data cleansing should be part of the ELT process. In this reference architecture, one source of bad data is the city population table, where some cities have zero population, perhaps because no data was available. During processing, the ELT pipeline removes those cities from the city population table. Perform data cleansing on staging tables, rather than external tables.

Here is the stored procedure that removes the cities with zero population from the City Population table. (You can find the source file [here](#).)

```
DELETE FROM [Integration].[CityPopulation_Staging]
WHERE RowNumber in (SELECT DISTINCT RowNumber
FROM [Integration].[CityPopulation_Staging]
WHERE POPULATION = 0
GROUP BY RowNumber
HAVING COUNT(RowNumber) = 4)
```

## External data sources

Data warehouses often consolidate data from multiple sources. This reference architecture loads an external data source that contains demographics data. This dataset is available in Azure blob storage as part of the [WorldWideImportersDW](#) sample.

Azure Data Factory can copy directly from blob storage, using the [blob storage connector](#). However, the connector requires a connection string or a shared access signature, so it can't be used to copy a blob with public read access. As a workaround, you can use PolyBase to create an external table over Blob storage and then copy the external tables into SQL Data Warehouse.

## Handling large binary data

In the source database, the Cities table has a Location column that holds a [geography](#) spatial data type. SQL Data Warehouse doesn't support the **geography** type natively, so this field is converted to a **varbinary** type during loading. (See [Workarounds for unsupported data types](#).)

However, PolyBase supports a maximum column size of `varbinary(8000)`, which means some data could be truncated. A workaround for this problem is to break the data up into chunks during export, and then reassemble the chunks, as follows:

1. Create a temporary staging table for the Location column.
2. For each city, split the location data into 8000-byte chunks, resulting in 1 – N rows for each city.
3. To reassemble the chunks, use the T-SQL [PIVOT](#) operator to convert rows into columns and then concatenate the column values for each city.

The challenge is that each city will be split into a different number of rows, depending on the size of geography data. For the PIVOT operator to work, every city must have the same number of rows. To make this work, the T-SQL query (which you can view [here](#)) does some tricks to pad out the rows with blank values, so that every city has the same number of columns after the pivot. The resulting query turns out to be much faster than looping through the rows one at a time.

The same approach is used for image data.

## Slowly changing dimensions

Dimension data is relatively static, but it can change. For example, a product might get reassigned to a different product category. There are several approaches to handling slowly changing dimensions. A common technique, called [Type 2](#), is to add a new record whenever a dimension changes.

In order to implement the Type 2 approach, dimension tables need additional columns that specify the effective date range for a given record. Also, primary keys from the source database will be duplicated, so the dimension table must have an artificial primary key.

The following image shows the Dimension.City table. The `WWI_City_ID` column is the primary key from the source database. The `City_Key` column is an artificial key generated during the ETL pipeline. Also notice that the table has `Valid_From` and `Valid_To` columns, which define the range when each row was valid. Current values have a

Valid To equal to '9999-12-31'.

	City Key	WWI City ID	City	State Province	Valid From	Valid To	Lineage Key
1	1	1091	Arden	New York	2013-01-01 00:02:00.0000000	2013-07-01 16:00:00.0000000	47
2	2	20189	Lucasville	Ohio	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
3	3	20731	Manhattan	New York	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
4	4	36176	Waxhaw	North Carolina	2014-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
5	5	29413	Rolling Fields	Kentucky	2013-01-01 00:05:00.0000000	2013-07-01 16:00:00.0000000	47
6	6	3359	Blue Ridge	Alabama	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
7	7	7780	Coupeville	Washington	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
8	8	2404	Belen	New Mexico	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
9	9	29733	Royal Palm Beach	Florida	2013-01-01 00:01:00.0000000	2013-07-01 16:00:00.0000000	47
10	10	17000	Jenkins	Minnesota	2013-01-01 00:02:00.0000000	2014-07-01 16:00:00.0000000	47

The advantage of this approach is that it preserves historical data, which can be valuable for analysis. However, it also means there will be multiple rows for the same entity. For example, here are the records that match

WWI City ID = 28561:

	City Key	WWI City ID	City	State Province	Valid From	Valid To
1	25321	28561	Redmond	Washington	2013-01-01 00:00:00.0000000	2013-01-01 00:01:00.0000000
2	38479	28561	Redmond	Washington	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999
3	106171	28561	Redmond	Washington	2013-07-01 16:00:00.0000000	2015-07-01 16:00:00.0000000
4	81223	28561	Redmond	Washington	2013-01-01 00:01:00.0000000	2013-07-01 16:00:00.0000000

For each Sales fact, you want to associate that fact with a single row in City dimension table, corresponding to the invoice date. As part of the ETL process, create an additional column that

The following T-SQL query creates a temporary table that associates each invoice with the correct City Key from the City dimension table.

```
CREATE TABLE CityHolder
WITH (HEAP , DISTRIBUTION = HASH([WWI Invoice ID]))
AS
SELECT DISTINCT s1.[WWI Invoice ID] AS [WWI Invoice ID],
       c.[City Key] AS [City Key]
  FROM [Integration].[Sale_Staging] s1
 CROSS APPLY (
    SELECT TOP 1 [City Key]
      FROM [Dimension].[City]
     WHERE [WWI City ID] = s1.[WWI City ID]
       AND s1.[Last Modified When] > [Valid From]
       AND s1.[Last Modified When] <= [Valid To]
    ORDER BY [Valid From], [City Key] DESC
   ) c
```

This table is used to populate a column in the Sales fact table:

```
UPDATE [Integration].[Sale_Staging]
SET [Integration].[Sale_Staging].[WWI Customer ID] = CustomerHolder.[WWI Customer ID]
```

This column enables a Power BI query to find the correct City record for a given sales invoice.

## Security considerations

For additional security, you can use [Virtual Network service endpoints](#) to secure Azure service resources to only your virtual network. This fully removes public Internet access to those resources, allowing traffic only from your

virtual network.

With this approach, you create a VNet in Azure and then create private service endpoints for Azure services. Those services are then restricted to traffic from that virtual network. You can also reach them from your on-premises network through a gateway.

Be aware of the following limitations:

- At the time this reference architecture was created, VNet service endpoints are supported for Azure Storage and Azure SQL Data Warehouse, but not for Azure Analysis Service. Check the latest status [here](#).
- If service endpoints are enabled for Azure Storage, PolyBase cannot copy data from Storage into SQL Data Warehouse. There is a mitigation for this issue. For more information, see [Impact of using VNet Service Endpoints with Azure storage](#).
- To move data from on-premises into Azure Storage, you will need to whitelist public IP addresses from your on-premises or ExpressRoute. For details, see [Securing Azure services to virtual networks](#).
- To enable Analysis Services to read data from SQL Data Warehouse, deploy a Windows VM to the virtual network that contains the SQL Data Warehouse service endpoint. Install [Azure On-premises Data Gateway](#) on this VM. Then connect your Azure Analysis service to the data gateway.

## Deploy the solution

A deployment for this reference architecture is available on [GitHub](#). It deploys the following:

- A Windows VM to simulate an on-premises database server. It includes SQL Server 2017 and related tools, along with Power BI Desktop.
- An Azure storage account that provides Blob storage to hold data exported from the SQL Server database.
- An Azure SQL Data Warehouse instance.
- An Azure Analysis Services instance.
- Azure Data Factory and the Data Factory pipeline for the ELT job.

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Variables

The steps that follow include some user-defined variables. You will need to replace these with values that you define.

- <data\_factory\_name> . Data Factory name.
- <analysis\_server\_name> . Analysis Services server name.
- <active\_directory\_upn> . Your Azure Active Directory user principal name (UPN). For example, `user@contoso.com` .
- <data\_warehouse\_server\_name> . SQL Data Warehouse server name.

- <data\_warehouse\_password> . SQL Data Warehouse administrator password.
- <resource\_group\_name> . The name of the resource group.
- <region> . The Azure region where the resources will be deployed.
- <storage\_account\_name> . Storage account name. Must follow the [naming rules](#) for Storage accounts.
- <sql-db-password> . SQL Server login password.

## Deploy Azure Data Factory

1. Navigate to the `data\enterprise_bi_sqldw_advanced\azure\templates` folder of the [GitHub repository](#).
2. Run the following Azure CLI command to create a resource group.

```
az group create --name <resource_group_name> --location <region>
```

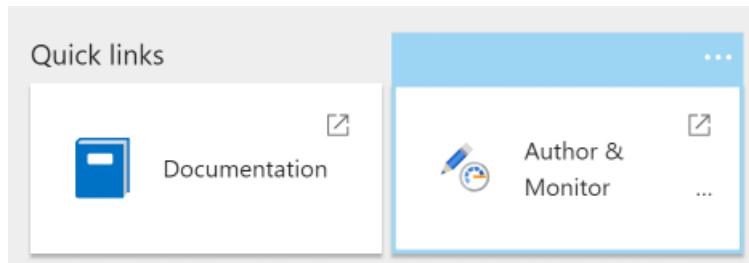
Specify a region that supports SQL Data Warehouse, Azure Analysis Services, and Data Factory v2. See [Azure Products by Region](#)

3. Run the following command

```
az group deployment create --resource-group <resource_group_name> \
--template-file adf-create-deploy.json \
--parameters factoryName=<data_factory_name> location=<location>
```

Next, use the Azure Portal to get the authentication key for the Azure Data Factory [integration runtime](#), as follows:

1. In the [Azure Portal](#), navigate to the Data Factory instance.
2. In the Data Factory blade, click **Author & Monitor**. This opens the Azure Data Factory portal in another browser window.



3. In the Azure Data Factory portal, select the pencil icon ("Author").
4. Click **Connections**, and then select **Integration Runtimes**.
5. Under **sourceIntegrationRuntime**, click the pencil icon ("Edit").

### NOTE

The portal will show the status as "unavailable". This is expected until you deploy the on-premises server.

6. Find **Key1** and copy the value of the authentication key.

You will need the authentication key for the next step.

## Deploy the simulated on-premises server

This step deploys a VM as a simulated on-premises server, which includes SQL Server 2017 and related tools. It also loads the [Wide World Importers OLTP database](#) into SQL Server.

1. Navigate to the `data\enterprise_bi_sqldw_advanced\onprem\templates` folder of the repository.
2. In the `onprem.parameters.json` file, search for `adminPassword`. This is the password to log into the SQL Server VM. Replace the value with another password.
3. In the same file, search for `SqlUserCredentials`. This property specifies the SQL Server account credentials. Replace the password with a different value.
4. In the same file, paste the Integration Runtime authentication key into the `IntegrationRuntimeGatewayKey` parameter, as shown below:

```
"protectedSettings": {
  "configurationArguments": {
    "SqlUserCredentials": {
      "userName": ".\\adminUser",
      "password": "<sql-db-password>"
    },
    "IntegrationRuntimeGatewayKey": "<authentication key>"
  }
}
```

5. Run the following command.

```
azbb -s <subscription_id> -g <resource_group_name> -l <region> -p onprem.parameters.json --deploy
```

This step may take 20 to 30 minutes to complete. It includes running a [DSC](#) script to install the tools and restore the database.

## Deploy Azure resources

This step provisions SQL Data Warehouse, Azure Analysis Services, and Data Factory.

1. Navigate to the `data\enterprise_bi_sqldw_advanced\azure\templates` folder of the [GitHub repository](#).
2. Run the following Azure CLI command. Replace the parameter values shown in angle brackets.

```
az group deployment create --resource-group <resource_group_name> \
--template-file azure-resources-deploy.json \
--parameters "dwServerName"<data_warehouse_server_name>" \
"dwAdminLogin"="adminuser" "dwAdminPassword"=<data_warehouse_password>" \
"storageAccountName"=<storage_account_name>" \
"analysisServerName"=<analysis_server_name>" \
"analysisServerAdmin"=<user@contoso.com>"
```

- The `storageAccountName` parameter must follow the [naming rules](#) for Storage accounts.
  - For the `analysisServerAdmin` parameter, use your Azure Active Directory user principal name (UPN).
3. Run the following Azure CLI command to get the access key for the storage account. You will use this key in the next step.

```
az storage account keys list -n <storage_account_name> -g <resource_group_name> --query [0].value
```

4. Run the following Azure CLI command. Replace the parameter values shown in angle brackets.

```
az group deployment create --resource-group <resource_group_name> \
--template-file adf-pipeline-deploy.json \
--parameters "factoryName"<data_factory_name>" \
"sinkDWConnectionString"="Server=tcp:<data_warehouse_server_name>.database.windows.net,1433;Initial
Catalog=wwi;Persist Security Info=False;User ID=adminuser;Password=
<data_warehouse_password>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Conn
ection Timeout=30;" \
"blobConnectionString"="DefaultEndpointsProtocol=https;AccountName=<storage_account_name>;AccountKey=
<storage_account_key>;EndpointSuffix=core.windows.net" \
"sourceDBConnectionString"="Server=sql1;Database=WideWorldImporters;User Id=adminuser;Password=<sql-db-
password>;Trusted_Connection=True;"
```

The connection strings have substrings shown in angle brackets that must be replaced. For `<storage_account_key>`, use the key that you got in the previous step. For `<sql-db-password>`, use the SQL Server account password that you specified in the `onprem.parameters.json` file previously.

### Run the data warehouse scripts

1. In the [Azure Portal](#), find the on-premises VM, which is named `sql-vm1`. The user name and password for the VM are specified in the `onprem.parameters.json` file.
2. Click **Connect** and use Remote Desktop to connect to the VM.
3. From your Remote Desktop session, open a command prompt and navigate to the following folder on the VM:

```
cd C:\SampleDataFiles\reference-architectures\data\enterprise_bi_sqldw_advanced\azure\sqldw_scripts
```

4. Run the following command:

```
deploy_database.cmd -S <data_warehouse_server_name>.database.windows.net -d wwi -U adminuser -P
<data_warehouse_password> -N -I
```

For `<data_warehouse_server_name>` and `<data_warehouse_password>`, use the data warehouse server name and password from earlier.

To verify this step, you can use SQL Server Management Studio (SSMS) to connect to the SQL Data Warehouse database. You should see the database table schemas.

### Run the Data Factory pipeline

1. From the same Remote Desktop session, open a PowerShell window.
2. Run the following PowerShell command. Choose **Yes** when prompted.

```
Install-Module -Name AzureRM -AllowClobber
```

3. Run the following PowerShell command. Enter your Azure credentials when prompted.

```
Connect-AzureRmAccount
```

4. Run the following PowerShell commands. Replace the values in angle brackets.

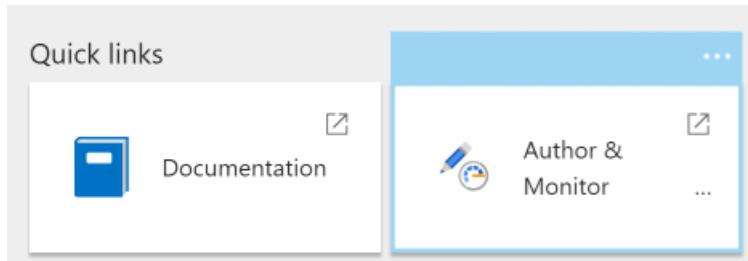
```

Set-AzureRmContext -SubscriptionId <subscription id>

Invoke-AzureRmDataFactoryV2Pipeline -DataFactory <data-factory-name> -PipelineName "MasterPipeline" -
ResourceGroupName <resource_group_name>

```

5. In the Azure Portal, navigate to the Data Factory instance that was created earlier.
6. In the Data Factory blade, click **Author & Monitor**. This opens the Azure Data Factory portal in another browser window.



7. In the Azure Data Factory portal, click the **Monitor** icon.
8. Verify that the pipeline completes successfully. It can take a few minutes.

<input type="checkbox"/>	Pipeline Name	Actions	Run Start	Duration	Triggered By	Status
	EmployeePipeline		05/30/2018, 5:21:35 PM	00:01:32	PipelineActivity	Succeeded
	CityPipeline		05/30/2018, 5:21:35 PM	00:00:00	PipelineActivity	In Progress
	CustomerPipeline		05/30/2018, 5:21:35 PM	00:00:00	PipelineActivity	In Progress
	StockItemsPipeline		05/30/2018, 5:21:35 PM	00:00:00	PipelineActivity	In Progress
	CityPopulationPipeline		05/30/2018, 5:21:17 PM	00:00:34	Manual trigger	Succeeded
	MasterPipeline		05/30/2018, 5:21:11 PM	00:00:00	Manual trigger	In Progress

## Build the Analysis Services model

In this step, you will create a tabular model that imports data from the data warehouse. Then you will deploy the model to Azure Analysis Services.

### Create a new tabular project

1. From your Remote Desktop session, launch SQL Server Data Tools 2015.
2. Select **File > New > Project**.
3. In the **New Project** dialog, under **Templates**, select **Business Intelligence > Analysis Services > Analysis Services Tabular Project**.
4. Name the project and click **OK**.
5. In the **Tabular model designer** dialog, select **Integrated workspace** and set **Compatibility level** to **SQL Server 2017 / Azure Analysis Services (1400)**.
6. Click **OK**.

### Import data

1. In the **Tabular Model Explorer** window, right-click the project and select **Import from Data Source**.
2. Select **Azure SQL Data Warehouse** and click **Connect**.
3. For **Server**, enter the fully qualified name of your Azure SQL Data Warehouse server. You can get this value from the Azure Portal. For **Database**, enter `wwi`. Click **OK**.
4. In the next dialog, choose **Database** authentication and enter your Azure SQL Data Warehouse user name and password, and click **OK**.
5. In the **Navigator** dialog, select the checkboxes for the **Fact.\*** and **Dimension.\*** tables.

The screenshot shows the 'Navigator' dialog box. On the left, there is a list of tables with checkboxes. The 'Dimension.StockItem' checkbox is selected and highlighted with a yellow background. On the right, there is a grid titled 'Dimension.StockItem' displaying data with columns: Stock Item Key, WWI Stock Item ID, and Stock Item. The data includes rows such as 4 (Developer j), 10 (Void fill 400), 12 (DBA joke m), etc.

Stock Item Key	WWI Stock Item ID	Stock Item
4	45	Developer j
10	219	Void fill 400
12	19	DBA joke m
22	118	Dinosaur ba
35	27	DBA joke m
51	152	Pack of 12 a
64	46	Developer j
70	218	Void fill 300
72	18	DBA joke m
82	117	Superhero a
95	26	DBA joke m

6. Click **Load**. When processing is complete, click **Close**. You should now see a tabular view of the data.

### Create measures

1. In the model designer, select the **Fact Sale** table.
2. Click a cell in the the measure grid. By default, the measure grid is displayed below the table.

The screenshot shows the 'Model.bim' window. At the top, there is a formula bar with a dropdown for 'Invoice Date...'. Below it is a table titled 'Fact Sale' with columns: Invoice Date Key, Delivery Date Key, WWI Invoice ID, Description, Package, Quantity, and Unit Price. There are three rows of data. At the bottom, there is a navigation bar with tabs for 'Fact Sale', 'Fact CityPopulation', 'Dimension City', 'Dimension Customer', 'Dimension Date', 'Dimension Employee', and a record counter 'Record: 1 of 228,265'.

3. In the formula bar, enter the following and press ENTER:

```
Total Sales:=SUM('Fact Sale'[Total Including Tax])
```

4. Repeat these steps to create the following measures:

```
Number of Years:=(MAX('Fact CityPopulation'[YearNumber])-MIN('Fact CityPopulation'[YearNumber]))+1
```

```
Beginning Population:=CALCULATE(SUM('Fact CityPopulation'[Population]),FILTER('Fact CityPopulation','Fact CityPopulation'[YearNumber]=MIN('Fact CityPopulation'[YearNumber])))
```

```
Ending Population:=CALCULATE(SUM('Fact CityPopulation'[Population]),FILTER('Fact CityPopulation','Fact CityPopulation'[YearNumber]=MAX('Fact CityPopulation'[YearNumber])))
```

```
CAGR:=IFERROR(((Ending Population)/Beginning Population)^(1/Number of Years))-1,0)
```

Model.bim*			
[WWI Bill To ...	f(x)	Total Sales:=SUM([Total Including Tax])	
WWI Bill To Customer ID	WWI Stock Item ID	WWI Saleperson ID	
1	1	33	13
2	1	40	13
3	1	18	13
4	1	96	13
5	1	181	13
6	1	46	13

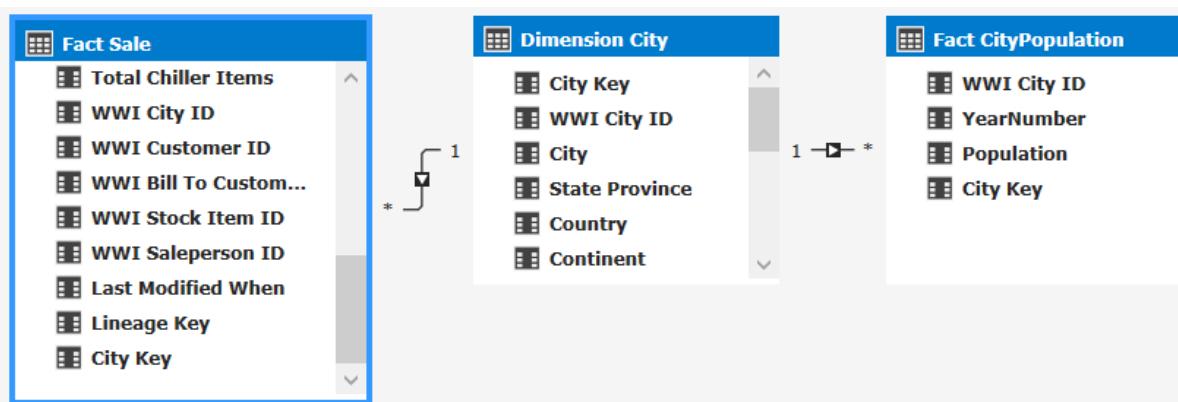
  

Total Sales: 198066178
Number of Years: 4
Beginning Population:
180936628
Ending Population:
227341818
CAGR:
0.0587372883826314

For more information about creating measures in SQL Server Data Tools, see [Measures](#).

## Create relationships

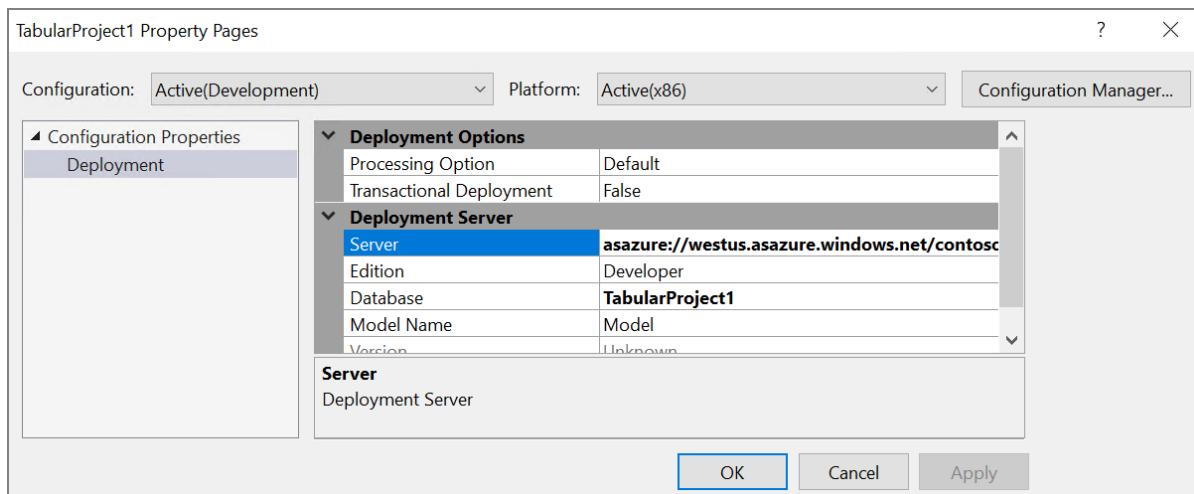
1. In the **Tabular Model Explorer** window, right-click the project and select **Model View > Diagram View**.
2. Drag the **[Fact Sale].[City Key]** field to the **[Dimension City].[City Key]** field to create a relationship.
3. Drag the **[Fact CityPopulation].[City Key]** field to the **[Dimension City].[City Key]** field.



## Deploy the model

1. From the **File** menu, choose **Save All**.
2. In **Solution Explorer**, right-click the project and select **Properties**.
3. Under **Server**, enter the URL of your Azure Analysis Services instance. You can get this value from the

Azure Portal. In the portal, select the Analysis Services resource, click the Overview pane, and look for the **Server Name** property. It will be similar to `asazure://westus.asazure.windows.net/contoso`. Click **OK**.



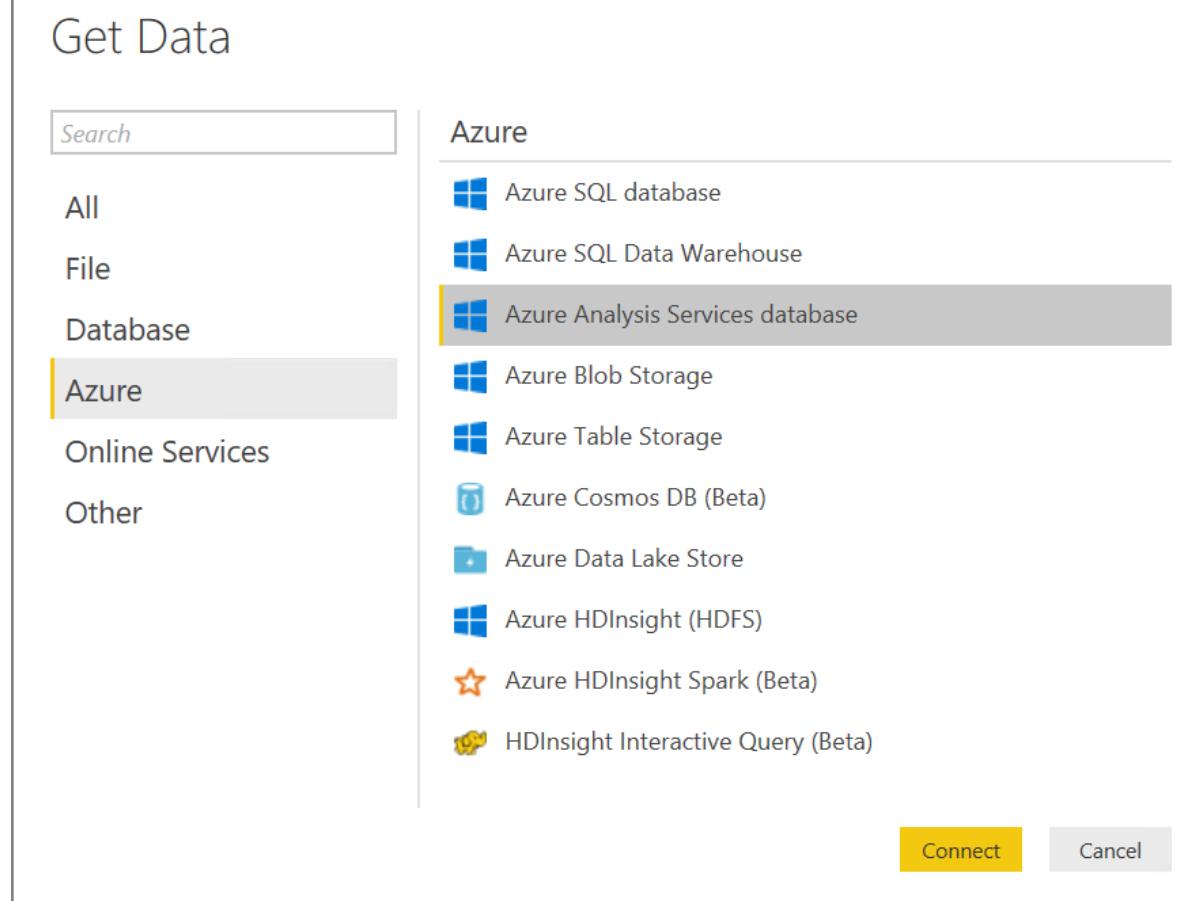
4. In **Solution Explorer**, right-click the project and select **Deploy**. Sign into Azure if prompted. When processing is complete, click **Close**.
5. In the Azure portal, view the details for your Azure Analysis Services instance. Verify that your model appears in the list of models.

Models on Analysis Services Server		
NAME	COMPATIBILITY	DATE MODIFIED
TabularProject1	1400	3/24/2018, 8:59 PM

## Analyze the data in Power BI Desktop

In this step, you will use Power BI to create a report from the data in Analysis Services.

1. From your Remote Desktop session, launch Power BI Desktop.
2. In the Welcome Screen, click **Get Data**.
3. Select **Azure > Azure Analysis Services database**. Click **Connect**



4. Enter the URL of your Analysis Services instance, then click **OK**. Sign into Azure if prompted.
5. In the **Navigator** dialog, expand the tabular project, select the model, and click **OK**.
6. In the **Visualizations** pane, select the **Table** icon. In the Report view, resize the visualization to make it larger.
7. In the **Fields** pane, expand **Dimension City**.
8. From **Dimension City**, drag **City** and **State Province** to the **Values** well.
9. In the **Fields** pane, expand **Fact Sale**.
10. From **Fact Sale**, drag **CAGR**, **Ending Population**, and **Total Sales** to the **Value** well.
11. Under **Visual Level Filters**, select **Ending Population**. Set the filter to "is greater than 100000" and click **Apply filter**.
12. Under **Visual Level Filters**, select **Total Sales**. Set the filter to "is 0" and click **Apply filter**.

[Back to Report](#)

City	State Province	CAGR	Ending Population
Abilene	Texas	0.06	117063
Akron	Ohio	0.06	199110
Albuquerque	New Mexico	0.06	545852
Alexandria	Virginia	0.06	139966
Allentown	Pennsylvania	0.06	118032
Amarillo	Texas	0.06	190695
Anaheim	California	0.06	336265
Anchorage	Alaska	0.06	291826
Ann Arbor	Michigan	0.06	113934
Antioch	California	0.06	102372
Arlington	Texas	0.06	365438
Arlington	Virginia	0.06	207627
Arvada	Colorado	0.06	106433
Atlanta	Georgia	0.06	420003
Aurora	Colorado	0.06	325078
Aurora	Illinois	0.06	197899
Austin	Texas	0.06	790390
Bakersfield	California	0.06	347483
Baltimore	Maryland	0.06	620961
Baton Rouge	Louisiana	0.06	229493
Bayamón	Puerto Rico (US Territory)	0.06	185996
Beaumont	Texas	0.06	118296
Bellevue	Washington	0.06	122363
Berkeley	California	0.06	112580
Billings	Montana	0.06	104170
Birmingham	Alabama	0.06	212237
Boise	Idaho	0.06	205671
Boise City	Idaho	0.06	205671
Boston	Massachusetts[E]	0.06	617594
Brandon	Florida	0.06	103483
Bridgeport	Connecticut	0.06	144229
<b>Total</b>		<b>0.06</b>	<b>93158546</b>

**VISUALIZATIONS**

**VALUES**

- City
- State Province
- CAGR
- Ending Population

**FILTERS**

Visual level filters

- CAGR (All)
- City (All)
- Ending Population  
is greater than 100000
- State Province (All)
- Total Sales  
is 0

Page level filters

**FIELDS**

Search

- Dimension City**
  - City
  - City Key
  - Continent
  - Country
  - Latest Recorded Population
  - Lineage Key
  - Region
  - Sales Territory
  - State Province
  - StateProvinceCode
  - Subregion
  - Valid From
  - Valid To
  - WWI City ID
- Dimension Customer**
- Dimension Date**
- Dimension Employee**
- Dimension StockItem**
  - Barcode
  - Brand
  - Buying Package
  - Color
  - Is Chiller Stock

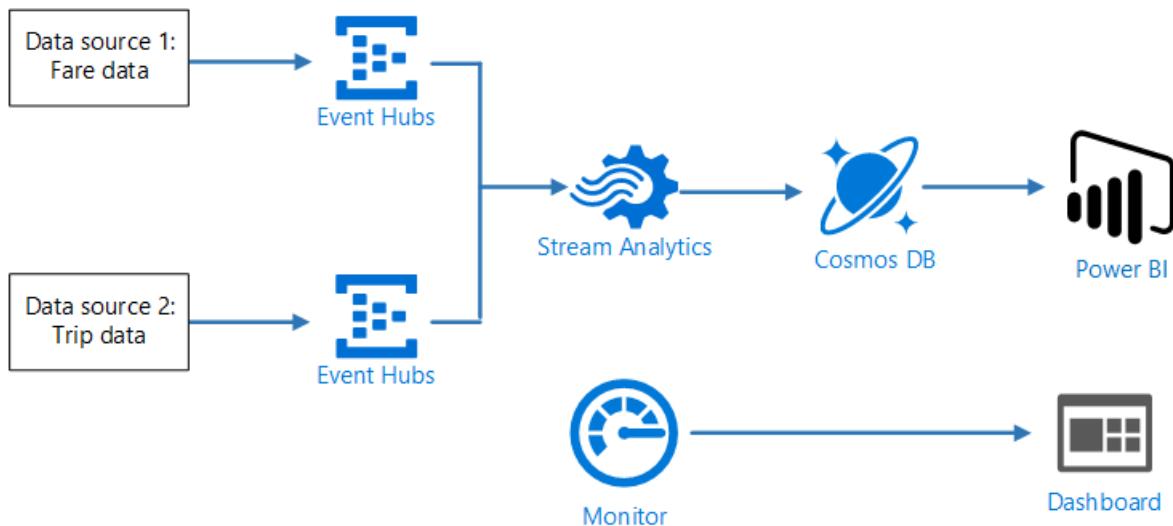
The table now shows cities with population greater than 100,000 and zero sales. CAGR stands for Compounded Annual Growth Rate and measures the rate of population growth per city. You could use this value to find cities with high growth rates, for example. However, note that the values for CAGR in the model aren't accurate, because they are derived from sample data.

To learn more about Power BI Desktop, see [Getting started with Power BI Desktop](#).

# Stream processing with Azure Stream Analytics

8/9/2018 • 11 minutes to read • [Edit Online](#)

This reference architecture shows an end-to-end stream processing pipeline. The pipeline ingests data from two sources, correlates records in the two streams, and calculates a rolling average across a time window. The results are stored for further analysis. [Deploy this solution](#).



**Scenario:** A taxi company collects data about each taxi trip. For this scenario, we assume there are two separate devices sending data. The taxi has a meter that sends information about each ride — the duration, distance, and pickup and dropoff locations. A separate device accepts payments from customers and sends data about fares. The taxi company wants to calculate the average tip per mile driven, in real time, in order to spot trends.

## Architecture

The architecture consists of the following components.

**Data sources.** In this architecture, there are two data sources that generate data streams in real time. The first stream contains ride information, and the second contains fare information. The reference architecture includes a simulated data generator that reads from a set of static files and pushes the data to Event Hubs. In a real application, the data sources would be devices installed in the taxi cabs.

**Azure Event Hubs.** [Event Hubs](#) is an event ingestion service. This architecture uses two event hub instances, one for each data source. Each data source sends a stream of data to the associated event hub.

**Azure Stream Analytics.** [Stream Analytics](#) is an event-processing engine. A Stream Analytics job reads the data streams from the two event hubs and performs stream processing.

**Cosmos DB.** The output from the Stream Analytics job is a series of records, which are written as JSON documents to a Cosmos DB document database.

**Microsoft Power BI.** Power BI is a suite of business analytics tools to analyze data for business insights. In this architecture, it loads the data from Cosmos DB. This allows users to analyze the complete set of historical data that's been collected. You could also stream the results directly from Stream Analytics to Power BI for a real-time view of the data. For more information, see [Real-time streaming in Power BI](#).

**Azure Monitor.** [Azure Monitor](#) collects performance metrics about the Azure services deployed in the solution. By visualizing these in a dashboard, you can get insights into the health of the solution.

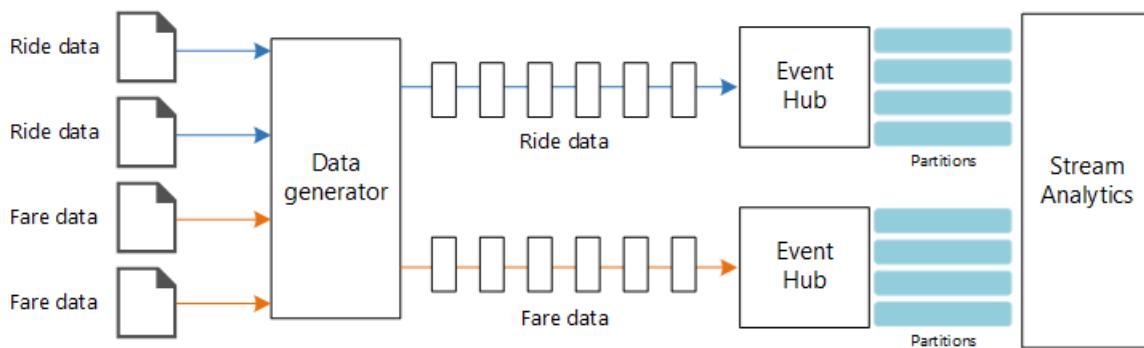
# Data ingestion

To simulate a data source, this reference architecture uses the [New York City Taxi Data](#) dataset<sup>[1]</sup>. This dataset contains data about taxi trips in New York City over a 4-year period (2010 – 2013). It contains two types of record: Ride data and fare data. Ride data includes trip duration, trip distance, and pickup and dropoff location. Fare data includes fare, tax, and tip amounts. Common fields in both record types include medallion number, hack license, and vendor ID. Together these three fields uniquely identify a taxi plus a driver. The data is stored in CSV format.

The data generator is a .NET Core application that reads the records and sends them to Azure Event Hubs. The generator sends ride data in JSON format and fare data in CSV format.

Event Hubs uses [partitions](#) to segment the data. Partitions allow a consumer to read each partition in parallel. When you send data to Event Hubs, you can specify the partition key explicitly. Otherwise, records are assigned to partitions in round-robin fashion.

In this particular scenario, ride data and fare data should end up with the same partition ID for a given taxi cab. This enables Stream Analytics to apply a degree of parallelism when it correlates the two streams. A record in partition  $n$  of the ride data will match a record in partition  $n$  of the fare data.



In the data generator, the common data model for both record types has a `PartitionKey` property which is the concatenation of `Medallion`, `HackLicense`, and `VendorId`.

```
public abstract class TaxiData
{
    public TaxiData()
    {
    }

    [JsonProperty]
    public long Medallion { get; set; }

    [JsonProperty]
    public long HackLicense { get; set; }

    [JsonProperty]
    public string VendorId { get; set; }

    [JsonProperty]
    public DateTimeOffset PickupTime { get; set; }

    [JsonIgnore]
    public string PartitionKey
    {
        get => $"{Medallion}_{HackLicense}_{VendorId}";
    }
}
```

This property is used to provide an explicit partition key when sending to Event Hubs:

```

using (var client = pool.GetObject())
{
    return client.Value.SendAsync(new EventData(Encoding.UTF8.GetBytes(
        t.GetData(dataFormat))), t.PartitionKey);
}

```

## Stream processing

The stream processing job is defined using a SQL query with several distinct steps. The first two steps simply select records from the two input streams.

```

WITH
Step1 AS (
    SELECT PartitionId,
        TRY_CAST(Medallion AS nvarchar(max)) AS Medallion,
        TRY_CAST(HackLicense AS nvarchar(max)) AS HackLicense,
        VendorId,
        TRY_CAST(PickupTime AS datetime) AS PickupTime,
        TripDistanceInMiles
    FROM [TaxiRide] PARTITION BY PartitionId
),
Step2 AS (
    SELECT PartitionId,
        medallion AS Medallion,
        hack_license AS HackLicense,
        vendor_id AS VendorId,
        TRY_CAST(pickup_datetime AS datetime) AS PickupTime,
        tip_amount AS TipAmount
    FROM [TaxiFare] PARTITION BY PartitionId
),

```

The next step joins the two input streams to select matching records from each stream.

```

Step3 AS (
    SELECT
        tr.Medallion,
        tr.HackLicense,
        tr.VendorId,
        tr.PickupTime,
        tr.TripDistanceInMiles,
        tf.TipAmount
    FROM [Step1] tr
    PARTITION BY PartitionId
    JOIN [Step2] tf PARTITION BY PartitionId
        ON tr.Medallion = tf.Medallion
        AND tr.HackLicense = tf.HackLicense
        AND tr.VendorId = tf.VendorId
        AND tr.PickupTime = tf.PickupTime
        AND tr.PartitionId = tf.PartitionId
        AND DATEDIFF(minute, tr, tf) BETWEEN 0 AND 15
)

```

This query joins records on a set of fields that uniquely identify matching records (Medallion, HackLicense, VendorId, and PickupTime). The `JOIN` statement also includes the partition ID. As mentioned, this takes advantage of the fact that matching records always have the same partition ID in this scenario.

In Stream Analytics, joins are *temporal*, meaning records are joined within a particular window of time. Otherwise, the job might need to wait indefinitely for a match. The `DATEDIFF` function specifies how far two matching records can be separated in time for a match.

The last step in the job computes the average tip per mile, grouped by a hopping window of 5 minutes.

```
SELECT System.Timestamp AS WindowTime,
       SUM(tr.TipAmount) / SUM(tr.TripDistanceInMiles) AS AverageTipPerMile
    INTO [TaxiDrain]
   FROM [Step3] tr
GROUP BY HoppingWindow(Duration(minute, 5), Hop(minute, 1))
```

Stream Analytics provides several [windowing functions](#). A hopping window moves forward in time by a fixed period, in this case 1 minute per hop. The result is to calculate a moving average over the past 5 minutes.

In the architecture shown here, only the results of the Stream Analytics job are saved to Cosmos DB. For a big data scenario, consider also using [Event Hubs Capture](#) to save the raw event data into Azure Blob storage. Keeping the raw data will allow you to run batch queries over your historical data at later time, in order to derive new insights from the data.

## Scalability considerations

### Event Hubs

The throughput capacity of Event Hubs is measured in [throughput units](#). You can autoscale an event hub by enabling [auto-inflate](#), which automatically scales the throughput units based on traffic, up to a configured maximum.

### Stream Analytics

For Stream Analytics, the computing resources allocated to a job are measured in Streaming Units. Stream Analytics jobs scale best if the job can be parallelized. That way, Stream Analytics can distribute the job across multiple compute nodes.

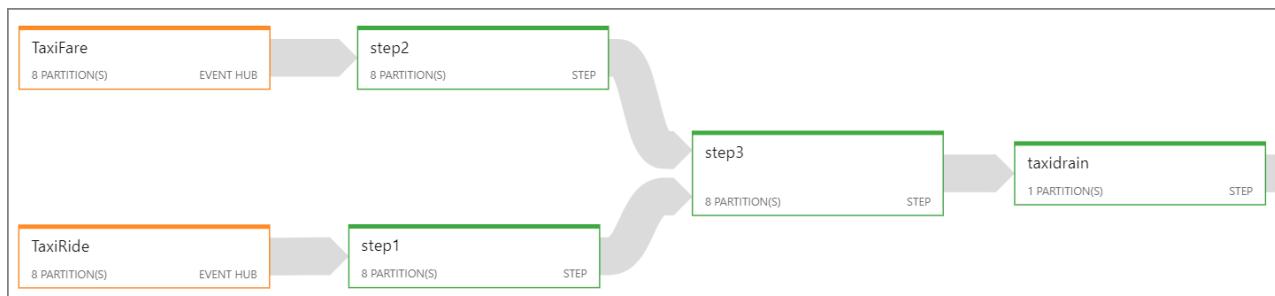
For Event Hubs input, use the `PARTITION BY` keyword to partition the Stream Analytics job. The data will be divided into subsets based on the Event Hubs partitions.

Windowing functions and temporal joins require additional SU. When possible, use `PARTITION BY` so that each partition is processed separately. For more information, see [Understand and adjust Streaming Units](#).

If it's not possible to parallelize the entire Stream Analytics job, try to break the job into multiple steps, starting with one or more parallel steps. That way, the first steps can run in parallel. For example, in this reference architecture:

- Steps 1 and 2 are simple `SELECT` statements that select records within a single partition.
- Step 3 performs a partitioned join across two input streams. This step takes advantage of the fact that matching records share the same partition key, and so are guaranteed to have the same partition ID in each input stream.
- Step 4 aggregates across all of the partitions. This step cannot be parallelized.

Use the Stream Analytics [job diagram](#) to see how many partitions are assigned to each step in the job. The following diagram shows the job diagram for this reference architecture:



### Cosmos DB

Throughput capacity for Cosmos DB is measured in [Request Units](#) (RU). In order to scale a Cosmos DB container past 10,000 RU, you must specify a [partition key](#) when you create the container, and include the partition key in every document.

In this reference architecture, new documents are created only once per minute (the hopping window interval), so the throughput requirements are quite low. For that reason, there's no need to assign a partition key in this scenario.

## Monitoring considerations

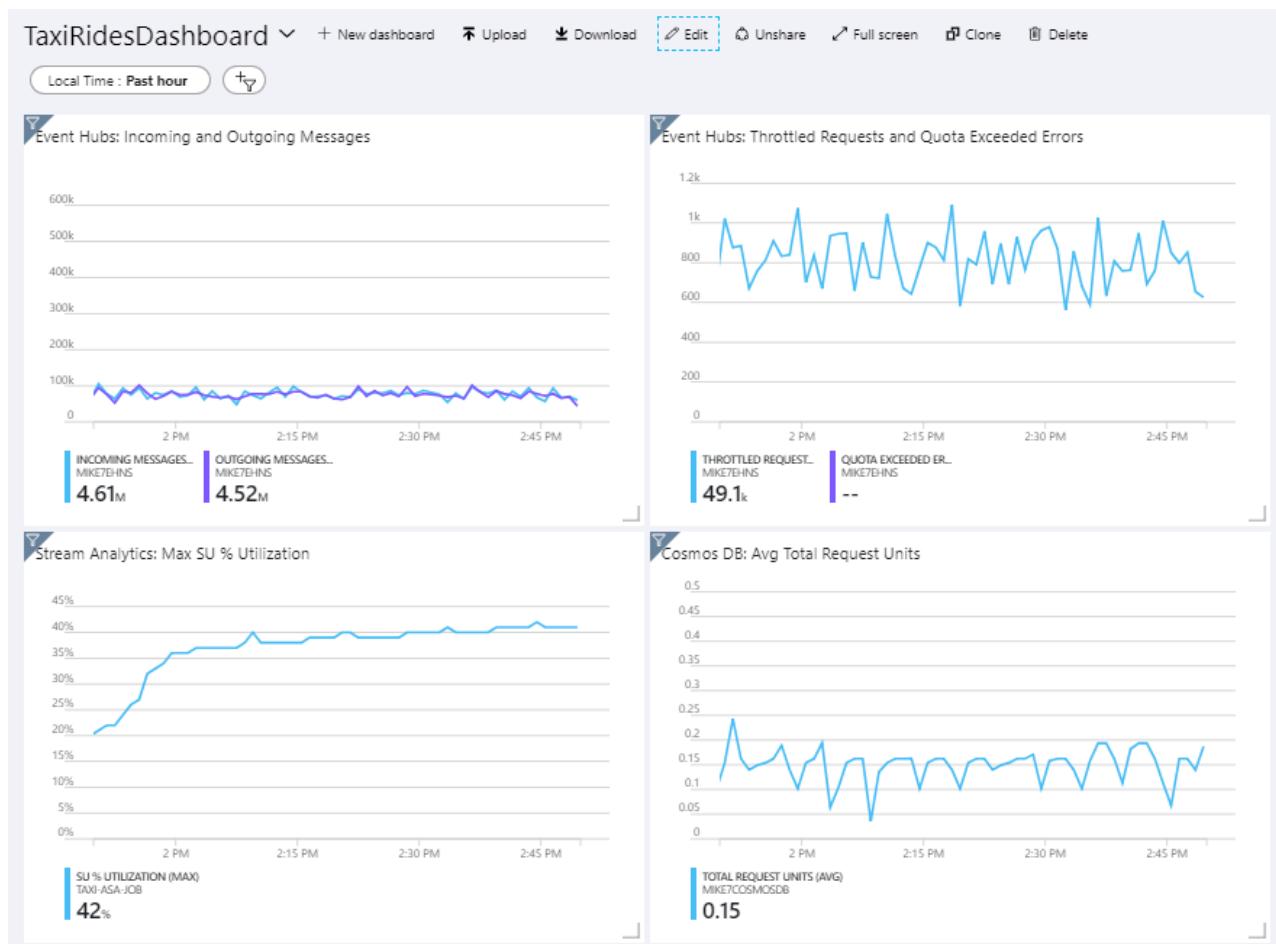
With any stream processing solution, it's important to monitor the performance and health of the system. [Azure Monitor](#) collects metrics and diagnostics logs for the Azure services used in the architecture. Azure Monitor is built into the Azure platform and does not require any additional code in your application.

Any of the following warning signals indicate that you should scale out the relevant Azure resource:

- Event Hubs throttles requests or is close to the daily message quota.
- The Stream Analytics job consistently uses more than 80% of allocated Streaming Units (SU).
- Cosmos DB begins to throttle requests.

The reference architecture includes a custom dashboard, which is deployed to the Azure portal. After you deploy the architecture, you can view the dashboard by opening the [Azure Portal](#) and selecting [TaxiRidesDashboard](#) from list of dashboards. For more information about creating and deploying custom dashboards in the Azure portal, see [Programmatically create Azure Dashboards](#).

The following image shows the dashboard after the Stream Analytics job ran for about an hour.



The panel on the lower left shows that the SU consumption for the Stream Analytics job climbs during the first 15 minutes and then levels off. This is a typical pattern as the job reaches a steady state.

Notice that Event Hubs is throttling requests, shown in the upper right panel. An occasional throttled request is not a problem, because the Event Hubs client SDK automatically retries when it receives a throttling error. However, if you see consistent throttling errors, it means the event hub needs more throughput units. The following graph shows a test run using the Event Hubs auto-inflate feature, which automatically scales out the throughput units as needed.



Auto-inflate was enabled at about the 06:35 mark. You can see the drop in throttled requests, as Event Hubs automatically scaled up to 3 throughput units.

Interestingly, this had the side effect of increasing the SU utilization in the Stream Analytics job. By throttling, Event Hubs was artificially reducing the ingestion rate for the Stream Analytics job. It's actually common that resolving one performance bottleneck reveals another. In this case, allocating additional SU for the Stream Analytics job resolved the issue.

## Deploy the solution

A deployment for this reference architecture is available on [GitHub](#).

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Docker](#) to run the data generator.
3. Install [Azure CLI 2.0](#).
4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Download the source data files

1. Create a directory named `DataFile` under the `data/streaming_asa` directory in the GitHub repo.
2. Open a web browser and navigate to <https://uofi.app.box.com/v/NYCTaxidata/folder/2332219935>.
3. Click the **Download** button on this page to download a zip file of all the taxi data for that year.
4. Extract the zip file to the `DataFile` directory.

#### NOTE

This zip file contains other zip files. Don't extract the child zip files.

The directory structure should look like the following:

```
/data
  /streaming_asa
    /DataFile
      /FOIL2013
        trip_data_1.zip
        trip_data_2.zip
        trip_data_3.zip
        ...
```

## Deploy the Azure resources

1. From a shell or Windows Command Prompt, run the following command and follow the sign-in prompt:

```
az login
```

2. Navigate to the folder `data/streaming_asa` in the GitHub repository

```
cd data/streaming_asa
```

3. Run the following commands to deploy the Azure resources:

```
export resourceGroup='[Resource group name]'
export resourceLocation='[Location]'
export cosmosDatabaseAccount='[Cosmos DB account name]'
export cosmosDatabase='[Cosmod DB database name]'
export cosmosDataBaseCollection='[Cosmos DB collection name]'
export eventHubNamespace='[Event Hubs namespace name]'

# Create a resource group
az group create --name $resourceGroup --location $resourceLocation

# Deploy resources
az group deployment create --resource-group $resourceGroup \
  --template-file ./azure/deployresources.json --parameters \
  eventHubNamespace=$eventHubNamespace \
  outputCosmosDatabaseAccount=$cosmosDatabaseAccount \
  outputCosmosDatabase=$cosmosDatabase \
  outputCosmosDataBaseCollection=$cosmosDataBaseCollection

# Create a database
az cosmosdb database create --name $cosmosDatabaseAccount \
  --db-name $cosmosDatabase --resource-group $resourceGroup

# Create a collection
az cosmosdb collection create --collection-name $cosmosDataBaseCollection \
  --name $cosmosDatabaseAccount --db-name $cosmosDatabase \
  --resource-group $resourceGroup
```

4. In the Azure portal, navigate to the resource group that was created.
5. Open the blade for the Stream Analytics job.
6. Click **Start** to start the job. Select **Now** as the output start time. Wait for the job to start.

## Run the data generator

1. Get the Event Hub connection strings. You can get these from the Azure portal, or by running the following CLI commands:

```

# RIDE_EVENT_HUB
az eventhubs eventhub authorization-rule keys list \
    --eventhub-name taxi-ride \
    --name taxi-ride-asa-access-policy \
    --namespace-name $eventHubNamespace \
    --resource-group $resourceGroup \
    --query primaryConnectionString

# FARE_EVENT_HUB
az eventhubs eventhub authorization-rule keys list \
    --eventhub-name taxi-fare \
    --name taxi-fare-asa-access-policy \
    --namespace-name $eventHubNamespace \
    --resource-group $resourceGroup \
    --query primaryConnectionString

```

2. Navigate to the directory `data/streaming_asa/onprem` in the GitHub repository

3. Update the values in the file `main.env` as follows:

```

RIDE_EVENT_HUB=[Connection string for taxi-ride event hub]
FARE_EVENT_HUB=[Connection string for taxi-fare event hub]
RIDE_DATA_FILE_PATH=/DataFile/FOIL2013
MINUTES_TO_LEAD=0
PUSH_RIDE_DATA_FIRST=false

```

4. Run the following command to build the Docker image.

```
docker build --no-cache -t dataloader .
```

5. Navigate back to the parent directory, `data/stream_asa`.

```
cd ..
```

6. Run the following command to run the Docker image.

```
docker run -v `pwd`/DataFile:/DataFile --env-file=onprem/main.env dataloader:latest
```

The output should look like the following:

```

Created 10000 records for TaxiFare
Created 10000 records for TaxiRide
Created 20000 records for TaxiFare
Created 20000 records for TaxiRide
Created 30000 records for TaxiFare
...

```

Let the program run for at least 5 minutes, which is the window defined in the Stream Analytics query. To verify the Stream Analytics job is running correctly, open the Azure portal and navigate to the Cosmos DB database. Open the **Data Explorer** blade and view the documents.

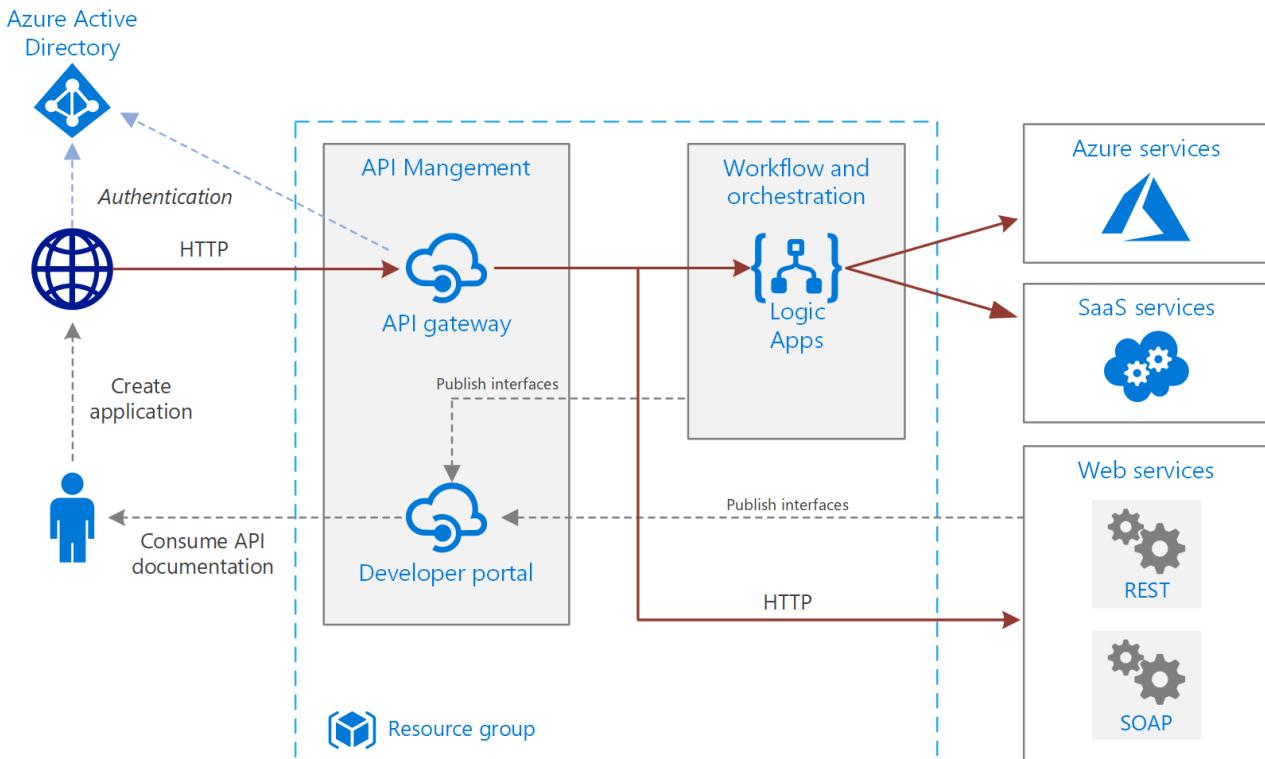
[1] Donovan, Brian; Work, Dan (2016): New York City Taxi Trip Data (2010-2013). University of Illinois at Urbana-Champaign. <https://doi.org/10.13012/J8PN93H8>

# Simple enterprise integration

10/24/2018 • 10 minutes to read • [Edit Online](#)

This reference architecture uses [Azure Integration Services](#) to orchestrate calls to enterprise backend systems. The backend systems may include software as a service (SaaS) systems, Azure services, and existing web services in your enterprise.

Azure Integration Services is a collection of services for integrating applications and data. This architecture uses two of those services: [Logic Apps](#) to orchestrate workflows, and [API Management](#) to create catalogs of APIs.



## Architecture

The architecture has the following components:

- **Backend systems.** On the right-hand side of the diagram are the various backend systems that the enterprise has deployed or relies on. These might include SaaS systems, other Azure services, or web services that expose REST or SOAP endpoints.
- **Azure Logic Apps.** [Logic Apps](#) is a serverless platform for building enterprise workflows that integrate applications, data, and services. In this architecture, the logic apps are triggered by HTTP requests. You can also nest workflows for more complex orchestration. Logic Apps uses [connectors](#) to integrate with commonly used services. Logic Apps offers hundreds of connectors, and you can create custom connectors.
- **Azure API Management.** [API Management](#) is a managed service for publishing catalogs of HTTP APIs, to promote re-use and discoverability. API Management consists of two related components:
  - **API gateway.** The API gateway accepts HTTP calls and routes them to the backend.
  - **Developer portal.** Each instance of Azure API Management provides access to a [developer portal](#). This portal gives your developers access to documentation and code samples for calling the APIs. You can also test APIs in the developer portal.

In this architecture, composite APIs are built by [importing logic apps](#) as APIs. You can also import existing web services by [importing OpenAPI](#) (Swagger) specifications or [importing SOAP APIs](#) from WSDL specifications.

The API gateway helps to decouple front-end clients from the back end. For example, it can rewrite URLs, or transform requests before they reach the backend. It also handles many cross-cutting concerns such as authentication, cross-origin resource sharing (CORS) support, and response caching.

- **Azure DNS.** [Azure DNS](#) is a hosting service for DNS domains. Azure DNS provides name resolution by using the Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records by using the same credentials, APIs, tools, and billing that you use for your other Azure services. To use a custom domain name, such as contoso.com, create DNS records that map the custom domain name to the IP address. For more information, see [Configure a custom domain name in API Management](#).
- **Azure Active Directory (Azure AD).** Use [Azure AD](#) to authenticate clients that call the API gateway. Azure AD supports the OpenID Connect (OIDC) protocol. Clients obtain an access token from Azure AD, and API Gateway [validates the token](#) to authorize the request. When using the Standard or Premium tier of API Management, Azure AD can also secure access to the developer portal.

## Recommendations

Your specific requirements might differ from the generic architecture shown here. Use the recommendations in this section as a starting point.

### API Management

Use the API Management Basic, Standard, or Premium tiers. These tiers offer a production service level agreement (SLA) and support scale out within the Azure region. Throughput capacity for API Management is measured in *units*. Each pricing tier has a maximum scale out. The Premium tier also supports scale out across multiple Azure regions. Choose your tier based on your feature set and the level of required throughput. For more information, see [API Management pricing](#) and [Capacity of an Azure API Management instance](#).

Each Azure API Management instance has a default domain name, which is a subdomain of `azure-api.net` &mdash; for example, `contoso.azure-api.net`. Consider configuring a [custom domain](#) for your organization.

### Logic Apps

Logic Apps works best in scenarios that don't require low latency. For example, Logic Apps works best for asynchronous or semi long-running API calls. If low latency is required, for example, a call that blocks a user interface, implement your API or operation by using a different technology. For example, use Azure Functions or a Web API that you deploy by using Azure App Service. Use API Management to front the API to your API consumers.

### Region

To minimize network latency, put API Management and Logic Apps in the same region. In general, choose the region that's closest to your users (or closest to your backend services).

The resource group also has a region. This region specifies where to store deployment metadata and where to execute the deployment template. To improve availability during deployment, put the resource group and resources in the same region.

## Scalability considerations

To increase the scalability of API Management, add [caching policies](#) where appropriate. Caching also helps reduce the load on back-end services.

To offer greater capacity, you can scale out Azure API Management Basic, Standard, and Premium tiers in an Azure region. To analyze the usage for your service, on the **Metrics** menu, select the **Capacity Metric** option and then

scale up or scale down as appropriate. The upgrade or scale process can take from 15 to 45 minutes to apply.

Recommendations for scaling an API Management service:

- Consider traffic patterns when scaling. Customers with more volatile traffic patterns need more capacity.
- Consistent capacity that's greater than 66% might indicate a need to scale up.
- Consistent capacity that's under 20% might indicate an opportunity to scale down.
- Before you enable the load in production, always load-test your API Management service with a representative load.

With the Premium tier, you can scale an API Management instance across multiple Azure regions. This makes API Management eligible for a higher SLA, and lets you provision services near users in multiple regions.

The Logic Apps serverless model means administrators don't have to plan for service scalability. The service automatically scales to meet demand.

## Availability considerations

Review the SLA for each service:

- [API Management SLA](#)
- [Logic Apps SLA](#)

If you deploy API Management across two or more regions with Premium tier, it is eligible for a higher SLA. See [API Management pricing](#).

### Backups

Regularly [back up](#) your API Management configuration. Store your backup files in a location or Azure region that differs from the region where the service is deployed. Based on your [RTO](#), choose a disaster recovery strategy:

- In a disaster recovery event, provision a new API Management instance, restore the backup to the new instance, and repoint the DNS records.
- Keep a passive instance of the API Management service in another Azure region. Regularly restore backups to that instance, to keep it in sync with the active service. To restore the service during a disaster recovery event, you need only repoint the DNS records. This approach incurs additional cost because you are paying for the passive instance, but reduces the time to recover.

For logic apps, we recommend a configuration-as-code approach to backup and restoring. Because logic apps are serverless, you can quickly recreate them from Azure Resource Manager templates. Save the templates in source control, integrate the templates with your continuous integration/continuous deployment (CI/CD) process. In a disaster recovery event, deploy the template to a new region.

If you deploy a logic app to a different region, update the configuration in API Management. You can update the API's **Backend** property by using a basic PowerShell script.

## Manageability considerations

Create separate resource groups for production, development, and test environments. Separate resource groups make it easier to manage deployments, delete test deployments, and assign access rights.

When you assign resources to resource groups, consider these factors:

- **Lifecycle.** In general, put resources that have the same lifecycle in the same resource group.
- **Access.** To apply access policies to the resources in a group, you can use [role-based access control \(RBAC\)](#).

- **Billing.** You can view rollup costs for the resource group.
- **Pricing tier for API Management.** Use the Developer tier for development and test environments. To minimize costs during preproduction, deploy a replica of your production environment, run your tests, and then shut down.

## Deployment

Use [Azure Resource Manager templates](#) to deploy the Azure resources. Templates make it easier to automate deployments using PowerShell or the Azure CLI.

Put API Management and any individual logic apps in their own separate Resource Manager templates. By using separate templates, you can store the resources in source control systems. You can then deploy these templates together or individually as part of a continuous integration/continuous deployment (CI/CD) process.

## Versions

Each time you change a logic app's configuration or deploy an update through a Resource Manager template, Azure keeps a copy of that version and keeps all versions that have a run history. You can use these versions to track historical changes or promote a version as the logic app's current configuration. For example, you can roll back a logic app to a previous version.

API Management supports two distinct but complementary versioning concepts:

- *Versions* allow API consumers to choose an API version based on their needs, for example, v1, v2, beta, or production.
- *Revisions* allow API administrators to make non-breaking changes in an API and deploy those changes, along with a change log to inform API consumers about the changes.

You can make a revision in a development environment and deploy that change in other environments by using Resource Manager templates. For more information, see [Publish multiple versions of your API](#)

You can also use revisions to test an API before making the changes current and accessible to users. However, this method isn't recommended for load testing or integration testing. Instead, use separate test or preproduction environments.

## Diagnostics and monitoring

Use [Azure Monitor](#) for operational monitoring in both API Management and Logic Apps. Azure Monitor provides information based on the metrics configured for each service and is enabled by default. For more information, see:

- [Monitor published APIs](#)
- [Monitor status, set up diagnostics logging, and turn on alerts for Azure Logic Apps](#)

Each service also has these options:

- For deeper analysis and dashboarding, send Logic Apps logs to [Azure Log Analytics](#).
- For DevOps monitoring, configure Azure Application Insights for API Management.
- API Management supports the [Power BI solution template for custom API analytics](#). You can use this solution template for creating your own analytics solution. For business users, Power BI makes reports available.

## Security considerations

Although this list doesn't completely describe all security best practices, here are some security considerations that apply specifically to this architecture:

- The Azure API Management service has a fixed public IP address. Restrict access for calling Logic Apps endpoints to only the IP address of API Management. For more information, see [Restrict incoming IP addresses](#).
- To make sure users have appropriate access levels, use role-based access control (RBAC).
- Secure public API endpoints in API Management by using OAuth or OpenID Connect. To secure public API endpoints, configure an identity provider, and add a JSON Web Token (JWT) validation policy. For more information, see [Protect an API by using OAuth 2.0 with Azure Active Directory and API Management](#).
- Connect to back-end services from API Management by using mutual certificates.
- Enforce HTTPS on the API Management APIs.

### Storing secrets

Never check passwords, access keys, or connection strings into source control. If these values are required, secure and deploy these values by using the appropriate techniques.

If a logic app requires any sensitive values that you can't create within a connector, store those values in Azure Key Vault and reference them from a Resource Manager template. Use deployment template parameters and parameter files for each environment. For more information, see [Secure parameters and inputs within a workflow](#).

API Management manages secrets by using objects called *named values* or *properties*. These objects securely store values that you can access through API Management policies. For more information, see [How to use Named Values in Azure API Management policies](#).

## Cost considerations

You are charged for all API Management instances when they are running. If you have scaled up and don't need that level of performance all the time, manually scale down or configure [autoscaling](#).

Logic Apps uses a [serverless](#) model. Billing is calculated based on action and connector execution. For more information, see [Logic Apps pricing](#). Currently, there are no tier considerations for Logic Apps.

## Next steps

- Learn about [enterprise integration with queues and events](#)

# Connect an on-premises network to Azure

7/11/2018 • 2 minutes to read • [Edit Online](#)

This article compares options for connecting an on-premises network to an Azure Virtual Network (VNet). For each option, a more detailed reference architecture is available.

## VPN connection

A [VPN gateway](#) is a type of virtual network gateway that sends encrypted traffic between an Azure virtual network and an on-premises location. The encrypted traffic goes over the public Internet.

This architecture is suitable for hybrid applications where the traffic between on-premises hardware and the cloud is likely to be light, or you are willing to trade slightly extended latency for the flexibility and processing power of the cloud.

### Benefits

- Simple to configure.

### Challenges

- Requires an on-premises VPN device.
- Although Microsoft guarantees 99.9% availability for each VPN Gateway, this [SLA](#) only covers the VPN gateway, and not your network connection to the gateway.
- A VPN connection over Azure VPN Gateway currently supports a maximum of 200 Mbps bandwidth. You may need to partition your Azure virtual network across multiple VPN connections if you expect to exceed this throughput.

### Reference architecture

- [Hybrid network with VPN gateway](#)

## Azure ExpressRoute connection

[ExpressRoute](#) connections use a private, dedicated connection through a third-party connectivity provider. The private connection extends your on-premises network into Azure.

This architecture is suitable for hybrid applications running large-scale, mission-critical workloads that require a high degree of scalability.

### Benefits

- Much higher bandwidth available; up to 10 Gbps depending on the connectivity provider.
- Supports dynamic scaling of bandwidth to help reduce costs during periods of lower demand. However, not all connectivity providers have this option.
- May allow your organization direct access to national clouds, depending on the connectivity provider.
- 99.9% availability SLA across the entire connection.

### Challenges

- Can be complex to set up. Creating an ExpressRoute connection requires working with a third-party connectivity provider. The provider is responsible for provisioning the network connection.
- Requires high-bandwidth routers on-premises.

## **Reference architecture**

- [Hybrid network with ExpressRoute](#)

# ExpressRoute with VPN failover

This options combines the previous two, using ExpressRoute in normal conditions, but failing over to a VPN connection if there is a loss of connectivity in the ExpressRoute circuit.

This architecture is suitable for hybrid applications that need the higher bandwidth of ExpressRoute, and also require highly available network connectivity.

## **Benefits**

- High availability if the ExpressRoute circuit fails, although the fallback connection is on a lower bandwidth network.

## **Challenges**

- Complex to configure. You need to set up both a VPN connection and an ExpressRoute circuit.
- Requires redundant hardware (VPN appliances), and a redundant Azure VPN Gateway connection for which you pay charges.

## **Reference architecture**

- [Hybrid network with ExpressRoute and VPN failover](#)

# Hub-spoke network topology

A hub-spoke network topology is a way to isolate workloads while sharing services such as identity and security. 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. Shared services are deployed in the hub, while individual workloads are deployed as spokes.

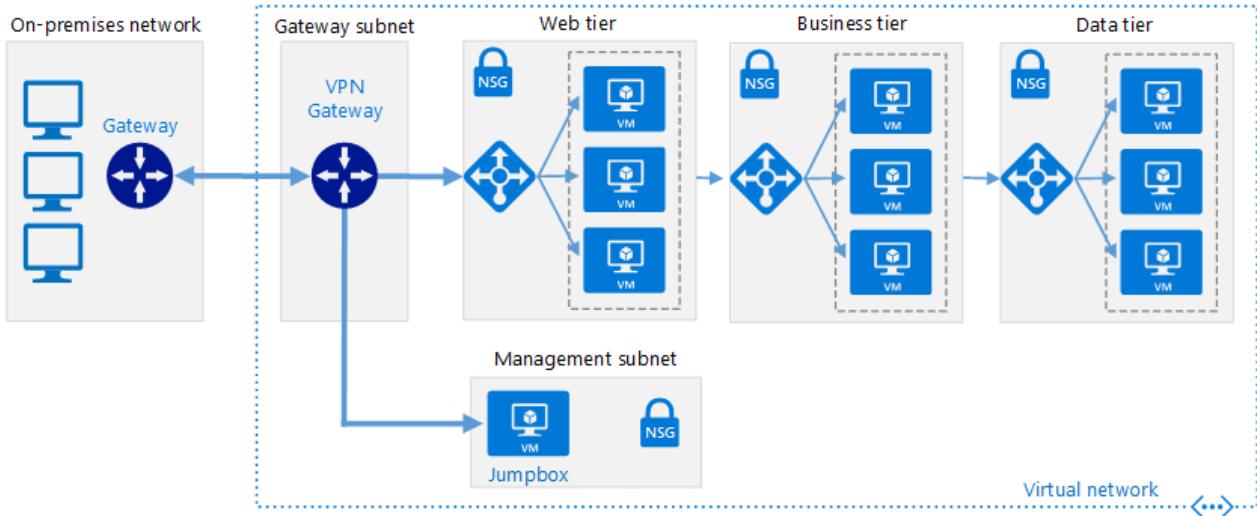
## **Reference architectures**

- [Hub-spoke topology](#)
- [Hub-spoke with shared services](#)

# Connect an on-premises network to Azure using a VPN gateway

9/28/2018 • 17 minutes to read • [Edit Online](#)

This reference architecture shows how to extend an on-premises network to Azure, using a site-to-site virtual private network (VPN). Traffic flows between the on-premises network and an Azure Virtual Network (VNet) through an IPSec VPN tunnel. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

## Architecture

The architecture consists of the following components.

- **On-premises network.** A private local-area network running within an organization.
- **VPN appliance.** A device or service that provides external connectivity to the on-premises network. The VPN appliance may be a hardware device, or it can be a software solution such as the Routing and Remote Access Service (RRAS) in Windows Server 2012. For a list of supported VPN appliances and information on configuring them to connect to an Azure VPN gateway, see the instructions for the selected device in the article [About VPN devices for Site-to-Site VPN Gateway connections](#).
- **Virtual network (VNet).** The cloud application and the components for the Azure VPN gateway reside in the same [VNet](#).
- **Azure VPN gateway.** The [VPN gateway](#) service enables you to connect the VNet to the on-premises network through a VPN appliance. For more information, see [Connect an on-premises network to a Microsoft Azure virtual network](#). The VPN gateway includes the following elements:
  - **Virtual network gateway.** A resource that provides a virtual VPN appliance for the VNet. It is responsible for routing traffic from the on-premises network to the VNet.
  - **Local network gateway.** An abstraction of the on-premises VPN appliance. Network traffic from the cloud application to the on-premises network is routed through this gateway.
  - **Connection.** The connection has properties that specify the connection type (IPSec) and the key shared with the on-premises VPN appliance to encrypt traffic.
  - **Gateway subnet.** The virtual network gateway is held in its own subnet, which is subject to various

requirements, described in the Recommendations section below.

- **Cloud application.** The application hosted in Azure. It might include multiple tiers, with multiple subnets connected through Azure load balancers. For more information about the application infrastructure, see [Running Windows VM workloads](#) and [Running Linux VM workloads](#).
- **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.

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### VNet and gateway subnet

Create an Azure VNet with an address space large enough for all of your required resources. Ensure that the VNet address space has sufficient room for growth if additional VMs are likely to be needed in the future. The address space of the VNet must not overlap with the on-premises network. For example, the diagram above uses the address space 10.20.0.0/16 for the VNet.

Create a subnet named *GatewaySubnet*, with an address range of /27. This subnet is required by the virtual network gateway. Allocating 32 addresses to this subnet will help to prevent reaching gateway size limitations in the future. Also, avoid placing this subnet in the middle of the address space. A good practice is to set the address space for the gateway subnet at the upper end of the VNet address space. The example shown in the diagram uses 10.20.255.224/27. Here is a quick procedure to calculate the [CIDR](#):

1. Set the variable bits in the address space of the VNet to 1, up to the bits being used by the gateway subnet, then set the remaining bits to 0.
2. Convert the resulting bits to decimal and express it as an address space with the prefix length set to the size of the gateway subnet.

For example, for a VNet with an IP address range of 10.20.0.0/16, applying step #1 above becomes 10.20.0b1111111.0b11100000. Converting that to decimal and expressing it as an address space yields 10.20.255.224/27.

#### WARNING

Do not deploy any VMs to the gateway subnet. Also, do not assign an NSG to this subnet, as it will cause the gateway to stop functioning.

### Virtual network gateway

Allocate a public IP address for the virtual network gateway.

Create the virtual network gateway in the gateway subnet and assign it the newly allocated public IP address. Use the gateway type that most closely matches your requirements and that is enabled by your VPN appliance:

- Create a [policy-based gateway](#) if you need to closely control how requests are routed based on policy criteria such as address prefixes. Policy-based gateways use static routing, and only work with site-to-site connections.
- Create a [route-based gateway](#) if you connect to the on-premises network using RRAS, support multi-site or cross-region connections, or implement VNet-to-VNet connections (including routes that traverse multiple VNets). Route-based gateways use dynamic routing to direct traffic between networks. They can tolerate failures in the network path better than static routes because they can try alternative routes. Route-based gateways can also reduce the management overhead because routes might not need to be updated manually when network addresses change.

For a list of supported VPN appliances, see [About VPN devices for Site-to-Site VPN Gateway connections](#).

**NOTE**

After the gateway has been created, you cannot change between gateway types without deleting and re-creating the gateway.

Select the Azure VPN gateway SKU that most closely matches your throughput requirements. For more information, see [Gateway SKUs](#)

**NOTE**

The Basic SKU is not compatible with Azure ExpressRoute. You can [change the SKU](#) after the gateway has been created.

You are charged based on the amount of time that the gateway is provisioned and available. See [VPN Gateway Pricing](#).

Create routing rules for the gateway subnet that direct incoming application traffic from the gateway to the internal load balancer, rather than allowing requests to pass directly to the application VMs.

**On-premises network connection**

Create a local network gateway. Specify the public IP address of the on-premises VPN appliance, and the address space of the on-premises network. Note that the on-premises VPN appliance must have a public IP address that can be accessed by the local network gateway in Azure VPN Gateway. The VPN device cannot be located behind a network address translation (NAT) device.

Create a site-to-site connection for the virtual network gateway and the local network gateway. Select the site-to-site (IPSec) connection type, and specify the shared key. Site-to-site encryption with the Azure VPN gateway is based on the IPSec protocol, using preshared keys for authentication. You specify the key when you create the Azure VPN gateway. You must configure the VPN appliance running on-premises with the same key. Other authentication mechanisms are not currently supported.

Ensure that the on-premises routing infrastructure is configured to forward requests intended for addresses in the Azure VNet to the VPN device.

Open any ports required by the cloud application in the on-premises network.

Test the connection to verify that:

- The on-premises VPN appliance correctly routes traffic to the cloud application through the Azure VPN gateway.
- The VNet correctly routes traffic back to the on-premises network.
- Prohibited traffic in both directions is blocked correctly.

## Scalability considerations

You can achieve limited vertical scalability by moving from the Basic or Standard VPN Gateway SKUs to the High Performance VPN SKU.

For VNets that expect a large volume of VPN traffic, consider distributing the different workloads into separate smaller VNets and configuring a VPN gateway for each of them.

You can partition the VNet either horizontally or vertically. To partition horizontally, move some VM instances from each tier into subnets of the new VNet. The result is that each VNet has the same structure and functionality. To partition vertically, redesign each tier to divide the functionality into different logical areas (such as handling

orders, invoicing, customer account management, and so on). Each functional area can then be placed in its own VNet.

Replicating an on-premises Active Directory domain controller in the VNet, and implementing DNS in the VNet, can help to reduce some of the security-related and administrative traffic flowing from on-premises to the cloud. For more information, see [Extending Active Directory Domain Services \(AD DS\) to Azure](#).

## Availability considerations

If you need to ensure that the on-premises network remains available to the Azure VPN gateway, implement a failover cluster for the on-premises VPN gateway.

If your organization has multiple on-premises sites, create [multi-site connections](#) to one or more Azure VNets. This approach requires dynamic (route-based) routing, so make sure that the on-premises VPN gateway supports this feature.

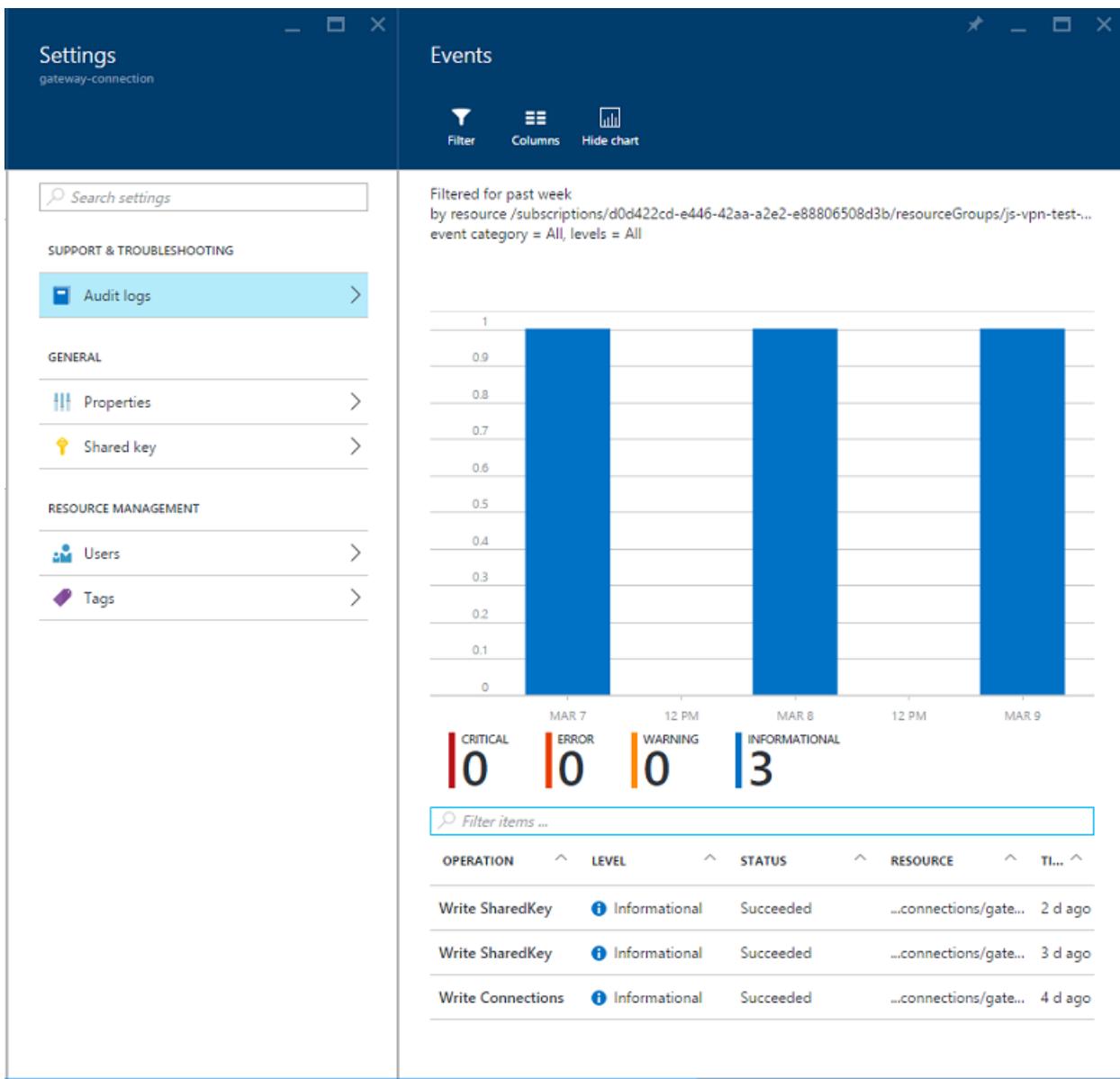
For details about service level agreements, see [SLA for VPN Gateway](#).

## Manageability considerations

Monitor diagnostic information from on-premises VPN appliances. This process depends on the features provided by the VPN appliance. For example, if you are using the Routing and Remote Access Service on Windows Server 2012, [RRAS logging](#).

Use [Azure VPN gateway diagnostics](#) to capture information about connectivity issues. These logs can be used to track information such as the source and destinations of connection requests, which protocol was used, and how the connection was established (or why the attempt failed).

Monitor the operational logs of the Azure VPN gateway using the audit logs available in the Azure portal. Separate logs are available for the local network gateway, the Azure network gateway, and the connection. This information can be used to track any changes made to the gateway, and can be useful if a previously functioning gateway stops working for some reason.



Monitor connectivity, and track connectivity failure events. You can use a monitoring package such as [Nagios](#) to capture and report this information.

## Security considerations

Generate a different shared key for each VPN gateway. Use a strong shared key to help resist brute-force attacks.

### NOTE

Currently, you cannot use Azure Key Vault to preshare keys for the Azure VPN gateway.

Ensure that the on-premises VPN appliance uses an encryption method that is [compatible with the Azure VPN gateway](#). For policy-based routing, the Azure VPN gateway supports the AES256, AES128, and 3DES encryption algorithms. Route-based gateways support AES256 and 3DES.

If your on-premises VPN appliance is on a perimeter network (DMZ) that has a firewall between the perimeter network and the Internet, you might have to configure [additional firewall rules](#) to allow the site-to-site VPN connection.

If the application in the VNet sends data to the Internet, consider [implementing forced tunneling](#) to route all Internet-bound traffic through the on-premises network. This approach enables you to audit outgoing requests made by the application from the on-premises infrastructure.

#### **NOTE**

Forced tunneling can impact connectivity to Azure services (the Storage Service, for example) and the Windows license manager.

## Troubleshooting

For general information on troubleshooting common VPN-related errors, see [Troubleshooting common VPN related errors](#).

The following recommendations are useful for determining if your on-premises VPN appliance is functioning correctly.

- **Check any log files generated by the VPN appliance for errors or failures.**

This will help you determine if the VPN appliance is functioning correctly. The location of this information will vary according to your appliance. For example, if you are using RRAS on Windows Server 2012, you can use the following PowerShell command to display error event information for the RRAS service:

```
Get-EventLog -LogName System -EntryType Error -Source RemoteAccess | Format-List -Property *
```

The *Message* property of each entry provides a description of the error. Some common examples are:

```
- Inability to connect, possibly due to an incorrect IP address specified for the Azure VPN gateway  
in the RRAS VPN network interface configuration.  
  
...  
EventID : 20111  
MachineName : on-prem-vm  
Data : {41, 3, 0, 0}  
Index : 14231  
Category : (0)  
CategoryNumber : 0  
EntryType : Error  
Message : RoutingDomainID- {00000000-0000-0000-0000-000000000000}: A demand dial  
connection to the remote  
interface AzureGateway on port VPN2-4 was successfully initiated but failed to  
complete  
successfully because of the following error: The network connection between  
your computer and  
the VPN server could not be established because the remote server is not  
responding. This could  
be because one of the network devices (for example, firewalls, NAT, routers, and  
so on) between your computer  
and the remote server is not configured to allow VPN connections. Please contact  
your  
Administrator or your service provider to determine which device may be causing  
the problem.  
Source : RemoteAccess  
ReplacementStrings : {{00000000-0000-0000-0000-000000000000}, AzureGateway, VPN2-4, The network  
connection between  
your computer and the VPN server could not be established because the remote  
server is not  
responding. This could be because one of the network devices (for example,  
firewalls, NAT, routers, and so on)  
between your computer and the remote server is not configured to allow VPN  
connections. Please  
contact your Administrator or your service provider to determine which device  
may be causing the  
problem.  
InstanceId : 20111  
TimeGenerated : 2/10/2016 1:26:02 PM
```

```

TimeGenerated      : 3/18/2016 1:26:02 PM
TimeWritten       : 3/18/2016 1:26:02 PM
UserName          :
Site              :
Container         :
```
- The wrong shared key being specified in the RRAS VPN network interface configuration.

```
EventID           : 20111
MachineName       : on-prem-vm
Data              : {233, 53, 0, 0}
Index             : 14245
Category          : (0)
CategoryNumber    : 0
EntryType         : Error
Message           : RoutingDomainID- {00000000-0000-0000-0000-000000000000}: A demand dial
connection to the remote
                           interface AzureGateway on port VPN2-4 was successfully initiated but failed to
complete
                           successfully because of the following error: Internet key exchange (IKE)
authentication credentials are unacceptable.

Source            : RemoteAccess
ReplacementStrings : {{00000000-0000-0000-0000-000000000000}, AzureGateway, VPN2-4, IKE
authentication credentials are
                           unacceptable.
}
InstanceId        : 20111
TimeGenerated     : 3/18/2016 1:34:22 PM
TimeWritten       : 3/18/2016 1:34:22 PM
UserName          :
Site              :
Container         :
```

```

You can also obtain event log information about attempts to connect through the RRAS service using the following PowerShell command:

```
Get-EventLog -LogName Application -Source RasClient | Format-List -Property *
```

In the event of a failure to connect, this log will contain errors that look similar to the following:

```

EventID           : 20227
MachineName       : on-prem-vm
Data              : {}
Index             : 4203
Category          : (0)
CategoryNumber    : 0
EntryType         : Error
Message           : CoId={B4000371-A67F-452F-AA4C-3125AA9CFC78}: The user SYSTEM dialed a connection
named
                           AzureGateway that has failed. The error code returned on failure is 809.
Source            : RasClient
ReplacementStrings : {{B4000371-A67F-452F-AA4C-3125AA9CFC78}, SYSTEM, AzureGateway, 809}
InstanceId        : 20227
TimeGenerated     : 3/18/2016 1:29:21 PM
TimeWritten       : 3/18/2016 1:29:21 PM
UserName          :
Site              :
Container         :

```

- **Verify connectivity and routing across the VPN gateway.**

The VPN appliance may not be correctly routing traffic through the Azure VPN Gateway. Use a tool such as [PsPing](#) to verify connectivity and routing across the VPN gateway. For example, to test connectivity from an on-premises machine to a web server located on the VNet, run the following command (replacing <> with the address of the web server):

```
PsPing -t <>:80
```

If the on-premises machine can route traffic to the web server, you should see output similar to the following:

```
D:\PSTools>psping -t 10.20.0.5:80

PsPing v2.01 - PsPing - ping, latency, bandwidth measurement utility
Copyright (C) 2012-2014 Mark Russinovich
Sysinternals - www.sysinternals.com

TCP connect to 10.20.0.5:80:
Infinite iterations (warmup 1) connecting test:
Connecting to 10.20.0.5:80 (warmup): 6.21ms
Connecting to 10.20.0.5:80: 3.79ms
Connecting to 10.20.0.5:80: 3.44ms
Connecting to 10.20.0.5:80: 4.81ms

Sent = 3, Received = 3, Lost = 0 (0% loss),
Minimum = 3.44ms, Maximum = 4.81ms, Average = 4.01ms
```

If the on-premises machine cannot communicate with the specified destination, you will see messages like this:

```
D:\PSTools>psping -t 10.20.1.6:80

PsPing v2.01 - PsPing - ping, latency, bandwidth measurement utility
Copyright (C) 2012-2014 Mark Russinovich
Sysinternals - www.sysinternals.com

TCP connect to 10.20.1.6:80:
Infinite iterations (warmup 1) connecting test:
Connecting to 10.20.1.6:80 (warmup): This operation returned because the timeout period expired.
Connecting to 10.20.1.6:80: This operation returned because the timeout period expired.
Connecting to 10.20.1.6:80: This operation returned because the timeout period expired.
Connecting to 10.20.1.6:80: This operation returned because the timeout period expired.
Connecting to 10.20.1.6:80:
Sent = 3, Received = 0, Lost = 3 (100% loss),
Minimum = 0.00ms, Maximum = 0.00ms, Average = 0.00ms
```

- Verify that the on-premises firewall allows VPN traffic to pass and that the correct ports are opened.
- Verify that the on-premises VPN appliance uses an encryption method that is compatible with the Azure VPN gateway. For policy-based routing, the Azure VPN gateway supports the AES256, AES128, and 3DES encryption algorithms. Route-based gateways support AES256 and 3DES.

The following recommendations are useful for determining if there is a problem with the Azure VPN gateway:

- Examine [Azure VPN gateway diagnostic logs](#) for potential issues.
- Verify that the Azure VPN gateway and on-premises VPN appliance are configured with the same shared authentication key.

You can view the shared key stored by the Azure VPN gateway using the following Azure CLI command:

```
azure network vpn-connection shared-key show <<resource-group>> <<vpn-connection-name>>
```

Use the command appropriate for your on-premises VPN appliance to show the shared key configured for that appliance.

Verify that the *GatewaySubnet* subnet holding the Azure VPN gateway is not associated with an NSG.

You can view the subnet details using the following Azure CLI command:

```
azure network vnet subnet show -g <<resource-group>> -e <<vnet-name>> -n GatewaySubnet
```

Ensure there is no data field named *Network Security Group id*. The following example shows the results for an instance of the *GatewaySubnet* that has an assigned NSG (*VPN-Gateway-Group*). This can prevent the gateway from working correctly if there are any rules defined for this NSG.

```
C:\>azure network vnet subnet show -g profx-prod-rg -e profx-vnet -n GatewaySubnet
  info: Executing command network vnet subnet show
+ Looking up virtual network "profx-vnet"
+ Looking up the subnet "GatewaySubnet"
  data: Id : /subscriptions/#####-####-####-#####
  #####/resourceGroups/profx-prod-rg/providers/Microsoft.Network/virtualNetworks/profx-
vnet/subnets/GatewaySubnet
  data: Name : GatewaySubnet
  data: Provisioning state : Succeeded
  data: Address prefix : 10.20.3.0/27
  data: Network Security Group id : /subscriptions/#####-####-####-#####
  #####/resourceGroups/profx-prod-rg/providers/Microsoft.Network/networkSecurityGroups/VPN-
Gateway-Group
  info: network vnet subnet show command OK
```

- **Verify that the virtual machines in the Azure VNet are configured to permit traffic coming in from outside the VNet.**

Check any NSG rules associated with subnets containing these virtual machines. You can view all NSG rules using the following Azure CLI command:

```
azure network nsg show -g <<resource-group>> -n <<nsg-name>>
```

- **Verify that the Azure VPN gateway is connected.**

You can use the following Azure PowerShell command to check the current status of the Azure VPN connection. The *<<connection-name>>* parameter is the name of the Azure VPN connection that links the virtual network gateway and the local gateway.

```
Get-AzureRmVirtualNetworkGatewayConnection -Name <<connection-name>> - ResourceGroupName <<resource-
group>>
```

The following snippets highlight the output generated if the gateway is connected (the first example), and disconnected (the second example):

```
PS C:\> Get-AzureRmVirtualNetworkGatewayConnection -Name profx-gateway-connection -ResourceGroupName profx-prod-rg
```

```
AuthorizationKey      : 
VirtualNetworkGateway1 : Microsoft.Azure.Commands.Network.Models.PSVirtualNetworkGateway
VirtualNetworkGateway2 : 
LocalNetworkGateway2  : Microsoft.Azure.Commands.Network.Models.PSLocalNetworkGateway
Peer                 : 
ConnectionType       : IPsec
RoutingWeight        : 0
SharedKey             : #####
ConnectionStatus     : Connected
EgressBytesTransferred : 55254803
IngressBytesTransferred : 32227221
ProvisioningState    : Succeeded
...
...
```

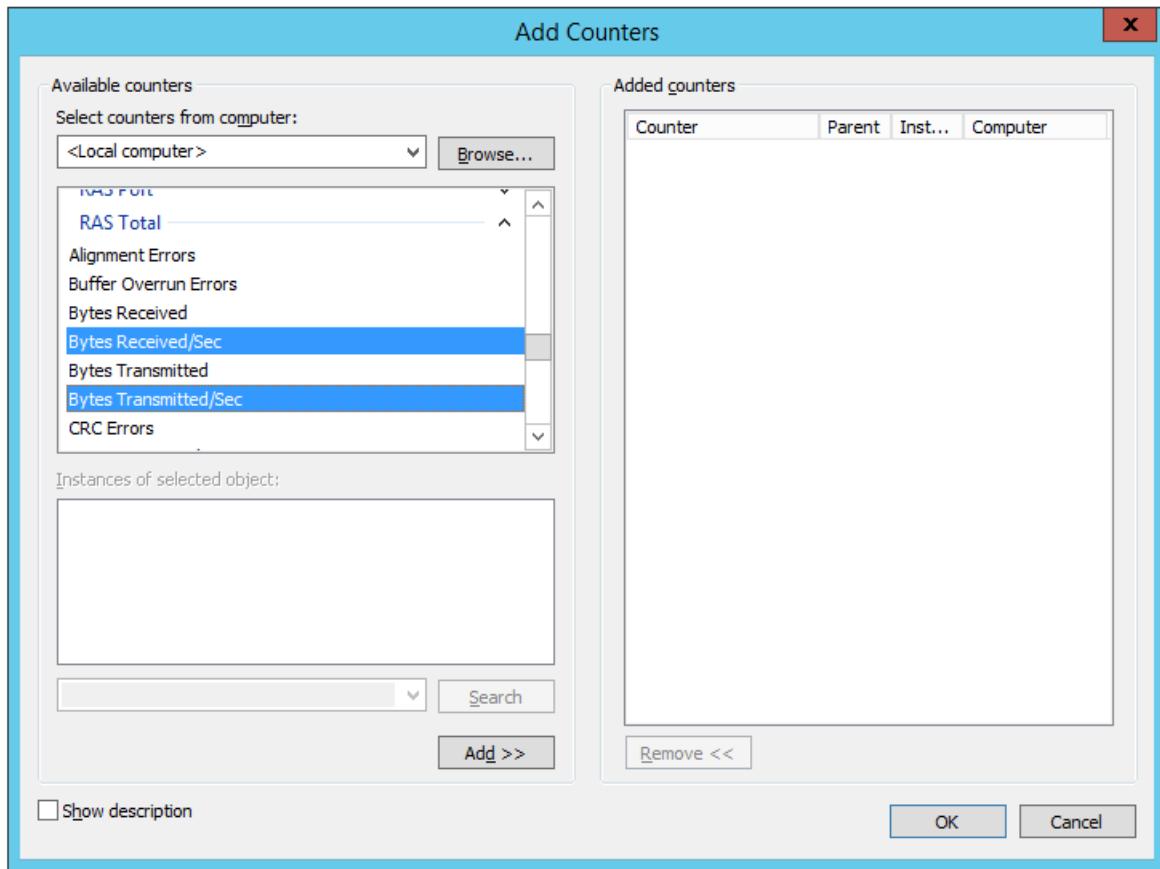
```
PS C:\> Get-AzureRmVirtualNetworkGatewayConnection -Name profx-gateway-connection2 -ResourceGroupName profx-prod-rg
```

```
AuthorizationKey      : 
VirtualNetworkGateway1 : Microsoft.Azure.Commands.Network.Models.PSVirtualNetworkGateway
VirtualNetworkGateway2 : 
LocalNetworkGateway2  : Microsoft.Azure.Commands.Network.Models.PSLocalNetworkGateway
Peer                 : 
ConnectionType       : IPsec
RoutingWeight        : 0
SharedKey             : #####
ConnectionStatus     : NotConnected
EgressBytesTransferred : 0
IngressBytesTransferred : 0
ProvisioningState    : Succeeded
...
...
```

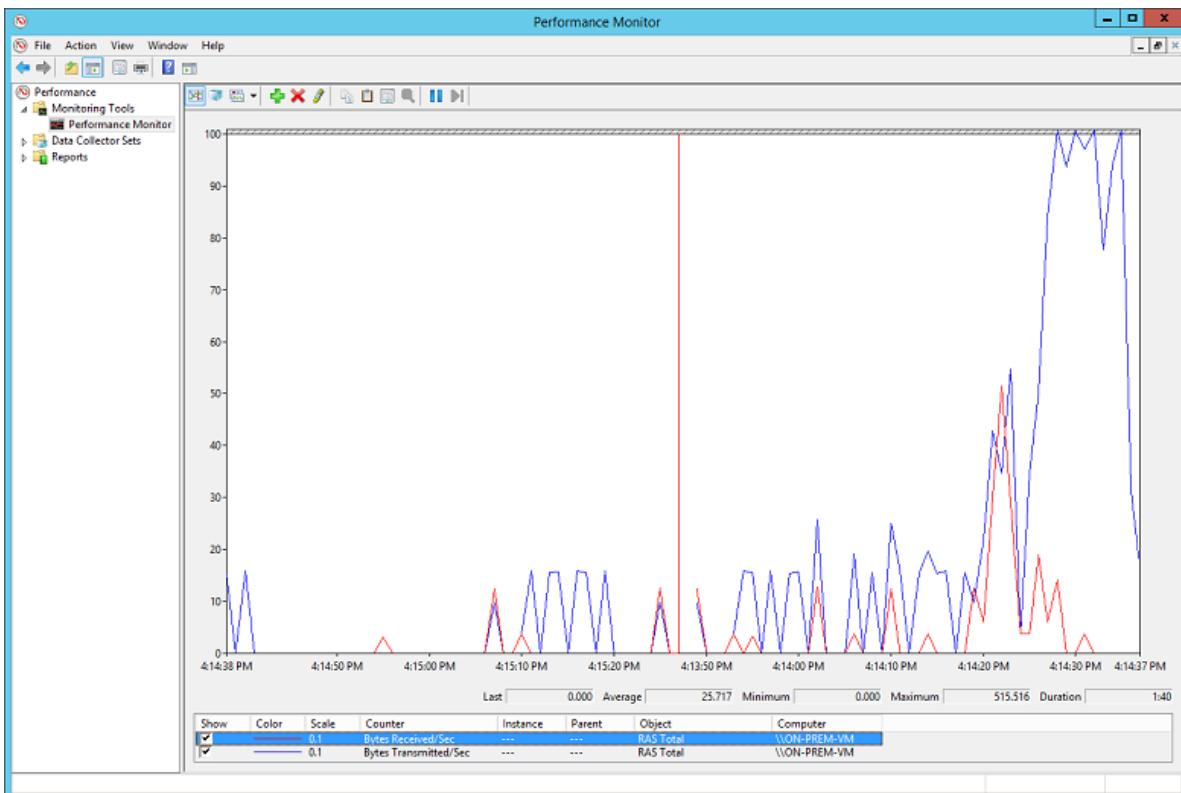
The following recommendations are useful for determining if there is an issue with Host VM configuration, network bandwidth utilization, or application performance:

- **Verify that the firewall in the guest operating system running on the Azure VMs in the subnet is configured correctly to allow permitted traffic from the on-premises IP ranges.**
- **Verify that the volume of traffic is not close to the limit of the bandwidth available to the Azure VPN gateway.**

How to verify this depends on the VPN appliance running on-premises. For example, if you are using RRAS on Windows Server 2012, you can use Performance Monitor to track the volume of data being received and transmitted over the VPN connection. Using the *RAS Total* object, select the *Bytes Received/Sec* and *Bytes Transmitted/Sec* counters:



You should compare the results with the bandwidth available to the VPN gateway (100 Mbps for the Basic and Standard SKUs, and 200 Mbps for the High Performance SKU):



- Verify that you have deployed the right number and size of VMs for your application load.

Determine if any of the virtual machines in the Azure VNet are running slowly. If so, they may be overloaded, there may be too few to handle the load, or the load-balancers may not be configured correctly. To determine this, [capture and analyze diagnostic information](#). You can examine the results using the Azure portal, but many third-party tools are also available that can provide detailed insights into the performance data.

- Verify that the application is making efficient use of cloud resources.

Instrument application code running on each VM to determine whether applications are making the best use of resources. You can use tools such as [Application Insights](#).

## Deploy the solution

**Prerequisites.** You must have an existing on-premises infrastructure already configured with a suitable network appliance.

To deploy the solution, perform the following steps.

1. Click the button below:



2. Wait for the link to open in the Azure portal, then follow these steps:

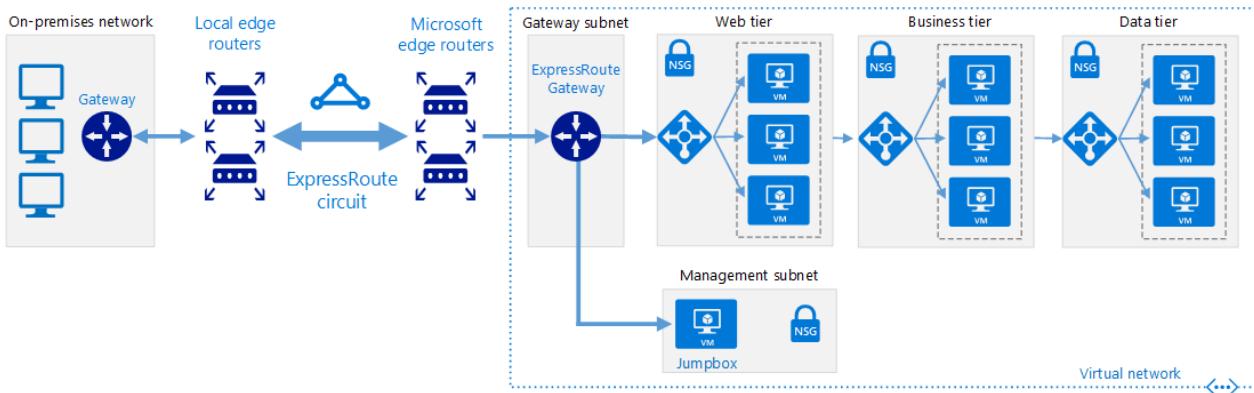
- The **Resource group** name is already defined in the parameter file, so select **Create New** and enter `ra-hybrid-vpn-rg` in the text box.
- Select the region from the **Location** drop down box.
- Do not edit the **Template Root Uri** or the **Parameter Root Uri** text boxes.
- Review the terms and conditions, then click the **I agree to the terms and conditions stated above** checkbox.
- Click the **Purchase** button.

3. Wait for the deployment to complete.

# Connect an on-premises network to Azure using ExpressRoute

10/5/2018 • 12 minutes to read • [Edit Online](#)

This reference architecture shows how to connect an on-premises network to virtual networks on Azure, using [Azure ExpressRoute](#). ExpressRoute connections use a private, dedicated connection through a third-party connectivity provider. The private connection extends your on-premises network into Azure. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

## Architecture

The architecture consists of the following components.

- **On-premises corporate network.** A private local-area network running within an organization.
- **ExpressRoute circuit.** A layer 2 or layer 3 circuit supplied by the connectivity provider that joins the on-premises network with Azure through the edge routers. The circuit uses the hardware infrastructure managed by the connectivity provider.
- **Local edge routers.** Routers that connect the on-premises network to the circuit managed by the provider. Depending on how your connection is provisioned, you may need to provide the public IP addresses used by the routers.
- **Microsoft edge routers.** Two routers in an active-active highly available configuration. These routers enable a connectivity provider to connect their circuits directly to their datacenter. Depending on how your connection is provisioned, you may need to provide the public IP addresses used by the routers.
- **Azure virtual networks (VNets).** Each VNet resides in a single Azure region, and can host multiple application tiers. Application tiers can be segmented using subnets in each VNet.
- **Azure public services.** Azure services that can be used within a hybrid application. These services are also available over the Internet, but accessing them using an ExpressRoute circuit provides low latency and more predictable performance, because traffic does not go through the Internet. Connections are performed using [public peering](#), with addresses that are either owned by your organization or supplied by your connectivity provider.
- **Office 365 services.** The publicly available Office 365 applications and services provided by Microsoft. Connections are performed using [Microsoft peering](#), with addresses that are either owned by your organization or supplied by your connectivity provider. You can also connect directly to Microsoft CRM

Online through Microsoft peering.

- **Connectivity providers** (not shown). Companies that provide a connection either using layer 2 or layer 3 connectivity between your datacenter and an Azure datacenter.

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### Connectivity providers

Select a suitable ExpressRoute connectivity provider for your location. To get a list of connectivity providers available at your location, use the following Azure PowerShell command:

```
Get-AzureRmExpressRouteServiceProvider
```

ExpressRoute connectivity providers connect your datacenter to Microsoft in the following ways:

- **Co-located at a cloud exchange.** If you're co-located in a facility with a cloud exchange, you can order virtual cross-connections to Azure through the co-location provider's Ethernet exchange. Co-location providers can offer either layer 2 cross-connections, or managed layer 3 cross-connections between your infrastructure in the co-location facility and Azure.
- **Point-to-point Ethernet connections.** You can connect your on-premises datacenters/offices to Azure through point-to-point Ethernet links. Point-to-point Ethernet providers can offer layer 2 connections, or managed layer 3 connections between your site and Azure.
- **Any-to-any (IPVPN) networks.** You can integrate your wide area network (WAN) with Azure. Internet protocol virtual private network (IPVPN) providers (typically a multiprotocol label switching VPN) offer any-to-any connectivity between your branch offices and datacenters. Azure can be interconnected to your WAN to make it look just like any other branch office. WAN providers typically offer managed layer 3 connectivity.

For more information about connectivity providers, see the [ExpressRoute introduction](#).

### ExpressRoute circuit

Ensure that your organization has met the [ExpressRoute prerequisite requirements](#) for connecting to Azure.

If you haven't already done so, add a subnet named `GatewaySubnet` to your Azure VNet and create an ExpressRoute virtual network gateway using the Azure VPN gateway service. For more information about this process, see [ExpressRoute workflows for circuit provisioning and circuit states](#).

Create an ExpressRoute circuit as follows:

1. Run the following PowerShell command:

```
New-AzureRmExpressRouteCircuit -Name <<circuit-name>> -ResourceGroupName <<resource-group>> -Location <<location>> -SkuTier <<sku-tier>> -SkuFamily <<sku-family>> -ServiceProviderName <<service-provider-name>> -PeeringLocation <<peering-location>> -BandwidthInMbps <<bandwidth-in-mbps>>
```

2. Send the `ServiceKey` for the new circuit to the service provider.

3. Wait for the provider to provision the circuit. To verify the provisioning state of a circuit, run the following PowerShell command:

```
Get-AzureRmExpressRouteCircuit -Name <<circuit-name>> -ResourceGroupName <<resource-group>>
```

The `Provisioning state` field in the `Service Provider` section of the output will change from

`NotProvisioned` to `Provisioned` when the circuit is ready.

#### NOTE

If you're using a layer 3 connection, the provider should configure and manage routing for you. You provide the information necessary to enable the provider to implement the appropriate routes.

#### 4. If you're using a layer 2 connection:

- a. Reserve two /30 subnets composed of valid public IP addresses for each type of peering you want to implement. These /30 subnets will be used to provide IP addresses for the routers used for the circuit. If you are implementing private, public, and Microsoft peering, you'll need 6 /30 subnets with valid public IP addresses.
- b. Configure routing for the ExpressRoute circuit. Run the following PowerShell commands for each type of peering you want to configure (private, public, and Microsoft). For more information, see [Create and modify routing for an ExpressRoute circuit](#).

```
Set-AzureRmExpressRouteCircuitPeeringConfig -Name <>peering-name<> -Circuit <>circuit-name<> -  
PeeringType <>peering-type<> -PeerASN <>peer-asn<> -PrimaryPeerAddressPrefix <>primary-peer-  
address-prefix<> -SecondaryPeerAddressPrefix <>secondary-peer-address-prefix<> -VlanId <>vlan-  
id<>  
  
Set-AzureRmExpressRouteCircuit -ExpressRouteCircuit <>circuit-name<>
```

- c. Reserve another pool of valid public IP addresses to use for network address translation (NAT) for public and Microsoft peering. It is recommended to have a different pool for each peering. Specify the pool to your connectivity provider, so they can configure border gateway protocol (BGP) advertisements for those ranges.

#### 5. Run the following PowerShell commands to link your private VNet(s) to the ExpressRoute circuit. For more information, see [Link a virtual network to an ExpressRoute circuit](#).

```
$circuit = Get-AzureRmExpressRouteCircuit -Name <>circuit-name<> -ResourceGroupName <>resource-group<>  
$gw = Get-AzureRmVirtualNetworkGateway -Name <>gateway-name<> -ResourceGroupName <>resource-group<>  
New-AzureRmVirtualNetworkGatewayConnection -Name <>connection-name<> -ResourceGroupName <>resource-  
group<> -Location <>location<> -VirtualNetworkGateway1 $gw -PeerId $circuit.Id -ConnectionType  
ExpressRoute
```

You can connect multiple VNets located in different regions to the same ExpressRoute circuit, as long as all VNets and the ExpressRoute circuit are located within the same geopolitical region.

#### Troubleshooting

If a previously functioning ExpressRoute circuit now fails to connect, in the absence of any configuration changes on-premises or within your private VNet, you may need to contact the connectivity provider and work with them to correct the issue. Use the following Powershell commands to verify that the ExpressRoute circuit has been provisioned:

```
Get-AzureRmExpressRouteCircuit -Name <>circuit-name<> -ResourceGroupName <>resource-group<>
```

The output of this command shows several properties for your circuit, including `ProvisioningState`, `CircuitProvisioningState`, and `ServiceProviderProvisioningState` as shown below.

```

ProvisioningState          : Succeeded
Sku                         :
    "Name": "Standard_MeteredData",
    "Tier": "Standard",
    "Family": "MeteredData"
}
CircuitProvisioningState   : Enabled
ServiceProviderProvisioningState : NotProvisioned

```

If the `ProvisioningState` is not set to `Succeeded` after you tried to create a new circuit, remove the circuit by using the command below and try to create it again.

```
Remove-AzureRmExpressRouteCircuit -Name <><circut-name>> -ResourceGroupName <><resource-group>>
```

If your provider had already provisioned the circuit, and the `ProvisioningState` is set to `Failed`, or the `CircuitProvisioningState` is not `Enabled`, contact your provider for further assistance.

## Scalability considerations

ExpressRoute circuits provide a high bandwidth path between networks. Generally, the higher the bandwidth the greater the cost.

ExpressRoute offers two [pricing plans](#) to customers, a metered plan and an unlimited data plan. Charges vary according to circuit bandwidth. Available bandwidth will likely vary from provider to provider. Use the `Get-AzureRmExpressRouteServiceProvider` cmdlet to see the providers available in your region and the bandwidths that they offer.

A single ExpressRoute circuit can support a certain number of peerings and VNet links. See [ExpressRoute limits](#) for more information.

For an extra charge, the ExpressRoute Premium add-on provides some additional capability:

- Increased route limits for public and private peering.
- Increased number of VNet links per ExpressRoute circuit.
- Global connectivity for services.

See [ExpressRoute pricing](#) for details.

ExpressRoute circuits are designed to allow temporary network bursts up to two times the bandwidth limit that you procured for no additional cost. This is achieved by using redundant links. However, not all connectivity providers support this feature. Verify that your connectivity provider enables this feature before depending on it.

Although some providers allow you to change your bandwidth, make sure you pick an initial bandwidth that surpasses your needs and provides room for growth. If you need to increase bandwidth in the future, you are left with two options:

- Increase the bandwidth. You should avoid this option as much as possible, and not all providers allow you to increase bandwidth dynamically. But if a bandwidth increase is needed, check with your provider to verify they support changing ExpressRoute bandwidth properties via Powershell commands. If they do, run the commands below.

```
$ckt = Get-AzureRmExpressRouteCircuit -Name <><circut-name>> -ResourceGroupName <><resource-group>>
$ckt.ServiceProviderProperties.BandwidthInMbps = <><bandwidth-in-mbps>>
Set-AzureRmExpressRouteCircuit -ExpressRouteCircuit $ckt
```

You can increase the bandwidth without loss of connectivity. Downgrading the bandwidth will result in

disruption in connectivity, because you must delete the circuit and recreate it with the new configuration.

- Change your pricing plan and/or upgrade to Premium. To do so, run the following commands. The `Sku.Tier` property can be `Standard` or `Premium`; the `Sku.Name` property can be `MeteredData` or `UnlimitedData`.

```
$ckt = Get-AzureRmExpressRouteCircuit -Name <><> -ResourceGroupName <><>  
  
$ckt.Sku.Tier = "Premium"  
$ckt.Sku.Family = "MeteredData"  
$ckt.Sku.Name = "Premium_MeteredData"  
  
Set-AzureRmExpressRouteCircuit -ExpressRouteCircuit $ckt
```

#### IMPORTANT

Make sure the `Sku.Name` property matches the `Sku.Tier` and `Sku.Family`. If you change the family and tier, but not the name, your connection will be disabled.

You can upgrade the SKU without disruption, but you cannot switch from the unlimited pricing plan to metered. When downgrading the SKU, your bandwidth consumption must remain within the default limit of the standard SKU.

## Availability considerations

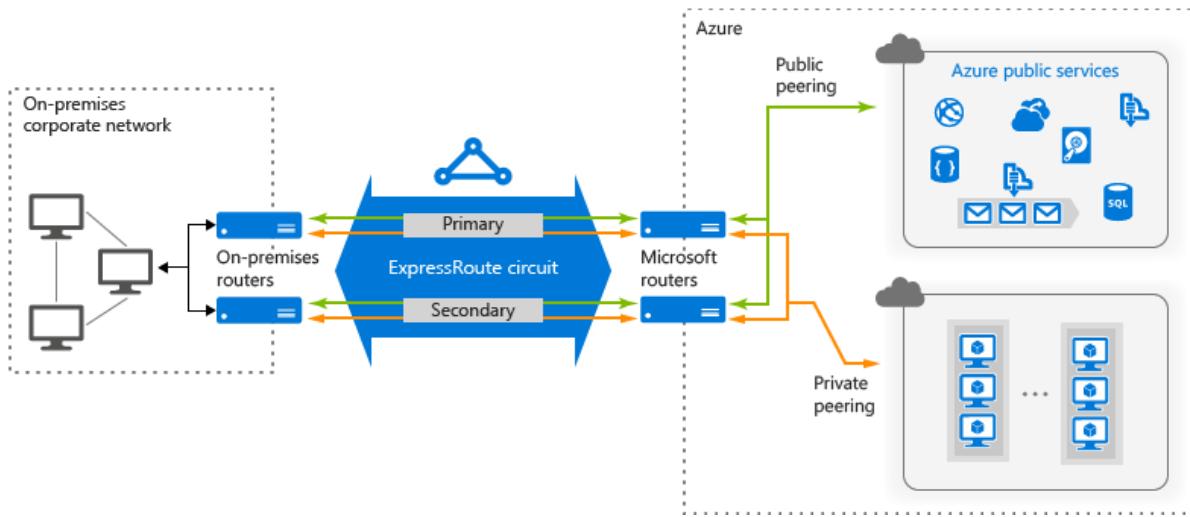
ExpressRoute does not support router redundancy protocols such as hot standby routing protocol (HSRP) and virtual router redundancy protocol (VRRP) to implement high availability. Instead, it uses a redundant pair of BGP sessions per peering. To facilitate highly-available connections to your network, Azure provisions you with two redundant ports on two routers (part of the Microsoft edge) in an active-active configuration.

By default, BGP sessions use an idle timeout value of 60 seconds. If a session times out three times (180 seconds total), the router is marked as unavailable, and all traffic is redirected to the remaining router. This 180-second timeout might be too long for critical applications. If so, you can change your BGP time-out settings on the on-premises router to a smaller value.

You can configure high availability for your Azure connection in different ways, depending on the type of provider you use, and the number of ExpressRoute circuits and virtual network gateway connections you're willing to configure. The following summarizes your availability options:

- If you're using a layer 2 connection, deploy redundant routers in your on-premises network in an active-active configuration. Connect the primary circuit to one router, and the secondary circuit to the other. This will give you a highly available connection at both ends of the connection. This is necessary if you require the ExpressRoute service level agreement (SLA). See [SLA for Azure ExpressRoute](#) for details.

The following diagram shows a configuration with redundant on-premises routers connected to the primary and secondary circuits. Each circuit handles the traffic for a public peering and a private peering (each peering is designated a pair of /30 address spaces, as described in the previous section).



- If you're using a layer 3 connection, verify that it provides redundant BGP sessions that handle availability for you.
- Connect the VNet to multiple ExpressRoute circuits, supplied by different service providers. This strategy provides additional high-availability and disaster recovery capabilities.
- Configure a site-to-site VPN as a failover path for ExpressRoute. For more about this option, see [Connect an on-premises network to Azure using ExpressRoute with VPN failover](#). This option only applies to private peering. For Azure and Office 365 services, the Internet is the only failover path.

## Manageability considerations

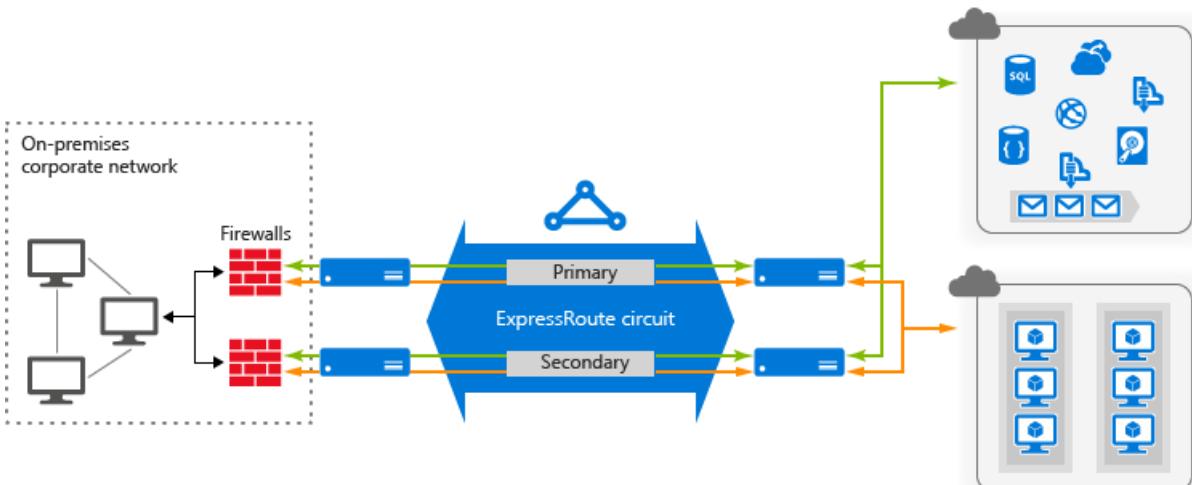
You can use the [Azure Connectivity Toolkit \(AzureCT\)](#) to monitor connectivity between your on-premises datacenter and Azure.

## Security considerations

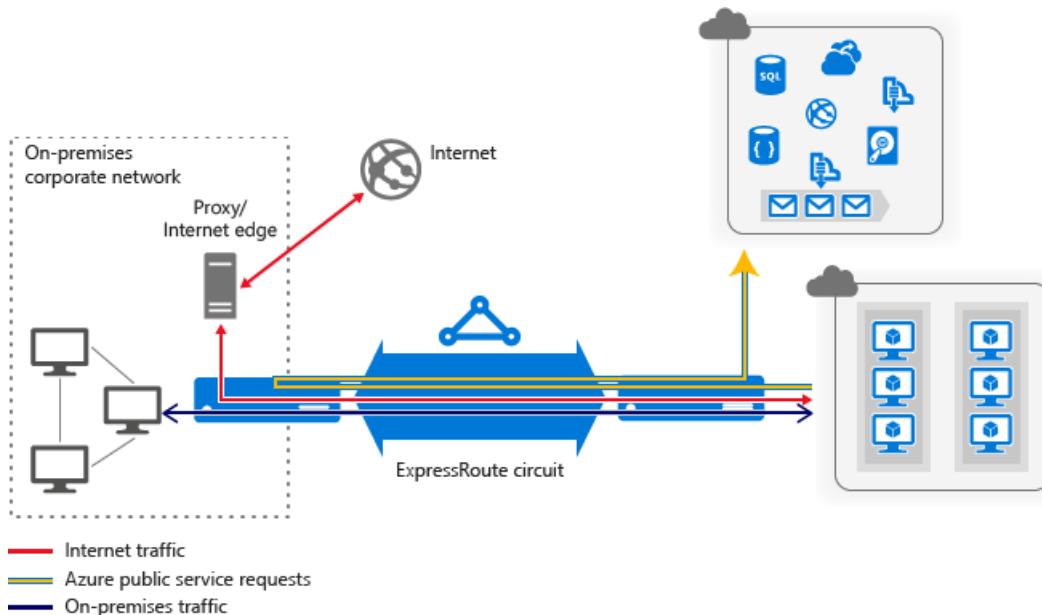
You can configure security options for your Azure connection in different ways, depending on your security concerns and compliance needs.

ExpressRoute operates in layer 3. Threats in the application layer can be prevented by using a network security appliance that restricts traffic to legitimate resources. Additionally, ExpressRoute connections using public peering can only be initiated from on-premises. This prevents a rogue service from accessing and compromising on-premises data from the Internet.

To maximize security, add network security appliances between the on-premises network and the provider edge routers. This will help to restrict the inflow of unauthorized traffic from the VNet:



For auditing or compliance purposes, it may be necessary to prohibit direct access from components running in the VNet to the Internet and implement [forced tunneling](#). In this situation, Internet traffic should be redirected back through a proxy running on-premises where it can be audited. The proxy can be configured to block unauthorized traffic flowing out, and filter potentially malicious inbound traffic.



To maximize security, do not enable a public IP address for your VMs, and use NSGs to ensure that these VMs aren't publicly accessible. VMs should only be available using the internal IP address. These addresses can be made accessible through the ExpressRoute network, enabling on-premises DevOps staff to perform configuration or maintenance.

If you must expose management endpoints for VMs to an external network, use NSGs or access control lists to restrict the visibility of these ports to a whitelist of IP addresses or networks.

#### NOTE

By default, Azure VMs deployed through the Azure portal include a public IP address that provides login access.

## Deploy the solution

**Prerequisites.** You must have an existing on-premises infrastructure already configured with a suitable network appliance.

To deploy the solution, perform the following steps.

1. Click the button below:

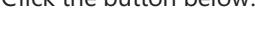


2. Wait for the link to open in the Azure portal, then follow these steps:

- The **Resource group** name is already defined in the parameter file, so select **Create New** and enter `ra-hybrid-er-rg` in the text box.
- Select the region from the **Location** drop down box.
- Do not edit the **Template Root Uri** or the **Parameter Root Uri** text boxes.
- Review the terms and conditions, then click the **I agree to the terms and conditions stated above** checkbox.
- Click the **Purchase** button.

3. Wait for the deployment to complete.

4. Click the button below:





5. Wait for the link to open in the Azure portal, then follow these steps:

- Select **Use existing** in the **Resource group** section and enter `ra-hybrid-er-rg` in the text box.
- Select the region from the **Location** drop down box.
- Do not edit the **Template Root Uri** or the **Parameter Root Uri** text boxes.
- Review the terms and conditions, then click the **I agree to the terms and conditions stated above** checkbox.
- Click the **Purchase** button.

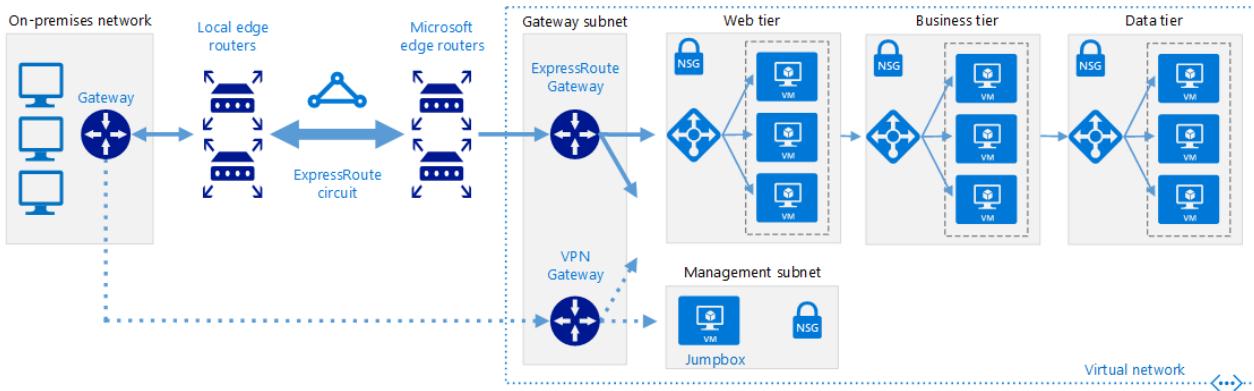
6. Wait for the deployment to complete.

# Connect an on-premises network to Azure using ExpressRoute with VPN failover

9/28/2018 • 4 minutes to read • [Edit Online](#)

This reference architecture shows how to connect an on-premises network to an Azure virtual network (VNet) using ExpressRoute, with a site-to-site virtual private network (VPN) as a failover connection. Traffic flows between the on-premises network and the Azure VNet through an ExpressRoute connection. If there is a loss of connectivity in the ExpressRoute circuit, traffic is routed through an IPSec VPN tunnel. [Deploy this solution](#).

Note that if the ExpressRoute circuit is unavailable, the VPN route will only handle private peering connections. Public peering and Microsoft peering connections will pass over the Internet.



[Download a Visio file](#) of this architecture.

## Architecture

The architecture consists of the following components.

- **On-premises network.** A private local-area network running within an organization.
- **VPN appliance.** A device or service that provides external connectivity to the on-premises network. The VPN appliance may be a hardware device, or it can be a software solution such as the Routing and Remote Access Service (RRAS) in Windows Server 2012. For a list of supported VPN appliances and information on configuring selected VPN appliances for connecting to Azure, see [About VPN devices for Site-to-Site VPN Gateway connections](#).
- **ExpressRoute circuit.** A layer 2 or layer 3 circuit supplied by the connectivity provider that joins the on-premises network with Azure through the edge routers. The circuit uses the hardware infrastructure managed by the connectivity provider.
- **ExpressRoute virtual network gateway.** The ExpressRoute virtual network gateway enables the VNet to connect to the ExpressRoute circuit used for connectivity with your on-premises network.
- **VPN virtual network gateway.** The VPN virtual network gateway enables the VNet to connect to the VPN appliance in the on-premises network. The VPN virtual network gateway is configured to accept requests from the on-premises network only through the VPN appliance. For more information, see [Connect an on-premises network to a Microsoft Azure virtual network](#).
- **VPN connection.** The connection has properties that specify the connection type (IPSec) and the key shared with the on-premises VPN appliance to encrypt traffic.

- **Azure Virtual Network (VNet).** Each VNet resides in a single Azure region, and can host multiple application tiers. Application tiers can be segmented using subnets in each VNet.
- **Gateway subnet.** The virtual network gateways are held in the same subnet.
- **Cloud application.** The application hosted in Azure. It might include multiple tiers, with multiple subnets connected through Azure load balancers. For more information about the application infrastructure, see [Running Windows VM workloads](#) and [Running Linux VM workloads](#).

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### VNet and GatewaySubnet

Create the ExpressRoute virtual network gateway and the VPN virtual network gateway in the same VNet. This means that they should share the same subnet named *GatewaySubnet*.

If the VNet already includes a subnet named *GatewaySubnet*, ensure that it has a /27 or larger address space. If the existing subnet is too small, use the following PowerShell command to remove the subnet:

```
$vnet = Get-AzureRmVirtualNetworkGateway -Name <yourvnetname> -ResourceGroupName <yourresourcegroup>
Remove-AzureRmVirtualNetworkSubnetConfig -Name GatewaySubnet -VirtualNetwork $vnet
```

If the VNet does not contain a subnet named **GatewaySubnet**, create a new one using the following Powershell command:

```
$vnet = Get-AzureRmVirtualNetworkGateway -Name <yourvnetname> -ResourceGroupName <yourresourcegroup>
Add-AzureRmVirtualNetworkSubnetConfig -Name "GatewaySubnet" -VirtualNetwork $vnet -AddressPrefix
"10.200.255.224/27"
$vnet = Set-AzureRmVirtualNetwork -VirtualNetwork $vnet
```

### VPN and ExpressRoute gateways

Verify that your organization meets the [ExpressRoute prerequisite requirements](#) for connecting to Azure.

If you already have a VPN virtual network gateway in your Azure VNet, use the following Powershell command to remove it:

```
Remove-AzureRmVirtualNetworkGateway -Name <yourgatewayname> -ResourceGroupName <yourresourcegroup>
```

Follow the instructions in [Implementing a hybrid network architecture with Azure ExpressRoute](#) to establish your ExpressRoute connection.

Follow the instructions in [Implementing a hybrid network architecture with Azure and On-premises VPN](#) to establish your VPN virtual network gateway connection.

After you have established the virtual network gateway connections, test the environment as follows:

1. Make sure you can connect from your on-premises network to your Azure VNet.
2. Contact your provider to stop ExpressRoute connectivity for testing.
3. Verify that you can still connect from your on-premises network to your Azure VNet using the VPN virtual network gateway connection.
4. Contact your provider to reestablish ExpressRoute connectivity.

# Considerations

For ExpressRoute considerations, see the [Implementing a Hybrid Network Architecture with Azure ExpressRoute](#) guidance.

For site-to-site VPN considerations, see the [Implementing a Hybrid Network Architecture with Azure and On-premises VPN](#) guidance.

For general Azure security considerations, see [Microsoft cloud services and network security](#).

## Deploy the solution

**Prerequisites.** You must have an existing on-premises infrastructure already configured with a suitable network appliance.

To deploy the solution, perform the following steps.

1. Click the button below:



2. Wait for the link to open in the Azure portal, then follow these steps:

- The **Resource group** name is already defined in the parameter file, so select **Create New** and enter `ra-hybrid-vpn-er-rg` in the text box.
- Select the region from the **Location** drop down box.
- Do not edit the **Template Root Uri** or the **Parameter Root Uri** text boxes.
- Review the terms and conditions, then click the **I agree to the terms and conditions stated above** checkbox.
- Click the **Purchase** button.

3. Wait for the deployment to complete.

4. Click the button below:



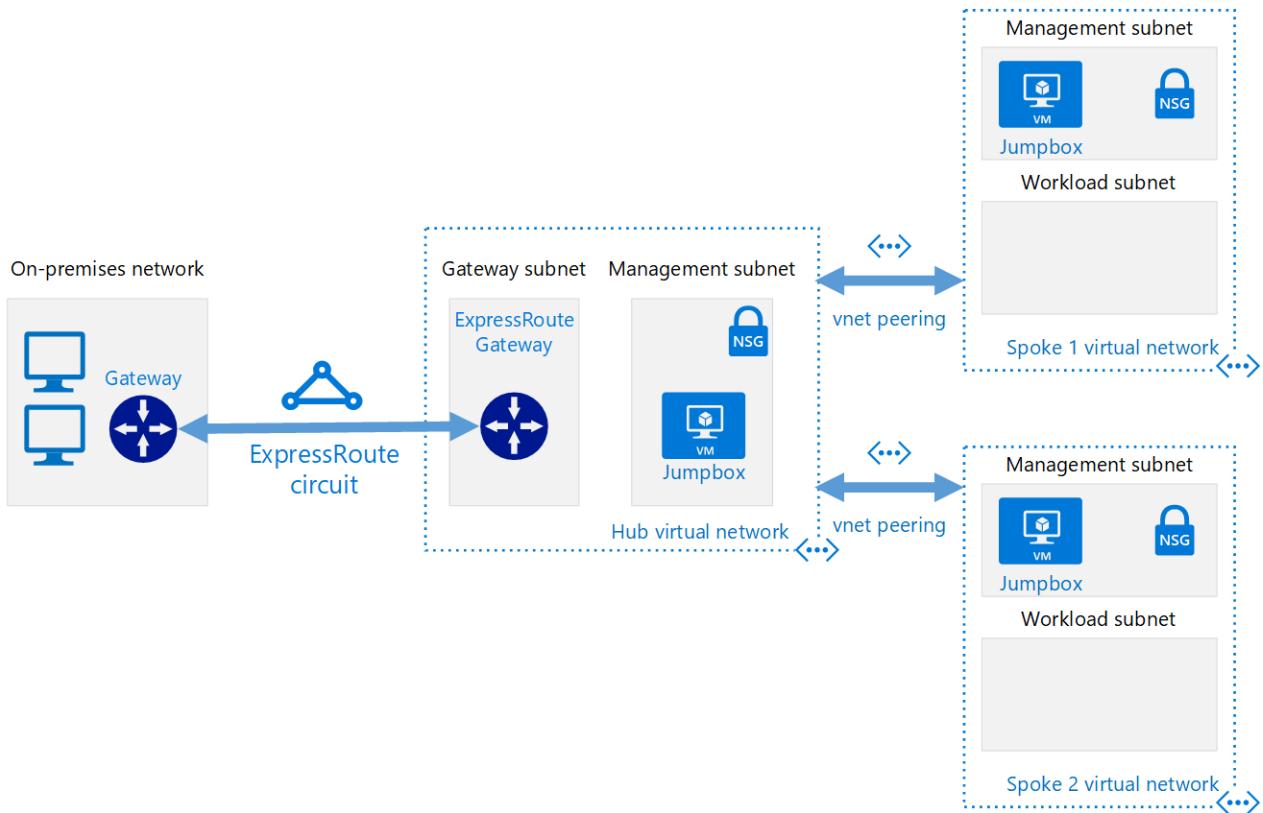
5. Wait for the link to open in the Azure portal, then enter then follow these steps:

- Select **Use existing** in the **Resource group** section and enter `ra-hybrid-vpn-er-rg` in the text box.
- Select the region from the **Location** drop down box.
- Do not edit the **Template Root Uri** or the **Parameter Root Uri** text boxes.
- Review the terms and conditions, then click the **I agree to the terms and conditions stated above** checkbox.
- Click the **Purchase** button.

# Implement a hub-spoke network topology in Azure

10/8/2018 • 10 minutes to read • [Edit Online](#)

This reference architecture shows how to implement a hub-spoke 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. Traffic flows between the on-premises datacenter and the hub through an ExpressRoute or VPN gateway connection. [Deploy this solution.](#)



Download a [Visio file](#) of this architecture

The benefits of this topology include:

- **Cost savings** by centralizing services that can be shared by multiple workloads, such as network virtual appliances (NVAs) and DNS servers, in a single location.
- **Overcome subscriptions limits** by peering VNets from different subscriptions to the central hub.
- **Separation of concerns** between central IT (SecOps, InfraOps) and workloads (DevOps).

Typical uses for this architecture include:

- Workloads deployed in different environments, such as development, testing, and production, that require shared services such as DNS, IDS, NTP, or AD DS. Shared services are placed in the hub VNet, while each environment is deployed to a spoke to maintain isolation.
- Workloads that do not require connectivity to each other, but require access to shared services.
- Enterprises that require central control over security aspects, such as a firewall in the hub as a DMZ, and segregated management for the workloads in each spoke.

## Architecture

The architecture consists of the following components.

- **On-premises network.** A private local-area network running within an organization.
- **VPN device.** A device or service that provides external connectivity to the on-premises network. The VPN device may be a hardware device, or a software solution such as the Routing and Remote Access Service (RRAS) in Windows Server 2012. For a list of supported VPN appliances and information on configuring selected VPN appliances for connecting to Azure, see [About VPN devices for Site-to-Site VPN Gateway connections](#).
- **VPN virtual network gateway or ExpressRoute gateway.** The virtual network gateway enables the VNet to connect to the VPN device, or ExpressRoute circuit, used for connectivity with your on-premises network. For more information, see [Connect an on-premises network to a Microsoft Azure virtual network](#).

**NOTE**

The deployment scripts for this reference architecture use a VPN gateway for connectivity, and a VNet in Azure to simulate your on-premises network.

- **Hub VNet.** Azure VNet used as the hub in the hub-spoke topology. The hub is the central point of connectivity to your on-premises network, and a place to host services that can be consumed by the different workloads hosted in the spoke VNets.
- **Gateway subnet.** The virtual network gateways are held in the same subnet.
- **Spoke VNets.** One or more Azure VNets that are used as spokes in the hub-spoke topology. Spokes can be used to isolate workloads in their own VNets, managed separately from other spokes. Each workload might include multiple tiers, with multiple subnets connected through Azure load balancers. For more information about the application infrastructure, see [Running Windows VM workloads](#) and [Running Linux VM workloads](#).
- **VNet peering.** Two VNets can be connected using a [peering connection](#). Peering connections are non-transitive, low latency connections between VNets. Once peered, the VNets exchange traffic by using the Azure backbone, without the need for a router. In a hub-spoke network topology, you use VNet peering to connect the hub to each spoke. You can peer virtual networks in the same region, or different regions. For more information, see [Requirements and constraints](#).

**NOTE**

This article only covers [Resource Manager](#) deployments, but you can also connect a classic VNet to a Resource Manager VNet in the same subscription. That way, your spokes can host classic deployments and still benefit from services shared in the hub.

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### Resource groups

The hub VNet, and each spoke VNet, can be implemented in different resource groups, and even different subscriptions. When you peer virtual networks in different subscriptions, both subscriptions can be associated to the same or different Azure Active Directory tenant. This allows for a decentralized management of each workload, while sharing services maintained in the hub VNet.

### VNet and GatewaySubnet

Create a subnet named *GatewaySubnet*, with an address range of /27. This subnet is required by the virtual

network gateway. Allocating 32 addresses to this subnet will help to prevent reaching gateway size limitations in the future.

For more information about setting up the gateway, see the following reference architectures, depending on your connection type:

- [Hybrid network using ExpressRoute](#)
- [Hybrid network using a VPN gateway](#)

For higher availability, you can use ExpressRoute plus a VPN for failover. See [Connect an on-premises network to Azure using ExpressRoute with VPN failover](#).

A hub-spoke topology can also be used without a gateway, if you don't need connectivity with your on-premises network.

### VNet peering

VNet peering is a non-transitive relationship between two VNets. If you require spokes to connect to each other, consider adding a separate peering connection between those spokes.

However, if you have several spokes that need to connect with each other, you will run out of possible peering connections very quickly due to the [limitation on number of VNets peerings per VNet](#). In this scenario, consider using user defined routes (UDRs) to force traffic destined to a spoke to be sent to an NVA acting as a router at the hub VNet. This will allow the spokes to connect to each other.

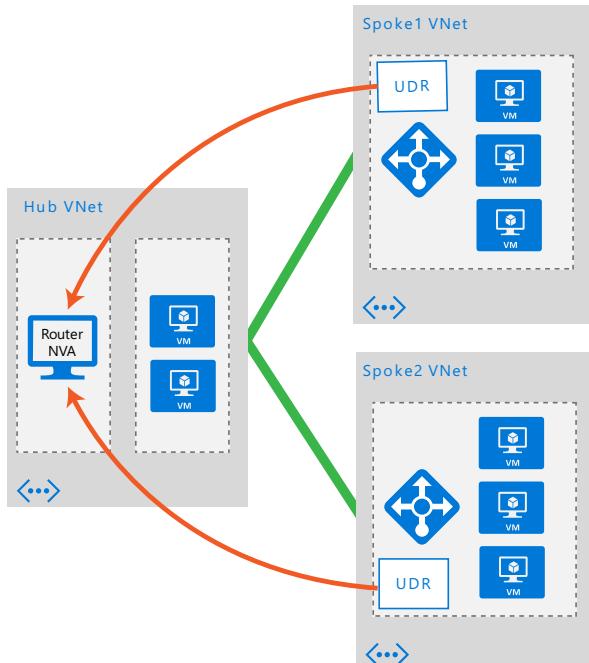
You can also configure spokes to use the hub VNet gateway to communicate with remote networks. To allow gateway traffic to flow from spoke to hub, and connect to remote networks, you must:

- Configure the VNet peering connection in the hub to **allow gateway transit**.
- Configure the VNet peering connection in each spoke to **use remote gateways**.
- Configure all VNet peering connections to **allow forwarded traffic**.

## Considerations

### Spoke connectivity

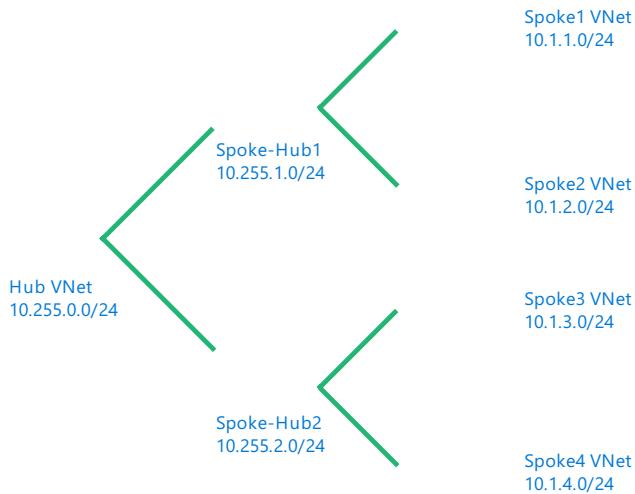
If you require connectivity between spokes, consider implementing an NVA for routing in the hub, and using UDRs in the spoke to forward traffic to the hub.



In this scenario, you must configure the peering connections to **allow forwarded traffic**.

## Overcoming VNet peering limits

Make sure you consider the [limitation on number of VNets peerings per VNet](#) in Azure. If you decide you need more spokes than the limit will allow, consider creating a hub-spoke-hub-spoke topology, where the first level of spokes also act as hubs. The following diagram shows this approach.



Also consider what services are shared in the hub, to ensure the hub scales for a larger number of spokes. For instance, if your hub provides firewall services, consider the bandwidth limits of your firewall solution when adding multiple spokes. You might want to move some of these shared services to a second level of hubs.

## Deploy the solution

A deployment for this architecture is available on [GitHub](#). It uses VMs in each VNet to test connectivity. There are no actual services hosted in the **shared-services** subnet in the **hub VNet**.

The deployment creates the following resource groups in your subscription:

- hub-nva-rg
- hub-vnet-rg
- onprem-jb-rg
- onprem-vnet-rg
- spoke1-vnet-rg
- spoke2-vent-rg

The template parameter files refer to these names, so if you change them, update the parameter files to match.

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Deploy the simulated on-premises datacenter

To deploy the simulated on-premises datacenter as an Azure VNet, follow these steps:

1. Navigate to the `hybrid-networking/hub-spoke` folder of the reference architectures repository.

2. Open the `onprem.json` file. Replace the values for `adminUsername` and `adminPassword`.

```
"adminUsername": "<user name>",
"adminPassword": "<password>,"
```

3. (Optional) For a Linux deployment, set `osType` to `Linux`.

4. Run the following command:

```
azbb -s <subscription_id> -g onprem-vnet-rg -l <location> -p onprem.json --deploy
```

5. Wait for the deployment to finish. This deployment creates a virtual network, a virtual machine, and a VPN gateway. It can take about 40 minutes to create the VPN gateway.

## Deploy the hub VNet

To deploy the hub VNet, perform the following steps.

1. Open the `hub-vnet.json` file. Replace the values for `adminUsername` and `adminPassword`.

```
"adminUsername": "<user name>",
"adminPassword": "<password>,"
```

2. (Optional) For a Linux deployment, set `osType` to `Linux`.

3. Find both instances of `sharedKey` and enter a shared key for the VPN connection. The values must match.

```
"sharedKey": "",
```

4. Run the following command:

```
azbb -s <subscription_id> -g hub-vnet-rg -l <location> -p hub-vnet.json --deploy
```

5. Wait for the deployment to finish. This deployment creates a virtual network, a virtual machine, a VPN gateway, and a connection to the gateway. It can take about 40 minutes to create the VPN gateway.

## Test connectivity with the hub

Test connectivity from the simulated on-premises environment to the hub VNet.

## Windows deployment

1. Use the Azure portal to find the VM named `jb-vm1` in the `onprem-jb-rg` resource group.

2. Click `Connect` to open a remote desktop session to the VM. Use the password that you specified in the `onprem.json` parameter file.

3. Open a PowerShell console in the VM, and use the `Test-NetConnection` cmdlet to verify that you can connect to the jumpbox VM in the hub VNet.

```
Test-NetConnection 10.0.0.68 -CommonTCPPort RDP
```

The output should look similar to the following:

```
ComputerName      : 10.0.0.68
RemoteAddress     : 10.0.0.68
RemotePort        : 3389
InterfaceAlias    : Ethernet 2
SourceAddress     : 192.168.1.000
TcpTestSucceeded  : True
```

#### NOTE

By default, Windows Server VMs do not allow ICMP responses in Azure. If you want to use `ping` to test connectivity, you need to enable ICMP traffic in the Windows Advanced Firewall for each VM.

## Linux deployment

1. Use the Azure portal to find the VM named `jb-vm1` in the `onprem-jb-rg` resource group.
2. Click `Connect` and copy the `ssh` command shown in the portal.
3. From a Linux prompt, run `ssh` to connect to the simulated on-premises environment. Use the password that you specified in the `onprem.json` parameter file.
4. Use the `ping` command to test connectivity to the jumpbox VM in the hub VNet:

```
ping 10.0.0.68
```

## Deploy the spoke VNets

To deploy the spoke VNets, perform the following steps.

1. Open the `spoke1.json` file. Replace the values for `adminUsername` and `adminPassword`.

```
"adminUsername": "<user name>",
"adminPassword": "<password>,"
```

2. (Optional) For a Linux deployment, set `osType` to `Linux`.

3. Run the following command:

```
azbb -s <subscription_id> -g spoke1-vnet-rg -l <location> -p spoke1.json --deploy
```

4. Repeat steps 1-2 for the `spoke2.json` file.

5. Run the following command:

```
azbb -s <subscription_id> -g spoke2-vnet-rg -l <location> -p spoke2.json --deploy
```

6. Run the following command:

```
azbb -s <subscription_id> -g hub-vnet-rg -l <location> -p hub-vnet-peering.json --deploy
```

## Test connectivity

Test connectivity from the simulated on-premises environment to the spoke VNets.

## Windows deployment

1. Use the Azure portal to find the VM named `jb-vm1` in the `onprem-jb-rg` resource group.
2. Click `Connect` to open a remote desktop session to the VM. Use the password that you specified in the `onprem.json` parameter file.
3. Open a PowerShell console in the VM, and use the `Test-NetConnection` cmdlet to verify that you can connect to the jumpbox VMs in the spoke VNets.

```
Test-NetConnection 10.1.0.68 -CommonTCPPort RDP  
Test-NetConnection 10.2.0.68 -CommonTCPPort RDP
```

## Linux deployment

To test connectivity from the simulated on-premises environment to the spoke VNets using Linux VMs, perform the following steps:

1. Use the Azure portal to find the VM named `jb-vm1` in the `onprem-jb-rg` resource group.
2. Click `Connect` and copy the `ssh` command shown in the portal.
3. From a Linux prompt, run `ssh` to connect to the simulated on-premises environment. Use the password that you specified in the `onprem.json` parameter file.
4. Use the `ping` command to test connectivity to the jumpbox VMs in each spoke:

```
ping 10.1.0.68  
ping 10.2.0.68
```

## Add connectivity between spokes

This step is optional. If you want to allow spokes to connect to each other, you must use a network virtual appliance (NVA) as a router in the hub VNet, and force traffic from spokes to the router when trying to connect to another spoke. To deploy a basic sample NVA as a single VM, along with user-defined routes (UDRs) to allow the two spoke VNets to connect, perform the following steps:

1. Open the `hub-nva.json` file. Replace the values for `adminUsername` and `adminPassword`.

```
"adminUsername": "<user name>","  
"adminPassword": "<password>,"
```

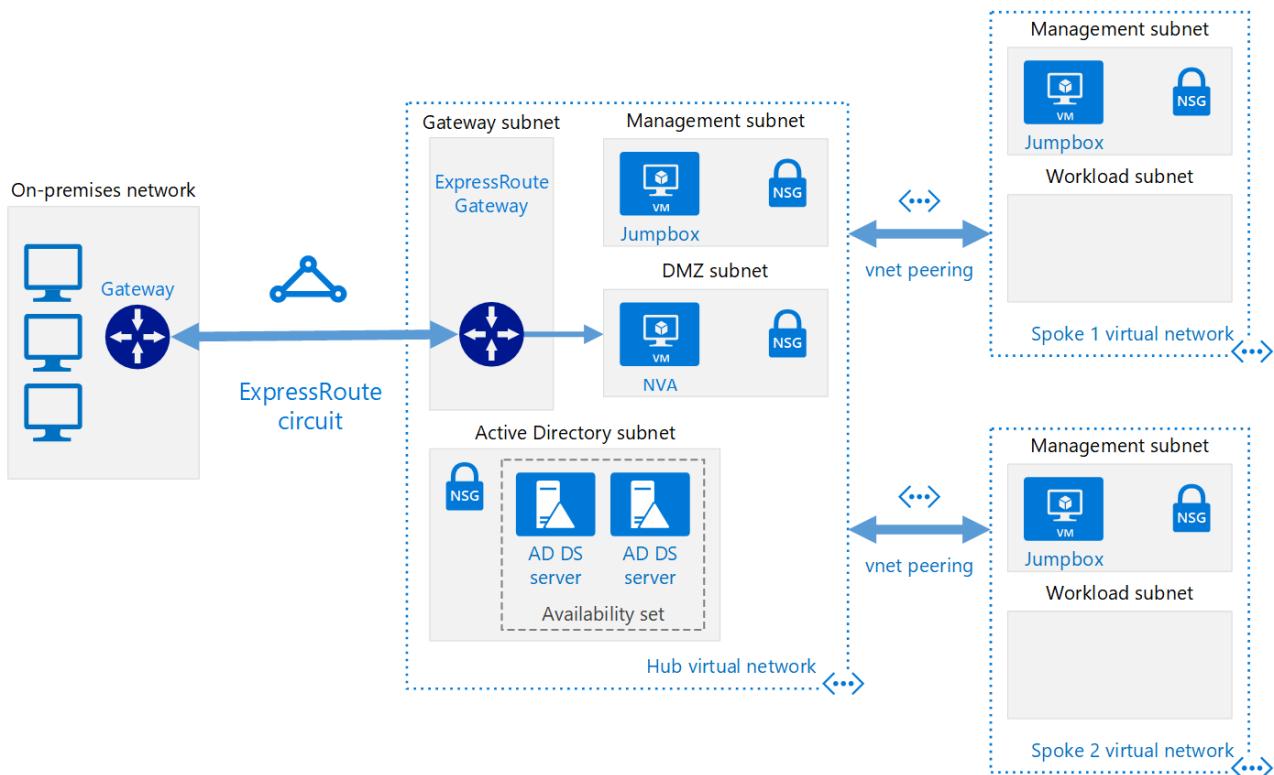
2. Run the following command:

```
azbb -s <subscription_id> -g hub-nva-rg -l <location> -p hub-nva.json --deploy
```

# Implement a hub-spoke network topology with shared services in Azure

10/9/2018 • 8 minutes to read • [Edit Online](#)

This reference architecture builds on the [hub-spoke](#) reference architecture to include shared services in the hub that can be consumed by all spokes. As a first step toward migrating a datacenter to the cloud, and building a [virtual datacenter](#), the first services you need to share are identity and security. This reference architecture shows you how to extend your Active Directory services from your on-premises datacenter to Azure, and how to add a network virtual appliance (NVA) that can act as a firewall, in a hub-spoke topology. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture

The benefits of this topology include:

- **Cost savings** by centralizing services that can be shared by multiple workloads, such as network virtual appliances (NVAs) and DNS servers, in a single location.
- **Overcome subscriptions limits** by peering VNets from different subscriptions to the central hub.
- **Separation of concerns** between central IT (SecOps, InfraOps) and workloads (DevOps).

Typical uses for this architecture include:

- Workloads deployed in different environments, such as development, testing, and production, that require shared services such as DNS, IDS, NTP, or AD DS. Shared services are placed in the hub VNet, while each environment is deployed to a spoke to maintain isolation.
- Workloads that do not require connectivity to each other, but require access to shared services.
- Enterprises that require central control over security aspects, such as a firewall in the hub as a DMZ, and segregated management for the workloads in each spoke.

## Architecture

The architecture consists of the following components.

- **On-premises network.** A private local-area network running within an organization.
- **VPN device.** A device or service that provides external connectivity to the on-premises network. The VPN device may be a hardware device, or a software solution such as the Routing and Remote Access Service (RRAS) in Windows Server 2012. For a list of supported VPN appliances and information on configuring selected VPN appliances for connecting to Azure, see [About VPN devices for Site-to-Site VPN Gateway connections](#).
- **VPN virtual network gateway or ExpressRoute gateway.** The virtual network gateway enables the VNet to connect to the VPN device, or ExpressRoute circuit, used for connectivity with your on-premises network. For more information, see [Connect an on-premises network to a Microsoft Azure virtual network](#).

#### NOTE

The deployment scripts for this reference architecture use a VPN gateway for connectivity, and a VNet in Azure to simulate your on-premises network.

- **Hub VNet.** Azure VNet used as the hub in the hub-spoke topology. The hub is the central point of connectivity to your on-premises network, and a place to host services that can be consumed by the different workloads hosted in the spoke VNets.
- **Gateway subnet.** The virtual network gateways are held in the same subnet.
- **Shared services subnet.** A subnet in the hub VNet used to host services that can be shared among all spokes, such as DNS or AD DS.
- **DMZ subnet.** A subnet in the hub VNet used to host NVAs that can act as security appliances, such as firewalls.
- **Spoke VNets.** One or more Azure VNets that are used as spokes in the hub-spoke topology. Spokes can be used to isolate workloads in their own VNets, managed separately from other spokes. Each workload might include multiple tiers, with multiple subnets connected through Azure load balancers. For more information about the application infrastructure, see [Running Windows VM workloads](#) and [Running Linux VM workloads](#).
- **VNet peering.** Two VNets in the same Azure region can be connected using a [peering connection](#). Peering connections are non-transitive, low latency connections between VNets. Once peered, the VNets exchange traffic by using the Azure backbone, without the need for a router. In a hub-spoke network topology, you use VNet peering to connect the hub to each spoke.

#### NOTE

This article only covers [Resource Manager](#) deployments, but you can also connect a classic VNet to a Resource Manager VNet in the same subscription. That way, your spokes can host classic deployments and still benefit from services shared in the hub.

## Recommendations

All the recommendations for the [hub-spoke](#) reference architecture also apply to the shared services reference architecture.

Also, the following recommendations apply for most scenarios under shared services. Follow these recommendations unless you have a specific requirement that overrides them.

## Identity

Most enterprise organizations have an Active Directory Directory Services (ADDS) environment in their on-premises datacenter. To facilitate management of assets moved to Azure from your on-premises network that depend on ADDS, it is recommended to host ADDS domain controllers in Azure.

If you make use of Group Policy Objects, that you want to control separately for Azure and your on-premises environment, use a different AD site for each Azure region. Place your domain controllers in a central VNet (hub) that dependent workloads can access.

## Security

As you move workloads from your on-premises environment to Azure, some of these workloads will require to be hosted in VMs. For compliance reasons, you may need to enforce restrictions on traffic traversing those workloads.

You can use network virtual appliances (NVAs) in Azure to host different types of security and performance services. If you are familiar with a given set of appliances on-premises today, it is recommended to use the same virtualized appliances in Azure, where applicable.

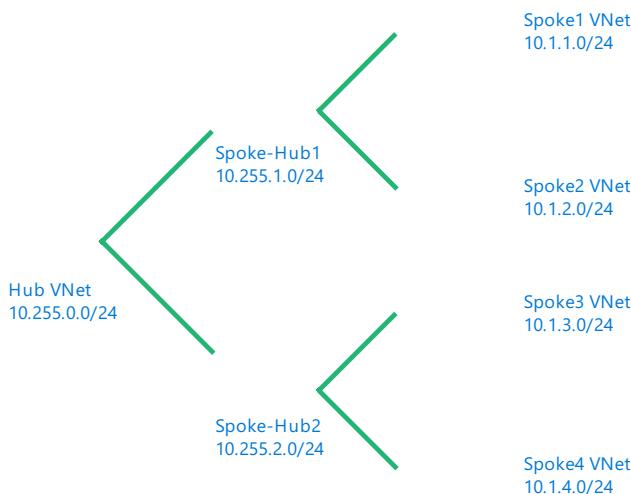
### NOTE

The deployment scripts for this reference architecture use an Ubuntu VM with IP forwarding enabled to mimic a network virtual appliance.

## Considerations

### Overcoming VNet peering limits

Make sure you consider the [limitation on number of VNets peerings per VNet](#) in Azure. If you decide you need more spokes than the limit will allow, consider creating a hub-spoke-hub-spoke topology, where the first level of spokes also act as hubs. The following diagram shows this approach.



Also consider what services are shared in the hub, to ensure the hub scales for a larger number of spokes. For instance, if your hub provides firewall services, consider the bandwidth limits of your firewall solution when adding multiple spokes. You might want to move some of these shared services to a second level of hubs.

## Deploy the solution

A deployment for this architecture is available on [GitHub](#). The deployment creates the following resource groups in your subscription:

- hub-adds-rg
- hub-nva-rg

- hub-vnet-rg
- onprem-vnet-rg
- spoke1-vnet-rg
- spoke2-vnet-rg

The template parameter files refer to these names, so if you change them, update the parameter files to match.

## Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

## Deploy the simulated on-premises datacenter using azbb

This step deploys the simulated on-premises datacenter as an Azure VNet.

1. Navigate to the `hybrid-networking\shared-services-stack\` folder of the GitHub repository.
2. Open the `onprem.json` file.
3. Search for all instances of `UserName`, `adminUserName`, `Password`, and `adminPassword`. Enter values for the user name and password in the parameters and save the file.
4. Run the following command:

```
azbb -s <subscription_id> -g onprem-vnet-rg -l <location> -p onprem.json --deploy
```

5. Wait for the deployment to finish. This deployment creates a virtual network, a virtual machine running Windows, and a VPN gateway. The VPN gateway creation can take more than 40 minutes to complete.

## Deploy the hub VNet

This step deploys the hub VNet and connects it to the simulated on-premises VNet.

1. Open the `hub-vnet.json` file.
2. Search for `adminPassword` and enter a user name and password in the parameters.
3. Search for all instances of `sharedKey` and enter a value for a shared key. Save the file.

```
"sharedKey": "abc123",
```

4. Run the following command:

```
azbb -s <subscription_id> -g hub-vnet-rg -l <location> -p hub-vnet.json --deploy
```

5. Wait for the deployment to finish. This deployment creates a virtual network, a virtual machine, a VPN gateway, and a connection to the gateway created in the previous section. The VPN gateway can take more

than 40 minutes to complete.

## Deploy AD DS in Azure

This step deploys AD DS domain controllers in Azure.

1. Open the `hub-adds.json` file.
2. Search for all instances of `Password` and `adminPassword`. Enter values for the user name and password in the parameters and save the file.
3. Run the following command:

```
azbb -s <subscription_id> -g hub-adds-rg -l <location> -p hub-adds.json --deploy
```

This deployment step may take several minutes, because it joins the two VMs to the domain hosted in the simulated on-premises datacenter, and installs AD DS on them.

## Deploy the spoke VNets

This step deploys the spoke VNets.

1. Open the `spoke1.json` file.
2. Search for `adminPassword` and enter a user name and password in the parameters.
3. Run the following command:

```
azbb -s <subscription_id> -g spoke1-vnet-rg -l <location> -p spoke1.json --deploy
```

4. Repeat steps 1 and 2 for the file `spoke2.json`.

5. Run the following command:

```
azbb -s <subscription_id> -g spoke2-vnet-rg -l <location> -p spoke2.json --deploy
```

## Peer the hub VNet to the spoke VNets

To create a peering connection from the hub VNet to the spoke VNets, run the following command:

```
azbb -s <subscription_id> -g hub-vnet-rg -l <location> -p hub-vnet-peering.json --deploy
```

## Deploy the NVA

This step deploys an NVA in the `dmz` subnet.

1. Open the `hub-nva.json` file.
2. Search for `adminPassword` and enter a user name and password in the parameters.
3. Run the following command:

```
azbb -s <subscription_id> -g hub-nva-rg -l <location> -p hub-nva.json --deploy
```

## Test connectivity

Test connectivity from the simulated on-premises environment to the hub VNet.

1. Use the Azure portal to find the VM named `jb-vm1` in the `onprem-jb-rg` resource group.

2. Click `Connect` to open a remote desktop session to the VM. Use the password that you specified in the `onprem.json` parameter file.
3. Open a PowerShell console in the VM, and use the `Test-NetConnection` cmdlet to verify that you can connect to the jumpbox VM in the hub VNet.

```
Test-NetConnection 10.0.0.68 -CommonTCPPort RDP
```

The output should look similar to the following:

```
ComputerName      : 10.0.0.68
RemoteAddress     : 10.0.0.68
RemotePort        : 3389
InterfaceAlias    : Ethernet 2
SourceAddress     : 192.168.1.000
TcpTestSucceeded  : True
```

**NOTE**

By default, Windows Server VMs do not allow ICMP responses in Azure. If you want to use `ping` to test connectivity, you need to enable ICMP traffic in the Windows Advanced Firewall for each VM.

Repeat the same steps to test connectivity to the spoke VNets:

```
Test-NetConnection 10.1.0.68 -CommonTCPPort RDP
Test-NetConnection 10.2.0.68 -CommonTCPPort RDP
```

# Choose a solution for integrating on-premises Active Directory with Azure

9/28/2018 • 4 minutes to read • [Edit Online](#)

This article compares options for integrating your on-premises Active Directory (AD) environment with an Azure network. For each option, a more detailed reference architecture is available.

Many organizations use Active Directory Domain Services (AD DS) to authenticate identities associated with users, computers, applications, or other resources that are included in a security boundary. Directory and identity services are typically hosted on-premises, but if your application is hosted partly on-premises and partly in Azure, there may be latency sending authentication requests from Azure back to on-premises. Implementing directory and identity services in Azure can reduce this latency.

Azure provides two solutions for implementing directory and identity services in Azure:

- Use [Azure AD](#) to create an Active Directory domain in the cloud and connect it to your on-premises Active Directory domain. [Azure AD Connect](#) integrates your on-premises directories with Azure AD.
- Extend your existing on-premises Active Directory infrastructure to Azure, by deploying a VM in Azure that runs AD DS as a domain controller. This architecture is more common when the on-premises network and the Azure virtual network (VNet) are connected by a VPN or ExpressRoute connection. Several variations of this architecture are possible:
  - Create a domain in Azure and join it to your on-premises AD forest.
  - Create a separate forest in Azure that is trusted by domains in your on-premises forest.
  - Replicate an Active Directory Federation Services (AD FS) deployment to Azure.

The next sections describe each of these options in more detail.

## Integrate your on-premises domains with Azure AD

Use Azure Active Directory (Azure AD) to create a domain in Azure and link it to an on-premises AD domain.

The Azure AD directory is not an extension of an on-premises directory. Rather, it's a copy that contains the same objects and identities. Changes made to these items on-premises are copied to Azure AD, but changes made in Azure AD are not replicated back to the on-premises domain.

You can also use Azure AD without using an on-premises directory. In this case, Azure AD acts as the primary source of all identity information, rather than containing data replicated from an on-premises directory.

### Benefits

- You don't need to maintain an AD infrastructure in the cloud. Azure AD is entirely managed and maintained by Microsoft.
- Azure AD provides the same identity information that is available on-premises.
- Authentication can happen in Azure, reducing the need for external applications and users to contact the on-premises domain.

### Challenges

- Identity services are limited to users and groups. There is no ability to authenticate service and computer accounts.

- You must configure connectivity with your on-premises domain to keep the Azure AD directory synchronized.
- Applications may need to be rewritten to enable authentication through Azure AD.

### Reference architecture

- [Integrate on-premises Active Directory domains with Azure Active Directory](#)

## AD DS in Azure joined to an on-premises forest

Deploy AD Domain Services (AD DS) servers to Azure. Create a domain in Azure and join it to your on-premises AD forest.

Consider this option if you need to use AD DS features that are not currently implemented by Azure AD.

### Benefits

- Provides access to the same identity information that is available on-premises.
- You can authenticate user, service, and computer accounts on-premises and in Azure.
- You don't need to manage a separate AD forest. The domain in Azure can belong to the on-premises forest.
- You can apply group policy defined by on-premises Group Policy Objects to the domain in Azure.

### Challenges

- You must deploy and manage your own AD DS servers and domain in the cloud.
- There may be some synchronization latency between the domain servers in the cloud and the servers running on-premises.

### Reference architecture

- [Extend Active Directory Domain Services \(AD DS\) to Azure](#)

## AD DS in Azure with a separate forest

Deploy AD Domain Services (AD DS) servers to Azure, but create a separate Active Directory **forest** that is separate from the on-premises forest. This forest is trusted by domains in your on-premises forest.

Typical uses for this architecture include maintaining security separation for objects and identities held in the cloud, and migrating individual domains from on-premises to the cloud.

### Benefits

- You can implement on-premises identities and separate Azure-only identities.
- You don't need to replicate from the on-premises AD forest to Azure.

### Challenges

- Authentication within Azure for on-premises identities requires extra network hops to the on-premises AD servers.
- You must deploy your own AD DS servers and forest in the cloud, and establish the appropriate trust relationships between forests.

### Reference architecture

- [Create an Active Directory Domain Services \(AD DS\) resource forest in Azure](#)

## Extend AD FS to Azure

Replicate an Active Directory Federation Services (AD FS) deployment to Azure, to perform federated

authentication and authorization for components running in Azure.

Typical uses for this architecture:

- Authenticate and authorize users from partner organizations.
- Allow users to authenticate from web browsers running outside of the organizational firewall.
- Allow users to connect from authorized external devices such as mobile devices.

## **Benefits**

- You can leverage claims-aware applications.
- Provides the ability to trust external partners for authentication.
- Compatibility with large set of authentication protocols.

## **Challenges**

- You must deploy your own AD DS, AD FS, and AD FS Web Application Proxy servers in Azure.
- This architecture can be complex to configure.

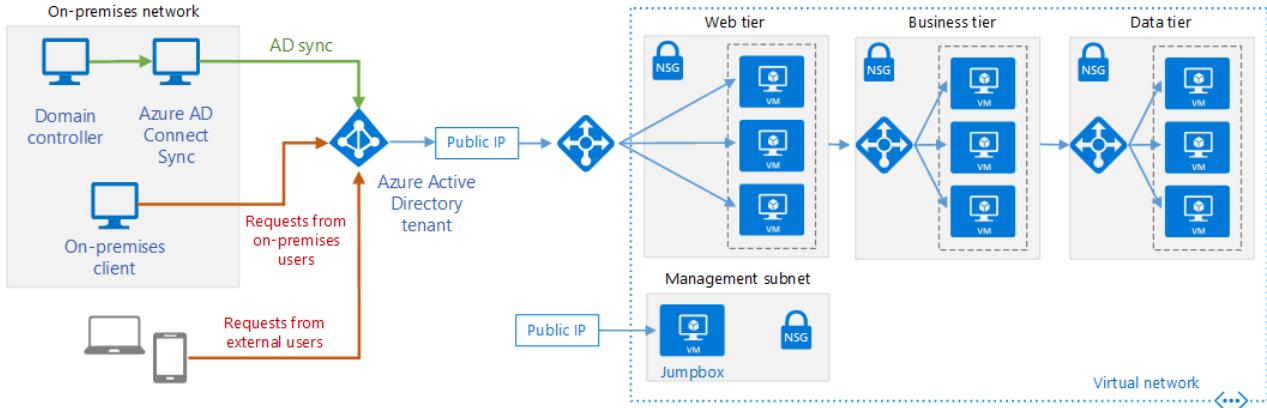
## **Reference architecture**

- [Extend Active Directory Federation Services \(AD FS\) to Azure](#)

# Integrate on-premises Active Directory domains with Azure Active Directory

10/5/2018 • 16 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) is a cloud based multi-tenant directory and identity service. This reference architecture shows best practices for integrating on-premises Active Directory domains with Azure AD to provide cloud-based identity authentication. [Deploy this solution.](#)



Download a [Visio file](#) of this architecture.

## NOTE

For simplicity, this diagram only shows the connections directly related to Azure AD, and not protocol-related traffic that may occur as part of authentication and identity federation. For example, a web application may redirect the web browser to authenticate the request through Azure AD. Once authenticated, the request can be passed back to the web application, with the appropriate identity information.

Typical uses for this reference architecture include:

- Web applications deployed in Azure that provide access to remote users who belong to your organization.
- Implementing self-service capabilities for end-users, such as resetting their passwords, and delegating group management. Note that this requires Azure AD Premium edition.
- Architectures in which the on-premises network and the application's Azure VNet are not connected using a VPN tunnel or ExpressRoute circuit.

## NOTE

Azure AD can authenticate the identity of users and applications that exist in an organization's directory. Some applications and services, such as SQL Server, may require computer authentication, in which case this solution is not appropriate.

For additional considerations, see [Choose a solution for integrating on-premises Active Directory with Azure](#).

## Architecture

The architecture has the following components.

- **Azure AD tenant.** An instance of Azure AD created by your organization. It acts as a directory service for cloud applications by storing objects copied from the on-premises Active Directory and provides identity

services.

- **Web tier subnet.** This subnet holds VMs that run a web application. Azure AD can act as an identity broker for this application.
- **On-premises AD DS server.** An on-premise directory and identity service. The AD DS directory can be synchronized with Azure AD to enable it to authenticate on-premise users.
- **Azure AD Connect sync server.** An on-premises computer that runs the [Azure AD Connect](#) sync service. This service synchronizes information held in the on-premises Active Directory to Azure AD. For example, if you provision or deprovision groups and users on-premises, these changes propagate to Azure AD.

**NOTE**

For security reasons, Azure AD stores user's passwords as a hash. If a user requires a password reset, this must be performed on-premises and the new hash must be sent to Azure AD. Azure AD Premium editions include features that can automate this task to enable users to reset their own passwords.

- **VMs for N-tier application.** The deployment includes infrastructure for an N-tier application. For more information about these resources, see [Run VMs for an N-tier architecture](#).

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### Azure AD Connect sync service

The Azure AD Connect sync service ensures that identity information stored in the cloud is consistent with that held on-premises. You install this service using the Azure AD Connect software.

Before implementing Azure AD Connect sync, determine the synchronization requirements of your organization. For example, what to synchronize, from which domains, and how frequently. For more information, see [Determine directory synchronization requirements](#).

You can run the Azure AD Connect sync service on a VM or a computer hosted on-premises. Depending on the volatility of the information in your Active Directory directory, the load on the Azure AD Connect sync service is unlikely to be high after the initial synchronization with Azure AD. Running the service on a VM makes it easier to scale the server if needed. Monitor the activity on the VM as described in the Monitoring considerations section to determine whether scaling is necessary.

If you have multiple on-premises domains in a forest, we recommend storing and synchronizing information for the entire forest to a single Azure AD tenant. Filter information for identities that occur in more than one domain, so that each identity appears only once in Azure AD, rather than being duplicated. Duplication can lead to inconsistencies when data is synchronized. For more information, see the Topology section below.

Use filtering so that only necessary data is stored in Azure AD. For example, your organization might not want to store information about inactive accounts in Azure AD. Filtering can be group-based, domain-based, organization unit (OU)-based, or attribute-based. You can combine filters to generate more complex rules. For example, you could synchronize objects held in a domain that have a specific value in a selected attribute. For detailed information, see [Azure AD Connect sync: Configure Filtering](#).

To implement high availability for the AD Connect sync service, run a secondary staging server. For more information, see the Topology recommendations section.

### Security recommendations

**User password management.** The Azure AD Premium editions support password writeback, enabling your on-

on-premises users to perform self-service password resets from within the Azure portal. This feature should only be enabled after reviewing your organization's password security policy. For example, you can restrict which users can change their passwords, and you can tailor the password management experience. For more information, see [Customizing Password Management to fit your organization's needs](#).

**Protect on-premises applications that can be accessed externally.** Use the Azure AD Application Proxy to provide controlled access to on-premises web applications for external users through Azure AD. Only users that have valid credentials in your Azure directory have permission to use the application. For more information, see the article [Enable Application Proxy in the Azure portal](#).

**Actively monitor Azure AD for signs of suspicious activity.** Consider using Azure AD Premium P2 edition, which includes Azure AD Identity Protection. Identity Protection uses adaptive machine learning algorithms and heuristics to detect anomalies and risk events that may indicate that an identity has been compromised. For example, it can detect potentially unusual activity such as irregular sign-in activities, sign-ins from unknown sources or from IP addresses with suspicious activity, or sign-ins from devices that may be infected. Using this data, Identity Protection generates reports and alerts that enables you to investigate these risk events and take appropriate action. For more information, see [Azure Active Directory Identity Protection](#).

You can use the reporting feature of Azure AD in the Azure portal to monitor security-related activities occurring in your system. For more information about using these reports, see [Azure Active Directory Reporting Guide](#).

## Topology recommendations

Configure Azure AD Connect to implement a topology that most closely matches the requirements of your organization. Topologies that Azure AD Connect supports include the following:

- **Single forest, single Azure AD directory.** In this topology, Azure AD Connect synchronizes objects and identity information from one or more domains in a single on-premises forest into a single Azure AD tenant. This is the default topology implemented by the express installation of Azure AD Connect.

### NOTE

Don't use multiple Azure AD Connect sync servers to connect different domains in the same on-premises forest to the same Azure AD tenant, unless you are running a server in staging mode, described below.

- **Multiple forests, single Azure AD directory.** In this topology, Azure AD Connect synchronizes objects and identity information from multiple forests into a single Azure AD tenant. Use this topology if your organization has more than one on-premises forest. You can consolidate identity information so that each unique user is represented once in the Azure AD directory, even if the same user exists in more than one forest. All forests use the same Azure AD Connect sync server. The Azure AD Connect sync server does not have to be part of any domain, but it must be reachable from all forests.

### NOTE

In this topology, don't use separate Azure AD Connect sync servers to connect each on-premises forest to a single Azure AD tenant. This can result in duplicated identity information in Azure AD if users are present in more than one forest.

- **Multiple forests, separate topologies.** This topology merges identity information from separate forests into a single Azure AD tenant, treating all forests as separate entities. This topology is useful if you are combining forests from different organizations and the identity information for each user is held in only one forest.

#### **NOTE**

If the global address lists (GAL) in each forest are synchronized, a user in one forest may be present in another as a contact. This can occur if your organization has implemented GALSync with Forefront Identity manager 2010 or Microsoft Identity Manager 2016. In this scenario, you can specify that users should be identified by their *Mail* attribute. You can also match identities using the *ObjectSID* and *msExchMasterAccountSID* attributes. This is useful if you have one or more resource forests with disabled accounts.

- **Staging server.** In this configuration, you run a second instance of the Azure AD Connect sync server in parallel with the first. This structure supports scenarios such as:

- High availability.
- Testing and deploying a new configuration of the Azure AD Connect sync server.
- Introducing a new server and decommissioning an old configuration.

In these scenarios, the second instance runs in *staging mode*. The server records imported objects and synchronization data in its database, but does not pass the data to Azure AD. If you disable staging mode, the server starts writing data to Azure AD, and also starts performing password write-backs into the on-premises directories where appropriate. For more information, see [Azure AD Connect sync: Operational tasks and considerations](#).

- **Multiple Azure AD directories.** It is recommended that you create a single Azure AD directory for an organization, but there may be situations where you need to partition information across separate Azure AD directories. In this case, avoid synchronization and password write-back issues by ensuring that each object from the on-premises forest appears in only one Azure AD directory. To implement this scenario, configure separate Azure AD Connect sync servers for each Azure AD directory, and use filtering so that each Azure AD Connect sync server operates on a mutually exclusive set of objects.

For more information about these topologies, see [Topologies for Azure AD Connect](#).

#### **User authentication**

By default, the Azure AD Connect sync server configures password hash synchronization between the on-premises domain and Azure AD, and the Azure AD service assumes that users authenticate by providing the same password that they use on-premises. For many organizations, this is appropriate, but you should consider your organization's existing policies and infrastructure. For example:

- The security policy of your organization may prohibit synchronizing password hashes to the cloud. In this case your organization should consider [pass-through authentication](#).
- You might require that users experience seamless single sign-on (SSO) when accessing cloud resources from domain-joined machines on the corporate network.
- Your organization might already have Active Directory Federation Services (AD FS) or a third party federation provider deployed. You can configure Azure AD to use this infrastructure to implement authentication and SSO rather than by using password information held in the cloud.

For more information, see [Azure AD Connect User Sign on options](#).

#### **Azure AD application proxy**

Use Azure AD to provide access to on-premises applications.

Expose your on-premises web applications using application proxy connectors managed by the Azure AD application proxy component. The application proxy connector opens an outbound network connection to the Azure AD application proxy, and remote users' requests are routed back from Azure AD through this connection to the web apps. This removes the need to open inbound ports in the on-premises firewall and reduces the attack surface exposed by your organization.

For more information, see [Publish applications using Azure AD Application proxy](#).

## Object synchronization

Azure AD Connect's default configuration synchronizes objects from your local Active Directory directory based on the rules specified in the article [Azure AD Connect sync: Understanding the default configuration](#). Objects that satisfy these rules are synchronized while all other objects are ignored. Some example rules:

- User objects must have a unique *sourceAnchor* attribute and the *accountEnabled* attribute must be populated.
- User objects must have a *sAMAccountName* attribute and cannot start with the text *Azure AD\_* or *MSOL\_*.

Azure AD Connect applies several rules to User, Contact, Group, ForeignSecurityPrincipal, and Computer objects. Use the Synchronization Rules Editor installed with Azure AD Connect if you need to modify the default set of rules. For more information, see [Azure AD Connect sync: Understanding the default configuration](#).

You can also define your own filters to limit the objects to be synchronized by domain or OU. Alternatively, you can implement more complex custom filtering such as that described in [Azure AD Connect sync: Configure Filtering](#).

## Monitoring

Health monitoring is performed by the following agents installed on-premises:

- Azure AD Connect installs an agent that captures information about synchronization operations. Use the Azure AD Connect Health blade in the Azure portal to monitor its health and performance. For more information, see [Using Azure AD Connect Health for sync](#).
- To monitor the health of the AD DS domains and directories from Azure, install the Azure AD Connect Health for AD DS agent on a machine within the on-premises domain. Use the Azure Active Directory Connect Health blade in the Azure portal for health monitoring. For more information, see [Using Azure AD Connect Health with AD DS](#)
- Install the Azure AD Connect Health for AD FS agent to monitor the health of services running on on-premises, and use the Azure Active Directory Connect Health blade in the Azure portal to monitor AD FS. For more information, see [Using Azure AD Connect Health with AD FS](#)

For more information on installing the AD Connect Health agents and their requirements, see [Azure AD Connect Health Agent Installation](#).

## Scalability considerations

The Azure AD service supports scalability based on replicas, with a single primary replica that handles write operations plus multiple read-only secondary replicas. Azure AD transparently redirects attempted writes made against secondary replicas to the primary replica and provides eventual consistency. All changes made to the primary replica are propagated to the secondary replicas. This architecture scales well because most operations against Azure AD are reads rather than writes. For more information, see [Azure AD: Under the hood of our geo-redundant, highly available, distributed cloud directory](#).

For the Azure AD Connect sync server, determine how many objects you are likely to synchronize from your local directory. If you have less than 100,000 objects, you can use the default SQL Server Express LocalDB software provided with Azure AD Connect. If you have a larger number of objects, you should install a production version of SQL Server and perform a custom installation of Azure AD Connect, specifying that it should use an existing instance of SQL Server.

## Availability considerations

The Azure AD service is geo-distributed and runs in multiple data centers spread around the world with automated failover. If a data center becomes unavailable, Azure AD ensures that your directory data is available for instance access in at least two more regionally dispersed data centers.

#### **NOTE**

The service level agreement (SLA) for Azure AD Basic and Premium services guarantees at least 99.9% availability. There is no SLA for the Free tier of Azure AD. For more information, see [SLA for Azure Active Directory](#).

Consider provisioning a second instance of Azure AD Connect sync server in staging mode to increase availability, as discussed in the topology recommendations section.

If you are not using the SQL Server Express LocalDB instance that comes with Azure AD Connect, consider using SQL clustering to achieve high availability. Solutions such as mirroring and Always On are not supported by Azure AD Connect.

For additional considerations about achieving high availability of the Azure AD Connect sync server and also how to recover after a failure, see [Azure AD Connect sync: Operational tasks and considerations - Disaster Recovery](#).

## Manageability considerations

There are two aspects to managing Azure AD:

- Administering Azure AD in the cloud.
- Maintaining the Azure AD Connect sync servers.

Azure AD provides the following options for managing domains and directories in the cloud:

- **Azure Active Directory PowerShell Module.** Use this [module](#) if you need to script common Azure AD administrative tasks such as user management, domain management, and configuring single sign-on.
- **Azure AD management blade in the Azure portal.** This blade provides an interactive management view of the directory, and enables you to control and configure most aspects of Azure AD.

Azure AD Connect installs the following tools to maintain Azure AD Connect sync services from your on-premises machines:

- **Microsoft Azure Active Directory Connect console.** This tool enables you to modify the configuration of the Azure AD Sync server, customize how synchronization occurs, enable or disable staging mode, and switch the user sign-in mode. Note that you can enable Active Directory FS sign-in using your on-premises infrastructure.
- **Synchronization Service Manager.** Use the *Operations* tab in this tool to manage the synchronization process and detect whether any parts of the process have failed. You can trigger synchronizations manually using this tool. The *Connectors* tab enables you to control the connections for the domains that the synchronization engine is attached to.
- **Synchronization Rules Editor.** Use this tool to customize the way objects are transformed when they are copied between an on-premises directory and Azure AD. This tool enables you to specify additional attributes and objects for synchronization, then executes filters to determine which objects should or should not be synchronized. For more information, see the Synchronization Rule Editor section in the document [Azure AD Connect sync: Understanding the default configuration](#).

For more information and tips for managing Azure AD Connect, see [Azure AD Connect sync: Best practices for changing the default configuration](#).

## Security considerations

Use conditional access control to deny authentication requests from unexpected sources:

- Trigger [Azure Multi-Factor Authentication \(MFA\)](#) if a user attempts to connect from a nontrusted location such as across the Internet instead of a trusted network.

- Use the device platform type of the user (iOS, Android, Windows Mobile, Windows) to determine access policy to applications and features.
- Record the enabled/disabled state of users' devices, and incorporate this information into the access policy checks. For example, if a user's phone is lost or stolen it should be recorded as disabled to prevent it from being used to gain access.
- Control user access to resources based on group membership. Use [Azure AD dynamic membership rules](#) to simplify group administration. For a brief overview of how this works, see [Introduction to Dynamic Memberships for Groups](#).
- Use conditional access risk policies with Azure AD Identity Protection to provide advanced protection based on unusual sign-in activities or other events.

For more information, see [Azure Active Directory conditional access](#).

## Deploy the solution

A deployment for a reference architecture that implements these recommendations and considerations is available on GitHub. This reference architecture deploys a simulated on-premises network in Azure that you can use to test and experiment. The reference architecture can be deployed with either with Windows or Linux VMs by following the directions below:

1. Click the button below:



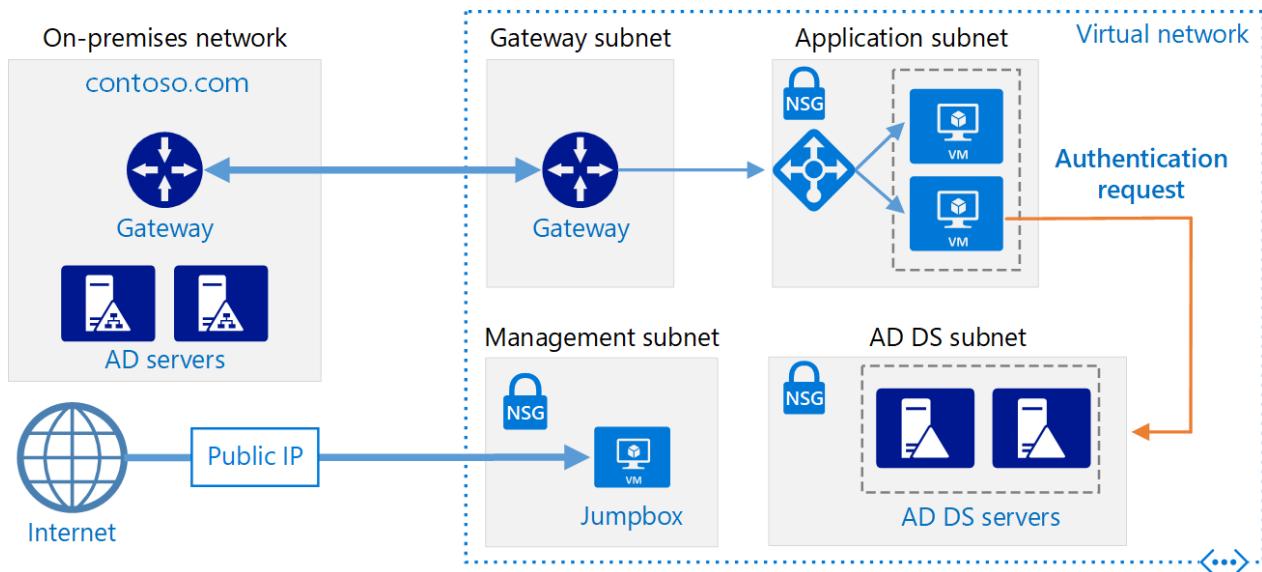
2. Once the link has opened in the Azure portal, you must enter values for some of the settings:

- The **Resource group** name is already defined in the parameter file, so select **Create New** and enter `ra-aad-onpremise-rg` in the text box.
  - Select the region from the **Location** drop down box.
  - Do not edit the **Template Root Uri** or the **Parameter Root Uri** text boxes.
  - Select **windows** or **linux** in the **Os Type** the drop down box.
  - Review the terms and conditions, then click the **I agree to the terms and conditions stated above** checkbox.
  - Click the **Purchase** button.
3. Wait for the deployment to complete.
  4. The parameter files include a hard-coded administrator user names and passwords, and it is strongly recommended that you immediately change both on all the VMs. Click each VM in the Azure Portal then click on **Reset password** in the **Support + troubleshooting** blade. Select **Reset password** in the **Mode** drop down box, then select a new **User name** and **Password**. Click the **Update** button to persist the new user name and password.

# Extend Active Directory Domain Services (AD DS) to Azure

9/28/2018 • 8 minutes to read • [Edit Online](#)

This reference architecture shows how to extend your Active Directory environment to Azure to provide distributed authentication services using Active Directory Domain Services (AD DS). [Deploy this solution.](#)



Download a [Visio file](#) of this architecture.

AD DS is used to authenticate user, computer, application, or other identities that are included in a security domain. It can be hosted on-premises, but if your application is hosted partly on-premises and partly in Azure, it may be more efficient to replicate this functionality in Azure. This can reduce the latency caused by sending authentication and local authorization requests from the cloud back to AD DS running on-premises.

This architecture is commonly used when the on-premises network and the Azure virtual network are connected by a VPN or ExpressRoute connection. This architecture also supports bidirectional replication, meaning changes can be made either on-premises or in the cloud, and both sources will be kept consistent. Typical uses for this architecture include hybrid applications in which functionality is distributed between on-premises and Azure, and applications and services that perform authentication using Active Directory.

For additional considerations, see [Choose a solution for integrating on-premises Active Directory with Azure](#).

## Architecture

This architecture extends the architecture shown in [DMZ between Azure and the Internet](#). It has the following components.

- **On-premises network.** The on-premises network includes local Active Directory servers that can perform authentication and authorization for components located on-premises.
- **Active Directory servers.** These are domain controllers implementing directory services (AD DS) running as VMs in the cloud. These servers can provide authentication of components running in your Azure virtual network.
- **Active Directory subnet.** The AD DS servers are hosted in a separate subnet. Network security group (NSG) rules protect the AD DS servers and provide a firewall against traffic from unexpected sources.
- **Azure Gateway and Active Directory synchronization.** The Azure gateway provides a connection

between the on-premises network and the Azure VNet. This can be a [VPN connection](#) or [Azure ExpressRoute](#). All synchronization requests between the Active Directory servers in the cloud and on-premises pass through the gateway. User-defined routes (UDRs) handle routing for on-premises traffic that passes to Azure. Traffic to and from the Active Directory servers does not pass through the network virtual appliances (NVAs) used in this scenario.

For more information about configuring UDRs and the NVAs, see [Implementing a secure hybrid network architecture in Azure](#).

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### VM recommendations

Determine your [VM size](#) requirements based on the expected volume of authentication requests. Use the specifications of the machines hosting AD DS on premises as a starting point, and match them with the Azure VM sizes. Once deployed, monitor utilization and scale up or down based on the actual load on the VMs. For more information about sizing AD DS domain controllers, see [Capacity Planning for Active Directory Domain Services](#).

Create a separate virtual data disk for storing the database, logs, and SYSVOL for Active Directory. Do not store these items on the same disk as the operating system. Note that by default, data disks that are attached to a VM use write-through caching. However, this form of caching can conflict with the requirements of AD DS. For this reason, set the *Host Cache Preference* setting on the data disk to *None*. For more information, see [Placement of the Windows Server AD DS database and SYSVOL](#).

Deploy at least two VMs running AD DS as domain controllers and add them to an [availability set](#).

### Networking recommendations

Configure the VM network interface (NIC) for each AD DS server with a static private IP address for full domain name service (DNS) support. For more information, see [How to set a static private IP address in the Azure portal](#).

#### NOTE

Do not configure the VM NIC for any AD DS with a public IP address. See [Security considerations](#) for more details.

The Active Directory subnet NSG requires rules to permit incoming traffic from on-premises. For detailed information on the ports used by AD DS, see [Active Directory and Active Directory Domain Services Port Requirements](#). Also, ensure the UDR tables do not route AD DS traffic through the NVAs used in this architecture.

### Active Directory site

In AD DS, a site represents a physical location, network, or collection of devices. AD DS sites are used to manage AD DS database replication by grouping together AD DS objects that are located close to one another and are connected by a high speed network. AD DS includes logic to select the best strategy for replicating the AD DS database between sites.

We recommend that you create an AD DS site including the subnets defined for your application in Azure. Then, configure a site link between your on-premises AD DS sites, and AD DS will automatically perform the most efficient database replication possible. Note that this database replication requires little beyond the initial configuration.

### Active Directory operations masters

The operations masters role can be assigned to AD DS domain controllers to support consistency checking

between instances of replicated AD DS databases. There are five operations master roles: schema master, domain naming master, relative identifier master, primary domain controller master emulator, and infrastructure master. For more information about these roles, see [What are Operations Masters?](#).

We recommend you do not assign operations masters roles to the domain controllers deployed in Azure.

## Monitoring

Monitor the resources of the domain controller VMs as well as the AD DS Services and create a plan to quickly correct any problems. For more information, see [Monitoring Active Directory](#). You can also install tools such as [Microsoft Systems Center](#) on the monitoring server (see the architecture diagram) to help perform these tasks.

## Scalability considerations

AD DS is designed for scalability. You don't need to configure a load balancer or traffic controller to direct requests to AD DS domain controllers. The only scalability consideration is to configure the VMs running AD DS with the correct size for your network load requirements, monitor the load on the VMs, and scale up or down as necessary.

## Availability considerations

Deploy the VMs running AD DS into an [availability set](#). Also, consider assigning the role of [standby operations master](#) to at least one server, and possibly more depending on your requirements. A standby operations master is an active copy of the operations master that can be used in place of the primary operations masters server during fail over.

## Manageability considerations

Perform regular AD DS backups. Don't simply copy the VHD files of domain controllers instead of performing regular backups, because the AD DS database file on the VHD may not be in a consistent state when it's copied, making it impossible to restart the database.

Do not shut down a domain controller VM using Azure portal. Instead, shut down and restart from the guest operating system. Shutting down through the portal causes the VM to be deallocated, which resets both the `VM-GenerationID` and the `invocationID` of the Active Directory repository. This discards the AD DS relative identifier (RID) pool and marks SYSVOL as nonauthoritative, and may require reconfiguration of the domain controller.

## Security considerations

AD DS servers provide authentication services and are an attractive target for attacks. To secure them, prevent direct Internet connectivity by placing the AD DS servers in a separate subnet with an NSG acting as a firewall. Close all ports on the AD DS servers except those necessary for authentication, authorization, and server synchronization. For more information, see [Active Directory and Active Directory Domain Services Port Requirements](#).

Consider implementing an additional security perimeter around servers with a pair of subnets and NVAs, as described in [Implementing a secure hybrid network architecture with Internet access in Azure](#).

Use either BitLocker or Azure disk encryption to encrypt the disk hosting the AD DS database.

## Deploy the solution

A deployment for this architecture is available on [GitHub](#). Note that the entire deployment can take up to two hours, which includes creating the VPN gateway and running the scripts that configure AD DS.

## Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

## Deploy the simulated on-premises datacenter

1. Navigate to the `identity/adds-extend-domain` folder of the GitHub repository.
2. Open the `onprem.json` file. Search for instances of `adminPassword` and `Password` and add values for the passwords.
3. Run the following command and wait for the deployment to finish:

```
azbb -s <subscription_id> -g <resource group> -l <location> -p onprem.json --deploy
```

## Deploy the Azure VNet

1. Open the `azure.json` file. Search for instances of `adminPassword` and `Password` and add values for the passwords.
2. In the same file, search for instances of `sharedKey` and enter shared keys for the VPN connection.

```
"sharedKey": "",
```

3. Run the following command and wait for the deployment to finish.

```
azbb -s <subscription_id> -g <resource group> -l <location> -p onoprem.json --deploy
```

Deploy to the same resource group as the on-premises VNet.

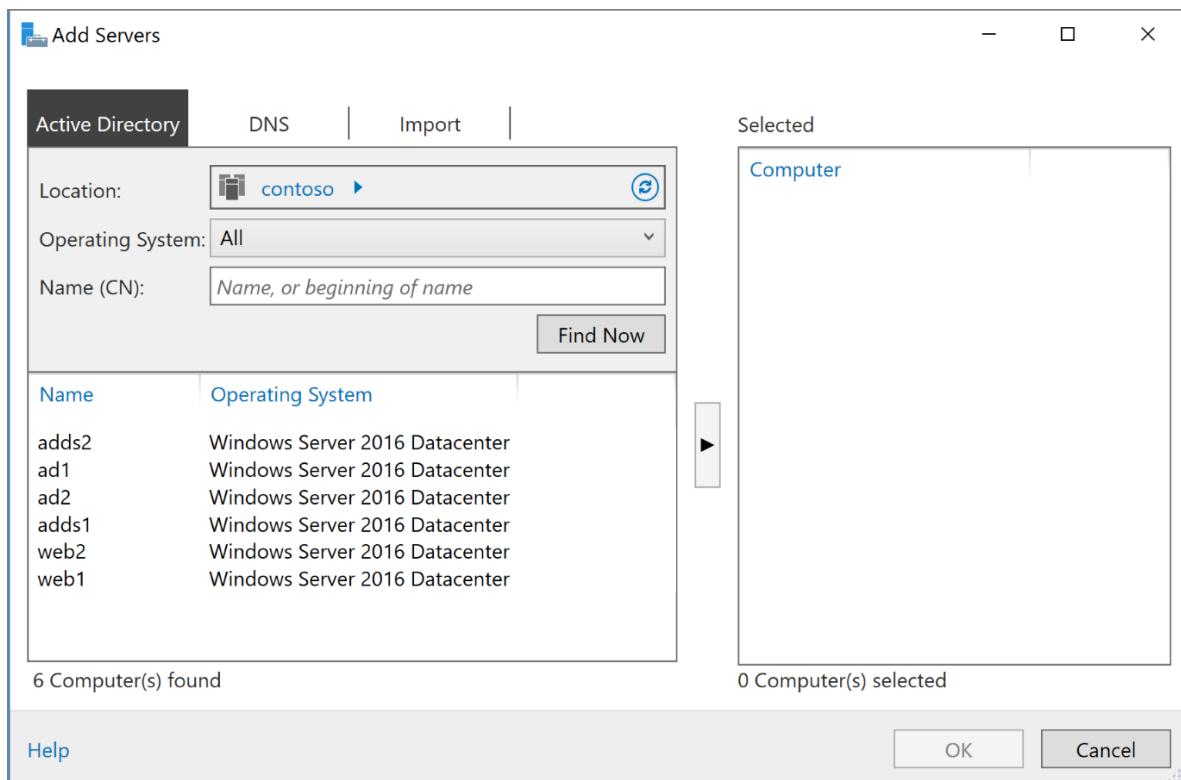
## Test connectivity with the Azure VNet

After deployment completes, you can test connectivity from the simulated on-premises environment to the Azure VNet.

1. Use the Azure portal, navigate to the resource group that you created.
2. Find the VM named `ra-onpremise-mgmt-vm1`.
3. Click `Connect` to open a remote desktop session to the VM. The username is `contoso\testuser`, and the password is the one that you specified in the `onprem.json` parameter file.
4. From inside your remote desktop session, open another remote desktop session to 10.0.4.4, which is the IP address of the VM named `adds-vm1`. The username is `contoso\testuser`, and the password is the one that you specified in the `azure.json` parameter file.
5. From inside the remote desktop session for `adds-vm1`, go to **Server Manager** and click **Add other**

## servers to manage.

6. In the **Active Directory** tab, click **Find now**. You should see a list of the AD, AD DS, and Web VMs.



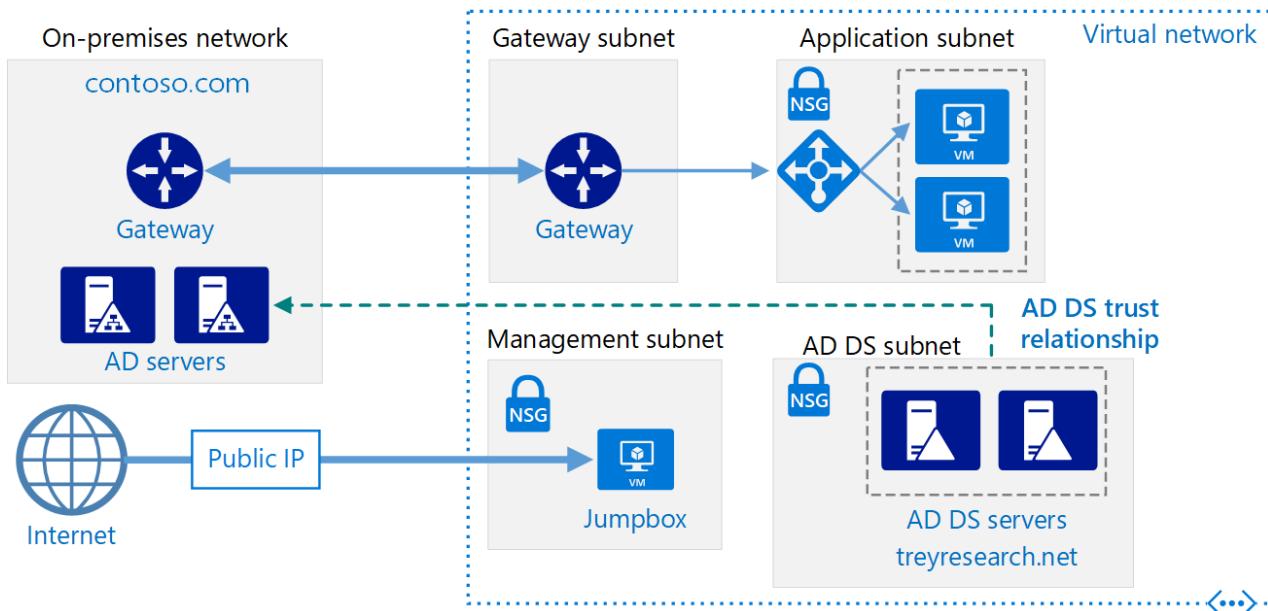
## Next steps

- Learn the best practices for [creating an AD DS resource forest](#) in Azure.
- Learn the best practices for [creating an Active Directory Federation Services \(AD FS\) infrastructure](#) in Azure.

# Create an Active Directory Domain Services (AD DS) resource forest in Azure

7/30/2018 • 6 minutes to read • [Edit Online](#)

This reference architecture shows how to create a separate Active Directory domain in Azure that is trusted by domains in your on-premises AD forest. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

Active Directory Domain Services (AD DS) stores identity information in a hierarchical structure. The top node in the hierarchical structure is known as a forest. A forest contains domains, and domains contain other types of objects. This reference architecture creates an AD DS forest in Azure with a one-way outgoing trust relationship with an on-premises domain. The forest in Azure contains a domain that does not exist on-premises. Because of the trust relationship, logons made against on-premises domains can be trusted for access to resources in the separate Azure domain.

Typical uses for this architecture include maintaining security separation for objects and identities held in the cloud, and migrating individual domains from on-premises to the cloud.

For additional considerations, see [Choose a solution for integrating on-premises Active Directory with Azure](#).

## Architecture

The architecture has the following components.

- **On-premises network.** The on-premises network contains its own Active Directory forest and domains.
- **Active Directory servers.** These are domain controllers implementing domain services running as VMs in the cloud. These servers host a forest containing one or more domains, separate from those located on-premises.
- **One-way trust relationship.** The example in the diagram shows a one-way trust from the domain in Azure to the on-premises domain. This relationship enables on-premises users to access resources in the domain in Azure, but not the other way around. It is possible to create a two-way trust if cloud users also require access to on-premises resources.
- **Active Directory subnet.** The AD DS servers are hosted in a separate subnet. Network security group (NSG) rules protect the AD DS servers and provide a firewall against traffic from unexpected sources.

- **Azure gateway.** The Azure gateway provides a connection between the on-premises network and the Azure VNet. This can be a [VPN connection](#) or [Azure ExpressRoute](#). For more information, see [Implementing a secure hybrid network architecture in Azure](#).

## Recommendations

For specific recommendations on implementing Active Directory in Azure, see the following articles:

- [Extending Active Directory Domain Services \(AD DS\) to Azure](#).
- [Guidelines for Deploying Windows Server Active Directory on Azure Virtual Machines](#).

### Trust

The on-premises domains are contained within a different forest from the domains in the cloud. To enable authentication of on-premises users in the cloud, the domains in Azure must trust the logon domain in the on-premises forest. Similarly, if the cloud provides a logon domain for external users, it may be necessary for the on-premises forest to trust the cloud domain.

You can establish trusts at the forest level by [creating forest trusts](#), or at the domain level by [creating external trusts](#). A forest level trust creates a relationship between all domains in two forests. An external domain level trust only creates a relationship between two specified domains. You should only create external domain level trusts between domains in different forests.

Trusts can be unidirectional (one-way) or bidirectional (two-way):

- A one-way trust enables users in one domain or forest (known as the *incoming* domain or forest) to access the resources held in another (the *outgoing* domain or forest).
- A two-way trust enables users in either domain or forest to access resources held in the other.

The following table summarizes trust configurations for some simple scenarios:

| SCENARIO                                                                                              | ON-PREMISES TRUST              | CLOUD TRUST                    |
|-------------------------------------------------------------------------------------------------------|--------------------------------|--------------------------------|
| On-premises users require access to resources in the cloud, but not vice versa                        | One-way, incoming              | One-way, outgoing              |
| Users in the cloud require access to resources located on-premises, but not vice versa                | One-way, outgoing              | One-way, incoming              |
| Users in the cloud and on-premises both require access to resources held in the cloud and on-premises | Two-way, incoming and outgoing | Two-way, incoming and outgoing |

## Scalability considerations

Active Directory is automatically scalable for domain controllers that are part of the same domain. Requests are distributed across all controllers within a domain. You can add another domain controller, and it synchronizes automatically with the domain. Do not configure a separate load balancer to direct traffic to controllers within the domain. Ensure that all domain controllers have sufficient memory and storage resources to handle the domain database. Make all domain controller VMs the same size.

## Availability considerations

Provision at least two domain controllers for each domain. This enables automatic replication between servers. Create an availability set for the VMs acting as Active Directory servers handling each domain. Put at least two

servers in this availability set.

Also, consider designating one or more servers in each domain as [standby operations masters](#) in case connectivity to a server acting as a flexible single master operation (FSMO) role fails.

## Manageability considerations

For information about management and monitoring considerations, see [Extending Active Directory to Azure](#).

For additional information, see [Monitoring Active Directory](#). You can install tools such as [Microsoft Systems Center](#) on a monitoring server in the management subnet to help perform these tasks.

## Security considerations

Forest level trusts are transitive. If you establish a forest level trust between an on-premises forest and a forest in the cloud, this trust is extended to other new domains created in either forest. If you use domains to provide separation for security purposes, consider creating trusts at the domain level only. Domain level trusts are non-transitive.

For Active Directory-specific security considerations, see the security considerations section in [Extending Active Directory to Azure](#).

## Deploy the solution

A deployment for this architecture is available on [GitHub](#). Note that the entire deployment can take up to two hours, which includes creating the VPN gateway and running the scripts that configure AD DS.

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Deploy the simulated on-premises datacenter

1. Navigate to the `identity/adds-forest` folder of the GitHub repository.
2. Open the `onprem.json` file. Search for instances of `adminPassword` and `Password` and add values for the passwords.
3. Run the following command and wait for the deployment to finish:

```
azbb -s <subscription_id> -g <resource group> -l <location> -p onprem.json --deploy
```

### Deploy the Azure VNet

1. Open the `azure.json` file. Search for instances of `adminPassword` and `Password` and add values for the passwords.

2. In the same file, search for instances of `sharedKey` and enter shared keys for the VPN connection.

```
"sharedKey": "",
```

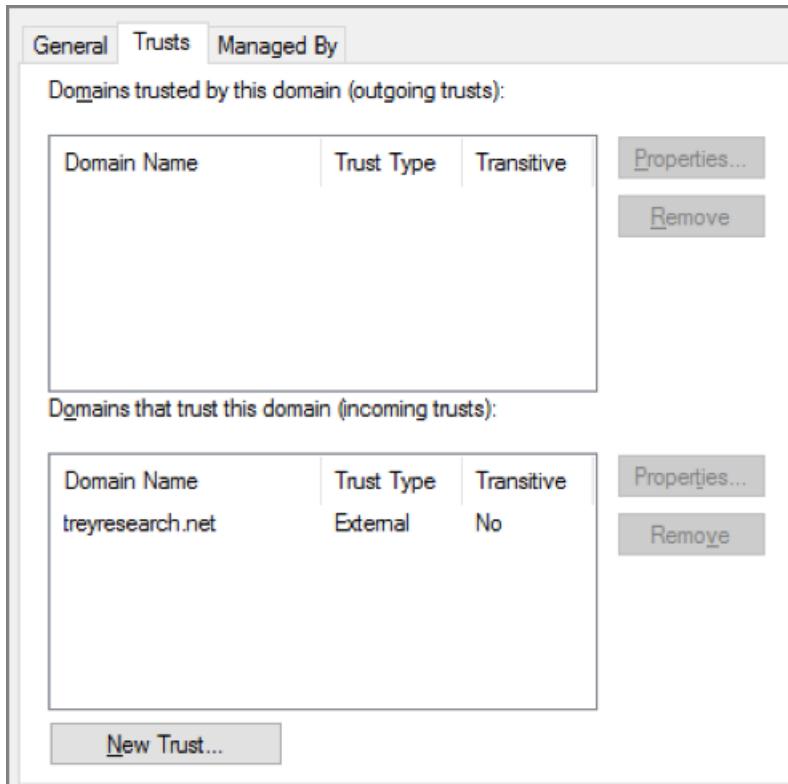
3. Run the following command and wait for the deployment to finish.

```
azbb -s <subscription_id> -g <resource group> -l <location> -p onoprem.json --deploy
```

Deploy to the same resource group as the on-premises VNet.

### Test the AD trust relation

1. Use the Azure portal, navigate to the resource group that you created.
2. Use the Azure portal to find the VM named `ra-adt-mgmt-vm1`.
3. Click `Connect` to open a remote desktop session to the VM. The username is `contoso\testuser`, and the password is the one that you specified in the `onprem.json` parameter file.
4. From inside your remote desktop session, open another remote desktop session to 192.168.0.4, which is the IP address of the VM named `ra-adtrust-onpremise-ad-vm1`. The username is `contoso\testuser`, and the password is the one that you specified in the `azure.json` parameter file.
5. From inside the remote desktop session for `ra-adtrust-onpremise-ad-vm1`, go to **Server Manager** and click **Tools > Active Directory Domains and Trusts**.
6. In the left pane, right-click on the contoso.com and select **Properties**.
7. Click the **Trusts** tab. You should see treyresearch.net listed as an incoming trust.



## Next steps

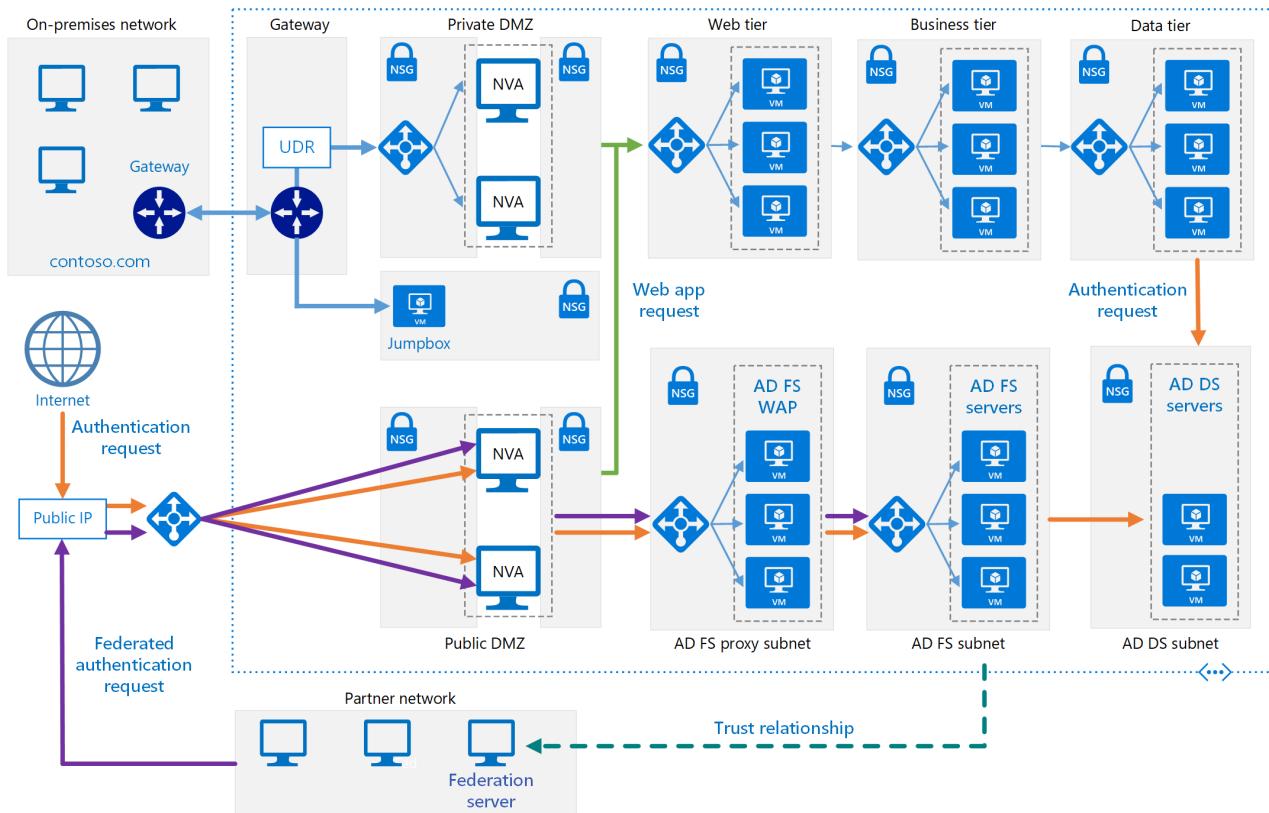
- Learn the best practices for [extending your on-premises AD DS domain to Azure](#)
- Learn the best practices for [creating an AD FS infrastructure](#) in Azure.



# Extend Active Directory Federation Services (AD FS) to Azure

9/28/2018 • 15 minutes to read • [Edit Online](#)

This reference architecture implements a secure hybrid network that extends your on-premises network to Azure and uses [Active Directory Federation Services \(AD FS\)](#) to perform federated authentication and authorization for components running in Azure. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

AD FS can be hosted on-premises, but if your application is a hybrid in which some parts are implemented in Azure, it may be more efficient to replicate AD FS in the cloud.

The diagram shows the following scenarios:

- Application code from a partner organization accesses a web application hosted inside your Azure VNet.
- An external, registered user with credentials stored inside Active Directory Domain Services (DS) accesses a web application hosted inside your Azure VNet.
- A user connected to your VNet using an authorized device executes a web application hosted inside your Azure VNet.

Typical uses for this architecture include:

- Hybrid applications where workloads run partly on-premises and partly in Azure.
- Solutions that use federated authorization to expose web applications to partner organizations.
- Systems that support access from web browsers running outside of the organizational firewall.
- Systems that enable users to access to web applications by connecting from authorized external devices such as remote computers, notebooks, and other mobile devices.

This reference architecture focuses on *passive federation*, in which the federation servers decide how and when to authenticate a user. The user provides sign in information when the application is started. This mechanism is most commonly used by web browsers and involves a protocol that redirects the browser to a site where the user authenticates. AD FS also supports *active federation*, where an application takes on responsibility for supplying credentials without further user interaction, but that scenario is outside the scope of this architecture.

For additional considerations, see [Choose a solution for integrating on-premises Active Directory with Azure](#).

## Architecture

This architecture extends the implementation described in [Extending AD DS to Azure](#). It contains the following components.

- **AD DS subnet.** The AD DS servers are contained in their own subnet with network security group (NSG) rules acting as a firewall.
- **AD DS servers.** Domain controllers running as VMs in Azure. These servers provide authentication of local identities within the domain.
- **AD FS subnet.** The AD FS servers are located within their own subnet with NSG rules acting as a firewall.
- **AD FS servers.** The AD FS servers provide federated authorization and authentication. In this architecture, they perform the following tasks:
  - Receiving security tokens containing claims made by a partner federation server on behalf of a partner user. AD FS verifies that the tokens are valid before passing the claims to the web application running in Azure to authorize requests.

The web application running in Azure is the *relying party*. The partner federation server must issue claims that are understood by the web application. The partner federation servers are referred to as *account partners*, because they submit access requests on behalf of authenticated accounts in the partner organization. The AD FS servers are called *resource partners* because they provide access to resources (the web application).

- Authenticating and authorizing incoming requests from external users running a web browser or device that needs access to web applications, by using AD DS and the [Active Directory Device Registration Service](#).

The AD FS servers are configured as a farm accessed through an Azure load balancer. This implementation improves availability and scalability. The AD FS servers are not exposed directly to the Internet. All Internet traffic is filtered through AD FS web application proxy servers and a DMZ (also referred to as a perimeter network).

For more information about how AD FS works, see [Active Directory Federation Services Overview](#). Also, the article [AD FS deployment in Azure](#) contains a detailed step-by-step introduction to implementation.

- **AD FS proxy subnet.** The AD FS proxy servers can be contained within their own subnet, with NSG rules providing protection. The servers in this subnet are exposed to the Internet through a set of network virtual appliances that provide a firewall between your Azure virtual network and the Internet.
- **AD FS web application proxy (WAP) servers.** These VMs act as AD FS servers for incoming requests from partner organizations and external devices. The WAP servers act as a filter, shielding the AD FS servers from direct access from the Internet. As with the AD FS servers, deploying the WAP servers in a farm with load balancing gives you greater availability and scalability than deploying a collection of stand-alone servers.

**NOTE**

For detailed information about installing WAP servers, see [Install and Configure the Web Application Proxy Server](#)

- **Partner organization.** A partner organization running a web application that requests access to a web application running in Azure. The federation server at the partner organization authenticates requests locally, and submits security tokens containing claims to AD FS running in Azure. AD FS in Azure validates the security tokens, and if valid can pass the claims to the web application running in Azure to authorize them.

**NOTE**

You can also configure a VPN tunnel using Azure gateway to provide direct access to AD FS for trusted partners. Requests received from these partners do not pass through the WAP servers.

For more information about the parts of the architecture that are not related to AD FS, see the following:

- [Implementing a secure hybrid network architecture in Azure](#)
- [Implementing a secure hybrid network architecture with Internet access in Azure](#)
- [Implementing a secure hybrid network architecture with Active Directory identities in Azure](#).

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### VM recommendations

Create VMs with sufficient resources to handle the expected volume of traffic. Use the size of the existing machines hosting AD FS on premises as a starting point. Monitor the resource utilization. You can resize the VMs and scale down if they are too large.

Follow the recommendations listed in [Running a Windows VM on Azure](#).

### Networking recommendations

Configure the network interface for each of the VMs hosting AD FS and WAP servers with static private IP addresses.

Do not give the AD FS VMs public IP addresses. For more information, see the Security considerations section.

Set the IP address of the preferred and secondary domain name service (DNS) servers for the network interfaces for each AD FS and WAP VM to reference the Active Directory DS VMs. The Active Directory DS VMs should be running DNS. This step is necessary to enable each VM to join the domain.

### AD FS availability

Create an AD FS farm with at least two servers to increase availability of the service. Use different storage accounts for each AD FS VM in the farm. This approach helps to ensure that a failure in a single storage account does not make the entire farm inaccessible.

**IMPORTANT**

We recommend the use of [managed disks](#). Managed disks do not require a storage account. You simply specify the size and type of disk and it is deployed in a highly available way. Our [reference architectures](#) do not currently deploy managed disks but the [template building blocks](#) will be updated to deploy managed disks in version 2.

Create separate Azure availability sets for the AD FS and WAP VMs. Ensure that there are at least two VMs in each set. Each availability set must have at least two update domains and two fault domains.

Configure the load balancers for the AD FS VMs and WAP VMs as follows:

- Use an Azure load balancer to provide external access to the WAP VMs, and an internal load balancer to distribute the load across the AD FS servers in the farm.
- Only pass traffic appearing on port 443 (HTTPS) to the AD FS/WAP servers.
- Give the load balancer a static IP address.
- Create a health probe using HTTP against `/adfs/probe`. For more information, see [Hardware Load Balancer Health Checks and Web Application Proxy / AD FS 2012 R2](#).

#### NOTE

AD FS servers use the Server Name Indication (SNI) protocol, so attempting to probe using an HTTPS endpoint from the load balancer fails.

- Add a DNS A record to the domain for the AD FS load balancer. Specify the IP address of the load balancer, and give it a name in the domain (such as adfs.contoso.com). This is the name clients and the WAP servers use to access the AD FS server farm.

### AD FS security

Prevent direct exposure of the AD FS servers to the Internet. AD FS servers are domain-joined computers that have full authorization to grant security tokens. If a server is compromised, a malicious user can issue full access tokens to all web applications and to all federation servers that are protected by AD FS. If your system must handle requests from external users not connecting from trusted partner sites, use WAP servers to handle these requests. For more information, see [Where to Place a Federation Server Proxy](#).

Place AD FS servers and WAP servers in separate subnets with their own firewalls. You can use NSG rules to define firewall rules. If you require more comprehensive protection you can implement an additional security perimeter around servers by using a pair of subnets and network virtual appliances (NVAs), as described in the document [Implementing a secure hybrid network architecture with Internet access in Azure](#). All firewalls should allow traffic on port 443 (HTTPS).

Restrict direct sign in access to the AD FS and WAP servers. Only DevOps staff should be able to connect.

Do not join the WAP servers to the domain.

### AD FS installation

The article [Deploying a Federation Server Farm](#) provides detailed instructions for installing and configuring AD FS. Perform the following tasks before configuring the first AD FS server in the farm:

1. Obtain a publicly trusted certificate for performing server authentication. The *subject name* must contain the name clients use to access the federation service. This can be the DNS name registered for the load balancer, for example, `adfs.contoso.com` (avoid using wildcard names such as `*.contoso.com`, for security reasons). Use the same certificate on all AD FS server VMs. You can purchase a certificate from a trusted certification authority, but if your organization uses Active Directory Certificate Services you can create your own.

The *subject alternative name* is used by the device registration service (DRS) to enable access from external devices. This should be of the form `enterpriseregistration.contoso.com`.

For more information, see [Obtain and Configure a Secure Sockets Layer \(SSL\) Certificate for AD FS](#).

2. On the domain controller, generate a new root key for the Key Distribution Service. Set the effective time to

the current time minus 10 hours (this configuration reduces the delay that can occur in distributing and synchronizing keys across the domain). This step is necessary to support creating the group service account that is used to run the AD FS service. The following PowerShell command shows an example of how to do this:

```
Add-KdsRootKey -EffectiveTime (Get-Date).AddHours(-10)
```

3. Add each AD FS server VM to the domain.

#### **NOTE**

To install AD FS, the domain controller running the primary domain controller (PDC) emulator flexible single master operation (FSMO) role for the domain must be running and accessible from the AD FS VMs. <<RBC: Is there a way to make this less repetitive?>>

## **AD FS trust**

Establish federation trust between your AD FS installation, and the federation servers of any partner organizations. Configure any claims filtering and mapping required.

- DevOps staff at each partner organization must add a relying party trust for the web applications accessible through your AD FS servers.
- DevOps staff in your organization must configure claims-provider trust to enable your AD FS servers to trust the claims that partner organizations provide.
- DevOps staff in your organization must also configure AD FS to pass claims on to your organization's web applications.

For more information, see [Establishing Federation Trust](#).

Publish your organization's web applications and make them available to external partners by using preauthentication through the WAP servers. For more information, see [Publish Applications using AD FS Preauthentication](#)

AD FS supports token transformation and augmentation. Azure Active Directory does not provide this feature. With AD FS, when you set up the trust relationships, you can:

- Configure claim transformations for authorization rules. For example, you can map group security from a representation used by a non-Microsoft partner organization to something that Active Directory DS can authorize in your organization.
- Transform claims from one format to another. For example, you can map from SAML 2.0 to SAML 1.1 if your application only supports SAML 1.1 claims.

## **AD FS monitoring**

The [Microsoft System Center Management Pack for Active Directory Federation Services 2012 R2](#) provides both proactive and reactive monitoring of your AD FS deployment for the federation server. This management pack monitors:

- Events that the AD FS service records in its event logs.
- The performance data that the AD FS performance counters collect.
- The overall health of the AD FS system and web applications (relying parties), and provides alerts for critical issues and warnings.

## **Scalability considerations**

The following considerations, summarized from the article [Plan your AD FS deployment](#), give a starting point for

sizing AD FS farms:

- If you have fewer than 1000 users, do not create dedicated servers, but instead install AD FS on each of the Active Directory DS servers in the cloud. Make sure that you have at least two Active Directory DS servers to maintain availability. Create a single WAP server.
- If you have between 1000 and 15000 users, create two dedicated AD FS servers and two dedicated WAP servers.
- If you have between 15000 and 60000 users, create between three and five dedicated AD FS servers and at least two dedicated WAP servers.

These considerations assume that you are using dual quad-core VM (Standard D4\_v2, or better) sizes in Azure.

If you are using the Windows Internal Database to store AD FS configuration data, you are limited to eight AD FS servers in the farm. If you anticipate that you will need more in the future, use SQL Server. For more information, see [The Role of the AD FS Configuration Database](#).

## Availability considerations

You can use either SQL Server or the Windows Internal Database to hold AD FS configuration information. The Windows Internal Database provides basic redundancy. Changes are written directly to only one of the AD FS databases in the AD FS cluster, while the other servers use pull replication to keep their databases up to date. Using SQL Server can provide full database redundancy and high availability using failover clustering or mirroring.

## Manageability considerations

DevOps staff should be prepared to perform the following tasks:

- Managing the federation servers, including managing the AD FS farm, managing trust policy on the federation servers, and managing the certificates used by the federation services.
- Managing the WAP servers including managing the WAP farm and certificates.
- Managing web applications including configuring relying parties, authentication methods, and claims mappings.
- Backing up AD FS components.

## Security considerations

AD FS utilizes the HTTPS protocol, so make sure that the NSG rules for the subnet containing the web tier VMs permit HTTPS requests. These requests can originate from the on-premises network, the subnets containing the web tier, business tier, data tier, private DMZ, public DMZ, and the subnet containing the AD FS servers.

Consider using a set of network virtual appliances that logs detailed information on traffic traversing the edge of your virtual network for auditing purposes.

## Deploy the solution

A solution is available on [GitHub](#) to deploy this reference architecture. You will need the latest version of the [Azure CLI](#) to run the Powershell script that deploys the solution. To deploy the reference architecture, follow these steps:

1. Download or clone the solution folder from [GitHub](#) to your local machine.
2. Open the Azure CLI and navigate to the local solution folder.
3. Run the following command:

```
.\Deploy-ReferenceArchitecture.ps1 <subscription id> <location> <mode>
```

Replace `<subscription id>` with your Azure subscription ID.

For `<location>`, specify an Azure region, such as `eastus` or `westus`.

The `<mode>` parameter controls the granularity of the deployment, and can be one of the following values:

- `Onpremise`: Deploys a simulated on-premises environment. You can use this deployment to test and experiment if you do not have an existing on-premises network, or if you want to test this reference architecture without changing the configuration of your existing on-premises network.
- `Infrastructure`: deploys the VNet infrastructure and jump box.
- `CreateVpn`: deploys an Azure virtual network gateway and connects it to the simulated on-premises network.
- `AzureADDS`: deploys the VMs acting as Active Directory DS servers, deploys Active Directory to these VMs, and creates the domain in Azure.
- `AdfsVm`: deploys the AD FS VMs and joins them to the domain in Azure.
- `PublicDMZ`: deploys the public DMZ in Azure.
- `ProxyVm`: deploys the AD FS proxy VMs and joins them to the domain in Azure.
- `Prepare`: deploys all of the preceding deployments. **This is the recommended option if you are building an entirely new deployment and you don't have an existing on-premises infrastructure.**
- `Workload`: optionally deploys web, business, and data tier VMs and supporting network. Not included in the `Prepare` deployment mode.
- `PrivateDMZ`: optionally deploys the private DMZ in Azure in front of the `Workload` VMs deployed above. Not included in the `Prepare` deployment mode.

4. Wait for the deployment to complete. If you used the `Prepare` option, the deployment takes several hours to complete, and finishes with the message

```
Preparation is completed. Please install certificate to all AD FS and proxy VMs.
```

5. Restart the jump box (`ra-adfs-mgmt-vm1` in the `ra-adfs-security-rg` group) to allow its DNS settings to take effect.

6. [Obtain an SSL Certificate for AD FS](#) and install this certificate on the AD FS VMs. Note that you can connect to them through the jump box. The IP addresses are `10.0.5.4` and `10.0.5.5`. The default username is `contoso\testuser` with password `AweSome@PW`.

#### NOTE

The comments in the `Deploy-ReferenceArchitecture.ps1` script at this point provides detailed instructions for creating a self-signed test certificate and authority using the `makecert` command. However, perform these steps as a **test** only and do not use the certificates generated by `makecert` in a production environment.

7. Run the following PowerShell command to deploy the AD FS server farm:

```
.\Deploy-ReferenceArchitecture.ps1 <subscription id> <location> Adfs
```

8. On the jump box, browse to <https://adfs.contoso.com/adfs/ls/idpinitiatedsignon.htm> to test the AD FS installation (you may receive a certificate warning that you can ignore for this test). Verify that the Contoso Corporation sign-in page appears. Sign in as `contoso\testuser` with password `AweSome@PW`.

9. Install the SSL certificate on the AD FS proxy VMs. The IP addresses are *10.0.6.4* and *10.0.6.5*.

10. Run the following PowerShell command to deploy the first AD FS proxy server:

```
.\Deploy-ReferenceArchitecture.ps1 <subscription id> <location> Proxy1
```

11. Follow the instructions displayed by the script to test the installation of the first proxy server.

12. Run the following PowerShell command to deploy the second proxy server:

```
.\Deploy-ReferenceArchitecture.ps1 <subscription id> <location> Proxy2
```

13. Follow the instructions displayed by the script to test the complete proxy configuration.

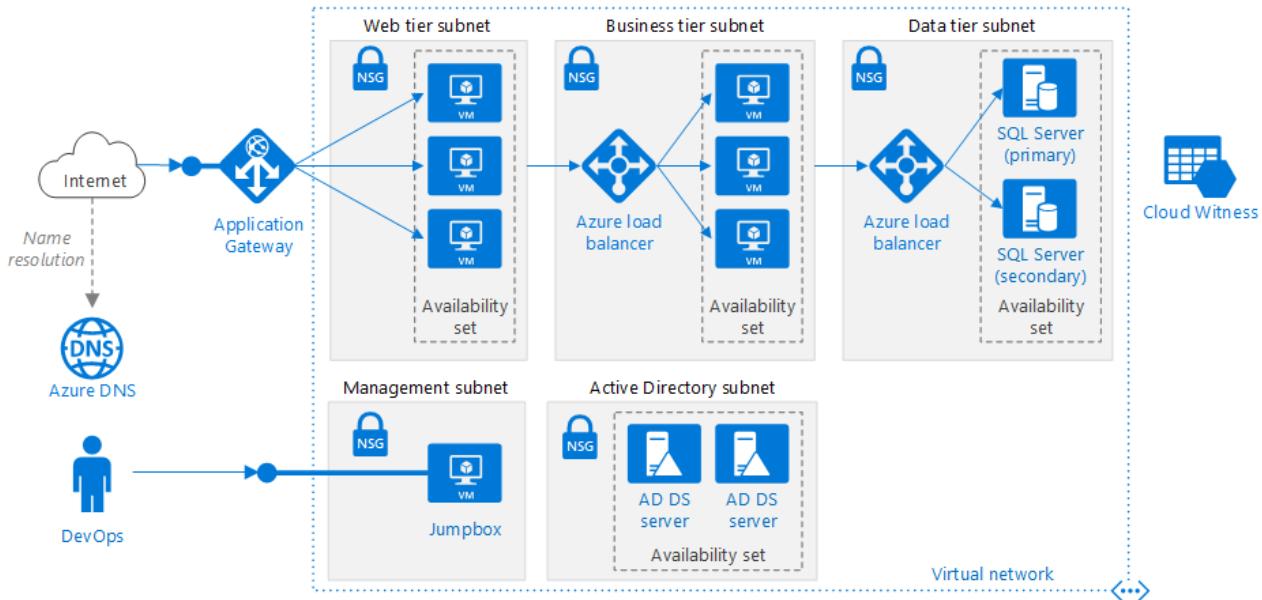
## Next steps

- Learn about [Azure Active Directory](#).
- Learn about [Azure Active Directory B2C](#).

# N-tier application with SQL Server

9/28/2018 • 12 minutes to read • [Edit Online](#)

This reference architecture shows how to deploy VMs and a virtual network configured for an N-tier application, using SQL Server on Windows for the data tier. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

## Architecture

The architecture has the following components:

- **Resource group.** [Resource groups](#) are used to group resources so they can be managed by lifetime, owner, or other criteria.
- **Virtual network (VNet) and subnets.** Every Azure VM is deployed into a VNet that can be segmented into multiple subnets. Create a separate subnet for each tier.
- **Application gateway.** [Azure Application Gateway](#) is a layer 7 load balancer. In this architecture, it routes HTTP requests to the web front end. Application Gateway also provides a [web application firewall](#) (WAF) that protects the application from common exploits and vulnerabilities.
- **NSGs.** Use [network security groups](#) (NSGs) to restrict network traffic within the VNet. For example, in the 3-tier architecture shown here, the database tier does not accept traffic from the web front end, only from the business tier and the management subnet.
- **Virtual machines.** For recommendations on configuring VMs, see [Run a Windows VM on Azure](#) and [Run a Linux VM on Azure](#).
- **Availability sets.** Create an [availability set](#) for each tier, and provision at least two VMs in each tier. This makes the VMs eligible for a higher [service level agreement \(SLA\)](#) for VMs.
- **VM scale set** (not shown). A [VM scale set](#) is an alternative to using an availability set. A scale set makes it easy to scale out the VMs in a tier, either manually or automatically based on predefined rules.
- **Load balancers.** Use [Azure Load Balancer](#) to distribute network traffic from the web tier to the business tier, and from the business tier to SQL Server.

- **Public IP address.** A public IP address is needed for the application to receive Internet traffic.
- **Jumpbox.** Also called a [bastion host](#). A secure VM on the network that administrators use to connect to the other VMs. The jumpbox has an NSG that allows remote traffic only from public IP addresses on a safe list. The NSG should permit remote desktop (RDP) traffic.
- **SQL Server Always On Availability Group.** Provides high availability at the data tier, by enabling replication and failover. It uses Windows Server Failover Cluster (WSFC) technology for failover.
- **Active Directory Domain Services (AD DS) Servers.** The computer objects for the failover cluster and its associated clustered roles are created in Active Directory Domain Services (AD DS).
- **Cloud Witness.** A failover cluster requires more than half of its nodes to be running, which is known as having quorum. If the cluster has just two nodes, a network partition could cause each node to think it's the master node. In that case, you need a *witness* to break ties and establish quorum. A witness is a resource such as a shared disk that can act as a tie breaker to establish quorum. Cloud Witness is a type of witness that uses Azure Blob Storage. To learn more about the concept of quorum, see [Understanding cluster and pool quorum](#). For more information about Cloud Witness, see [Deploy a Cloud Witness for a Failover Cluster](#).
- **Azure DNS.** [Azure DNS](#) is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.

## Recommendations

Your requirements might differ from the architecture described here. Use these recommendations as a starting point.

### VNet / Subnets

When you create the VNet, determine how many IP addresses your resources in each subnet require. Specify a subnet mask and a VNet address range large enough for the required IP addresses, using [CIDR](#) notation. Use an address space that falls within the standard [private IP address blocks](#), which are 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16.

Choose an address range that does not overlap with your on-premises network, in case you need to set up a gateway between the VNet and your on-premise network later. Once you create the VNet, you can't change the address range.

Design subnets with functionality and security requirements in mind. All VMs within the same tier or role should go into the same subnet, which can be a security boundary. For more information about designing VNets and subnets, see [Plan and design Azure Virtual Networks](#).

### Load balancers

Do not expose the VMs directly to the Internet, but instead give each VM a private IP address. Clients connect using the public IP address associated with the Application Gateway.

Define load balancer rules to direct network traffic to the VMs. For example, to enable HTTP traffic, create a rule that maps port 80 from the front-end configuration to port 80 on the back-end address pool. When a client sends an HTTP request to port 80, the load balancer selects a back-end IP address by using a [hashing algorithm](#) that includes the source IP address. In that way, client requests are distributed across all the VMs.

### Network security groups

Use NSG rules to restrict traffic between tiers. For example, in the 3-tier architecture shown above, the web tier does not communicate directly with the database tier. To enforce this, the database tier should block incoming traffic from the web tier subnet.

1. Deny all inbound traffic from the VNet. (Use the `VIRTUAL_NETWORK` tag in the rule.)
2. Allow inbound traffic from the business tier subnet.
3. Allow inbound traffic from the database tier subnet itself. This rule allows communication between the database VMs, which is needed for database replication and failover.
4. Allow RDP traffic (port 3389) from the jumpbox subnet. This rule lets administrators connect to the database tier from the jumpbox.

Create rules 2 – 4 with higher priority than the first rule, so they override it.

## SQL Server Always On Availability Groups

We recommend [Always On Availability Groups](#) for SQL Server high availability. Prior to Windows Server 2016, Always On Availability Groups require a domain controller, and all nodes in the availability group must be in the same AD domain.

Other tiers connect to the database through an [availability group listener](#). The listener enables a SQL client to connect without knowing the name of the physical instance of SQL Server. VMs that access the database must be joined to the domain. The client (in this case, another tier) uses DNS to resolve the listener's virtual network name into IP addresses.

Configure the SQL Server Always On Availability Group as follows:

1. Create a Windows Server Failover Clustering (WSFC) cluster, a SQL Server Always On Availability Group, and a primary replica. For more information, see [Getting Started with Always On Availability Groups](#).
2. Create an internal load balancer with a static private IP address.
3. Create an availability group listener, and map the listener's DNS name to the IP address of an internal load balancer.
4. Create a load balancer rule for the SQL Server listening port (TCP port 1433 by default). The load balancer rule must enable *floating IP*, also called Direct Server Return. This causes the VM to reply directly to the client, which enables a direct connection to the primary replica.

### NOTE

When floating IP is enabled, the front-end port number must be the same as the back-end port number in the load balancer rule.

When a SQL client tries to connect, the load balancer routes the connection request to the primary replica. If there is a failover to another replica, the load balancer automatically routes subsequent requests to a new primary replica. For more information, see [Configure an ILB listener for SQL Server Always On Availability Groups](#).

During a failover, existing client connections are closed. After the failover completes, new connections will be routed to the new primary replica.

If your application makes significantly more reads than writes, you can offload some of the read-only queries to a secondary replica. See [Using a Listener to Connect to a Read-Only Secondary Replica \(Read-Only Routing\)](#).

Test your deployment by [forcing a manual failover](#) of the availability group.

## Jumpbox

Do not allow RDP access from the public Internet to the VMs that run the application workload. Instead, all RDP access to these VMs must come through the jumpbox. An administrator logs into the jumpbox, and then logs into the other VM from the jumpbox. The jumpbox allows RDP traffic from the Internet, but only from known, safe IP addresses.

The jumpbox has minimal performance requirements, so select a small VM size. Create a [public IP address](#) for the

jumpbox. Place the jumpbox in the same VNet as the other VMs, but in a separate management subnet.

To secure the jumpbox, add an NSG rule that allows RDP connections only from a safe set of public IP addresses. Configure the NSGs for the other subnets to allow RDP traffic from the management subnet.

## Scalability considerations

[VM scale sets](#) help you to deploy and manage a set of identical VMs. Scale sets support autoscaling based on performance metrics. As the load on the VMs increases, additional VMs are automatically added to the load balancer. Consider scale sets if you need to quickly scale out VMs, or need to autoscale.

There are two basic ways to configure VMs deployed in a scale set:

- Use extensions to configure the VM after it is provisioned. With this approach, new VM instances may take longer to start up than a VM with no extensions.
- Deploy a [managed disk](#) with a custom disk image. This option may be quicker to deploy. However, it requires you to keep the image up to date.

For additional considerations, see [Design considerations for scale sets](#).

### TIP

When using any autoscale solution, test it with production-level workloads well in advance.

Each Azure subscription has default limits in place, including a maximum number of VMs per region. You can increase the limit by filing a support request. For more information, see [Azure subscription and service limits, quotas, and constraints](#).

## Availability considerations

If you are not using VM scale sets, put VMs in the same tier into an availability set. Create at least two VMs in the availability set to support the [availability SLA for Azure VMs](#). For more information, see [Manage the availability of virtual machines](#).

The load balancer uses [health probes](#) to monitor the availability of VM instances. If a probe cannot reach an instance within a timeout period, the load balancer stops sending traffic to that VM. However, the load balancer will continue to probe, and if the VM becomes available again, the load balancer resumes sending traffic to that VM.

Here are some recommendations on load balancer health probes:

- Probes can test either HTTP or TCP. If your VMs run an HTTP server, create an HTTP probe. Otherwise create a TCP probe.
- For an HTTP probe, specify the path to an HTTP endpoint. The probe checks for an HTTP 200 response from this path. This can be the root path (""/"), or a health-monitoring endpoint that implements some custom logic to check the health of the application. The endpoint must allow anonymous HTTP requests.
- The probe is sent from a [known IP address](#), 168.63.129.16. Make sure you don't block traffic to or from this IP address in any firewall policies or network security group (NSG) rules.
- Use [health probe logs](#) to view the status of the health probes. Enable logging in the Azure portal for each load balancer. Logs are written to Azure Blob storage. The logs show how many VMs on the back end are not receiving network traffic due to failed probe responses.

If you need higher availability than the [Azure SLA for VMs](#) provides, consider replicating the application across two regions, using Azure Traffic Manager for failover. For more information, see [Multi-region N-tier application for high availability](#).

# Security considerations

Virtual networks are a traffic isolation boundary in Azure. VMs in one VNet cannot communicate directly with VMs in a different VNet. VMs within the same VNet can communicate, unless you create [network security groups](#) (NSGs) to restrict traffic. For more information, see [Microsoft cloud services and network security](#).

Consider adding a network virtual appliance (NVA) to create a DMZ between the Internet and the Azure virtual network. NVA is a generic term for a virtual appliance that can perform network-related tasks, such as firewall, packet inspection, auditing, and custom routing. For more information, see [Implementing a DMZ between Azure and the Internet](#).

Encrypt sensitive data at rest and use [Azure Key Vault](#) to manage the database encryption keys. Key Vault can store encryption keys in hardware security modules (HSMs). For more information, see [Configure Azure Key Vault Integration for SQL Server on Azure VMs](#). It's also recommended to store application secrets, such as database connection strings, in Key Vault.

We recommend enabling [DDoS Protection Standard](#), which provides additional DDoS mitigation for resources in a VNet. Although basic DDoS protection is automatically enabled as part of the Azure platform, DDoS Protection Standard provides mitigation capabilities that are tuned specifically to Azure Virtual Network resources.

## Deploy the solution

A deployment for this reference architecture is available on [GitHub](#). Note that the entire deployment can take up to two hours, which includes running the scripts to configure AD DS, the Windows Server failover cluster, and the SQL Server availability group.

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

## Deploy the solution

1. Run the following command to create a resource group.

```
az group create --location <location> --name <resource-group-name>
```

2. Run the following command to create a Storage account for the Cloud Witness.

```
az storage account create --location <location> \
--name <storage-account-name> \
--resource-group <resource-group-name> \
--sku Standard_LRS
```

3. Navigate to the `virtual-machines\nt-tier-windows` folder of the reference architectures GitHub repository.

4. Open the `nt-tier-windows.json` file.

5. Search for all instances of "witnessStorageBlobEndPoint" and replace the placeholder text with the name of the Storage account from step 2.

```
"witnessStorageBlobEndPoint": "https://[replace-with-storageaccountname].blob.core.windows.net",
```

6. Run the following command to list the account keys for the storage account.

```
az storage account keys list \
--account-name <storage-account-name> \
--resource-group <resource-group-name>
```

The output should look like the following. Copy the value of `key1`.

```
[  
{  
    "keyName": "key1",  
    "permissions": "Full",  
    "value": "..."  
,  
{  
    "keyName": "key2",  
    "permissions": "Full",  
    "value": "..."  
}  
]
```

7. In the `n-tier-windows.json` file, search for all instances of "witnessStorageAccountKey" and paste in the account key.

```
"witnessStorageAccountKey": "[replace-with-storagekey]"
```

8. In the `n-tier-windows.json` file, search for all instances of `[replace-with-password]` and `[replace-with-sql-password]` replace them with a strong password. Save the file.

**NOTE**

If you change the administrator user name, you must also update the `extensions` blocks in the JSON file.

9. Run the following command to deploy the architecture.

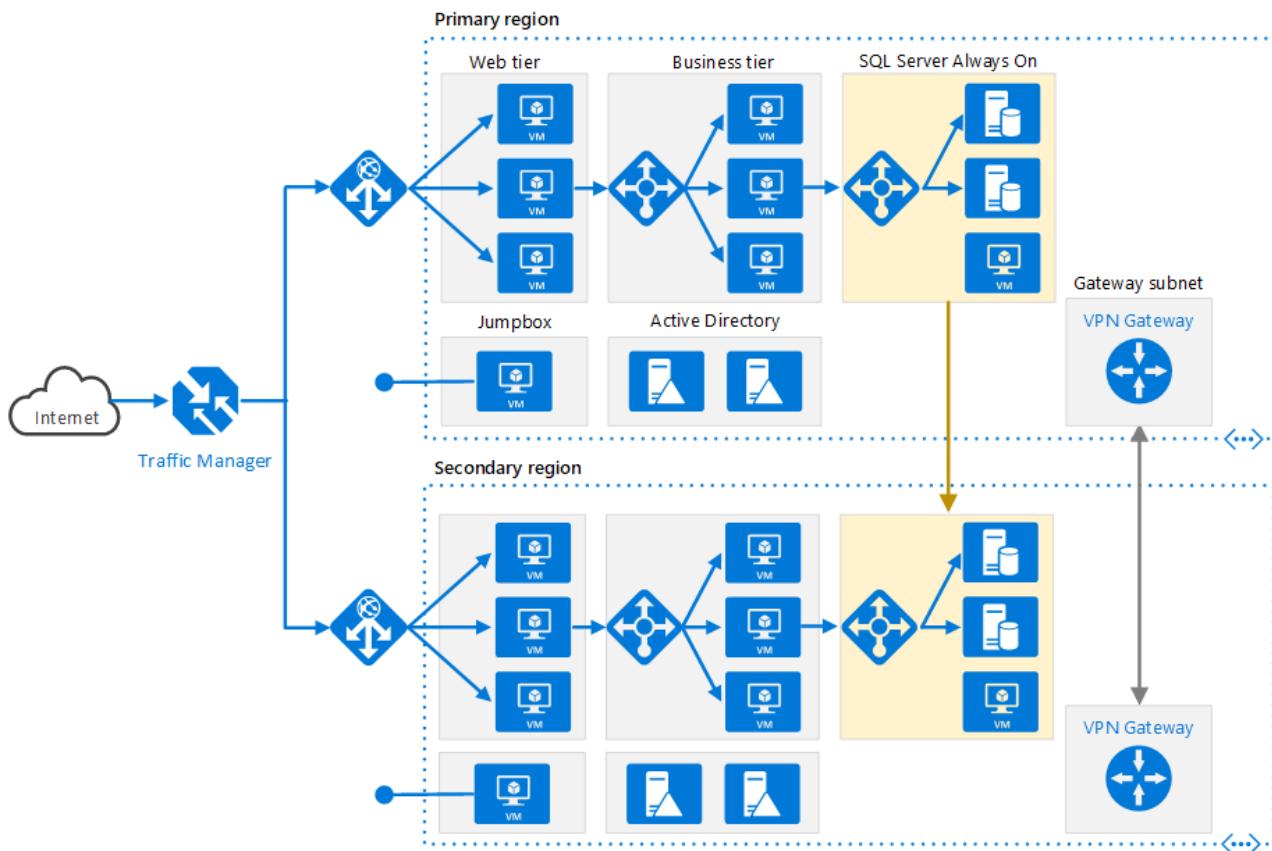
```
azbb -s <your subscription_id> -g <resource_group_name> -l <location> -p n-tier-windows.json --deploy
```

For more information on deploying this sample reference architecture using Azure Building Blocks, visit the [GitHub repository](#).

# Multi-region N-tier application for high availability

10/5/2018 • 8 minutes to read • [Edit Online](#)

This reference architecture shows a set of proven practices for running an N-tier application in multiple Azure regions, in order to achieve availability and a robust disaster recovery infrastructure.



[Download a Visio file of this architecture.](#)

## Architecture

This architecture builds on the one shown in [N-tier application with SQL Server](#).

- **Primary and secondary regions.** Use two regions to achieve higher availability. One is the primary region. The other region is for failover.
- **Azure Traffic Manager.** [Traffic Manager](#) routes incoming requests to one of the regions. During normal operations, it routes 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](#).
- **Resource groups.** Create separate [resource groups](#) for the primary region, the secondary region, and for Traffic Manager. This gives you the flexibility to manage each region as a single collection of resources. For example, you could redeploy one region, without taking down the other one. [Link the resource groups](#), so that you can run a query to list all the resources for the application.
- **VNets.** Create a separate VNet for each region. Make sure the address spaces do not overlap.
- **SQL Server Always On Availability Group.** If you are using SQL Server, we recommend [SQL Always On Availability Groups](#) for high availability. Create a single availability group that includes the SQL Server instances in both regions.

#### NOTE

Also consider [Azure SQL Database](#), which provides a relational database as a cloud service. With SQL Database, you don't need to configure an availability group or manage failover.

- **VPN Gateways.** Create a [VPN gateway](#) in each VNet, and configure a [VNet-to-VNet connection](#), to enable network traffic between the two VNets. This is required for the SQL Always On Availability Group.

## Recommendations

A multi-region architecture can provide higher availability than deploying to a single region. If a regional outage affects the primary region, you can use [Traffic Manager](#) to fail over to the secondary region. This architecture can also help if an individual subsystem of the application fails.

There are several general approaches to achieving high availability across regions:

- Active/passive with hot standby. Traffic goes to one region, while the other waits on hot standby. Hot standby means the VMs in the secondary region are allocated and running at all times.
- Active/passive with cold standby. Traffic goes to one region, while the other waits on cold standby. Cold standby means the VMs in the secondary region are not allocated until needed for failover. This approach costs less to run, but will generally take longer to come online during a failure.
- Active/active. Both regions are active, and requests are load balanced between them. If one region becomes unavailable, it is taken out of rotation.

This reference architecture focuses on active/passive with hot standby, using Traffic Manager for failover. Note that you could deploy a small number of VMs for hot standby and then scale out as needed.

### Regional pairing

Each Azure region is paired with another region within the same geography. In general, choose regions from the same regional pair (for example, East US 2 and US Central). Benefits of doing so include:

- If there is a broad outage, recovery of at least one region out of every pair is prioritized.
- Planned Azure system updates are rolled out to paired regions sequentially, to minimize possible downtime.
- Pairs reside within the same geography, to meet data residency requirements.

However, make sure that both regions support all of the Azure services needed for your application (see [Services by region](#)). For more information about regional pairs, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

### Traffic Manager configuration

Consider the following points when configuring Traffic Manager:

- **Routing.** Traffic Manager supports several [routing algorithms](#). For the scenario described in this article, use [priority routing](#) (formerly called *failover* routing). With this setting, Traffic Manager sends all requests to the primary region, unless the primary region becomes unreachable. At that point, it automatically fails over to the secondary region. See [Configure Failover routing method](#).
- **Health probe.** Traffic Manager uses an HTTP (or HTTPS) [probe](#) to monitor the availability of each region. The probe checks for an HTTP 200 response for a specified URL path. As a best practice, create an endpoint that reports the overall health of the application, and use this endpoint for the health probe. Otherwise, the probe might report a healthy endpoint when critical parts of the application are actually failing. For more information, see [Health Endpoint Monitoring Pattern](#).

When Traffic Manager fails over there is a period of time when clients cannot reach the application. The duration is affected by the following factors:

- The health probe must detect that the primary region has become unreachable.
- DNS servers must update the cached DNS records for the IP address, which depends on the DNS time-to-live (TTL). The default TTL is 300 seconds (5 minutes), but you can configure this value when you create the Traffic Manager profile.

For details, see [About Traffic Manager Monitoring](#).

If Traffic Manager fails over, we recommend performing a manual failback rather than implementing an automatic failback. Otherwise, you can create a situation where the application flips back and forth between regions. Verify that all application subsystems are healthy before failing back.

Note that Traffic Manager automatically fails back by default. To prevent this, manually lower the priority of the primary region after a failover event. For example, suppose the primary region is priority 1 and the secondary is priority 2. After a failover, set the primary region to priority 3, to prevent automatic failback. When you are ready to switch back, update the priority to 1.

The following [Azure CLI](#) command updates the priority:

```
az network traffic-manager endpoint update --resource-group <resource-group> --profile-name <profile>
--name <endpoint-name> --type azureEndpoints --priority 3
```

Another approach is to temporarily disable the endpoint until you are ready to fail back:

```
az network traffic-manager endpoint update --resource-group <resource-group> --profile-name <profile>
--name <endpoint-name> --type azureEndpoints --endpoint-status Disabled
```

Depending on the cause of a failover, you might need to redeploy the resources within a region. Before failing back, perform an operational readiness test. The test should verify things like:

- VMs are configured correctly. (All required software is installed, IIS is running, and so on.)
- Application subsystems are healthy.
- Functional testing. (For example, the database tier is reachable from the web tier.)

## Configure SQL Server Always On Availability Groups

Prior to Windows Server 2016, SQL Server Always On Availability Groups require a domain controller, and all nodes in the availability group must be in the same Active Directory (AD) domain.

To configure the availability group:

- At a minimum, place two domain controllers in each region.
- Give each domain controller a static IP address.
- Create a VNet-to-VNet connection to enable communication between the VNets.
- For each VNet, add the IP addresses of the domain controllers (from both regions) to the DNS server list.  
You can use the following CLI command. For more information, see [Change DNS servers](#).

```
az network vnet update --resource-group <resource-group> --name <vnet-name> --dns-servers
"10.0.0.4,10.0.0.6,172.16.0.4,172.16.0.6"
```

- Create a [Windows Server Failover Clustering](#) (WSFC) cluster that includes the SQL Server instances in both regions.
- Create a SQL Server Always On Availability Group that includes the SQL Server instances in both the primary and secondary regions. See [Extending Always On Availability Group to Remote Azure Datacenter](#)

([PowerShell](#)) for the steps.

- Put the primary replica in the primary region.
- Put one or more secondary replicas in the primary region. Configure these to use synchronous commit with automatic failover.
- Put one or more secondary replicas in the secondary region. Configure these to use *asynchronous* commit, for performance reasons. (Otherwise, all T-SQL transactions have to wait on a round trip over the network to the secondary region.)

**NOTE**

Asynchronous commit replicas do not support automatic failover.

## Availability considerations

With a complex N-tier app, you may not need to replicate the entire application in the secondary region. Instead, you might just replicate a critical subsystem that is needed to support business continuity.

Traffic Manager is a possible failure point in the system. If the Traffic Manager service fails, clients cannot access your application during the downtime. 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 fallback. If the Azure Traffic Manager service fails, change your CNAME records in DNS to point to the other traffic management service. (This step must be performed manually, and your application will be unavailable until the DNS changes are propagated.)

For the SQL Server cluster, there are two failover scenarios to consider:

- All of the SQL Server database replicas in the primary region fail. For example, this could happen during a regional outage. In that case, you must manually fail over the availability group, even though Traffic Manager automatically fails over on the front end. Follow the steps in [Perform a Forced Manual Failover of a SQL Server Availability Group](#), which describes how to perform a forced failover by using SQL Server Management Studio, Transact-SQL, or PowerShell in SQL Server 2016.

**WARNING**

With forced failover, there is a risk of data loss. Once the primary region is back online, take a snapshot of the database and use [tablediff](#) to find the differences.

- Traffic Manager fails over to the secondary region, but the primary SQL Server database replica is still available. For example, the front-end tier might fail, without affecting the SQL Server VMs. In that case, Internet traffic is routed to the secondary region, and that region can still connect to the primary replica. However, there will be increased latency, because the SQL Server connections are going across regions. In this situation, you should perform a manual failover as follows:

1. Temporarily switch a SQL Server database replica in the secondary region to *synchronous* commit. This ensures there won't be data loss during the failover.
2. Fail over to that replica.
3. When you fail back to the primary region, restore the asynchronous commit setting.

## Manageability considerations

When you update your deployment, update one region at a time to reduce the chance of a global failure from an incorrect configuration or an error in the application.

Test the resiliency of the system to failures. Here are some common failure scenarios to test:

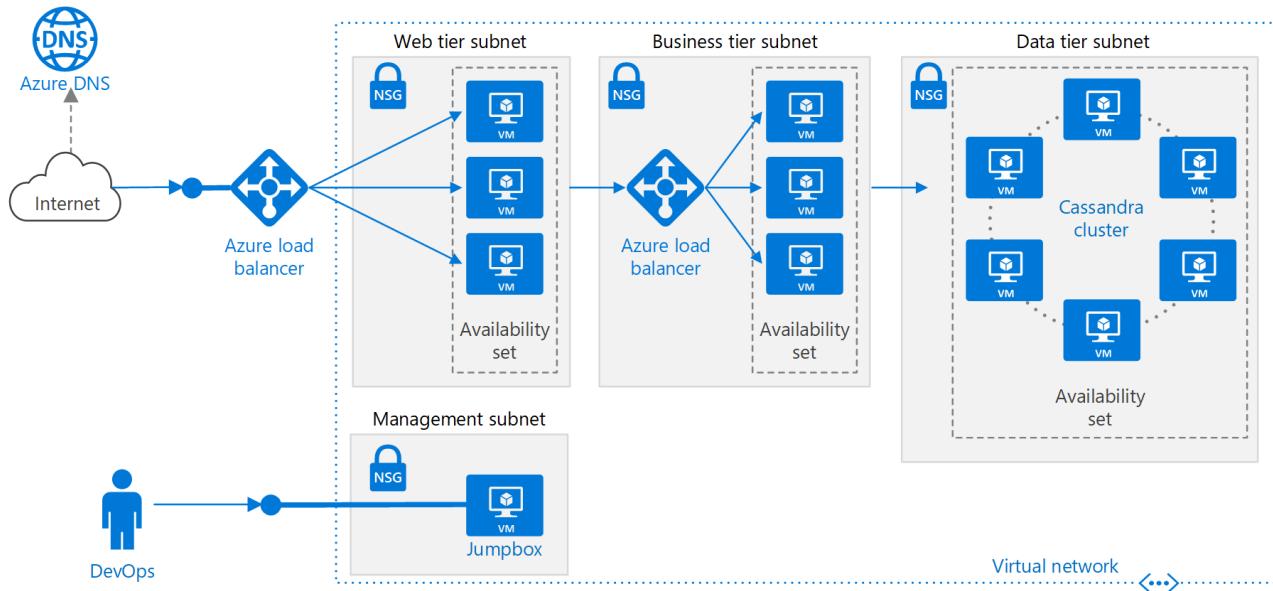
- Shut down VM instances.
- Pressure resources such as CPU and memory.
- Disconnect/delay network.
- Crash processes.
- Expire certificates.
- Simulate hardware faults.
- Shut down the DNS service on the domain controllers.

Measure the recovery times and verify they meet your business requirements. Test combinations of failure modes, as well.

# N-tier application with Apache Cassandra

10/9/2018 • 10 minutes to read • [Edit Online](#)

This reference architecture shows how to deploy VMs and a virtual network configured for an N-tier application, using Apache Cassandra on Linux for the data tier. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

## Architecture

The architecture has the following components:

- **Resource group.** [Resource groups](#) are used to group resources so they can be managed by lifetime, owner, or other criteria.
- **Virtual network (VNet) and subnets.** Every Azure VM is deployed into a VNet that can be segmented into multiple subnets. Create a separate subnet for each tier.
- **NSGs.** Use [network security groups](#) (NSGs) to restrict network traffic within the VNet. For example, in the 3-tier architecture shown here, the database tier does not accept traffic from the web front end, only from the business tier and the management subnet.
- **Virtual machines.** For recommendations on configuring VMs, see [Run a Windows VM on Azure](#) and [Run a Linux VM on Azure](#).
- **Availability sets.** Create an [availability set](#) for each tier, and provision at least two VMs in each tier. This makes the VMs eligible for a higher [service level agreement \(SLA\)](#) for VMs.
- **VM scale set** (not shown). A [VM scale set](#) is an alternative to using an availability set. A scale sets makes it easy to scale out the VMs in a tier, either manually or automatically based on predefined rules.
- **Azure Load balancers.** The [load balancers](#) distribute incoming Internet requests to the VM instances. Use a [public load balancer](#) to distribute incoming Internet traffic to the web tier, and an [internal load balancer](#) to distribute network traffic from the web tier to the business tier.
- **Public IP address.** A public IP address is needed for the public load balancer to receive Internet traffic.

- **Jumpbox.** Also called a [bastion host](#). A secure VM on the network that administrators use to connect to the other VMs. The jumpbox has an NSG that allows remote traffic only from public IP addresses on a safe list. The NSG should permit ssh traffic.
- **Apache Cassandra database.** Provides high availability at the data tier, by enabling replication and failover.
- **Azure DNS.** [Azure DNS](#) is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.

## Recommendations

Your requirements might differ from the architecture described here. Use these recommendations as a starting point.

### VNet / Subnets

When you create the VNet, determine how many IP addresses your resources in each subnet require. Specify a subnet mask and a VNet address range large enough for the required IP addresses, using [CIDR](#) notation. Use an address space that falls within the standard [private IP address blocks](#), which are 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16.

Choose an address range that does not overlap with your on-premises network, in case you need to set up a gateway between the VNet and your on-premise network later. Once you create the VNet, you can't change the address range.

Design subnets with functionality and security requirements in mind. All VMs within the same tier or role should go into the same subnet, which can be a security boundary. For more information about designing VNets and subnets, see [Plan and design Azure Virtual Networks](#).

### Load balancers

Do not expose the VMs directly to the Internet, but instead give each VM a private IP address. Clients connect using the IP address of the public load balancer.

Define load balancer rules to direct network traffic to the VMs. For example, to enable HTTP traffic, create a rule that maps port 80 from the front-end configuration to port 80 on the back-end address pool. When a client sends an HTTP request to port 80, the load balancer selects a back-end IP address by using a [hashing algorithm](#) that includes the source IP address. In that way, client requests are distributed across all the VMs.

### Network security groups

Use NSG rules to restrict traffic between tiers. For example, in the 3-tier architecture shown above, the web tier does not communicate directly with the database tier. To enforce this, the database tier should block incoming traffic from the web tier subnet.

1. Deny all inbound traffic from the VNet. (Use the `VIRTUAL_NETWORK` tag in the rule.)
2. Allow inbound traffic from the business tier subnet.
3. Allow inbound traffic from the database tier subnet itself. This rule allows communication between the database VMs, which is needed for database replication and failover.
4. Allow ssh traffic (port 22) from the jumpbox subnet. This rule lets administrators connect to the database tier from the jumpbox.

Create rules 2 – 4 with higher priority than the first rule, so they override it.

### Cassandra

We recommend [DataStax Enterprise](#) for production use, but these recommendations apply to any Cassandra edition. For more information on running DataStax in Azure, see [DataStax Enterprise Deployment Guide for Azure](#).

Put the VMs for a Cassandra cluster in an availability set to ensure that the Cassandra replicas are distributed across multiple fault domains and upgrade domains. For more information about fault domains and upgrade domains, see [Manage the availability of virtual machines](#).

Configure three fault domains (the maximum) per availability set and 18 upgrade domains per availability set. This provides the maximum number of upgrade domains that can still be distributed evenly across the fault domains.

Configure nodes in rack-aware mode. Map fault domains to racks in the `cassandra-rackdc.properties` file.

You don't need a load balancer in front of the cluster. The client connects directly to a node in the cluster.

For high availability, deploy Cassandra in more than one Azure region. Within each region, nodes are configured in rack-aware mode with fault and upgrade domains, for resiliency inside the region.

### Jumpbox

Do not allow ssh access from the public Internet to the VMs that run the application workload. Instead, all ssh access to these VMs must come through the jumpbox. An administrator logs into the jumpbox, and then logs into the other VM from the jumpbox. The jumpbox allows ssh traffic from the Internet, but only from known, safe IP addresses.

The jumpbox has minimal performance requirements, so select a small VM size. Create a [public IP address](#) for the jumpbox. Place the jumpbox in the same VNet as the other VMs, but in a separate management subnet.

To secure the jumpbox, add an NSG rule that allows ssh connections only from a safe set of public IP addresses. Configure the NSGs for the other subnets to allow ssh traffic from the management subnet.

## Scalability considerations

[VM scale sets](#) help you to deploy and manage a set of identical VMs. Scale sets support autoscaling based on performance metrics. As the load on the VMs increases, additional VMs are automatically added to the load balancer. Consider scale sets if you need to quickly scale out VMs, or need to autoscale.

There are two basic ways to configure VMs deployed in a scale set:

- Use extensions to configure the VM after it is provisioned. With this approach, new VM instances may take longer to start up than a VM with no extensions.
- Deploy a [managed disk](#) with a custom disk image. This option may be quicker to deploy. However, it requires you to keep the image up to date.

For additional considerations, see [Design considerations for scale sets](#).

#### TIP

When using any autoscale solution, test it with production-level workloads well in advance.

Each Azure subscription has default limits in place, including a maximum number of VMs per region. You can increase the limit by filing a support request. For more information, see [Azure subscription and service limits, quotas, and constraints](#).

## Availability considerations

If you are not using VM scale sets, put VMs in the same tier into an availability set. Create at least two VMs in the availability set to support the [availability SLA for Azure VMs](#). For more information, see [Manage the availability of virtual machines](#).

The load balancer uses [health probes](#) to monitor the availability of VM instances. If a probe cannot reach an

instance within a timeout period, the load balancer stops sending traffic to that VM. However, the load balancer will continue to probe, and if the VM becomes available again, the load balancer resumes sending traffic to that VM.

Here are some recommendations on load balancer health probes:

- Probes can test either HTTP or TCP. If your VMs run an HTTP server, create an HTTP probe. Otherwise create a TCP probe.
- For an HTTP probe, specify the path to an HTTP endpoint. The probe checks for an HTTP 200 response from this path. This can be the root path (""/"), or a health-monitoring endpoint that implements some custom logic to check the health of the application. The endpoint must allow anonymous HTTP requests.
- The probe is sent from a [known IP address](#), 168.63.129.16. Make sure you don't block traffic to or from this IP address in any firewall policies or network security group (NSG) rules.
- Use [health probe logs](#) to view the status of the health probes. Enable logging in the Azure portal for each load balancer. Logs are written to Azure Blob storage. The logs show how many VMs on the back end are not receiving network traffic due to failed probe responses.

For the Cassandra cluster, the failover scenarios to consider depend on the consistency levels used by the application, as well as the number of replicas used. For consistency levels and usage in Cassandra, see [Configuring data consistency](#) and [Cassandra: How many nodes are talked to with Quorum?](#) Data availability in Cassandra is determined by the consistency level used by the application and the replication mechanism. For replication in Cassandra, see [Data Replication in NoSQL Databases Explained](#).

## Security considerations

Virtual networks are a traffic isolation boundary in Azure. VMs in one VNet cannot communicate directly with VMs in a different VNet. VMs within the same VNet can communicate, unless you create [network security groups](#) (NSGs) to restrict traffic. For more information, see [Microsoft cloud services and network security](#).

For incoming Internet traffic, the load balancer rules define which traffic can reach the back end. However, load balancer rules don't support IP safe lists, so if you want to add certain public IP addresses to a safe list, add an NSG to the subnet.

Consider adding a network virtual appliance (NVA) to create a DMZ between the Internet and the Azure virtual network. NVA is a generic term for a virtual appliance that can perform network-related tasks, such as firewall, packet inspection, auditing, and custom routing. For more information, see [Implementing a DMZ between Azure and the Internet](#).

Encrypt sensitive data at rest and use [Azure Key Vault](#) to manage the database encryption keys. Key Vault can store encryption keys in hardware security modules (HSMs). It's also recommended to store application secrets, such as database connection strings, in Key Vault.

We recommend enabling [DDoS Protection Standard](#), which provides additional DDoS mitigation for resources in a VNet. Although basic DDoS protection is automatically enabled as part of the Azure platform, DDoS Protection Standard provides mitigation capabilities that are tuned specifically to Azure Virtual Network resources.

## Deploy the solution

A deployment for this reference architecture is available on [GitHub](#).

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Deploy the solution using azbb

To deploy the Linux VMs for an N-tier application reference architecture, follow these steps:

1. Navigate to the `virtual-machines\n-tier-linux` folder for the repository you cloned in step 1 of the prerequisites above.
2. The parameter file specifies a default administrator user name and password for each VM in the deployment. You must change these before you deploy the reference architecture. Open the `n-tier-linux.json` file and replace each **adminUsername** and **adminPassword** field with your new settings. Save the file.
3. Deploy the reference architecture using the **azbb** command line tool as shown below.

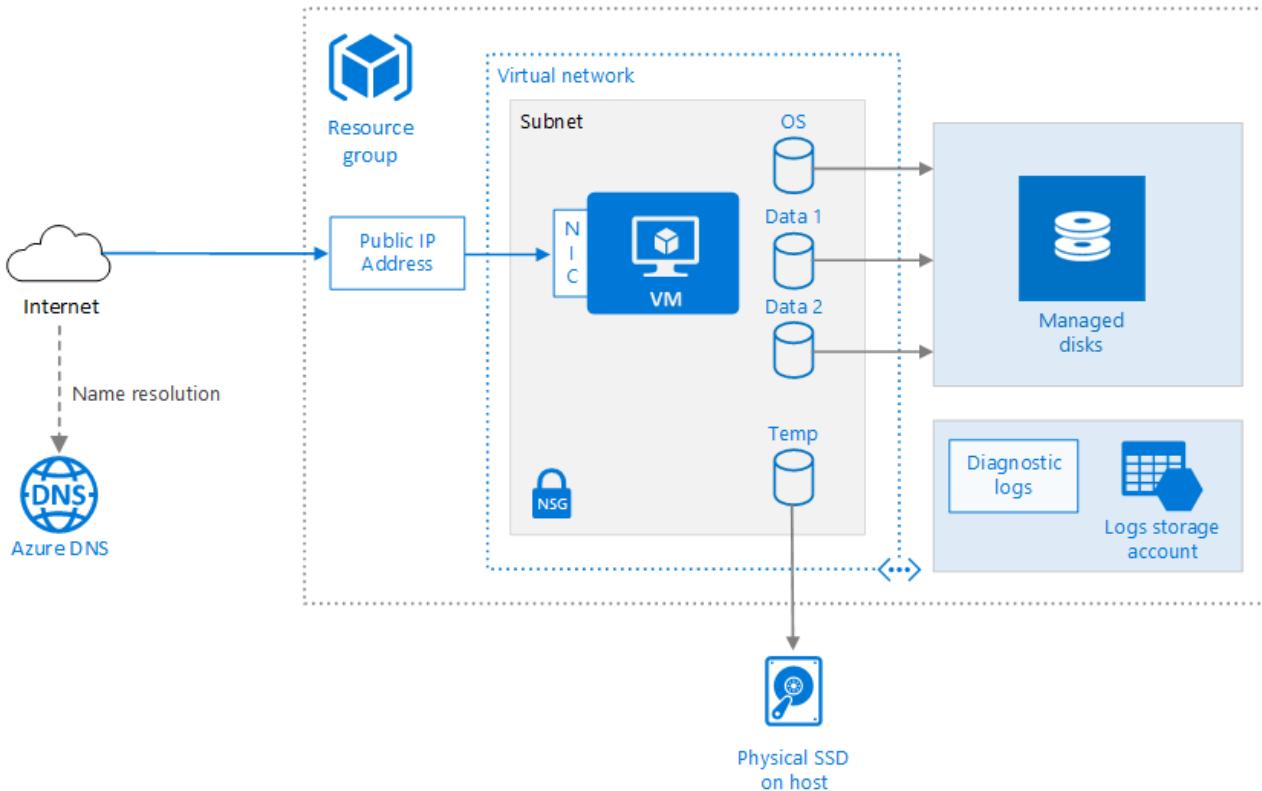
```
azbb -s <your subscription_id> -g <your resource_group_name> -l <azure region> -p n-tier-linux.json --deploy
```

For more information on deploying this sample reference architecture using Azure Building Blocks, visit the [GitHub repository](#).

# Run a Linux VM on Azure

9/28/2018 • 10 minutes to read • [Edit Online](#)

This article describes a set of proven practices for running a Linux virtual machine (VM) on Azure. It includes recommendations for provisioning the VM along with networking and storage components. [Deploy this solution.](#)



## Components

Provisioning an Azure VM requires some additional components besides the VM itself, including networking and storage resources.

- **Resource group.** A [resource group](#) is a logical container that holds related Azure resources. In general, group resources based on their lifetime and who will manage them.
- **VM.** You can provision a VM from a list of published images, or from a custom managed image or virtual hard disk (VHD) file uploaded to Azure Blob storage. Azure supports running various popular Linux distributions, including CentOS, Debian, Red Hat Enterprise, Ubuntu, and FreeBSD. For more information, see [Azure and Linux](#).
- **Managed Disks.** [Azure Managed Disks](#) simplify disk management by handling the storage for you. The OS disk is a VHD stored in [Azure Storage](#), so it persists even when the host machine is down. For Linux VMs, the OS disk is `/dev/sda1`. We also recommend creating one or more [data disks](#), which are persistent VHDs used for application data.
- **Temporary disk.** The VM is created with a temporary disk. This disk is stored on a physical drive on the host machine. It is *not* saved in Azure Storage and may be deleted during reboots and other VM lifecycle events. Use this disk only for temporary data, such as page or swap files. For Linux VMs, the temporary disk is `/dev/sdb1` and is mounted at `/mnt/resource` or `/mnt`.

- **Virtual network (VNet).** Every Azure VM is deployed into a VNet that can be segmented into multiple subnets.
- **Network interface (NIC).** The NIC enables the VM to communicate with the virtual network.
- **Public IP address.** A public IP address is needed to communicate with the VM — for example, via SSH.
- **Azure DNS.** [Azure DNS](#) is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.
- **Network security group (NSG).** [Network security groups](#) are used to allow or deny network traffic to VMs. NSGs can be associated either with subnets or with individual VM instances.
- **Diagnostics.** Diagnostic logging is crucial for managing and troubleshooting the VM.

## VM recommendations

Azure offers many different virtual machine sizes. For more information, see [Sizes for virtual machines in Azure](#). If you are moving an existing workload to Azure, start with the VM size that's the closest match to your on-premises servers. Then measure the performance of your actual workload with respect to CPU, memory, and disk input/output operations per second (IOPS), and adjust the size as needed. If you require multiple NICs for your VM, be aware that a maximum number of NICs is defined for each [VM size](#).

Generally, choose an Azure region that is closest to your internal users or customers. However, not all VM sizes are available in all regions. For more information, see [Services by region](#). For a list of the VM sizes available in a specific region, run the following command from the Azure command-line interface (CLI):

```
az vm list-sizes --location <location>
```

For information about choosing a published VM image, see [Find Linux VM images](#).

Enable monitoring and diagnostics, including basic health metrics, diagnostics infrastructure logs, and [boot diagnostics](#). Boot diagnostics can help you diagnose boot failure if your VM gets into a non-bootable state. For more information, see [Enable monitoring and diagnostics](#).

## Disk and storage recommendations

For best disk I/O performance, we recommend [Premium Storage](#), which stores data on solid-state drives (SSDs). Cost is based on the capacity of the provisioned disk. IOPS and throughput (that is, data transfer rate) also depend on disk size, so when you provision a disk, consider all three factors (capacity, IOPS, and throughput).

We also recommend using [Managed Disks](#). Managed disks do not require a storage account. You simply specify the size and type of disk and it is deployed as a highly available resource.

Add one or more data disks. When you create a VHD, it is unformatted. Log into the VM to format the disk. In the Linux shell, data disks are displayed as `/dev/sdc`, `/dev/sdd`, and so on. You can run `lsblk` to list the block devices, including the disks. To use a data disk, create a partition and file system, and mount the disk. For example:

```
# Create a partition.  
sudo fdisk /dev/sdc      # Enter 'n' to partition, 'w' to write the change.  
  
# Create a file system.  
sudo mkfs -t ext3 /dev/sdc1  
  
# Mount the drive.  
sudo mkdir /data1  
sudo mount /dev/sdc1 /data1
```

When you add a data disk, a logical unit number (LUN) ID is assigned to the disk. Optionally, you can specify the LUN ID — for example, if you're replacing a disk and want to retain the same LUN ID, or you have an application that looks for a specific LUN ID. However, remember that LUN IDs must be unique for each disk.

You may want to change the I/O scheduler to optimize for performance on SSDs because the disks for VMs with premium storage accounts are SSDs. A common recommendation is to use the NOOP scheduler for SSDs, but you should use a tool such as [iostat](#) to monitor disk I/O performance for your workload.

Create a storage account to hold diagnostic logs. A standard locally redundant storage (LRS) account is sufficient for diagnostic logs.

#### NOTE

If you aren't using Managed Disks, create separate Azure storage accounts for each VM to hold the virtual hard disks (VHDs), in order to avoid hitting the [\(IOPS\) limits](#) for storage accounts. Be aware of the total I/O limits of the storage account. For more information, see [virtual machine disk limits](#).

## Network recommendations

The public IP address can be dynamic or static. The default is dynamic.

- Reserve a [static IP address](#) if you need a fixed IP address that won't change — for example, if you need to create an A record in DNS, or need the IP address to be added to a safe list.
- You can also create a fully qualified domain name (FQDN) for the IP address. You can then register a [CNAME record](#) in DNS that points to the FQDN. For more information, see [Create a fully qualified domain name in the Azure portal](#). You can use [Azure DNS](#) or another DNS service.

All NSGs contain a set of [default rules](#), including a rule that blocks all inbound Internet traffic. The default rules cannot be deleted, but other rules can override them. To enable Internet traffic, create rules that allow inbound traffic to specific ports — for example, port 80 for HTTP.

To enable SSH, add an NSG rule that allows inbound traffic to TCP port 22.

## Scalability considerations

You can scale a VM up or down by [changing the VM size](#). To scale out horizontally, put two or more VMs behind a load balancer. For more information, see the [N-tier reference architecture](#).

## Availability considerations

For higher availability, deploy multiple VMs in an availability set. This also provides a higher [service level agreement \(SLA\)](#).

Your VM may be affected by [planned maintenance](#) or [unplanned maintenance](#). You can use [VM reboot logs](#) to determine whether a VM reboot was caused by planned maintenance.

To protect against accidental data loss during normal operations (for example, because of user error), you should also implement point-in-time backups, using [blob snapshots](#) or another tool.

## Manageability considerations

**Resource groups.** Put closely associated resources that share the same lifecycle into the same [resource group](#). Resource groups allow you to deploy and monitor resources as a group and track billing costs by resource group. You can also delete resources as a set, which is very useful for test deployments. Assign meaningful resource names to simplify locating a specific resource and understanding its role. For more information, see [Recommended naming conventions for Azure resources](#).

**SSH.** Before you create a Linux VM, generate a 2048-bit RSA public-private key pair. Use the public key file when you create the VM. For more information, see [How to Use SSH with Linux and Mac on Azure](#).

**Stopping a VM.** Azure makes a distinction between "stopped" and "deallocated" states. You are charged when the VM status is stopped, but not when the VM is deallocated. In the Azure portal, the **Stop** button deallocates the VM. If you shut down through the OS while logged in, the VM is stopped but **not** deallocated, so you will still be charged.

**Deleting a VM.** If you delete a VM, the VHDs are not deleted. That means you can safely delete the VM without losing data. However, you will still be charged for storage. To delete the VHD, delete the file from [Blob storage](#). To prevent accidental deletion, use a [resource lock](#) to lock the entire resource group or lock individual resources, such as a VM.

## Security considerations

Use [Azure Security Center](#) to get a central view of the security state of your Azure resources. Security Center monitors potential security issues and provides a comprehensive picture of the security health of your deployment. Security Center is configured per Azure subscription. Enable security data collection as described in the [Azure Security Center quick start guide](#). When data collection is enabled, Security Center automatically scans any VMs created under that subscription.

**Patch management.** If enabled, Security Center checks whether any security and critical updates are missing.

**Antimalware.** If enabled, Security Center checks whether antimalware software is installed. You can also use Security Center to install antimalware software from inside the Azure portal.

**Operations.** Use [role-based access control \(RBAC\)](#) to control access to the Azure resources that you deploy. RBAC lets you assign authorization roles to members of your DevOps team. For example, the Reader role can view Azure resources but not create, manage, or delete them. Some roles are specific to particular Azure resource types. For example, the Virtual Machine Contributor role can restart or deallocate a VM, reset the administrator password, create a new VM, and so on. Other [built-in RBAC roles](#) that may be useful for this architecture include [DevTest Labs User](#) and [Network Contributor](#). A user can be assigned to multiple roles, and you can create custom roles for even more fine-grained permissions.

### NOTE

RBAC does not limit the actions that a user logged into a VM can perform. Those permissions are determined by the account type on the guest OS.

Use [audit logs](#) to see provisioning actions and other VM events.

**Data encryption.** Consider [Azure Disk Encryption](#) if you need to encrypt the OS and data disks.

**DDoS protection.** We recommend enabling [DDoS Protection Standard](#), which provides additional DDoS mitigation for resources in a VNet. Although basic DDoS protection is automatically enabled as part of the Azure

platform, DDoS Protection Standard provides mitigation capabilities that are tuned specifically to Azure Virtual Network resources.

## Deploy the solution

A deployment is available on [GitHub](#). It deploys the following:

- A virtual network with a single subnet named **web** used to host the VM.
- An NSG with two incoming rules to allow SSH and HTTP traffic to the VM.
- A VM running the latest version of Ubuntu 16.04.3 LTS.
- A sample custom script extension that formats the two data disks and deploys Apache HTTP Server to the Ubuntu VM.

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

5. Create an SSH key pair. For more information, see [How to create and use an SSH public and private key pair for Linux VMs in Azure](#).

### Deploy the solution using azbb

1. Navigate to the `virtual-machines/single-vm/parameters/linux` folder for the repository you downloaded in the prerequisites step above.
2. Open the `single-vm-v2.json` file and enter a username and your SSH public key between the quotes, then save the file.

```
"adminUsername": "<your username>",
"sshPublicKey": "ssh-rsa AAAAB3NzaC1...",
```

3. Run `azbb` to deploy the sample VM as shown below.

```
azbb -s <subscription_id> -g <resource_group_name> -l <location> -p single-vm-v2.json --deploy
```

To verify the deployment, run the following Azure CLI command to find the public IP address of the VM:

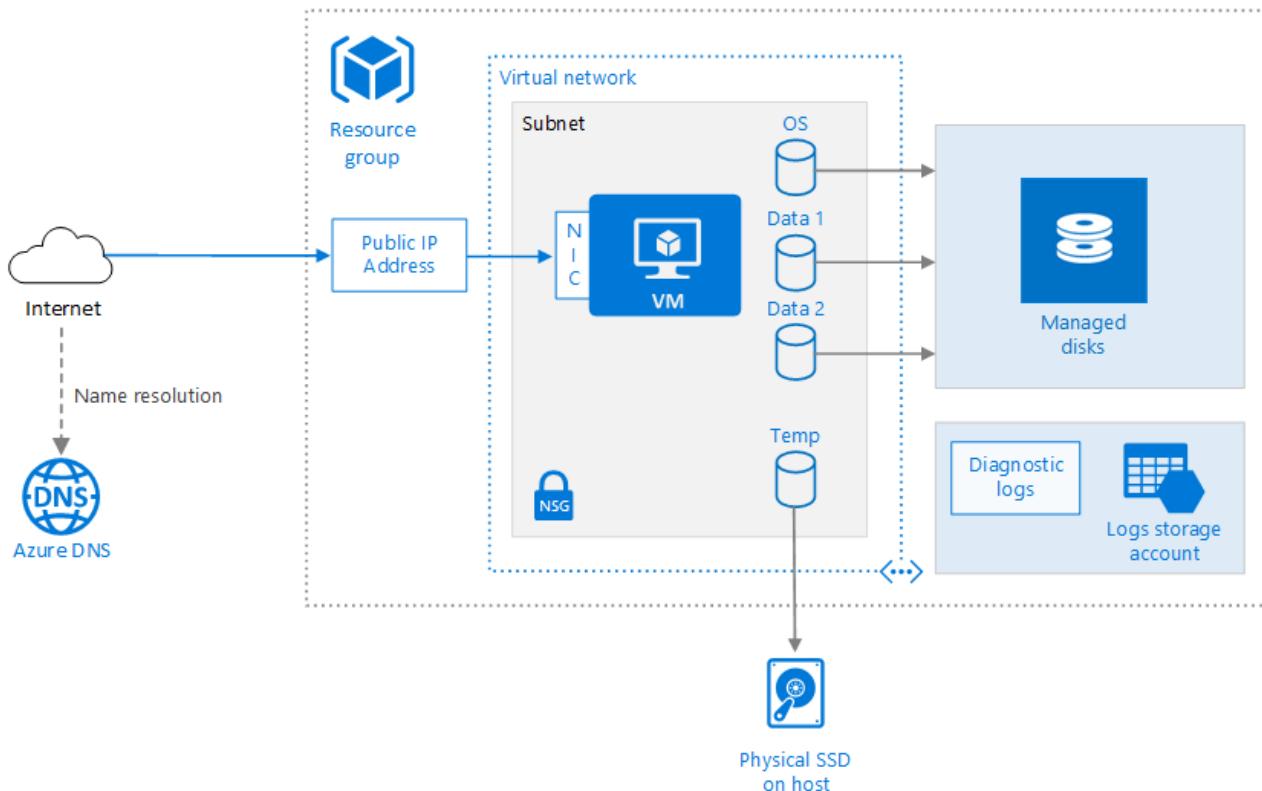
```
az vm show -n ra-single-linux-vm1 -g <resource-group-name> -d -o table
```

If you navigate to this address in a web browser, you should see the default Apache2 homepage.

# Run a Windows VM on Azure

9/14/2018 • 9 minutes to read • [Edit Online](#)

This article describes a set of proven practices for running a Windows virtual machine (VM) on Azure. It includes recommendations for provisioning the VM along with networking and storage components. [Deploy this solution.](#)



## Components

Provisioning an Azure VM requires some additional components besides the VM itself, including networking and storage resources.

- **Resource group.** A [resource group](#) is a logical container that holds related Azure resources. In general, group resources based on their lifetime and who will manage them.
- **VM.** You can provision a VM from a list of published images, or from a custom managed image or virtual hard disk (VHD) file uploaded to Azure Blob storage.
- **Managed Disks.** [Azure Managed Disks](#) simplify disk management by handling the storage for you. The OS disk is a VHD stored in [Azure Storage](#), so it persists even when the host machine is down. We also recommend creating one or more [data disks](#), which are persistent VHDs used for application data.
- **Temporary disk.** The VM is created with a temporary disk (the **D:** drive on Windows). This disk is stored on a physical drive on the host machine. It is *not* saved in Azure Storage and may be deleted during reboots and other VM lifecycle events. Use this disk only for temporary data, such as page or swap files.
- **Virtual network (VNet).** Every Azure VM is deployed into a VNet that can be segmented into multiple subnets.
- **Network interface (NIC).** The NIC enables the VM to communicate with the virtual network.

- **Public IP address.** A public IP address is needed to communicate with the VM — for example, via remote desktop (RDP).
- **Azure DNS.** Azure DNS is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.
- **Network security group (NSG).** Network security groups are used to allow or deny network traffic to VMs. NSGs can be associated either with subnets or with individual VM instances.
- **Diagnostics.** Diagnostic logging is crucial for managing and troubleshooting the VM.

## VM recommendations

Azure offers many different virtual machine sizes. For more information, see [Sizes for virtual machines in Azure](#). If you are moving an existing workload to Azure, start with the VM size that's the closest match to your on-premises servers. Then measure the performance of your actual workload with respect to CPU, memory, and disk input/output operations per second (IOPS), and adjust the size as needed. If you require multiple NICs for your VM, be aware that a maximum number of NICs is defined for each [VM size](#).

Generally, choose an Azure region that is closest to your internal users or customers. However, not all VM sizes are available in all regions. For more information, see [Services by region](#). For a list of the VM sizes available in a specific region, run the following command from the Azure command-line interface (CLI):

```
az vm list-sizes --location <location>
```

For information about choosing a published VM image, see [Find Windows VM images](#).

Enable monitoring and diagnostics, including basic health metrics, diagnostics infrastructure logs, and [boot diagnostics](#). Boot diagnostics can help you diagnose boot failure if your VM gets into a non-bootable state. For more information, see [Enable monitoring and diagnostics](#).

## Disk and storage recommendations

For best disk I/O performance, we recommend [Premium Storage](#), which stores data on solid-state drives (SSDs). Cost is based on the capacity of the provisioned disk. IOPS and throughput (that is, data transfer rate) also depend on disk size, so when you provision a disk, consider all three factors (capacity, IOPS, and throughput).

We also recommend using [Managed Disks](#). Managed disks do not require a storage account. You simply specify the size and type of disk and it is deployed as a highly available resource.

Add one or more data disks. When you create a VHD, it is unformatted. Log into the VM to format the disk. When possible, install applications on a data disk, not the OS disk. Some legacy applications might need to install components on the C: drive; in that case, you can [resize the OS disk](#) using PowerShell.

Create a storage account to hold diagnostic logs. A standard locally redundant storage (LRS) account is sufficient for diagnostic logs.

### NOTE

If you aren't using Managed Disks, create separate Azure storage accounts for each VM to hold the virtual hard disks (VHDs), in order to avoid hitting the [\(IOPS\) limits](#) for storage accounts. Be aware of the total I/O limits of the storage account. For more information, see [virtual machine disk limits](#).

## Network recommendations

The public IP address can be dynamic or static. The default is dynamic.

- Reserve a [static IP address](#) if you need a fixed IP address that won't change — for example, if you need to create a DNS 'A' record or add the IP address to a safe list.
- You can also create a fully qualified domain name (FQDN) for the IP address. You can then register a [CNAME record](#) in DNS that points to the FQDN. For more information, see [Create a fully qualified domain name in the Azure portal](#).

All NSGs contain a set of [default rules](#), including a rule that blocks all inbound Internet traffic. The default rules cannot be deleted, but other rules can override them. To enable Internet traffic, create rules that allow inbound traffic to specific ports — for example, port 80 for HTTP.

To enable RDP, add an NSG rule that allows inbound traffic to TCP port 3389.

## Scalability considerations

You can scale a VM up or down by [changing the VM size](#). To scale out horizontally, put two or more VMs behind a load balancer. For more information, see the [N-tier reference architecture](#).

## Availability considerations

For higher availability, deploy multiple VMs in an availability set. This also provides a higher [service level agreement \(SLA\)](#).

Your VM may be affected by [planned maintenance](#) or [unplanned maintenance](#). You can use [VM reboot logs](#) to determine whether a VM reboot was caused by planned maintenance.

To protect against accidental data loss during normal operations (for example, because of user error), you should also implement point-in-time backups, using [blob snapshots](#) or another tool.

## Manageability considerations

**Resource groups.** Put closely associated resources that share the same lifecycle into the same [resource group](#). Resource groups allow you to deploy and monitor resources as a group and track billing costs by resource group. You can also delete resources as a set, which is very useful for test deployments. Assign meaningful resource names to simplify locating a specific resource and understanding its role. For more information, see [Recommended Naming Conventions for Azure Resources](#).

**Stopping a VM.** Azure makes a distinction between "stopped" and "deallocated" states. You are charged when the VM status is stopped, but not when the VM is deallocated. In the Azure portal, the **Stop** button deallocates the VM. If you shut down through the OS while logged in, the VM is stopped but **not** deallocated, so you will still be charged.

**Deleting a VM.** If you delete a VM, the VHDs are not deleted. That means you can safely delete the VM without losing data. However, you will still be charged for storage. To delete the VHD, delete the file from [Blob storage](#). To prevent accidental deletion, use a [resource lock](#) to lock the entire resource group or lock individual resources, such as a VM.

## Security considerations

Use [Azure Security Center](#) to get a central view of the security state of your Azure resources. Security Center monitors potential security issues and provides a comprehensive picture of the security health of your deployment. Security Center is configured per Azure subscription. Enable security data collection as described in the [Azure Security Center quick start guide](#). When data collection is enabled, Security Center automatically scans any VMs created under that subscription.

**Patch management.** If enabled, Security Center checks whether any security and critical updates are missing. Use [Group Policy settings](#) on the VM to enable automatic system updates.

**Antimalware.** If enabled, Security Center checks whether antimalware software is installed. You can also use Security Center to install antimalware software from inside the Azure portal.

**Operations.** Use [role-based access control \(RBAC\)](#) to control access to the Azure resources that you deploy. RBAC lets you assign authorization roles to members of your DevOps team. For example, the Reader role can view Azure resources but not create, manage, or delete them. Some roles are specific to particular Azure resource types. For example, the Virtual Machine Contributor role can restart or deallocate a VM, reset the administrator password, create a new VM, and so on. Other [built-in RBAC roles](#) that may be useful for this architecture include [DevTest Labs User](#) and [Network Contributor](#). A user can be assigned to multiple roles, and you can create custom roles for even more fine-grained permissions.

#### NOTE

RBAC does not limit the actions that a user logged into a VM can perform. Those permissions are determined by the account type on the guest OS.

Use [audit logs](#) to see provisioning actions and other VM events.

**Data encryption.** Consider [Azure Disk Encryption](#) if you need to encrypt the OS and data disks.

**DDoS protection.** We recommend enabling [DDoS Protection Standard](#), which provides additional DDoS mitigation for resources in a VNet. Although basic DDoS protection is automatically enabled as part of the Azure platform, DDoS Protection Standard provides mitigation capabilities that are tuned specifically to Azure Virtual Network resources.

## Deploy the solution

A deployment for this architecture is available on [GitHub](#). It deploys the following:

- A virtual network with a single subnet named **web** used to host the VM.
- An NSG with two incoming rules to allow RDP and HTTP traffic to the VM.
- A VM running the latest version of Windows Server 2016 Datacenter Edition.
- A sample custom script extension that formats the two data disks, and a PowerShell DSC script that deploys Internet Information Services (IIS).

#### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

#### Deploy the solution using azbb

To deploy this reference architecture, follow these steps:

1. Navigate to the `virtual-machines\single-vm\parameters\windows` folder for the repository you downloaded in the prerequisites step above.
2. Open the `single-vm-v2.json` file and enter a username and password between the quotes, then save the file.

```
"adminUsername": "",  
"adminPassword": "",
```

3. Run `azbb` to deploy the sample VM as shown below.

```
azbb -s <subscription_id> -g <resource_group_name> -l <location> -p single-vm-v2.json --deploy
```

To verify the deployment, run the following Azure CLI command to find the public IP address of the VM:

```
az vm show -n ra-single-windows-vm1 -g <resource-group-name> -d -o table
```

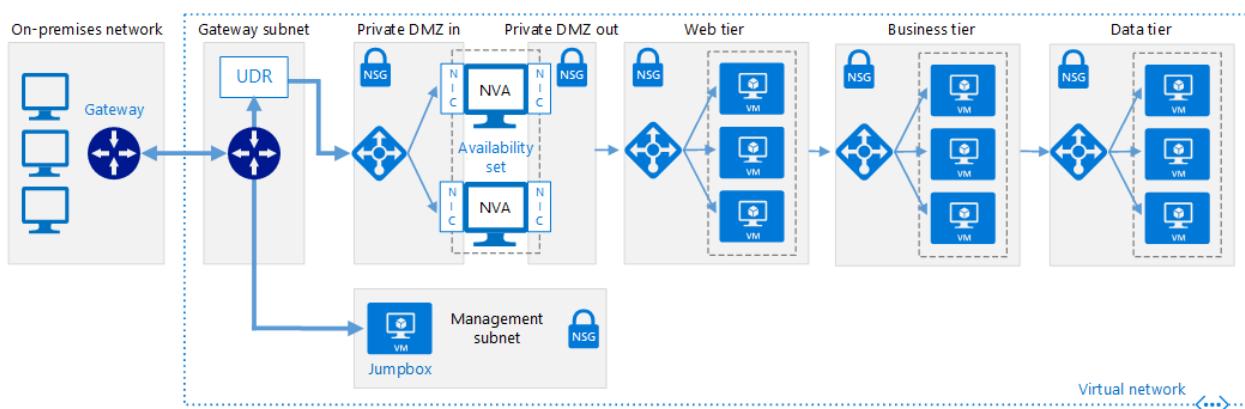
If you navigate to this address in a web browser, you should see the default IIS homepage.

For information about customizing this deployment, visit our [GitHub repository](#).

# DMZ between Azure and your on-premises datacenter

9/4/2018 • 11 minutes to read • [Edit Online](#)

This reference architecture shows a secure hybrid network that extends an on-premises network to Azure. The architecture implements a DMZ, also called a *perimeter network*, between the on-premises network and an Azure virtual network (VNet). The DMZ includes network virtual appliances (NVAs) that implement security functionality such as firewalls and packet inspection. All outgoing traffic from the VNet is force-tunneled to the Internet through the on-premises network, so that it can be audited. [Deploy this solution](#).



[Download a Visio file of this architecture.](#)

This architecture requires a connection to your on-premises datacenter, using either a [VPN gateway](#) or an [ExpressRoute](#) connection. Typical uses for this architecture include:

- Hybrid applications where workloads run partly on-premises and partly in Azure.
- Infrastructure that requires granular control over traffic entering an Azure VNet from an on-premises datacenter.
- Applications that must audit outgoing traffic. This is often a regulatory requirement of many commercial systems and can help to prevent public disclosure of private information.

## Architecture

The architecture consists of the following components.

- **On-premises network.** A private local-area network implemented in an organization.
- **Azure virtual network (VNet).** The VNet hosts the application and other resources running in Azure.
- **Gateway.** The gateway provides connectivity between the routers in the on-premises network and the VNet.
- **Network virtual appliance (NVA).** NVA is a generic term that describes a VM performing tasks such as allowing or denying access as a firewall, optimizing wide area network (WAN) operations (including network compression), custom routing, or other network functionality.
- **Web tier, business tier, and data tier subnets.** Subnets hosting the VMs and services that implement an example 3-tier application running in the cloud. See [Running Windows VMs for an N-tier architecture on Azure](#) for more information.
- **User defined routes (UDR).** [User defined routes](#) define the flow of IP traffic within Azure VNets.

#### **NOTE**

Depending on the requirements of your VPN connection, you can configure Border Gateway Protocol (BGP) routes instead of using UDRs to implement the forwarding rules that direct traffic back through the on-premises network.

- **Management subnet.** This subnet contains VMs that implement management and monitoring capabilities for the components running in the VNet.

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### Access control recommendations

Use [Role-Based Access Control](#) (RBAC) to manage the resources in your application. Consider creating the following [custom roles](#):

- A DevOps role with permissions to administer the infrastructure for the application, deploy the application components, and monitor and restart VMs.
- A centralized IT administrator role to manage and monitor network resources.
- A security IT administrator role to manage secure network resources such as the NVAs.

The DevOps and IT administrator roles should not have access to the NVA resources. This should be restricted to the security IT administrator role.

### Resource group recommendations

Azure resources such as VMs, VNets, and load balancers can be easily managed by grouping them together into resource groups. Assign RBAC roles to each resource group to restrict access.

We recommend creating the following resource groups:

- A resource group containing the VNet (excluding the VMs), NSGs, and the gateway resources for connecting to the on-premises network. Assign the centralized IT administrator role to this resource group.
- A resource group containing the VMs for the NVAs (including the load balancer), the jumpbox and other management VMs, and the UDR for the gateway subnet that forces all traffic through the NVAs. Assign the security IT administrator role to this resource group.
- Separate resource groups for each application tier that contain the load balancer and VMs. Note that this resource group shouldn't include the subnets for each tier. Assign the DevOps role to this resource group.

### Virtual network gateway recommendations

On-premises traffic passes to the VNet through a virtual network gateway. We recommend an [Azure VPN gateway](#) or an [Azure ExpressRoute gateway](#).

### NVA recommendations

NVAs provide different services for managing and monitoring network traffic. The [Azure Marketplace](#) offers several third-party vendor NVAs that you can use. If none of these third-party NVAs meet your requirements, you can create a custom NVA using VMs.

For example, the solution deployment for this reference architecture implements an NVA with the following functionality on a VM:

- Traffic is routed using [IP forwarding](#) on the NVA network interfaces (NICs).
- Traffic is permitted to pass through the NVA only if it is appropriate to do so. Each NVA VM in the reference architecture is a simple Linux router. Inbound traffic arrives on network interface `eth0`, and outbound traffic

matches rules defined by custom scripts dispatched through network interface *eth1*.

- The NVAs can only be configured from the management subnet.
- Traffic routed to the management subnet does not pass through the NVAs. Otherwise, if the NVAs fail, there would be no route to the management subnet to fix them.
- The VMs for the NVA are placed in an [availability set](#) behind a load balancer. The UDR in the gateway subnet directs NVA requests to the load balancer.

Include a layer-7 NVA to terminate application connections at the NVA level and maintain affinity with the backend tiers. This guarantees symmetric connectivity, in which response traffic from the backend tiers returns through the NVA.

Another option to consider is connecting multiple NVAs in series, with each NVA performing a specialized security task. This allows each security function to be managed on a per-NVA basis. For example, an NVA implementing a firewall could be placed in series with an NVA running identity services. The tradeoff for ease of management is the addition of extra network hops that may increase latency, so ensure that this doesn't affect your application's performance.

### NSG recommendations

The VPN gateway exposes a public IP address for the connection to the on-premises network. We recommend creating a network security group (NSG) for the inbound NVA subnet, with rules to block all traffic not originating from the on-premises network.

We also recommend NSGs for each subnet to provide a second level of protection against inbound traffic bypassing an incorrectly configured or disabled NVA. For example, the web tier subnet in the reference architecture implements an NSG with a rule to ignore all requests other than those received from the on-premises network (192.168.0.0/16) or the VNet, and another rule that ignores all requests not made on port 80.

### Internet access recommendations

**Force-tunnel** all outbound Internet traffic through your on-premises network using the site-to-site VPN tunnel, and route to the Internet using network address translation (NAT). This prevents accidental leakage of any confidential information stored in your data tier and allows inspection and auditing of all outgoing traffic.

#### NOTE

Don't completely block Internet traffic from the application tiers, as this will prevent these tiers from using Azure PaaS services that rely on public IP addresses, such as VM diagnostics logging, downloading of VM extensions, and other functionality. Azure diagnostics also requires that components can read and write to an Azure Storage account.

Verify that outbound internet traffic is force-tunneled correctly. If you're using a VPN connection with the [routing and remote access service](#) on an on-premises server, use a tool such as [WireShark](#) or [Microsoft Message Analyzer](#).

### Management subnet recommendations

The management subnet contains a jumpbox that performs management and monitoring functionality. Restrict execution of all secure management tasks to the jumpbox.

Do not create a public IP address for the jumpbox. Instead, create one route to access the jumpbox through the incoming gateway. Create NSG rules so the management subnet only responds to requests from the allowed route.

## Scalability considerations

The reference architecture uses a load balancer to direct on-premises network traffic to a pool of NVA devices, which route the traffic. The NVAs are placed in an [availability set](#). This design allows you to monitor the

throughput of the NVAs over time and add NVA devices in response to increases in load.

The standard SKU VPN gateway supports sustained throughput of up to 100 Mbps. The High Performance SKU provides up to 200 Mbps. For higher bandwidths, consider upgrading to an ExpressRoute gateway. ExpressRoute provides up to 10 Gbps bandwidth with lower latency than a VPN connection.

For more information about the scalability of Azure gateways, see the scalability consideration section in [Implementing a hybrid network architecture with Azure and on-premises VPN](#) and [Implementing a hybrid network architecture with Azure ExpressRoute](#).

## Availability considerations

As mentioned, the reference architecture uses a pool of NVA devices behind a load balancer. The load balancer uses a health probe to monitor each NVA and will remove any unresponsive NVAs from the pool.

If you're using Azure ExpressRoute to provide connectivity between the VNet and on-premises network, [configure a VPN gateway to provide failover](#) if the ExpressRoute connection becomes unavailable.

For specific information on maintaining availability for VPN and ExpressRoute connections, see the availability considerations in [Implementing a hybrid network architecture with Azure and on-premises VPN](#) and [Implementing a hybrid network architecture with Azure ExpressRoute](#).

## Manageability considerations

All application and resource monitoring should be performed by the jumpbox in the management subnet.

Depending on your application requirements, you may need additional monitoring resources in the management subnet. If so, these resources should be accessed through the jumpbox.

If gateway connectivity from your on-premises network to Azure is down, you can still reach the jumpbox by deploying a public IP address, adding it to the jumpbox, and remoting in from the internet.

Each tier's subnet in the reference architecture is protected by NSG rules. You may need to create a rule to open port 3389 for remote desktop protocol (RDP) access on Windows VMs or port 22 for secure shell (SSH) access on Linux VMs. Other management and monitoring tools may require rules to open additional ports.

If you're using ExpressRoute to provide the connectivity between your on-premises datacenter and Azure, use the [Azure Connectivity Toolkit \(AzureCT\)](#) to monitor and troubleshoot connection issues.

You can find additional information specifically aimed at monitoring and managing VPN and ExpressRoute connections in the articles [Implementing a hybrid network architecture with Azure and on-premises VPN](#) and [Implementing a hybrid network architecture with Azure ExpressRoute](#).

## Security considerations

This reference architecture implements multiple levels of security.

### **Routing all on-premises user requests through the NVA**

The UDR in the gateway subnet blocks all user requests other than those received from on-premises. The UDR passes allowed requests to the NVAs in the private DMZ subnet, and these requests are passed on to the application if they are allowed by the NVA rules. You can add other routes to the UDR, but make sure they don't inadvertently bypass the NVAs or block administrative traffic intended for the management subnet.

The load balancer in front of the NVAs also acts as a security device by ignoring traffic on ports that are not open in the load balancing rules. The load balancers in the reference architecture only listen for HTTP requests on port 80 and HTTPS requests on port 443. Document any additional rules that you add to the load balancers, and monitor traffic to ensure there are no security issues.

## Using NSGs to block/pass traffic between application tiers

Traffic between tiers is restricted by using NSGs. The business tier blocks all traffic that doesn't originate in the web tier, and the data tier blocks all traffic that doesn't originate in the business tier. If you have a requirement to expand the NSG rules to allow broader access to these tiers, weigh these requirements against the security risks. Each new inbound pathway represents an opportunity for accidental or purposeful data leakage or application damage.

### DevOps access

Use [RBAC](#) to restrict the operations that DevOps can perform on each tier. When granting permissions, use the [principle of least privilege](#). Log all administrative operations and perform regular audits to ensure any configuration changes were planned.

## Deploy the solution

A deployment for a reference architecture that implements these recommendations is available on [GitHub](#).

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Deploy resources

1. Navigate to the `/dmz/secure-vnet-hybrid` folder of the reference architectures GitHub repository.
2. Run the following command:

```
azbb -s <subscription_id> -g <resource_group_name> -l <region> -p onprem.json --deploy
```

3. Run the following command:

```
azbb -s <subscription_id> -g <resource_group_name> -l <region> -p secure-vnet-hybrid.json --deploy
```

### Connect the on-premises and Azure gateways

In this step, you will connect the two local network gateways.

1. In the Azure Portal, navigate to the resource group that you created.
2. Find the resource named `ra-vpn-vgw-pip` and copy the IP address shown in the **Overview** blade.
3. Find the resource named `onprem-vpn-1gw`.
4. Click the **Configuration** blade. Under **IP address**, paste in the IP address from step 2.

The screenshot shows the Azure portal interface for managing a Local network gateway. The left sidebar lists navigation options: Overview, Activity log, Access control (IAM), Tags, SETTINGS (Configuration selected), and Connections. The main content area displays the Configuration blade for the resource 'onprem-vpn-lgw'. It includes sections for IP address (set to 10.0.0.0/16), Address space, and BGP settings. A red box highlights the 'IP address' input field.

5. Click **Save** and wait for the operation to complete. It can take about 5 minutes.
6. Find the resource named `onprem-vpn-gateway1-pip`. Copy the IP address shown in the **Overview** blade.
7. Find the resource named `ra-vpn-1gw`.
8. Click the **Configuration** blade. Under **IP address**, paste in the IP address from step 6.
9. Click **Save** and wait for the operation to complete.
10. To verify the connection, go to the **Connections** blade for each gateway. The status should be **Connected**.

#### Verify that network traffic reaches the web tier

1. In the Azure Portal, navigate to the resource group that you created.
2. Find the resource named `int-dmz-1b`, which is the load balancer in front of the private DMZ. Copy the private IP address from the **Overview** blade.
3. Find the VM named `jb-vm1`. Click **Connect** and use Remote Desktop to connect to the VM. The user name and password are specified in the `onprem.json` file.
4. From the Remote Desktop Session, open a web browser and navigate to the IP address from step 2. You should see the default Apache2 server home page.

## Next steps

- Learn how to implement a [DMZ between Azure and the Internet](#).
- Learn how to implement a [highly available hybrid network architecture](#).
- For more information about managing network security with Azure, see [Microsoft cloud services and network security](#).
- For detailed information about protecting resources in Azure, see [Getting started with Microsoft Azure security](#).
- For additional details on addressing security concerns across an Azure gateway connection, see [Implementing a hybrid network architecture with Azure and on-premises VPN](#) and [Implementing a hybrid network](#)

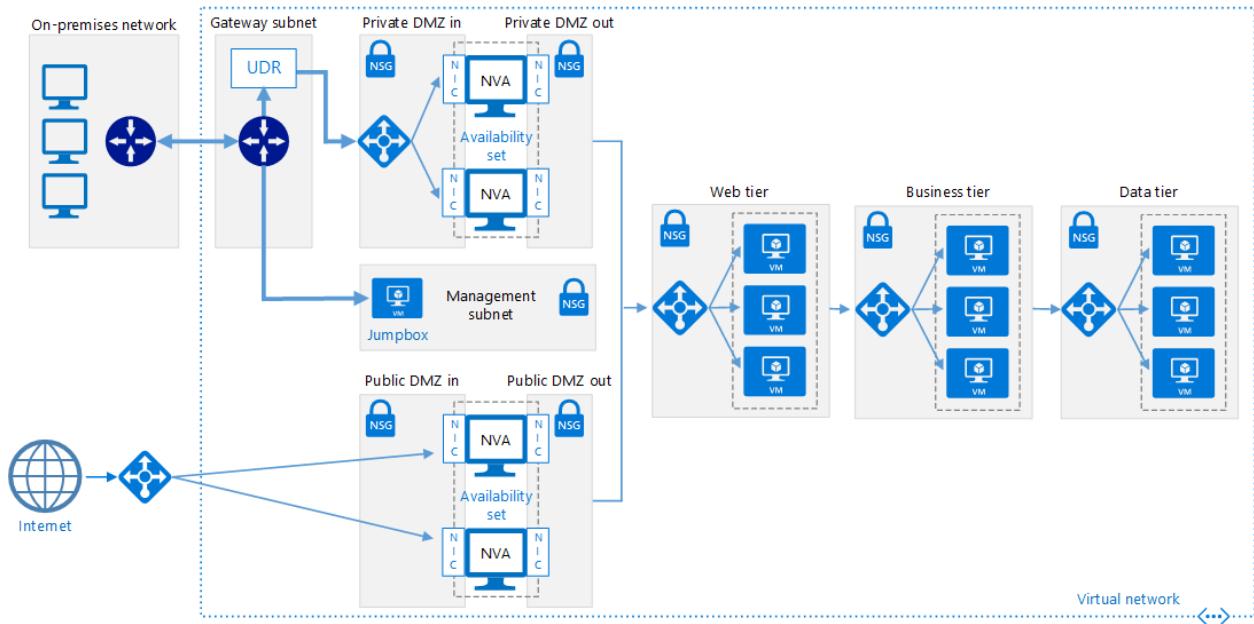
architecture with Azure ExpressRoute.

- Troubleshoot network virtual appliance issues in Azure

# DMZ between Azure and the Internet

7/3/2018 • 5 minutes to read • [Edit Online](#)

This reference architecture shows a secure hybrid network that extends an on-premises network to Azure and also accepts Internet traffic. [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

This reference architecture extends the architecture described in [Implementing a DMZ between Azure and your on-premises datacenter](#). It adds a public DMZ that handles Internet traffic, in addition to the private DMZ that handles traffic from the on-premises network.

Typical uses for this architecture include:

- Hybrid applications where workloads run partly on-premises and partly in Azure.
- Azure infrastructure that routes incoming traffic from on-premises and the Internet.

## Architecture

The architecture consists of the following components.

- **Public IP address (PIP).** The IP address of the public endpoint. External users connected to the Internet can access the system through this address.
- **Network virtual appliance (NVA).** This architecture includes a separate pool of NVAs for traffic originating on the Internet.
- **Azure load balancer.** All incoming requests from the Internet pass through the load balancer and are distributed to the NVAs in the public DMZ.
- **Public DMZ inbound subnet.** This subnet accepts requests from the Azure load balancer. Incoming requests are passed to one of the NVAs in the public DMZ.
- **Public DMZ outbound subnet.** Requests that are approved by the NVA pass through this subnet to the internal load balancer for the web tier.

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### NVA recommendations

Use one set of NVAs for traffic originating on the Internet, and another for traffic originating on-premises. Using only one set of NVAs for both is a security risk, because it provides no security perimeter between the two sets of network traffic. Using separate NVAs reduces the complexity of checking security rules, and makes it clear which rules correspond to each incoming network request. One set of NVAs implements rules for Internet traffic only, while another set of NVAs implement rules for on-premises traffic only.

Include a layer-7 NVA to terminate application connections at the NVA level and maintain compatibility with the backend tiers. This guarantees symmetric connectivity where response traffic from the backend tiers returns through the NVA.

### Public load balancer recommendations

For scalability and availability, deploy the public DMZ NVAs in an [availability set](#) and use an [Internet facing load balancer](#) to distribute Internet requests across the NVAs in the availability set.

Configure the load balancer to accept requests only on the ports necessary for Internet traffic. For example, restrict inbound HTTP requests to port 80 and inbound HTTPS requests to port 443.

## Scalability considerations

Even if your architecture initially requires a single NVA in the public DMZ, we recommend putting a load balancer in front of the public DMZ from the beginning. That will make it easier to scale to multiple NVAs in the future, if needed.

## Availability considerations

The Internet facing load balancer requires each NVA in the public DMZ inbound subnet to implement a [health probe](#). A health probe that fails to respond on this endpoint is considered to be unavailable, and the load balancer will direct requests to other NVAs in the same availability set. Note that if all NVAs fail to respond, your application will fail, so it's important to have monitoring configured to alert DevOps when the number of healthy NVA instances falls below a defined threshold.

## Manageability considerations

All monitoring and management for the NVAs in the public DMZ should be performed by the jumpbox in the management subnet. As discussed in [Implementing a DMZ between Azure and your on-premises datacenter](#), define a single network route from the on-premises network through the gateway to the jumpbox, in order to restrict access.

If gateway connectivity from your on-premises network to Azure is down, you can still reach the jumpbox by deploying a public IP address, adding it to the jumpbox, and logging in from the Internet.

## Security considerations

This reference architecture implements multiple levels of security:

- The Internet facing load balancer directs requests to the NVAs in the inbound public DMZ subnet, and only on the ports necessary for the application.
- The NSG rules for the inbound and outbound public DMZ subnets prevent the NVAs from being compromised, by blocking requests that fall outside of the NSG rules.
- The NAT routing configuration for the NVAs directs incoming requests on port 80 and port 443 to the web tier load balancer, but ignores requests on all other ports.

You should log all incoming requests on all ports. Regularly audit the logs, paying attention to requests that fall outside of expected parameters, as these may indicate intrusion attempts.

## Deploy the solution

A deployment for a reference architecture that implements these recommendations is available on [GitHub](#).

### Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

### Deploy resources

1. Navigate to the `/dmz/secure-vnet-hybrid` folder of the reference architectures GitHub repository.
2. Run the following command:

```
azbb -s <subscription_id> -g <resource_group_name> -l <region> -p onprem.json --deploy
```

3. Run the following command:

```
azbb -s <subscription_id> -g <resource_group_name> -l <region> -p secure-vnet-hybrid.json --deploy
```

### Connect the on-premises and Azure gateways

In this step, you will connect the two local network gateways.

1. In the Azure Portal, navigate to the resource group that you created.
2. Find the resource named `ra-vpn-vgw-pip` and copy the IP address shown in the **Overview** blade.
3. Find the resource named `onprem-vpn-lgw`.
4. Click the **Configuration** blade. Under **IP address**, paste in the IP address from step 2.

The screenshot shows the Azure portal interface for a resource named "onprem-vpn-lgw". The left sidebar has a search bar and links for Overview, Activity log, Access control (IAM), Tags, and Configuration (which is selected and highlighted in blue). The main content area shows the "Configuration" blade with fields for "IP address" (containing "192.168.1.100") and "Address space" (containing "10.0.0.0/16"). There is also a link to "Add additional address range" and a checkbox for "Configure BGP settings".

5. Click **Save** and wait for the operation to complete. It can take about 5 minutes.
6. Find the resource named `onprem-vpn-gateway1-pip`. Copy the IP address shown in the **Overview** blade.
7. Find the resource named `ra-vpn-lgw`.
8. Click the **Configuration** blade. Under **IP address**, paste in the IP address from step 6.
9. Click **Save** and wait for the operation to complete.
10. To verify the connection, go to the **Connections** blade for each gateway. The status should be **Connected**.

#### Verify that network traffic reaches the web tier

1. In the Azure Portal, navigate to the resource group that you created.
2. Find the resource named `pub-dmz-1b`, which is the load balancer in front of the public DMZ.
3. Copy the public IP address from the **Overview** blade and open this address in a web browser. You should see the default Apache2 server home page.
4. Find the resource named `int-dmz-1b`, which is the load balancer in front of the private DMZ. Copy the private IP address from the **Overview** blade.
5. Find the VM named `jb-vm1`. Click **Connect** and use Remote Desktop to connect to the VM. The user name and password are specified in the `onprem.json` file.
6. From the Remote Desktop Session, open a web browser and navigate to the IP address from step 4. You should see the default Apache2 server home page.

# Deploy highly available network virtual appliances

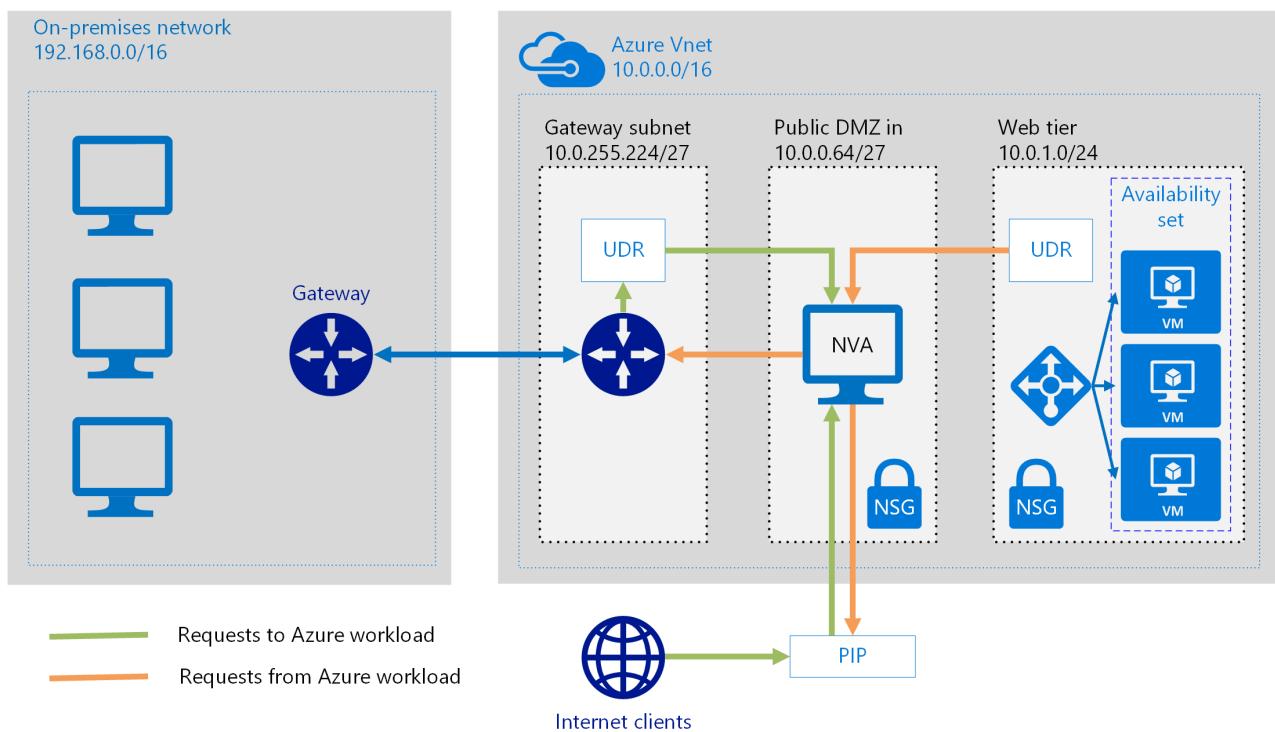
9/4/2018 • 6 minutes to read • [Edit Online](#)

This article shows how to deploy a set of network virtual appliances (NVAs) for high availability in Azure. An NVA is typically used to control the flow of network traffic from a perimeter network, also known as a DMZ, to other networks or subnets. To learn about implementing a DMZ in Azure, see [Microsoft cloud services and network security](#). The article includes example architectures for ingress only, egress only, and both ingress and egress.

**Prerequisites:** This article assumes a basic understanding of Azure networking, [Azure load balancers](#), and [user-defined routes](#) (UDRs).

## Architecture Diagrams

An NVA can be deployed to a DMZ in many different architectures. For example, the following figure illustrates the use of a [single NVA](#) for ingress.



In this architecture, the NVA provides a secure network boundary by checking all inbound and outbound network traffic and passing only the traffic that meets network security rules. However, the fact that all network traffic must pass through the NVA means that the NVA is a single point of failure in the network. If the NVA fails, there is no other path for network traffic and all the back-end subnets are unavailable.

To make an NVA highly available, deploy more than one NVA into an availability set.

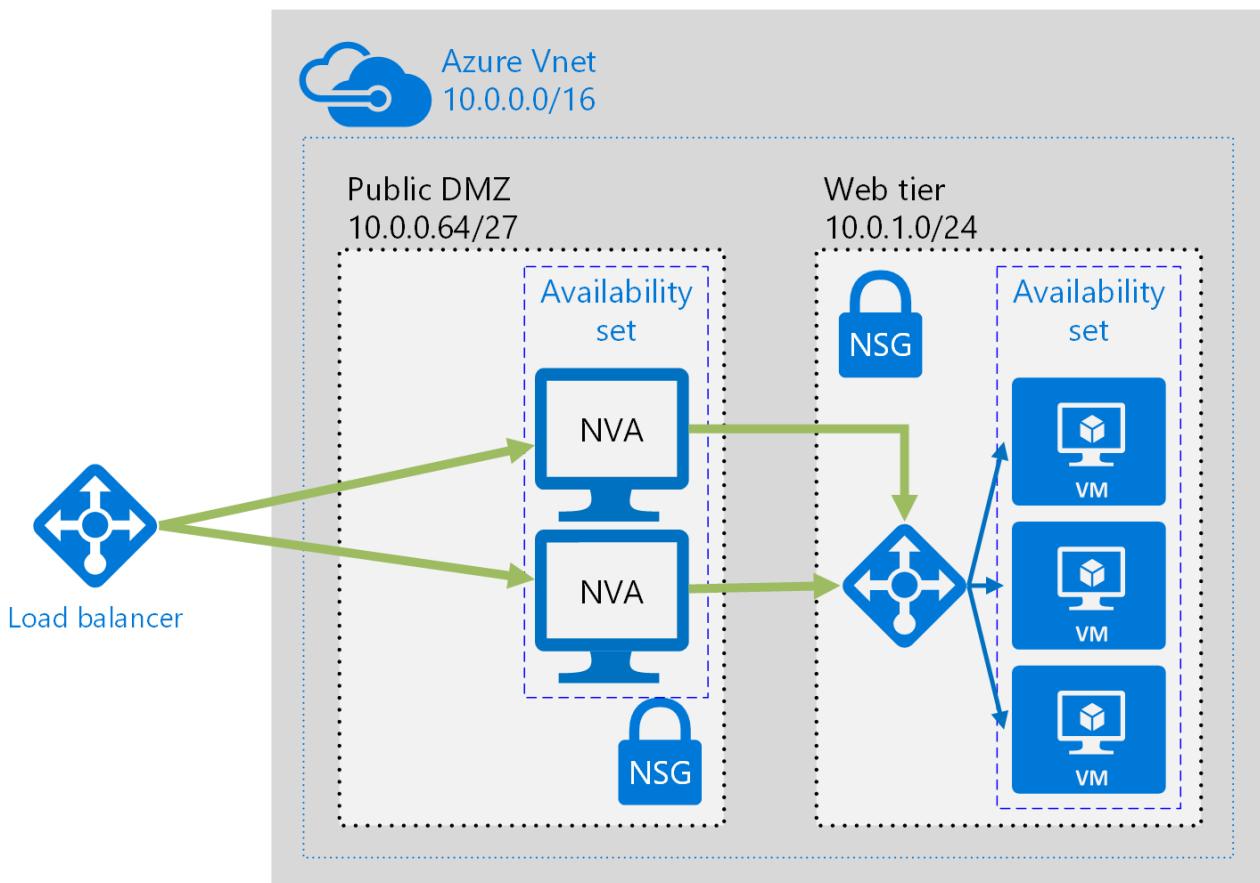
The following architectures describe the resources and configuration necessary for highly available NVAs:

| SOLUTION | BENEFITS | CONSIDERATIONS |
|----------|----------|----------------|
|----------|----------|----------------|

| SOLUTION                         | BENEFITS                                                                              | CONSIDERATIONS                                                                                                                                                                                               |
|----------------------------------|---------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ingress with layer 7 NVAs        | All NVA nodes are active                                                              | Requires an NVA that can terminate connections and use SNAT<br>Requires a separate set of NVAs for traffic coming from the Internet and from Azure<br>Can only be used for traffic originating outside Azure |
| Egress with layer 7 NVAs         | All NVA nodes are active                                                              | Requires an NVA that can terminate connections and implements source network address translation (SNAT)                                                                                                      |
| Ingress-Egress with layer 7 NVAs | All nodes are active<br>Able to handle traffic originated in Azure                    | Requires an NVA that can terminate connections and use SNAT<br>Requires a separate set of NVAs for traffic coming from the Internet and from Azure                                                           |
| PIP-UDR switch                   | Single set of NVAs for all traffic<br>Can handle all traffic (no limit on port rules) | Active-passive<br>Requires a failover process                                                                                                                                                                |

## Ingress with layer 7 NVAs

The following figure shows a high availability architecture that implements an ingress DMZ behind an internet-facing load balancer. This architecture is designed to provide connectivity to Azure workloads for layer 7 traffic, such as HTTP or HTTPS:



The benefit of this architecture is that all NVAs are active, and if one fails the load balancer directs network traffic to the other NVA. Both NVAs route traffic to the internal load balancer so as long as one NVA is active, traffic

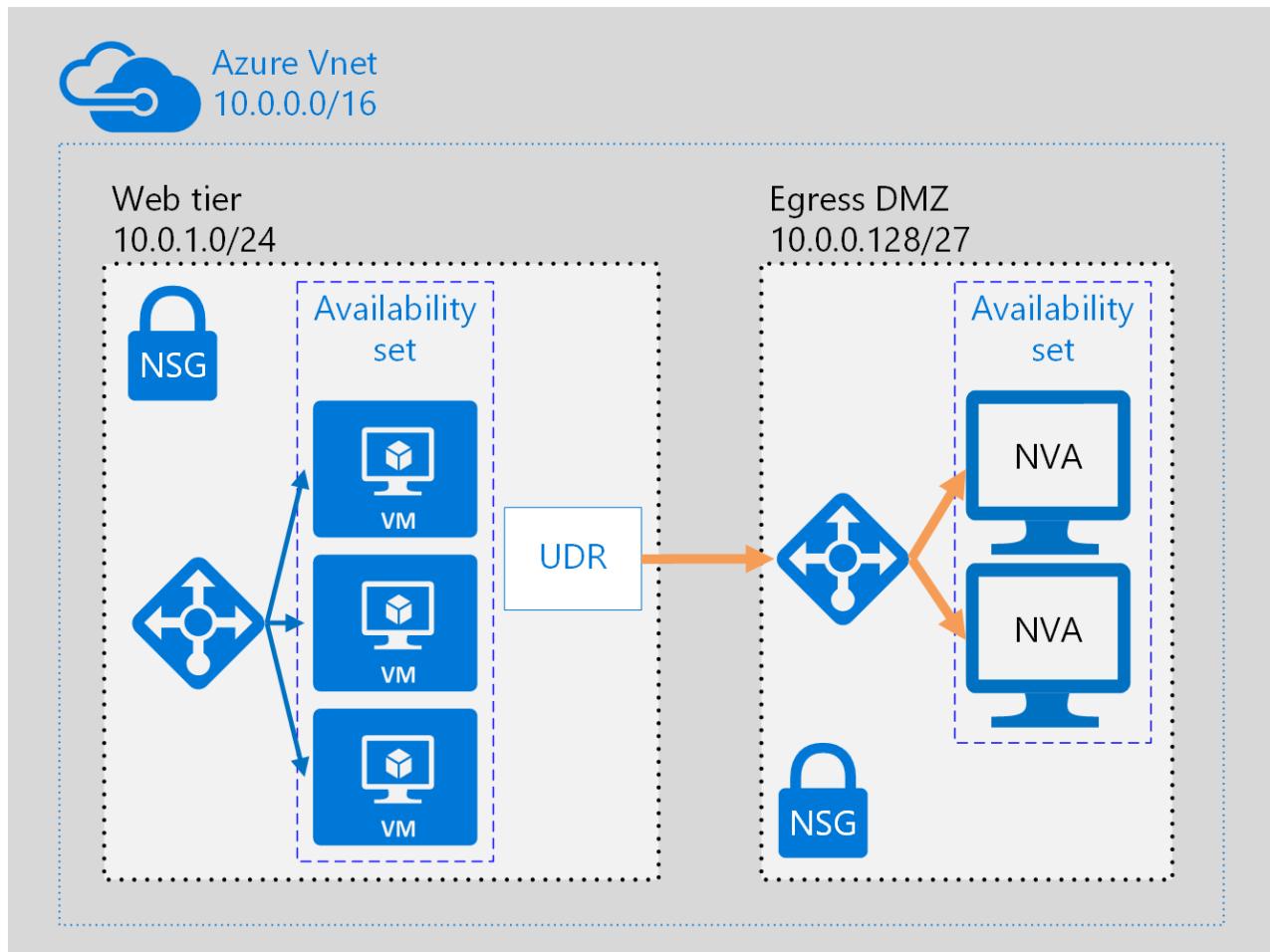
continues to flow. The NVAs are required to terminate SSL traffic intended for the web tier VMs. These NVAs cannot be extended to handle on-premises traffic because on-premises traffic requires another dedicated set of NVAs with their own network routes.

**NOTE**

This architecture is used in the [DMZ between Azure and your on-premises datacenter](#) reference architecture and the [DMZ between Azure and the Internet](#) reference architecture. Each of these reference architectures includes a deployment solution that you can use. Follow the links for more information.

## Egress with layer 7 NVAs

The previous architecture can be expanded to provide an egress DMZ for requests originating in the Azure workload. The following architecture is designed to provide high availability of the NVAs in the DMZ for layer 7 traffic, such as HTTP or HTTPS:



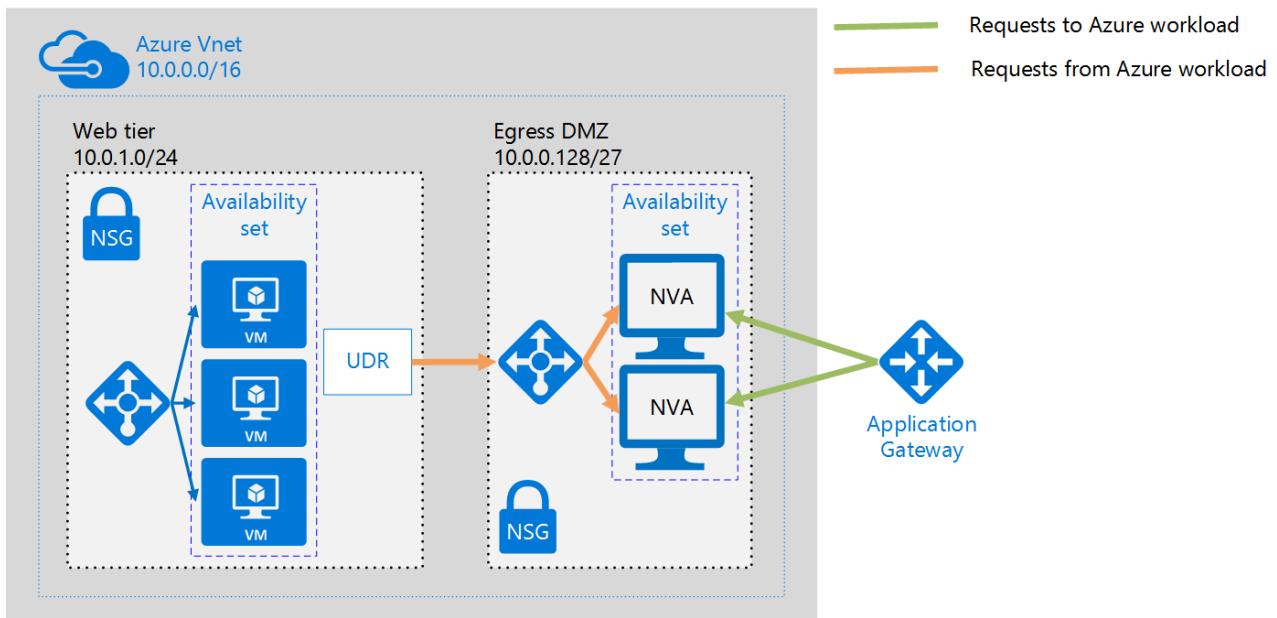
In this architecture, all traffic originating in Azure is routed to an internal load balancer. The load balancer distributes outgoing requests between a set of NVAs. These NVAs direct traffic to the Internet using their individual public IP addresses.

**NOTE**

This architecture is used in the [DMZ between Azure and your on-premises datacenter](#) reference architecture and the [DMZ between Azure and the Internet](#) reference architecture. Each of these reference architectures includes a deployment solution that you can use. Follow the links for more information.

## Ingress-egress with layer 7 NVAs

In the two previous architectures, there was a separate DMZ for ingress and egress. The following architecture demonstrates how to create a DMZ that can be used for both ingress and egress for layer 7 traffic, such as HTTP or HTTPS:



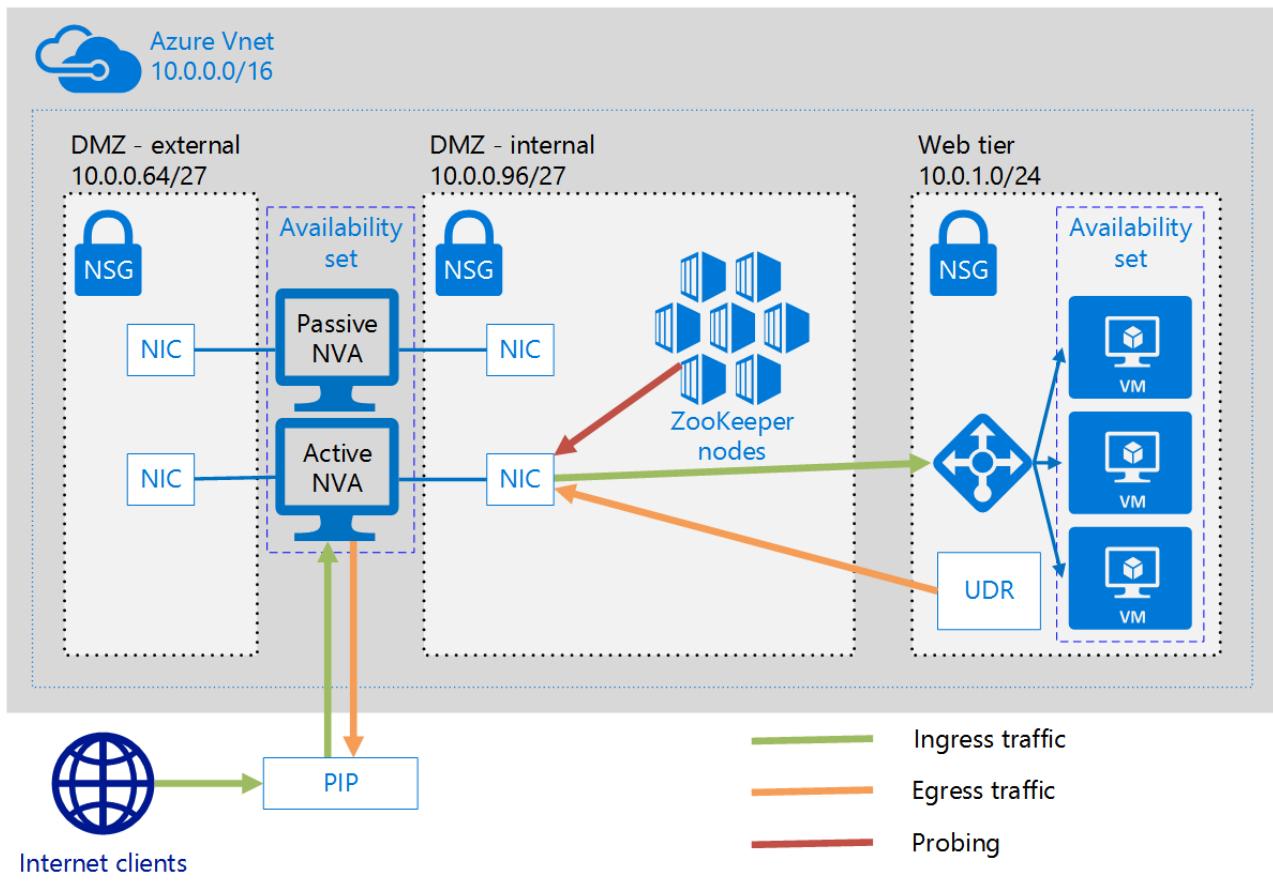
In this architecture, the NVAs process incoming requests from the application gateway. The NVAs also process outgoing requests from the workload VMs in the back-end pool of the load balancer. Because incoming traffic is routed with an application gateway and outgoing traffic is routed with a load balancer, the NVAs are responsible for maintaining session affinity. That is, the application gateway maintains a mapping of inbound and outbound requests so it can forward the correct response to the original requestor. However, the internal load balancer does not have access to the application gateway mappings, and uses its own logic to send responses to the NVAs. It's possible the load balancer could send a response to an NVA that did not initially receive the request from the application gateway. In this case, the NVAs must communicate and transfer the response between them so the correct NVA can forward the response to the application gateway.

#### NOTE

You can also solve the asymmetric routing issue by ensuring the NVAs perform inbound source network address translation (SNAT). This would replace the original source IP of the requestor to one of the IP addresses of the NVA used on the inbound flow. This ensures that you can use multiple NVAs at a time, while preserving the route symmetry.

## PIP-UDR switch with layer 4 NVAs

The following architecture demonstrates an architecture with one active and one passive NVA. This architecture handles both ingress and egress for layer 4 traffic:



This architecture is similar to the first architecture discussed in this article. That architecture included a single NVA accepting and filtering incoming layer 4 requests. This architecture adds a second passive NVA to provide high availability. If the active NVA fails, the passive NVA is made active and the UDR and PIP are changed to point to the NICs on the now active NVA. These changes to the UDR and PIP can either be done manually or using an automated process. The automated process is typically daemon or other monitoring service running in Azure. It queries a health probe on the active NVA and performs the UDR and PIP switch when it detects a failure of the NVA.

The preceding figure shows an example **ZooKeeper** cluster providing a high availability daemon. Within the ZooKeeper cluster, a quorum of nodes elects a leader. If the leader fails, the remaining nodes hold an election to elect a new leader. For this architecture, the leader node executes the daemon that queries the health endpoint on the NVA. If the NVA fails to respond to the health probe, the daemon activates the passive NVA. The daemon then calls the Azure REST API to remove the PIP from the failed NVA and attaches it to newly activated NVA. The daemon then modifies the UDR to point to the newly activated NVA's internal IP address.

#### NOTE

Do not include the ZooKeeper nodes in a subnet that is only accessible using a route that includes the NVA. Otherwise, the ZooKeeper nodes are inaccessible if the NVA fails. Should the daemon fail for any reason, you won't be able to access any of the ZooKeeper nodes to diagnose the problem.

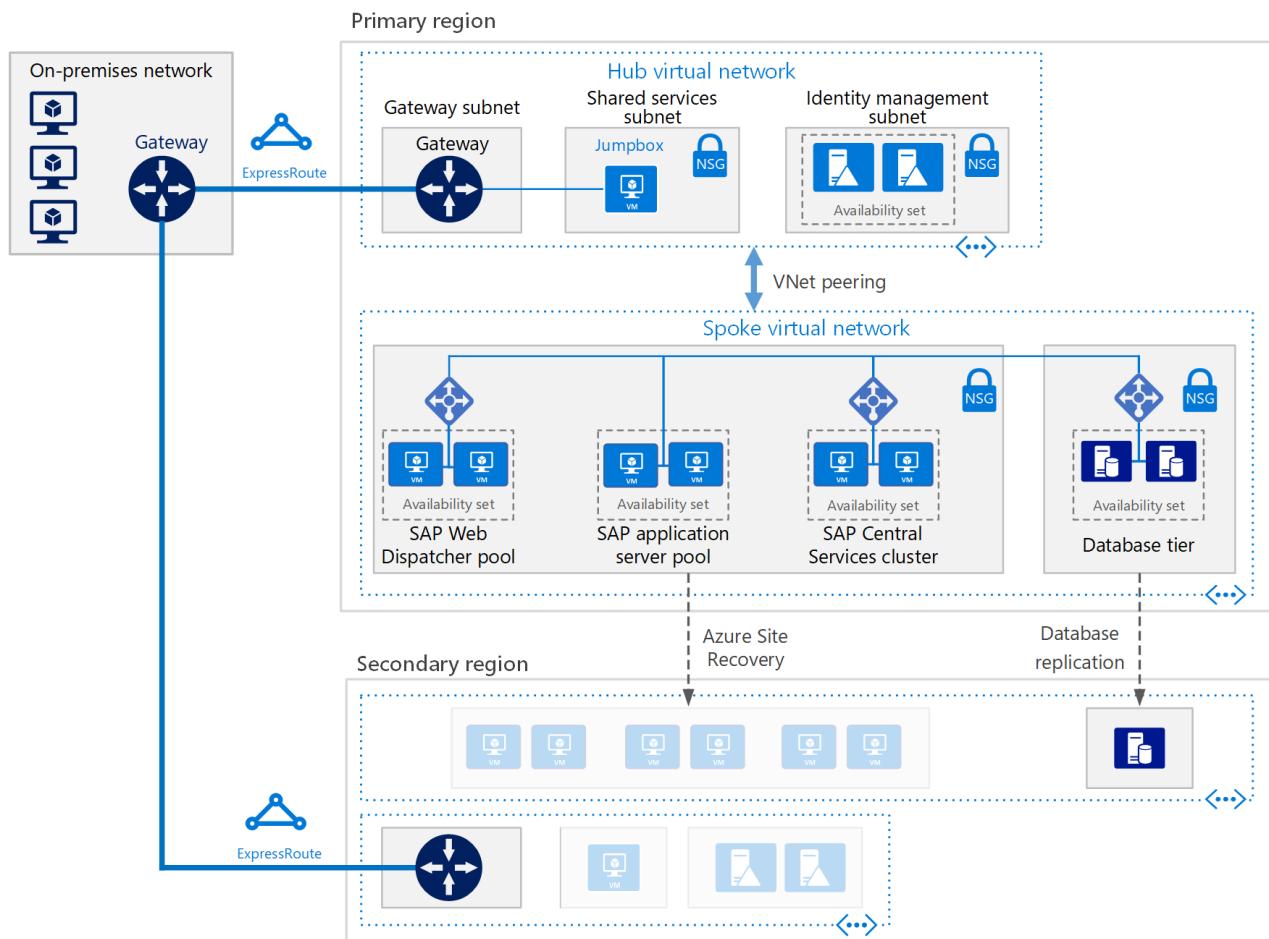
## Next steps

- Learn how to [implement a DMZ between Azure and your on-premises datacenter](#) using layer-7 NVAs.
- Learn how to [implement a DMZ between Azure and the Internet](#) using layer-7 NVAs.
- [Troubleshoot network virtual appliance issues in Azure](#)

# Deploy SAP NetWeaver (Windows) for AnyDB on Azure Virtual Machines

9/28/2018 • 13 minutes to read • [Edit Online](#)

This reference architecture shows a set of proven practices for running SAP NetWeaver in a Windows environment on Azure with high availability. The database is AnyDB, the SAP term for any supported DBMS besides SAP HANA. This architecture is deployed with specific virtual machine (VM) sizes that can be changed to accommodate your organization's needs.



[Download a Visio file of this architecture.](#)

## NOTE

Deploying this reference architecture requires appropriate licensing of SAP products and other non-Microsoft technologies.

## Architecture

The architecture consists of the following infrastructure and key software components.

**Virtual network.** The Azure Virtual Network service securely connects Azure resources to each other. In this architecture, the virtual network connects to an on-premises environment through a VPN gateway deployed in the hub of a **hub-spoke**. The spoke is the virtual network used for the SAP applications and database tier.

**Subnets.** The virtual network is subdivided into separate subnets for each tier: application (SAP NetWeaver),

database, shared services (the jumpbox), and Active Directory.

**Virtual machines.** This architecture uses virtual machines for the application tier and database tier, grouped as follows:

- **SAP NetWeaver.** The application tier uses Windows virtual machines and runs SAP Central Services and SAP application servers. The VMs that run Central Services are configured as a Windows Server Failover Cluster for high availability, supported by SIOS DataKeeper Cluster Edition.
- **AnyDB.** The database tier runs AnyDB as the source database, such as Microsoft SQL Server, Oracle, or IBM DB2.
- **Jumpbox.** Also called a bastion host. This is a secure virtual machine on the network that administrators use to connect to the other virtual machines.
- **Windows Server Active Directory domain controllers.** The domain controllers are used on all VMs and users in the domain.

**Load balancers.** [Azure Load Balancer](#) instances are used to distribute traffic to virtual machines in the application tier subnet. At the data tier, high availability may be achieved using built-in SAP load balancers, Azure Load Balancer, or other mechanisms, depending on the DBMS. For more information, see [Azure Virtual Machines DBMS deployment for SAP NetWeaver](#).

**Availability sets.** Virtual machines for the SAP Web Dispatcher, SAP application server, and (A)SCS, roles are grouped into separate [availability sets](#), and at least two virtual machines are provisioned per role. This makes the virtual machines eligible for a higher [service level agreement](#) (SLA).

**NICs.** [Network interface cards](#) (NICs) enable all communication of virtual machines on a virtual network.

**Network security groups.** To restrict incoming, outgoing, and intra-subnet traffic in the virtual network, you can create [network security groups](#) (NSGs).

**Gateway.** A gateway extends your on-premises network to the Azure virtual network. [ExpressRoute](#) is the recommended Azure service for creating private connections that do not go over the public Internet, but a [Site-to-Site](#) connection can also be used.

**Azure Storage.** To provide persistent storage of a virtual machine's virtual hard disk (VHD), [Azure Storage](#) is required. It is also used by [Cloud Witness](#) to implement a failover cluster operation.

## Recommendations

Your requirements might differ from the architecture described here. Use these recommendations as a starting point.

### SAP Web Dispatcher pool

The Web Dispatcher component is used as a load balancer for SAP traffic among the SAP application servers. To achieve high availability for the Web Dispatcher component, Azure Load Balancer is used to implement the parallel Web Dispatcher setup. Web Dispatcher uses in a round-robin configuration for HTTP(S) traffic distribution among the available Web Dispatchers in the balancers pool.

For details about running SAP NetWeaver in Azure VMs, see [Azure Virtual Machines planning and implementation for SAP NetWeaver](#).

### Application servers pool

To manage logon groups for ABAP application servers, the SMLG transaction is used. It uses the load balancing function within the message server of the Central Services to distribute workload among SAP application servers pool for SAPGUIs and RFC traffic. The application server connection to the highly available Central Services is through the cluster virtual network name.

### SAP Central Services cluster

This reference architecture runs Central Services on VMs in the application tier. The Central Services is a potential single point of failure (SPOF) when deployed to a single VM—typical deployment when high availability is not a requirement. To implement a high availability solution, either a shared disk cluster or a file share cluster can be used.

To configure VMs for a shared disk cluster, use [Windows Server Failover Cluster](#). [Cloud Witness](#) is recommended as a quorum witness. To support the failover cluster environment, [SIOS DataKeeper Cluster Edition](#) performs the cluster shared volume function by replicating independent disks owned by the cluster nodes. Azure does not natively support shared disks and therefore requires solutions provided by SIOS.

For details, see "3. Important Update for SAP Customers Running ASCS on SIOS on Azure" at [Running SAP applications on the Microsoft platform](#).

Another way to handle clustering is to implement a file share cluster using Windows Server Failover Cluster. [SAP](#) recently modified the Central Services deployment pattern to access the /sapmnt global directories via a UNC path. This change [removes the requirement](#) for SIOS or other shared disk solutions on the Central Services VMs. It is still recommended to ensure that the /sapmnt UNC share is [highly available](#). This can be done on the Central Services instance by using Windows Server Failover Cluster with [Scale Out File Server](#) (SOFS) and the [Storage Spaces Direct](#) (S2D) feature in Windows Server 2016.

## Availability sets

Availability sets distribute servers to different physical infrastructure and update groups to improve service availability. Put virtual machines that perform the same role into an availability sets to help guard against downtime caused by Azure infrastructure maintenance and to meet [SLAs](#) (SLAs). Two or more virtual machines per availability set is recommended.

All virtual machines in a set must perform the same role. Do not mix servers of different roles in the same availability set. For example, don't place a Central Services node in the same availability set with the application server.

## NICs

Traditional on-premises SAP deployments implement multiple network interface cards (NICs) per machine to segregate administrative traffic from business traffic. On Azure, the virtual network is a software-defined network that sends all traffic through the same network fabric. Therefore, the use of multiple NICs is unnecessary. However, if your organization needs to segregate traffic, you can deploy multiple NICs per VM, connect each NIC to a different subnet, and then use NSGs to enforce different access control policies.

## Subnets and NSGs

This architecture subdivides the virtual network address space into subnets. This reference architecture focuses primarily on the application tier subnet. Each subnet can be associated with a NSG that defines the access policies for the subnet. Place application servers on a separate subnet so you can secure them more easily by managing the subnet security policies, not the individual servers.

When a NSG is associated with a subnet, it applies to all the servers within the subnet. For more information about using NSGs for fine-grained control over the servers in a subnet, see [Filter network traffic with network security groups](#).

## Load balancers

[SAP Web Dispatcher](#) handles load balancing of HTTP(S) traffic to a pool of SAP application servers.

For traffic from SAP GUI clients connecting a SAP server via DIAG protocol or Remote Function Calls (RFC), the Central Services message server balances the load through SAP application server [logon groups](#), so no additional load balancer is needed.

## Azure Storage

For all database server virtual machines, we recommend using Azure Premium Storage for consistent read/write

latency. For any single instance virtual machine using Premium Storage for all operating system disks and data disks, see [SLA for Virtual Machines](#). Also, for production SAP systems, we recommend using Premium [Azure Managed Disks](#) in all cases. For reliability, Managed Disks are used to manage the VHD files for the disks. Managed disks ensure that the disks for virtual machines within an availability set are isolated to avoid single points of failure.

For SAP application servers, including the Central Services virtual machines, you can use Azure Standard Storage to reduce cost, because application execution takes place in memory and disks are used for logging only. However, at this time, Standard Storage is only certified for unmanaged storage. Since application servers do not host any data, you can also use the smaller P4 and P6 Premium Storage disks to help minimize cost.

Azure Storage is also used by [Cloud Witness](#) to maintain quorum with a device in a remote Azure region away from the primary region where the cluster resides.

For the backup data store, we recommend using Azure [coolaccess tier](#) and [archive access tier storage](#). These storage tiers are cost-effective ways to store long-lived data that is infrequently accessed.

## Performance considerations

SAP application servers carry on constant communications with the database servers. For performance-critical applications running on any database platforms, including SAP HANA, consider enabling [Write Accelerator](#) to improve log write latency. To optimize inter-server communications, use the [Accelerated Network](#). Note that these accelerators are available only for certain VM series.

To achieve high IOPS and disk bandwidth throughput, the common practices in storage volume [performance optimization](#) apply to Azure storage layout. For example, combining multiple disks together to create a striped disk volume improves IO performance. Enabling the read cache on storage content that changes infrequently enhances the speed of data retrieval.

For SAP on SQL, the [Top 10 Key Considerations for Deploying SAP Applications on Azure](#) blog offers excellent advice on optimizing Azure storage for SAP workloads on SQL Server.

## Scalability considerations

At the SAP application layer, Azure offers a wide range of virtual machine sizes for scaling up and scaling out. For an inclusive list, see [SAP note 1928533 - SAP Applications on Azure: Supported Products and Azure VM Types](#). (SAP Service Marketplace account required for access). SAP application servers and the Central Services clusters can scale up/down or scale out by adding more instances. The AnyDB database can scale up/down but does not scale out. The SAP database container for AnyDB does not support sharding.

## Availability considerations

Resource redundancy is the general theme in highly available infrastructure solutions. For enterprises that have a less stringent SLA, single-instance Azure VMs offer an uptime SLA. For more information, see [Azure Service Level Agreement](#).

In this distributed installation of the SAP application, the base installation is replicated to achieve high availability. For each layer of the architecture, the high availability design varies.

### Application tier

High availability for SAP Web Dispatcher is achieved with redundant instances. See [SAP Web Dispatcher](#) in the SAP Documentation.

High availability of the Central Services is implemented with Windows Server Failover Cluster. When deployed on Azure, the cluster storage for the failover cluster can be configured using two approaches: either a clustered shared volume or a file share.

Since shared disks are not possible on Azure, SIOS Datakeeper is used to replicate the content of independent disks attached to the cluster nodes and to abstract the drives as a cluster shared volume for the cluster manager. For implementation details, see [Clustering SAP ASCS on Azure](#).

Another option is to use a file share served up by the [Scale Out Fileserver](#) (SOFS). SOFS offers resilient file shares you can use as a cluster shared volume for the Windows cluster. A SOFS cluster can be shared among multiple Central Services nodes. As of this writing, SOFS is used only for high availability design, because the SOFS cluster does not extend across regions to provide disaster recovery support.

High availability for the SAP application servers is achieved by load balancing traffic within a pool of application servers. See [SAP certifications and configurations running on Microsoft Azure](#).

### Database tier

This reference architecture assumes the source database is running on AnyDB—that is, a DBMS such as SQL Server, SAP ASE, IBM DB2, or Oracle. The database tier's native replication feature provides either manual or automatic failover between replicated nodes.

For implementation details about specific database systems, see [Azure Virtual Machines DBMS deployment for SAP NetWeaver](#).

## Disaster recovery considerations

For disaster recovery (DR), you must be able to fail over to a secondary region. Each tier uses a different strategy to provide disaster recovery (DR) protection.

- **Application servers tier.** SAP application servers do not contain business data. On Azure, a simple DR strategy is to create SAP application servers in the secondary region, then shut them down. Upon any configuration changes or kernel updates on the primary application server, the same changes must be copied to the virtual machines in the secondary region. For example, the kernel executables copied to the DR virtual machines. For automatic replication of application servers to a secondary region, [Azure Site Recovery](#) is the recommended solution.
- **Central Services.** This component of the SAP application stack also does not persist business data. You can build a VM in the disaster recovery region to run the Central Services role. The only content from the primary Central Services node to synchronize is the /sapmnt share content. Also, if configuration changes or kernel updates take place on the primary Central Services servers, they must be repeated on the VM in the disaster recovery region running Central Services. To synchronize the two servers, you can use either Azure Site Recovery to replicate the cluster nodes or simply use a regularly scheduled copy job to copy /sapmnt to the disaster recovery region. For details about this simple replication method's build, copy, and test failover process, download [SAP NetWeaver: Building a Hyper-V and Microsoft Azure-based Disaster Recovery Solution](#), and refer to "4.3. SAP SPOF layer (ASCS)."
- **Database tier.** DR is best implemented with the database's own integrated replication technology. In the case of SQL Server, for example, we recommend using AlwaysOn Availability Group to establish a replica in a remote region, replicating transactions asynchronously with manual failover. Asynchronous replication avoids an impact to the performance of interactive workloads at the primary site. Manual failover offers the opportunity for a person to evaluate the DR impact and decide if operating from the DR site is justified.

To use Azure Site Recovery to automatically build out a fully replicated production site of your original, you must run customized [deployment scripts](#). Site Recovery first deploys the VMs in availability sets, then runs scripts to add resources such as load balancers.

## Manageability considerations

Azure provides several functions for [monitoring and diagnostics](#) of the overall infrastructure. Also, enhanced monitoring of Azure virtual machines is handled by Azure Operations Management Suite (OMS).

To provide SAP-based monitoring of resources and service performance of the SAP infrastructure, the [Azure SAP Enhanced Monitoring](#) extension is used. This extension feeds Azure monitoring statistics into the SAP application for operating system monitoring and DBA Cockpit functions.

## Security considerations

SAP has its own Users Management Engine (UME) to control role-based access and authorization within the SAP application. For details, see [SAP NetWeaver Application Server for ABAP Security Guide] ([https://help.sap.com/doc/7b932ef4728810148a4b1a83b0e91070/1610\\_001/en-US/frameset.htm?4dde53b3e9142e51e1000000a42189c.html](https://help.sap.com/doc/7b932ef4728810148a4b1a83b0e91070/1610_001/en-US/frameset.htm?4dde53b3e9142e51e1000000a42189c.html)) and [SAP NetWeaver Application Server Java Security Guide](#).

For additional network security, consider implementing a [network DMZ](#), which uses a network virtual appliance to create a firewall in front of the subnet for Web Dispatcher.

For infrastructure security, data is encrypted in transit and at rest. The "Security considerations" section of the [SAP NetWeaver on Azure Virtual Machines \(VMs\) – Planning and Implementation Guide](#) begins to address network security. The guide also specifies the network ports you must open on the firewalls to allow application communication.

To encrypt Windows virtual machine disks, you can use [Azure Disk Encryption](#). It uses the BitLocker feature of Windows to provide volume encryption for the operating system and the data disks. The solution also works with Azure Key Vault to help you control and manage the disk-encryption keys and secrets in your key vault subscription. Data on the virtual machine disks are encrypted at rest in your Azure storage.

## Communities

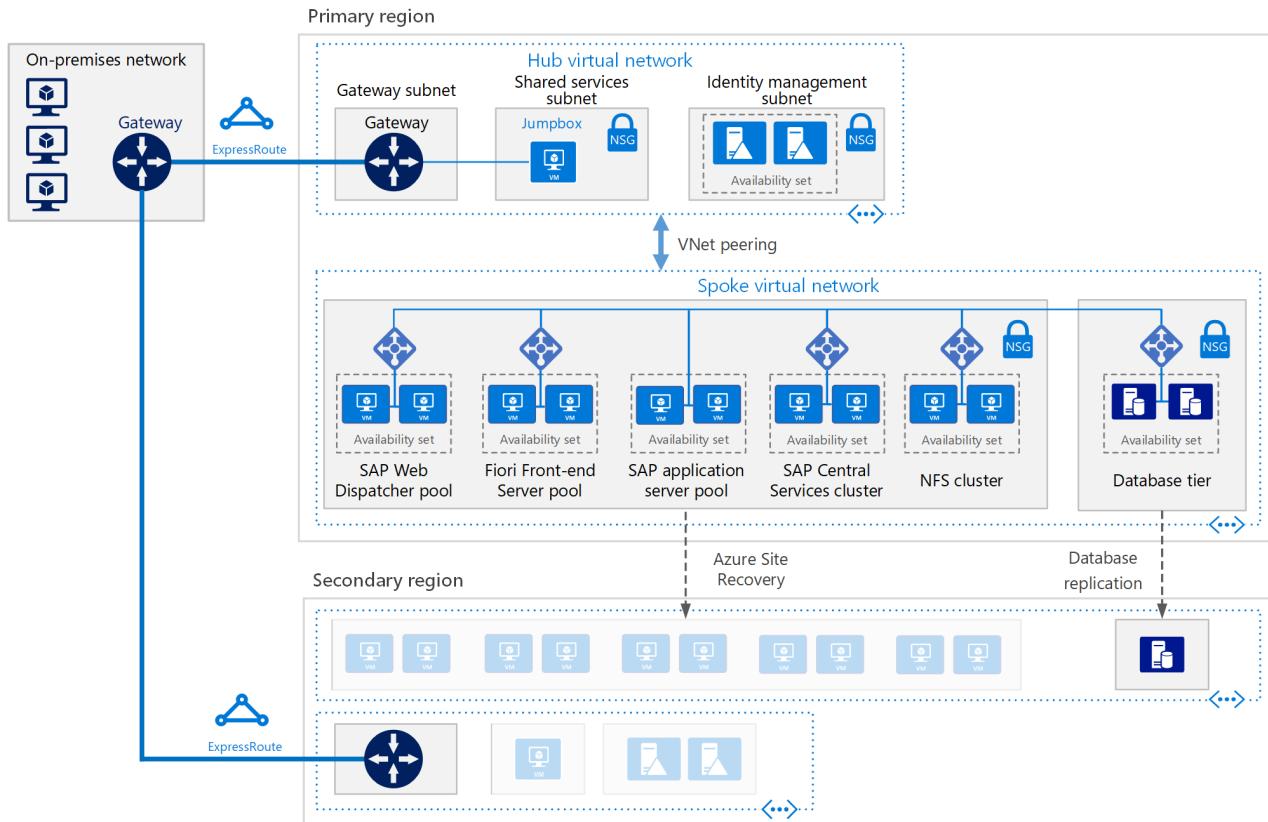
Communities can answer questions and help you set up a successful deployment. Consider the following:

- [Running SAP Applications on the Microsoft Platform Blog](#)
- [Azure Community Support](#)
- [SAP Community](#)
- [Stack Overflow](#)

# SAP S/4HANA for Linux Virtual Machines on Azure

9/28/2018 • 14 minutes to read • [Edit Online](#)

This reference architecture shows a set of proven practices for running S/4HANA in a high availability environment that supports disaster recovery on Azure. This architecture is deployed with specific virtual machine (VM) sizes that can be changed to accommodate your organization's needs.



Download a [Visio file](#) of this architecture.

## NOTE

Deploying this reference architecture requires appropriate licensing of SAP products and other non-Microsoft technologies.

## Architecture

This reference architecture describes a enterprise-grade, production-level system. To suit your business needs, this configuration can be reduced to a single virtual machine. However, the following components are required:

**Virtual network.** The [Azure Virtual Network](#) service securely connects Azure resources to each other. In this architecture, the virtual network connects to an on-premises environment through a gateway deployed in the hub of a [hub-spoke topology](#). The spoke is the virtual network used for the SAP applications.

**Subnets.** The virtual network is subdivided into separate [subnets](#) for each tier: gateway, application, database, and shared services .

**Virtual machines.** This architecture uses virtual machines running Linux for the application tier and database tier, grouped as follows:

- **Application tier.** Includes the Fiori Front-end Server pool, SAP Web Dispatcher pool, application server pool,

and SAP Central Services cluster. For high availability of Central Services on Azure Linux virtual machines, a highly available Network File System (NFS) service is required.

- **NFS cluster.** This architecture uses an [NFS](#) server running on a Linux cluster to store data shared between SAP systems. This centralized cluster can be shared across multiple SAP systems. For high availability of the NFS service, the appropriate High Availability Extension for the selected Linux distribution is used.
- **SAP HANA.** The database tier uses two or more Linux virtual machines in a cluster to achieve high availability. HANA System Replication (HSR) is used to replicate contents between primary and secondary HANA systems. Linux clustering is used to detect system failures and facilitate automatic failover. A storage-based or cloud-based fencing mechanism can be used to ensure the failed system is isolated or shut down to avoid the cluster split-brain condition.
- **Jumpbox.** Also called a bastion host. This is a secure virtual machine on the network that administrators use to connect to the other virtual machines. It can run Windows or Linux. Use a Windows jumpbox for web browsing convenience when using HANA Cockpit or HANA Studio management tools.

**Load balancers.** Both built-in SAP load balancers and [Azure Load Balancer](#) are used to achieve HA. Azure Load Balancer instances are used to distribute traffic to virtual machines in the application tier subnet.

**Availability sets.** Virtual machines for all pools and clusters (Web Dispatcher, SAP application servers, Central Services, NFS, and HANA) are grouped into separate [availability sets](#), and at least two virtual machines are provisioned per role. This makes the virtual machines eligible for a higher [service level agreement](#) (SLA).

**NICs.** [Network interface cards](#) (NICs) enable all communication of virtual machines on a virtual network.

**Network security groups.** To restrict incoming, outgoing, and intra-subnet traffic in the virtual network, [network security groups](#) (NSGs) are used.

**Gateway.** A gateway extends your on-premises network to the Azure virtual network. [ExpressRoute](#) is the recommended Azure service for creating private connections that do not go over the public Internet, but a [Site-to-Site](#) connection can also be used.

**Azure Storage.** To provide persistent storage of a virtual machine's virtual hard disk (VHD), [Azure Storage](#) is required.

## Recommendations

This architecture describes a small production-level enterprise deployment. Your deployment will differ based on your business requirements. Use these recommendations as a starting point.

### Virtual machines

In application server pools and clusters, adjust the number of virtual machines based on your requirements. The [Azure Virtual Machines planning and implementation guide](#) includes details about running SAP NetWeaver on virtual machines, but the information applies to SAP S/4HANA as well.

For details about SAP support for Azure virtual machine types and throughput metrics (SAPS), see [SAP Note 1928533](#).

### SAP Web Dispatcher pool

The Web Dispatcher component is used as a load balancer for SAP traffic among the SAP application servers. To achieve high availability for the Web Dispatcher component, Azure Load Balancer is used to implement the parallel Web Dispatcher setup in a round-robin configuration for HTTP(S) traffic distribution among the available Web Dispatchers in the balancers back-end pool.

### Fiori Front-end Server

The Fiori Front-end Server uses a [NetWeaver Gateway](#). For small deployments, it can be loaded on the Fiori server. For large deployments, a separate server for the NetWeaver Gateway may be deployed in front of the Fiori Front-end Server pool.

## **Application servers pool**

To manage logon groups for ABAP application servers, the SMLG transaction is used. It uses the load balancing function within the message server of the Central Services to distribute workload among SAP application servers pool for SAPGUIs and RFC traffic. The application server connection to the highly available Central Services is through the cluster virtual network name. This avoids the need to change the application server profile for Central Services connectivity after a local failover.

## **SAP Central Services cluster**

Central Services can be deployed to a single virtual machine when high availability is not a requirement. However, the single virtual machine becomes a potential single point of failure (SPOF) for the SAP environment. For a highly available Central Services deployment, a highly available NFS cluster and a highly available Central Services cluster are used.

## **NFS cluster**

DRBD (Distributed Replicated Block Device) is used for replication between the nodes of the NFS cluster.

## **Availability sets**

Availability sets distribute servers to different physical infrastructure and update groups to improve service availability. Put virtual machines that perform the same role into an availability sets to help guard against downtime caused by Azure infrastructure maintenance and to meet [SLAs](#). Two or more virtual machines per availability set is recommended.

All virtual machines in a set must perform the same role. Do not mix servers of different roles in the same availability set. For example, don't place a ASCS node in the same availability set with the application server.

## **NICs**

Traditional on-premises SAP landscapes implement multiple network interface cards (NICs) per machine to segregate administrative traffic from business traffic. On Azure, the virtual network is a software-defined network that sends all traffic through the same network fabric. Therefore, the use of multiple NICs is unnecessary. However, if your organization needs to segregate traffic, you can deploy multiple NICs per VM, connect each NIC to a different subnet, and then use NSGs to enforce different access control policies.

## **Subnets and NSGs**

This architecture subdivides the virtual network address space into subnets. Each subnet can be associated with a NSG that defines the access policies for the subnet. Place application servers on a separate subnet so you can secure them more easily by managing the subnet security policies, not the individual servers.

When a NSG is associated with a subnet, it then applies to all the servers within the subnet. For more information about using NSGs for fine-grained control over the servers in a subnet, see [Filter network traffic with network security groups](#).

See also [Planning and design for VPN Gateway](#).

## **Load balancers**

[SAP Web Dispatcher](#) handles load balancing of HTTP(S) traffic including Fiori style applications to a pool of SAP application servers.

For traffic from SAP GUI clients connecting a SAP server via DIAG or Remote Function Calls (RFC), the Central Service message server balances the load through SAP application server [logon groups](#), so no additional load balancer is needed.

## **Azure Storage**

We recommend using Azure Premium Storage for the database server virtual machines. Premium storage provides consistent read/write latency. For details about using Premium Storage for the operating system disks and data disks of a single-instance virtual machine, see [SLA for Virtual Machines](#).

For all production SAP systems, we recommend using Premium [Azure Managed Disks](#). Managed Disks are used to manage the VHD files for the disks, adding reliability. They also ensure that the disks for virtual machines within an availability set are isolated to avoid single points of failure.

For SAP application servers, including the Central Services virtual machines, you can use Azure Standard Storage to reduce cost, because application execution takes place in memory and uses disks for logging only. However, at this time, Standard Storage is only certified for unmanaged storage. Since application servers do not host any data, you can also use the smaller P4 and P6 Premium Storage disks to help minimize cost.

For the backup data store, we recommend using Azure [cool access tier storage and/or archive access tier storage](#). These storage tiers are cost-effective ways to store long-lived data that is less frequently accessed.

## Performance considerations

SAP application servers carry on constant communications with the database servers. For the HANA database virtual machines, consider enabling [Write Accelerator](#) to improve log write latency. To optimize inter-server communications, use the [Accelerated Network](#). Note that these accelerators are available only for certain VM series.

To achieve high IOPS and disk bandwidth throughput, the common practices in storage volume [performance optimization](#) apply to Azure storage layout. For example, combining multiple disks together to create a striped disk volume improves IO performance. Enabling the read cache on storage content that changes infrequently enhances the speed of data retrieval. For details about performance requirements, see [SAP note 1943937 - Hardware Configuration Check Tool](#) (SAP Service Marketplace account required for access).

## Scalability considerations

At the SAP application layer, Azure offers a wide range of virtual machine sizes for scaling up and scaling out. For an inclusive list, see [SAP Note 1928533 - SAP Applications on Azure: Supported Products and Azure VM types](#) (SAP Service Marketplace account required for access). As we continue to certify more virtual machines types, you can scale up or down with the same cloud deployment.

At the database layer, this architecture runs HANA on VMs. If your workload exceeds the maximum VM size, Microsoft also offers [Azure Large Instances](#) for SAP HANA. These physical servers are co-located in a Microsoft Azure certified datacenter and as of this writing, provide up to 20 TB of memory capacity for a single instance. Multi-node configuration is also possible with a total memory capacity of up to 60 TB.

## Availability considerations

Resource redundancy is the general theme in highly available infrastructure solutions. For enterprises that have a less stringent SLA, single-instance Azure VMs offer an uptime SLA. For more information, see [Azure Service Level Agreement](#).

In this distributed installation of the SAP application, the base installation is replicated to achieve high availability. For each layer of the architecture, the high availability design varies.

### Application tier

- Web Dispatcher. High availability is achieved with redundant Web Dispatcher instances. See [SAP Web Dispatcher](#) in the SAP documentation.
- Fiori servers. High availability is achieved by load balancing traffic within a pool of servers.
- Central Services. For high availability of Central Services on Azure Linux virtual machines, the appropriate High Availability Extension for the selected Linux distribution is used, and the highly available NFS cluster hosts DRBD storage.
- Application servers. High availability is achieved by load balancing traffic within a pool of application servers.

## Database tier

This reference architecture depicts a highly available SAP HANA database system consisting of two Azure virtual machines. The database tier's native system replication feature provides either manual or automatic failover between replicated nodes:

- For manual failover, deploy more than one HANA instance and use HANA System Replication (HSR).
- For automatic failover, use both HSR and Linux High Availability Extension (HAE) for your Linux distribution. Linux HAE provides the cluster services to the HANA resources, detecting failure events and orchestrating the failover of errant services to the healthy node.

See [SAP certifications and configurations running on Microsoft Azure](#).

## Disaster recovery considerations

Each tier uses a different strategy to provide disaster recovery (DR) protection.

- **Application servers tier.** SAP application servers do not contain business data. On Azure, a simple DR strategy is to create SAP application servers in the secondary region, then shut them down. Upon any configuration changes or kernel updates on the primary application server, the same changes must be applied to the virtual machines in the secondary region. For example, copy the SAP kernel executables to the DR virtual machines. For automatic replication of application servers to a secondary region, [Azure Site Recovery](#) is the recommended solution. As of the writing of this paper, ASR doesn't yet support the replication of the Accelerated Network configuration setting in Azure VMs.
- **Central Services.** This component of the SAP application stack also does not persist business data. You can build a VM in the secondary region to run the Central Services role. The only content from the primary Central Services node to synchronize is the /sapmnt share content. Also, if configuration changes or kernel updates take place on the primary Central Services servers, they must be repeated on the VM in the secondary region running Central Services. To synchronize the two servers, you can use either Azure Site Recovery, to replicate the cluster nodes, or simply use a regularly scheduled copy job to copy /sapmnt to the DR side. For details about the build, copy, and test failover process, download [SAP NetWeaver: Building a Hyper-V and Microsoft Azure-based Disaster Recovery Solution](#), and refer to section 4.3, "SAP SPOF layer (ASCS)." This paper applies to NetWeaver running on Windows, but you can create the equivalent configuration for Linux. For Central Services, use [Azure Site Recovery](#) to replicate the cluster nodes and storage. For Linux, create a three node geo-cluster using a High Availability Extension.
- **SAP database tier.** Use HSR for HANA-supported replication. In addition to a local, two-node high availability setup, HSR supports multi-tier replication where a third node in a separate Azure region acts as a foreign entity, not part of the cluster, and registers to the secondary replica of the clustered HSR pair as its replication target. This form a replication daisy chain. The failover to the DR node is a manual process.

To use Azure Site Recovery to automatically build a fully replicated production site of your original, you must run customized [deployment scripts](#). Site Recovery first deploys the virtual machines in availability sets, then runs scripts to add resources such as load balancers.

## Manageability considerations

SAP HANA has a backup feature that makes use of the underlying Azure infrastructure. To back up the SAP HANA database running on Azure virtual machines, both the SAP HANA snapshot and Azure storage snapshot are used to ensure the backup files' consistency. For details, see [Backup guide for SAP HANA on Azure Virtual Machines](#) and the [Azure Backup service FAQ](#). Only HANA single container deployments support Azure storage snapshot.

## Identity management

Control access to resources by using a centralized identity management system at all levels:

- Provide access to Azure resources through [role-based access control](#) (RBAC).
- Grant access to Azure VMs through LDAP, Azure Active Directory, Kerberos, or another system.
- Support access within the apps themselves through the services that SAP provides, or use [OAuth 2.0 and Azure Active Directory](#).

## Monitoring

Azure provides several functions for [monitoring and diagnostics](#) of the overall infrastructure. Also, enhanced monitoring of Azure virtual machines (Linux or Windows) is handled by Azure Operations Management Suite (OMS).

To provide SAP-based monitoring of resources and service performance of the SAP infrastructure, the [Azure SAP Enhanced Monitoring](#) extension is used. This extension feeds Azure monitoring statistics into the SAP application for operating system monitoring and DBA Cockpit functions. SAP enhanced monitoring is a mandatory prerequisite to run SAP on Azure. For details, see [SAP Note 2191498](#) – "SAP on Linux with Azure: Enhanced Monitoring."

## Security considerations

SAP has its own Users Management Engine (UME) to control role-based access and authorization within the SAP application. For details, see [SAP HANA Security—An Overview](#) (SAP Service Marketplace account required for access.)

For additional network security, consider implementing a [Network DMZ](#), which uses a network virtual appliance to create a firewall in front of the subnet for Web Dispatcher and Fiori Front-End Server pools.

For infrastructure security, data is encrypted in transit and at rest. The "Security considerations" section of the [SAP NetWeaver on Azure Virtual Machines—Planning and Implementation Guide](#) begins to address network security and applies to S/4HANA. The guide also specifies the network ports you must open on the firewalls to allow application communication.

To encrypt Linux IaaS virtual machine disks, you can use [Azure Disk Encryption](#). It uses the DM-Crypt feature of Linux to provide volume encryption for the operating system and the data disks. The solution also works with Azure Key Vault to help you control and manage the disk-encryption keys and secrets in your key vault subscription. Data on the virtual machine disks are encrypted at rest in your Azure storage.

For SAP HANA data-at-rest encryption, we recommend using the SAP HANA native encryption technology.

### NOTE

Do not use the HANA data-at-rest encryption with Azure Disk Encryption on the same server. For HANA, use only HANA data encryption.

## Communities

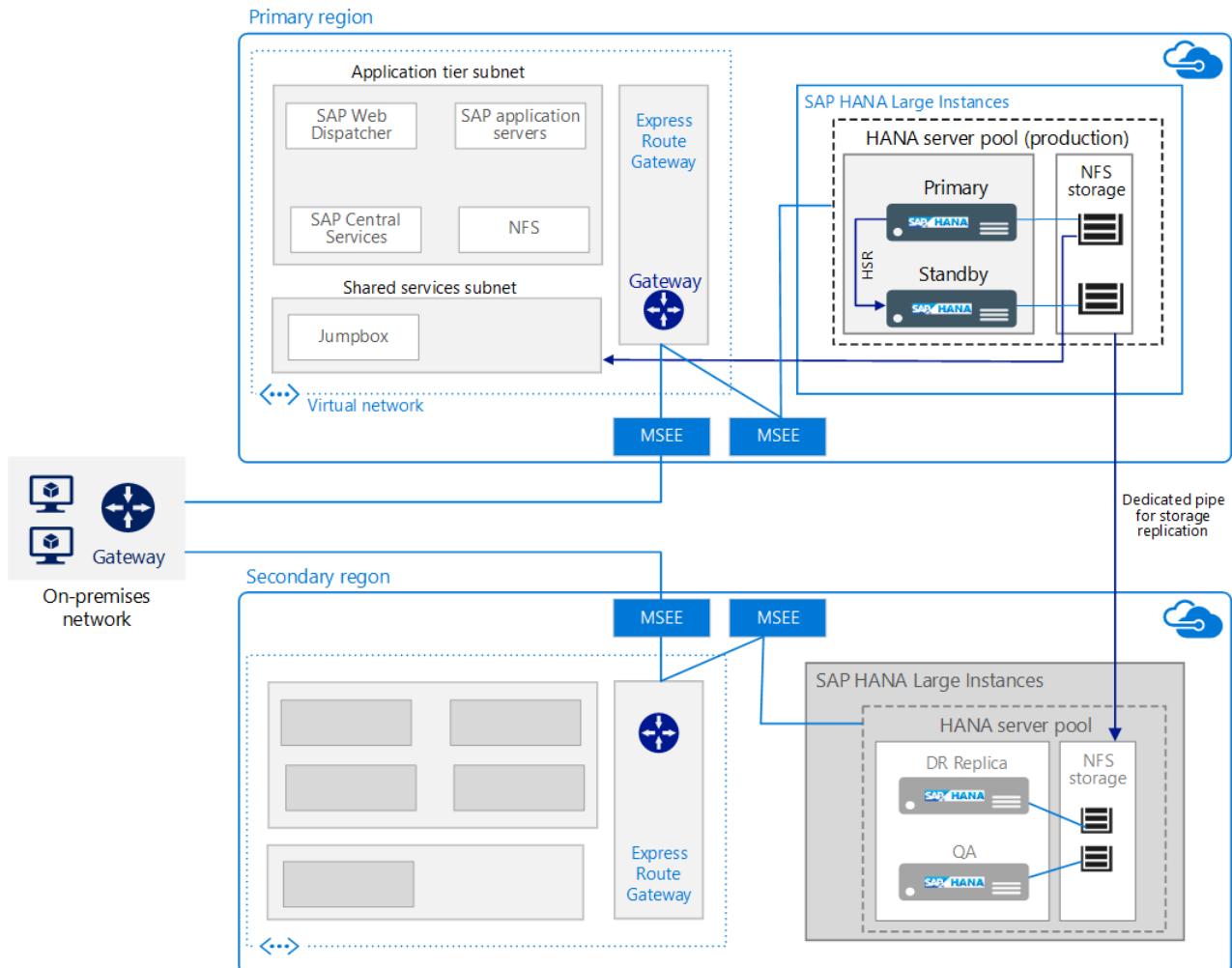
Communities can answer questions and help you set up a successful deployment. Consider the following:

- [Running SAP Applications on the Microsoft Platform Blog](#)
- [Azure Community Support](#)
- [SAP Community](#)
- [Stack Overflow](#)

# Run SAP HANA on Azure Large Instances

9/28/2018 • 10 minutes to read • [Edit Online](#)

This reference architecture shows a set of proven practices for running SAP HANA on Azure (Large Instances) with high availability and disaster recovery (DR). Called HANA Large Instances, this offering is deployed on physical servers in Azure regions.



Download a [Visio file](#) of this architecture.

## NOTE

Deploying this reference architecture requires appropriate licensing of SAP products and other non-Microsoft technologies.

## Architecture

This architecture consists of the following infrastructure components.

- **Virtual network.** The [Azure Virtual Network](#) service securely connects Azure resources to each other and is subdivided into separate [subnets](#) for each layer. SAP application layers are deployed on Azure virtual machines (VMs) to connect to the HANA database layer residing on large instances.
- **Virtual machines.** Virtual machines are used in the SAP application layer and shared services layer. The latter includes a jumpbox used by administrators to set up HANA Large Instances and to provide access to

other virtual machines.

- **HANA Large Instance.** A [physical server](#) certified to meet SAP HANA Tailored Datacenter Integration (TDI) standards runs SAP HANA. This architecture uses two HANA Large Instances: a primary and a secondary compute unit. High availability at the data layer is provided through HANA System Replication (HSR).
- **High Availability Pair.** A group of HANA Large Instances blades are managed together to provide application redundancy and reliability.
- **MSEE (Microsoft Enterprise Edge).** MSEE is a connection point from a connectivity provider or your network edge through an ExpressRoute circuit.
- **Network interface cards (NICs).** To enable communication, the HANA Large Instance server provides four virtual NICs by default. This architecture requires one NIC for client communication, a second NIC for the node-to-node connectivity needed by HSR, a third NIC for HANA Large Instance storage, and a fourth for iSCSI used in high availability clustering.
- **Network File System (NFS) storage.** The [NFS](#) server supports the network file share that provides secure data persistence for HANA Large Instance.
- **ExpressRoute.** [ExpressRoute](#) is the recommended Azure networking service for creating private connections between an on-premises network and Azure virtual networks that do not go over the public Internet. Azure VMs connect to HANA Large Instances using another ExpressRoute connection. The ExpressRoute connection between the Azure virtual network and the HANA Large Instances is set up as part of the Microsoft offering.
- **Gateway.** The ExpressRoute Gateway is used to connect the Azure virtual network used for the SAP application layer to the HANA Large Instance network. Use the [High Performance or Ultra Performance](#) SKU.
- **Disaster recovery (DR).** Upon request, storage replication is supported and will be configured from the primary to the [DR site](#) located in another region.

## Recommendations

Requirements can vary, so use these recommendations as a starting point.

### HANA Large Instances compute

[Large Instances](#) are physical servers based on the Intel EX E7 CPU architecture and configured in a large instance stamp—that is, a specific set of servers or blades. A compute unit equals one server or blade, and a stamp is made up of multiple servers or blades. Within a large instance stamp, servers are not shared and are dedicated to running one customer’s deployment of SAP HANA.

A variety of SKUs are available for HANA Large Instances, supporting up to 20 TB single instance (60 TB scale-out) of memory for S/4HANA or other SAP HANA workloads. [Two classes](#) of servers are offered:

- Type I class: S72, S72m, S144, S144m, S192, and S192m
- Type II class: S384, S384m, S384xm, S576m, S768m, and S960m

For example, the S72 SKU comes with 768 GB RAM, 3 terabytes (TB) of storage, and 2 Intel Xeon processors (E7-8890 v3) with 36 cores. Choose a SKU that fulfills the sizing requirements you determined in your architecture and design sessions. Always ensure that your sizing applies to the correct SKU. Capabilities and deployment requirements [vary by type](#), and availability varies by [region](#). You can also step up from one SKU to a larger SKU.

Microsoft helps establish the large instance setup, but it is your responsibility to verify the operating system’s configuration settings. Make sure to review the most current SAP Notes for your exact Linux release.

## Storage

Storage layout is implemented according to the recommendation of the TDI for SAP HANA. HANA Large Instances come with a specific storage configuration for the standard TDI specifications. However, you can purchase additional storage in 1 TB increments.

To support the requirements of mission-critical environments including fast recovery, NFS is used and not direct attached storage. The NFS storage server for HANA Large Instances is hosted in a multi-tenant environment, where tenants are segregated and secured using compute, network, and storage isolation.

To support high availability at the primary site, use different storage layouts. For example, in a multi-host scale-out, the storage is shared. Another high availability option is application-based replication such as HSR. For DR, however, a snapshot-based storage replication is used.

## Networking

This architecture uses both virtual and physical networks. The virtual network is part of Azure IaaS and connects to a discrete HANA Large Instances physical network through [ExpressRoute](#) circuits. A cross-premises gateway connects your workloads in the Azure virtual network to your on-premises sites.

HANA Large Instances networks are isolated from each other for security. Instances residing in different regions do not communicate with each other, except for the dedicated storage replication. However, to use HSR, inter-region communications are required. [IP routing tables](#) or proxies can be used to enable cross-regions HSR.

All Azure virtual networks that connect to HANA Large Instances in one region can be [cross-connected](#) via ExpressRoute to HANA Large Instances in a secondary region.

ExpressRoute for HANA Large Instances is included by default during provisioning. For setup, a specific network layout is needed, including required CIDR address ranges and domain routing. For details, see [SAP HANA \(large instances\) infrastructure and connectivity on Azure](#).

## Scalability considerations

To scale up or down, you can choose from many sizes of servers that are available for HANA Large Instances. They are categorized as [Type I and Type II](#) and tailored for different workloads. Choose a size that can grow with your workload for the next three years. One-year commitments are also available.

A multi-host scale-out deployment is generally used for BW/4HANA deployments as a kind of database partitioning strategy. To scale out, plan the placement of HANA tables prior to installation. From an infrastructure standpoint, multiple hosts are connected to a shared storage volume, enabling quick takeover by standby hosts in case one of the compute worker nodes in the HANA system fails.

S/4HANA and SAP Business Suite on HANA on a single blade can be scaled up to 20 TB with a single HANA Large Instances instance.

For greenfield scenarios, the [SAP Quick Sizer](#) is available to calculate memory requirements of the implementation of SAP software on top of HANA. Memory requirements for HANA increase as data volume grows. Use your system's current memory consumption as the basis for predicting future consumption, and then map your demand into one of the HANA Large Instances sizes.

If you already have SAP deployments, SAP provides reports you can use to check the data used by existing systems and calculate memory requirements for a HANA instance. For example, see the following SAP Notes:

- SAP Note [1793345](#) - Sizing for SAP Suite on HANA
- SAP Note [1872170](#) - Suite on HANA and S/4 HANA sizing report
- SAP Note [2121330](#) - FAQ: SAP BW on HANA Sizing Report
- SAP Note [1736976](#) - Sizing Report for BW on HANA
- SAP Note [2296290](#) - New Sizing Report for BW on HANA

## Availability considerations

Resource redundancy is the general theme in highly available infrastructure solutions. For enterprises that have a less stringent SLA, single-instance Azure VMs offer an uptime SLA. For more information, see [Azure Service Level Agreement](#).

Work with SAP, your system integrator, or Microsoft to properly architect and implement a [high availability and disaster-recovery](#) strategy. This architecture follows the Azure [service-level agreement](#) (SLA) for HANA on Azure (Large Instances). To assess your availability requirements, consider any single points of failure, the desired level of uptime for services, and these common metrics:

- Recovery Time Objective (RTO) means the duration of time in which the HANA Large Instances server is unavailable.
- Recovery Point Objective (RPO) means the maximum tolerable period in which customer data might be lost due to a failure.

For high availability, deploy more than one instance in a HA Pair and use HSR in a synchronous mode to minimize data loss and downtime. In addition to a local, two-node high availability setup, HSR supports multi-tier replication, where a third node in a separate Azure region registers to the secondary replica of the clustered HSR pair as its replication target. This forms a replication daisy chain. The failover to the DR node is a manual process.

When you set up HANA Large Instances HSR with automatic failover, you can request the Microsoft Service Management team to set up a [STONITH device](#) for your existing servers.

## Disaster recovery considerations

This architecture supports [disaster recovery](#) between HANA Large Instances in different Azure regions. There are two ways to support DR with HANA Large Instances:

- Storage replication. The primary storage contents are constantly replicated to the remote DR storage systems that are available on the designated DR HANA Large Instances server. In storage replication, the HANA database is not loaded into memory. This DR option is simpler from an administration perspective. To determine if this is a suitable strategy, consider the database load time against the availability SLA. Storage replication also enables you to perform point-in-time recovery. If multi-purpose (cost-optimized) DR is set up, you must purchase additional storage of the same size at the DR location. Microsoft provides self-services [storage snapshot and failover scripts](#) for HANA failover as part of the HANA Large Instances offering.
- Multi-tier HSR with a third replica in the DR region (where the HANA database is loaded onto memory). This option supports a faster recovery time but does not support a point-in-time recovery. HSR requires a secondary system. HANA system replication for the DR site is handled through proxies such as nginx or IP tables.

### NOTE

You can optimize this reference architecture for costs by running in a single-instance environment. This [cost-optimized scenario](#) is suitable for non-production HANA workloads.

## Backup considerations

Based on your business requirements, choose from several options available for [backup and recovery](#).

| Backup Option      | Pros                                                                                                   | Cons                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| HANA backup        | Native to SAP. Built-in consistency check.                                                             | Long backup and recovery times.<br>Storage space consumption. |
| HANA snapshot      | Native to SAP. Rapid backup and restore.                                                               |                                                               |
| Storage snapshot   | Included with HANA Large Instances. Optimized DR for HANA Large Instances. Boot volume backup support. | Maximum 254 snapshots per volume.                             |
| Log backup         | Required for point in time recovery.                                                                   |                                                               |
| Other backup tools | Redundant backup location.                                                                             | Additional licensing costs.                                   |

## Manageability considerations

Monitor HANA Large Instances resources such as CPU, memory, network bandwidth, and storage space using SAP HANA Studio, SAP HANA Cockpit, SAP Solution Manager, and other native Linux tools. HANA Large Instances does not come with built-in monitoring tools. Microsoft offers resources to help you [troubleshoot and monitor](#) according to your organization's requirements, and the Microsoft support team can assist you in troubleshooting technical issues.

If you need more computing capability, you must get a larger SKU.

## Security considerations

- By default, HANA Large Instances use storage encryption based on TDE (transparent data encryption) for the data at rest.
- Data in transit between HANA Large Instances and the virtual machines is not encrypted. To encrypt the data transfer, enable the application-specific encryption. See SAP Note [2159014](#) - FAQ: SAP HANA Security.
- Isolation provides security between the tenants in the multi-tenant HANA Large Instance environment. Tenants are isolated using their own VLAN.
- [Azure network security best practices](#) provide helpful guidance.
- As with any deployment, [operating system hardening](#) is recommended.
- For physical security, access to Azure datacenters is limited to authorized personnel only. No customers can access the physical servers.

For more information, see [SAP HANA Security—An Overview](#). (A SAP Service Marketplace account is required for access.)

## Communities

Communities can answer questions and help you set up a successful deployment. Consider the following:

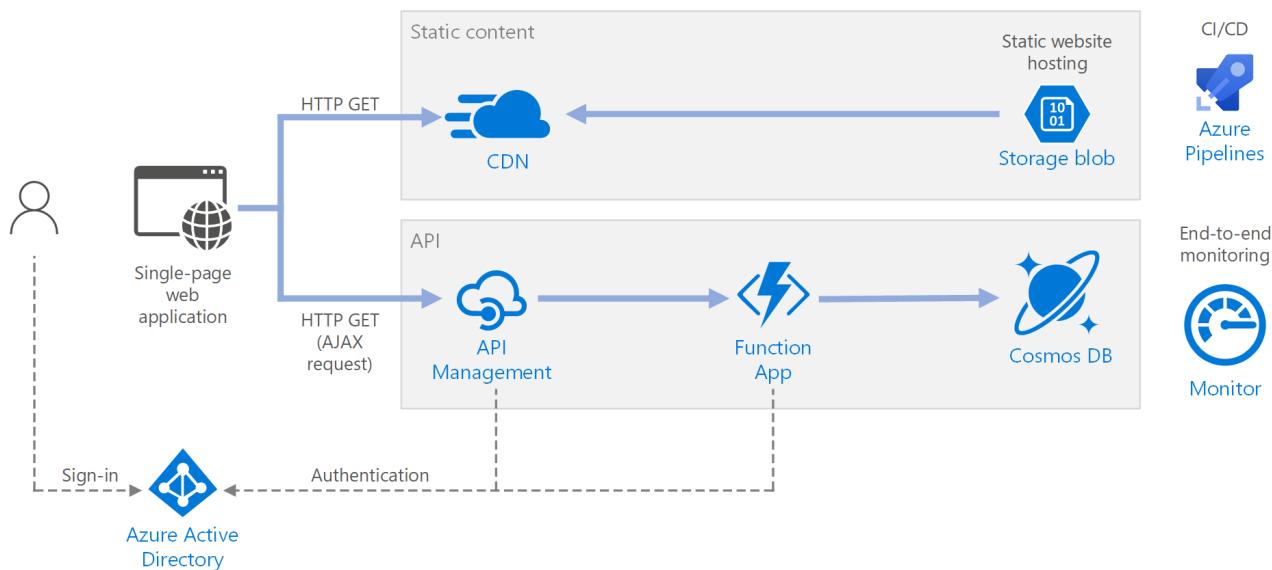
- [Running SAP Applications on the Microsoft Platform Blog](#)
- [Azure Community Support](#)
- [SAP Community](#)

- Stack Overflow SAP

# Serverless web application

10/16/2018 • 15 minutes to read • [Edit Online](#)

This reference architecture shows a serverless web application. The application serves static content from Azure Blob Storage, and implements an API using Azure Functions. The API reads data from Cosmos DB and returns the results to the web app. A reference implementation for this architecture is available on [GitHub](#).



The term **serverless** has two distinct but related meanings:

- **Backend as a service (BaaS)**. Backend cloud services, such as databases and storage, provide APIs that enable client applications to connect directly to these services.
- **Functions as a service (FaaS)**. In this model, a "function" is a piece of code that is deployed to the cloud and runs inside a hosting environment that completely abstracts the servers that run the code.

Both definitions have in common the idea that developers and DevOps personnel don't need to deploy, configure, or manage servers. This reference architecture focuses on FaaS using Azure Functions, although serving web content from Azure Blob Storage is an example of BaaS. Some important characteristics of FaaS are:

1. Compute resources are allocated dynamically as needed by the platform.
2. Consumption-based pricing: You are charged only for the compute resources used to execute your code.
3. The compute resources scale on demand based on traffic, without the developer needing to do any configuration.

Functions are executed when an external trigger occurs, such as an HTTP request or a message arriving on a queue. This makes an [event-driven architecture style](#) natural for serverless architectures. To coordinate work between components in the architecture, consider using message brokers or pub/sub patterns. For help choosing between messaging technologies in Azure, see [Choose between Azure services that deliver messages](#).

## Architecture

The architecture consists of the following components.

**Blob Storage.** Static web content, such as HTML, CSS, and JavaScript files, are stored in Azure Blob Storage and served to clients by using [static website hosting](#). All dynamic interaction happens through JavaScript code making calls to the backend APIs. There is no server-side code to render the web page. Static website hosting supports index documents and custom 404 error pages.

#### NOTE

Static website hosting is currently in [preview](#).

**CDN.** Use [Azure Content Delivery Network](#) (CDN) to cache content for lower latency and faster delivery of content, as well as providing an HTTPS endpoint.

**Function Apps.** [Azure Functions](#) is a serverless compute option. It uses an event-driven model, where a piece of code (a "function") is invoked by a trigger. In this architecture, the function is invoked when a client makes an HTTP request. The request is always routed through an API gateway, described below.

**API Management.** [API Management](#) provides a API gateway that sits in front of the HTTP function. You can use API Management to publish and manage APIs used by client applications. Using a gateway helps to decouple the front-end application from the back-end APIs. For example, API Management can rewrite URLs, transform requests before they reach the backend, set request or response headers, and so forth.

API Management can also be used to implement cross-cutting concerns such as:

- Enforcing usage quotas and rate limits
- Validating OAuth tokens for authentication
- Enabling cross-origin requests (CORS)
- Caching responses
- Monitoring and logging requests

If you don't need all of the functionality provided by API Management, another option is to use [Functions Proxies](#). This feature of Azure Functions lets you define a single API surface for multiple function apps, by creating routes to back-end functions. Function proxies can also perform limited transformations on the HTTP request and response. However, they don't provide the same rich policy-based capabilities of API Management.

**Cosmos DB.** [Cosmos DB](#) is a multi-model database service. For this scenario, the function application fetches documents from Cosmos DB in response to HTTP GET requests from the client.

**Azure Active Directory** (Azure AD). Users sign into the web application by using their Azure AD credentials. Azure AD returns an access token for the API, which the web application uses to authenticate API requests (see [Authentication](#)).

**Azure Monitor.** [Monitor](#) collects performance metrics about the Azure services deployed in the solution. By visualizing these in a dashboard, you can get visibility into the health of the solution. It also collected application logs.

**Azure Pipelines.** [Pipelines](#) is a continuous integration (CI) and continuous delivery (CD) service that builds, tests, and deploys the application.

## Recommendations

### Function App plans

Azure Functions supports two hosting models. With the **consumption plan**, compute power is automatically allocated when your code is running. With the **App Service** plan, a set of VMs are allocated for your code. The App Service plan defines the number of VMs and the VM size.

Note that the App Service plan is not strictly *serverless*, according to the definition given above. The programming model is the same, however — the same function code can run in both a consumption plan and an App Service plan.

Here are some factors to consider when choosing which type of plan to use:

- **Cold start.** With the consumption plan, a function that hasn't been invoked recently will incur some additional latency the next time it runs. This additional latency is due to allocating and preparing the runtime environment. It is usually on the order of seconds but depends on several factors, including the number of dependencies that need to be loaded. For more information, see [Understanding Serverless Cold Start](#). Cold start is usually more of a concern for interactive workloads (HTTP triggers) than asynchronous message-driven workloads (queue or event hubs triggers), because the additional latency is directly observed by users.
- **Timeout period.** In the consumption plan, a function execution times out after a [configurable](#) period of time (to a maximum of 10 minutes)
- **Virtual network isolation.** Using an App Service plan allows functions to run inside of an [App Service Environment](#), which is a dedicated and isolated hosting environment.
- **Pricing model.** The consumption plan is billed by the number of executions and resource consumption ( $\text{memory} \times \text{execution time}$ ). The App Service plan is billed hourly based on VM instance SKU. Often, the consumption plan can be cheaper than an App Service plan, because you pay only for the compute resources that you use. This is especially true if your traffic experiences peaks and troughs. However, if an application experiences constant high-volume throughput, an App Service plan may cost less than the consumption plan.
- **Scaling.** A big advantage of the consumption model is that it scales dynamically as needed, based on the incoming traffic. While this scaling occurs quickly, there is still a ramp-up period. For some workloads, you might want to deliberately overprovision the VMs, so that you can handle bursts of traffic with zero ramp-up time. In that case, consider an App Service plan.

## Function App boundaries

A *function app* hosts the execution of one or more *functions*. You can use a function app to group several functions together as a logical unit. Within a function app, the functions share the same application settings, hosting plan, and deployment lifecycle. Each function app has its own hostname.

Use function apps to group functions that share the same lifecycle and settings. Functions that don't share the same lifecycle should be hosted in different function apps.

Consider taking a microservices approach, where each function app represents one microservice, possibly consisting of several related functions. In a microservices architecture, services should have loose coupling and high functional cohesion. *Loosely coupled* means you can change one service without requiring other services to be updated at the same time. *Cohesive* means a service has a single, well-defined purpose. For more discussion of these ideas, see [Designing microservices: Domain analysis](#).

## Function bindings

Use Functions [bindings](#) when possible. Bindings provide a declarative way to connect your code to data and integrate with other Azure services. An input binding populates an input parameter from an external data source. An output binding sends the function's return value to a data sink, such as a queue or database.

For example, the `GetStatus` function in the reference implementation uses the Cosmos DB [input binding](#). This binding is configured to look up a document in Cosmos DB, using query parameters that are taken from the query string in the HTTP request. If the document is found, it is passed to the function as a parameter.

```

[FunctionName("GetStatusFunction")]
public static Task<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,
    ILogger log)
{
    ...
}

```

By using bindings, you don't need to write code that talks directly to the service, which makes the function code simpler and also abstracts the details of the data source or sink. In some cases, however, you may need more complex logic than the binding provides. In that case, use the Azure client SDKs directly.

## Scalability considerations

**Functions.** For the consumption plan, the HTTP trigger scales based on the traffic. There is a limit to the number of concurrent function instances, but each instance can process more than one request at a time. For an App Service plan, the HTTP trigger scales according to the number of VM instances, which can be a fixed value or can autoscale based on a set of autoscaling rules. For information, see [Azure Functions scale and hosting](#).

**Cosmos DB.** Throughput capacity for Cosmos DB is measured in [Request Units \(RU\)](#). A 1-RU throughput corresponds to the throughput need to GET a 1KB document. In order to scale a Cosmos DB container past 10,000 RU, you must specify a [partition key](#) when you create the container and include the partition key in every document that you create. For more information about partition keys, see [Partition and scale in Azure Cosmos DB](#).

**API Management.** API Management can scale out and supports rule-based autoscaling. Note that the scaling process takes at least 20 minutes. If your traffic is bursty, you should provision for the maximum burst traffic that you expect. However, autoscaling is useful for handling hourly or daily variations in traffic. For more information, see [Automatically scale an Azure API Management instance](#).

## Disaster recovery considerations

The deployment shown here resides in a single Azure region. For a more resilient approach to disaster-recovery, take advantage of the geo-distribution features in the various services:

- API Management supports multi-region deployment, which can be used to distribute a single API Management instance across any number of Azure regions. For more information, see [How to deploy an Azure API Management service instance to multiple Azure regions](#).
- Use [Traffic Manager](#) to route HTTP requests to the primary region. If the Function App running in that region becomes unavailable, Traffic Manager can fail over to a secondary region.
- Cosmos DB supports [multiple master regions](#), which enables writes to any region that you add to your Cosmos DB account. If you don't enable multi-master, you can still fail over the primary write region. The Cosmos DB client SDKs and the Azure Function bindings automatically handle the failover, so you don't need to update any application configuration settings.

## Security considerations

### Authentication

The `GetStatus` API in the reference implementation uses Azure AD to authenticate requests. Azure AD supports

the Open ID Connect protocol, which is an authentication protocol built on top of the OAuth 2 protocol.

In this architecture, the client application is a single-page application (SPA) that runs in the browser. This type of client application cannot keep a client secret or an authorization code hidden, so the implicit grant flow is appropriate. (See [Which OAuth 2.0 flow should I use?](#)). Here's the overall flow:

1. The user clicks the "Sign in" link in the web application.
2. The browser is redirected to the Azure AD sign in page.
3. The user signs in.
4. Azure AD redirects back to the client application, including an access token in the URL fragment.
5. When the web application calls the API, it includes the access token in the Authentication header. The application ID is sent as the audience ('aud') claim in the access token.
6. The backend API validates the access token.

To configure authentication:

- Register an application in your Azure AD tenant. This generates an application ID, which the client includes with the login URL.
- Enable Azure AD authentication inside the Function App. For more information, see [Authentication and authorization in Azure App Service](#).
- Add a policy to API Management to pre-authorize the request by validating the access token:

```
<validate-jwt header-name="Authorization" failed-validation-httpcode="401" failed-validation-error-
message="Unauthorized. Access token is missing or invalid.">
    <openid-config url="https://login.microsoftonline.com/[Azure AD tenant ID]/.well-known/openid-
    configuration" />
    <required-claims>
        <claim name="aud">
            <value>[Application ID]</value>
        </claim>
    </required-claims>
</validate-jwt>
```

For more details, see the [GitHub readme](#).

## Authorization

In many applications, the backend API must check whether a user has permission to perform a given action. It's recommended to use [claims-based authorization](#), where information about the user is conveyed by the identity provider (in this case, Azure AD) and used to make authorization decisions.

Some claims are provided inside the ID token that Azure AD returns to the client. You can get these claims from within the function app by examining the X-MS-CLIENT-PRINCIPAL header in the request. For other claims, use [Microsoft Graph](#) to query Azure AD (requires user consent during sign-in).

For example, when you register an application in Azure AD, you can define a set of application roles in the application's registration manifest. When a user signs into the application, Azure AD includes a "roles" claim for each role that the user has been granted (including roles that are inherited through group membership).

In the reference implementation, the function checks whether the authenticated user is a member of the [GetStatus](#) application role. If not, the function returns an HTTP Unauthorized (401) response.

```

[FunctionName("GetStatusFunction")]
public static Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", 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,
    ILogger log)
{
    log.LogInformation("Processing GetStatus request.");

    return req.HandleIfAuthorizedForRoles(new[] { GetDeviceStatusRoleName },
        async () =>
    {
        string deviceId = req.Query["deviceId"];
        if (deviceId == null)
        {
            return new BadRequestObjectResult("Missing DeviceId");
        }

        return await Task.FromResult<IActionResult>(deviceStatus != null
            ? (ActionResult)new OkObjectResult(deviceStatus)
            : new NotFoundResult());
    },
    log);
}

```

In this code example, `HandleIfAuthorizedForRoles` is an extension method that checks for the role claim and returns HTTP 401 if the claim isn't found. You can find the source code [here](#). Notice that `HandleIfAuthorizedForRoles` takes an `ILogger` parameter. You should log unauthorized requests so that you have an audit trail and can diagnose issues if needed. At the same time, avoid leaking any detailed information inside the HTTP 401 response.

## CORS

In this reference architecture, the web application and the API do not share the same origin. That means when the application calls the API, it is a cross-origin request. Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the *same-origin policy* and prevents a malicious site from reading sensitive data from another site. To enable a cross-origin request, add a Cross-Origin Resource Sharing (CORS) [policy](#) to the API Management gateway:

```

<cors allow-credentials="true">
    <allowed-origins>
        <origin>[Website URL]</origin>
    </allowed-origins>
    <allowed-methods>
        <method>GET</method>
    </allowed-methods>
    <allowed-headers>
        <header>*</header>
    </allowed-headers>
</cors>

```

In this example, the **allow-credentials** attribute is **true**. This authorizes the browser to send credentials (including cookies) with the request. Otherwise, by default the browser does not send credentials with a cross-origin request.

## NOTE

Be very careful about setting **allow-credentials** to **true**, because it means a website can send the user's credentials to your API on the user's behalf, without the user being aware. You must trust the allowed origin.

## Enforce HTTPS

For maximum security, require HTTPS throughout the request pipeline:

- **CDN.** Azure CDN supports HTTPS on the `*.azureedge.net` subdomain by default. To enable HTTPS in the CDN for custom domain names, see [Tutorial: Configure HTTPS on an Azure CDN custom domain](#).
- **Static website hosting.** Enable the "Secure transfer required" option on the Storage account. When this option is enabled, the storage account only allows requests from secure HTTPS connections.
- **API Management.** Configure the APIs to use HTTPS protocol only. You can configure this in the Azure portal or through a Resource Manager template:

```
{  
    "apiVersion": "2018-01-01",  
    "type": "apis",  
    "name": "dronedeliveryapi",  
    "dependsOn": [  
        "[concat('Microsoft.ApiManagement/service/', variables('apiManagementServiceName'))]"  
    ],  
    "properties": {  
        "displayName": "Drone Delivery API",  
        "description": "Drone Delivery API",  
        "path": "api",  
        "protocols": [ "HTTPS" ]  
    },  
    ...  
}
```

- **Azure Functions.** Enable the "HTTPS Only" setting.

## Lock down the function app

All calls to the function should go through the API gateway. You can achieve this as follows:

- Configure the function app to require a function key. The API Management gateway will include the function key when it calls the function app. This prevents clients from calling the function directly, bypassing the gateway.
- The API Management gateway has a [static IP address](#). Restrict the Azure Function to allow only calls from that static IP address. For more information, see [Azure App Service Static IP Restrictions](#). (This feature is available for Standard tier services only.)

## Protect application secrets

Don't store application secrets, such as database credentials, in your code or configuration files. Instead, use App settings, which are stored encrypted in Azure. For more information, see [Security in Azure App Service and Azure Functions](#).

Alternatively, you can store application secrets in Key Vault. This allows you to centralize the storage of secrets, control their distribution, and monitor how and when secrets are being accessed. For more information, see [Configure an Azure web application to read a secret from Key Vault](#). However, note that Functions triggers and bindings load their configuration settings from app settings. There is no built-in way to configure the triggers and bindings to use Key Vault secrets.

# DevOps considerations

## API versioning

An API is a contract between a service and clients or consumers of that service. Support versioning in your API contract. If you introduce a breaking API change, introduce a new API version. Deploy the new version side-by-side with the original version, in a separate Function App. This lets you migrate existing clients to the new API without breaking client applications. Eventually, you can deprecate the previous version. For more information about API versioning, see [Versioning a RESTful web API](#).

For updates that are not breaking API changes, deploy the new version to a staging slot in the same Function App. Verify the deployment succeeded and then swap the staged version with the production version.

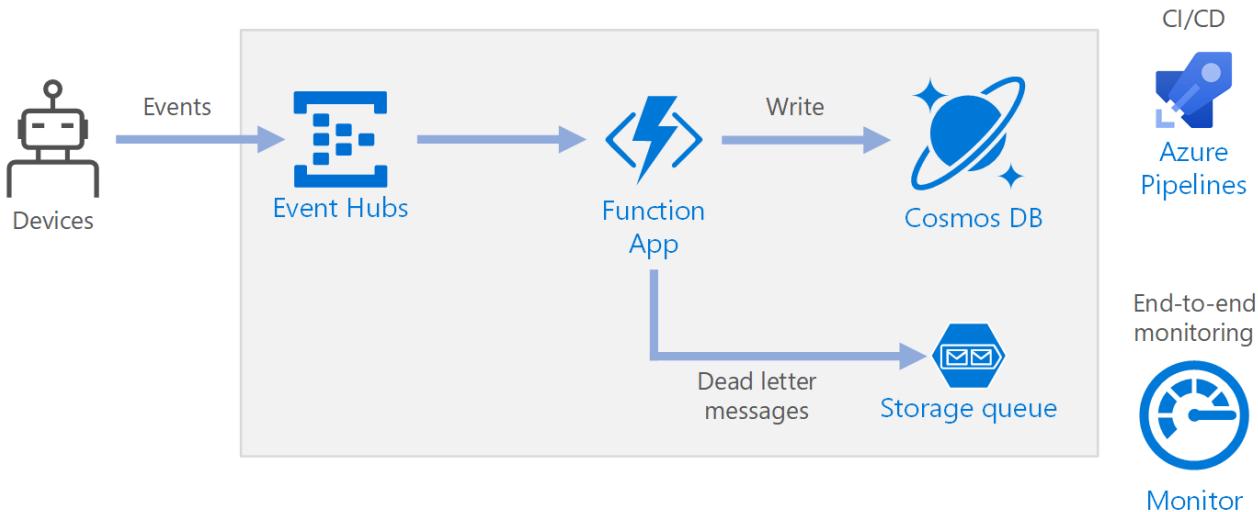
## Deploy the solution

To deploy this reference architecture, view the [GitHub readme](#).

# Serverless event processing using Azure Functions

10/16/2018 • 5 minutes to read • [Edit Online](#)

This reference architecture shows a serverless, event-driven architecture that ingests a stream of data, processes the data, and writes the results to a back-end database. A reference implementation for this architecture is available on [GitHub](#).



## Architecture

**Event Hubs** ingests the data stream. [Event Hubs](#) is designed for high-throughput data streaming scenarios.

### NOTE

For IoT scenarios, we recommend IoT Hub. IoT Hub has a built-in endpoint that's compatible with the Azure Event Hubs API, so you can use either service in this architecture with no major changes in the backend processing. For more information, see [Connecting IoT Devices to Azure: IoT Hub and Event Hubs](#).

**Function App.** [Azure Functions](#) is a serverless compute option. It uses an event-driven model, where a piece of code (a "function") is invoked by a trigger. In this architecture, when events arrive at Event Hubs, they trigger a function that processes the events and writes the results to storage.

Function Apps are suitable for processing individual records from Event Hubs. For more complex stream processing scenarios, consider Apache Spark using Azure Databricks, or Azure Stream Analytics.

**Cosmos DB.** [Cosmos DB](#) is a multi-model database service. For this scenario, the event-processing function stores JSON records, using the Cosmos DB [SQL API](#).

**Queue storage.** [Queue storage](#) is used for dead letter messages. If an error occurs while processing an event, the function stores the event data in a dead letter queue for later processing. For more information, see [Resiliency Considerations](#).

**Azure Monitor.** [Monitor](#) collects performance metrics about the Azure services deployed in the solution. By visualizing these in a dashboard, you can get visibility into the health of the solution.

**Azure Pipelines.** [Pipelines](#) is a continuous integration (CI) and continuous delivery (CD) service that builds, tests, and deploys the application.

# Scalability considerations

## Event Hubs

The throughput capacity of Event Hubs is measured in [throughput units](#). You can autoscale an event hub by enabling [auto-inflate](#), which automatically scales the throughput units based on traffic, up to a configured maximum.

The [Event Hub trigger](#) in the function app scales according to the number of partitions in the event hub. Each partition is assigned one function instance at a time. To maximize throughput, receive the events in a batch, instead of one at a time.

## Cosmos DB

Throughput capacity for Cosmos DB is measured in [Request Units](#) (RU). In order to scale a Cosmos DB container past 10,000 RU, you must specify a [partition key](#) when you create the container, and include the partition key in every document that you create.

Here are some characteristics of a good partition key:

- The key value space is large.
- There will be an even distribution of reads/writes per key value, avoiding hot keys.
- The maximum data stored for any single key value will not exceed the maximum physical partition size (10 GB).
- The partition key for a document won't change. You can't update the partition key on an existing document.

In the scenario for this reference architecture, the function stores exactly one document per device that is sending data. The function continually updates the documents with latest device status, using an upsert operation. Device ID is a good partition key for this scenario, because writes will be evenly distributed across the keys, and the size of each partition will be strictly bounded, because there is a single document for each key value. For more information about partition keys, see [Partition and scale in Azure Cosmos DB](#).

# Resiliency considerations

When using the Event Hubs trigger with Functions, catch exceptions within your processing loop. If an unhandled exception occurs, the Functions runtime does not retry the messages. If a message cannot be processed, put the message into a dead letter queue. Use an out-of-band process to examine the messages and determine corrective action.

The following code shows how the ingestion function catches exceptions and puts unprocessed messages onto a dead letter queue.

```

[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]
public static async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection", ConsumerGroup
    ="%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage> deadLetterMessages,
    ILogger logger)
{
    foreach (var message in messages)
    {
        DeviceState deviceState = null;

        try
        {
            deviceState = telemetryProcessor.Deserialize(message.Body.Array, logger);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error deserializing message", message.SystemProperties.PartitionKey,
message.SystemProperties.SequenceNumber);
            await deadLetterMessages.AddAsync(new DeadLetterMessage { Issue = ex.Message, EventData = message
});;
        }

        try
        {
            await stateChangeProcessor.UpdateState(deviceState, logger);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error updating status document", deviceState);
            await deadLetterMessages.AddAsync(new DeadLetterMessage { Issue = ex.Message, EventData = message,
DeviceState = deviceState });
        }
    }
}

```

Notice that the function uses the [Queue storage output binding](#) to put items in the queue.

The code shown above also logs exceptions to Application Insights. You can use the partition key and sequence number to correlate dead letter messages with the exceptions in the logs.

Messages in the dead letter queue should have enough information so that you can understand the context of error. In this example, the `DeadLetterMessage` class contains the exception message, the original event data, and the serialized event message (if available).

```

public class DeadLetterMessage
{
    public string Issue { get; set; }
    public EventData EventData { get; set; }
    public DeviceState DeviceState { get; set; }
}

```

Use [Azure Monitor](#) to monitor the event hub. If you see there is input but no output, it means that messages are not being processed. In that case, go into [Log Analytics](#) and look for exceptions or other errors.

## Disaster recovery considerations

The deployment shown here resides in a single Azure region. For a more resilient approach to disaster-recovery, take advantage of geo-distribution features in the various services:

- **Event Hubs.** Create two Event Hubs namespaces, a primary (active) namespace and a secondary (passive)

namespace. Messages are automatically routed to the active namespace unless you fail over to the secondary namespace. For more information, see [Azure Event Hubs Geo-disaster recovery](#).

- **Function App.** Deploy a second function app that is waiting to read from the secondary Event Hubs namespace. This function writes to a secondary storage account for dead letter queue.
- **Cosmos DB.** Cosmos DB supports [multiple master regions](#), which enables writes to any region that you add to your Cosmos DB account. If you don't enable multi-master, you can still fail over the primary write region. The Cosmos DB client SDKs and the Azure Function bindings automatically handle the failover, so you don't need to update any application configuration settings.
- **Azure Storage.** Use [RA-GRS](#) storage for the dead letter queue. This creates a read-only replica in another region. If the primary region becomes unavailable, you can read the items currently in the queue. In addition, provision another storage account in the secondary region that the function can write to after a fail-over.

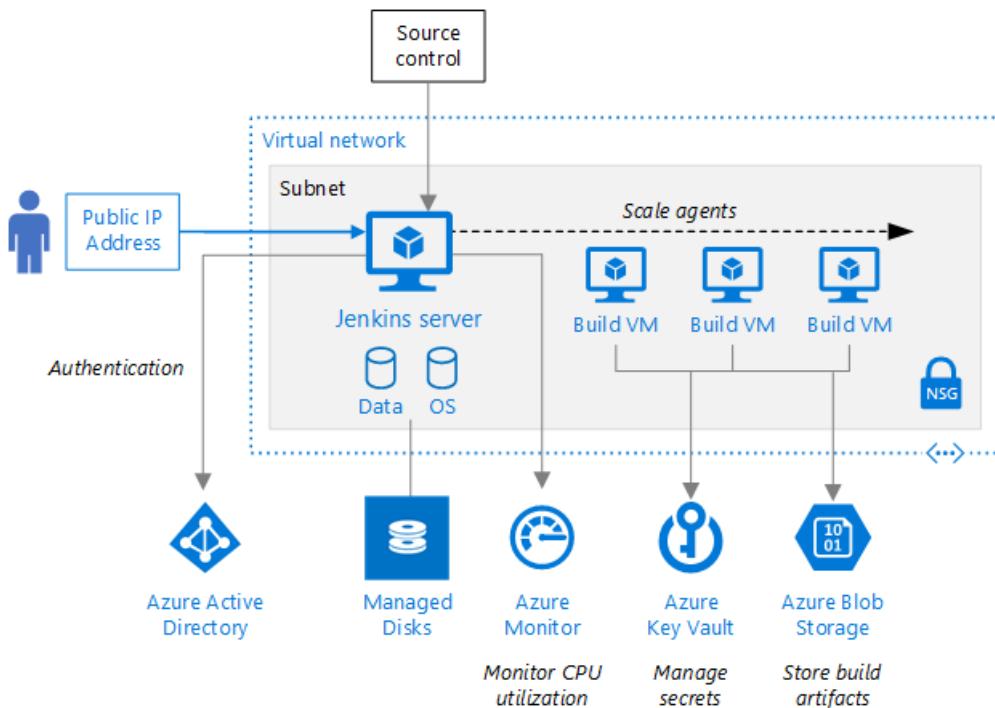
## Deploy the solution

To deploy this reference architecture, view the [GitHub readme](#).

# Run a Jenkins server on Azure

5/2/2018 • 12 minutes to read • [Edit Online](#)

This reference architecture shows how to deploy and operate a scalable, enterprise-grade Jenkins server on Azure secured with single sign-on (SSO). The architecture also uses Azure Monitor to monitor the state of the Jenkins server. [Deploy this solution.](#)



Download a [Visio file](#) that contains this architecture diagram.

This architecture supports disaster recovery with Azure services but does not cover more advanced scale-out scenarios involving multiple masters or high availability (HA) with no downtime. For general insights about the various Azure components, including a step-by-step tutorial about building out a CI/CD pipeline on Azure, see [Jenkins on Azure](#).

The focus of this document is on the core Azure operations needed to support Jenkins, including the use of Azure Storage to maintain build artifacts, the security items needed for SSO, other services that can be integrated, and scalability for the pipeline. The architecture is designed to work with an existing source control repository. For example, a common scenario is to start Jenkins jobs based on GitHub commits.

## Architecture

The architecture consists of the following components:

- **Resource group.** A [resource group](#) is used to group Azure assets so they can be managed by lifetime, owner, and other criteria. Use resource groups to deploy and monitor Azure assets as a group and track billing costs by resource group. You can also delete resources as a set, which is very useful for test deployments.
- **Jenkins server.** A virtual machine is deployed to run [Jenkins](#) as an automation server and serve as Jenkins Master. This reference architecture uses the [solution template for Jenkins on Azure](#), installed on a Linux (Ubuntu 16.04 LTS) virtual machine on Azure. Other Jenkins offerings are available in the Azure Marketplace.

#### NOTE

Nginx is installed on the VM to act as a reverse proxy to Jenkins. You can configure Nginx to enable SSL for the Jenkins server.

- **Virtual network.** A [virtual network](#) connects Azure resources to each other and provides logical isolation. In this architecture, the Jenkins server runs in a virtual network.
- **Subnets.** The Jenkins server is isolated in a [subnet](#) to make it easier to manage and segregate network traffic without impacting performance.
- **NSGs.** Use [network security groups](#) (NSGs) to restrict network traffic from the Internet to the subnet of a virtual network.
- **Managed disks.** A [managed disk](#) is a persistent virtual hard disk (VHD) used for application storage and also to maintain the state of the Jenkins server and provide disaster recovery. Data disks are stored in Azure Storage. For high performance, [premium storage](#) is recommended.
- **Azure Blob Storage.** The [Windows Azure Storage plugin](#) uses Azure Blob Storage to store the build artifacts that are created and shared with other Jenkins builds.
- **Azure Active Directory (Azure AD).** [Azure AD](#) supports user authentication, allowing you to set up SSO. Azure AD [service principals](#) define the policy and permissions for each role authorization in the workflow, using [role-based access control](#) (RBAC). Each service principal is associated with a Jenkins job.
- **Azure Key Vault.** To manage secrets and cryptographic keys used to provision Azure resources when secrets are required, this architecture uses [Key Vault](#). For added help storing secrets associated with the application in the pipeline, see also the [Azure Credentials](#) plugin for Jenkins.
- **Azure monitoring services.** This service [monitors](#) the Azure virtual machine hosting Jenkins. This deployment monitors the virtual machine status and CPU utilization and sends alerts.

## Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

### Azure AD

The [Azure AD](#) tenant for your Azure subscription is used to enable SSO for Jenkins users and set up [service principals](#) that enable Jenkins jobs to access Azure resources.

SSO authentication and authorization are implemented by the Azure AD plugin installed on the Jenkins server. SSO allows you to authenticate using your organization credentials from Azure AD when logging on to the Jenkins server. When configuring the Azure AD plugin, you can specify the level of a user's authorized access to the Jenkins server.

To provide Jenkins jobs with access to Azure resources, an Azure AD administrator creates service principals. These grant applications—in this case, the Jenkins jobs—[authenticated, authorized access](#) to Azure resources.

[RBAC](#) further defines and controls access to Azure resources for users or service principals through their assigned role. Both built-in and custom roles are supported. Roles also help secure the pipeline and ensure that a user's or agent's responsibilities are assigned and authorized correctly. In addition, RBAC can be set up to limit access to Azure assets. For example, a user can be limited to working with only the assets in a particular resource group.

### Storage

Use the Jenkins [Windows Azure Storage plugin](#), which is installed from the Azure Marketplace, to store build

artifacts that can be shared with other builds and tests. An Azure Storage account must be configured before this plugin can be used by the Jenkins jobs.

## Jenkins Azure plugins

The solution template for Jenkins on Azure installs several Azure plugins. The Azure DevOps Team builds and maintains the solution template and the following plugins, which work with other Jenkins offerings in Azure Marketplace as well as any Jenkins master set up on premises:

- [Azure AD plugin](#) allows the Jenkins server to support SSO for users based on Azure AD.
- [Azure VM Agents](#) plugin uses an Azure Resource Manager template to create Jenkins agents in Azure virtual machines.
- [Azure Credentials](#) plugin allows you to store Azure service principal credentials in Jenkins.
- [Windows Azure Storage plugin](#) uploads build artifacts to, or downloads build dependencies from, [Azure Blob storage](#).

We also recommend reviewing the growing list of all available Azure plugins that work with Azure resources. To see all the latest list, visit [Jenkins Plugin Index](#) and search for Azure. For example, the following plugins are available for deployment:

- [Azure Container Agents](#) helps you to run a container as an agent in Jenkins.
- [Kubernetes Continuous Deploy](#) deploys resource configurations to a Kubernetes cluster.
- [Azure Container Service](#) deploys configurations to Azure Container Service with Kubernetes, DC/OS with Marathon, or Docker Swarm.
- [Azure Functions](#) deploys your project to Azure Function.
- [Azure App Service](#) deploys to Azure App Service.

## Scalability considerations

Jenkins can scale to support very large workloads. For elastic builds, do not run builds on the Jenkins master server. Instead, offload build tasks to Jenkins agents, which can be elastically scaled in and out as need. Consider two options for scaling agents:

- Use the [Azure VM Agents](#) plugin to create Jenkins agents that run in Azure VMs. This plugin enables elastic scale-out for agents and can use distinct types of virtual machines. You can select a different base image from Azure Marketplace or use a custom image. For details about how the Jenkins agents scale, see [Architecting for Scale](#) in the Jenkins documentation.
- Use the [Azure Container Agents](#) plugin to run a container as an agent in either [Azure Container Service with Kubernetes](#), or [Azure Container Instances](#).

Virtual machines generally cost more to scale than containers. To use containers for scaling, however, your build process must run with containers.

Also, use Azure Storage to share build artifacts that may be used in the next stage of the pipeline by other build agents.

## Scaling the Jenkins server

You can scale the Jenkins server VM up or down by changing the VM size. The [solution template for Jenkins on Azure](#) specifies the DS2 v2 size (with two CPUs, 7 GB) by default. This size handles a small to medium team workload. Change the VM size by choosing a different option when building out the server.

Selecting the correct server size depends on the size of the expected workload. The Jenkins community maintains

a [selection guide](#) to help identify the configuration that best meets your requirements. Azure offers many [sizes for Linux VMs](#) to meet any requirements. For more information about scaling the Jenkins master, refer to the Jenkins community of [best practices](#), which also includes details about scaling Jenkins master.

## Availability considerations

Availability in the context of a Jenkins server means being able to recover any state information associated with your workflow, such as test results, libraries you have created, or other artifacts. Critical workflow state or artifacts must be maintained to recover the workflow if the Jenkins server goes down. To assess your availability requirements, consider two common metrics:

- Recovery Time Objective (RTO) specifies how long you can go without Jenkins.
- Recovery Point Objective (RPO) indicates how much data you can afford to lose if a disruption in service affects Jenkins.

In practice, RTO and RPO imply redundancy and backup. Availability is not a question of hardware recovery—that is part of Azure—but rather ensuring you maintain the state of your Jenkins server. Microsoft offers a [service level agreement](#) (SLA) for single VM instances. If this SLA doesn't meet your uptime requirements, make sure you have a plan for disaster recovery, or consider using a [multi-master Jenkins server](#) deployment (not covered in this document).

Consider using the disaster recovery [scripts](#) in step 7 of the deployment to create an Azure Storage account with managed disks to store the Jenkins server state. If Jenkins goes down, it can be restored to the state stored in this separate storage account.

## Security considerations

Use the following approaches to help lock down security on a basic Jenkins server, since in its basic state, it is not secure.

- Set up a secure way to log into the Jenkins server. This architecture uses HTTP and has a public IP, but HTTP is not secure by default. Consider setting up [HTTPS on the Nginx server](#) being used for a secure logon.

### NOTE

When adding SSL to your server, create an NSG rule for the Jenkins subnet to open port 443. For more information, see [How to open ports to a virtual machine with the Azure portal](#).

- Ensure that the Jenkins configuration prevents cross site request forgery (Manage Jenkins > Configure Global Security). This is the default for Microsoft Jenkins Server.
- Configure read-only access to the Jenkins dashboard by using the [Matrix Authorization Strategy Plugin](#).
- Install the [Azure Credentials](#) plugin to use Key Vault to handle secrets for the Azure assets, the agents in the pipeline, and third-party components.
- Use RBAC to restrict the access of the service principal to the minimum required to run the jobs. This helps limit the scope of damage from a rogue job.

Jenkins jobs often require secrets to access Azure services that require authorization, such as Azure Container Service. Use [Key Vault](#) along with the [Azure Credential plugin](#) to manage these secrets securely. Use Key Vault to store service principal credentials, passwords, tokens, and other secrets.

To get a central view of the security state of your Azure resources, use [Azure Security Center](#). Security Center monitors potential security issues and provides a comprehensive picture of the security health of your deployment.

Security Center is configured per Azure subscription. Enable security data collection as described in the [Azure Security Center quick start guide](#). When data collection is enabled, Security Center automatically scans any virtual machines created under that subscription.

The Jenkins server has its own user management system, and the Jenkins community provides best practices for [securing a Jenkins instance on Azure](#). The solution template for Jenkins on Azure implements these best practices.

## Manageability considerations

Use resource groups to organize the Azure resources that are deployed. Deploy production environments and development/test environments in separate resource groups, so that you can monitor each environment's resources and roll up billing costs by resource group. You can also delete resources as a set, which is very useful for test deployments.

Azure provides several features for [monitoring and diagnostics](#) of the overall infrastructure. To monitor CPU usage, this architecture deploys Azure Monitor. For example, you can use Azure Monitor to monitor CPU utilization, and send a notification if CPU usage exceeds 80 percent. (High CPU usage indicates that you might want to scale up the Jenkins server VM.) You can also notify a designated user if the VM fails or becomes unavailable.

## Communities

Communities can answer questions and help you set up a successful deployment. Consider the following:

- [Jenkins Community Blog](#)
- [Azure Forum](#)
- [Stack Overflow Jenkins](#)

For more best practices from the Jenkins community, visit [Jenkins best practices](#).

## Deploy the solution

To deploy this architecture, follow the steps below to install the [solution template for Jenkins on Azure](#), then install the scripts that set up monitoring and disaster recovery in the steps below.

### Prerequisites

- This reference architecture requires an Azure subscription.
- To create an Azure service principal, you must have admin rights to the Azure AD tenant that is associated with the deployed Jenkins server.
- These instructions assume that the Jenkins administrator is also an Azure user with at least Contributor privileges.

### Step 1: Deploy the Jenkins server

1. Open the [Azure Marketplace image for Jenkins](#) in your web browser and select **GET IT NOW** from the left side of the page.
2. Review the pricing details and select **Continue**, then select **Create** to configure the Jenkins server in the Azure portal.

For detailed instructions, see [Create a Jenkins server on an Azure Linux VM from the Azure portal](#). For this reference architecture, it is sufficient to get the server up and running with the admin logon. Then you can provision it to use various other services.

### Step 2: Set up SSO

The step is run by the Jenkins administrator, who must also have a user account in the subscription's Azure AD

directory and must be assigned the Contributor role.

Use the [Azure AD Plugin](#) from the Jenkins Update Center in the Jenkins server and follow the instructions to set up SSO.

### **Step 3: Provision Jenkins server with Azure VM Agent plugin**

The step is run by the Jenkins administrator to set up the Azure VM Agent plugin, which is already installed.

[Follow these steps to configure the plugin.](#) For a tutorial about setting up service principals for the plugin, see [Scale your Jenkins deployments to meet demand with Azure VM agents.](#)

### **Step 4: Provision Jenkins server with Azure Storage**

The step is run by the Jenkins administrator, who sets up the Windows Azure Storage Plugin, which is already installed.

[Follow these steps to configure the plugin.](#)

### **Step 5: Provision Jenkins server with Azure Credential plugin**

The step is run by the Jenkins administrator to set up the Azure Credential plugin, which is already installed.

[Follow these steps to configure the plugin.](#)

### **Step 6: Provision Jenkins server for monitoring by the Azure Monitor Service**

To set up monitoring for your Jenkins server, follow the instructions in [Create metric alerts in Azure Monitor for Azure services.](#)

### **Step 7: Provision Jenkins server with Managed Disks for disaster recovery**

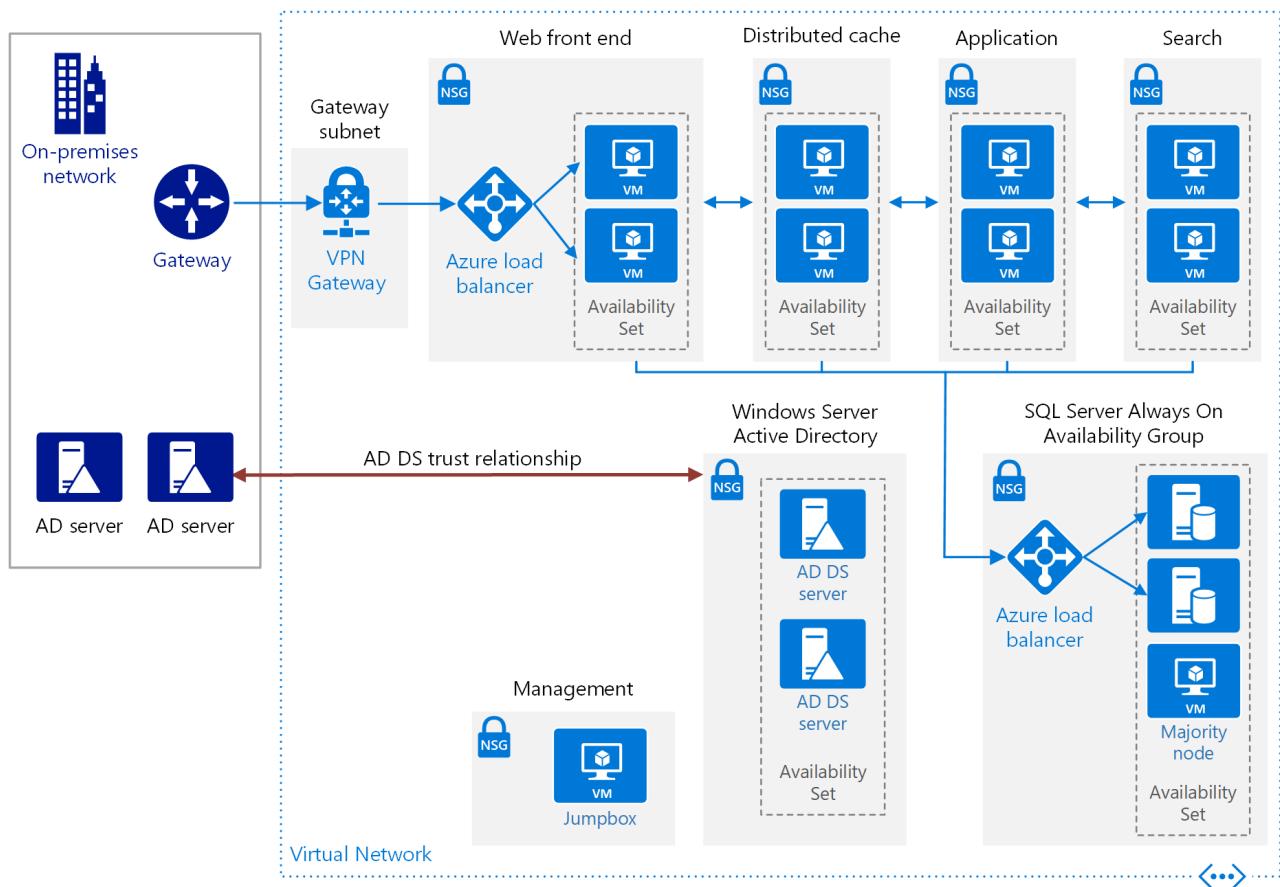
The Microsoft Jenkins product group has created disaster recovery scripts that build a managed disk used to save the Jenkins state. If the server goes down, it can be restored to its latest state.

Download and run the disaster recovery scripts from [GitHub](#).

# Run a high availability SharePoint Server 2016 farm in Azure

9/28/2018 • 13 minutes to read • [Edit Online](#)

This reference architecture shows a set of proven practices for setting up a high availability SharePoint Server 2016 farm on Azure, using MinRole topology and SQL Server Always On availability groups. The SharePoint farm is deployed in a secured virtual network with no Internet-facing endpoint or presence. [Deploy this solution.](#)



[Download a Visio file of this architecture.](#)

## Architecture

This architecture builds on the one shown in [Run Windows VMs for an N-tier application](#). It deploys a SharePoint Server 2016 farm with high availability inside an Azure virtual network (VNet). This architecture is suitable for a test or production environment, a SharePoint hybrid infrastructure with Office 365, or as the basis for a disaster recovery scenario.

The architecture consists of the following components:

- **Resource groups.** A [resource group](#) is a container that holds related Azure resources. One resource group is used for the SharePoint servers, and another resource group is used for infrastructure components that are independent of VMs, such as the virtual network and load balancers.
- **Virtual network (VNet).** The VMs are deployed in a VNet with a unique intranet address space. The VNet is further subdivided into subnets.
- **Virtual machines (VMs).** The VMs are deployed into the VNet, and private static IP addresses are

assigned to all of the VMs. Static IP addresses are recommended for the VMs running SQL Server and SharePoint Server 2016, to avoid issues with IP address caching and changes of addresses after a restart.

- **Availability sets.** Place the VMs for each SharePoint role into separate [availability sets](#), and provision at least two virtual machines (VMs) for each role. This makes the VMs eligible for a higher service level agreement (SLA).
- **Internal load balancer.** The [load balancer](#) distributes SharePoint request traffic from the on-premises network to the front-end web servers of the SharePoint farm.
- **Network security groups (NSGs).** For each subnet that contains virtual machines, a [network security group](#) is created. Use NSGs to restrict network traffic within the VNet, in order to isolate subnets.
- **Gateway.** The gateway provides a connection between your on-premises network and the Azure virtual network. Your connection can use ExpressRoute or site-to-site VPN. For more information, see [Connect an on-premises network to Azure](#).
- **Windows Server Active Directory (AD) domain controllers.** This reference architecture deploys Windows Server AD domain controllers. These domain controllers run in the Azure VNet and have a trust relationship with the on-premises Windows Server AD forest. Client web requests for SharePoint farm resources are authenticated in the VNet rather than sending that authentication traffic across the gateway connection to the on-premises network. In DNS, intranet A or CNAME records are created so that intranet users can resolve the name of the SharePoint farm to the private IP address of the internal load balancer.

SharePoint Server 2016 also supports using [Azure Active Directory Domain Services](#). Azure AD Domain Services provides managed domain services, so that you don't need to deploy and manage domain controllers in Azure.

- **SQL Server Always On Availability Group.** For high availability of the SQL Server database, we recommend [SQL Server Always On Availability Groups](#). Two virtual machines are used for SQL Server. One contains the primary database replica and the other contains the secondary replica.
- **Majority node VM.** This VM allows the failover cluster to establish quorum. For more information, see [Understanding Quorum Configurations in a Failover Cluster](#).
- **SharePoint servers.** The SharePoint servers perform the web front-end, caching, application, and search roles.
- **Jumpbox.** Also called a [bastion host](#). This is a secure VM on the network that administrators use to connect to the other VMs. The jumpbox has an NSG that allows remote traffic only from public IP addresses on a safe list. The NSG should permit remote desktop (RDP) traffic.

## Recommendations

Your requirements might differ from the architecture described here. Use these recommendations as a starting point.

### Resource group recommendations

We recommend separating resource groups according to the server role, and having a separate resource group for infrastructure components that are global resources. In this architecture, the SharePoint resources form one group, while the SQL Server and other utility assets form another.

### Virtual network and subnet recommendations

Use one subnet for each SharePoint role, plus a subnet for the gateway and one for the jumpbox.

The gateway subnet must be named *GatewaySubnet*. Assign the gateway subnet address space from the last part of the virtual network address space. For more information, see [Connect an on-premises network to Azure using a](#)

VPN gateway.

## VM recommendations

This architecture requires a minimum of 44 cores:

- 8 SharePoint servers on Standard\_DS3\_v2 (4 cores each) = 32 cores
- 2 Active Directory domain controllers on Standard\_DS1\_v2 (1 core each) = 2 cores
- 2 SQL Server VMs on Standard\_DS3\_v2 = 8 cores
- 1 majority node on Standard\_DS1\_v2 = 1 core
- 1 management server on Standard\_DS1\_v2 = 1 core

Make sure your Azure subscription has enough VM core quota for the deployment, or the deployment will fail. See [Azure subscription and service limits, quotas, and constraints](#).

For all SharePoint roles except the Search Indexer, we recommend using the [Standard\\_DS3\\_v2](#) VM size. The Search Indexer should be at least the [Standard\\_DS13\\_v2](#) size. For testing, the parameter files for this reference architecture specify the smaller DS3\_v2 size for the Search Indexer role. For a production deployment, update the parameter files to use the DS13 size or larger. For more information, see [Hardware and software requirements for SharePoint Server 2016](#).

For the SQL Server VMs, we recommend a minimum of 4 cores and 8 GB RAM. The parameter files for this reference architecture specify the DS3\_v2 size. For a production deployment, you might need to specify a larger VM size. For more information, see [Storage and SQL Server capacity planning and configuration \(SharePoint Server\)](#).

## NSG recommendations

We recommend having one NSG for each subnet that contains VMs, to enable subnet isolation. If you want to configure subnet isolation, add NSG rules that define the allowed or denied inbound or outbound traffic for each subnet. For more information, see [Filter network traffic with network security groups](#).

Do not assign an NSG to the gateway subnet, or the gateway will stop functioning.

## Storage recommendations

The storage configuration of the VMs in the farm should match the appropriate best practices used for on-premises deployments. SharePoint servers should have a separate disk for logs. SharePoint servers hosting search index roles require additional disk space for the search index to be stored. For SQL Server, the standard practice is to separate data and logs. Add more disks for database backup storage, and use a separate disk for [tempdb](#).

For best reliability, we recommend using [Azure Managed Disks](#). Managed disks ensure that the disks for VMs within an availability set are isolated to avoid single points of failure.

### NOTE

Currently the Resource Manager template for this reference architecture does not use managed disks. We are planning to update the template to use managed disks.

Use Premium managed disks for all SharePoint and SQL Server VMs. You can use Standard managed disks for the majority node server, the domain controllers, and the management server.

## SharePoint Server recommendations

Before configuring the SharePoint farm, make sure you have one Windows Server Active Directory service account per service. For this architecture, you need at a minimum the following domain-level accounts to isolate privilege per role:

- SQL Server Service account

- Setup User account
- Server Farm account
- Search Service account
- Content Access account
- Web App Pool accounts
- Service App Pool accounts
- Cache Super User account
- Cache Super Reader account

To meet the support requirement for disk throughput of 200 MB per second minimum, make sure to plan the Search architecture. See [Plan enterprise search architecture in SharePoint Server 2013](#). Also follow the guidelines in [Best practices for crawling in SharePoint Server 2016](#).

In addition, store the search component data on a separate storage volume or partition with high performance. To reduce load and improve throughput, configure the object cache user accounts, which are required in this architecture. Split the Windows Server operating system files, the SharePoint Server 2016 program files, and diagnostics logs across three separate storage volumes or partitions with normal performance.

For more information about these recommendations, see [Initial deployment administrative and service accounts in SharePoint Server 2016](#).

### **Hybrid workloads**

This reference architecture deploys a SharePoint Server 2016 farm that can be used as a [SharePoint hybrid environment](#) — that is, extending SharePoint Server 2016 to Office 365 SharePoint Online. If you have Office Online Server, see [Office Web Apps and Office Online Server supportability in Azure](#).

The default service applications in this deployment are designed to support hybrid workloads. All SharePoint Server 2016 and Office 365 hybrid workloads can be deployed to this farm without changes to the SharePoint infrastructure, with one exception: The Cloud Hybrid Search Service Application must not be deployed onto servers hosting an existing search topology. Therefore, one or more search-role-based VMs must be added to the farm to support this hybrid scenario.

### **SQL Server Always On Availability Groups**

This architecture uses SQL Server virtual machines because SharePoint Server 2016 cannot use Azure SQL Database. To support high availability in SQL Server, we recommend using Always On Availability Groups, which specify a set of databases that fail over together, making them highly-available and recoverable. In this reference architecture, the databases are created during deployment, but you must manually enable Always On Availability Groups and add the SharePoint databases to an availability group. For more information, see [Create the availability group and add the SharePoint databases](#).

We also recommend adding a listener IP address to the cluster, which is the private IP address of the internal load balancer for the SQL Server virtual machines.

For recommended VM sizes and other performance recommendations for SQL Server running in Azure, see [Performance best practices for SQL Server in Azure Virtual Machines](#). Also follow the recommendations in [Best practices for SQL Server in a SharePoint Server 2016 farm](#).

We recommend that the majority node server reside on a separate computer from the replication partners. The server enables the secondary replication partner server in a high-safety mode session to recognize whether to initiate an automatic failover. Unlike the two partners, the majority node server doesn't serve the database but rather supports automatic failover.

## **Scalability considerations**

To scale up the existing servers, simply change the VM size.

With the [MinRoles](#) capability in SharePoint Server 2016, you can scale out servers based on the server's role and also remove servers from a role. When you add servers to a role, you can specify any of the single roles or one of the combined roles. If you add servers to the Search role, however, you must also reconfigure the search topology using PowerShell. You can also convert roles using MinRoles. For more information, see [Managing a MinRole Server Farm in SharePoint Server 2016](#).

Note that SharePoint Server 2016 doesn't support using virtual machine scale sets for auto-scaling.

## Availability considerations

This reference architecture supports high availability within an Azure region, because each role has at least two VMs deployed in an availability set.

To protect against a regional failure, create a separate disaster recovery farm in a different Azure region. Your recovery time objectives (RTOs) and recovery point objectives (RPOs) will determine the setup requirements. For details, see [Choose a disaster recovery strategy for SharePoint 2016](#). The secondary region should be a *paired region* with the primary region. In the event of a broad outage, recovery of one region is prioritized out of every pair. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

## Manageability considerations

To operate and maintain servers, server farms, and sites, follow the recommended practices for SharePoint operations. For more information, see [Operations for SharePoint Server 2016](#).

The tasks to consider when managing SQL Server in a SharePoint environment may differ from the ones typically considered for a database application. A best practice is to fully back up all SQL databases weekly with incremental nightly backups. Back up transaction logs every 15 minutes. Another practice is to implement SQL Server maintenance tasks on the databases while disabling the built-in SharePoint ones. For more information, see [Storage and SQL Server capacity planning and configuration](#).

## Security considerations

The domain-level service accounts used to run SharePoint Server 2016 require Windows Server AD domain controllers for domain-join and authentication processes. Azure Active Directory Domain Services can't be used for this purpose. To extend the Windows Server AD identity infrastructure already in place in the intranet, this architecture uses two Windows Server AD replica domain controllers of an existing on-premises Windows Server AD forest.

In addition, it's always wise to plan for security hardening. Other recommendations include:

- Add rules to NSGs to isolate subnets and roles.
- Don't assign public IP addresses to VMs.
- For intrusion detection and analysis of payloads, consider using a network virtual appliance in front of the front-end web servers instead of an internal Azure load balancer.
- As an option, use IPsec policies for encryption of cleartext traffic between servers. If you are also doing subnet isolation, update your network security group rules to allow IPsec traffic.
- Install anti-malware agents for the VMs.

## Deploy the solution

A deployment for this reference architecture is available on [GitHub](#). The entire deployment can take several hours to complete.

The deployment creates the following resource groups in your subscription:

- ra-onprem-sp2016-rg
- ra-sp2016-network-rg

The template parameter files refer to these names, so if you change them, update the parameter files to match.

The parameter files include a hard-coded password in various places. Change these values before you deploy.

## Prerequisites

1. Clone, fork, or download the zip file for the [reference architectures](#) GitHub repository.
2. Install [Azure CLI 2.0](#).
3. Install the [Azure building blocks](#) npm package.

```
npm install -g @mspnp/azure-building-blocks
```

4. From a command prompt, bash prompt, or PowerShell prompt, sign into your Azure account as follows:

```
az login
```

## Deploy the solution

1. Run the following command to deploy a simulated on-premises network.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p onprem.json --deploy
```

2. Run the following command to deploy the Azure VNet and the VPN gateway.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p connections.json --deploy
```

3. Run the following command to deploy the jumpbox, AD domain controllers, and SQL Server VMs.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p azure1.json --deploy
```

4. Run the following command to create the failover cluster and the availability group.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p azure2-cluster.json --deploy
```

5. Run the following command to deploy the remaining VMs.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p azure3.json --deploy
```

At this point, verify that you can make a TCP connection from the web front end to the load balancer for the SQL Server Always On availability group. To do so, perform the following steps:

1. Use the Azure portal to find the VM named `ra-sp-jb-vm1` in the `ra-sp2016-network-rg` resource group. This is the jumpbox VM.
2. Click `Connect` to open a remote desktop session to the VM. Use the password that you specified in the `azure1.json` parameter file.
3. From the Remote Desktop session, log into 10.0.5.4. This is the IP address of the VM named `ra-sp-app-vm1`.

4. Open a PowerShell console in the VM, and use the `Test-NetConnection` cmdlet to verify that you can connect to the load balancer.

```
Test-NetConnection 10.0.3.100 -Port 1433
```

The output should look similar to the following:

```
ComputerName      : 10.0.3.100
RemoteAddress     : 10.0.3.100
RemotePort        : 1433
InterfaceAlias    : Ethernet 3
SourceAddress     : 10.0.0.132
TcpTestSucceeded  : True
```

If it fails, use the Azure Portal to restart the VM named `ra-sp-sql-vm2`. After the VM restarts, run the `Test-NetConnection` command again. You may need to wait about a minute after the VM restarts for the connection to succeed.

Now complete the deployment as follows.

1. Run the following command to deploy the SharePoint farm primary node.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p azure4-sharepoint-server.json --deploy
```

2. Run the following command to deploy the SharePoint cache, search, and web.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p azure5-sharepoint-farm.json --deploy
```

3. Run the following command to create the NSG rules.

```
azbb -s <subscription_id> -g ra-onprem-sp2016-rg -l <location> -p azure6-security.json --deploy
```

## Validate the deployment

1. In the [Azure portal](#), navigate to the `ra-onprem-sp2016-rg` resource group.
2. In the list of resources, select the VM resource named `ra-onprem-sp2016-rg`.
3. Connect to the VM, as described in [Connect to virtual machine](#). The user name is `\onpremuser`.
4. When the remote connection to the VM is established, open a browser in the VM and navigate to <http://portal.contoso.local>.
5. In the **Windows Security** box, log on to the SharePoint portal using `contoso.local\testuser` for the user name.

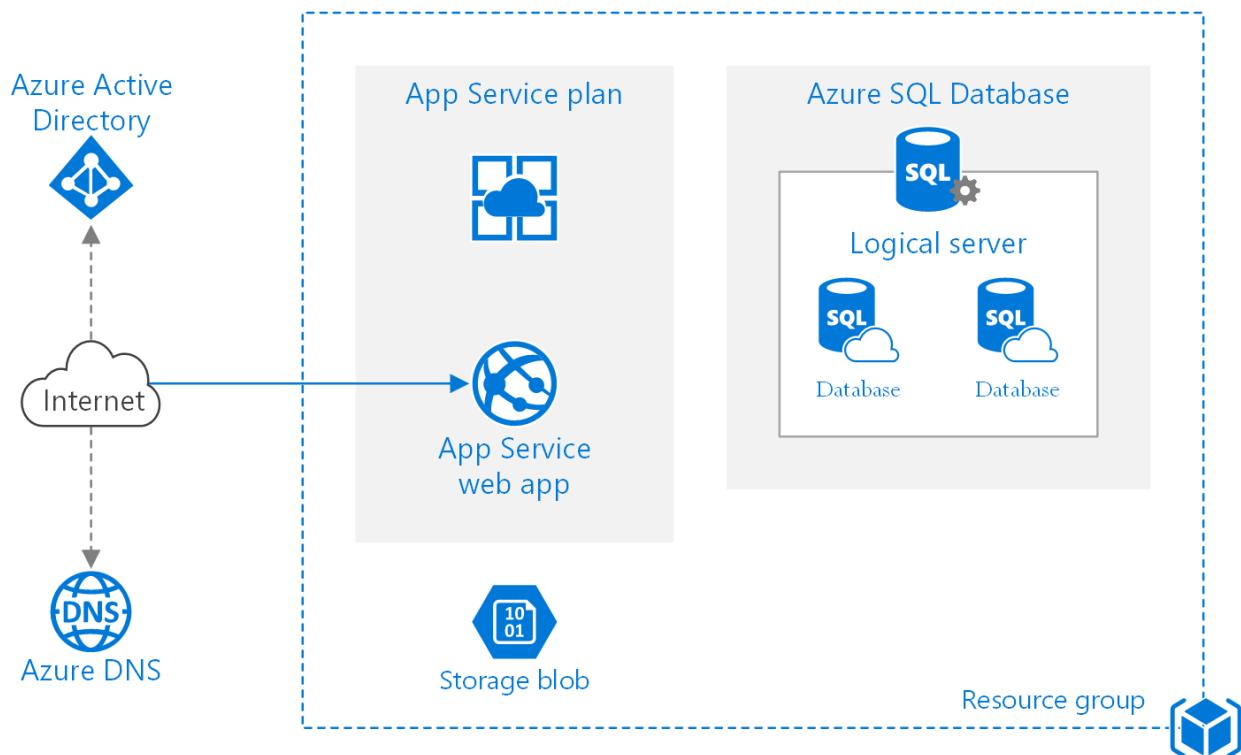
This logon tunnels from the Fabrikam.com domain used by the on-premises network to the contoso.local domain used by the SharePoint portal. When the SharePoint site opens, you'll see the root demo site.

**Contributors to this reference architecture** — Joe Davies, Bob Fox, Neil Hodgkinson, Paul Stork

# Basic web application

9/28/2018 • 12 minutes to read • [Edit Online](#)

This reference architecture shows a set of proven practices for a web application that uses [Azure App Service](#) and [Azure SQL Database](#). [Deploy this solution.](#)



[Download a Visio file](#) of this architecture.

## Architecture

### NOTE

This architecture does not focus on application development, and does not assume any particular application framework. The goal is to understand how various Azure services fit together.

The architecture has the following components:

- **Resource group**. A [resource group](#) is a logical container for Azure resources.
- **App Service app**. [Azure App Service](#) is a fully managed platform for creating and deploying cloud applications.
- **App Service plan**. An [App Service plan](#) provides the managed virtual machines (VMs) that host your app. All apps associated with a plan run on the same VM instances.
- **Deployment slots**. A [deployment slot](#) lets you stage a deployment and then swap it with the production deployment. That way, you avoid deploying directly into production. See the [Manageability](#) section for specific recommendations.
- **IP address**. The App Service app has a public IP address and a domain name. The domain name is a subdomain of `azurewebsites.net`, such as `contoso.azurewebsites.net`.

- **Azure DNS.** [Azure DNS](#) is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services. To use a custom domain name (such as `contoso.com`) create DNS records that map the custom domain name to the IP address. For more information, see [Configure a custom domain name in Azure App Service](#).
- **Azure SQL Database.** [SQL Database](#) is a relational database-as-a-service in the cloud. SQL Database shares its code base with the Microsoft SQL Server database engine. Depending on your application requirements, you can also use [Azure Database for MySQL](#) or [Azure Database for PostgreSQL](#). These are fully managed database services, based on the open source MySQL Server and Postgres database engines, respectively.
- **Logical server.** In Azure SQL Database, a logical server hosts your databases. You can create multiple databases per logical server.
- **Azure Storage.** Create an Azure storage account with a blob container to store diagnostic logs.
- **Azure Active Directory** (Azure AD). Use Azure AD or another identity provider for authentication.

## Recommendations

Your requirements might differ from the architecture described here. Use the recommendations in this section as a starting point.

### App Service plan

Use the Standard or Premium tiers, because they support scale out, autoscale, and secure sockets layer (SSL). Each tier supports several *instance sizes* that differ by number of cores and memory. You can change the tier or instance size after you create a plan. For more information about App Service plans, see [App Service Pricing](#).

You are charged for the instances in the App Service plan, even if the app is stopped. Make sure to delete plans that you aren't using (for example, test deployments).

### SQL Database

Use the [V12 version](#) of SQL Database. SQL Database supports Basic, Standard, and Premium [service tiers](#), with multiple performance levels within each tier measured in [Database Transaction Units \(DTUs\)](#). Perform capacity planning and choose a tier and performance level that meets your requirements.

### Region

Provision the App Service plan and the SQL Database in the same region to minimize network latency. Generally, choose the region closest to your users.

The resource group also has a region, which specifies where deployment metadata is stored. Put the resource group and its resources in the same region. This can improve availability during deployment.

## Scalability considerations

A major benefit of Azure App Service is the ability to scale your application based on load. Here are some considerations to keep in mind when planning to scale your application.

### Scaling the App Service app

There are two ways to scale an App Service app:

- *Scale up*, which means changing the instance size. The instance size determines the memory, number of cores, and storage on each VM instance. You can scale up manually by changing the instance size or the plan tier.
- *Scale out*, which means adding instances to handle increased load. Each pricing tier has a maximum

number of instances.

You can scale out manually by changing the instance count, or use [autoscaling](#) to have Azure automatically add or remove instances based on a schedule and/or performance metrics. Each scale operation happens quickly—typically within seconds.

To enable autoscaling, create an autoscale *profile* that defines the minimum and maximum number of instances. Profiles can be scheduled. For example, you might create separate profiles for weekdays and weekends. Optionally, a profile contains rules for when to add or remove instances. (Example: Add two instances if CPU usage is above 70% for 5 minutes.)

Recommendations for scaling a web app:

- As much as possible, avoid scaling up and down, because it may trigger an application restart. Instead, select a tier and size that meet your performance requirements under typical load and then scale out the instances to handle changes in traffic volume.
- Enable autoscaling. If your application has a predictable, regular workload, create profiles to schedule the instance counts ahead of time. If the workload is not predictable, use rule-based autoscaling to react to changes in load as they occur. You can combine both approaches.
- CPU usage is generally a good metric for autoscale rules. However, you should load test your application, identify potential bottlenecks, and base your autoscale rules on that data.
- Autoscale rules include a *cool-down* period, which is the interval to wait after a scale action has completed before starting a new scale action. The cool-down period lets the system stabilize before scaling again. Set a shorter cool-down period for adding instances, and a longer cool-down period for removing instances. For example, set 5 minutes to add an instance, but 60 minutes to remove an instance. It's better to add new instances quickly under heavy load to handle the additional traffic, and then gradually scale back.

## Scaling SQL Database

If you need a higher service tier or performance level for SQL Database, you can scale up individual databases with no application downtime. For more information, see [SQL Database options and performance: Understand what's available in each service tier](#).

## Availability considerations

At the time of writing, the service level agreement (SLA) for App Service is 99.95% and the SLA for SQL Database is 99.99% for Basic, Standard, and Premium tiers.

### NOTE

The App Service SLA applies to both single and multiple instances.

## Backups

In the event of data loss, SQL Database provides point-in-time restore and geo-restore. These features are available in all tiers and are automatically enabled. You don't need to schedule or manage the backups.

- Use point-in-time restore to [recover from human error](#) by returning the database to an earlier point in time.
- Use geo-restore to [recover from a service outage](#) by restoring a database from a geo-redundant backup.

For more information, see [Cloud business continuity and database disaster recovery with SQL Database](#).

App Service provides a [backup and restore](#) feature for your application files. However, be aware that the backed-up files include app settings in plain text and these may include secrets, such as connection strings. Avoid using the App Service backup feature to back up your SQL databases because it exports the database to a SQL .bacpac file, consuming [DTUs](#). Instead, use SQL Database point-in-time restore described above.

# Manageability considerations

Create separate resource groups for production, development, and test environments. This makes it easier to manage deployments, delete test deployments, and assign access rights.

When assigning resources to resource groups, consider the following:

- Lifecycle. In general, put resources with the same lifecycle into the same resource group.
- Access. You can use [role-based access control](#) (RBAC) to apply access policies to the resources in a group.
- Billing. You can view the rolled-up costs for the resource group.

For more information, see [Azure Resource Manager overview](#).

## Deployment

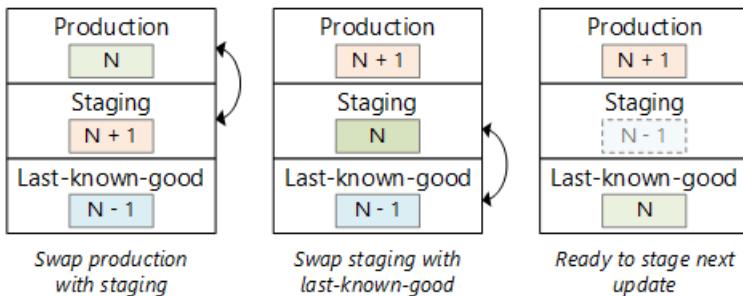
Deployment involves two steps:

1. Provisioning the Azure resources. We recommend that you use [Azure Resource Manager templates](#) for this step. Templates make it easier to automate deployments via PowerShell or the Azure command line interface (CLI).
2. Deploying the application (code, binaries, and content files). You have several options, including deploying from a local Git repository, using Visual Studio, or continuous deployment from cloud-based source control. See [Deploy your app to Azure App Service](#).

An App Service app always has one deployment slot named `production`, which represents the live production site. We recommend creating a staging slot for deploying updates. The benefits of using a staging slot include:

- You can verify the deployment succeeded, before swapping it into production.
- Deploying to a staging slot ensures that all instances are warmed up before being swapped into production. Many applications have a significant warmup and cold-start time.

We also recommend creating a third slot to hold the last-known-good deployment. After you swap staging and production, move the previous production deployment (which is now in staging) into the last-known-good slot. That way, if you discover a problem later, you can quickly revert to the last-known-good version.



If you revert to a previous version, make sure any database schema changes are backward compatible.

Don't use slots on your production deployment for testing because all apps within the same App Service plan share the same VM instances. For example, load tests might degrade the live production site. Instead, create separate App Service plans for production and test. By putting test deployments into a separate plan, you isolate them from the production version.

## Configuration

Store configuration settings as [app settings](#). Define the app settings in your Resource Manager templates, or using PowerShell. At runtime, app settings are available to the application as environment variables.

Never check passwords, access keys, or connection strings into source control. Instead, pass these as parameters to a deployment script that stores these values as app settings.

When you swap a deployment slot, the app settings are swapped by default. If you need different settings for production and staging, you can create app settings that stick to a slot and don't get swapped.

## Diagnostics and monitoring

Enable [diagnostics logging](#), including application logging and web server logging. Configure logging to use Blob storage. For performance reasons, create a separate storage account for diagnostic logs. Don't use the same storage account for logs and application data. For more detailed guidance on logging, see [Monitoring and diagnostics guidance](#).

Use a service such as [New Relic](#) or [Application Insights](#) to monitor application performance and behavior under load. Be aware of the [data rate limits](#) for Application Insights.

Perform load testing, using a tool such as [Azure DevOps](#) or [Visual Studio Team Foundation Server](#). For a general overview of performance analysis in cloud applications, see [Performance Analysis Primer](#).

Tips for troubleshooting your application:

- Use the [troubleshoot blade](#) in the Azure portal to find solutions to common problems.
- Enable [log streaming](#) to see logging information in near-real time.
- The [Kudu dashboard](#) has several tools for monitoring and debugging your application. For more information, see [Azure Websites online tools you should know about](#) (blog post). You can reach the Kudu dashboard from the Azure portal. Open the blade for your app and click **Tools**, then click **Kudu**.
- If you use Visual Studio, see the article [Troubleshoot a web app in Azure App Service using Visual Studio](#) for debugging and troubleshooting tips.

## Security considerations

This section lists security considerations that are specific to the Azure services described in this article. It's not a complete list of security best practices. For some additional security considerations, see [Secure an app in Azure App Service](#).

### SQL Database auditing

Auditing can help you maintain regulatory compliance and get insight into discrepancies and irregularities that could indicate business concerns or suspected security violations. See [Get started with SQL database auditing](#).

### Deployment slots

Each deployment slot has a public IP address. Secure the nonproduction slots using [Azure Active Directory login](#) so that only members of your development and DevOps teams can reach those endpoints.

### Logging

Logs should never record users' passwords or other information that might be used to commit identity fraud. Scrub those details from the data before storing it.

### SSL

An App Service app includes an SSL endpoint on a subdomain of `azurewebsites.net` at no additional cost. The SSL endpoint includes a wildcard certificate for the `*.azurewebsites.net` domain. If you use a custom domain name, you must provide a certificate that matches the custom domain. The simplest approach is to buy a certificate directly through the Azure portal. You can also import certificates from other certificate authorities. For more information, see [Buy and Configure an SSL Certificate for your Azure App Service](#).

As a security best practice, your app should enforce HTTPS by redirecting HTTP requests. You can implement this inside your application or use a URL rewrite rule as described in [Enable HTTPS for an app in Azure App Service](#).

### Authentication

We recommend authenticating through an identity provider (IDP), such as Azure AD, Facebook, Google, or Twitter. Use OAuth 2 or OpenID Connect (OIDC) for the authentication flow. Azure AD provides functionality to manage users and groups, create application roles, integrate your on-premises identities, and consume backend services such as Office 365 and Skype for Business.

Avoid having the application manage user logins and credentials directly, as it creates a potential attack surface. At a minimum, you would need to have email confirmation, password recovery, and multi-factor authentication; validate password strength; and store password hashes securely. The large identity providers handle all of those things for you, and are constantly monitoring and improving their security practices.

Consider using [App Service authentication](#) to implement the OAuth/OIDC authentication flow. The benefits of App Service authentication include:

- Easy to configure.
- No code is required for simple authentication scenarios.
- Supports delegated authorization using OAuth access tokens to consume resources on behalf of the user.
- Provides a built-in token cache.

Some limitations of App Service authentication:

- Limited customization options.
- Delegated authorization is restricted to one backend resource per login session.
- If you use more than one IDP, there is no built-in mechanism for home realm discovery.
- For multi-tenant scenarios, the application must implement the logic to validate the token issuer.

## Deploy the solution

An example Resource Manager template for this architecture is [available on GitHub](#).

To deploy the template using PowerShell, run the following commands:

```
New-AzureRmResourceGroup -Name <resource-group-name> -Location "West US"

$parameters = @{"appName"=<app-name>;"environment"="dev";"locationShort"="uw";"databaseName"="app-
db";"administratorLogin"=<admin>;"administratorLoginPassword"=<password>"}

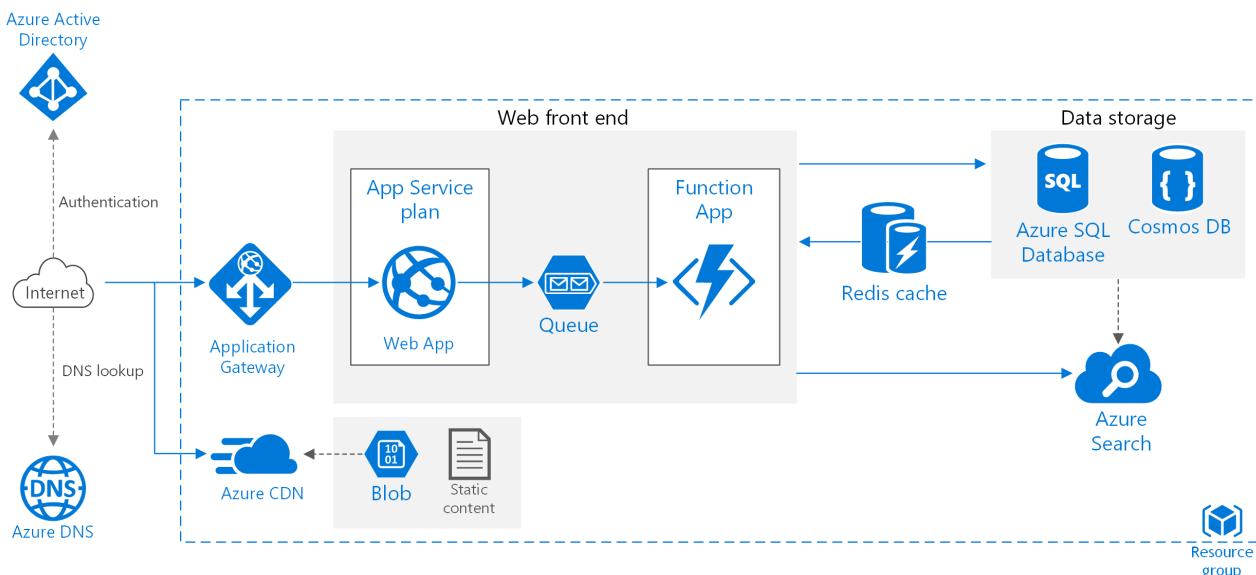
New-AzureRmResourceGroupDeployment -Name <deployment-name> -ResourceGroupName <resource-group-name> -
TemplateFile .\PaaS-Basic.json -TemplateParameterObject $parameters
```

For more information, see [Deploy resources with Azure Resource Manager templates](#).

# Improve scalability in a web application

10/26/2018 • 5 minutes to read • [Edit Online](#)

This reference architecture shows proven practices for improving scalability and performance in an Azure App Service web application.



Download a [Visio file](#) of this architecture.

## Architecture

This architecture builds on the one shown in [Basic web application](#). It includes the following components:

- **Resource group.** A [resource group](#) is a logical container for Azure resources.
- **Web app.** A typical modern application might include both a website and one or more RESTful web APIs. A web API might be consumed by browser clients through AJAX, by native client applications, or by server-side applications. For considerations on designing web APIs, see [API design guidance](#).
- **Function App.** Use [Function Apps](#) to run background tasks. Functions are invoked by a trigger, such as a timer event or a message being placed on queue. For long-running stateful tasks, use [Durable Functions](#).
- **Queue.** In the architecture shown here, the application queues background tasks by putting a message onto an [Azure Queue storage](#) queue. The message triggers a function app. Alternatively, you can use Service Bus queues. For a comparison, see [Azure Queues and Service Bus queues - compared and contrasted](#).
- **Cache.** Store semi-static data in [Azure Redis Cache](#).
- **CDN.** Use [Azure Content Delivery Network \(CDN\)](#) to cache publicly available content for lower latency and faster delivery of content.
- **Data storage.** Use [Azure SQL Database](#) for relational data. For non-relational data, consider [Cosmos DB](#).
- **Azure Search.** Use [Azure Search](#) to add search functionality such as search suggestions, fuzzy search, and language-specific search. Azure Search is typically used in conjunction with another data store, especially if the primary data store requires strict consistency. In this approach, store authoritative data in the other data store and the search index in Azure Search. Azure Search can also be used to consolidate a single search index from multiple data stores.
- **Azure DNS.** [Azure DNS](#) is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.

- **Application gateway.** Application Gateway is a layer 7 load balancer. In this architecture, it routes HTTP requests to the web front end. Application Gateway also provides a [web application firewall](#) (WAF) that protects the application from common exploits and vulnerabilities.

## Recommendations

Your requirements might differ from the architecture described here. Use the recommendations in this section as a starting point.

### App Service apps

We recommend creating the web application and the web API as separate App Service apps. This design lets you run them in separate App Service plans so they can be scaled independently. If you don't need that level of scalability initially, you can deploy the apps into the same plan and move them into separate plans later if necessary.

#### NOTE

For the Basic, Standard, and Premium plans, you are billed for the VM instances in the plan, not per app. See [App Service Pricing](#)

### Cache

You can improve performance and scalability by using [Azure Redis Cache](#) to cache some data. Consider using Redis Cache for:

- Semi-static transaction data.
- Session state.
- HTML output. This can be useful in applications that render complex HTML output.

For more detailed guidance on designing a caching strategy, see [Caching guidance](#).

### CDN

Use [Azure CDN](#) to cache static content. The main benefit of a CDN is to reduce latency for users, because content is cached at an edge server that is geographically close to the user. CDN can also reduce load on the application, because that traffic is not being handled by the application.

If your app consists mostly of static pages, consider using [CDN to cache the entire app](#). Otherwise, put static content such as images, CSS, and HTML files, into [Azure Storage](#) and use [CDN to cache those files](#).

#### NOTE

Azure CDN cannot serve content that requires authentication.

For more detailed guidance, see [Content Delivery Network \(CDN\) guidance](#).

### Storage

Modern applications often process large amounts of data. In order to scale for the cloud, it's important to choose the right storage type. Here are some baseline recommendations.

| WHAT YOU WANT TO STORE | EXAMPLE                                | RECOMMENDED STORAGE |
|------------------------|----------------------------------------|---------------------|
| Files                  | Images, documents, PDFs                | Azure Blob Storage  |
| Key/Value pairs        | User profile data looked up by user ID | Azure Table storage |

| WHAT YOU WANT TO STORE                                                                   | EXAMPLE           | RECOMMENDED STORAGE                                                    |
|------------------------------------------------------------------------------------------|-------------------|------------------------------------------------------------------------|
| Short messages intended to trigger further processing                                    | Order requests    | Azure Queue storage, Service Bus queue, or Service Bus topic           |
| Non-relational data with a flexible schema requiring basic querying                      | Product catalog   | Document database, such as Azure Cosmos DB, MongoDB, or Apache CouchDB |
| Relational data requiring richer query support, strict schema, and/or strong consistency | Product inventory | Azure SQL Database                                                     |

See [Choose the right data store](#).

## Scalability considerations

A major benefit of Azure App Service is the ability to scale your application based on load. Here are some considerations to keep in mind when planning to scale your application.

### App Service app

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.

Similarly, consider putting a function app into its own plan so that background tasks don't run on the same instances that handle HTTP requests. If background tasks run intermittently, consider using a [consumption plan](#), which is billed based on the number of executions, rather than hourly.

### SQL Database

Increase scalability of a SQL database by *sharding* the database. Sharding refers to partitioning the database horizontally. Sharding allows you to scale out the database horizontally using [Elastic Database tools](#). Potential benefits of sharding include:

- Better transaction throughput.
- Queries can run faster over a subset of the data.

### Azure Search

Azure Search removes the overhead of performing complex data searches from the primary data store, and it can scale to handle load. See [Scale resource levels for query and indexing workloads in Azure Search](#).

## Security considerations

This section lists security considerations that are specific to the Azure services described in this article. It's not a complete list of security best practices. For some additional security considerations, see [Secure an app in Azure App Service](#).

### Cross-Origin Resource Sharing (CORS)

If you create a website and web API as separate apps, the website cannot make client-side AJAX calls to the API unless you enable CORS.

#### NOTE

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the same-origin policy, and prevents a malicious site from reading sensitive data from another site. CORS is a W3C standard that allows a server to relax the same-origin policy and allow some cross-origin requests while rejecting others.

App Services has built-in support for CORS, without needing to write any application code. See [Consume an API app from JavaScript using CORS](#). Add the website to the list of allowed origins for the API.

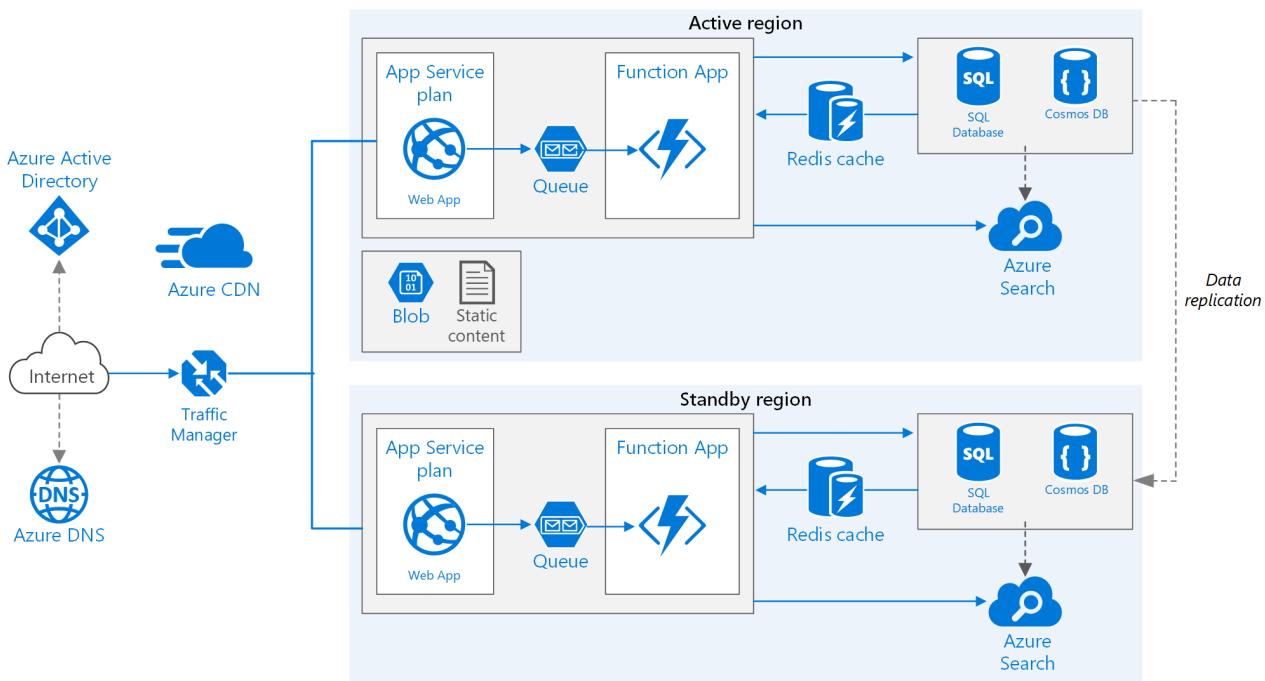
## **SQL Database encryption**

Use [Transparent Data Encryption](#) if you need to encrypt data at rest in the database. This feature performs real-time encryption and decryption of an entire database (including backups and transaction log files) and requires no changes to the application. Encryption does add some latency, so it's a good practice to separate the data that must be secure into its own database and enable encryption only for that database.

# Run a web application in multiple regions

10/26/2018 • 8 minutes to read • [Edit Online](#)

This reference architecture shows how to run an Azure App Service application in multiple regions to achieve high availability.



[Download a Visio file of this architecture.](#)

## Architecture

This architecture builds on the one shown in [Improve scalability in a web application](#). The main differences are:

- **Primary and secondary regions.** This architecture uses two regions to achieve higher availability. The application is deployed to each region. During normal operations, network traffic is routed to the primary region. If the primary region becomes unavailable, traffic is routed to the secondary region.
- **Azure DNS.** [Azure DNS](#) is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.
- **Azure Traffic Manager.** [Traffic Manager](#) routes incoming requests to the primary region. If the application running that region becomes unavailable, Traffic Manager fails over to the secondary region.
- **Geo-replication** of SQL Database and Cosmos DB.

A multi-region architecture can provide higher availability than deploying to a single region. If a regional outage affects the primary region, you can use [Traffic Manager](#) to fail over to the secondary region. This architecture can also help if an individual subsystem of the application fails.

There are several general approaches to achieving high availability across regions:

- Active/passive with hot standby. Traffic goes to one region, while the other waits on hot standby. Hot standby means the VMs in the secondary region are allocated and running at all times.
- Active/passive with cold standby. Traffic goes to one region, while the other waits on cold standby. Cold standby means the VMs in the secondary region are not allocated until needed for failover. This approach costs less to

run, but will generally take longer to come online during a failure.

- Active/active. Both regions are active, and requests are load balanced between them. If one region becomes unavailable, it is taken out of rotation.

This reference architecture focuses on active/passive with hot standby, using Traffic Manager for failover.

## Recommendations

Your requirements might differ from the architecture described here. Use the recommendations in this section as a starting point.

### Regional pairing

Each Azure region is paired with another region within the same geography. In general, choose regions from the same regional pair (for example, East US 2 and Central US). Benefits of doing so include:

- If there is a broad outage, recovery of at least one region out of every pair is prioritized.
- Planned Azure system updates are rolled out to paired regions sequentially to minimize possible downtime.
- In most cases, regional pairs reside within the same geography to meet data residency requirements.

However, make sure that both regions support all of the Azure services needed for your application. See [Services by region](#). For more information about regional pairs, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

### Resource groups

Consider placing the primary region, secondary region, and Traffic Manager into separate [resource groups](#). This lets you manage the resources deployed to each region as a single collection.

### Traffic Manager configuration

**Routing.** Traffic Manager supports several [routing algorithms](#). For the scenario described in this article, use *priority* routing (formerly called *failover* routing). With this setting, Traffic Manager sends all requests to the primary region unless the endpoint for that region becomes unreachable. At that point, it automatically fails over to the secondary region. See [Configure Failover routing method](#).

**Health probe.** Traffic Manager uses an HTTP (or HTTPS) probe to monitor the availability of each endpoint. The probe gives Traffic Manager a pass/fail test for failing over to the secondary region. It works by sending a request to a specified URL path. If it gets a non-200 response within a timeout period, the probe fails. After four failed requests, Traffic Manager marks the endpoint as degraded and fails over to the other endpoint. For details, see [Traffic Manager endpoint monitoring and failover](#).

As a best practice, create a health probe endpoint that reports the overall health of the application and use this endpoint for the health probe. The endpoint should check critical dependencies such as the App Service apps, storage queue, and SQL Database. Otherwise, the probe might report a healthy endpoint when critical parts of the application are actually failing.

On the other hand, don't use the health probe to check lower priority services. For example, if an email service goes down the application can switch to a second provider or just send emails later. This is not a high enough priority to cause the application to fail over. For more information, see [Health Endpoint Monitoring Pattern](#).

### SQL Database

Use [Active Geo-Replication](#) to create a readable secondary replica in a different region. You can have up to four readable secondary replicas. Fail over to a secondary database if your primary database fails or needs to be taken offline. Active Geo-Replication can be configured for any database in any elastic database pool.

### Cosmos DB

Cosmos DB supports geo-replication across regions with multi-master (multiple write regions). Alternatively, you can designate one region as the writable region and the others as read-only replicas. If there is a regional outage,

you can fail over by selecting another region to be the write region. The client SDK automatically sends write requests to the current write region, so you don't need to update the client configuration after a failover. For more information, see [Global data distribution with Azure Cosmos DB](#).

#### **NOTE**

All of the replicas belong to the same resource group.

### **Storage**

For Azure Storage, use [read-access geo-redundant storage](#) (RA-GRS). With RA-GRS storage, the data is replicated to a secondary region. You have read-only access to the data in the secondary region through a separate endpoint. If there is a regional outage or disaster, the Azure Storage team might decide to perform a geo-failover to the secondary region. There is no customer action required for this failover.

For Queue storage, create a backup queue in the secondary region. During failover, the app can use the backup queue until the primary region becomes available again. That way, the application can still process new requests.

## Availability considerations

### **Traffic Manager**

Traffic Manager automatically fails over if the primary region becomes unavailable. When Traffic Manager fails over, there is a period of time when clients cannot reach the application. The duration is affected by the following factors:

- The health probe must detect that the primary data center has become unreachable.
- Domain name service (DNS) servers must update the cached DNS records for the IP address, which depends on the DNS time-to-live (TTL). The default TTL is 300 seconds (5 minutes), but you can configure this value when you create the Traffic Manager profile.

For details, see [About Traffic Manager Monitoring](#).

Traffic Manager is a possible failure point in the system. If the service fails, clients cannot access your application during the downtime. Review the [Traffic Manager service level agreement \(SLA\)](#) and determine whether using Traffic Manager alone meets your business requirements for high availability. If not, consider adding another traffic management solution as a fallback. If the Azure Traffic Manager service fails, change your canonical name (CNAME) records in DNS to point to the other traffic management service. This step must be performed manually, and your application will be unavailable until the DNS changes are propagated.

### **SQL Database**

The recovery point objective (RPO) and estimated recovery time (ERT) for SQL Database are documented in [Overview of business continuity with Azure SQL Database](#).

### **Storage**

RA-GRS storage provides durable storage, but it's important to understand what can happen during an outage:

- If a storage outage occurs, there will be a period of time when you don't have write-access to the data. You can still read from the secondary endpoint during the outage.
- If a regional outage or disaster affects the primary location and the data there cannot be recovered, the Azure Storage team may decide to perform a geo-failover to the secondary region.
- Data replication to the secondary region is performed asynchronously. Therefore, if a geo-failover is performed, some data loss is possible if the data can't be recovered from the primary region.
- Transient failures, such as a network outage, will not trigger a storage failover. Design your application to be

resilient to transient failures. Possible mitigations:

- Read from the secondary region.
- Temporarily switch to another storage account for new write operations (for example, to queue messages).
- Copy data from the secondary region to another storage account.
- Provide reduced functionality until the system fails back.

For more information, see [What to do if an Azure Storage outage occurs](#).

## Manageability Considerations

### Traffic Manager

If Traffic Manager fails over, we recommend performing a manual failback rather than implementing an automatic failback. Otherwise, you can create a situation where the application flips back and forth between regions. Verify that all application subsystems are healthy before failing back.

Note that Traffic Manager automatically fails back by default. To prevent this, manually lower the priority of the primary region after a failover event. For example, suppose the primary region is priority 1 and the secondary is priority 2. After a failover, set the primary region to priority 3, to prevent automatic failback. When you are ready to switch back, update the priority to 1.

The following commands update the priority.

### PowerShell

```
$endpoint = Get-AzureRmTrafficManagerEndpoint -Name <endpoint> -ProfileName <profile> -ResourceGroupName <resource-group> -Type AzureEndpoints  
$endpoint.Priority = 3  
Set-AzureRmTrafficManagerEndpoint -TrafficManagerEndpoint $endpoint
```

For more information, see [Azure Traffic Manager Cmdlets](#).

### Azure CLI

```
az network traffic-manager endpoint update --resource-group <resource-group> --profile-name <profile> \  
--name <endpoint-name> --type azureEndpoints --priority 3
```

### SQL Database

If the primary database fails, perform a manual failover to the secondary database. See [Restore an Azure SQL Database or failover to a secondary](#). The secondary database remains read-only until you fail over.

# Designing, building, and operating microservices on Azure

10/24/2018 • 6 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 and run a microservices architecture on Azure. Topics include:

- Using Domain Driven Design (DDD) to design a microservices architecture.
- Choosing the right Azure technologies for compute, storage, messaging, and other elements of the design.
- Understanding microservices design patterns.
- Designing for resiliency, scalability, and performance.
- Building a CI/CD pipeline.

Throughout, we focus on an end-to-end scenario: A drone delivery service that lets customers schedule packages to be picked up and delivered via drone. You can find the code for our reference implementation on GitHub



[Reference implementation](#)

But first, let's start with fundamentals. What are microservices, and what are the advantages of adopting a microservices architecture?

## Why build microservices?

In a microservices architecture, the application is composed of small, independent services. Here are some of the defining characteristics of microservices:

- Each microservice implements a single business capability.
- A microservice is small enough that a single small team of developers can write and maintain it.
- Microservices run in separate processes, communicating through well-defined APIs or messaging patterns.
- Microservices do not share data stores or data schemas. Each microservice is responsible for managing its own data.
- Microservices have separate code bases, and do not share source code. They may use common utility libraries, however.
- Each microservice can be deployed and updated independently of other services.

Done correctly, microservices can provide a number of useful 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; as a result, new features may be held up waiting for a bug fix to be integrated, tested, and published.
- **Small code, small teams.** A microservice should be small enough that a single feature team can build, test, and deploy it. Small code bases are easier to understand. In a large monolithic application, there is a

tendency over time for code dependencies to become tangled, so that 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. Small team sizes also promote greater agility. The "two-pizza rule" says that a team should be small enough that two pizzas can feed the team. Obviously that's not an exact metric and depends on team appetites! But the point is that large groups tend to be less productive, because communication is slower, management overhead goes up, and agility diminishes.

- **Mix of technologies.** Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Resiliency.** 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.** A microservices architecture allows each microservice to be scaled independently of the others. That lets you scale out subsystems that require more resources, without scaling out the entire application. If you deploy services inside containers, you can also pack a higher density of microservices 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 impacted. 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.

## No free lunch

These benefits don't come for free. This series of articles is designed to address some of the challenges of building microservices that are resilient, scalable, and manageable.

- **Service boundaries.** When you build microservices, you need to think carefully about where to draw the boundaries between services. Once services are built and deployed in production, it can be hard to refactor across those boundaries. Choosing the right service boundaries is one of the biggest challenges when designing a microservices architecture. How big should each service be? When should functionality be factored across several services, and when should it be kept inside the same service? In this guide, we describe an approach that uses domain-driven design to find service boundaries. It starts with [Domain analysis](#) to find the bounded contexts, then applies a set of [Tactical DDD patterns](#) based on functional and non-functional requirements.
- **Data consistency and integrity.** A basic principle of microservices is that each service manages its own data. This keeps services decoupled, but can lead to challenges with data integrity or redundancy. We explore some of these issues in the [Data considerations](#).
- **Network congestion and latency.** The use of many small, granular services can result in more interservice communication and longer end-to-end latency. The chapter [Interservice communication](#) describes considerations for messaging between services. Both synchronous and asynchronous communication have a place in microservices architectures. Good [API design](#) is important so that services remain loosely coupled, and can be independently deployed and updated.
- **Complexity.** A microservices application has more moving parts. Each service may be simple, but the services have to work together as a whole. A single user operation may involve multiple services. In the chapter [Ingestion and workflow](#), we examine some of the issues around ingesting requests at high throughput, coordinating a workflow, and handling failures.
- **Communication between clients and the application.** When you decompose an application into many small services, how should clients communicate with those services? Should a client call each individual service directly, or route requests through an [API Gateway](#)?
- **Monitoring.** Monitoring a distributed application can be a lot harder than a monolithic application, because

you must correlate telemetry from multiple services. The chapter [Logging and monitoring](#) addresses these concerns.

- **Continuous integration and delivery (CI/CD).** One of the main goals of microservices is agility. To achieve this, you must have automated and robust [CI/CD](#), so that you can quickly and reliably deploy individual services into test and production environments.

## The Drone Delivery application

To explore these issues, and to illustrate some of the best practices for a microservices architecture, we created a reference implementation that we call the Drone Delivery application. You can find the reference implementation on [GitHub](#).

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](#).

### [Domain analysis](#)

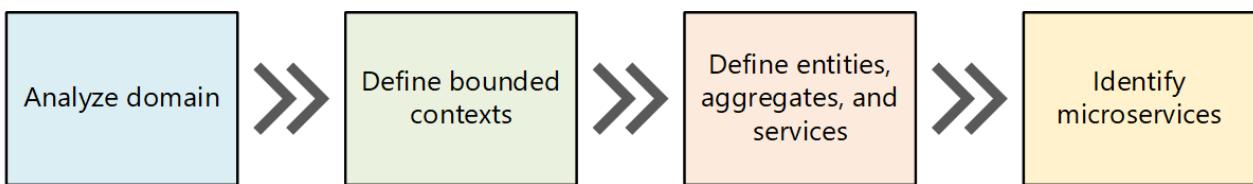
# Designing microservices: Domain analysis

10/24/2018 • 13 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. In this chapter, we take a domain-driven approach to designing microservices.

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 chapter 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 chapter, we cover the first three steps, which are primarily concerned with DDD. In the next chapter, we will 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 chapter is not meant to show a complete and comprehensive domain analysis. We deliberately kept the example brief, in order 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.

## 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.

## Drone Delivery: Analyzing the business domain.

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 Pattern](#) or the [Anti-Corruption Layer Pattern](#) for this purpose.

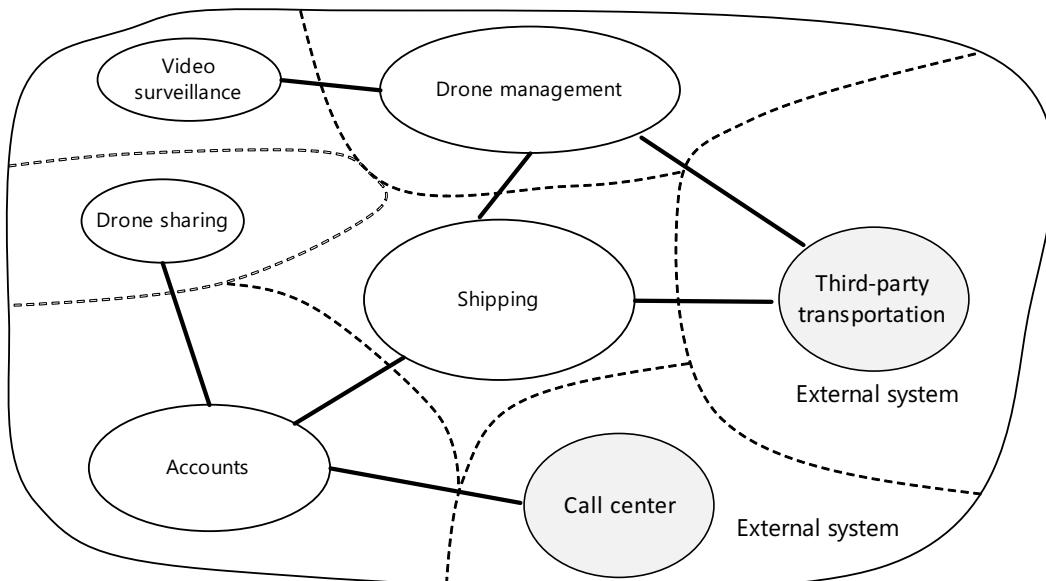
## Define bounded contexts

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 chapter on [API Design](#), we discuss using [OpenAPI](#)

[Specification](#) (formerly known as Swagger) to define language-agnostic interface descriptions for REST APIs, expressed in JSON or YAML format.

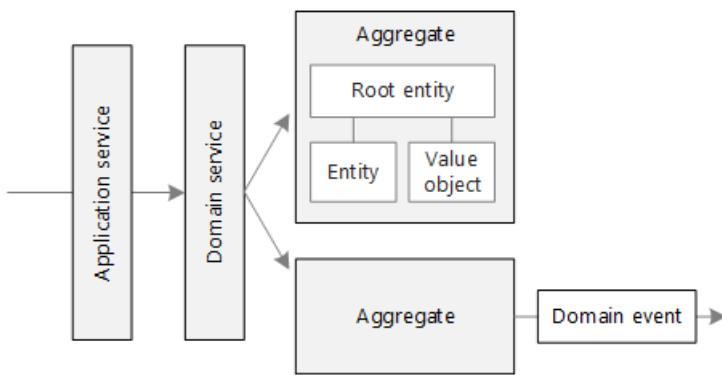
For the rest of this journey, we will focus on the Shipping bounded context.

## Tactical DDD

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 [next chapter](#)). 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 chapter [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.

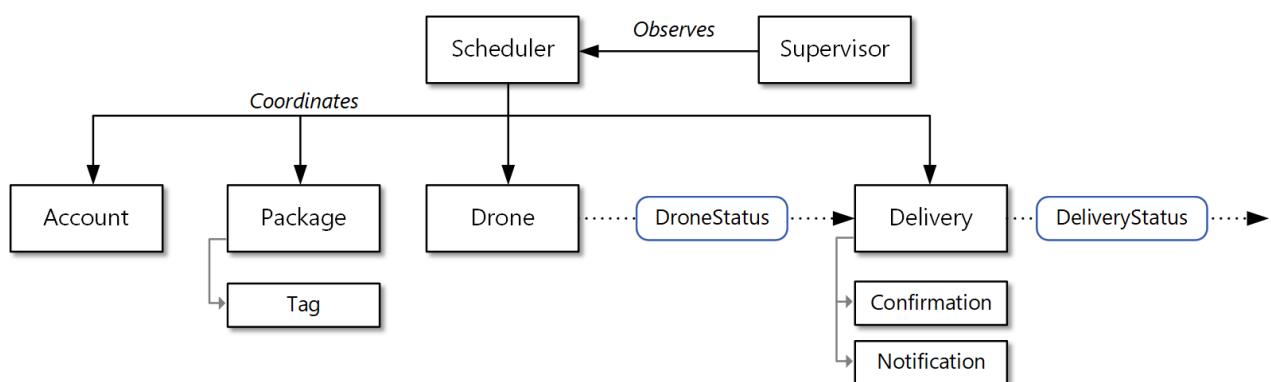


There are two domain events:

- 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](#).

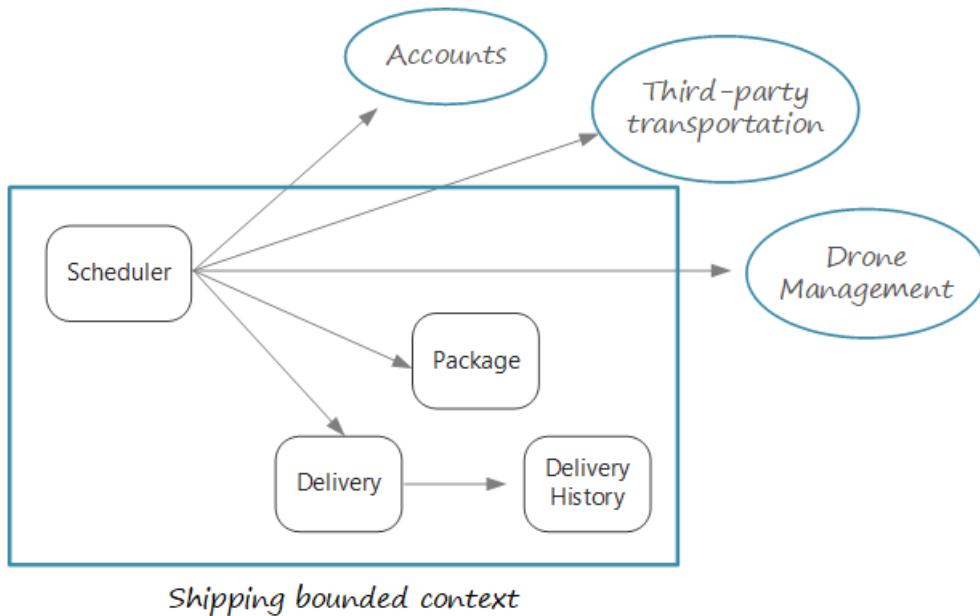


[Identifying microservice boundaries](#)

# Designing microservices: Identifying microservice boundaries

10/24/2018 • 9 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.



## From domain model to microservices

In the [previous chapter](#), we defined a set of bounded contexts for the 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.

## Drone Delivery: Defining the microservices

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:

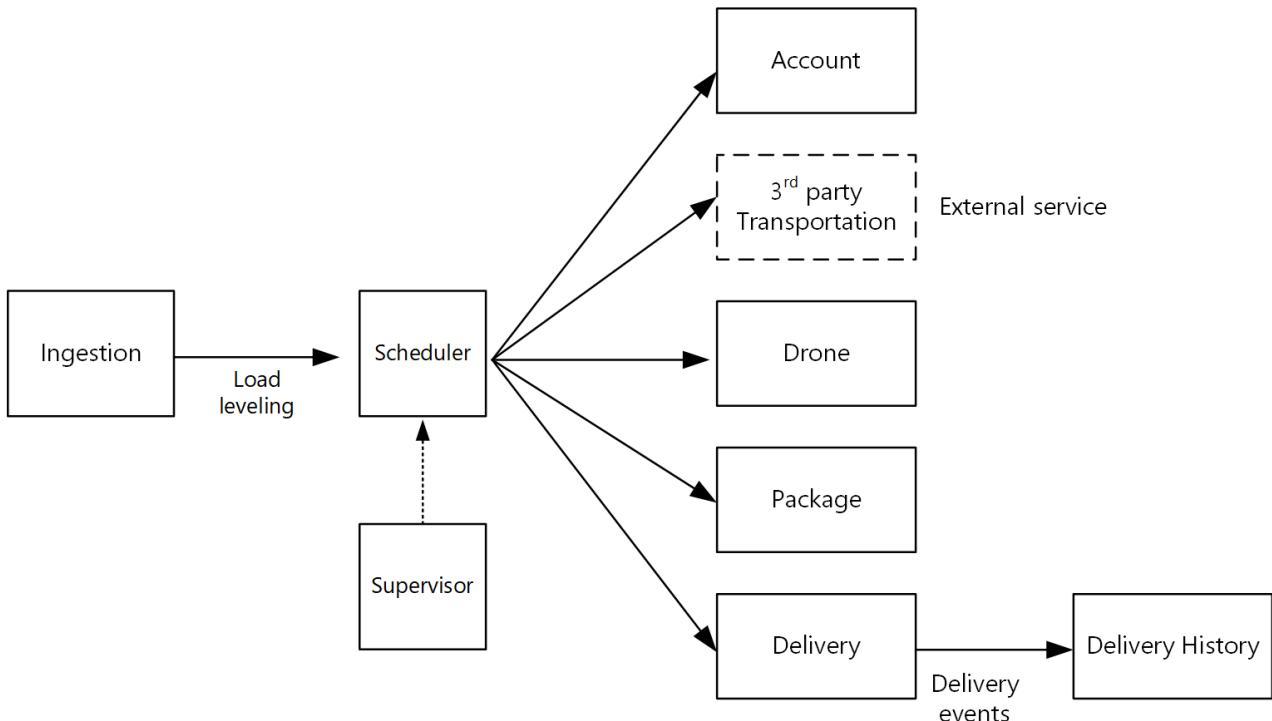
- What is the network overhead of calling directly into the other bounded context?
- Is the data schema for the other bounded context suitable for this context, or is it better to have a schema that's tailored to this bounded context?
- Is the other bounded context a legacy system? If so, you might create 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:



## Choosing a compute option

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](#).
- [Azure Container Service \(ACS\)](#) is an Azure service that lets you deploy a production-ready DC/OS, Docker Swarm, or Kubernetes cluster.

#### NOTE

Although Kubernetes is supported by ACS, we recommend AKS for running Kubernetes on Azure. AKS provides enhanced management capabilities and cost benefits.

## 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 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. 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.

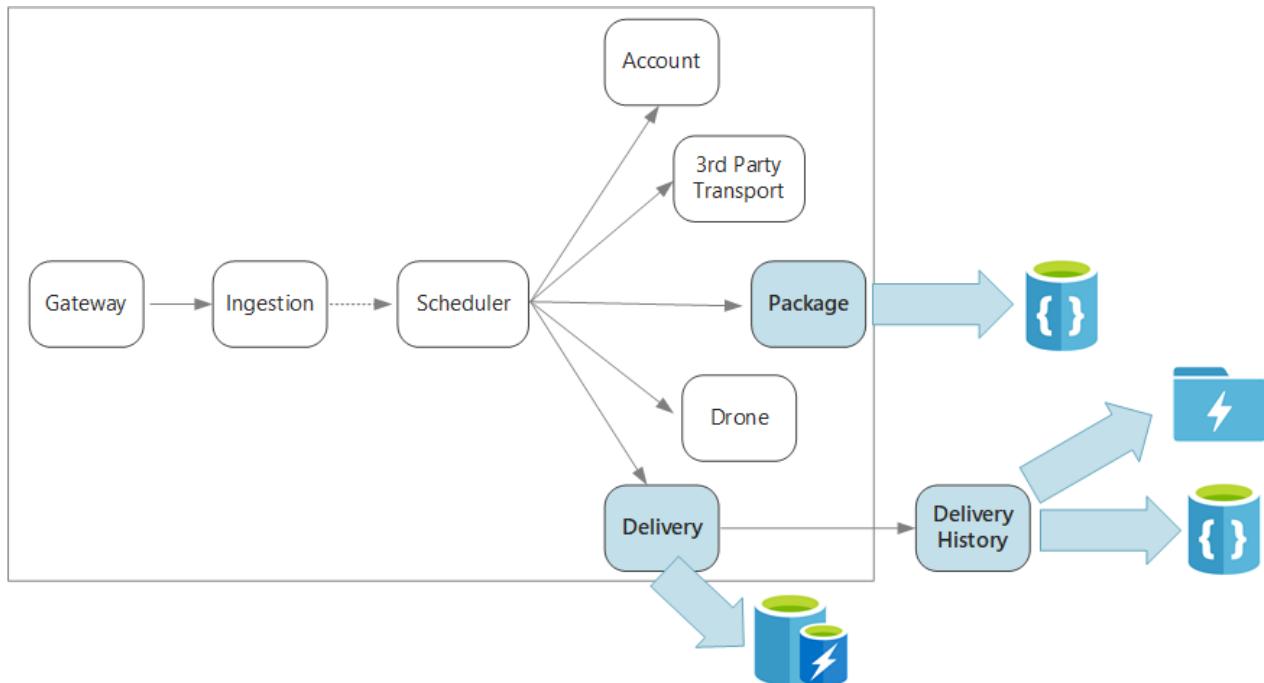
Our reference implementation primarily uses Kubernetes, but we did use Azure Functions for one service, namely the Delivery History service. Azure Functions was a good fit for this particular service, because it's an event-driven workload. By using an Event Hubs trigger to invoke the function, the service 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.

#### [Data considerations](#)

# Designing microservices: Data considerations

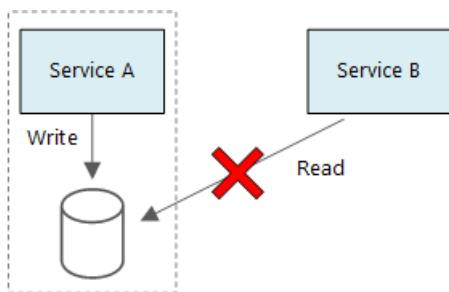
10/24/2018 • 7 minutes to read • [Edit Online](#)

This chapter 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.

## Drone Delivery: Choosing the data stores

Even with only a few services, the Shipping bounded context illustrates several of the points discussed in this section.

When a user schedules a new delivery, the client request includes information about both the delivery, such as the pickup and dropoff locations, and about the package, such as the size and weight. This information defines a unit of work, which the Ingestion service sends to Event Hubs. It's important that the unit of work stays in persistent storage while the Scheduler service is executing the workflow, so that no delivery requests are lost. For more discussion of the workflow, see [Ingestion and workflow](#).

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 Redis Cache 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 particularly 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.

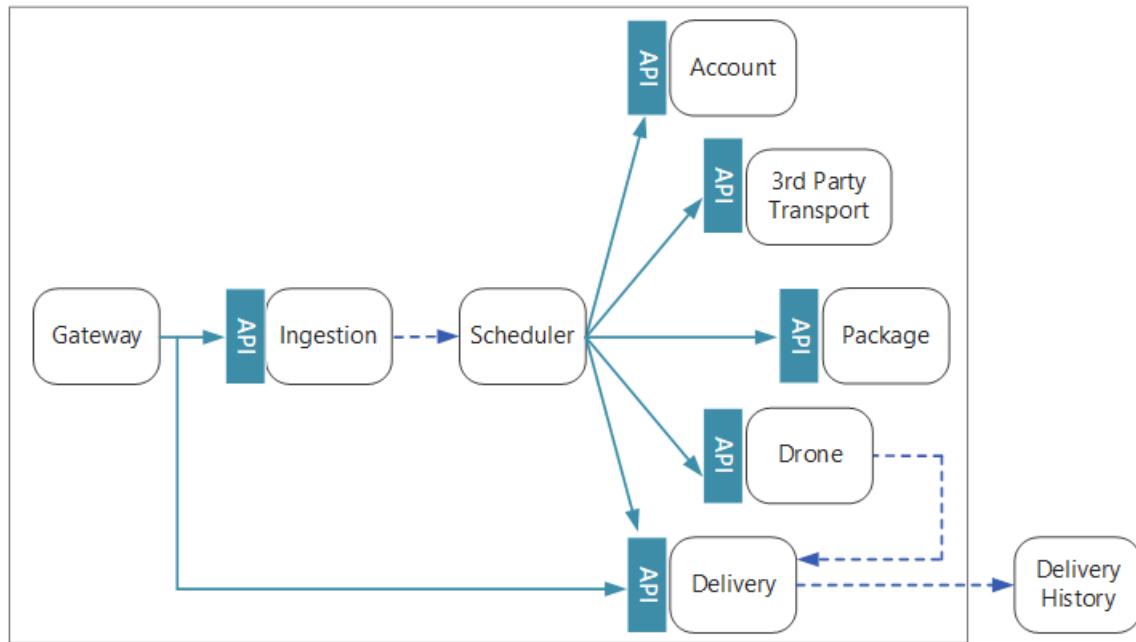
Because the package data is not relational, a document oriented database is appropriate, and Cosmos DB can achieve very high throughput by using sharded collections. The team that works on the Package service is familiar with the MEAN stack (MongoDB, Express.js, AngularJS, and Node.js), so they select 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.

#### [Interservice communication](#)

# Designing microservices: Interservice communication

10/24/2018 • 10 minutes to read • [Edit Online](#)

Communication between microservices must be efficient and robust. With lots of small services interacting to complete a single transaction, this can be a challenge. In this chapter, we look at the tradeoffs between asynchronous messaging versus synchronous APIs. Then we look at some of the challenges in designing resilient interservice communication, and the role that a service mesh can play.



## Challenges

Here are some of the main challenges arising from service-to-service communication. Service meshes, described later in this chapter, 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 chapter [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 very 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.

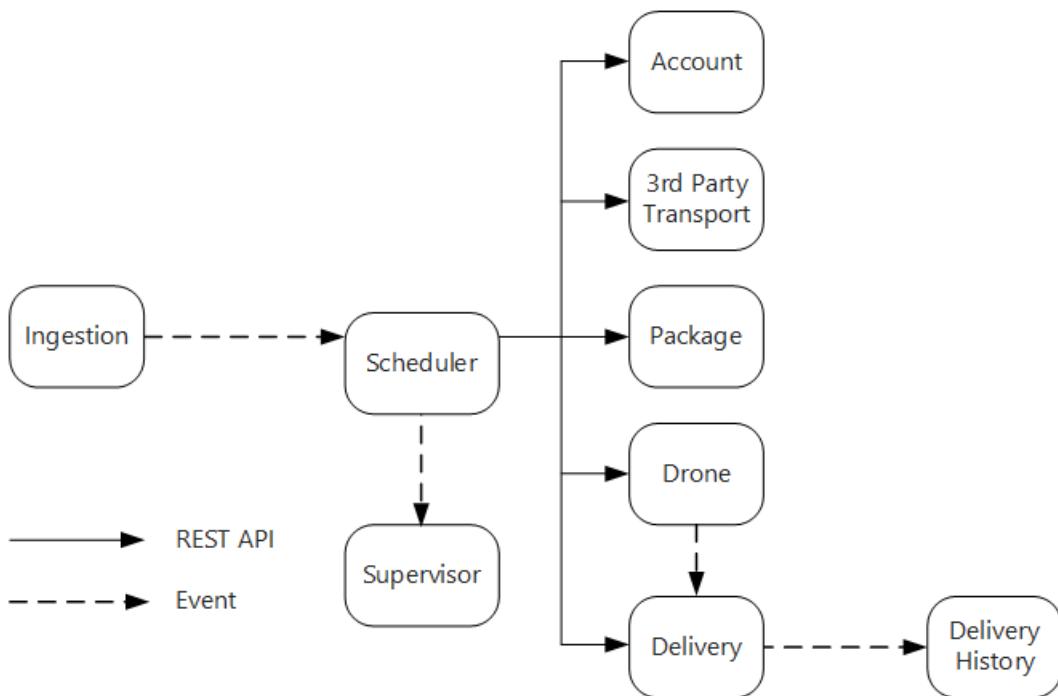
However, there are also some challenges to using asynchronous messaging effectively.

- **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. For details on how the Ingestion and Scheduler services interact, see [Ingestion and workflow](#).
- 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 non-transient 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, as described in the chapter [Ingestion and workflow](#).
- The Delivery service exposes a public API that clients can use to get the status of a delivery. In the chapter [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 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.

### 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.

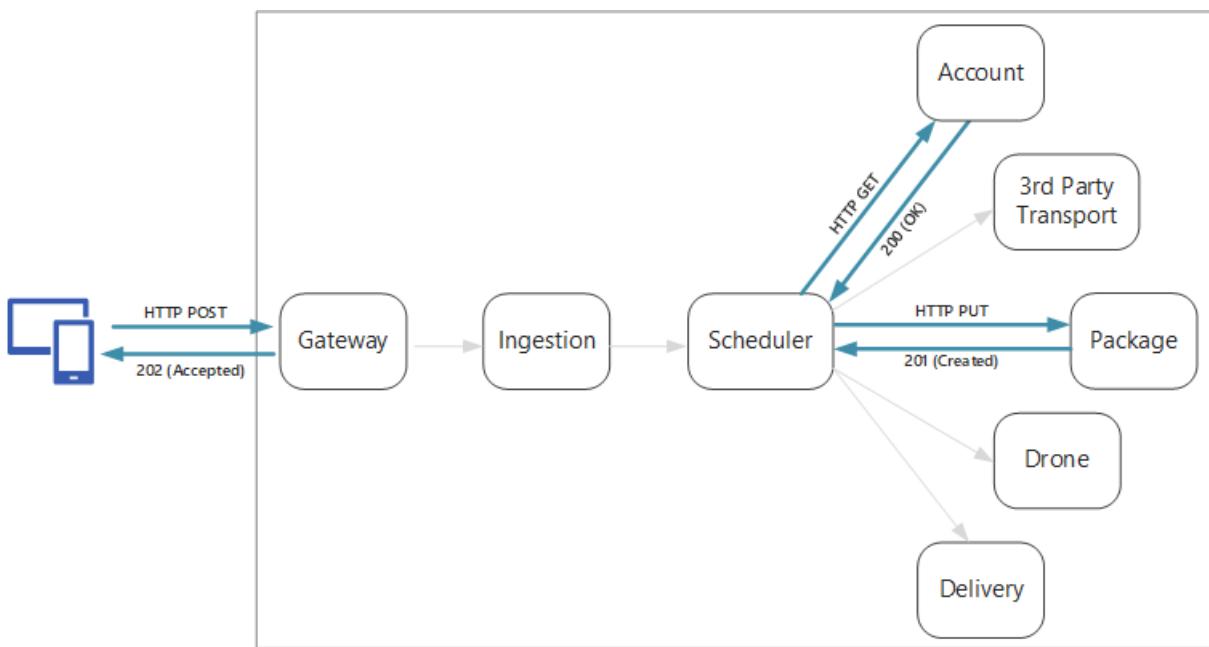
Do you need a service mesh? The value they add to a distributed system is certainly compelling. If you don't have a service mesh, you will need to consider each of the challenges mentioned at the beginning of the chapter. You can solve problems like retry, circuit breaker, and distributed tracing without a service mesh, but a service mesh moves these concerns out of the individual services and into a dedicated layer. On the other hand, service meshes are a relatively new technology that is still maturing. Deploying 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.

[API design](#)

# Designing microservices: API design

10/24/2018 • 10 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 vs 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 is 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 chapters [Ingestion and workflow](#) and [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.

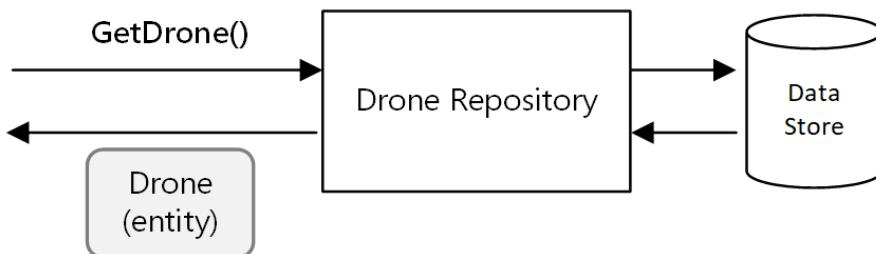
## Mapping REST to DDD patterns

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:

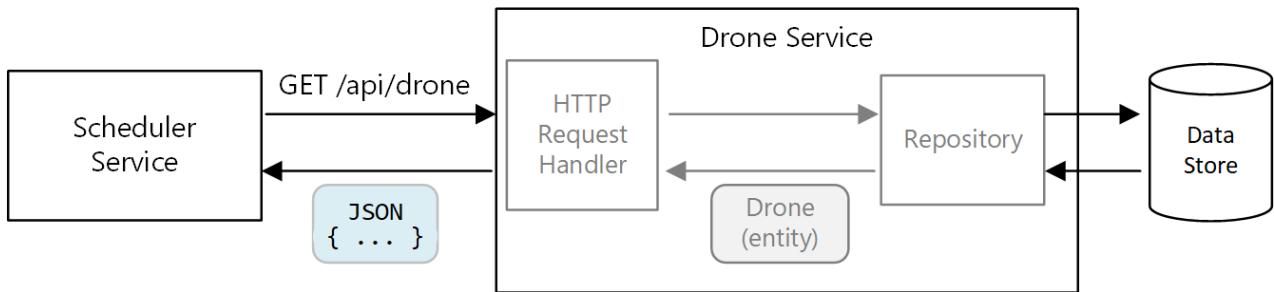
```
export class Location {  
    readonly latitude: number;  
    readonly longitude: number;  
  
    constructor(latitude: number, longitude: number) {  
        if (latitude < -90 || latitude > 90) {  
            throw new RangeError('latitude must be between -90 and 90');  
        }  
        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             | <a href="https://delivery-service/deliveries/1">https://delivery-service/deliveries/1</a>                     |
| Child entities       | Links           | { "href": "/deliveries/1/confirmation" }                                                                      |
| Update value objects | PUT or PATCH    | PUT <a href="https://delivery-service/deliveries/1/dropoff">https://delivery-service/deliveries/1/dropoff</a> |

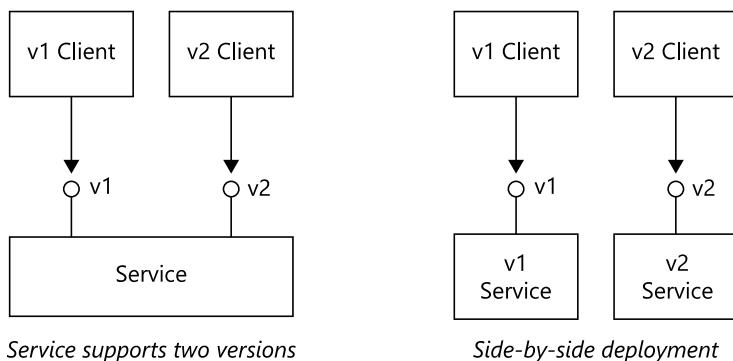
| DDD CONCEPT | REST EQUIVALENT | EXAMPLE                                                         |
|-------------|-----------------|-----------------------------------------------------------------|
| Repository  | Collection      | <code>https://delivery-service/deliveries?status=pending</code> |

## 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.



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.

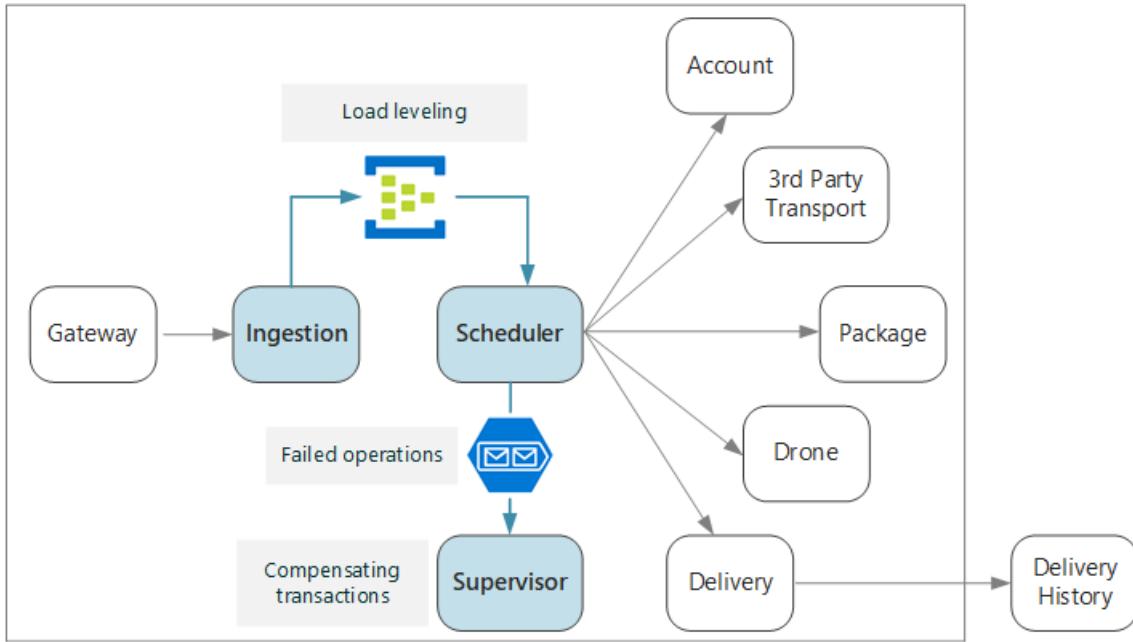
For further discussion of API versioning, see [Versioning a RESTful web API](#).

[Ingestion and workflow](#)

# Designing microservices: Ingestion and workflow

10/24/2018 • 17 minutes to read • [Edit Online](#)

Microservices often have a workflow that spans multiple services for a single transaction. The workflow must be reliable; it can't lose transactions or leave them in a partially completed state. It's also critical to control the ingestion rate of incoming requests. With many small services communicating with each other, a burst of incoming requests can overwhelm the interservice communication.



## The drone delivery workflow

In the Drone Delivery application, the following operations must be performed to schedule a delivery:

1. Check the status of the customer's account (Account service).
2. Create a new package entity (Package service).
3. Check whether any third-party transportation is required for this delivery, based on the pickup and delivery locations (Third-party Transportation service).
4. Schedule a drone for pickup (Drone service).
5. Create a new delivery entity (Delivery service).

This is the core of the entire application, so the end-to-end process must be performant as well as reliable. Some particular challenges must be addressed:

- **Load leveling.** Too many client requests can overwhelm the system with interservice network traffic. It can also overwhelm backend dependencies such as storage or remote services. These may react by throttling the services calling them, creating backpressure in the system. Therefore, it's important to load level the requests coming into the system, by putting them into a buffer or queue for processing.
- **Guaranteed delivery.** To avoid dropping any client requests, the ingestion component must guarantee at-least-once delivery of messages.
- **Error handling.** If any of the services returns an error code or experiences a non-transient failure, the delivery cannot be scheduled. An error code might indicate an expected error condition (for example, the customer's account is suspended) or an unexpected server error (HTTP 5xx). A service might also be

unavailable, causing the network call to time out.

First we'll look at the ingestion side of the equation — how the system can ingest incoming user requests at high throughput. Then we'll consider how the drone delivery application can implement a reliable workflow. It turns out that the design of the ingestion subsystem affects the workflow backend.

## Ingestion

Based on business requirements, the development team identified the following non-functional requirements for ingestion:

- Sustained throughput of 10K requests/sec.
- Able to handle spikes of up to 50K/sec without dropping client requests or timing out.
- Less than 500ms latency in the 99th percentile.

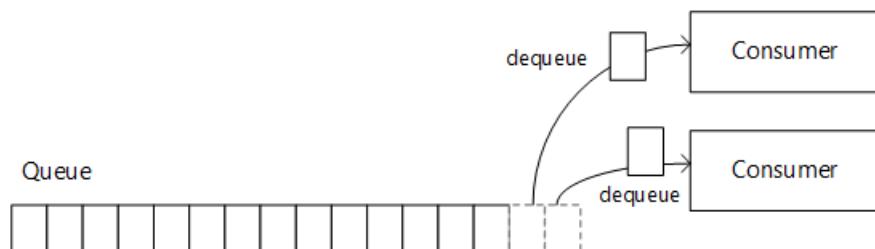
The requirement to handle occasional spikes in traffic presents a design challenge. In theory, the system could be scaled out to handle the maximum expected traffic. However, provisioning that many resources would be very inefficient. Most of the time, the application will not need that much capacity, so there would be idle cores, costing money without adding value.

A better approach is to put the incoming requests into a buffer, and let the buffer act as a load leveler. With this design, the Ingestion service must be able to handle the maximum ingestion rate over short periods, but the backend services only need to handle the maximum sustained load. By buffering at the front end, the backend services shouldn't need to handle large spikes in traffic. At the scale required for the Drone Delivery application, [Azure Event Hubs](#) is a good choice for load leveling. Event Hubs offers low latency and high throughput, and is a cost effective solution at high ingestion volumes.

For our testing, we used a Standard tier event hub with 32 partitions and 100 throughput units. We observed about 32K events / second ingestion, with latency around 90ms. Currently the default limit is 20 throughput units, but Azure customers can request additional throughput units by filing a support request. See [Event Hubs quotas](#) for more information. As with all performance metrics, many factors can affect performance, such as message payload size, so don't interpret these numbers as a benchmark. If more throughput is needed, the Ingestion service can shard across more than one event hub. For even higher throughput rates, [Event Hubs Dedicated](#) offers single-tenant deployments that can ingress over 2 million events per second.

It's important to understand how Event Hubs can achieve such high throughput, because that affects how a client should consume messages from Event Hubs. Event Hubs does not implement a *queue*. Rather, it implements an *event stream*.

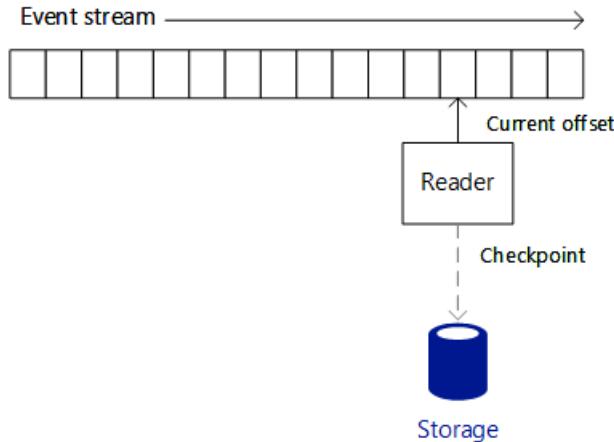
With a queue, an individual consumer can remove a message from the queue, and the next consumer won't see that message. Queues therefore allow you to use a [Competing Consumers pattern](#) to process messages in parallel and improve scalability. For greater resiliency, the consumer holds a lock on the message and releases the lock when it's done processing the message. If the consumer fails — for example, the node it's running on crashes — the lock times out and the message goes back onto the queue.



Event Hubs, on the other hand, uses streaming semantics. Consumers read the stream independently at their own pace. Each consumer is responsible for keeping track of its current position in the stream. A consumer should write its current position to persistent storage at some predefined interval. That way, if the consumer experiences

a fault (for example, the consumer crashes, or the host fails), then a new instance can resume reading the stream from the last recorded position. This process is called *checkpointing*.

For performance reasons, a consumer generally doesn't checkpoint after each message. Instead, it checkpoints at some fixed interval, for example after processing  $n$  messages, or every  $n$  seconds. As a consequence, if a consumer fails, some events may get processed twice, because a new instance always picks up from the last checkpoint. There is a tradeoff: Frequent checkpoints can hurt performance, but sparse checkpoints mean you will replay more events after a failure.



Event Hubs is not designed for competing consumers. Although multiple consumers can read a stream, each traverses the stream independently. Instead, Event Hubs uses a partitioned consumer pattern. An event hub has up to 32 partitions. Horizontal scale is achieved by assigning a separate consumer to each partition.

What does this mean for the drone delivery workflow? To get the full benefit of Event Hubs, the Delivery Scheduler cannot wait for each message to be processed before moving onto the next. If it does that, it will spend most of its time waiting for network calls to complete. Instead, it needs to process batches of messages in parallel, using asynchronous calls to the backend services. As we'll see, choosing the right checkpointing strategy is also important.

## Workflow

We looked at three options for reading and processing the messages: Event Processor Host, Service Bus queues, and the IoT Hub React library. We chose IoT Hub React, but to understand why, it helps to start with Event Processor Host.

### Event Processor Host

Event Processor Host is designed for message batching. The application implements the `IEventProcessor` interface, and the Processor Host creates one event processor instance for each partition in the event hub. The Event Processor Host then calls each event processor's `ProcessEventsAsync` method with batches of event messages. The application controls when to checkpoint inside the `ProcessEventsAsync` method, and the Event Processor Host writes the checkpoints to Azure storage.

Within a partition, Event Processor Host waits for `ProcessEventsAsync` to return before calling again with the next batch. This approach simplifies the programming model, because your event processing code doesn't need to be reentrant. However, it also means that the event processor handles one batch at a time, and this gates the speed at which the Processor Host can pump messages.

#### NOTE

The Processor Host doesn't actually *wait* in the sense of blocking a thread. The `ProcessEventsAsync` method is asynchronous, so the Processor Host can do other work while the method is completing. But it won't deliver another batch of messages for that partition until the method returns.

In the drone application, a batch of messages can be processed in parallel. But waiting for the whole batch to complete can still cause a bottleneck. Processing can only be as fast as the slowest message within a batch. Any variation in response times can create a "long tail," where a few slow responses drag down the entire system. Our performance tests showed that we did not achieve our target throughput using this approach. This does *not* mean that you should avoid using Event Processor Host. But for high throughput, avoid doing any long-running tasks inside the `ProcessEventsAsync` method. Process each batch quickly.

### IoTHub React

[IoTHub React](#) is an Akka Streams library for reading events from Event Hub. Akka Streams is a stream-based programming framework that implements the [Reactive Streams](#) specification. It provides a way to build efficient streaming pipelines, where all streaming operations are performed asynchronously, and the pipeline gracefully handles backpressure. Backpressure occurs when an event source produces events at a faster rate than the downstream consumers can receive them — which is exactly the situation when the drone delivery system has a spike in traffic. If backend services go slower, IoTHub React will slow down. If capacity is increased, IoTHub React will push more messages through the pipeline.

Akka Streams is also a very natural programming model for streaming events from Event Hubs. Instead of looping through a batch of events, you define a set of operations that will be applied to each event, and let Akka Streams handle the streaming. Akka Streams defines a streaming pipeline in terms of *Sources*, *Flows*, and *Sinks*. A source generates an output stream, a flow processes an input stream and produces an output stream, and a sink consumes a stream without producing any output.

Here is the code in the Scheduler service that sets up the Akka Streams pipeline:

```
IoHub iotHub = new IoHub();
Source<MessageFromDevice, NotUsed> messages = iotHub.source(options);

messages.map(msg -> DeliveryRequestEventProcessor.parseDeliveryRequest(msg))
    .filter(ad -> ad.getDelivery() != null).via(deliveryProcessor()).to(iotHub.checkpointSink())
    .run(streamMaterializer);
```

This code configures Event Hubs as a source. The `map` statement deserializes each event message into a Java class that represents a delivery request. The `filter` statement removes any `null` objects from the stream; this guards against the case where a message can't be serialized. The `via` statement joins the source to a flow that processes each delivery request. The `to` method joins the flow to the checkpoint sink, which is built into IoTHub React.

IoTHub React uses a different checkpointing strategy than Event Host Processor. Checkpoints are written by the checkpoint sink, which is the terminating stage in the pipeline. The design of Akka Streams allows the pipeline to continue streaming data while the sink is writing the checkpoint. That means the upstream processing stages don't need to wait for checkpointing to happen. You can configure checkpointing to occur after a timeout or after a certain number of messages have been processed.

The `deliveryProcessor` method creates the Akka Streams flow:

```

private static Flow<AkkaDelivery, MessageFromDevice, NotUsed> deliveryProcessor() {
    return Flow.of(AkkaDelivery.class).map(delivery -> {
        CompletableFuture<DeliverySchedule> completableSchedule = DeliveryRequestEventProcessor
            .processDeliveryRequestAsync(delivery.getDelivery(),
                delivery.getMessageFromDevice().properties());

        completableSchedule.whenComplete((deliverySchedule,error) -> {
            if (error!=null){
                Log.info("failed delivery" + error.getStackTrace());
            }
            else{
                Log.info("Completed Delivery",deliverySchedule.toString());
            }
        });
        completableSchedule = null;
        return delivery.getMessageFromDevice();
    });
}

```

The flow calls a static `processDeliveryRequestAsync` method that does the actual work of processing each message.

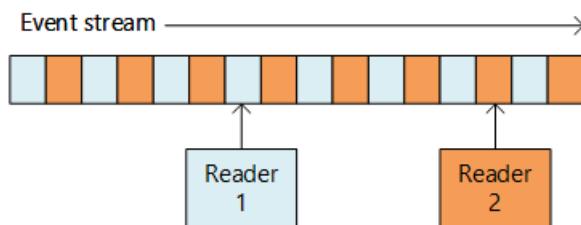
## Scaling with IoT Hub React

The Scheduler service is designed so that each container instance reads from a single partition. For example, if the Event Hub has 32 partitions, the Scheduler service is deployed with 32 replicas. This allows for a lot of flexibility in terms of horizontal scaling.

Depending on the size of the cluster, a node in the cluster might have more than one Scheduler service pod running on it. But if the Scheduler service needs more resources, the cluster can be scaled out, in order to distribute the pods across more nodes. Our performance tests showed that the Scheduler service is memory- and thread-bound, so performance depended greatly on the VM size and the number of pods per node.

Each instance needs to know which Event Hubs partition to read from. To configure the partition number, we took advantage of the [StatefulSet](#) resource type in Kubernetes. Pods in a StatefulSet have a persistent identifier that includes a numeric index. Specifically, the pod name is `<statefulset name>-<index>`, and this value is available to the container through the Kubernetes [Downward API](#). At run time, the Scheduler services reads the pod name and uses the pod index as the partition ID.

If you needed to scale out the Scheduler service even further, you could assign more than one pod per event hub partition, so that multiple pods are reading each partition. However, in that case, each instance would read all of the events in the assigned partition. To avoid duplicate processing, you would need to use a hashing algorithm, so that each instance skips over a portion of the messages. That way, multiple readers can consume the stream, but every message is processed by only one instance.



## Service Bus queues

A third option that we considered was to copy messages from Event Hubs into a Service Bus queue, and then have the Scheduler service read the messages from Service Bus. It might seem strange to writing the incoming requests into Event Hubs only to copy them in Service Bus. However, the idea was to leverage the different strengths of each service: Use Event Hubs to absorb spikes of heavy traffic, while taking advantage of the queue semantics in Service Bus to process the workload with a competing consumers pattern. Remember that our target

for sustained throughput is less than our expected peak load, so processing the Service Bus queue would not need to be as fast the message ingestion.

With this approach, our proof-of-concept implementation achieved about 4K operations per second. These tests used mock backend services that did not do any real work, but simply added a fixed amount of latency per service. Note that our performance numbers were much less than the theoretical maximum for Service Bus. Possible reasons for the discrepancy include:

- Not having optimal values for various client parameters, such as the connection pool limit, the degree of parallelization, the prefetch count, and the batch size.
- Network I/O bottlenecks.
- Use of [PeekLock](#) mode rather than [ReceiveAndDelete](#), which was needed to ensure at-least-once delivery of messages.

Further performance tests might have discovered the root cause and allowed us to resolve these issues. However, IoTHub React met our performance target, so we chose that option. That said, Service Bus is a viable option for this scenario.

## Handling failures

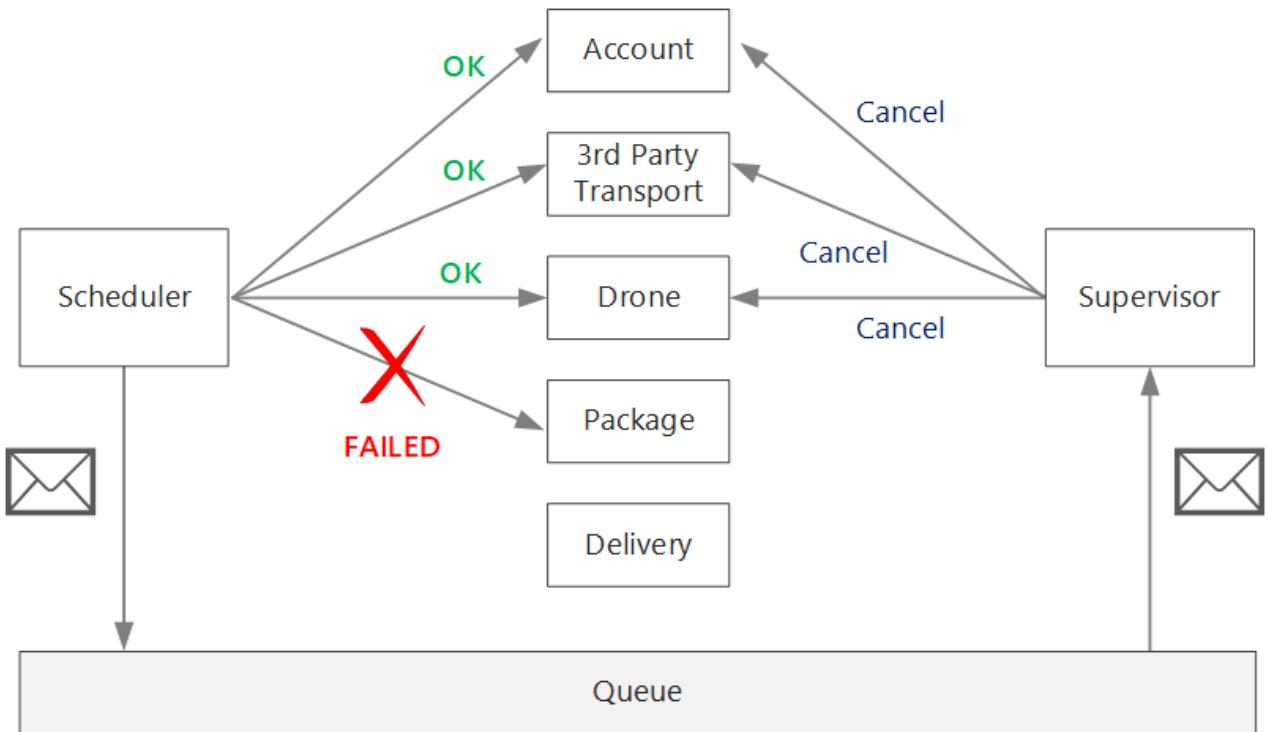
There are three general classes of failure to consider.

1. A downstream service may have a non-transient failure, which is any failure that's unlikely to go away by itself. Non-transient failures include normal error conditions, such as invalid input to a method. 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. (See [Compensating Transactions](#), below.)
2. A downstream 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 non-transient failure.
3. The Scheduler service itself might fault (for example, because a node crashes). In that case, Kubernetes will bring up a new instance of the service. However, any transactions that were already in progress must be resumed.

## Compensating transactions

If a non-transient failure happens, 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.



## Idempotent vs non-idempotent operations

To avoid losing any requests, the Scheduler service must guarantee that all messages are processed at least once. Event Hubs can guarantee at-least-once delivery if the client checkpoints correctly.

If the Scheduler service crashes, it may be in the middle of processing one or more client requests. Those messages will be picked up by another instance of the Scheduler and reprocessed. What happens if a request is processed twice? It's important to avoid duplicating any work. After all, we don't want the system to send two drones for the same package.

One approach 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. In other words, a client can invoke the operation once, twice, or many times, and the result will be the same. Essentially, the service should ignore duplicate calls. For a method with side effects to be idempotent, the service must be able to detect duplicate calls. For example, you can have the caller assign the ID, rather than having the service generate a new ID. The service can then check for duplicate IDs.

### NOTE

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.

It's not always straightforward to write idempotent methods. Another option is for the Scheduler to track the progress of every transaction in a durable store. Whenever it processes a message, it would look up the state in the durable store. After each step, it would write the result to the store. There may be performance implications to this approach.

## Example: Idempotent operations

The HTTP specification states that PUT methods must be 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();
    }
}
```

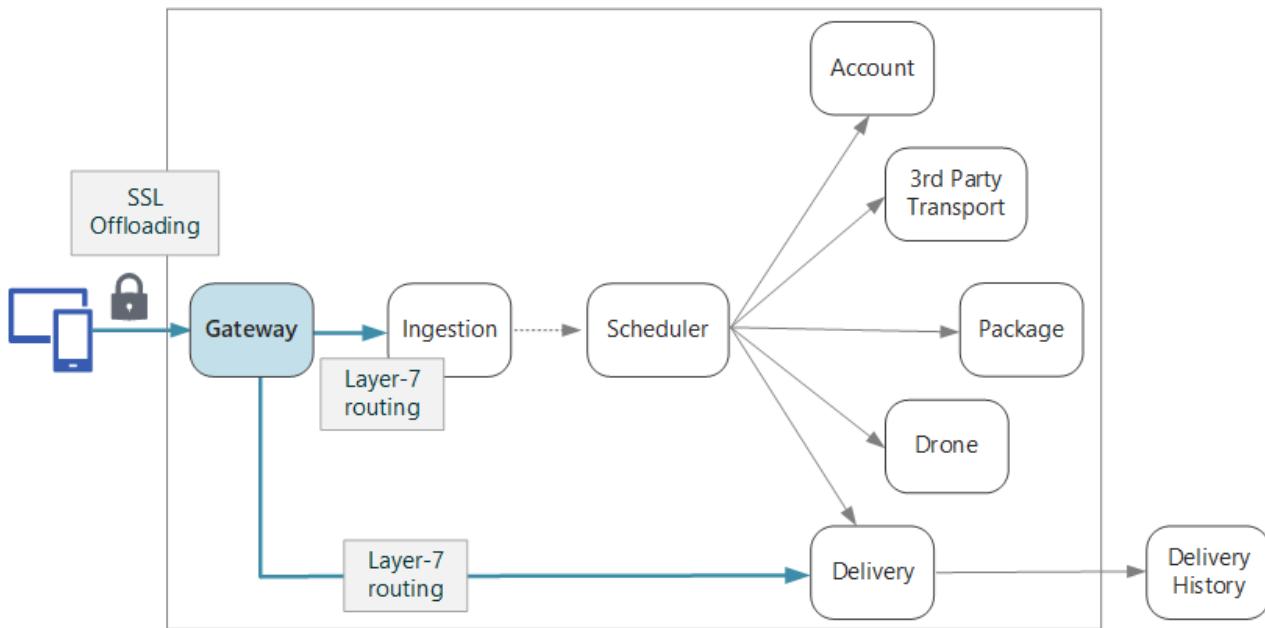
It's expected that most requests will create a new entity, so the method optimistically calls `CreateAsync` on the repository object, and then handles any duplicate-resource exceptions by updating the resource instead.

[API gateways](#)

# Designing microservices: API gateways

10/24/2018 • 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 require 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 whitelisting
- 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 called 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 white listing, 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 whitelists, 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:

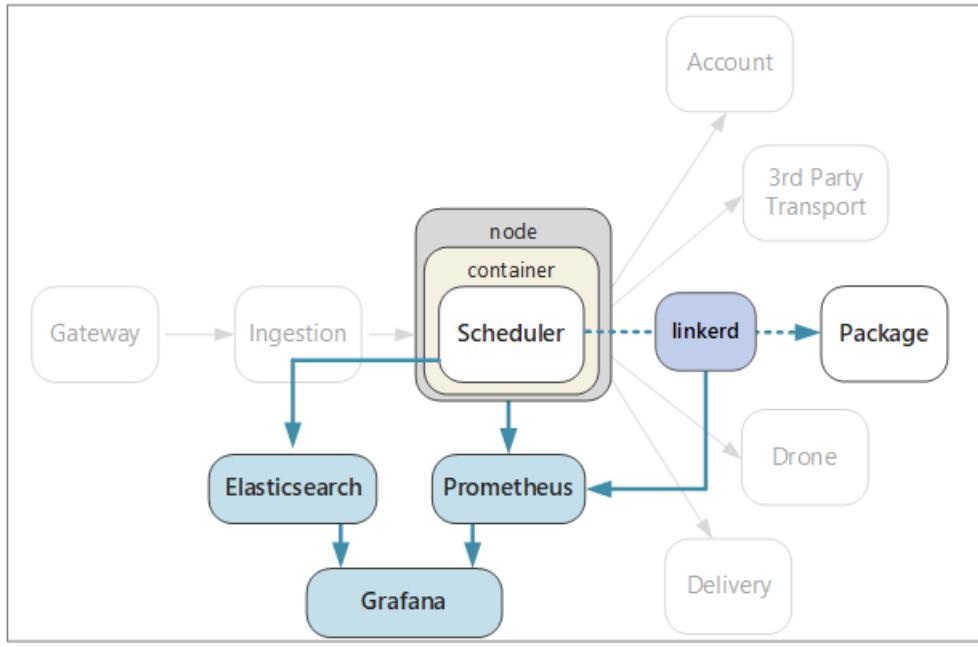
- Isolation. All inbound traffic goes to a fixed set of nodes, which can be isolated from backend services.
- Stable configuration. If the gateway is misconfigured, the entire application may become unavailable.
- Performance. You may want to use a specific VM configuration for the gateway for performance reasons.

[Logging and monitoring](#)

# Designing microservices: Logging and monitoring

10/24/2018 • 13 minutes to read • [Edit Online](#)

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. Logging and monitoring are critically important to give you a holistic view of the system.



In a microservices architecture, it can be especially challenging to pinpoint the exact cause of errors or performance bottlenecks. A single user operation might span multiple services. Services may hit network I/O limits inside the cluster. A chain of calls across services may cause backpressure in the system, resulting in high latency or cascading failures. Moreover, you generally don't know which node a particular container will run in. Containers placed on the same node may be competing for limited CPU or memory.

To make sense of what's happening, you must collect telemetry from the application. Telemetry can be divided into *logs* and *metrics*. [Azure Monitor](#) collects both logs and metrics across the Azure platform.

**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. Metrics can be further subcategorized as follows:

- **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. If services are run inside containers, you need to collect metrics at the container level, not just at the VM level. You can set up Azure Monitor to monitor container workloads in Azure Kubernetes Service (AKS). For more information, see [Azure Monitor for containers overview](#). For other container orchestrators, use the [Container Monitoring solution in Log Analytics](#).
- **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. Use [Application Insights](#) to enable application metrics.

- **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.

## Considerations

The article [Monitoring and diagnostics](#) describes general best practices for monitoring an application. Here are some particular things to think about in the context of a microservices architecture.

**Configuration and management.** Will you use a managed service for logging and monitoring, or deploy logging and monitoring components as containers inside the cluster? For more discussion of these options, see the section [Technology Options](#) below.

**Ingestion rate.** What is the throughput at which the system can ingest telemetry events? What happens if that rate is exceeded? For example, the system may throttle clients, in which case telemetry data is lost, or it may downsample the data. Sometimes you can mitigate this problem by reducing the amount of data that you collect:

- Aggregate metrics by calculating statistics, such as average and standard deviation, and send that statistical data to the monitoring system.
- Downsample the data — that is, process only a percentage of the events.
- Batch the data to reduce the number of network calls to the monitoring service.

**Cost.** The cost of ingesting and storing telemetry data may be high, especially at high volumes. In some cases it could even exceed the cost of running the application. In that case, you may need to reduce the volume of telemetry by aggregating, downsampling, or batching the data, as described above.

**Data fidelity.** How accurate are the metrics? Averages can hide outliers, especially at scale. Also, if the sampling rate is too low, it can smooth out fluctuations in the data. It may appear that all requests have about the same end-to-end latency, when in fact a significant fraction of requests are taking much longer.

**Latency.** To enable real-time monitoring and alerts, telemetry data should be available quickly. How "real-time" is the data that appears on the monitoring dashboard? A few seconds old? More than a minute?

**Storage.** For logs, it may be most efficient to write the log events to ephemeral storage in the cluster, and configure an agent to ship the log files to more persistent storage. Data should eventually be moved to long-term storage so that it's available for retrospective analysis. A microservices architecture can generate a large volume of telemetry data, so the cost of storing that data is an important consideration. Also consider how you will query the data.

**Dashboard and visualization.** Do you get a holistic view of the system, across all of the services, both within the cluster and external services? If you are writing telemetry data and logs to more than one location, can the dashboard show all of them and correlate? The monitoring dashboard should show at least the following information:

- Overall resource allocation for capacity and growth. This includes the number of containers, file system metrics, network, and core allocation.
- Container metrics correlated at the service level.
- System metrics correlated with containers.
- Service errors and outliers.

## Distributed tracing

As mentioned, one challenge in microservices is understanding the flow of events across services. A single operation or 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. Similarly, 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.

Here are some considerations when implementing distributed tracing:

- There is currently no standard HTTP header for correlation IDs. Your team should standardize on a custom header value. The choice may be decided by your logging/monitoring framework 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.
- Rather than a single opaque identifier, you might send a *correlation context* that includes richer information, such as caller-callee relationships.
- The Azure Application Insights SDK automatically injects correlation context into HTTP headers, and includes the correlation ID in Application Insights logs. If you decide to use the correlation features built into Application Insights, some services may still need to explicitly propagate the correlation headers, depending on the libraries being used. For more information, see [Telemetry correlation in Application Insights](#).
- 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.
  - Istio: [Distributed Request Tracing](#)
  - linkerd: [Context Headers](#)
- Consider how you will aggregate logs. You may want to standardize across teams on how to include correlation IDs in logs. Use a structured or semi-structured format, such as JSON, and define a common field to hold the correlation ID.

## Technology options

**Application Insights** is a managed service in Azure that ingests and stores telemetry data, and provides tools for analyzing and searching the data. To use Application Insights, 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. Application Insights provides built-in correlation and dependency tracking. It lets you track system metrics, application metrics, and Azure service metrics, all in one place.

Be aware that Application Insights throttles 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. 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.

In addition, make sure that you understand the pricing model for Application Insights, because you are charged based on data volume. For more information, see [Manage pricing and data volume in Application Insights](#). If your application generates a large volume of telemetry, and you don't wish to perform sampling or aggregation of the data, then Application Insights may not be the appropriate choice.

If Application Insights doesn't meet your requirements, here are some suggested approaches that use popular open-source technologies.

For system and container metrics, 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 [Heapster](#), which is a service that collects cluster-wide metrics from kubelet, aggregates the data, and pushes it to InfluxDB or other time-series storage solution. Azure Container Service deploys Heapster as part of the cluster setup. Another option is [Telegraf](#), which is an agent for collecting and reporting metrics.
- 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). Whereas Heapster aggregates metrics that Kubernetes generates and forwards them to a sink, kube-state-metrics generates its own metrics and makes them available through an endpoint for scraping. 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.
- Use a dashboard tool such as **Kibana** or **Grafana** to visualize and monitor the data. The dashboard service can also run inside a container in the cluster.

For application logs, consider using **Fluentd** and **Elasticsearch**. Fluentd is an open source data collector, and Elasticsearch is a document database that is optimized to act as a search engine. Using this approach, each service sends logs to `stdout` and `stderr`, and Kubernetes writes these streams to the local file system. Fluentd collects the logs, optionally enriches them with additional metadata from Kubernetes, and sends the logs to Elasticsearch. Use Kibana, Grafana, or a similar tool to create a dashboard for Elasticsearch. Fluentd runs as a daemonset in the cluster, which ensures that one Fluentd pod is assigned to each node. You can configure Fluentd to collect kubelet logs as well as container logs. At high volumes, writing logs to the local file system could become a performance bottleneck, especially when multiple services are running on the same node. Monitor disk latency and file system utilization in production.

One advantage of using Fluentd with Elasticsearch for logs is that services do not require any additional library dependencies. Each service just writes to `stdout` and `stderr`, and Fluentd handles exporting the logs into Elasticsearch. Also, the teams writing services don't need to understand how to configure the logging infrastructure. One challenge is to configure the Elasticsearch cluster for a production deployment, so that it scales to handle your traffic.

Another option is to send logs to Operations Management Suite (OMS) Log Analytics. The [Log Analytics](#) service collects log data into a central repository, and can also consolidate data from other Azure services that your application uses. For more information, see [Monitor an Azure Container Service cluster with Microsoft Operations Management Suite \(OMS\)](#).

## Example: Logging with correlation IDs

To illustrate some of the points discussed in this chapter, here is an extended example of how the Package service implements logging. The Package service was written in TypeScript and uses the [Koa](#) web framework for Node.js. There are several Node.js logging libraries to choose from. We picked [Winston](#), a popular logging library that met

our performance requirements when we tested it.

To encapsulate the implementation details, we defined an abstract `ILogger` interface:

```
export interface ILogger {
  log(level: string, msg: string, meta?: any)
  debug(msg: string, meta?: any)
  info(msg: string, meta?: any)
  warn(msg: string, meta?: any)
  error(msg: string, meta?: any)
}
```

Here is an `ILogger` implementation that wraps the Winston library. It takes the correlation ID as a constructor parameter, and injects the ID into every log message.

```
class WinstonLogger implements ILogger {
  constructor(private correlationId: string) {}
  log(level: string, msg: string, payload?: any) {
    var meta : any = {};
    if (payload) { meta.payload = payload };
    if (this.correlationId) { meta.correlationId = this.correlationId }
    winston.log(level, msg, meta)
  }

  info(msg: string, payload?: any) {
    this.log('info', msg, payload);
  }
  debug(msg: string, payload?: any) {
    this.log('debug', msg, payload);
  }
  warn(msg: string, payload?: any) {
    this.log('warn', msg, payload);
  }
  error(msg: string, payload?: any) {
    this.log('error', msg, payload);
  }
}
```

The Package service needs to extract the correlation ID from the HTTP request. For example, if you're using linkerd, the correlation ID is found in the `15d-ctx-trace` header. In Koa, the HTTP request is stored in a Context object that gets passed through the request processing pipeline. We can define a middleware function to get the correlation ID from the Context and initialize the logger. (A middleware function in Koa is simply a function that gets executed for each request.)

```
export type CorrelationIdFn = (ctx: Context) => string;

export function logger(level: string, getCorrelationId: CorrelationIdFn) {
  winston.configure({
    level,
    transports: [new (winston.transports.Console)()]
  });
  return async function(ctx: any, next: any) {
    ctx.state.logger = new WinstonLogger(getCorrelationId(ctx));
    await next();
  }
}
```

This middleware invokes a caller-defined function, `getCorrelationId`, to get the correlation ID. Then it creates an instance of the logger and stashes it inside `ctx.state`, which is a key-value dictionary used in Koa to pass information through the pipeline.

The logger middleware is added to the pipeline on startup:

```
app.use(logger(Settings.logLevel(), function (ctx) {
  return ctx.headers[Settings.correlationHeader()];
}));
```

Once everything is configured, it's easy to add logging statements to the code. For example, here is the method that looks up a package. It makes two calls to the `ILogger.info` method.

```
async getById(ctx: any, next: any) {
  var logger : ILogger = ctx.state.logger;
  var packageId = ctx.params.packageId;
  logger.info('Entering getById, packageId = %s', packageId);

  await next();

  let pkg = await this.repository.findPackage(ctx.params.packageId)

  if (pkg == null) {
    logger.info(`getById: %s not found`, packageId);
    ctx.response.status= 404;
    return;
  }

  ctx.response.status = 200;
  ctx.response.body = this.mapPackageDbToApi(pkg);
}
```

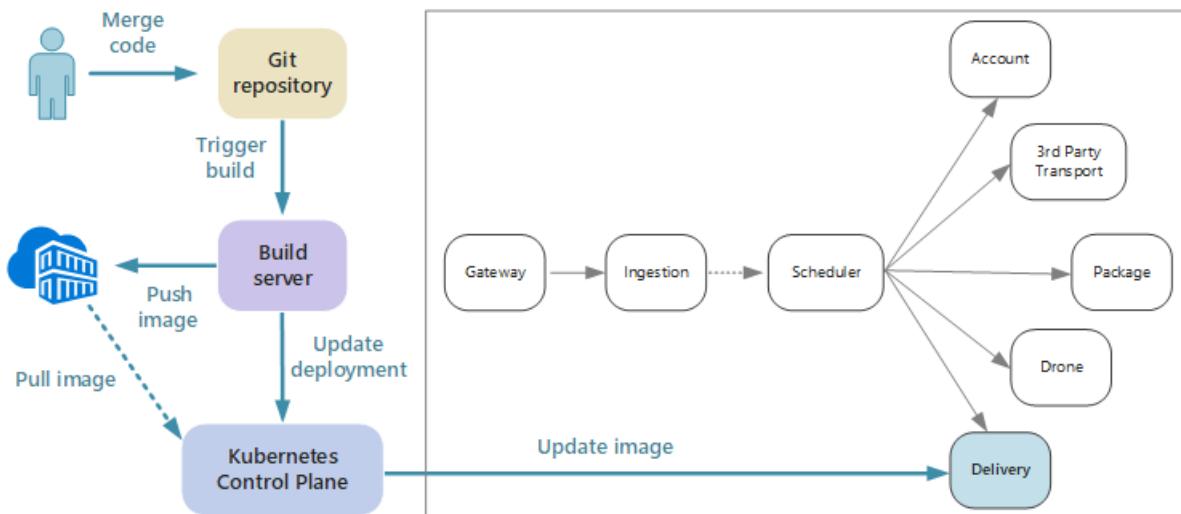
We don't need to include the correlation ID in the logging statements, because that's done automatically by the middleware function. This makes the logging code cleaner, and reduces the chance that a developer will forget to include the correlation ID. And because all of the logging statements use the abstract `ILogger` interface, it would be easy to replace the logger implementation later.

[Continuous integration and delivery](#)

# Designing microservices: Continuous integration

10/24/2018 • 11 minutes to read • [Edit Online](#)

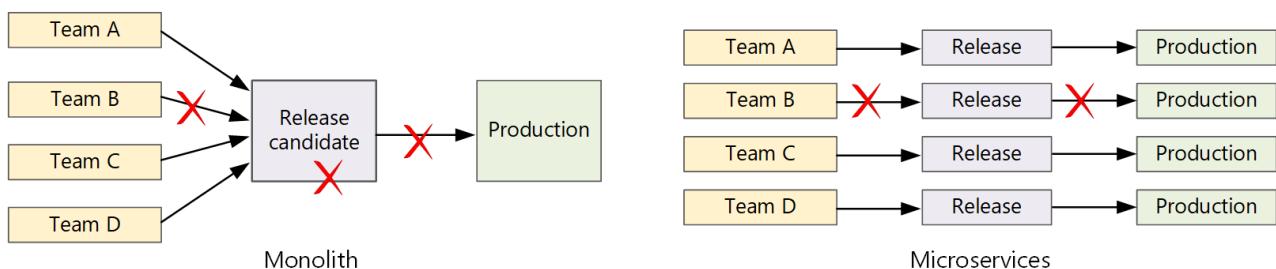
Continuous integration and continuous delivery (CI/CD) are a key requirement for achieving success with microservices. Without a good CI/CD process, you will not achieve the agility that microservices promise. Some of the CI/CD challenges for microservices arise from having multiple code bases and heterogeneous build environments for the various services. This chapter describes the challenges and recommends some approaches to the problem.



Faster release cycles are one of the biggest reasons to adopt a microservices architecture.

In a purely 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. It's true that 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. The CI/CD process is critical to making this possible. Your release pipeline must be automated and highly reliable, so that the risks of deploying updates are minimized. If you are releasing to production daily or multiple times a day, regressions or service disruptions must be very 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.



When we talk about CI/CD, we are really talking about several related processes: Continuous integration, continuous delivery, and continuous deployment.

- Continuous integration means that code changes are frequently merged into the main branch, using automated build and test processes to ensure that code in the main branch is always production-quality.
- Continuous delivery means that 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 means that code changes that pass the CI/CD process are automatically deployed into production.

In the context of Kubernetes and microservices, the CI stage is concerned with building and testing container images, and pushing those images to a container registry. In the deployment stage, pod specs are updated to pick up the latest production image.

## 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. This could 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?
- **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.
- **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 be able to run its own full cluster at production scales, just for testing.
- **Release management.** Every team should have the ability 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. 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 does not mean there should be zero governance.
- **Container image versioning.** During the development and test cycle, the CI/CD process will build many container images. Only some of those are candidates for release, and then only some of those release candidates will get pushed into production. You should have a clear versioning strategy, so that you know which images are currently deployed to production, and can roll back to a previous version if necessary.
- **Service updates.** When you update a service to a new version, it shouldn't break other services that depend on it. If you do a rolling update, there will be a period of time when a mix of versions is running.

These challenges reflect a fundamental tension. On the one hand, teams need to work as independently as possible. On the other hand, some coordination is needed so that a single person can do tasks like running an integration test, redeploying the entire solution to a new cluster, or rolling back a bad update.

## CI/CD approaches for microservices

It's a good practice for every service team to containerize their build environment. This container should have all of the build tools necessary to build the code artifacts for their service. Often you can find an official Docker image for your language and framework. Then you can use `docker run` or Docker Compose to run the build.

With this approach, it's trivial to set up a new build environment. A developer who wants to build your code doesn't need to install a set of build tools, but simply runs the container image. Perhaps more importantly, your build server can be configured to do the same thing. That way, you don't need to install those tools onto the build server, or manage conflicting versions of tools.

For local development and testing, use Docker to run the service inside a container. As part of this process, you may need to run other containers that have mock services or test databases needed for local testing. You could use Docker Compose to coordinate these containers, or use Minikube to run Kubernetes locally.

When the code is ready, open a pull request and merge into master. This will start a job on the build server:

1. Build the code assets.
2. Run unit tests against the code.
3. Build the container image.
4. Test the container image by running functional tests on a running container. This step can catch errors in the Docker file, such as a bad entry point.
5. Push the image to a container registry.
6. Update the test cluster with the new image to run integration tests.

When the image is ready to go into production, update the deployment files as needed to specify the latest image, including any Kubernetes configuration files. Then apply the update to the production cluster.

Here are some recommendations for making deployments more reliable:

- 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.
- Create two separate container registries, one for development/testing and one for production. Don't push an image to the production registry 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.

## 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 update**

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.

Rolling updates are the default behavior in Kubernetes 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 use kubectl to roll back an update if needed.

If your service performs a long startup task, you can define a readiness probe. The readiness probe reports when the container is ready to start receiving traffic. Kubernetes won't send traffic to the pod until the probe reports success.

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. That may or may not cause problems, depending on the scope of the changes between the two versions.

### **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 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.

An advantage of blue-green deployments is that the service switches all the pods at the same time. After the service is updated, all new requests get routed to the new version. One drawback 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. 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 ten 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. Here are some resources that may be helpful:

- Kubernetes without service mesh: [Canary deployments](#)
- Linkerd: [Dynamic request routing](#)
- Istio: [Canary Deployments using Istio](#)

## Conclusion

In recent years, there has been a sea change in the industry, a movement from building *systems of record* to building *systems of engagement*.

Systems of record are traditional back-office data management applications. At the heart of these systems there often sits an RDBMS that is the single source of truth. The term "system of engagement" is credited to Geoffrey Moore, in his 2011 paper *Systems of Engagement and the Future of Enterprise IT*. Systems of engagement are applications focused on communication and collaboration. They connect people in real time. They must be available 24/7. New features are introduced regularly without taking the application offline. Users expect more and are less patient of unexpected delays or downtime.

In the consumer space, a better user experience can have measurable business value. The amount of time that a user engages with an application may translate directly into revenue. And in the realm of business systems, users' expectations have changed. If these systems aim to foster communication and collaboration, they must take their cue from consumer-facing applications.

Microservices are a response to this changing landscape. By decomposing a monolithic application into a group of loosely coupled services, we can control the release cycle of each service, and enable frequent updates without downtime or breaking changes. Microservices also help with scalability, failure isolation, and resiliency. Meanwhile, cloud platforms are making it easier to build and run microservices, with automated provisioning of compute

resources, container orchestrators as a service, and event-driven serverless environments.

But as we've seen, microservices architectures also bring a lot of challenges. To succeed, you must start from a solid design. You must put careful thought into analyzing the domain, choosing technologies, modeling data, designing APIs, and building a mature DevOps culture. We hope that this guide, and the accompanying [reference implementation](#), has helped to illuminate the journey.

# Manage Identity in Multitenant Applications

9/28/2018 • 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:

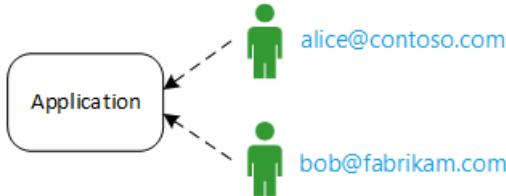
- Users sign in with their organizational credentials.
- Users should have access to their organization's data, but not data that belongs to other tenants.
- An organization can sign up for the application, and then assign application roles to its members.

Azure Active Directory (Azure AD) has some great features that support all of these scenarios.

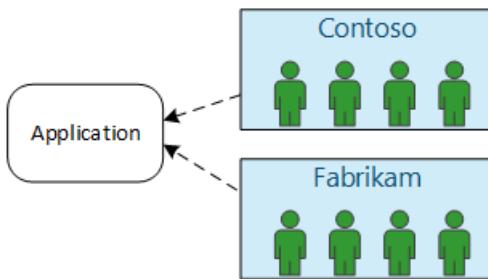
To accompany this series of articles, we created a complete [end-to-end implementation](#) of a multitenant application. The articles reflect what we learned in the process of building the application. To get started with the application, see [Run the Surveys application](#).

## 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.

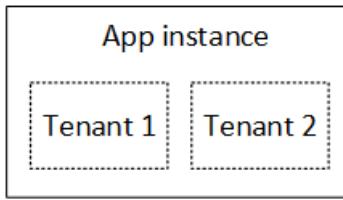
- Alice *should* have access to Contoso data.
- Alice *should not* have access to Fabrikam data.

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.

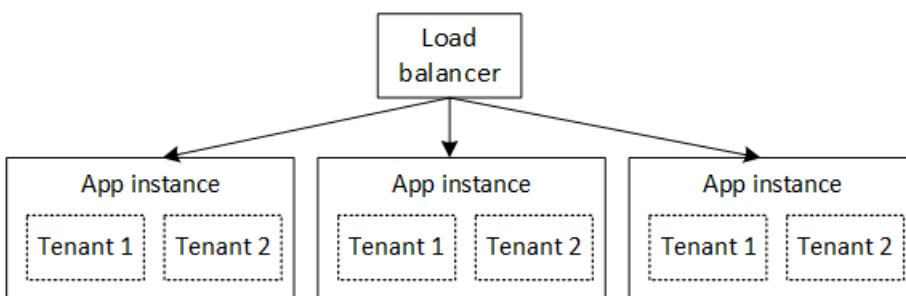


Compare this architecture with a single-tenant architecture, where each tenant has a dedicated physical instance. In a single-tenant architecture, you add tenants by spinning up new instances of the app.



### 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 multi-tenant, 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 login 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-premise Active Directory (AD) can use [Azure AD Connect](#) to sync their on-premise AD with Azure AD.

If a customer with on-premise AD cannot use Azure AD Connect (due to corporate IT policy or other reasons), the SaaS provider can federate with the customer's AD 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

8/14/2017 • 2 minutes to read • [Edit Online](#)



Sample code

Tailspin is a fictitious 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 [Run the Surveys application](#).

## 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`.

The screenshot shows the 'My Surveys' page of the Tailspin Surveys application. At the top, there is a dark navigation bar with the Tailspin logo, 'Tenant Surveys', 'My Surveys', the user's email address 'bob@contoso.com', and a 'Sign Out' link. Below the navigation bar, the main content area has a light gray background. It features three sections: 'Surveys I Created:', 'Surveys I Can Contribute To:', and 'Surveys I Published:'. Each section contains a light blue rectangular box with text indicating the current status. In the 'Surveys I Created:' section, it says 'No surveys have been added yet.' In the 'Surveys I Can Contribute To:' section, it says 'No surveys were found where you are a contributor.' In the 'Surveys I Published:' section, it says 'No surveys have been published yet.' A 'Create Survey' button is located in the bottom right corner of the main content area.

**Surveys I Created:**

No surveys have been added yet.

**Surveys I Can Contribute To:**

No surveys were found where you are a contributor.

**Surveys I Published:**

No surveys have been published yet.

This screenshot shows the Edit Survey page:

## Edit Survey

**Title**

Holiday party

[Edit Title](#)

## Questions

**Question**

Which holiday party activity do you prefer?

**Type**

MultipleChoice

**Answer Choices**

Bowling

Miniature Golf

[Edit](#)[Delete](#)[Add Question](#)

Users can also view any surveys created by other users within the same tenant.

## Tenant Surveys

### Unpublished:

|                             |                         |
|-----------------------------|-------------------------|
| Favorite foods              | <a href="#">Details</a> |
| Restaurant marketing survey | <a href="#">Details</a> |
| Holiday party               | <a href="#">Details</a> |

### Published:

No surveys have been published yet.

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.

The screenshot shows a user interface for adding a survey contributor. At the top, there is a navigation bar with the Tailspin logo, Tenant Surveys, My Surveys, the email address bob@contoso.com, and a Sign Out link. The main content area has a title "Survey Contributor Request" and a sub-section "Add Contributor:". Below this is an input field containing the email address alice@fabrikam.com and a "Add Contributor" button.

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".

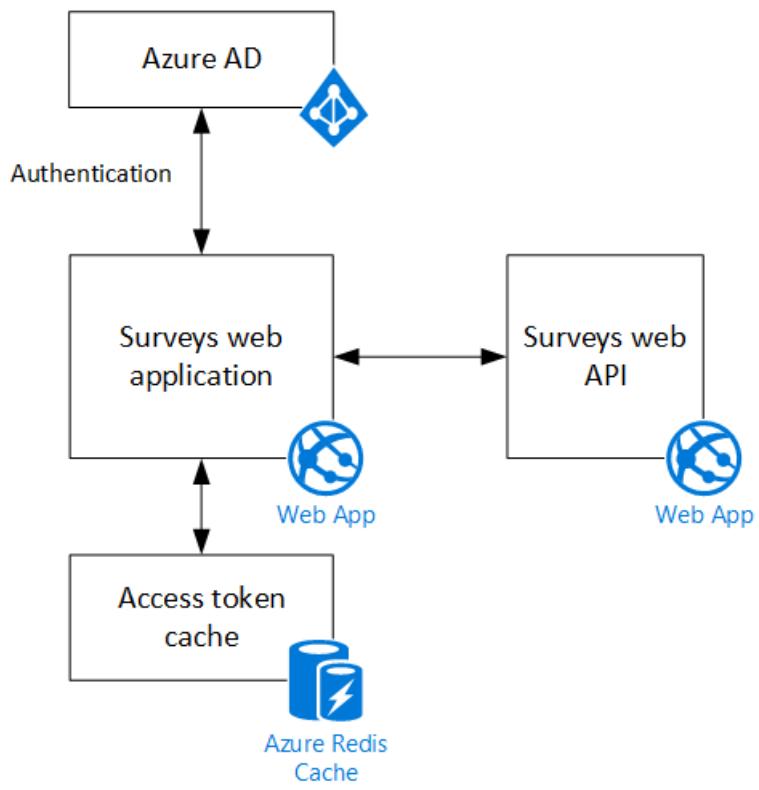
The screenshot shows a user interface for viewing surveys contributed to by Alice. At the top, there is a navigation bar with the Tailspin logo, Tenant Surveys, My Surveys, the email address alice@fabrikam.com, and a Sign Out link. The main content area has a title "My Surveys" and a sub-section "Surveys I Created:" with a message "No surveys have been added yet.". Below this is another section titled "Surveys I Can Contribute To:" with a single survey entry: "Holiday party" followed by "Details" and "Edit" buttons.

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 Redis Cache. The cache enables multiple instances to share the same token cache (e.g., in a server farm).



[Next](#)

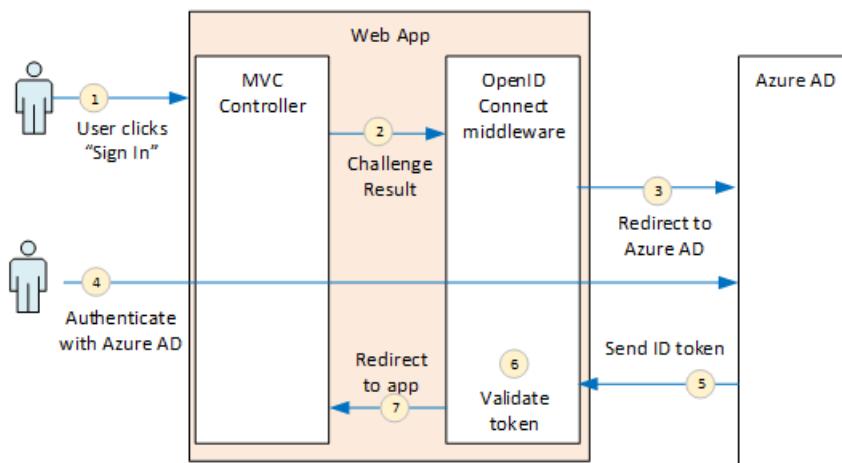
# Authenticate using Azure AD and OpenID Connect

9/28/2018 • 8 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 [Integrating Applications with Azure Active Directory](#), in the section [Adding an Application](#).

See [Run the Surveys application](#) for the specific steps for the Surveys application. Note the following:

- For a multitenant application, you must configure the multi-tenanted 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:

```
app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions {
    ClientId = configOptions.AzureAd.ClientId,
    ClientSecret = configOptions.AzureAd.ClientSecret, // for code flow
    Authority = Constants.AuthEndpointPrefix,
    ResponseType = OpenIdConnectResponseType.CodeIdToken,
    PostLogoutRedirectUri = configOptions.AzureAd.PostLogoutRedirectUri,
    SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme,
    TokenValidationParameters = new TokenValidationParameters { ValidateIssuer = false },
    Events = new SurveyAuthenticationEvents(configOptions.AzureAd, loggerFactory),
});
```

Notice that some of the settings are taken from runtime configuration options. Here's what the middleware options mean:

- **ClientId.** The application's client ID, which you got when you registered the application in Azure AD.
- **Authority.** For a multitenant application, set this to `https://login.microsoftonline.com/common/`. This is the URL for the Azure AD common endpoint, which enables users from any Azure AD tenant to sign in. For more information about the common endpoint, see [this blog post](#).
- In **TokenValidationParameters**, set **ValidateIssuer** to false. That means the app will be responsible for validating the issuer value in the ID token. (The middleware still validates the token itself.) For more information about validating the issuer, see [Issuer validation](#).
- **PostLogoutRedirectUri.** Specify a URL to redirect users after the sign out. This should be a page that allows anonymous requests — typically the home page.
- **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](#).

Also add the Cookie Authentication middleware to the pipeline. This middleware is responsible for writing the user claims to a cookie, and then reading the cookie during subsequent page loads.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions {
    AutomaticAuthenticate = true,
    AutomaticChallenge = true,
    AccessDeniedPath = "/Home/Forbidden",
    CookieSecure = CookieSecurePolicy.Always,

    // The default setting for cookie expiration is 14 days. SlidingExpiration is set to true by default
    ExpireTimeSpan = TimeSpan.FromHours(1),
    SlidingExpiration = true
});
```

## 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

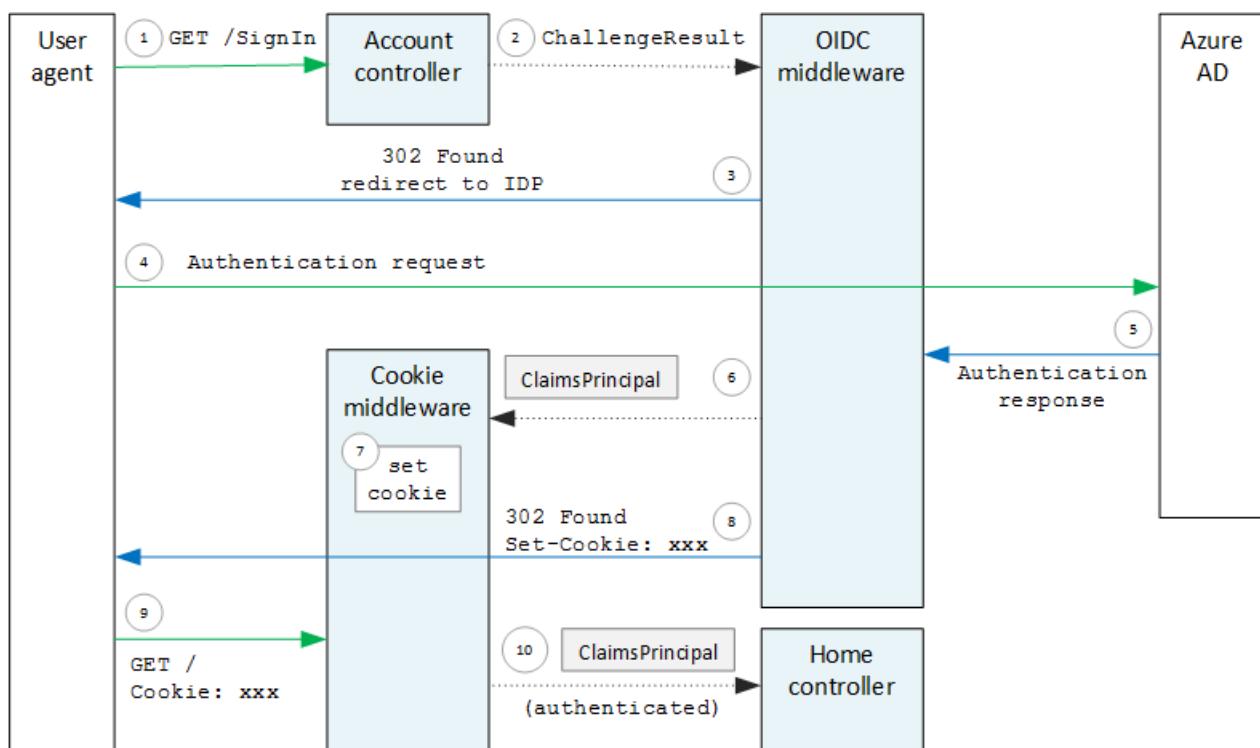
As mentioned, when the user first signs in, the Cookie Authentication middleware writes the user claims to a cookie. After that, HTTP requests are authenticated by reading the cookie.

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. You can access the ticket inside the **AuthenticationValidated** or **TicketReceived** event.

#### 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. For example:

```
app.UseOpenIdConnectAuthentication(options =>
{
    options.ResponseType = "code"; // Authorization code flow

    // Other options
})
```

[Next](#)

# Work with claims-based identities

9/28/2018 • 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 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 AAD 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.<br>Example: <code>"91464657-d17a-4327-91f3-2ed99386406f"</code>                                                                                                                                |
| groups                 | A list of AAD groups of which the user is a member. Example:<br><code>[ "93e8f556-8661-4955-87b6-890bc043c30f", "fc781505-18ef-4a31-a7d5-7d931d7b857e" ]</code>                                                                                                                                                                                                                    |
| iss                    | The <a href="#">issuer</a> of the OIDC token. Example:<br><code>https://sts.windows.net/b9bd2162-77ac-4fb2-8254-5c36e9c0a9c4/</code>                                                                                                                                                                                                                                               |
| name                   | The user's display name. Example: <code>"Alice A."</code>                                                                                                                                                                                                                                                                                                                          |
| oid                    | The object identifier for the user in AAD. 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:<br><code>"59f9d2dc-995a-4ddf-915e-b3bb314a7fa4"</code> |
| roles                  | A list of app roles for the user. Example: <code>[ "SurveyCreator" ]</code>                                                                                                                                                                                                                                                                                                        |

| CLAIM TYPE IN ID TOKEN | DESCRIPTION                                                                                                                 |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| tid                    | Tenant ID. This value is a unique identifier for the tenant in Azure AD. Example:<br>"b9bd2162-77ac-4fb2-8254-5c36e9c0a9c4" |
| unique_name            | A human readable display name of the user. Example:<br>"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. For example, Azure AD sends a "upn" claim that contains the user's email. Other IDPs might send an "email" claim. The following code converts the "upn" claim into an "email" claim:

```
var email = principal.FindFirst(ClaimTypes.Upn)?.Value;
if (!string.IsNullOrWhiteSpace(email))
{
    identity.AddClaim(new Claim(ClaimTypes.Email, email));
}
```

- 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 — e.g., use 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 an additional 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 multi-tenant app, you need to allow for multiple issuers, corresponding to the different tenants. Here is a general approach to use:

- In the OIDC middleware options, set **ValidateIssuer** to false. This turns off the automatic check.
- 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 blacklist 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

9/28/2018 • 6 minutes to read • [Edit Online](#)



[Sample code](#)

This article describes how to implement a *sign-up* process in a multi-tenant 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:

The screenshot shows a consent dialog box with the following content:

**Survey**  
App publisher website: localhost

Survey needs permission to:

- Sign in and read user profile ?

You're signed in as: alice@contoso.com (admin)

If you agree, this app will have access to the specified resources for all users in your organization. No one else will be prompted. [More details](#)

**Accept**   **Cancel**

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.

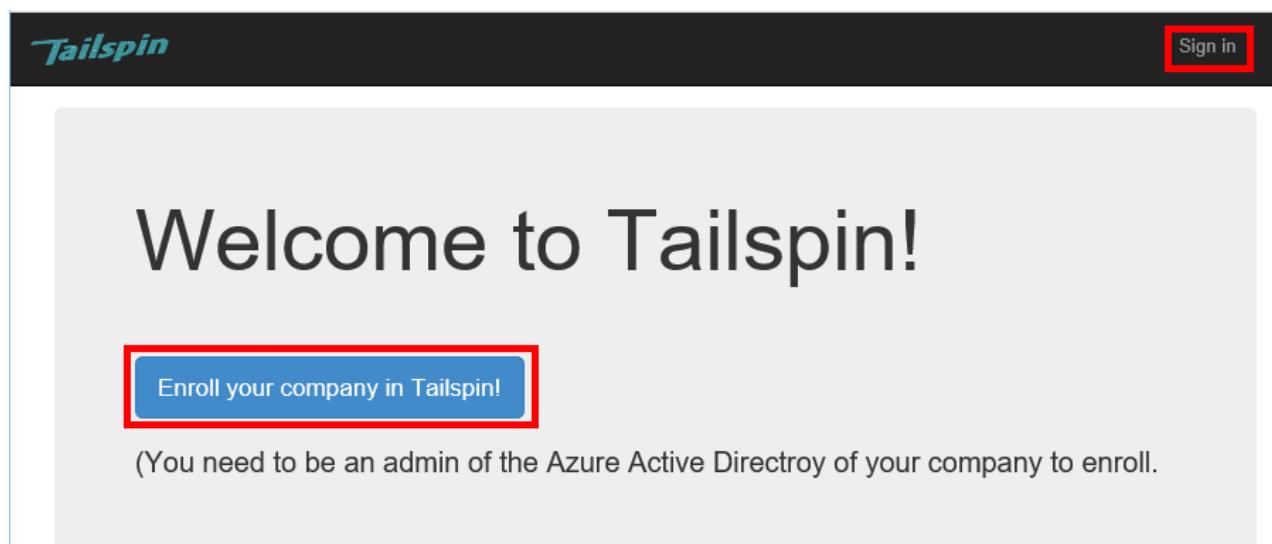
## Implementing tenant sign-up

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).



## Published Surveys

No surveys have been published.

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")
        });
}
```

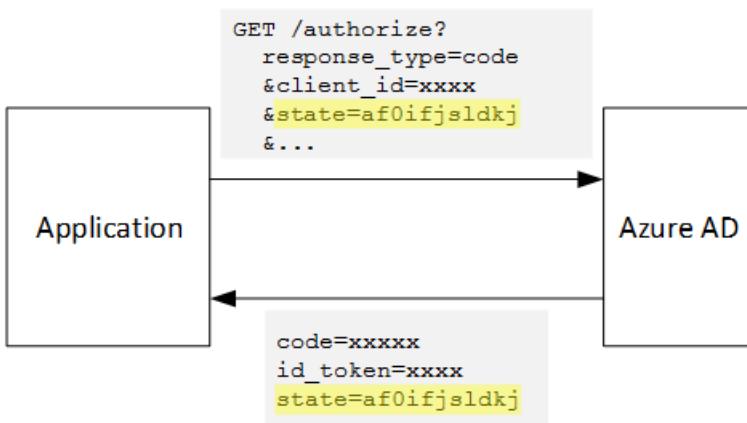
Now compare the `SignUp` action:

```
[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 `RedirectToAuthenticationEndpoint` event. This event is called right before the middleware redirects to the authentication endpoint.

```
public override Task RedirectToAuthenticationEndpoint(RedirectContext context)
{
    if (context.IsSignUpingUp())
    {
        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 IsSignUpingUp(this BaseControlContext context)
{
    Guard.ArgumentNotNull(context, nameof(context));

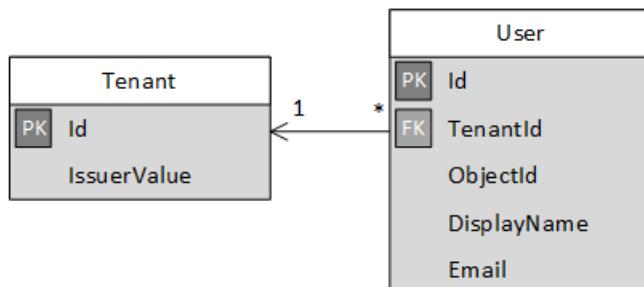
    string signUpValue;
    // Check the HTTP context and convert to string
    if ((context.Ticket == null) ||
        (!context.Ticket.Properties.Items.TryGetValue("signup", out signUpValue)))
    {
        return false;
    }

    // We have found the value, so see if it's valid
    bool isSignUpingUp;
    if (!bool.TryParse(signUpValue, out isSignUpingUp))
    {
        // The value for signup is not a valid boolean, throw
        throw new InvalidOperationException($"'{signUpValue}' is an invalid boolean value");
    }

    return isSignUpingUp;
}
```

## 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/<tenantID>> 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.AuthenticationTicket.Principal;
    var userId = principal.GetObjectIdentifierValue();
    var tenantManager = context.HttpContext.RequestServices.GetService<TenantManager>();
    var userManager = context.HttpContext.RequestServices.GetService<UserManager>();
    var issuerValue = principal.GetIssuerValue();
    _logger.AuthenticationValidated(userId, issuerValue);

    // Normalize the claims first.
    NormalizeClaims(principal);
    var tenant = await tenantManager.FindByIssuerValueAsync(issuerValue)
        .ConfigureAwait(false);

    if (context.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.Ticket, 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(BaseControlContext context, TenantManager tenantManager)
{
    Guard.ArgumentNotNull(context, nameof(context));
    Guard.ArgumentNotNull(tenantManager, nameof(tenantManager));

    var principal = context.Ticket.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

5/21/2018 • 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 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 AD admin.

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 AD tenant, the roles go away.
- The application doesn't need any extra Active Directory permissions, other than reading the user's profile.

Drawbacks:

- Customers without Azure AD Premium cannot assign security groups to roles. For these customers, all user assignments must be done by an AD administrator.
- If you have a backend web API, which is separate from the web app, then 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-b942bbd4cf1",
    "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 AD tenant. At this point, an AD admin for that tenant can assign users to roles.

#### NOTE

As noted earlier, customers with Azure AD Premium can also assign security groups to roles.

The following screenshot from the Azure portal shows users and groups for the Survey application. Admin and Creator are groups, assigned to SurveyAdmin and SurveyCreator roles respectively. Alice is a user who was assigned directly to the SurveyAdmin role. Bob and Charles are users that have not been directly assigned to a 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 a role yet.

| NAME    | TYPE |
|---------|------|
| Charles | User |

#### NOTE

An alternative approach is for the application to assign roles programmatically, using the Azure AD Graph API. However, this requires the application to obtain write permissions for the customer's AD directory. An application with those permissions could do a lot of mischief — the customer is trusting the app not to mess up their directory. Many customers might be unwilling to grant this level of access.

**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`.

A user can have 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 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.
- If the customer removes the application from their AD tenant, the security groups are left in their AD directory.

### Implementation

In the application manifest, set the `groupMembershipClaims` property to "SecurityGroup". This is needed to get group membership claims from AAD.

```
{
  // ...
  "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 Azure AD Graph API. This would be less error prone. However, it requires the application to obtain "read and write all groups" permissions for the customer's AD 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 [Authorization in Cloud Applications using AD Groups], under the section titled "Groups claim overage".
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 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 AD directory, as users are added or removed.

[Next](#)

# Role-based and resource-based authorization

6/22/2018 • 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 (e.g.,  $age \geq 21$ ) 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 multi-tenant 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 is assigned when the survey is created.)

The next step is to check the operation (read, update, delete, etc) 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

7/5/2018 • 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 OAuth2 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

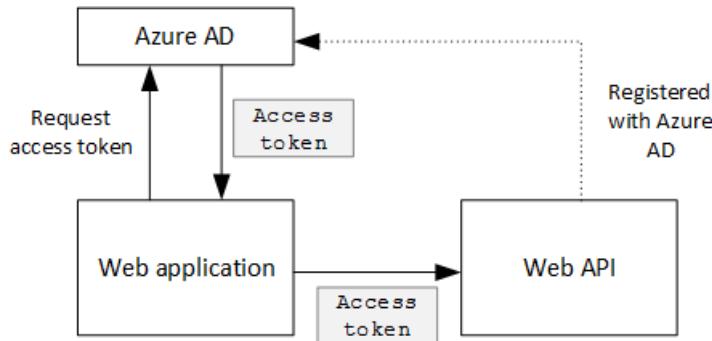
- The web API does not get information about the user.
- The web API cannot perform any authorization based on the user identity. All authorization decisions are made by the web application.

- 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 the IDP, 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.



## 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 [Update the application manifests](#).
3. Give the web application permission to call the web API. In the Azure Management Portal, you can set two types of permissions: "Application Permissions" for application identity (client credential flow), or "Delegated Permissions" for delegated user identity.

Required permissions
□ X

+ Add  
 ➡ Grant Permissions

| API                            | APPLICATION PERMI... | DELEGATED PERMISS... |
|--------------------------------|----------------------|----------------------|
| Surveys.WebAPI                 | 0                    | 1                    |
| Windows Azure Active Directory | 0                    | 1                    |

## 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 [Azure AD Authentication Library \(ADAL\) for .NET](#).

In the OAuth 2 authorization code flow, the application exchanges an authorization code for an access token. The following code uses ADAL to get the access token. This code is called during the `AuthorizationCodeReceived` event.

```

// The OpenID Connect middleware sends this event when it gets the authorization code.
public override async Task AuthorizationCodeReceived(AuthorizationCodeReceivedContext context)
{
    string authorizationCode = context.ProtocolMessage.Code;
    string authority = "https://login.microsoftonline.com/" + tenantID
    string resourceID = "https://tailspin.onmicrosoft.com/surveys.webapi" // App ID URI
    ClientCredential credential = new ClientCredential(clientId, clientSecret);

    AuthenticationContext authContext = new AuthenticationContext(authority, tokenCache);
    AuthenticationResult authResult = await authContext.AcquireTokenByAuthorizationCodeAsync(
        authorizationCode, new Uri(redirectUri), credential, resourceID);

    // If successful, the token is in authResult.AccessToken
}

```

Here are the various parameters that are needed:

- `authority`. Derived from the tenant ID of the signed in user. (Not the tenant ID of the SaaS provider)
- `authorizationCode`. the auth code that you got back from the IDP.
- `clientId`. The web application's client ID.
- `clientSecret`. The web application's client secret.
- `redirectUri`. The redirect URI that you set for OpenID connect. This is where the IDP calls back with the token.
- `resourceID`. The App ID URI of the web API, which you created when you registered the web API in Azure AD
- `tokenCache`. An object that caches the access tokens. See [Token caching](#).

If `AcquireTokenByAuthorizationCodeAsync` succeeds, ADAL caches the token. Later, you can get the token from the cache by calling `AcquireTokenSilentAsync`:

```

AuthenticationContext authContext = new AuthenticationContext(authority, tokenCache);
var result = await authContext.AcquireTokenSilentAsync(resourceID, credential, new UserIdentifier(userId,
UserIdentifierType.UniqueId));

```

where `userId` is the user's object ID, which is found in the  
`http://schemas.microsoft.com/identity/claims/objectidentifier` claim.

## Using the access token to call the web API

Once you have the token, send it in the Authorization header of the HTTP requests to the web API.

```
Authorization: Bearer xxxxxxxxxxxx
```

The following extension method from the Surveys application sets the Authorization header on an HTTP request, using the **HttpClient** class.

```

public static async Task<HttpResponseMessage> SendRequestWithBearerTokenAsync(this HttpClient httpClient,
    HttpMethod method, string path, object requestBody, string accessToken, CancellationToken ct)
{
    var request = new HttpRequestMessage(method, path);
    if (requestBody != null)
    {
        var json = JsonConvert.SerializeObject(requestBody, Formatting.None);
        var content = new StringContent(json, Encoding.UTF8, "application/json");
        request.Content = content;
    }

    request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", accessToken);
    request.Headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

    var response = await httpClient.SendAsync(request, ct);
    return response;
}

```

## Authenticating in the web API

The web API has to authenticate the bearer token. In ASP.NET Core, you can use the [Microsoft.AspNetCore.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.

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ApplicationDbContext dbContext,
    ILoggerFactory loggerFactory)
{
    // ...

    app.UseJwtBearerAuthentication(new JwtBearerOptions {
        Audience = configOptions.AzureAd.WebApiResourceId,
        Authority = Constants.AuthEndpointPrefix,
        TokenValidationParameters = new TokenValidationParameters {
            ValidateIssuer = false
        },
        Events= new SurveysJwtBearerEvents(loggerFactory.CreateLogger<SurveysJwtBearerEvents>())
    });

    // ...
}

```

- **Audience**. Set this to the App ID URL for the web API, which you created when you registered the web API with Azure AD.
- **Authority**. For a multitenant application, set this to <https://login.microsoftonline.com/common/>.
- **TokenValidationParameters**. For a multitenant application, set **ValidateIssuer** to false. That means the application will validate the issuer.
- **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);
            });
    });

    // ...
}
```

[Next](#)

# Cache access tokens

12/7/2017 • 4 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 [Azure AD Authentication Library](#) (ADAL) automatically caches tokens obtained from Azure AD, including refresh tokens.

ADAL provides a default token cache implementation. However, this token cache is intended for native client apps, and is **not** suitable for web apps:

- It is a static instance, and not thread safe.
- It doesn't scale to large numbers of users, because tokens from all users go into the same dictionary.
- It can't be shared across web servers in a farm.

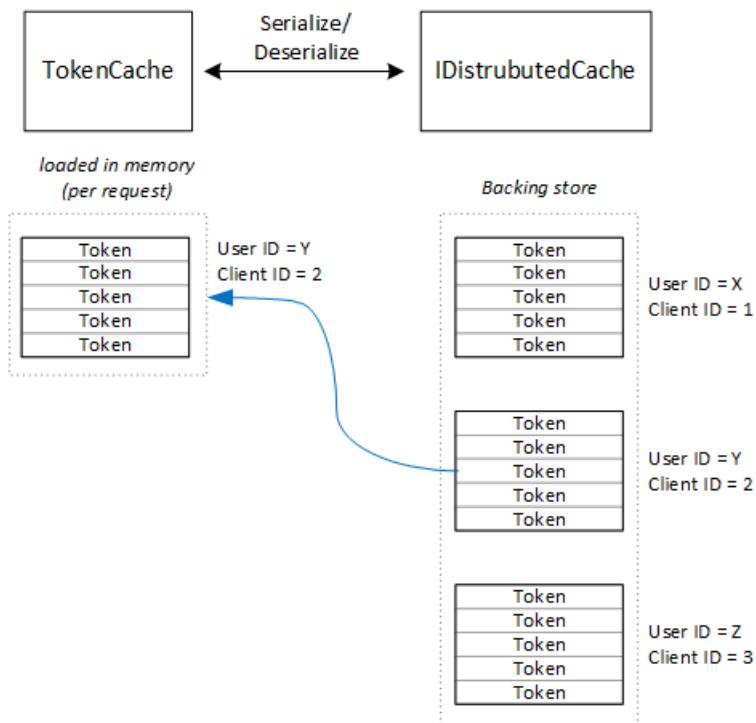
Instead, you should implement a custom token cache that derives from the ADAL `TokenCache` class but is suitable for a server environment and provides the desirable level of isolation between tokens for different users.

The `TokenCache` class stores a dictionary of tokens, indexed by issuer, resource, client ID, and user. A custom token cache should write this dictionary to a backing store, such as a Redis cache.

In the Tailspin Surveys application, the `DistributedTokenCache` class implements the token cache. This implementation uses the `IDistributedCache` abstraction from ASP.NET Core. That way, any `IDistributedCache` implementation can be used as a backing store.

- By default, the Surveys app uses a Redis cache.
- 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.)

`DistributedTokenCache` stores the cache data as key/value pairs in the backing store. The key is the user ID plus client ID, so the backing store holds separate cache data for each unique combination of user/client.



The backing store is partitioned by user. For each HTTP request, the tokens for that user are read from the backing store and loaded into the `TokenCache` dictionary. If Redis is used as the backing store, every server instance in a server farm reads/writes to the same cache, and this approach scales to many users.

## 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. For that reason, the `DistributedTokenCache` encrypts everything that it writes to the backing store. Encryption is done using the ASP.NET Core [data protection](#) APIs.

### NOTE

If you deploy to Azure Web Sites, the encryption keys are backed up to network storage and synchronized across all machines (see [Key management and lifetime](#)). By default, keys are not encrypted when running in Azure Web Sites, but you can [enable encryption using an X.509 certificate](#).

## DistributedTokenCache implementation

The `DistributedTokenCache` class derives from the ADAL `TokenCache` class.

In the constructor, the `DistributedTokenCache` class creates a key for the current user and loads the cache from the backing store:

```
public DistributedTokenCache(
    ClaimsPrincipal claimsPrincipal,
    IDistributedCache distributedCache,
    ILoggerFactory loggerFactory,
    IDataProtectionProvider dataProtectionProvider)
    : base()
{
    _claimsPrincipal = claimsPrincipal;
    _cacheKey = BuildCacheKey(_claimsPrincipal);
    _distributedCache = distributedCache;
    _logger = loggerFactory.CreateLogger<DistributedTokenCache>();
    _protector = dataProtectionProvider.CreateProtector(typeof(DistributedTokenCache).FullName);
    AfterAccess = AfterAccessNotification;
    LoadFromCache();
}
```

The key is created by concatenating the user ID and client ID. Both of these are taken from claims found in the user's `ClaimsPrincipal`:

```
private static string BuildCacheKey(ClaimsPrincipal claimsPrincipal)
{
    string clientId = claimsPrincipal.FindFirstValue("aud", true);
    return string.Format(
        "UserId:{0}::ClientId:{1}",
        claimsPrincipal.GetObjectIdentifierValue(),
        clientId);
}
```

To load the cache data, read the serialized blob from the backing store, and call `TokenCache.Deserialize` to convert the blob into cache data.

```

private void LoadFromCache()
{
    byte[] cacheData = _distributedCache.Get(_cacheKey);
    if (cacheData != null)
    {
        this.Deserialize(_protector.Unprotect(cacheData));
    }
}

```

Whenever ADAL access the cache, it fires an `AfterAccess` event. If the cache data has changed, the `HasStateChanged` property is true. In that case, update the backing store to reflect the change, and then set `HasStateChanged` to false.

```

public void AfterAccessNotification(TokenCacheNotificationArgs args)
{
    if (this.HasStateChanged)
    {
        try
        {
            if (this.Count > 0)
            {
                _distributedCache.Set(_cacheKey, _protector.Protect(this.Serialize()));
            }
            else
            {
                // There are no tokens for this user/client, so remove the item from the cache.
                _distributedCache.Remove(_cacheKey);
            }
            this.HasStateChanged = false;
        }
        catch (Exception exp)
        {
            _logger.WriteToCacheFailed(exp);
            throw;
        }
    }
}

```

TokenCache sends two other events:

- `BeforeWrite`. Called immediately before ADAL writes to the cache. You can use this to implement a concurrency strategy
- `BeforeAccess`. Called immediately before ADAL reads from the cache. Here you can reload the cache to get the latest version.

In our case, we decided not to handle these two events.

- For concurrency, last write wins. That's OK, because tokens are stored independently for each user + client, so a conflict would only happen if the same user had two concurrent login sessions.
- For reading, we load the cache on every request. Requests are short lived. If the cache gets modified in that time, the next request will pick up the new value.

[Next](#)

# Use client assertion to get access tokens from Azure AD

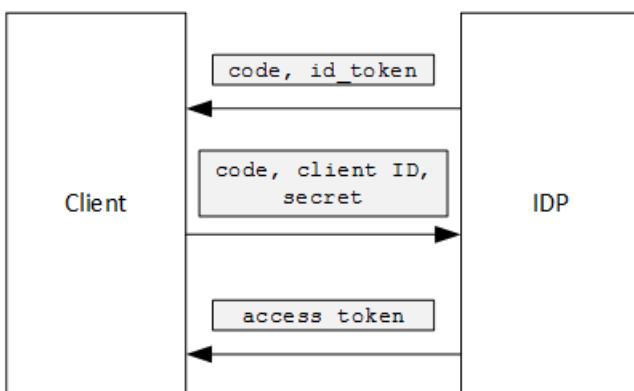
8/14/2017 • 2 minutes to read • [Edit Online](#)



[Sample code](#)

## Background

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

One way to authenticate the client is by using a client secret. That's how the [Tailspin Surveys](#) application is configured 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. Further, there is always a temptation to check secrets into source control (e.g., in deployment scripts), share them by email, and so on.

For additional security, you can use [client assertion](#) instead of a client secret. With client assertion, the client uses an X.509 certificate to prove the token request came from the client. The client certificate is installed on the web server. Generally, it will be easier to restrict access to the certificate, than to ensure that nobody inadvertently reveals a client secret. For more information about configuring certificates in a web app, see [Using Certificates in Azure Websites Applications](#)

Here is a token request using client assertion:

```
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_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer
&client_assertion=eyJhbGci...
&grant_type=authorization_code
&code= PG8wJG6Y...
```

Notice that the `client_secret` parameter is no longer used. Instead, the `client_assertion` parameter contains a JWT token that was signed using the client certificate. The `client_assertion_type` parameter specifies the type of assertion — in this case, JWT token. The server validates the JWT token. If the JWT token is invalid, the token request returns an error.

**NOTE**

X.509 certificates are not the only form of client assertion; we focus on it here because it is supported by Azure AD.

At run time, the web application reads the certificate from the certificate store. The certificate must be installed on the same machine as the web app.

The Surveys application includes a helper class that creates a `ClientAssertionCertificate` that you can pass to the `AuthenticationContext.AcquireTokenSilentAsync` method to acquire a token from Azure AD.

```
public class CertificateCredentialService : ICredentialService
{
    private Lazy<Task<AdalCredential>> _credential;

    public CertificateCredentialService(IOptions<ConfigurationOptions> options)
    {
        var aadOptions = options.Value?.AzureAd;
        _credential = new Lazy<Task<AdalCredential>>(() =>
        {
            X509Certificate2 cert = CertificateUtility.FindCertificateByThumbprint(
                aadOptions.Asymmetric.StoreName,
                aadOptions.Asymmetric.StoreLocation,
                aadOptions.Asymmetric.CertificateThumbprint,
                aadOptions.Asymmetric.ValidationRequired);
            string password = null;
            var certBytes = CertificateUtility.ExportCertificateWithPrivateKey(cert, out password);
            return Task.FromResult(new AdalCredential(new ClientAssertionCertificate(aadOptions.ClientId, new
X509Certificate2(certBytes, password)))));
        });
    }

    public async Task<AdalCredential> GetCredentialsAsync()
    {
        return await _credential.Value;
    }
}
```

For information about setting up client assertion in the Surveys application, see [Use Azure Key Vault to protect application secrets](#).

**Next**

# Use Azure Key Vault to protect application secrets

9/28/2018 • 7 minutes to read • [Edit Online](#)



[Sample code](#)

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](#).

## Setting up Key Vault in the Surveys app

Prerequisites:

- Install the [Azure Resource Manager Cmdlets](#).
- Configure the Surveys application as described in [Run the Surveys application](#).

High-level steps:

1. Set up an admin user in the tenant.
2. Set up a client certificate.
3. Create a key vault.
4. Add configuration settings to your key vault.
5. Uncomment the code that enables key vault.
6. Update the application's user secrets.

## Set up an admin user

### NOTE

To create a key vault, you must use an account which can manage your Azure subscription. Also, any application that you authorize to read from the key vault must be registered in the same tenant as that account.

In this step, you will make sure that you can create a key vault while signed in as a user from the tenant where the Surveys app is registered.

Create an administrator user within the Azure AD tenant where the Surveys application is registered.

1. Log into the [Azure portal](#).
2. Select the Azure AD tenant where your application is registered.
3. Click **More service > SECURITY + IDENTITY > Azure Active Directory > User and groups > All users**.
4. At the top of the portal, click **New user**.
5. Fill in the fields and assign the user to the **Global administrator** directory role.
6. Click **Create**.

### User

tailspin

\* Name ⓘ  
John ✓

\* User name ⓘ  
johnadmin@tailspinpnp.onmicrosoft.com ✓

Profile ⓘ >  
Not configured

Properties ⓘ >  
Default

Groups ⓘ >  
0 groups selected

Directory role  
User >

Password

**Create**

### Directory role

Directory role ⓘ

User

Global administrator

Limited administrator

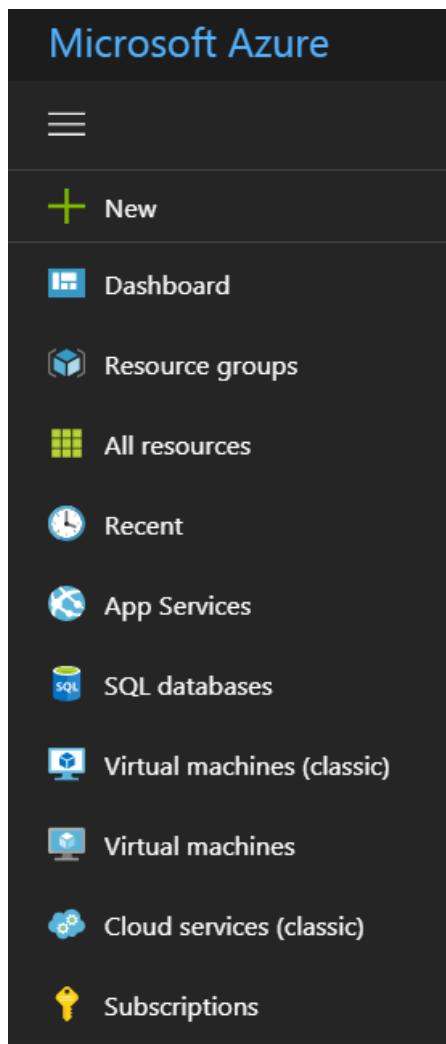
Global administrators have full control over all directory resources.

[Learn more about directory roles](#)

**Ok**

Now assign this user as the subscription owner.

1. On the Hub menu, select **Subscriptions**.



2. Select the subscription that you want the administrator to access.
3. In the subscription blade, select **Access control (IAM)**.
4. Click **Add**.
5. Under **Role**, select **Owner**.
6. Type the email address of the user you want to add as owner.
7. Select the user and click **Save**.

#### Set up a client certificate

1. Run the PowerShell script [/Scripts/Setup-KeyVault.ps1](#) as follows:

```
.\Setup-KeyVault.ps1 -Subject <>subject><
```

For the `Subject` parameter, enter any name, such as "surveysapp". The script generates a self-signed certificate and stores it in the "Current User/Personal" certificate store. The output from the script is a JSON fragment. Copy this value.

2. In the [Azure portal](#), switch to the directory where the Surveys application is registered, by selecting your account in the top right corner of the portal.
3. Select **Azure Active Directory** > **App Registrations** > Surveys
4. Click **Manifest** and then **Edit**.
5. Paste the output from the script into the `keyCredentials` property. It should look similar to the following:

```
"keyCredentials": [
  {
    "type": "AsymmetricX509Cert",
    "usage": "Verify",
    "keyId": "29d4f7db-0539-455e-b708-....",
    "customKeyIdentifier": "ZEPpP/+KJe2fVDBNaPNOTDoJMac=",
    "value": "MIIDAjCCAeqgAwIBAgIQFxeRiU59eL...."
  }
],
```

6. Click **Save**.
7. Repeat steps 3-6 to add the same JSON fragment to the application manifest of the web API (Surveys.WebAPI).
8. From the PowerShell window, run the following command to get the thumbprint of the certificate.

```
certutil -store -user my [subject]
```

For `[subject]`, use the value that you specified for Subject in the PowerShell script. The thumbprint is listed under "Cert Hash(sha1)". Copy this value. You will use the thumbprint later.

### Create a key vault

1. Run the PowerShell script [/Scripts/Setup-KeyVault.ps1](#) as follows:

```
.\Setup-KeyVault.ps1 -KeyVaultName <>key vault name>> -ResourceGroupName <>resource group name>> - Location <>location>>
```

When prompted for credentials, sign in as the Azure AD user that you created earlier. The script creates a new resource group, and a new key vault within that resource group.

2. Run SetupKeyVault.ps again as follows:

```
.\Setup-KeyVault.ps1 -KeyVaultName <>key vault name>> -ApplicationIds @("<>Surveys app id>>", "<>Surveys.WebAPI app ID>>")
```

Set the following parameter values:

```
* key vault name = The name that you gave the key vault in the previous step.  
* Surveys app ID = The application ID for the Surveys web application.  
* Surveys.WebApi app ID = The application ID for the Surveys.WebAPI application.
```

Example:

```
.\Setup-KeyVault.ps1 -KeyVaultName tailspinkv -ApplicationIds @("f84df9d1-91cc-4603-b662-302db51f1031", "8871a4c2-2a23-4650-8b46-0625ff3928a6")
```

This script authorizes the web app and web API to retrieve secrets from your key vault. See [Get started with Azure Key Vault](#) for more information.

### Add configuration settings to your key vault

1. Run SetupKeyVault.ps as follows::

```
.\Setup-KeyVault.ps1 -KeyVaultName <>key vault name> -KeyName Redis--Configuration -KeyValue "<<Redis  
DNS name>>.redis.cache.windows.net,password=<<Redis access key>>,ssl=true"
```

where

- key vault name = The name that you gave the key vault in the previous step.
- Redis DNS name = The DNS name of your Redis cache instance.
- Redis access key = The access key for your Redis cache instance.

2. At this point, it's a good idea to test whether you successfully stored the secrets to key vault. Run the following PowerShell command:

```
Get-AzureKeyVaultSecret <>key vault name> Redis--Configuration | Select-Object *
```

3. Run SetupKeyVault.ps again to add the database connection string:

```
.\Setup-KeyVault.ps1 -KeyVaultName <>key vault name> -KeyName Data--SurveysConnectionString -KeyValue  
<>DB connection string>> -ConfigName "Data:SurveysConnectionString"
```

where <>DB connection string>> is the value of the database connection string.

For testing with the local database, copy the connection string from the Tailspin.Surveys.Web/appsettings.json file. If you do that, make sure to change the double backslash ('\\') into a single backslash. The double backslash is an escape character in the JSON file.

Example:

```
.\Setup-KeyVault.ps1 -KeyVaultName mykeyvault -KeyName Data--SurveysConnectionString -KeyValue "Server=  
(localdb)\MSSQLLocalDB;Database=Tailspin.SurveysDB;Trusted_Connection=True;MultipleActiveResultSets=true"
```

### Uncomment the code that enables Key Vault

1. Open the Tailspin.Surveys solution.
2. In Tailspin.Surveys.Web/Startup.cs, locate the following code block and uncomment it.

```
//var config = builder.Build();  
//builder.AddAzureKeyVault(  
//    $"https://{{config["KeyVault:Name"]}}.vault.azure.net/",  
//    config["AzureAd:ClientId"],  
//    config["AzureAd:ClientSecret"]);
```

3. In Tailspin.Surveys.Web/Startup.cs, locate the code that registers the `ICredentialService`. Uncomment the line that uses `CertificateCredentialService`, and comment out the line that uses `ClientCredentialService`:

```
// Uncomment this:  
services.AddSingleton<ICredentialService, CertificateCredentialService>();  
// Comment out this:  
//services.AddSingleton<ICredentialService, ClientCredentialService>();
```

This change enables the web app to use [Client assertion](#) to get OAuth access tokens. With client assertion, you don't need an OAuth client secret. Alternatively, you could store the client secret in key vault. However, key vault and client assertion both use a client certificate, so if you enable key vault, it's a good practice to enable client assertion as well.

## Update the user secrets

In Solution Explorer, right-click the Tailspin.Surveys.Web project and select **Manage User Secrets**. In the secrets.json file, delete the existing JSON and paste in the following:

```
...
{
  "AzureAd": {
    "ClientId": "[Surveys web app client ID]",
    "ClientSecret": "[Surveys web app client secret]",
    "PostLogoutRedirectUri": "https://localhost:44300/",
    "WebApiResourceId": "[App ID URI of your Surveys.WebAPI application]",
    "Asymmetric": {
      "CertificateThumbprint": "[certificate thumbprint. Example: 105b2ff3bc842c53582661716db1b7cdc6b43ec9]",
      "StoreName": "My",
      "StoreLocation": "CurrentUser",
      "ValidationRequired": "false"
    }
  },
  "KeyVault": {
    "Name": "[key vault name]"
  }
}
...
```

Replace the entries in [square brackets] with the correct values.

- `AzureAd:ClientId` : The client ID of the Surveys app.
- `AzureAd:ClientSecret` : The key that you generated when you registered the Surveys application in Azure AD.
- `AzureAd:WebApiResourceId` : The App ID URI that you specified when you created the Surveys.WebAPI application in Azure AD.
- `Asymmetric:CertificateThumbprint` : The certificate thumbprint that you got previously, when you created the client certificate.
- `KeyVault:Name` : The name of your key vault.

### NOTE

`Asymmetric:ValidationRequired` is false because the certificate that you created previously was not signed by a root certificate authority (CA). In production, use a certificate that is signed by a root CA and set `ValidationRequired` to true.

Save the updated secrets.json file.

Next, in Solution Explorer, right-click the Tailspin.Surveys.WebApi project and select **Manage User Secrets**. Delete the existing JSON and paste in the following:

```
{  
  "AzureAd": {  
    "ClientId": "[Surveys.WebAPI client ID]",  
    "WebApiResourceId": "https://tailspin5.onmicrosoft.com/surveys.webapi",  
    "Asymmetric": {  
      "CertificateThumbprint": "[certificate thumbprint]",  
      "StoreName": "My",  
      "StoreLocation": "CurrentUser",  
      "ValidationRequired": "false"  
    }  
  },  
  "KeyVault": {  
    "Name": "[key vault name]"  
  }  
}
```

Replace the entries in [square brackets] and save the secrets.json file.

**NOTE**

For the web API, make sure to use the client ID for the Surveys.WebAPI application, not the Surveys application.

**Next**

# Federate with a customer's AD FS

9/28/2018 • 6 minutes to read • [Edit Online](#)

This article describes how a multi-tenant SaaS application can support authentication via Active Directory Federation Services (AD FS), in order to federate with a customer's AD FS.

## Overview

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-premise Active Directory on a corporate intranet?

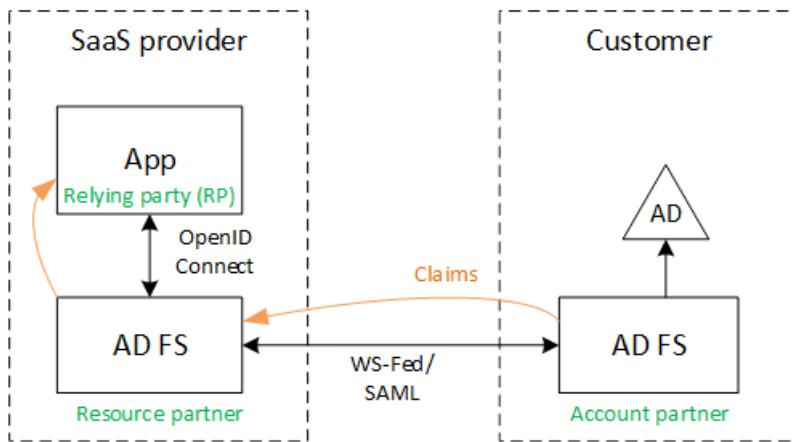
One option is for these customers to sync their on-premise AD with Azure AD, using [Azure AD Connect](#). However, some customers may be unable to use this approach, due to corporate IT policy or other reasons. In that case, another option is to federate through Active Directory Federation Services (AD 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.

ASP.NET Core does not include out-of-the-box support for WS-Federation.

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 | <b>Authentication instant.</b> The time at which authentication occurred.                         |
| c_hash                | Code hash value. This is a hash of the token contents.                                            |
| exp                   | <b>Expiration time.</b> 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        | <b>Name identifier.</b> 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.                  |

| CLAIM   | DESCRIPTION                                                                                                                             |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------|
| upn     | User principal name (UPN). Example:<br>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-premise 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, NSG's, azure VM's 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 AAD). 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 ADFS:

1. Add a claims provider trust.
2. Add claims rules.
3. Enable home-realm discovery.

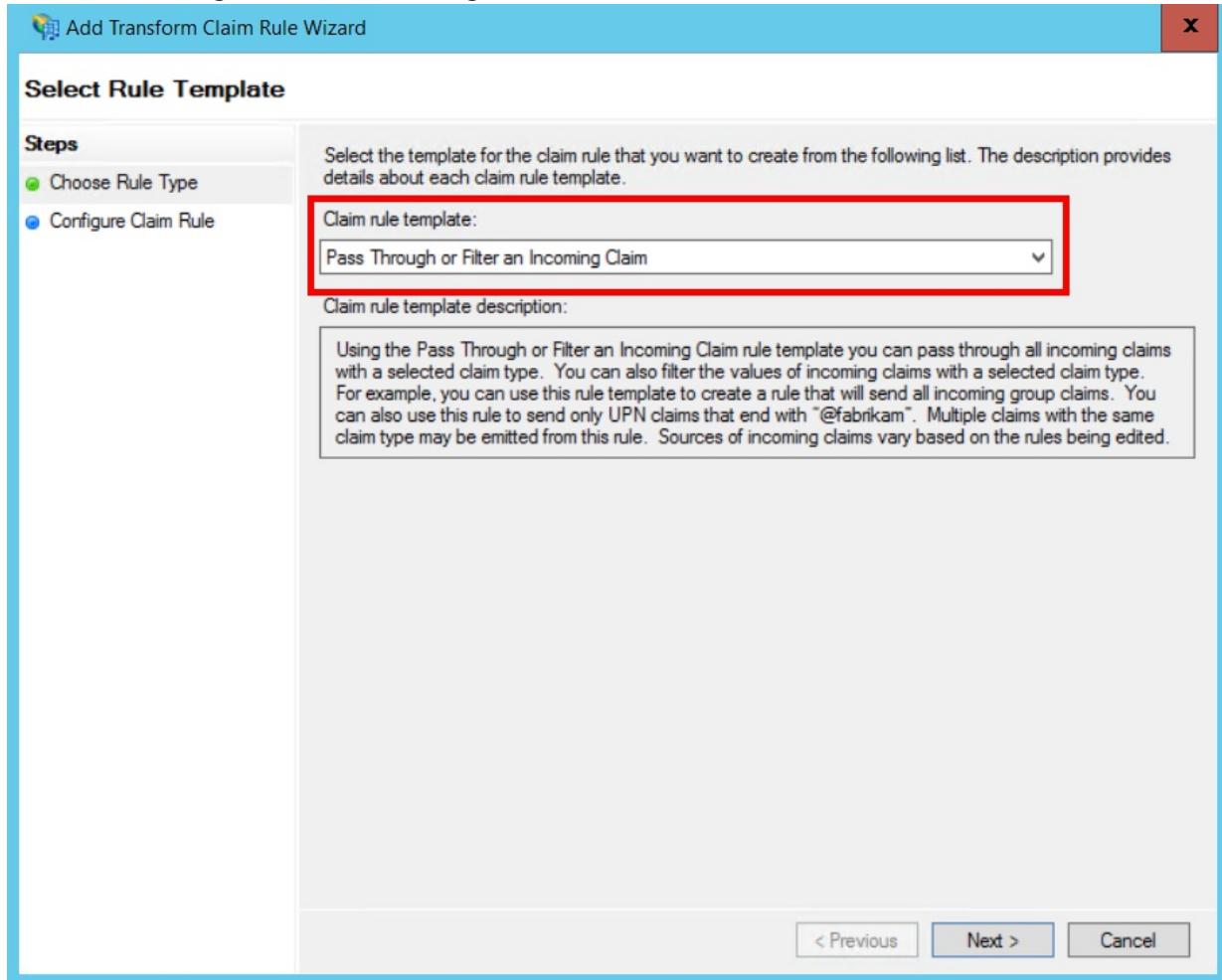
Here are the steps in more detail.

### Add the claims provider trust

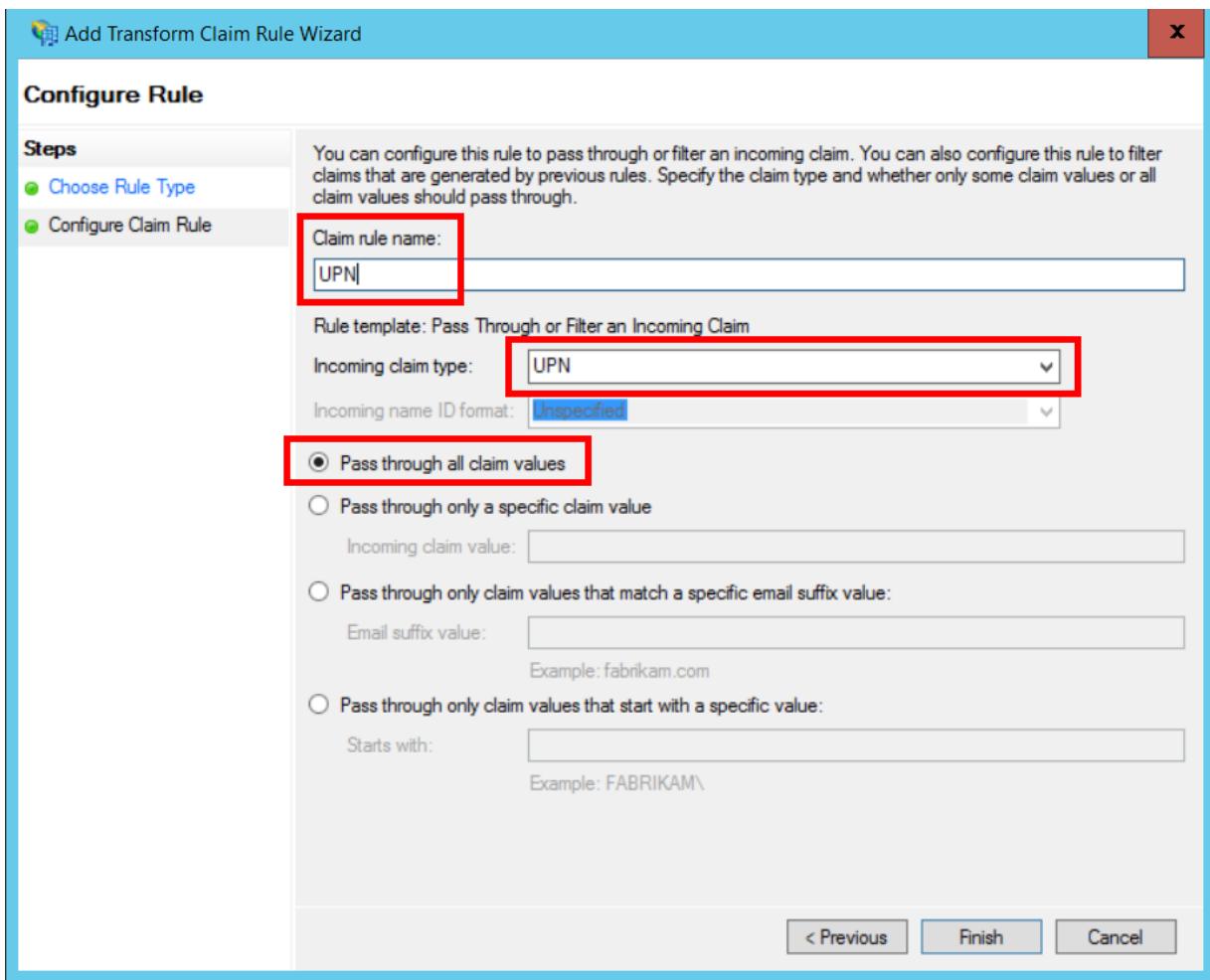
1. In Server Manager, click **Tools**, and then select **AD FS Management**.
2. In the console tree, under **AD FS**, right click **Claims Provider Trusts**. Select **Add Claims Provider Trust**.
3. Click **Start** to start the wizard.
4. Select the option "Import data about the claims provider published 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.

Add Relying Party Trust Wizard

**Select Data Source**

**Steps**

- Welcome
- Select Data Source
- Choose Access Control Policy
- Ready to Add Trust
- Finish

Select an option that this wizard will use to obtain data about this relying party:

Import data about the relying party published online or on a local network  
Use this option to import the necessary data and certificates from a relying party organization that publishes its federation metadata online or on a local network.

Federation metadata address (host name or URL):

Example: fs.contoso.com or https://www.contoso.com/app

Import data about the relying party from a file  
Use this option to import the necessary data and certificates from a relying party organization that has exported its federation metadata to a file. Ensure that this file is from a trusted source. This wizard will not validate the source of the file.

Federation metadata file location:

Enter data about the relying party manually  
Use this option to manually input the necessary data about this relying party organization.

< Previous  Cancel

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.

Add Relying Party Trust Wizard

### Choose Access Control Policy

**Steps**

- Welcome
- Select Data Source
- Specify Display Name
- Choose Access Control Policy
- Ready to Add Trust
- Finish

Choose an access control policy:

| Name                                                         | Description                                                              |
|--------------------------------------------------------------|--------------------------------------------------------------------------|
| Permit everyone                                              | Grant access to everyone.                                                |
| Permit everyone and require MFA                              | Grant access to everyone and require MFA.                                |
| Permit everyone and require MFA for specific group           | Grant access to everyone and require MFA for specific group.             |
| Permit everyone and require MFA from extranet access         | Grant access to the intranet users and require MFA from extranet access. |
| Permit everyone and require MFA from unauthenticated devices | Grant access to everyone and require MFA from unauthenticated devices.   |
| Permit everyone for intranet access                          | Grant access to the intranet users.                                      |
| Permit specific group                                        | Grant access to users of one or more specific groups.                    |

**Policy**

```
Permit users
from <parameter> groups
```

I do not want to configure access control policies at this time. No user will be permitted access for this application.

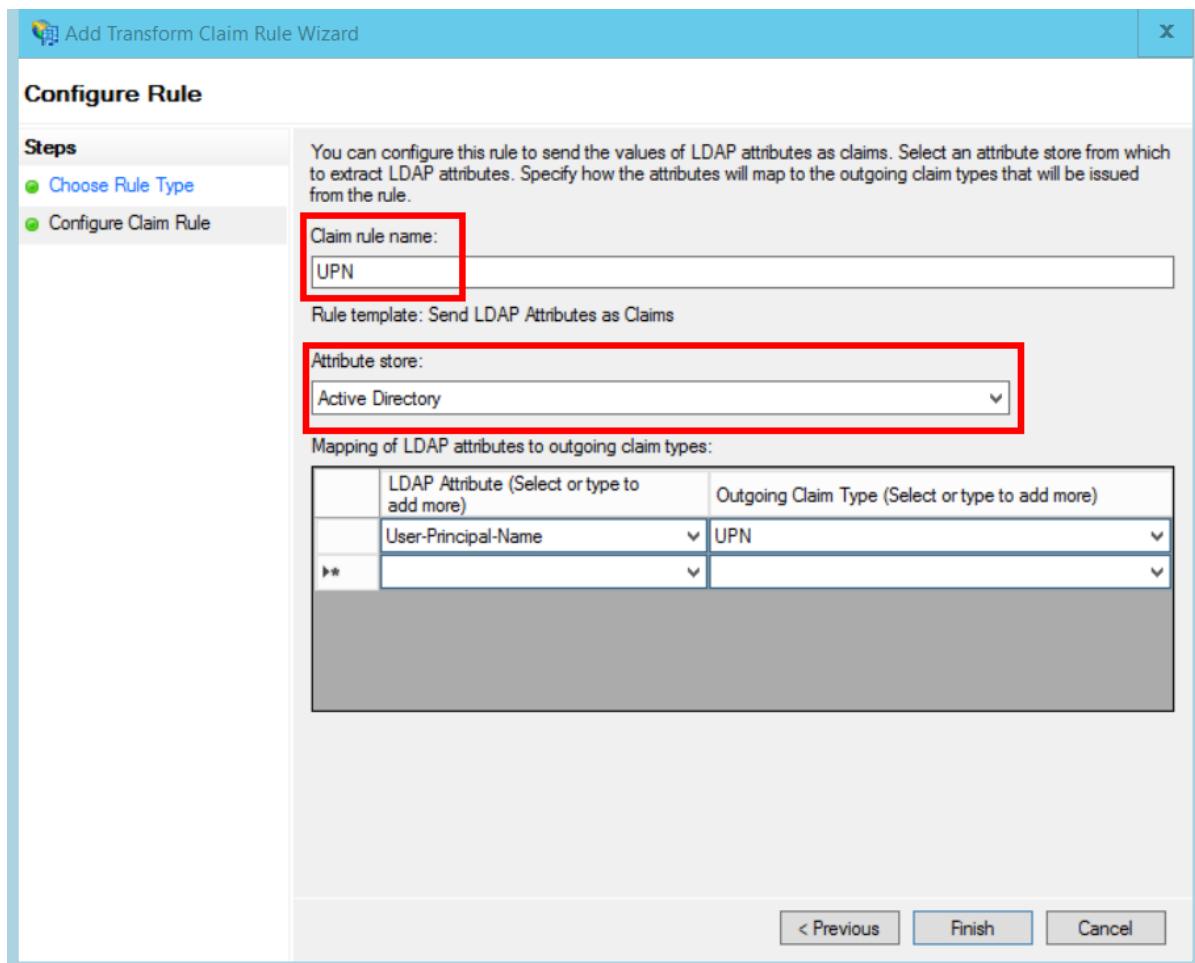
< Previous      Next >      Cancel

7. Enter any parameters required in the **Policy** box.

8. Click **Next** to complete the wizard.

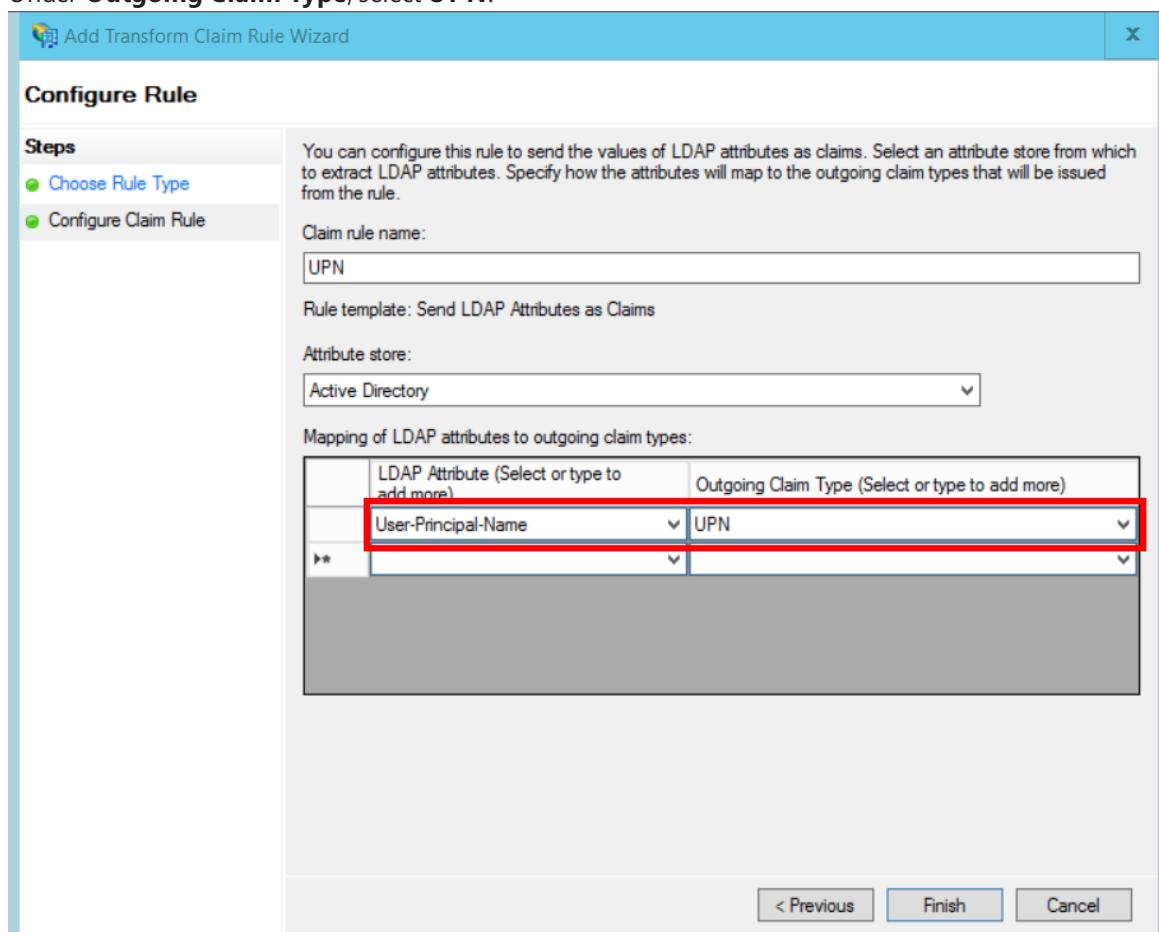
#### Add claims rules

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.

# Run the Surveys application

7/24/2018 • 7 minutes to read • [Edit Online](#)

This article describes how to run the [Tailspin Surveys](#) application locally, from Visual Studio. In these steps, you won't deploy the application to Azure. However, you will need to create some Azure resources — an Azure Active Directory (Azure AD) directory and a Redis cache.

Here is a summary of the steps:

1. Create an Azure AD directory (tenant) for the fictitious Tailspin company.
2. Register the Surveys application and the backend web API with Azure AD.
3. Create an Azure Redis Cache instance.
4. Configure application settings and create a local database.
5. Run the application and sign up a new tenant.
6. Add application roles to users.

## Prerequisites

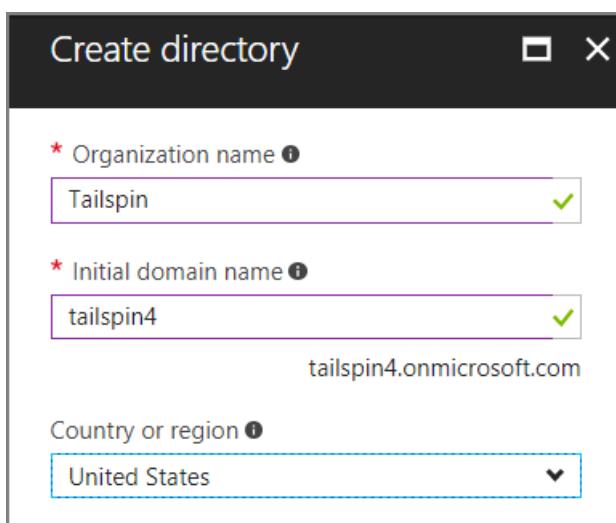
- [Visual Studio 2017](#)
- [Microsoft Azure](#) account

## Create the Tailspin tenant

Tailspin is the fictitious company that hosts the Surveys application. Tailspin uses Azure AD to enable other tenants to register with the app. Those customers can then use their Azure AD credentials to sign into the app.

In this step, you'll create an Azure AD directory for Tailspin.

1. Sign into the [Azure portal](#).
2. Click **+ Create a Resource > Identity > Azure Active Directory**.
3. Enter `Tailspin` for the organization name, and enter a domain name. The domain name will have the form `xxxx.onmicrosoft.com` and must be globally unique.



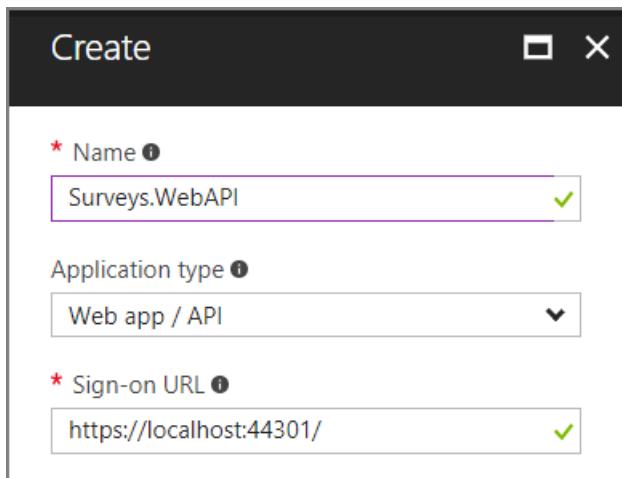
4. Click **Create**. It may take a few minutes to create the new directory.

To complete the end-to-end scenario, you'll need a second Azure AD directory to represent a customer that signs

up for the application. You can use your default Azure AD directory (not Tailspin), or create a new directory for this purpose. In the examples, we use Contoso as the fictitious customer.

## Register the Surveys web API

1. In the [Azure portal](#), switch to the new Tailspin directory by selecting your account in the top right corner of the portal.
2. In the left-hand navigation pane, choose **Azure Active Directory**.
3. Click **App registrations > New application registration**.
4. In the **Create** blade, enter the following information:
  - **Name:** `Surveys.WebAPI`
  - **Application type:** `Web app / API`
  - **Sign-on URL:** `https://localhost:44301/`



5. Click **Create**.
6. In the **App registrations** blade, select the new **Surveys.WebAPI** application.
7. Click **Settings > Properties**.
8. In the **App ID URI** edit box, enter `https://<domain>/surveys.webapi`, where `<domain>` is the domain name of the directory. For example: `https://tailspin.onmicrosoft.com/surveys.webapi`

The screenshot shows the 'Properties' blade for the 'Surveys.WebAPI' application. The 'Name' is set to 'Surveys.WebAPI'. The 'Object ID' is '4bed5cee-374e-44ad-9dba-89016e0688e0'. The 'Application ID' is '42b756d0-21fd-4291-b734-b81b2c35a9a0'. The 'App ID URI' field is highlighted with a red border and contains the value 'https://tailspin.onmicrosoft.com/surveys.w...'. A green checkmark icon is visible next to the URL.

9. Set **Multi-tenanted** to **YES**.

10. Click **Save**.

## Register the Surveys web app

1. Navigate back to the **App registrations** blade, and click **New application registration**.

2. In the **Create** blade, enter the following information:

- Name:** Surveys
- Application type:** Web app / API
- Sign-on URL:** <https://localhost:44300>

Notice that the sign-on URL has a different port number from the `Surveys.WebAPI` app in the previous step.

3. Click **Create**.

4. In the **App registrations** blade, select the new **Surveys** application.

5. Copy the application ID. You will need this later.

The screenshot shows the 'Surveys' application registration details. The 'Display name' is 'Surveys'. The 'Application type' is 'Web app / API'. The 'Home page' is '<https://localhost:44300>'. The 'Application ID' is '68c8a317-...'. The 'Object ID' is '2869e360-...'. The 'Managed application in local directory' is 'Surveys'. A 'Click to copy' button is shown next to the Application ID.

6. Click **Properties**.

7. In the **App ID URI** edit box, enter `https://<domain>/surveys`, where `<domain>` is the domain name of the directory.

The screenshot shows two blades side-by-side. The left blade is titled 'Settings' and contains sections for 'GENERAL' (Properties, Reply URLs, Owners) and 'API ACCESS' (Required permissions). The right blade is titled 'Properties' and shows the following fields:

- Name: Surveys.WebAPI
- Object ID: 4bed5cee-374e-44ad-9dba-89016e0688e0
- Application ID: 42b756d0-21fd-4291-b734-b81b2c35a9a0
- App ID URI: https://tailspin.onmicrosoft.com/surveys.w... (highlighted with a red box)

8. Set **Multi-tenanted** to **YES**.
9. Click **Save**.
10. In the **Settings** blade, click **Reply URLs**.
11. Add the following reply URL: `https://localhost:44300/signin-oidc`.
12. Click **Save**.
13. Under **API ACCESS**, click **Keys**.
14. Enter a description, such as `client secret`.
15. In the **Select Duration** dropdown, select **1 year**.
16. Click **Save**. The key will be generated when you save.
17. Before you navigate away from this blade, copy the value of the key.

**NOTE**

The key won't be visible again after you navigate away from the blade.

18. Under **API ACCESS**, click **Required permissions**.
19. Click **Add > Select an API**.
20. In the search box, search for `Surveys.WebAPI`.

The screenshot shows the 'Add API access' blade. Step 1, 'Select an API', shows a list with 'Surveys.WebAPI' selected. Step 2, 'Select permissions', is shown below.

21. Select `Surveys.WebAPI` and click **Select**.
22. Under **Delegated Permissions**, check **Access Surveys.WebAPI**.

Enable Access

| APPLICATION PERMISSIONS               | REQUIRES ADMIN |
|---------------------------------------|----------------|
| No application permissions available. |                |

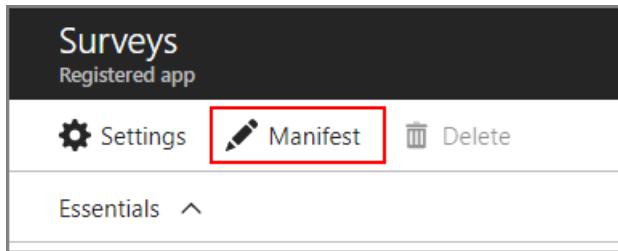
| DELEGATED PERMISSIONS                                     | REQUIRES ADMIN                        |
|-----------------------------------------------------------|---------------------------------------|
| <input checked="" type="checkbox"/> Access Surveys.WebAPI | <span style="color: red;">-</span> No |

23. Click **Select > Done**.

## Update the application manifests

1. Navigate back to the **Settings** blade for the `Surveys.WebAPI` app.

2. Click **Manifest > Edit**.



3. Add the following JSON to the `appRoles` element. Generate new GUIDs for the `id` properties.

```
{
  "allowedMemberTypes": ["User"],
  "description": "Creators can create surveys",
  "displayName": "SurveyCreator",
  "id": "<Generate a new GUID. Example: 1b4f816e-5eaf-48b9-8613-7923830595ad>",
  "isEnabled": true,
  "value": "SurveyCreator"
},
{
  "allowedMemberTypes": ["User"],
  "description": "Administrators can manage the surveys in their tenant",
  "displayName": "SurveyAdmin",
  "id": "<Generate a new GUID>",
  "isEnabled": true,
  "value": "SurveyAdmin"
}
```

4. In the `knownClientApplications` property, add the application ID for the Surveys web application, which you got when you registered the Surveys application earlier. For example:

```
"knownClientApplications": ["be2cea23-aa0e-4e98-8b21-2963d494912e"],
```

This setting adds the Surveys app to the list of clients authorized to call the web API.

5. Click **Save**.

Now repeat the same steps for the Surveys app, except do not add an entry for `knownClientApplications`. Use the same role definitions, but generate new GUIDs for the IDs.

## Create a new Redis Cache instance

The Surveys application uses Redis to cache OAuth 2 access tokens. To create the cache:

1. Go to [Azure Portal](#) and click + **Create a Resource** > **Databases** > **Redis Cache**.
2. Fill in the required information, including DNS name, resource group, location, and pricing tier. You can create a new resource group or use an existing resource group.
3. Click **Create**.
4. After the Redis cache is created, navigate to the resource in the portal.
5. Click **Access keys** and copy the primary key.

For more information about creating a Redis cache, see [How to Use Azure Redis Cache](#).

## Set application secrets

1. Open the Tailspin.Surveys solution in Visual Studio.
2. In Solution Explorer, right-click the Tailspin.Surveys.Web project and select **Manage User Secrets**.
3. In the secrets.json file, paste in the following:

```
{
  "AzureAd": {
    "ClientId": "<Surveys application ID>",
    "ClientSecret": "<Surveys app client secret>",
    "PostLogoutRedirectUri": "https://localhost:44300/",
    "WebApiResourceId": "<Surveys.WebAPI app ID URI>"
  },
  "Redis": {
    "Configuration": "<Redis DNS name>.redis.cache.windows.net,password=<Redis primary key>,ssl=true"
  }
}
```

Replace the items shown in angle brackets, as follows:

- `AzureAd:ClientId` : The application ID of the Surveys app.
  - `AzureAd:ClientSecret` : The key that you generated when you registered the Surveys application in Azure AD.
  - `AzureAd:WebApiResourceId` : The App ID URI that you specified when you created the Surveys.WebAPI application in Azure AD. It should have the form `https://<directory>.onmicrosoft.com/surveys.webapi`
  - `Redis:Configuration` : Build this string from the DNS name of the Redis cache and the primary access key. For example, "tailspin.redis.cache.windows.net,password=2h5tBxx,ssl=true".
4. Save the updated secrets.json file.
  5. Repeat these steps for the Tailspin.Surveys.WebAPI project, but paste the following into secrets.json. Replace the items in angle brackets, as before.

```
{  
    "AzureAd": {  
        "WebApiResourceId": "<Surveys.WebAPI app ID URI>"  
    },  
    "Redis": {  
        "Configuration": "<Redis DNS name>.redis.cache.windows.net,password=<Redis primary key>,ssl=true"  
    }  
}
```

## Initialize the database

In this step, you will use Entity Framework 7 to create a local SQL database, using LocalDB.

1. Open a command window
2. Navigate to the Tailspin.Surveys.Data project.
3. Run the following command:

```
dotnet ef database update --startup-project ..\Tailspin.Surveys.Web
```

## Run the application

To run the application, start both the Tailspin.Surveys.Web and Tailspin.Surveys.WebAPI projects.

You can set Visual Studio to run both projects automatically on F5, as follows:

1. In Solution Explorer, right-click the solution and click **Set Startup Projects**.
2. Select **Multiple startup projects**.
3. Set **Action = Start** for the Tailspin.Surveys.Web and Tailspin.Surveys.WebAPI projects.

## Sign up a new tenant

When the application starts, you are not signed in, so you see the welcome page:

The screenshot shows a web browser window with the title bar "Published Surveys - Tails X". The address bar says "localhost:44300". The page itself has a dark header with the Tailspin logo and a "Sign in" button. The main content area features a large "Welcome to Tailspin!" heading, a blue button labeled "Enroll your company in Tailspin!", and a note in parentheses: "(You need to be an admin of the Azure Active Directroy of your company to enroll.)". Below this is a section titled "Published Surveys" with a message: "No surveys have been published."

To sign up an organization:

1. Click **Enroll your company in Tailspin**.
2. Sign in to the Azure AD directory that represents the organization using the Surveys app. You must sign in as an admin user.
3. Accept the consent prompt.

The application registers the tenant, and then signs you out. The app signs you out because you need to set up the application roles in Azure AD, before using the application.

The screenshot shows a web browser window with the title bar "Sign up Success - Tailsp X". The address bar says "localhost:44300/Account/SignUpSuccess". The page has a dark header with the Tailspin logo and a "Sign in" button. The main content area displays the message "Sign up Success" and "You have successfully signed up. Please follow the documentation to assign application roles."

## Assign application roles

When a tenant signs up, an AD admin for the tenant must assign application roles to users.

1. In the [Azure portal](#), switch to the Azure AD directory that you used to sign up for the Surveys app.

2. In the left-hand navigation pane, choose **Azure Active Directory**.
3. Click **Enterprise applications > All applications**. The portal will list **Survey** and **Survey.WebAPI**. If not, make sure that you completed the sign up process.
4. Click on the Surveys application.
5. Click **Users and Groups**.
6. Click **Add user**.
7. If you have Azure AD Premium, click **Users and groups**. Otherwise, click **Users**. (Assigning a role to a group requires Azure AD Premium.)
8. Select one or more users and click **Select**.

**Add Assignment**  
contoso

Groups are not available for assignment due to your Active Directory plan level.

Users  
None Selected

Select Role  
None Selected

**Users**

+ Invite

Select ⓘ

Search by name or email address

alice  
alice@contoso.com

bob  
bob@contoso.com

9. Select the role and click **Select**.

**Add Assignment**  
contoso

Groups are not available for assignment due to your Active Directory plan level.

Users  
1 user selected.

Select Role  
None Selected

**Select Role**

Role assigned

Enter role name to filter items...

SurveyAdmin

SurveyCreator

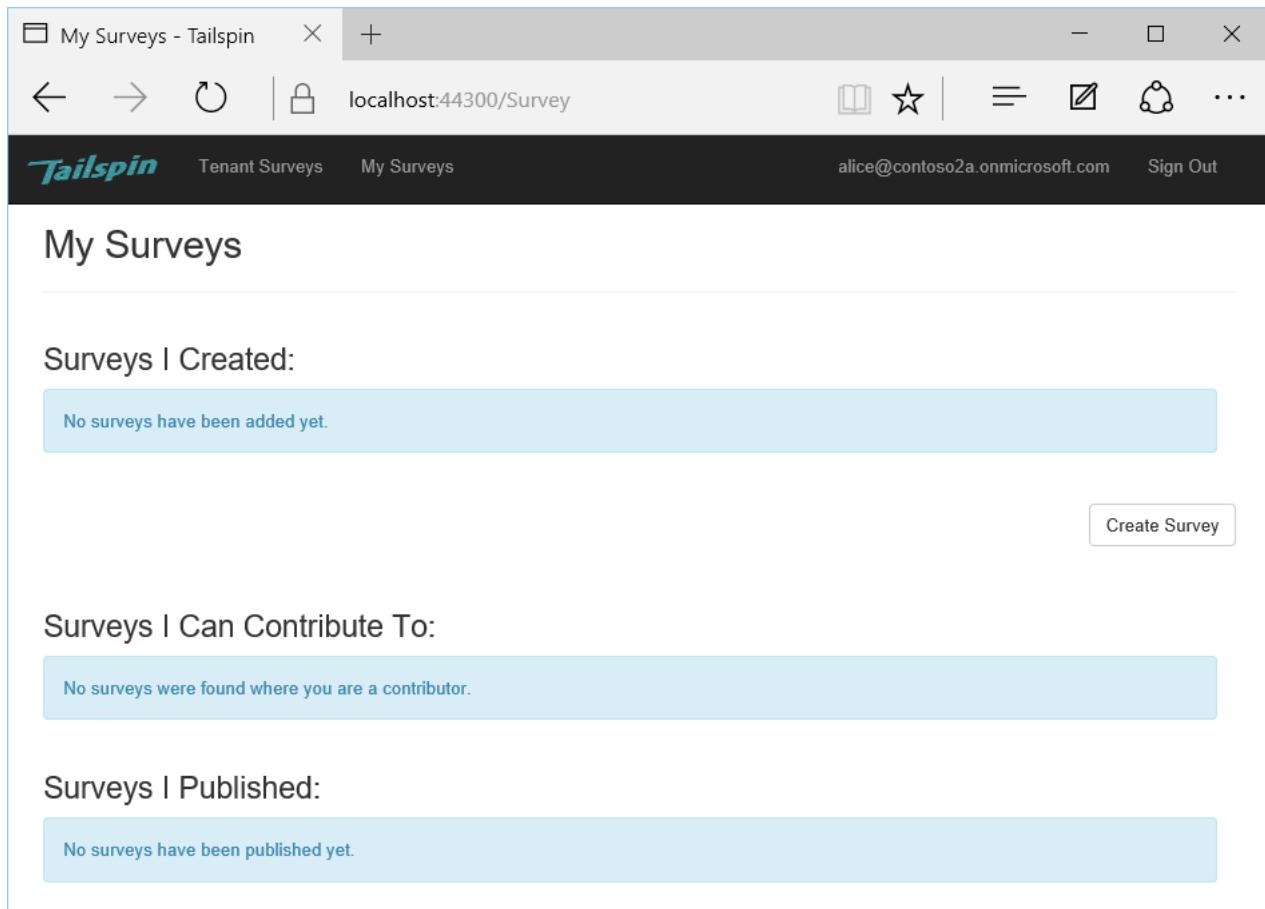
10. Click **Assign**.

Repeat the same steps to assign roles for the Survey.WebAPI application.

**Important:** A user should always have the same roles in both Survey and Survey.WebAPI. Otherwise, the user will have inconsistent permissions, which may lead to 403 (Forbidden) errors from the Web API.

Now go back to the app and sign in again. Click **My Surveys**. If the user is assigned to the SurveyAdmin or SurveyCreator role, you will see a **Create Survey** button, indicating that the user has permissions to create a new

survey.



The screenshot shows a web browser window titled "My Surveys - Tailspin". The address bar displays "localhost:44300/Survey". The top navigation bar includes links for "Tenant Surveys" and "My Surveys", along with user information "alice@contoso2a.onmicrosoft.com" and a "Sign Out" button. The main content area is titled "My Surveys" and contains three sections: "Surveys I Created:", "Surveys I Can Contribute To:", and "Surveys I Published:". Each section features a light blue callout box with the message "No surveys have been added yet." or "No surveys were found where you are a contributor." A "Create Survey" button is located in the bottom right corner of the "Surveys I Created:" section.

My Surveys

Surveys I Created:

No surveys have been added yet.

Create Survey

Surveys I Can Contribute To:

No surveys were found where you are a contributor.

Surveys I Published:

No surveys have been published yet.

# Migrate an Azure Cloud Services application to Azure Service Fabric

4/11/2018 • 13 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. In a later article, we will optimize the application for Service Fabric by adopting a microservices architecture.

Before reading this article, it will be useful to understand the basics of Service Fabric and microservices architectures in general. See the following articles:

- [Overview of Azure Service Fabric](#)
- [Why a microservices approach to building applications?](#)

## About the Surveys application

In 2012, the patterns & practices group created an application called Surveys, for a book called [Developing Multi-tenant Applications for the Cloud](#). The book describes a fictitious company named Tailspin that designs and implements the Surveys application.

Surveys is a multitenant application that allows customers to create surveys. After a customer signs up for the application, members of the customer's organization 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.

This article describes the first phase. A later article will describe the second phase. In a real-world project, it's likely that both stages would overlap. While porting to Service Fabric, you would also start to re-architect 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.

The cloud service is an updated version of the original application from the *Developing Multi-tenant Applications* book.

## Why Microservices?

An in-depth discussion of microservices is beyond scope of this article, but here are some of the benefits that

Tailspin hopes to get by moving to a microservices architecture:

- **Application upgrades.** Services can be deployed independently, so you can take an incremental approach to upgrading an application.
- **Resiliency and fault isolation.** If a service fails, other services continue to run.
- **Scalability.** Services can be scaled independently.
- **Flexibility.** Services are designed around business scenarios, not technology stacks, making it easier to migrate services to new technologies, frameworks, or data stores.
- **Agile development.** Individual services have less code than a monolithic application, making the code base easier to understand, reason about, and test.
- **Small, focused teams.** Because the application is broken down into many small services, each service can be built by a small focused team.

## Why Service Fabric?

Service Fabric is a good fit for a microservices architecture, 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](#).

|                         | 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                      |

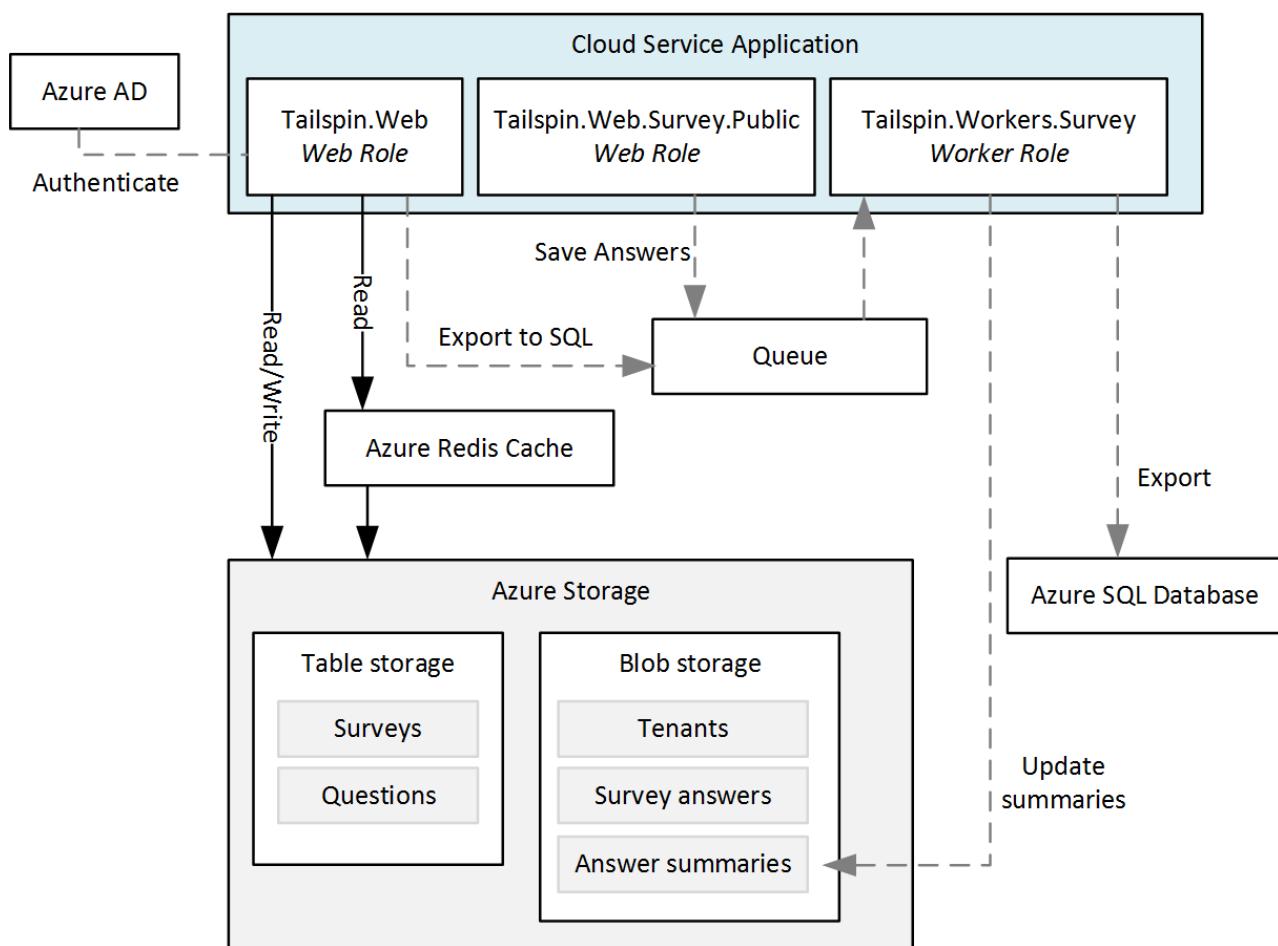
|                    | CLOUD SERVICES                       | SERVICE FABRIC                   |
|--------------------|--------------------------------------|----------------------------------|
| Web hosting        | IIS**                                | Self-hosting                     |
| Deployment 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                   |
| Auto-scaling       | Built-in service                     | VM 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.

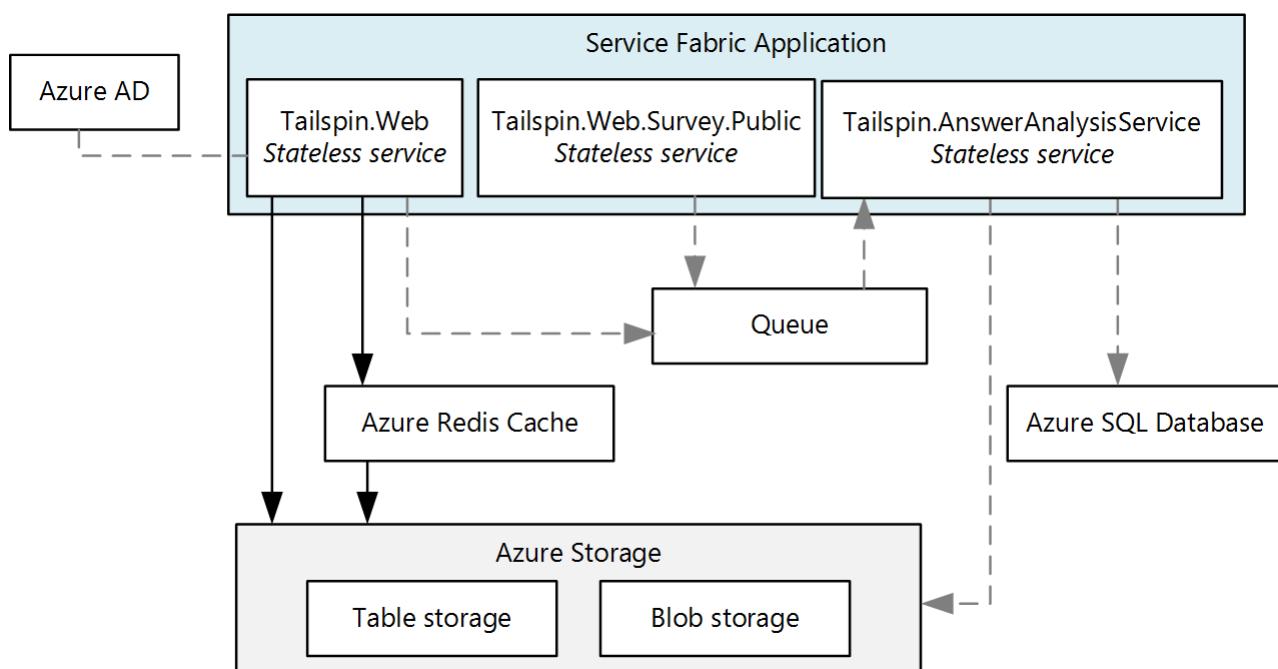
- The **Tailspin.Web.Survey.Public** web role hosts an ASP.NET website where people can take the surveys that Tailspin customers publish.
- The **Tailspin.Workers.Survey** worker role does background processing. The web roles put work items onto a queue, and the worker role processes the items. Two background 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 Redis Cache** 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 very 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

As mentioned, we migrated each cloud service role to a Service Fabric service. Because cloud service roles are stateless, for this phase it made sense to create stateless services in Service Fabric.

For the migration, we 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 requirement to self-host means that a Service Fabric service can't use ASP.NET MVC or ASP.NET Web Forms,

because those frameworks require IIS and do not support self-hosting. Options for self-hosting include:

- [ASP.NET Core](#), self-hosted using the [Kestrel](#) web server.
- [ASP.NET Web API](#), self-hosted using [OWIN](#).
- Third-party frameworks such as [Nancy](#).

The original Surveys application uses ASP.NET MVC. Because ASP.NET MVC cannot be self-hosted in Service Fabric, we 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 us to take advantage of the latest features in ASP.NET Core. To do the conversion, we 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, however, Service Fabric services can be co-located 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](#).

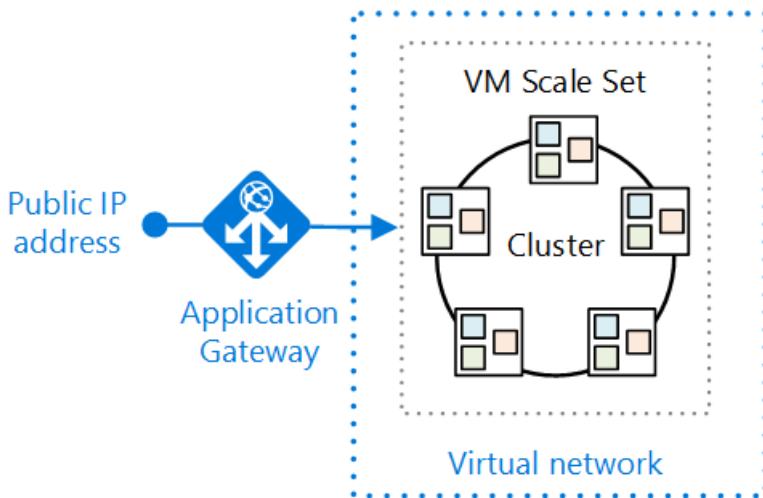
To support different configuration settings for multiple environments, use the following approach, described in [Manage application parameters for multiple environments](#):

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. In this article, we focus on deploying to Azure.

The following diagram shows a recommended deployment:



The Service Fabric cluster is deployed to a [VM scale set](#). Scale sets are an Azure Compute resource that can be used to deploy and manage a set of identical VMs.

As mentioned, the Kestrel web server requires 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                     |
|---|--|
| <code>http://tailspin.cloudapp.net</code> | <code>https://tailspin.cloudapp.net</code> |

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.

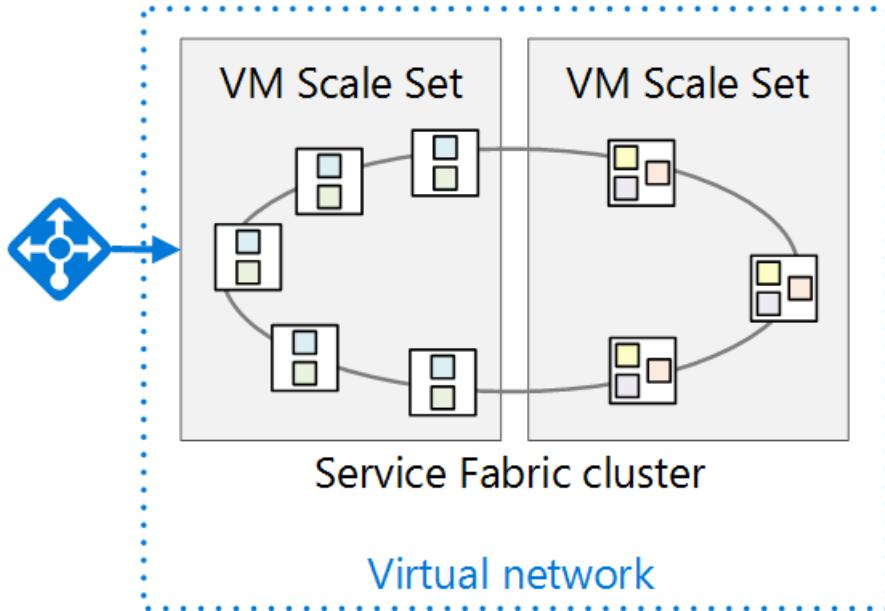
One approach that we *don't* recommend is allowing external clients to send requests through the Service Fabric [reverse proxy](#). Although this is possible, the reverse proxy is intended for service-to-service communication. Opening it to external clients exposes *any* service running in the cluster 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:

- When you create the cluster, define two or more node types.
- 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 VM 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. VM scale sets support auto-scaling using auto-scale rules based on performance counters. For more information, see [Scale a Service Fabric cluster in or out using auto-scale rules](#).

While the cluster is running, you should collect logs from all the nodes in a central location. For more information, see [Collect logs by using Azure Diagnostics](#).

## Conclusion

Porting the Surveys application to Service Fabric was fairly straightforward. To summarize, we did the following:

- Converted the roles to stateless services.
- Converted the web front ends to ASP.NET Core.
- Changed the packaging and configuration files to the Service Fabric model.

In addition, the deployment changed from Cloud Services to a Service Fabric cluster running in a VM Scale Set.

## Next steps

Now that the Surveys application has been successfully ported, Tailspin wants to take advantage of Service Fabric features such as independent service deployment and versioning. Learn how Tailspin decomposed these services to a more granular architecture to take advantage of these Service Fabric features in [Refactor an Azure Service Fabric Application migrated from Azure Cloud Services](#)

# Refactor an Azure Service Fabric Application migrated from Azure Cloud Services

10/5/2018 • 9 minutes to read • [Edit Online](#)

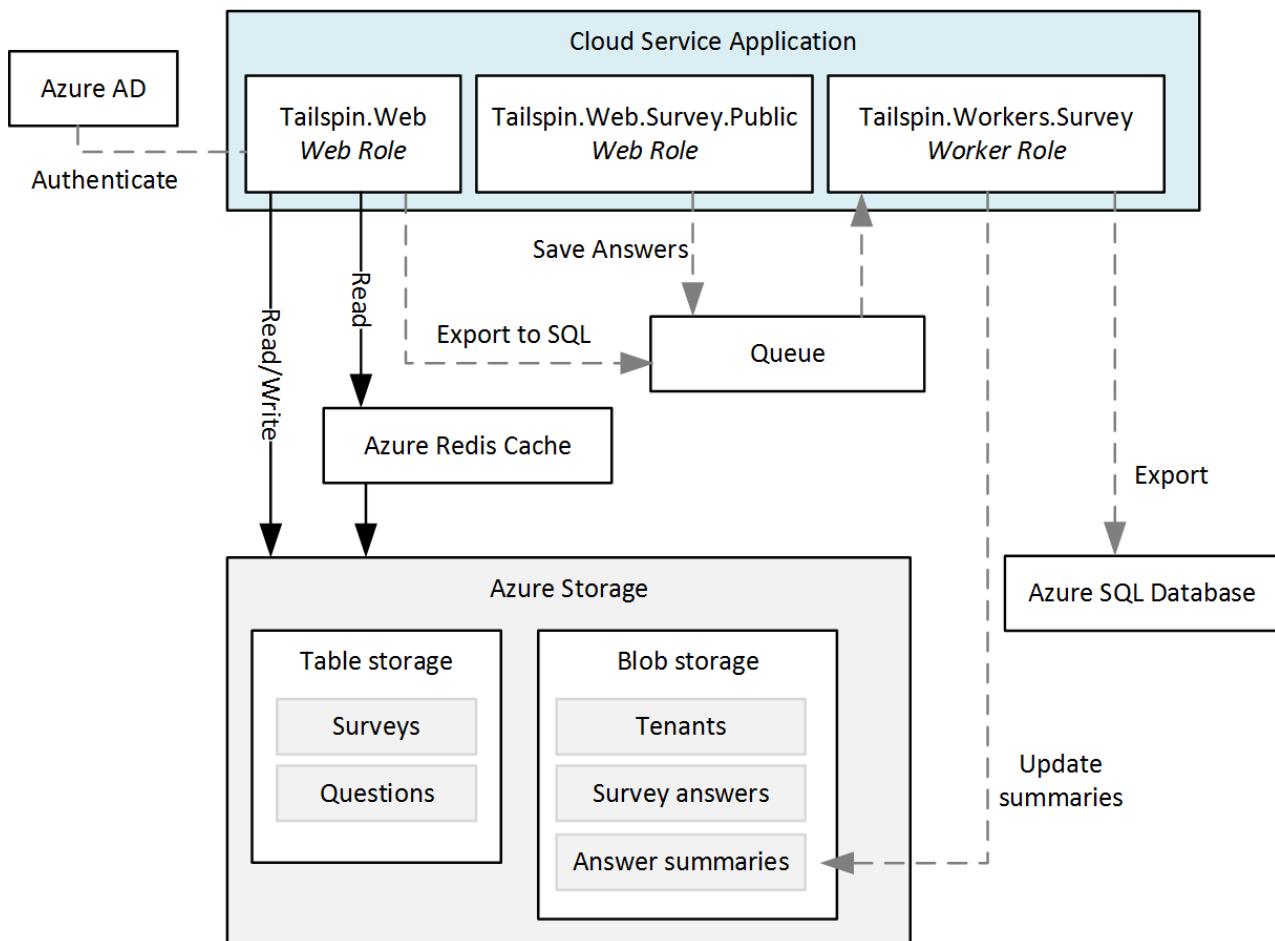


[Sample code](#)

This article describes refactoring an existing Azure Service Fabric application to a more granular architecture. This article focuses on the design, packaging, performance, and deployment considerations of the refactored Service Fabric application.

## Scenario

As discussed in the previous article, [Migrating an Azure Cloud Services application to Azure Service Fabric](#), the patterns & practices team authored a book in 2012 that documented the process for designing and implementing a Cloud Services application in Azure. The book describes a fictitious company named Tailspin that wants to create a Cloud Services application named **Surveys**. The Surveys application allows users to create and publish surveys that can be answered by the public. The following diagram shows the architecture of this version of the Surveys application:



The **Tailspin.Web** web role hosts an ASP.NET MVC site that Tailspin customers use to:

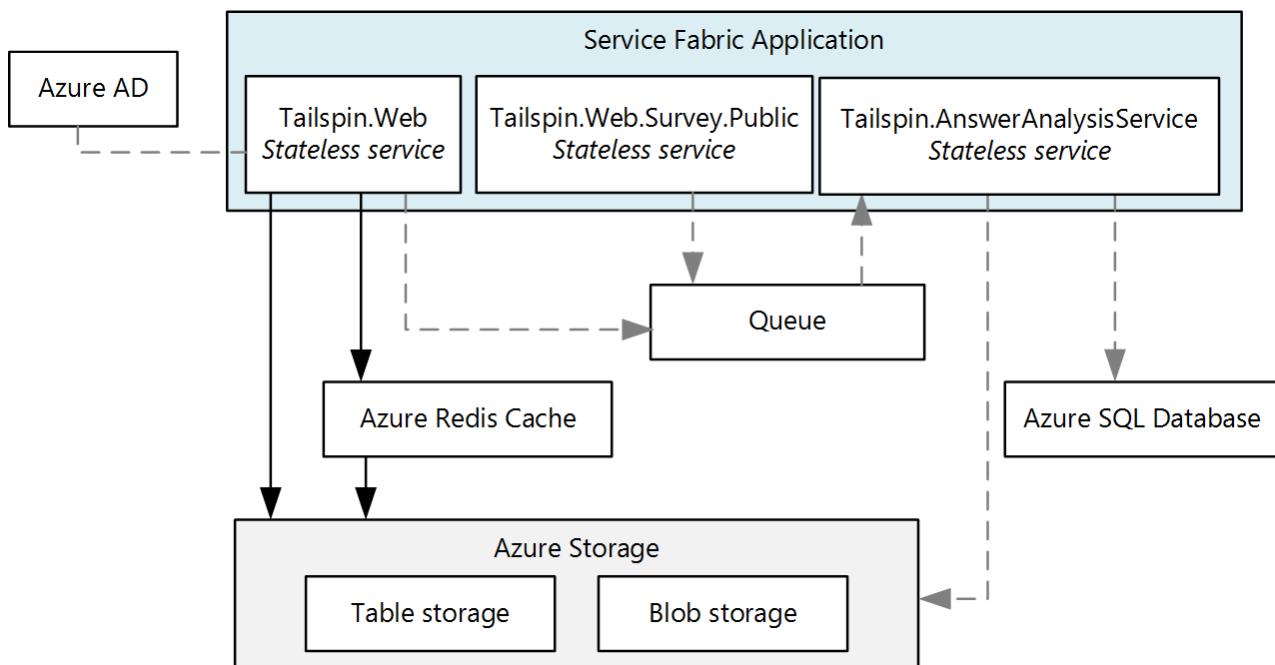
- sign up for the Surveys application,
- create or delete a single survey,
- view results for a single survey,

- request that survey results be exported to SQL, and
- view aggregated survey results and analysis.

The **Tailspin.Web.Survey.Public** web role also hosts an ASP.NET MVC site that the public visits to fill out the surveys. These responses are put in a queue to be saved.

The **Tailspin.Workers.Survey** worker role performs background processing by picking up requests from multiple queues.

The patterns & practices team then created a new project to port this application to Azure Service Fabric. The goal of this project was to make only the necessary code changes to get the application running in an Azure Service Fabric cluster. As a result, the original web and worker roles were not decomposed into a more granular architecture. The resulting architecture is very similar to the Cloud Service version of the application:



The **Tailspin.Web** service is ported from the original *Tailspin.Web* web role.

The **Tailspin.Web.Survey.Public** service is ported from the original *Tailspin.Web.Survey.Public* web role.

The **Tailspin.AnswerAnalysisService** service is ported from the original *Tailspin.Workers.Survey* worker role.

#### NOTE

While minimal code changes were made to each of the web and worker roles, **Tailspin.Web** and **Tailspin.Web.Survey.Public** were modified to self-host a [Kestrel](#) web server. The earlier Surveys application is an ASP.NET application that was hosted using Internet Information Services (IIS), but it is not possible to run IIS as a service in Service Fabric. Therefore, any web server must be capable of being self-hosted, such as [Kestrel](#). It is possible to run IIS in a container in Service Fabric in some situations. See [scenarios for using containers](#) for more information.

Now, Tailspin is refactoring the Surveys application 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.

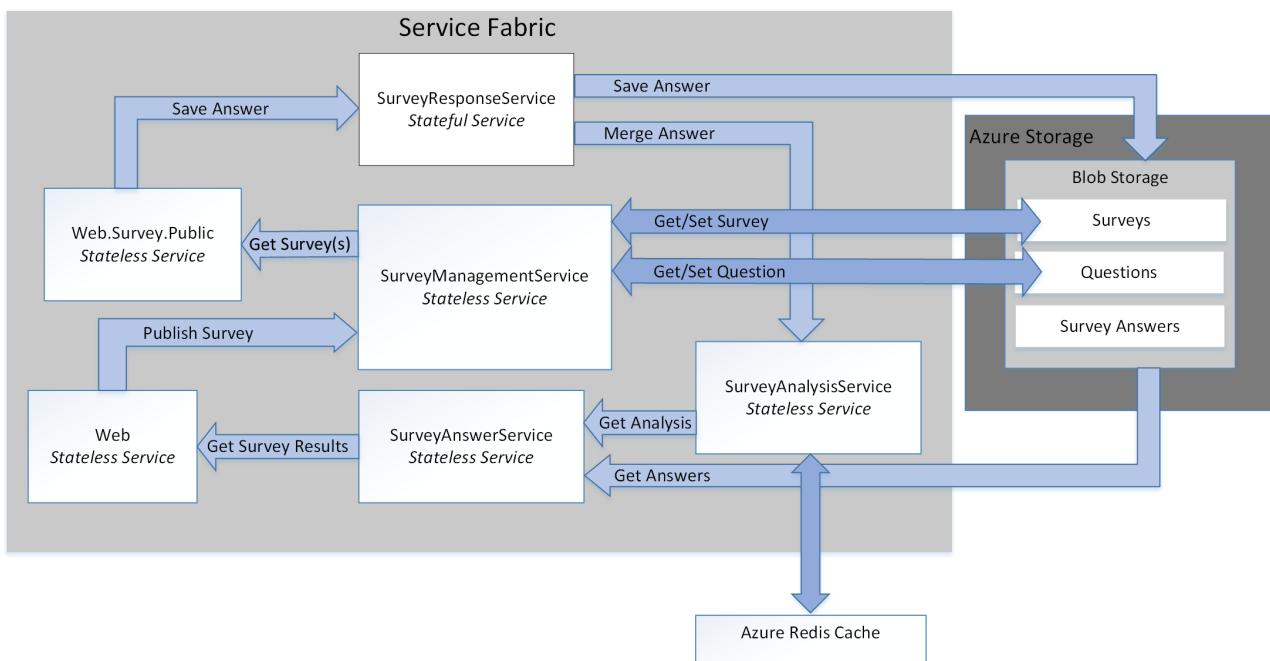
- Each service can be implemented using the best technology for that service. For example, a service fabric cluster can include services built using different versions of the .Net Frameworks, Java, or other languages such as C or C++.
- Each service can be independently scaled to respond to increases and decreases in load.

#### NOTE

Multitenancy is out of scope for the refactoring of this application. Tailspin has several options to support multitenancy and can make these design decisions later without affecting the initial design. For example, Tailspin can create separate instances of the services for each tenant within a cluster or create a separate cluster for each tenant.

## 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.

Note that 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 make use of 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 VM 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. Note that because the VMs are deployed in a VM scale set, they can be both scaled up and out as user load increases.

#### NOTE

As discussed earlier, in the migrated version of the Surveys application the two web front ends were self-hosted using ASP.NET Core and Kestrel as a web server. While the migrated version of the Survey application does not use a reverse proxy, it is strongly recommended to use a reverse proxy such as IIS, Nginx, or Apache. For more information see [introduction to Kestrel web server implementation in ASP.NET core](#). In the refactored Surveys application, the two web front ends are self-hosted using ASP.NET Core with [WebListener](#) as a web server so a reverse proxy is not necessary.

## 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.

# Designing resilient applications for Azure

10/5/2018 • 25 minutes to read • [Edit Online](#)

In a distributed system, failures will 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.

Building a reliable application in the cloud is different than building a reliable application in an enterprise setting. While historically you may have purchased higher-end hardware to scale up, in a cloud environment you must scale out instead of scaling up. Costs for cloud environments are kept low through the use of commodity hardware. Instead of focusing on preventing failures and optimizing "mean time between failures," in this new environment the focus shifts to "mean time to restore." The goal is to minimize the effect of a failure.

This article provides an overview of how to build resilient applications in Microsoft Azure. It starts with a definition of the term *resiliency* and related concepts. Then it describes a process for achieving resiliency, using a structured approach over the lifetime of an application, from design and implementation to deployment and operations.

## What is resiliency?

**Resiliency** is the ability of a system to recover from failures and continue to function. It's not about *avoiding* failures, but *responding* to failures in a way that avoids downtime or data loss. The goal of resiliency is to return the application to a fully functioning state following a failure.

Two important aspects of resiliency are high availability and disaster recovery.

- **High availability** (HA) is the ability of the application to continue running in a healthy state, without significant downtime. By "healthy state," we mean the application is responsive, and users can connect to the application and interact with it.
- **Disaster recovery** (DR) is the ability to recover from rare but major incidents: non-transient, wide-scale failures, such as service disruption that affects an entire region. Disaster recovery includes data backup and archiving, and may include manual intervention, such as restoring a database from backup.

One way to think about HA versus DR is that DR starts when the impact of a fault exceeds the ability of the HA design to handle it.

When you design resiliency, you must understand your availability requirements. How much downtime is acceptable? This 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? You also have to define what it means for the application to be available. For example, is the application "down" if a customer can submit an order but the system cannot process it within the normal timeframe? Also consider the probability of a particular type of outage occurring, and whether a mitigation strategy is cost-effective.

Another common term is **business continuity** (BC), which is the ability to perform essential business functions during and after adverse conditions, such as a natural disaster or a downed service. BC covers the entire operation of the business, including physical facilities, people, communications, transportation, and IT. This article focuses on cloud applications, but resilience planning must be done in the context of overall BC requirements.

**Data backup** is a critical part of DR. If the stateless components of an application fail, you can always redeploy them. But if data is lost, the system can't return to a stable state. Data must be backed up, ideally in a different region in case of a region-wide disaster.

Backup is distinct from **data replication**. Data replication involves copying data in near-real-time, so that the system can fail over quickly to a replica. Many databases support replication; for example, SQL Server supports SQL Server Always On Availability Groups. Data replication can reduce how long it takes to recover from an outage, by ensuring that a replica of the data is always standing by. However, data replication won't protect against human error. If data gets corrupted because of human error, the corrupted data just gets copied to the replicas. Therefore, you still need to include long-term backup in your DR strategy.

## Process to achieve resiliency

Resiliency is not an add-on. It must be designed into the system and put into operational practice. Here is a general model to follow:

1. **Define** your availability requirements, based on business needs.
2. **Design** the application for resiliency. Start with an architecture that follows proven practices, and then identify the possible failure points in that architecture.
3. **Implement** strategies to detect and recover from failures.
4. **Test** the implementation by simulating faults and triggering forced failovers.
5. **Deploy** the application into production using a reliable, repeatable process.
6. **Monitor** the application to detect failures. By monitoring the system, you can gauge the health of the application and respond to incidents if necessary.
7. **Respond** if there are failures that require manual interventions.

In the remainder of this article, we discuss each of these steps in more detail.

## Define your availability requirements

Resiliency planning starts with business requirements. Here are some approaches for thinking about resiliency in those terms.

### Decompose by workload

Many cloud solutions consist of multiple application workloads. The term "workload" in this context means a discrete capability or computing task, which can be logically separated from other tasks, in terms of business logic and data storage requirements. For example, an e-commerce app might include the following workloads:

- Browse and search a product catalog.
- Create and track orders.
- View recommendations.

These workloads might have different requirements for availability, scalability, data consistency, disaster recovery, and so forth. Again, these are business decisions.

Also consider usage patterns. Are there certain critical periods when the system must be available? For example, a tax-filing service can't go down right before the filing deadline, a video streaming service must stay up during a big sports event, and so on. During the critical periods, you might have redundant deployments across several regions, so the application could fail over if one region failed. However, a multi-region deployment is more expensive, so during less critical times, you might run the application in a single region.

### RTO and RPO

Two important metrics to consider are the recovery time objective and recovery point objective.

- **Recovery time objective** (RTO) is the maximum acceptable time that an application can be unavailable after an incident. If your RTO is 90 minutes, you must be able to restore the application to a running state within 90 minutes from the start of a disaster. If you have a very low RTO, you might keep a second deployment continually running on standby, to protect against a regional outage.

- **Recovery point objective** (RPO) is the maximum duration of data loss that is acceptable during a disaster. For example, if you store data in a single database, with no replication to other databases, and perform hourly backups, you could lose up to an hour of data.

RTO and RPO are business requirements. Conducting a risk assessment can help you define the application's RTO and RPO. Another common metric is **mean time to recover** (MTTR), which is the average time that it takes to restore the application after a failure. MTTR is an empirical fact about a system. If MTTR exceeds the RTO, then a failure in the system will cause an unacceptable business disruption, because it won't be possible to restore the system within the defined RTO.

## SLAs

In Azure, the [Service Level Agreement](#) (SLA) describes Microsoft's commitments for uptime and connectivity. If the SLA for a particular service is 99.9%, it means you should expect the service to be available 99.9% of the time.

### NOTE

The Azure SLA also includes provisions for obtaining a service credit if the SLA is not met, along with specific definitions of "availability" for each service. That aspect of the SLA acts as an enforcement policy.

You should define your own target SLAs for each workload in your solution. An SLA makes it possible to evaluate whether the architecture meets the business requirements. For example, if a workload requires 99.99% uptime, but depends on a service with a 99.9% SLA, that service cannot be a single-point of failure in the system. One remedy is to have a fallback path in case the service fails, or take other measures to recover from a failure in that service.

The following table shows the potential cumulative downtime for various SLA levels.

| SLA     | DOWNTIME PER WEEK | DOWNTIME PER MONTH | DOWNTIME PER YEAR |
|---------|-------------------|--------------------|-------------------|
| 99%     | 1.68 hours        | 7.2 hours          | 3.65 days         |
| 99.9%   | 10.1 minutes      | 43.2 minutes       | 8.76 hours        |
| 99.95%  | 5 minutes         | 21.6 minutes       | 4.38 hours        |
| 99.99%  | 1.01 minutes      | 4.32 minutes       | 52.56 minutes     |
| 99.999% | 6 seconds         | 25.9 seconds       | 5.26 minutes      |

Of course, higher availability is better, everything else being equal. But as you strive for more 9s, the cost and complexity to achieve that level of availability grows. An uptime of 99.99% translates to about 5 minutes of total downtime per month. Is it worth the additional complexity and cost to reach five 9s? The answer depends on the business requirements.

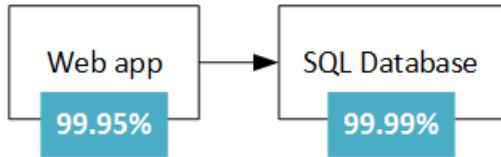
Here are some other considerations when defining an SLA:

- To achieve four 9's (99.99%), you probably can't rely on manual intervention to recover from failures. The application must be self-diagnosing and self-healing.
- Beyond four 9's, it is challenging to detect outages quickly enough to meet the SLA.
- Think about the time window that your SLA is measured against. The smaller the window, the tighter the tolerances. It probably doesn't make sense to define your SLA in terms of hourly or daily uptime.

## Composite SLAs

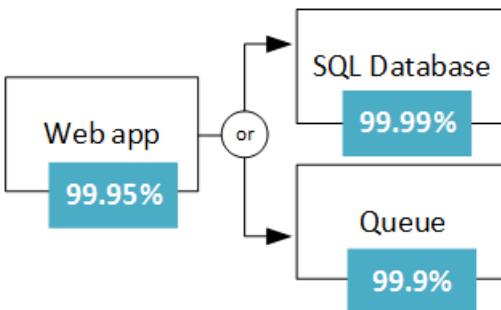
Consider an App Service web app that writes to Azure SQL Database. At the time of this writing, these Azure services have the following SLAs:

- App Service Web Apps = 99.95%
- SQL Database = 99.99%



What is the maximum downtime you would expect for this application? If either service fails, the whole application fails. In general, the probability of each service failing is independent, so the composite SLA for this application is  $99.95\% \times 99.99\% = 99.94\%$ . That's lower than the individual SLAs, which isn't surprising, because an application that relies on multiple services has more potential failure points.

On the other hand, you can improve the composite SLA by creating independent fallback paths. For example, if SQL Database is unavailable, put transactions into a queue, to be processed later.



With this design, the application is still available even if it can't connect to the database. However, it fails if the database and the queue both fail at the same time. The expected percentage of time for a simultaneous failure is  $0.0001 \times 0.001$ , so the composite SLA for this combined path is:

- Database OR queue =  $1.0 - (0.0001 \times 0.001) = 99.99999\%$

The total composite SLA is:

- Web app AND (database OR queue) =  $99.95\% \times 99.99999\% = \sim 99.95\%$

But there are tradeoffs to this approach. The application logic is more complex, you are paying for the queue, and there may be data consistency issues to consider.

**SLA for multi-region deployments.** Another HA technique is to deploy the application in more than one region, and use Azure Traffic Manager to fail over if the application fails in one region. For a two-region deployment, the composite SLA is calculated as follows.

Let  $N$  be the composite SLA for the application deployed in one region. The expected chance that the application will fail in both regions at the same time is  $(1 - N) \times (1 - N)$ . Therefore,

- Combined SLA for both regions =  $1 - (1 - N)(1 - N) = N + (1 - N)N$

Finally, you must factor in the [SLA for Traffic Manager](#). At the time of this writing, the SLA for Traffic Manager SLA is 99.99%.

- Composite SLA =  $99.99\% \times (\text{combined SLA for both regions})$

Also, failing over is not instantaneous and can result in some downtime during a failover. See [Traffic Manager endpoint monitoring and failover](#).

The calculated SLA number is a useful baseline, but it doesn't tell the whole story about availability. Often, an

application can degrade gracefully when a non-critical path fails. 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. In that case, failing to get the image does not reduce the application's uptime, although it affects the user experience.

## Design for resiliency

During the design phase, you should perform a failure mode analysis (FMA). The goal of an FMA is to identify possible points of failure, and define how the application will respond to those failures.

- How will the application detect this type of failure?
- How will the application respond to this type of failure?
- How will you log and monitor this type of failure?

For more information about the FMA process, with specific recommendations for Azure, see [Azure resiliency guidance: Failure mode analysis](#).

### Example of identifying failure modes and detection strategy

**Failure point:** Call to an external web service / API.

| FAILURE MODE           | DETECTION STRATEGY           |
|------------------------|------------------------------|
| Service is unavailable | HTTP 5xx                     |
| Throttling             | HTTP 429 (Too Many Requests) |
| Authentication         | HTTP 401 (Unauthorized)      |
| Slow response          | Request times out            |

### Redundancy and designing for failure

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 data center, such as loss of power in a data center. 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, there is a tradeoff between greater redundancy and reliability versus higher cost and complexity.

Azure has a number of features to make an application redundant at every level of failure, from an individual VM to an entire region.

**Single VM.** Azure provides an uptime SLA for single VMs. Although you can get a higher SLA by running two or more VMs, a single VM may be reliable enough for some workloads. For production workloads, we recommend using two or more VMs for redundancy.

**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.

**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.

When you design a multi-region application, take into account that network latency across regions is higher than within a region. For example, if you are replicating a database to enable failover, use synchronous data replication within a region, but asynchronous data replication across regions.

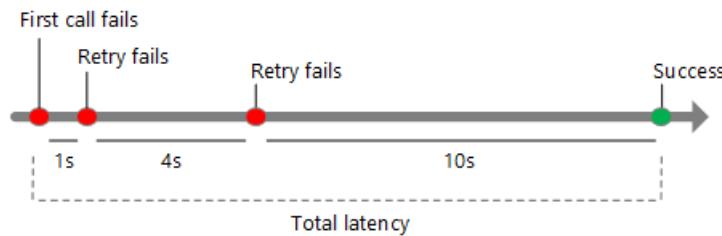
|                  | AVAILABILITY SET | AVAILABILITY ZONE        | PAIRED REGION             |
|------------------|------------------|--------------------------|---------------------------|
| Scope of failure | Rack             | Datacenter               | Region                    |
| Request routing  | Load Balancer    | Cross-zone Load Balancer | Traffic Manager           |
| Network latency  | Very low         | Low                      | Mid to high               |
| Virtual network  | VNet             | VNet                     | Cross-region VNet peering |

## Implement resiliency strategies

This section provides a survey of some common resiliency strategies. Most of these are not limited to a particular technology. The descriptions in this section summarize the general idea behind each technique, with links to further reading.

**Retry transient failures.** Transient failures can be caused by momentary loss of network connectivity, a dropped database connection, or a timeout when a service is busy. Often, a transient failure can be resolved simply by retrying the request. For many Azure services, the client SDK implements automatic retries, in a way that is transparent to the caller; see [Retry service specific guidance](#).

Each retry attempt adds to the total latency. Also, 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. To avoid this, increase the delay between each retry attempt, and limit the total number of failed requests.



**Load balance across instances.** For scalability, a cloud application should be able to scale out by adding more instances. This approach also improves resiliency, because unhealthy instances can be removed from rotation. For example:

- Put two or more VMs behind a load balancer. The load balancer distributes traffic to all the VMs. See [Run load-balanced VMs for scalability and availability](#).

- Scale out an Azure App Service app to multiple instances. App Service automatically balances load across instances. See [Basic web application](#).
- Use [Azure Traffic Manager](#) to distribute traffic across a set of endpoints.

**Replicate data.** Replicating data is a general strategy for handling non-transient failures in a data store. Many storage technologies provide built-in replication, including Azure SQL Database, Cosmos DB, and Apache Cassandra. It's important to consider both the read and write paths. Depending on the storage technology, you might have multiple writable replicas, or a single writable replica and multiple read-only replicas.

To maximize availability, replicas can be placed in multiple regions. However, this increases the latency when replicating the data. Typically, replicating across regions is done asynchronously, which implies an eventual consistency model and potential data loss if a replica fails.

**Degrade gracefully.** If a service fails and there is no failover path, the application may be able to degrade gracefully while still providing an acceptable user experience. For example:

- Put a work item on a queue, to be handled later.
- Return an estimated value.
- Use locally cached data.
- Show the user an error message. (This option is better than having the application stop responding to requests.)

**Throttle high-volume users.** Sometimes a small number of users create excessive load. That can have an impact on other users, reducing the overall availability of your application.

When a single client makes an excessive number of requests, the application might throttle the client for a certain period of time. During the throttling period, the application refuses some or all of the requests from that client (depending on the exact throttling strategy). The threshold for throttling might depend on the customer's service tier.

Throttling does not imply the client was necessarily acting maliciously, only that it exceeded its service quota. In some cases, a consumer might consistently exceed their quota or otherwise behave badly. In that case, you might go further and block the user. Typically, this is done by blocking an API key or an IP address range. For more information, see [Throttling Pattern](#).

**Use a circuit breaker.** The [Circuit Breaker](#) pattern can prevent an application from repeatedly trying an operation that is likely to fail. The circuit breaker wraps calls to a service and tracks the number of recent failures. If the failure count exceeds a threshold, the circuit breaker starts returning an error code without calling the service. This gives the service time to recover.

**Use load leveling to smooth out spikes in traffic.** Applications may experience sudden spikes in traffic, which can overwhelm services on the backend. If a backend service cannot respond to requests quickly enough, it may cause requests to queue (back up), or cause the service to throttle the application. To avoid this, you can use a queue as a buffer. When there is a new work item, instead of calling the backend service immediately, the application queues a work item to run asynchronously. The queue acts as a buffer that smooths out peaks in the load. For more information, see [Queue-Based Load Leveling Pattern](#).

**Isolate critical resources.** Failures in one subsystem can sometimes cascade, causing failures in other parts of the application. 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.

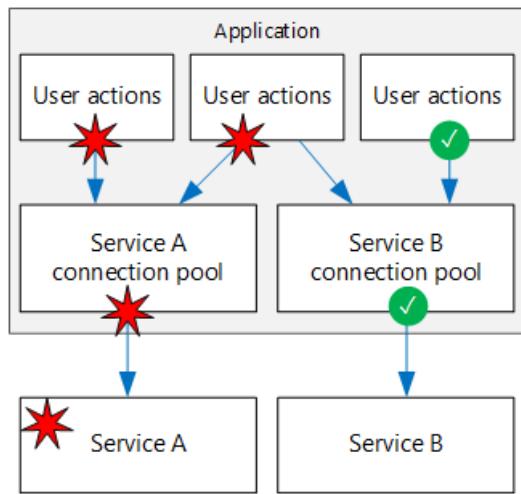
To avoid this, you can partition a system into isolated groups, so that a failure in one partition does not bring down the entire system. This technique is sometimes called the Bulkhead pattern.

Examples:

- Partition a database (for example, by tenant) and assign a separate pool of web server instances for each

partition.

- Use separate thread pools to isolate calls to different services. This helps to prevent cascading failures if one of the services fails. For an example, see the Netflix [Hystrix library](#).
- Use [containers](#) to limit the resources available to a particular subsystem.



**Apply compensating transactions.** A [compensating transaction](#) is a transaction that undoes the effects of another completed transaction. In a distributed system, it can be very difficult to achieve strong transactional consistency. Compensating transactions are a way to achieve consistency by using a series of smaller, individual transactions that can be undone at each step.

For example, to book a trip, a customer might reserve a car, a hotel room, and a flight. If any of these steps fails, the entire operation fails. Instead of trying to use a single distributed transaction for the entire operation, you can define a compensating transaction for each step. For example, to undo a car reservation, you cancel the reservation. In order to complete the whole operation, a coordinator executes each step. If any step fails, the coordinator applies compensating transactions to undo any steps that were completed.

## Test for resiliency

Generally, you can't test resiliency in the same way that you test application functionality (by running unit tests and so on). Instead, you must test how the end-to-end workload performs under failure conditions which only occur intermittently.

Testing is an iterative process. Test the application, measure the outcome, analyze and address any failures that result, and repeat the process.

**Fault injection testing.** Test the resiliency of the system during failures, either by triggering actual failures or by simulating them. Here are some common failure scenarios to test:

- Shut down 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.

Measure the recovery times and verify that your business requirements are met. Test combinations of failure modes as well. Make sure that failures don't cascade, and are handled in an isolated way.

This is another reason why it's important to analyze possible failure points during the design phase. The results

of that analysis should be inputs into your test plan.

**Load testing.** Load testing is crucial for identifying failures that only happen under load, such as the backend database being overwhelmed or service throttling. Test for peak load, using production data or synthetic data that is as close to production data as possible. The goal is to see how the application behaves under real-world conditions.

## Deploy using reliable processes

Once an application is deployed to production, updates are a possible source of errors. In the worst case, a bad update can cause downtime. To avoid this, the deployment process must be predictable and repeatable.

Deployment includes provisioning Azure resources, deploying application code, and applying configuration settings. An update may involve all three, or a subset.

The crucial point is that manual deployments are prone to error. Therefore, it's recommended to have an automated, idempotent process that you can run on demand, and re-run if something fails.

- Use Azure Resource Manager templates to automate provisioning of Azure resources.
- Use [Azure Automation Desired State Configuration \(DSC\)](#) to configure VMs.
- Use an automated deployment process for application code.

Two concepts related to resilient deployment are *infrastructure as code* and *immutable infrastructure*.

- **Infrastructure as code** is the practice of using code to provision and configure infrastructure. Infrastructure as code may use a declarative approach or an imperative approach (or a combination of both). Resource Manager templates are an example of a declarative approach. PowerShell scripts are an example of an imperative approach.
- **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, so it's hard to know exactly what changed, and hard to reason about the system.

Another question is how to roll out an application update. We recommend techniques such as blue-green deployment or canary releases, which push updates in highly controlled way to minimize possible impacts from a bad deployment.

- [Blue-green deployment](#) is a technique where an update is deployed into a production environment separate from the live application. After you validate the deployment, switch the traffic routing to the updated version. For example, Azure App Service Web Apps enables this with staging slots.
- [Canary releases](#) are similar to blue-green deployments. Instead of switching all traffic to the updated version, you roll out the update to a small percentage of users, by routing a portion of the traffic to the new deployment. If there is a problem, back off and revert to the old deployment. Otherwise, route more of the traffic to the new version, until it gets 100% of the traffic.

Whatever approach you take, make sure that you can roll back to the last-known-good deployment, in case the new version is not functioning. Also, if errors occur, the application logs must indicate which version caused the error.

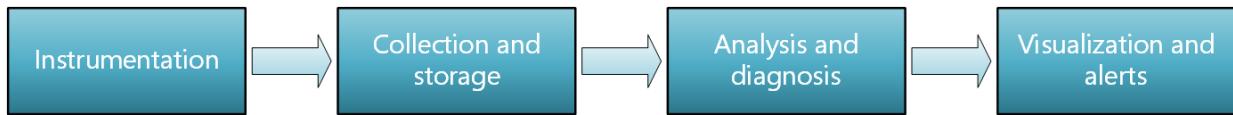
## Monitor to detect failures

Monitoring and diagnostics are crucial for resiliency. If something fails, you need to know that it failed, and you need insights into the cause of the failure.

Monitoring a large-scale distributed system poses a significant challenge. Think about an application that runs on a few dozen VMs — it's not practical to log into each VM, one at a time, and look through log files, trying to troubleshoot a problem. Moreover, the number of VM instances is probably not static. VMs get added and removed as the application scales in and out, and occasionally an instance may fail and need to be

reprovisioned. In addition, a typical cloud application might use multiple data stores (Azure storage, SQL Database, Cosmos DB, Redis cache), and a single user action may span multiple subsystems.

You can think of the monitoring and diagnostics process as a pipeline with several distinct stages:



- **Instrumentation.** The raw data for monitoring and diagnostics comes from a variety of sources, including application logs, web server logs, OS performance counters, database logs, and diagnostics built into the Azure platform. Most Azure services have a diagnostics feature that you can use to determine the cause of problems.
- **Collection and storage.** Raw instrumentation data can be held in various locations and with various formats (e.g., application trace logs, IIS logs, performance counters). These disparate sources are collected, consolidated, and put into reliable storage.
- **Analysis and diagnosis.** After the data is consolidated, it can be analyzed to troubleshoot issues and provide an overall view of application health.
- **Visualization and alerts.** In this stage, telemetry data is presented in such a way that an operator can quickly notice problems or trends. Examples include dashboards or email alerts.

Monitoring is not the same as failure detection. For example, your application might detect a transient error and retry, resulting in no downtime. But it should also log the retry operation, so that you can monitor the error rate, in order to get an overall picture of application health.

Application logs are an important source of diagnostics data. Best practices for application logging include:

- Log in production. Otherwise, you lose insight where you need it most.
- 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 will help you pinpoint why the transaction failed.
- Use semantic logging, also known as structured logging. Unstructured logs make it hard to automate the consumption and analysis of the log data, which is needed at cloud scale.
- Use asynchronous logging. Otherwise, the logging system itself can cause the application to fail by causing requests to back up, as they block while waiting to write a logging event.
- Application logging is not the same as auditing. Auditing may be done for compliance or regulatory reasons. As such, audit records must be complete, and it's not acceptable to drop any while processing transactions. If an application requires auditing, this should be kept separate from diagnostics logging.

For more information about monitoring and diagnostics, see [Monitoring and diagnostics guidance](#).

## Respond to failures

Previous sections have focused on automated recovery strategies, which are critical for high availability. However, sometimes manual intervention is needed.

- **Alerts.** Monitor your application for warning signs that may require proactive intervention. For example, if you see that SQL Database or Cosmos DB consistently throttles your application, you might need to increase your database capacity or optimize your queries. In this example, even though the application might handle the throttling errors transparently, your telemetry should still raise an alert so that you can follow up.
- **Manual failover.** Some systems cannot fail over automatically and require a manual failover.
- **Operational readiness testing.** If your application fails over to a secondary region, you should perform an operational readiness test before you fail back to the primary region. The test should verify that the primary region is healthy and ready to receive traffic again.

- **Data consistency check.** If a failure happens in a data store, there may be data inconsistencies when the store becomes available again, especially if the data was replicated.
- **Restoring from backup.** For example, if SQL Database experiences a regional outage, you can geo-restore the database from the latest backup.

Document and test your disaster recovery plan. Evaluate the business impact of application failures. Automate the process as much as possible, and document any manual steps, such as manual failover or data restoration from backups. Regularly test your disaster recovery process to validate and improve the plan.

## Summary

This article discussed resiliency from a holistic perspective, emphasizing some of the unique challenges of the cloud. These include the distributed nature of cloud computing, the use of commodity hardware, and the presence of transient network faults.

Here are the major points to take away from this article:

- Resiliency leads to higher availability, and lower mean time to recover from failures.
- Achieving resiliency in the cloud requires a different set of techniques from traditional on-premises solutions.
- Resiliency does not happen by accident. It must be designed and built in from the start.
- Resiliency touches every part of the application lifecycle, from planning and coding to operations.
- Test and monitor!

# Failure mode analysis

9/28/2018 • 17 minutes to read • [Edit Online](#)

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.

Here is the general process to conduct an FMA:

1. Identify all of the components in the system. Include external dependencies, such as identity providers, third-party services, and so on.
2. For each component, identify potential failures that could occur. A single component may have more than one failure mode. For example, you should consider read failures and write failures separately, because the impact and possible mitigations will be different.
3. Rate each failure mode according to its overall risk. Consider these factors:
  - What is the likelihood of the failure. Is it relatively common? Extremely rare? You don't need exact numbers; the purpose is to help rank the priority.
  - What is the impact on the application, in terms of availability, data loss, monetary cost, and business disruption?
4. For each failure mode, determine how the application will respond and recover. Consider tradeoffs in cost and application complexity.

As a starting point for your FMA process, this article contains a catalog of potential failure modes and their mitigations. The catalog is organized by technology or Azure service, plus a general category for application-level design. The catalog is not exhaustive, but covers many of the core Azure services.

## App Service

### **App Service app shuts down.**

**Detection.** Possible causes:

- Expected shutdown
  - An operator shuts down the application; for example, using the Azure portal.
  - The app was unloaded because it was idle. (Only if the `Always On` setting is disabled.)
- Unexpected shutdown
  - The app crashes.
  - An App Service VM instance becomes unavailable.

Application\_End logging will catch the app domain shutdown (soft process crash) and is the only way to catch the application domain shutdowns.

### **Recovery**

- If the shutdown was expected, use the application's shutdown event to shut down gracefully. For example, in ASP.NET, use the `Application_End` method.
- If the application was unloaded while idle, it is automatically restarted on the next request. However, you will incur the "cold start" cost.

- To prevent the application from being unloaded while idle, enable the `Always On` setting in the web app. See [Configure web apps in Azure App Service](#).
- To prevent an operator from shutting down the app, set a resource lock with `ReadOnly` level. See [Lock resources with Azure Resource Manager](#).
- If the app crashes or an App Service VM becomes unavailable, App Service automatically restarts the app.

**Diagnostics.** Application logs and web server logs. See [Enable diagnostics logging for web apps in Azure App Service](#).

**A particular user repeatedly makes bad requests or overloads the system.**

**Detection.** Authenticate users and include user ID in application logs.

#### Recovery

- Use [Azure API Management](#) to throttle requests from the user. See [Advanced request throttling with Azure API Management](#)
- Block the user.

**Diagnostics.** Log all authentication requests.

**A bad update was deployed.**

**Detection.** Monitor the application health through the Azure Portal (see [Monitor Azure web app performance](#)) or implement the [health endpoint monitoring pattern](#).

**Recovery.** Use multiple [deployment slots](#) and roll back to the last-known-good deployment. For more information, see [Basic web application](#).

## Azure Active Directory

**OpenID Connect (OIDC) authentication fails.**

**Detection.** Possible failure modes include:

1. Azure AD is not available, or cannot be reached due to a network problem. Redirection to the authentication endpoint fails, and the OIDC middleware throws an exception.
2. Azure AD tenant does not exist. Redirection to the authentication endpoint returns an HTTP error code, and the OIDC middleware throws an exception.
3. User cannot authenticate. No detection strategy is necessary; Azure AD handles login failures.

#### Recovery

1. Catch unhandled exceptions from the middleware.
2. Handle `AuthenticationFailed` events.
3. Redirect the user to an error page.
4. User retries.

## Azure Search

**Writing data to Azure Search fails.**

**Detection.** Catch `Microsoft.Rest.Azure.CloudException` errors.

#### Recovery

The [Search .NET SDK](#) automatically retries after transient failures. Any exceptions thrown by the client SDK should be treated as non-transient errors.

The default retry policy uses exponential back-off. To use a different retry policy, call `SetRetryPolicy` on the

`SearchIndexClient` or `SearchServiceClient` class. For more information, see [Automatic Retries](#).

**Diagnostics.** Use [Search Traffic Analytics](#).

**Reading data from Azure Search fails.**

**Detection.** Catch `Microsoft.Rest.Azure.CloudException` errors.

## Recovery

The [Search .NET SDK](#) automatically retries after transient failures. Any exceptions thrown by the client SDK should be treated as non-transient errors.

The default retry policy uses exponential back-off. To use a different retry policy, call `SetRetryPolicy` on the `SearchIndexClient` or `SearchServiceClient` class. For more information, see [Automatic Retries](#).

**Diagnostics.** Use [Search Traffic Analytics](#).

## Cassandra

**Reading or writing to a node fails.**

**Detection.** Catch the exception. For .NET clients, this will typically be `System.Web.HttpException`. Other client may have other exception types. For more information, see [Cassandra error handling done right](#).

## Recovery

- Each [Cassandra client](#) has its own retry policies and capabilities. For more information, see [Cassandra error handling done right](#).
- Use a rack-aware deployment, with data nodes distributed across the fault domains.
- Deploy to multiple regions with local quorum consistency. If a non-transient failure occurs, fail over to another region.

**Diagnostics.** Application logs

## Cloud Service

**Web or worker roles are unexpectedly being shut down.**

**Detection.** The `RoleEnvironmentStopping` event is fired.

**Recovery.** Override the `RoleEntryPoint.OnStop` method to gracefully clean up. For more information, see [The Right Way to Handle Azure OnStop Events](#) (blog).

## Cosmos DB

**Reading data fails.**

**Detection.** Catch `System.Net.Http.HttpRequestException` OR `Microsoft.Azure.Documents.DocumentClientException`.

## Recovery

- The SDK automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. 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 `DocumentClientException`. If you are getting error 429 consistently, consider increasing the throughput value of the collection.
  - If you are using the MongoDB API, the service returns error code 16500 when throttling.
- Replicate the Cosmos DB database across two or more regions. All replicas are readable. Using the client SDKs,

specify the `PreferredLocations` parameter. This is an ordered list of Azure regions. All reads will be sent to the first available region in the list. If the request fails, the client will try the other regions in the list, in order. For more information, see [How to setup Azure Cosmos DB global distribution using the SQL API](#).

**Diagnostics.** Log all errors on the client side.

**Writing data fails.**

**Detection.** Catch `System.Net.Http.HttpRequestException` or `Microsoft.Azure.Documents.DocumentClientException`.

## Recovery

- The SDK automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. 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 `DocumentClientException`. If you are getting error 429 consistently, consider increasing the throughput value of the collection.
- Replicate the Cosmos DB database across two or more regions. If the primary region fails, another region will be promoted to write. You can also trigger a failover manually. The SDK does automatic discovery and routing, so application code continues to work after a failover. During the failover period (typically minutes), write operations will have higher latency, as the SDK finds the new write region. For more information, see [How to setup Azure Cosmos DB global distribution using the SQL API](#).
- As a fallback, persist the document to a backup queue, and process the queue later.

**Diagnostics.** Log all errors on the client side.

## Elasticsearch

**Reading data from Elasticsearch fails.**

**Detection.** Catch the appropriate exception for the particular [Elasticsearch client](#) being used.

## Recovery

- Use a retry mechanism. Each client has its own retry policies.
- Deploy multiple Elasticsearch nodes and use replication for high availability.

For more information, see [Running Elasticsearch on Azure](#).

**Diagnostics.** You can use monitoring tools for Elasticsearch, or log all errors on the client side with the payload. See the 'Monitoring' section in [Running Elasticsearch on Azure](#).

**Writing data to Elasticsearch fails.**

**Detection.** Catch the appropriate exception for the particular [Elasticsearch client](#) being used.

## Recovery

- Use a retry mechanism. Each client has its own retry policies.
- If the application can tolerate a reduced consistency level, consider writing with `write_consistency` setting of `quorum`.

For more information, see [Running Elasticsearch on Azure](#).

**Diagnostics.** You can use monitoring tools for Elasticsearch, or log all errors on the client side with the payload. See the 'Monitoring' section in [Running Elasticsearch on Azure](#).

## Queue storage

## Writing a message to Azure Queue storage fails consistently.

**Detection.** After  $N$  retry attempts, the write operation still fails.

### Recovery

- Store the data in a local cache, and forward the writes to storage later, when the service becomes available.
- Create a secondary queue, and write to that queue if the primary queue is unavailable.

**Diagnostics.** Use [storage metrics](#).

## The application cannot process a particular message from the queue.

**Detection.** Application specific. For example, the message contains invalid data, or the business logic fails for some reason.

### Recovery

Move the message to a separate queue. Run a separate process to examine the messages in that queue.

Consider using Azure Service Bus Messaging queues, which provides a [dead-letter queue](#) functionality for this purpose.

#### NOTE

If you are using Storage queues with WebJobs, the WebJobs SDK provides built-in poison message handling. See [How to use Azure queue storage with the WebJobs SDK](#).

**Diagnostics.** Use application logging.

## Redis Cache

### Reading from the cache fails.

**Detection.** Catch `StackExchange.Redis.RedisConnectionException`.

### Recovery

1. Retry on transient failures. Azure Redis cache supports built-in retry through See [Redis Cache retry guidelines](#).
2. Treat non-transient failures as a cache miss, and fall back to the original data source.

**Diagnostics.** Use [Redis Cache diagnostics](#).

### Writing to the cache fails.

**Detection.** Catch `StackExchange.Redis.RedisConnectionException`.

### Recovery

1. Retry on transient failures. Azure Redis cache supports built-in retry through See [Redis Cache retry guidelines](#).
2. If the error is non-transient, ignore it and let other transactions write to the cache later.

**Diagnostics.** Use [Redis Cache diagnostics](#).

## SQL Database

### Cannot connect to the database in the primary region.

**Detection.** Connection fails.

### Recovery

Prerequisite: The database must be configured for active geo-replication. See [SQL Database Active Geo-Replication](#).

- For queries, read from a secondary replica.
- For inserts and updates, manually fail over to a secondary replica. See [Initiate a planned or unplanned failover for Azure SQL Database](#).

The replica uses a different connection string, so you will need to update the connection string in your application.

### Client runs out of connections in the connection pool.

**Detection.** Catch `System.InvalidOperationException` errors.

#### Recovery

- Retry the operation.
- As a mitigation plan, isolate the connection pools for each use case, so that one use case can't dominate all the connections.
- Increase the maximum connection pools.

**Diagnostics.** Application logs.

### Database connection limit is reached.

**Detection.** Azure SQL Database limits the number of concurrent workers, logins, and sessions. The limits depend on the service tier. For more information, see [Azure SQL Database resource limits](#).

To detect these errors, catch `System.Data.SqlClient.SqlException` and check the value of `SqlException.Number` for the SQL error code. For a list of relevant error codes, see [SQL error codes for SQL Database client applications: Database connection error and other issues](#).

**Recovery.** These errors are considered transient, so retrying may resolve the issue. If you consistently hit these errors, consider scaling the database.

**Diagnostics.** - The `sys.event_log` query returns successful database connections, connection failures, and deadlocks.

- Create an [alert rule](#) for failed connections.
- Enable [SQL Database auditing](#) and check for failed logins.

## Service Bus Messaging

### Reading a message from a Service Bus queue fails.

**Detection.** Catch exceptions from the client SDK. The base class for Service Bus exceptions is `MessagingException`. If the error is transient, the `IsTransient` property is true.

For more information, see [Service Bus messaging exceptions](#).

#### Recovery

1. Retry on transient failures. See [Service Bus retry guidelines](#).
2. Messages that cannot be delivered to any receiver are placed in a *dead-letter queue*. Use this queue to see which messages could not be received. There is no automatic cleanup of the dead-letter queue. Messages remain there until you explicitly retrieve them. See [Overview of Service Bus dead-letter queues](#).

### Writing a message to a Service Bus queue fails.

**Detection.** Catch exceptions from the client SDK. The base class for Service Bus exceptions is `MessagingException`. If the error is transient, the `IsTransient` property is true.

For more information, see [Service Bus messaging exceptions](#).

## Recovery

1. The Service Bus client automatically retries after transient errors. By default, it uses exponential back-off. After the maximum retry count or maximum timeout period, the client throws an exception. For more information, see [Service Bus retry guidelines](#).
2. If the queue quota is exceeded, the client throws [QuotaExceededException](#). The exception message gives more details. Drain some messages from the queue before retrying, and consider using the Circuit Breaker pattern to avoid continued retries while the quota is exceeded. Also, make sure the [BrokeredMessage.TimeToLive](#) property is not set too high.
3. Within a region, resiliency can be improved by using [partitioned queues or topics](#). A non-partitioned queue or topic is assigned to one messaging store. If this messaging store is unavailable, all operations on that queue or topic will fail. A partitioned queue or topic is partitioned across multiple messaging stores.
4. For additional resiliency, create two Service Bus namespaces in different regions, and replicate the messages. You can use either active replication or passive replication.
  - Active replication: The client sends every message to both queues. The receiver listens on both queues. Tag messages with a unique identifier, so the client can discard duplicate messages.
  - Passive replication: The client sends the message to one queue. If there is an error, the client falls back to the other queue. The receiver listens on both queues. This approach reduces the number of duplicate messages that are sent. However, the receiver must still handle duplicate messages.

For more information, see [GeoReplication sample](#) and [Best practices for insulating applications against Service Bus outages and disasters](#).

## Duplicate message

**Detection.** Examine the `MessageId` and `DeliveryCount` properties of the message.

## Recovery

- If possible, design your message processing operations to be idempotent. Otherwise, store message IDs of messages that are already processed, and check the ID before processing a message.
- Enable duplicate detection, by creating the queue with `RequiresDuplicateDetection` set to true. With this setting, Service Bus automatically deletes any message that is sent with the same `MessageId` as a previous message. Note the following:
  - This setting prevents duplicate messages from being put into the queue. It doesn't prevent a receiver from processing the same message more than once.
  - Duplicate detection has a time window. If a duplicate is sent beyond this window, it won't be detected.

**Diagnostics.** Log duplicated messages.

## The application cannot process a particular message from the queue.

**Detection.** Application specific. For example, the message contains invalid data, or the business logic fails for some reason.

## Recovery

There are two failure modes to consider.

- The receiver detects the failure. In this case, move the message to the dead-letter queue. Later, run a separate process to examine the messages in the dead-letter queue.
- The receiver fails in the middle of processing the message — for example, due to an unhandled exception. To

handle this case, use `PeekLock` mode. In this mode, if the lock expires, the message becomes available to other receivers. If the message exceeds the maximum delivery count or the time-to-live, the message is automatically moved to the dead-letter queue.

For more information, see [Overview of Service Bus dead-letter queues](#).

**Diagnostics.** Whenever the application moves a message to the dead-letter queue, write an event to the application logs.

## Service Fabric

**A request to a service fails.**

**Detection.** The service returns an error.

### Recovery

- Locate a proxy again (`serviceProxy` or `ActorProxy`) and call the service/actor method again.
- **Stateful service.** Wrap operations on reliable collections in a transaction. If there is an error, the transaction will be rolled back. The request, if pulled from a queue, will be processed again.
- **Stateless service.** If the service persists data to an external store, all operations need to be idempotent.

**Diagnostics.** Application log

**Service Fabric node is shut down.**

**Detection.** A cancellation token is passed to the service's `RunAsync` method. Service Fabric cancels the task before shutting down the node.

**Recovery.** Use the cancellation token to detect shutdown. When Service Fabric requests cancellation, finish any work and exit `RunAsync` as quickly as possible.

**Diagnostics.** Application logs

## Storage

**Writing data to Azure Storage fails**

**Detection.** The client receives errors when writing.

### Recovery

1. Retry the operation, to recover from transient failures. The [retry policy](#) in the client SDK handles this automatically.
2. Implement the Circuit Breaker pattern to avoid overwhelming storage.
3. If N retry attempts fail, perform a graceful fallback. For example:
  - Store the data in a local cache, and forward the writes to storage later, when the service becomes available.
  - If the write action was in a transactional scope, compensate the transaction.

**Diagnostics.** Use [storage metrics](#).

**Reading data from Azure Storage fails.**

**Detection.** The client receives errors when reading.

### Recovery

1. Retry the operation, to recover from transient failures. The [retry policy](#) in the client SDK handles this

automatically.

- For RA-GRS storage, if reading from the primary endpoint fails, try reading from the secondary endpoint. The client SDK can handle this automatically. See [Azure Storage replication](#).
- If  $N$  retry attempts fail, take a fallback action to degrade gracefully. For example, if a product image can't be retrieved from storage, show a generic placeholder image.

**Diagnostics.** Use [storage metrics](#).

## Virtual Machine

**Connection to a backend VM fails.**

**Detection.** Network connection errors.

### Recovery

- Deploy at least two backend VMs in an availability set, behind a load balancer.
- If the connection error is transient, sometimes TCP will successfully retry sending the message.
- Implement a retry policy in the application.
- For persistent or non-transient errors, implement the [Circuit Breaker](#) pattern.
- If the calling VM exceeds its network egress limit, the outbound queue will fill up. If the outbound queue is consistently full, consider scaling out.

**Diagnostics.** Log events at service boundaries.

**VM instance becomes unavailable or unhealthy.**

**Detection.** Configure a Load Balancer [health probe](#) that signals whether the VM instance is healthy. The probe should check whether critical functions are responding correctly.

**Recovery.** For each application tier, put multiple VM instances into the same availability set, and place a load balancer in front of the VMs. If the health probe fails, the Load Balancer stops sending new connections to the unhealthy instance.

**Diagnostics.** - Use Load Balancer [log analytics](#).

- Configure your monitoring system to monitor all of the health monitoring endpoints.

**Operator accidentally shuts down a VM.**

**Detection.** N/A

**Recovery.** Set a resource lock with `ReadOnly` level. See [Lock resources with Azure Resource Manager](#).

**Diagnostics.** Use [Azure Activity Logs](#).

## WebJobs

**Continuous job stops running when the SCM host is idle.**

**Detection.** Pass a cancellation token to the WebJob function. For more information, see [Graceful shutdown](#).

**Recovery.** Enable the `Always On` setting in the web app. For more information, see [Run Background tasks with WebJobs](#).

## Application design

**Application can't handle a spike in incoming requests.**

**Detection.** Depends on the application. Typical symptoms:

- The website starts returning HTTP 5xx error codes.
- Dependent services, such as database or storage, start to throttle requests. Look for HTTP errors such as HTTP 429 (Too Many Requests), depending on the service.
- HTTP queue length grows.

## Recovery

- Scale out to handle increased load.
- Mitigate failures to avoid having cascading failures disrupt the entire application. Mitigation strategies include:
  - Implement the [Throttling Pattern](#) to avoid overwhelming backend systems.
  - Use [queue-based load leveling](#) to buffer requests and process them at an appropriate pace.
  - Prioritize certain clients. For example, if the application has free and paid tiers, throttle customers on the free tier, but not paid customers. See [Priority queue pattern](#).

**Diagnostics.** Use [App Service diagnostic logging](#). Use a service such as [Azure Log Analytics](#), [Application Insights](#), or [New Relic](#) to help understand the diagnostic logs.

## One of the operations in a workflow or distributed transaction fails.

**Detection.** After  $N$  retry attempts, it still fails.

## Recovery

- As a mitigation plan, implement the [Scheduler Agent Supervisor](#) pattern to manage the entire workflow.
- Don't retry on timeouts. There is a low success rate for this error.
- Queue work, in order to retry later.

**Diagnostics.** Log all operations (successful and failed), including compensating actions. Use correlation IDs, so that you can track all operations within the same transaction.

## A call to a remote service fails.

**Detection.** HTTP error code.

## Recovery

1. Retry on transient failures.
2. If the call fails after  $N$  attempts, take a fallback action. (Application specific.)
3. Implement the [Circuit Breaker pattern](#) to avoid cascading failures.

**Diagnostics.** Log all remote call failures.

## Next steps

For more information about the FMA process, see [Resilience by design for cloud services](#) (PDF download).

# Availability checklist

10/24/2018 • 10 minutes to read • [Edit Online](#)

Availability is the proportion of time that a system is functional and working, and is one of the [pillars of software quality](#). Use this checklist to review your application architecture from an availability standpoint.

## Application design

**Avoid any single point of failure.** All components, services, resources, and compute instances should be deployed as multiple instances to prevent a single point of failure from affecting availability. This includes authentication mechanisms. Design the application to be configurable to use multiple instances, and to automatically detect failures and redirect requests to non-failed instances where the platform does not do this automatically.

**Decompose workloads by service-level objective.** If a service is composed of critical and less-critical workloads, manage them differently and specify the service features and number of instances to meet their availability requirements.

**Minimize and understand service dependencies.** Minimize the number of different services used where possible, and ensure you understand all of the feature and service dependencies that exist in the system. This includes the nature of these dependencies, and the impact of failure or reduced performance in each one on the overall application.

**Design tasks and messages to be idempotent where possible.** An operation is idempotent if it can be repeated multiple times and produce the same result. Idempotency can ensure that duplicated requests don't cause problems. Message consumers and the operations they carry out should be idempotent so that repeating a previously executed operation does not render the results invalid. This may mean detecting duplicated messages, or ensuring consistency by using an optimistic approach to handling conflicts.

**Use a message broker that implements high availability for critical transactions.** Many cloud applications use messaging to initiate tasks that are performed asynchronously. To guarantee delivery of messages, the messaging system should provide high availability. [Azure Service Bus Messaging](#) implements *at least once* semantics. This means that a message posted to a queue will not be lost, although duplicate copies may be delivered under certain circumstances. If message processing is idempotent (see the previous item), repeated delivery should not be a problem.

**Design applications to gracefully degrade.** The load on an application may exceed the capacity of one or more parts, causing reduced availability and failed connections. Scaling can help to alleviate this, but it may reach a limit imposed by other factors, such as resource availability or cost. When an application reaches a resource limit, it should take appropriate action to minimize the impact for the user. For example, in an ecommerce system, if the order-processing subsystem is under strain or fails, it can be temporarily disabled while allowing other functionality, such as browsing the product catalog. It might be appropriate to postpone requests to a failing subsystem, for example still enabling customers to submit orders but saving them for later processing, when the orders subsystem is available again.

**Gracefully handle rapid burst events.** Most applications need to handle varying workloads over time. Auto-scaling can help to handle the load, but it may take some time for additional instances to come online and handle requests. Prevent sudden and unexpected bursts of activity from overwhelming the application: design it to queue requests to the services it uses and degrade gracefully when queues are near to full capacity. Ensure that there is sufficient performance and capacity available under non-burst conditions to drain the queues and handle outstanding requests. For more information, see the [Queue-Based Load Leveling Pattern](#).

## Deployment and maintenance

**Deploy multiple instances of services.** If your application depends on a single instance of a service, it creates a single point of failure. Provisioning multiple instances improves both resiliency and scalability. 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 \(VMs\)](#), ensure that your VM architecture includes more than one VM and that each VM is included in an [availability set](#).

**Consider deploying your application across multiple regions.** If your application is deployed to a single region, in the rare event the entire region becomes unavailable, your application will also be unavailable. This may be unacceptable under the terms of your application's SLA. If so, consider deploying your application and its services across multiple regions.

**Automate and test deployment and maintenance tasks.** Distributed applications consist of multiple parts that must work together. Deployment should be automated, using tested and proven mechanisms such as scripts. These can update and validate configuration, and automate the deployment process. Use [Azure Resource Manager templates](#) to provision Azure resource. Also use automated techniques to perform application updates. It is vital to test all of these processes fully to ensure that errors do not cause additional downtime. All deployment tools must have suitable security restrictions to protect the deployed application; define and enforce deployment policies carefully and minimize the need for human intervention.

**Use staging and production features of the platform..** For example, Azure App Service supports [deployment slots](#), which you can use to stage a deployment before swapping it to production. Azure Service Fabric supports [rolling upgrades](#) to application services.

**Place virtual machines (VMs) in an availability set.** To maximize availability, create multiple instances of each VM role and place these instances in the same availability set. If you have multiple VMs that serve different roles, such as different application tiers, create an availability set for each VM role. For example, create an availability set for the web tier and another for the data tier.

## Data management

**Geo-replicate data in Azure Storage.** Data in Azure Storage is automatically replicated within in a datacenter. For even higher availability, use Read-access geo-redundant storage (-RAGRS), which replicates your data to a secondary region and provides read-only access to the data in the secondary location. The data is durable even in the case of a complete regional outage or a disaster. For more information, see [Azure Storage replication](#).

**Geo-replicate databases.** Azure SQL Database and Cosmos DB both support geo-replication, which enables you to configure secondary database replicas in other regions. Secondary databases are available for querying and for failover in the case of a data center outage or the inability to connect to the primary database. For more information, see [Failover groups and active geo-replication](#) (SQL Database) and [How to distribute data globally with Azure Cosmos DB](#).

**Use optimistic concurrency and eventual consistency.** Transactions that block access to resources through locking (pessimistic concurrency) can cause poor performance and considerably reduce availability. These problems can become especially acute in distributed systems. In many cases, careful design and techniques such as partitioning can minimize the chances of conflicting updates occurring. Where data is replicated, or is read from a separately updated store, the data will only be eventually consistent. But the advantages usually far outweigh the impact on availability of using transactions to ensure immediate consistency.

**Use periodic backup and point-in-time restore.** Regularly and automatically back up data that is not preserved elsewhere, and verify you can reliably restore both the data and the application itself should a failure occur. Ensure that backups meet your Recovery Point Objective (RPO). Data replication is not a backup feature, because human error or malicious operations can corrupt data across all the replicas. The backup process must be secure to protect the data in transit and in storage. Databases or parts of a data store can usually be recovered to

a previous point in time by using transaction logs. For more information, see [Recover from data corruption or accidental deletion](#)

## Errors and failures

**Configure request timeouts.** Services and resources may become unavailable, causing requests to fail. Ensure that the timeouts you apply are appropriate for each service or resource as well as the client that is accessing them. In some cases, you might allow a longer timeout for a particular instance of a client, depending on the context and other actions that the client is performing. Very short timeouts may cause excessive retry operations for services and resources that have considerable latency. Very long timeouts can cause blocking if a large number of requests are queued, waiting for a service or resource to respond.

**Retry failed operations caused by transient faults.** Design a retry strategy for access to all services and resources where they do not inherently support automatic connection retry. Use a strategy that includes an increasing delay between retries as the number of failures increases, to prevent overloading of the resource and to allow it to gracefully recover and handle queued requests. Continual retries with very short delays are likely to exacerbate the problem. For more information, see [Retry guidance for specific services](#).

**Implement circuit breaking to avoid cascading failures.** There may be situations in which transient or other faults, ranging in severity from a partial loss of connectivity to the complete failure of a service, take much longer than expected to return to normal. If a service is very busy, failure in one part of the system may lead to cascading failures, and result in many operations becoming blocked while holding onto critical system resources such as memory, threads, and database connections. Instead of continually retrying an operation that is unlikely to succeed, the application should quickly accept that the operation has failed, and gracefully handle this failure. Use the Circuit Breaker pattern to reject requests for specific operations for defined periods. For more information, see [Circuit Breaker Pattern](#).

**Compose or fall back to multiple components.** Design applications to use multiple instances without affecting operation and existing connections where possible. Use multiple instances and distribute requests between them, and detect and avoid sending requests to failed instances, in order to maximize availability.

**Fall back to a different service or workflow.** For example, if writing to SQL Database fails, temporarily store data in blob storage or Redis Cache. Provide a way to replay the writes to SQL Database when the service becomes available. In some cases, a failed operation may have an alternative action that allows the application to continue to work even when a component or service fails. If possible, detect failures and redirect requests to other services that can offer a suitable alternative functionality, or to back up or reduced functionality instances that can maintain core operations while the primary service is offline.

## Monitoring and disaster recovery

**Provide rich instrumentation for likely failures and failure events** to report the situation to operations staff. For failures that are likely but have not yet occurred, provide sufficient data to enable operations staff 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 attempt to continue running, albeit with reduced functionality. In all cases, the monitoring system should capture comprehensive details to enable operations staff to effect a quick recovery, and if necessary, for designers and developers to modify the system to prevent the situation from arising again.

**Monitor system health by implementing checking functions.** The health and performance of an application can degrade over time, without being noticeable until it fails. Implement probes or check functions that are executed 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 individual services that the application uses, or for individual components. Check functions can execute processes to ensure they produce valid results, measure latency and check availability, and extract information from the system.

**Regularly test all failover and fallback systems.** Changes to systems and operations may affect failover and fallback functions, but the impact may not be detected until the main system fails or becomes overloaded. Test it before it is required to compensate for a live problem at runtime.

**Test the monitoring systems.** Automated failover and fallback systems, and manual visualization of system health and performance by using dashboards, all depend on monitoring and instrumentation functioning correctly. If these elements fail, miss critical information, or report inaccurate data, an operator might not realize that the system is unhealthy or failing.

**Track the progress of long-running workflows and retry on failure.** Long-running workflows are often composed of multiple steps. Ensure that each step is independent and can be retried to minimize the chance that the entire workflow will need to be rolled back, or that multiple compensating transactions need to be executed. Monitor and manage the progress of long-running workflows by implementing a pattern such as [Scheduler Agent Supervisor Pattern](#).

**Plan for disaster recovery.** Create an accepted, fully-tested plan for recovery from any type of failure that may affect system availability. Choose a multi-site disaster recovery architecture for any mission-critical applications. Identify a specific owner of the disaster recovery plan, including automation and testing. Ensure the plan is well-documented, and automate the process as much as possible. Establish a backup strategy for all reference and transactional data, and test the restoration of these backups regularly. Train operations staff to execute the plan, and perform regular disaster simulations to validate and improve the plan.

# DevOps Checklist

4/11/2018 • 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 nonoptimal 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 just 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.

**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.

**Automate Deployments.** Automate all deployments, and have systems in place to detect any problems during rollout. Have a mitigation process for preserving the existing code and data in production, before the update replaces them in all production instances. Have an automated way to roll forward fixes or roll back changes.

**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 lets you correlate events across systems. [Azure Diagnostics](#) and [Application Insights](#) are the standard method of tracking the health and status of Azure resources. Microsoft [Operation Management Suite](#) 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 resilient applications for Azure](#). 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 role-based access control.** Assigning user accounts and access to resources should not be a manual process. Use [Role-Based Access Control \(RBAC\)](#) 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.

# Resiliency checklist

10/5/2018 • 17 minutes to read • [Edit Online](#)

Resiliency is the ability of a system to recover from failures and continue to function, and is one of the [pillars of software quality](#). Designing your application for resiliency requires planning for and mitigating a variety of failure modes that could occur. Use this checklist to review your application architecture from a resiliency standpoint. Also review the [Resiliency checklist for specific Azure services](#).

## Requirements

**Define your customer's availability requirements.** Your customer will have availability requirements for the components in your application and this will affect your application's design. Get agreement from your customer for the availability targets of each piece of your application, otherwise your design may not meet the customer's expectations. For more information, see [Designing resilient applications for Azure](#).

## Application Design

**Perform a failure mode analysis (FMA) for your application.** FMA is a process for building resiliency into an application early in the design stage. For more information, see [Failure mode analysis](#). The goals of an FMA include:

- Identify what types of failures an application might experience.
- Capture the potential effects and impact of each type of failure on the application.
- Identify recovery strategies.

**Deploy multiple instances of services.** If your application depends on a single instance of a service, it creates a single point of failure. Provisioning multiple instances improves both resiliency and scalability. 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 \(VMs\)](#), ensure that your VM architecture includes more than one VM and that each VM is included in an [availability set](#).

**Use autoscaling to respond to increases in load.** If your application is not 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. For more details, see the following:

- General: [Scalability checklist](#)
- Azure App Service: [Scale instance count manually or automatically](#)
- Cloud Services: [How to auto scale a cloud service](#)
- Virtual Machines: [Automatic scaling and virtual machine scale sets](#)

**Use load balancing to distribute requests.** Load balancing distributes your application's requests to healthy service instances by removing unhealthy instances from rotation. If your service uses Azure App Service or Azure Cloud Services, it is already load balanced for you. However, if your application uses Azure VMs, you will need to provision a load balancer. See the [Azure Load Balancer](#) overview for more details.

**Configure Azure Application Gateways to use multiple instances.** Depending on your application's requirements, an [Azure Application Gateway](#) may be better suited to distributing requests to your application's services. However, single instances of the Application Gateway service are not guaranteed by an SLA so it's possible that your application could fail if the Application Gateway instance fails. Provision more than one medium or larger Application Gateway instance to guarantee availability of the service under the terms of the [SLA](#).

**Use Availability Sets for each application tier.** Placing your instances in an [availability set](#) provides a higher SLA.

**Consider deploying your application across multiple regions.** If your application is deployed to a single region, in the rare event the entire region becomes unavailable, your application will also be unavailable. This may be unacceptable under the terms of your application's SLA. If so, consider deploying your application and its services across multiple regions. A multi-region deployment can use an active-active pattern (distributing requests across multiple active instances) or an active-passive pattern (keeping a "warm" instance in reserve, in case the primary instance fails). We recommend that you deploy multiple instances of your application's services across regional pairs. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

**Use Azure Traffic Manager to route your application's traffic to different regions.** Azure Traffic Manager performs load balancing at the DNS level and will route traffic to different regions based on the [traffic routing](#) method you specify and the health of your application's endpoints. Without Traffic Manager, you are limited to a single region for your deployment, which limits scale, increases latency for some users, and causes application downtime in the case of a region-wide service disruption.

**Configure and test health probes for your load balancers and traffic managers.** Ensure that your health logic checks the critical parts of the system and responds appropriately to health probes.

- The health probes for [Azure Traffic Manager](#) and [Azure Load Balancer](#) serve a specific function. For Traffic Manager, the health probe determines whether to fail over to another region. For a load balancer, it determines whether to remove a VM from rotation.
- For a Traffic Manager probe, your health endpoint should check any critical dependencies that are deployed within the same region, and whose failure should trigger a failover to another region.
- For a load balancer, 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.
- For guidance on implementing health monitoring in your application, see [Health Endpoint Monitoring Pattern](#).

**Monitor third-party services.** If your application has dependencies on third-party services, identify where and how these third-party services can fail and what effect those failures will have on your application. A third-party service may not include monitoring and diagnostics, so it's important to log your invocations of them and correlate them with your application's health and diagnostic logging using a unique identifier. For more information on proven practices for monitoring and diagnostics, see [Monitoring and Diagnostics guidance](#).

**Ensure that any third-party service you consume provides an SLA.** If your application depends on a third-party service, but the third party provides no guarantee of availability in the form of an SLA, your application's availability also cannot be guaranteed. Your SLA is only as good as the least available component of your application.

**Implement resiliency patterns for remote operations where appropriate.** If your application depends on communication between remote services, follow [design patterns](#) for dealing with transient failures, such as [Retry Pattern](#), and [Circuit Breaker Pattern](#).

**Implement asynchronous operations whenever possible.** Synchronous operations can monopolize resources and block other operations while the caller waits for the process to complete. Design each part of your application to allow for asynchronous operations whenever possible. For more information on how to implement asynchronous programming in C#, see [Asynchronous Programming with async and await](#).

## Data management

**Understand the replication methods for your application's data sources.** Your application data will be stored in different data sources and have different availability requirements. Evaluate the replication methods for each type of data storage in Azure, including [Azure Storage Replication](#) and [SQL Database Active Geo-Replication](#) to

ensure that your application's data requirements are satisfied.

**Ensure that no single user account has access to both production and backup data.** Your data backups are compromised if one single user account has permission to write to both production and backup sources. A malicious user could purposely delete all your data, while a regular user could accidentally delete it. Design your application to limit the permissions of each user account so that only the users that require write access have write access and it's only to either production or backup, but not both.

**Document your data source fail over and fail back process and test it.** In the case where your data source fails catastrophically, a human operator will have to follow a set of documented instructions to fail over to a new data source. If the documented steps have errors, an operator will not be able to successfully follow them and fail over the resource. Regularly test the instruction steps to verify that an operator following them is able to successfully fail over and fail back the data source.

**Validate your data backups.** Regularly verify that your backup data is what you expect by running a script to validate data integrity, schema, and queries. There's no point having a backup if it's not useful to restore your data sources. Log and report any inconsistencies so the backup service can be repaired.

**Consider using a storage account type that is geo-redundant.** Data stored in an Azure Storage account is always replicated locally. However, there are multiple replication strategies to choose from when a Storage Account is provisioned. Select [Azure Read-Access Geo Redundant Storage \(RA-GRS\)](#) to protect your application data against the rare case when an entire region becomes unavailable.

**NOTE**

For VMs, do not rely on RA-GRS replication to restore the VM disks (VHD files). Instead, use [Azure Backup](#).

## Security

**Implement application-level protection against distributed denial of service (DDoS) attacks.** Azure services are protected against DDoS attacks at the network layer. However, Azure cannot protect against application-layer attacks, because it is difficult to distinguish between true user requests from malicious user requests. For more information on how to protect against application-layer DDoS attacks, see the "Protecting against DDoS" section of [Microsoft Azure Network Security](#) (PDF download).

**Implement the principle of least privilege for access to the application's resources.** The default for access to the application's resources should be as restrictive as possible. Grant higher level permissions on an approval basis. Granting overly permissive access to your application's resources by default can result in someone purposely or accidentally deleting resources. Azure provides [role-based access control](#) to manage user privileges, but it's important to verify least privilege permissions for other resources that have their own permissions systems such as SQL Server.

## Testing

**Perform failover and fallback testing for your application.** If you haven't fully tested failover and fallback, you can't be certain that the dependent services in your application come back up in a synchronized manner during disaster recovery. Ensure that your application's dependent services failover and fail back in the correct order.

**Perform fault-injection testing for your application.** Your application can fail for many different reasons, such as certificate expiration, exhaustion of system resources in a VM, or storage failures. Test your application in an environment as close as possible to production, by simulating or triggering real failures. For example, delete certificates, artificially consume system resources, or delete a storage source. Verify your application's ability to recover from all types of faults, alone and in combination. Check that failures are not propagating or cascading through your system.

**Run tests in production using both synthetic and real user data.** Test and production are rarely identical, so it's important to use blue/green or a canary deployment and test your application in production. This allows you to test your application in production under real load and ensure it will function as expected when fully deployed.

## Deployment

**Document the release process for your application.** Without detailed release process documentation, an operator might deploy a bad update or improperly configure settings for your application. Clearly define and document your release process, and ensure that it's available to the entire operations team.

**Automate your application's deployment process.** If your operations staff is required to manually deploy your application, human error can cause the deployment to fail.

**Design your release process to maximize application availability.** If your release process requires services to go offline during deployment, your application will be unavailable until they come back online. Use the [blue/green](#) or [canary release](#) deployment technique to deploy your application to production. Both of these techniques involve deploying your release code alongside production code so users of release code can be redirected to production code in the event of a failure.

**Log and audit your application's deployments.** If you use staged deployment techniques such as blue/green or canary releases there will be more than one version of your application running in production. If a problem should occur, it's critical to determine which version of your application is causing a problem. Implement a robust logging strategy to capture as much version-specific information as possible.

**Have a rollback plan for deployment.** It's possible that your application deployment could fail and cause your application to become unavailable. Design a rollback process to go back to a last known good version and minimize downtime.

## Operations

**Implement best practices for monitoring and alerting in your application.** Without proper monitoring, diagnostics, and alerting, there is no way to detect failures in your application and alert an operator to fix them. For more information, see [Monitoring and Diagnostics guidance](#).

**Measure remote call statistics and make the information available to the application team.** If you don't track and report remote call statistics in real time and provide an easy way to review this information, the operations team will not have an instantaneous view into the health of your application. And if you only measure average remote call time, you will not have enough information to reveal issues in the services. 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 the number of transient exceptions and retries over an appropriate timeframe.** If you don't track and monitor transient exceptions and retry attempts over time, it's possible that an issue or failure could be hidden by your application's retry logic. That is, if your monitoring and logging only shows success or failure of an operation, the fact that the operation had to be retried multiple times due to exceptions will be hidden. A trend of increasing exceptions over time indicates that the service is having an issue and may fail. For more information, see [Retry service specific guidance](#).

**Implement an early warning system that alerts an operator.** Identify the key performance indicators of your 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. Set these thresholds at levels that identify issues before they become critical and require a recovery response.

**Ensure that more than one person on the team is trained to monitor the application and perform any manual recovery steps.** If you only have a single operator on the team who can monitor the application and kick

off recovery steps, that person becomes a single point of failure. Train multiple individuals on detection and recovery and make sure there is always at least one active at any time.

**Ensure that your application does not run up against Azure subscription limits.** Azure subscriptions have limits on certain resource types, such as number of resource groups, number of cores, and number of storage accounts. If your application requirements exceed Azure subscription limits, create another Azure subscription and provision sufficient resources there.

**Ensure that your application does not run up against per-service limits.** Individual Azure services have consumption limits — for example, limits on storage, throughput, number of connections, requests per second, and other metrics. Your application will fail if it attempts to use resources beyond these limits. This will result in service throttling and possible downtime for affected users. Depending on the specific service and your application requirements, you can often avoid these limits by scaling up (for example, choosing another pricing tier) or scaling out (adding new instances).

**Design your application's storage requirements to fall within Azure storage scalability and performance targets.** Azure storage is designed to function within predefined scalability and performance targets, so design your application to utilize storage within those targets. If you exceed these targets your application will experience storage throttling. To fix this, provision additional Storage Accounts. If you run up against the Storage Account limit, provision additional Azure Subscriptions and then provision additional Storage Accounts there. For more information, see [Azure Storage Scalability and Performance Targets](#).

**Select the right VM size for your application.** Measure the actual CPU, memory, disk, and I/O of your VMs in production and verify that the VM size you've selected is sufficient. If not, your application may experience capacity issues as the VMs approach their limits. VM sizes are described in detail in [Sizes for virtual machines in Azure](#).

**Determine if your application's workload is stable or fluctuating over time.** If your workload fluctuates over time, use Azure VM scale sets to automatically scale the number of VM instances. Otherwise, you will have to manually increase or decrease the number of VMs. For more information, see the [Virtual Machine Scale Sets Overview](#).

**Select the right service tier for Azure SQL Database.** If your application uses Azure SQL Database, ensure that you have selected the appropriate service tier. If you select a tier that is not able 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, see [SQL Database options and performance: Understand what's available in each service tier](#).

**Create a process for interacting with Azure support.** If the process for contacting [Azure support](#) is not set before the need to contact support arises, downtime will be prolonged as the support process is navigated for the first time. Include the process for contacting support and escalating issues as part of your application's resiliency from the outset.

**Ensure that your application doesn't use more than the maximum number of storage accounts per subscription.** Azure allows a maximum of 200 storage accounts per subscription. If your application requires more storage accounts than are currently available in your subscription, you will have to create a new subscription and create additional storage accounts there. For more information, see [Azure subscription and service limits, quotas, and constraints](#).

**Ensure that your application doesn't exceed the scalability targets for virtual machine disks.** An Azure IaaS VM supports attaching a number of data disks depending on several factors, including the VM size and 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. For more information, see [Azure Storage Scalability and Performance Targets](#)

## Telemetry

**Log telemetry data while the application is running in the production environment.** Capture robust telemetry information while the application is running in the production environment or you will not have sufficient information to diagnose the cause of issues while it's actively serving users. For more information, see [Monitoring and Diagnostics](#).

**Implement logging using an asynchronous pattern.** If logging operations are synchronous, they might block your application code. Ensure that your logging operations are implemented as asynchronous operations.

**Correlate log data across service boundaries.** In a typical n-tier application, a user request may traverse several service boundaries. For example, a user request typically originates in the web tier and is passed to the business tier and finally persisted in the data tier. In more complex scenarios, a user request may be distributed to many different services and data stores. Ensure that your logging system correlates calls across service boundaries so you can track the request throughout your application.

## Azure Resources

**Use Azure Resource Manager templates to provision resources.** Resource Manager templates make it easier to automate deployments via PowerShell or the Azure CLI, which leads to a more reliable deployment process. For more information, see [Azure Resource Manager overview](#).

**Give resources meaningful names.** Giving resources meaningful names makes it easier to locate a specific resource and understand its role. For more information, see [Naming conventions for Azure resources](#)

**Use role-based access control (RBAC).** Use RBAC to control access to the Azure resources that you deploy. RBAC lets you assign authorization roles to members of your DevOps team, to prevent accidental deletion or changes to deployed resources. For more information, see [Get started with access management in the Azure portal](#)

**Use resource locks for critical resources, such as VMs.** Resource locks prevent an operator from accidentally deleting a resource. For more information, see [Lock resources with Azure Resource Manager](#)

**Choose regional pairs.** When deploying to two regions, choose regions from the same regional pair. In the event of a broad outage, recovery of one region is prioritized out of every pair. Some services such as Geo-Redundant Storage provide automatic replication to the paired region. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#)

**Organize resource groups by function and lifecycle.** In general, a resource group should contain resources that share the same lifecycle. This makes it easier to manage deployments, delete test deployments, and assign access rights, reducing the chance that a production deployment is accidentally deleted or modified. Create separate resource groups for production, development, and test environments. In a multi-region deployment, put resources for each region into separate resource groups. This makes it easier to redeploy one region without affecting the other region(s).

## Next steps

- [Resiliency checklist for specific Azure services](#)
- [Failure mode analysis](#)

# Resiliency checklist for specific Azure services

10/5/2018 • 14 minutes to read • [Edit Online](#)

Resiliency is the ability of a system to recover from failures and continue to function, and is one of the [pillars of software quality](#). 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. Also review the [general resiliency checklist](#).

## App Service

**Use Standard or Premium tier.** These tiers support staging slots and automated backups. For more information, see [Azure App Service plans in-depth overview](#)

**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.

**Store configuration as app settings.** Use app settings to hold configuration settings as app settings. Define the settings in your Resource Manager templates, or using PowerShell, so that you can apply them as part of an automated deployment / update process, which is more reliable. For more information, see [Configure web apps in Azure App Service](#).

**Create separate App Service plans for production and test.** Don't use slots on your production deployment for testing. All apps within the same App Service plan share the same VM instances. If you put production and test deployments in the same plan, it can negatively affect the production deployment. For example, load tests might degrade the live production site. By putting test deployments into a separate plan, you isolate them from the production version.

**Separate web apps from web APIs.** If your solution has both a web front-end and a web API, consider decomposing them into separate App Service apps. This design makes it easier to decompose the solution by workload. You can run the web app and the API in separate App Service plans, so they can be scaled independently. If you don't need that level of scalability at first, you can deploy the apps into the same plan, and move them into separate plans later, if needed.

**Avoid using the App Service backup feature to back up Azure SQL databases.** Instead, use [SQL Database automated backups](#). App Service backup exports the database to a SQL .bacpac file, which costs DTUs.

**Deploy to a staging slot.** Create a deployment slot for staging. Deploy application updates to the staging slot, and verify the deployment before swapping it into production. This reduces the chance of a bad update in production. It also ensures that all instances are warmed up before being swapped into production. Many applications have a significant warmup and cold-start time. For more information, see [Set up staging environments for web apps in Azure App Service](#).

**Create a deployment slot to hold the last-known-good (LKG) deployment.** When you deploy an update to production, move the previous production deployment into the LKG slot. This makes it easier to roll back a bad deployment. If you discover a problem later, you can quickly revert to the LKG version. For more information, see [Basic web application](#).

**Enable diagnostics logging**, including application logging and web server logging. Logging is important for monitoring and diagnostics. See [Enable diagnostics logging for web apps in Azure App Service](#)

**Log to blob storage.** This makes it easier to collect and analyze the data.

**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 [New Relic](#) or [Application Insights](#) 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.

## Application Gateway

**Provision at least two instances.** Deploy Application Gateway with at least two instances. A single instance is a single point of failure. Use two or more instances for redundancy and scalability. In order to qualify for the [SLA](#), you must provision two or more medium or larger instances.

## Cosmos DB

**Replicate the database across regions.** Cosmos DB allows you to associate any number of Azure regions with a Cosmos DB database account. A Cosmos DB database can have one write region and multiple read regions. If there is a failure in the write region, you can read from another replica. The Client SDK handles this automatically. You can also fail over the write region to another region. For more information, see [How to distribute data globally with Azure Cosmos DB](#).

## Event Hubs

**Use checkpoints.** An event consumer should write its current position to persistent storage at some predefined interval. That way, if the consumer experiences a fault (for example, the consumer crashes, or the host fails), then a new instance can resume reading the stream from the last recorded position. For more information, see [Event consumers](#).

**Handle duplicate messages.** If an event consumer fails, message processing is resumed from the last recorded checkpoint. Any messages that were already processed after the last checkpoint will be processed again. Therefore, your message processing logic must be idempotent, or the application must be able to deduplicate messages.

**Handle exceptions.** An event consumer typically processes a batch of messages in a loop. You should handle exceptions within this processing loop to avoid losing an entire batch of messages if a single message causes an exception.

**Use a dead-letter queue.** If processing a message results in a non-transient failure, put the message onto a dead-letter queue, so that you can track the status. Depending on the scenario, you might retry the message later, apply a compensating transaction, or take some other action. Note that Event Hubs does not have any built-in dead-letter queue functionality. You can use Azure Queue Storage or Service Bus to implement a dead-letter queue, or use Azure Functions or some other eventing mechanism.

**Implement disaster recovery by failing over to a secondary Event Hubs namespace.** For more information, see [Azure Event Hubs Geo-disaster recovery](#).

## Redis Cache

**Configure Geo-replication.** Geo-replication provides a mechanism for linking two Premium tier Azure Redis Cache instances. Data written to the primary cache is replicated to a secondary read-only cache. For more information, see [How to configure Geo-replication for Azure Redis Cache](#)

**Configure data persistence.** Redis persistence allows you to persist data stored in Redis. You can also take snapshots and back up the data, which you can load in case of a hardware failure. For more information, see [How to configure data persistence for a Premium Azure Redis Cache](#)

If you are using Redis Cache as a temporary data cache and not as a persistent store, these recommendations may

not apply.

## Search

**Provision more than one replica.** Use at least two replicas for read high-availability, or three for read-write high-availability.

**Configure indexers for multi-region deployments.** If you have a multi-region deployment, consider your options for continuity in indexing.

- If the data source is geo-replicated, you should generally point each indexer of each regional Azure Search service to its local data source replica. However, that approach is not recommended for large datasets stored in Azure SQL Database. The reason is that Azure Search cannot perform incremental indexing from secondary SQL Database replicas, only from primary replicas. Instead, point all indexers to the primary replica. After a failover, point the Azure Search indexers at the new primary replica.
- If the data source is not geo-replicated, point multiple indexers at the same data source, so that Azure Search services in multiple regions continuously and independently index from the data source. For more information, see [Azure Search performance and optimization considerations](#).

## Service Bus

**Use Premium tier for production workloads.** Service Bus Premium Messaging provides dedicated and reserved processing resources, and memory capacity to support predictable performance and throughput. Premium Messaging tier also gives you access to new features that are available only to premium customers at first. You can decide the number of messaging units based on expected workloads.

**Handle duplicate messages.** If a publisher fails immediately after sending a message, or experiences network or system issues, it may erroneously fail to record that the message was delivered, and may send the same message to the system twice. Service Bus can handle this issue by enabling duplicate detection. For more information, see [Duplicate detection](#).

**Handle exceptions.** Messaging APIs generate exceptions when a user error, configuration error, or other error occurs. The client code (senders and receivers) should handle these exceptions in their code. This is especially important in batch processing, where exception handling can be used to avoid losing an entire batch of messages. For more information, see [Service Bus messaging exceptions](#).

**Retry policy.** Service Bus allows you to pick the best retry policy for your applications. The default policy is to allow 9 maximum retry attempts, and wait for 30 seconds but this can be further adjusted. For more information, see [Retry policy – Service Bus](#).

**Use a dead-letter queue.** If a message cannot be processed or delivered to any receiver after multiple retries, it is moved to a dead letter queue. Implement a process to read messages from the dead letter queue, inspect them, and remediate the problem. Depending on the scenario, you might retry the message as-is, make changes and retry, or discard the message. For more information, see [Overview of Service Bus dead-letter queues](#).

**Use Geo-Disaster Recovery.** Geo-disaster recovery ensures that data processing continues to operate in a different region or datacenter if an entire Azure region or datacenter becomes unavailable due to a disaster. For more information, see [Azure Service Bus Geo-disaster recovery](#).

## Storage

**For application data, use read-access geo-redundant storage (RA-GRS).** RA-GRS storage replicates the data to a secondary region, and provides read-only access from the secondary region. If there is a storage outage in the primary region, the application can read the data from the secondary region. For more information, see [Azure Storage replication](#).

**For VM disks, use Managed Disks.** [Managed Disks](#) provide better reliability for VMs in an availability set, because the disks are sufficiently isolated from each other to avoid single points of failure. Also, Managed Disks aren't subject to the IOPS limits of VHDs created in a storage account. For more information, see [Manage the availability of Windows virtual machines in Azure](#).

**For Queue storage, create a backup queue in another region.** For Queue storage, a read-only replica has limited use, because you can't queue or dequeue items. Instead, create a backup queue in a storage account in another region. If there is a storage outage, the application can use the backup queue, until the primary region becomes available again. That way, the application can still process new requests.

## SQL Database

**Use Standard or Premium tier.** These tiers provide a longer point-in-time restore period (35 days). For more information, see [SQL Database options and performance](#).

**Enable SQL Database auditing.** Auditing can be used to diagnose malicious attacks or human error. For more information, see [Get started with SQL database auditing](#).

**Use Active Geo-Replication** Use Active Geo-Replication to create a readable secondary in a different region. If your primary database fails, or simply needs to be taken offline, perform a manual failover to the secondary database. Until you fail over, the secondary database remains read-only. For more information, see [SQL Database Active Geo-Replication](#).

**Use sharding.** Consider using sharding to partition the database horizontally. Sharding can provide fault isolation. For more information, see [Scaling out with Azure SQL Database](#).

**Use point-in-time restore to recover from human error.** Point-in-time restore returns your database to an earlier point in time. For more information, see [Recover an Azure SQL database using automated database backups](#).

**Use geo-restore to recover from a service outage.** Geo-restore restores a database from a geo-redundant backup. For more information, see [Recover an Azure SQL database using automated database backups](#).

## SQL Data Warehouse

**Do not disable geo-backup.** By default, SQL Data Warehouse takes a full backup of your data every 24 hours for disaster recovery. It is not recommended to turn this feature off. For more information, see [Geo-backups](#).

## SQL Server running in a VM

**Replicate the database.** Use SQL Server Always On Availability Groups to replicate the database. Provides high availability if one SQL Server instance fails. For more information, see [Run Windows VMs for an N-tier application](#)

**Back up the database.** If you are already using [Azure Backup](#) to back up your VMs, consider using [Azure Backup for SQL Server workloads using DPM](#). With this approach, there is one backup administrator role for the organization and a unified recovery procedure for VMs and SQL Server. Otherwise, use [SQL Server Managed Backup to Microsoft Azure](#).

## Traffic Manager

**Perform manual failback.** After a Traffic Manager failover, perform manual failback, rather than automatically failing back. Before failing back, verify that all application subsystems are healthy. Otherwise, you can create a situation where the application flips back and forth between data centers. For more information, see [Run VMs in multiple regions for high availability](#).

**Create a health probe endpoint.** Create a custom endpoint that reports on the overall health of the application. This enables Traffic Manager to fail over if any critical path fails, not just the front end. The endpoint should return an HTTP error code if any critical dependency is unhealthy or unreachable. Don't report errors for non-critical services, however. Otherwise, the health probe might trigger failover when it's not needed, creating false positives. For more information, see [Traffic Manager endpoint monitoring and failover](#).

## Virtual Machines

**Avoid running a production workload on a single VM.** A single VM deployment is not resilient to planned or unplanned maintenance. Instead, put multiple VMs in an availability set or [VM scale set](#), with a load balancer in front.

**Specify an availability set when you provision the VM.** Currently, there is no way to add a VM to an availability set after the VM is provisioned. When you add a new VM to an existing availability set, make sure to create a NIC for the VM, and add the NIC to the back-end address pool on the load balancer. Otherwise, the load balancer won't route network traffic to that VM.

**Put each application tier into a separate Availability Set.** In an N-tier application, don't put VMs from different tiers into the same availability set. VMs in an availability set are placed across fault domains (FDs) and update domains (UD). However, to get the redundancy benefit of FDs and UD, every VM in the availability set must be able to handle the same client requests.

**Choose the right VM size based on performance requirements.** When moving an existing workload to Azure, start with the VM size that's the closest match to your on-premises servers. Then measure the performance of your actual workload with respect to CPU, memory, and disk IOPS, and adjust the size if needed. This helps to ensure the application behaves as expected in a cloud environment. Also, if you need multiple NICs, be aware of the NIC limit for each size.

**Use Managed Disks for VHDs.** [Managed Disks](#) provide better reliability for VMs in an availability set, because the disks are sufficiently isolated from each other to avoid single points of failure. Also, Managed Disks aren't subject to the IOPS limits of VHDs created in a storage account. For more information, see [Manage the availability of Windows virtual machines in Azure](#).

**Install applications on a data disk, not the OS disk.** Otherwise, you may reach the disk size limit.

**Use Azure Backup to back up VMs.** Backups protect against accidental data loss. For more information, see [Protect Azure VMs with a recovery services vault](#).

**Enable diagnostic logs,** including basic health metrics, infrastructure logs, and [boot diagnostics](#). Boot diagnostics can help you diagnose a boot failure if your VM gets into a non-bootable state. For more information, see [Overview of Azure Diagnostic Logs](#).

**Use the AzureLogCollector extension.** (Windows VMs only.) This extension aggregates Azure platform logs and uploads them to Azure storage, without the operator remotely logging into the VM. For more information, see [AzureLogCollector Extension](#).

## Virtual Network

**To whitelist or block public IP addresses, add an NSG to the subnet.** Block access from malicious users, or allow access only from users who have privilege to access the application.

**Create a custom health probe.** Load Balancer Health Probes can test either HTTP or TCP. If a VM runs an HTTP server, the HTTP probe is a better indicator of health status than a TCP probe. For an HTTP probe, use a custom endpoint that reports the overall health of the application, including all critical dependencies. For more information, see [Azure Load Balancer overview](#).

**Don't block the health probe.** The Load Balancer Health probe is sent from a known IP address, 168.63.129.16.

Don't block traffic to or from this IP in any firewall policies or network security group (NSG) rules. Blocking the health probe would cause the load balancer to remove the VM from rotation.

**Enable Load Balancer logging.** The logs show how many VMs on the back-end are not receiving network traffic due to failed probe responses. For more information, see [Log analytics for Azure Load Balancer](#).

# Scalability checklist

1/10/2018 • 14 minutes to read • [Edit Online](#)

Scalability is the ability of a system to handle increased load, and is one of the [pillars of software quality](#). Use this checklist to review your application architecture from a scalability 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 impacting current users. You should also implement configuration or auto-detection 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.

**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 Service Management API to adjust autoscaling, and to add custom counters to rules. For more information, see [Auto-scaling guidance](#).

**Offload intensive CPU/IO 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 towards 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 towards 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, and increases latency when accessing the data — which affects 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 impact 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 de-normalizing 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 upon 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](#).

The cloud presents a fundamental shift in the way that enterprises procure and utilize technology resources. In the past, enterprises assumed ownership and responsibility of all levels of technology from infrastructure to software. Now, the cloud offer the potential to transform the way enterprises utilize technology by provisioning and consuming resources as needed.

While the cloud offers nearly unlimited flexibility in terms of design choices, enterprises seek proven and consistent methodology for the adoption of cloud technologies. And, each enterprise has different goals and timelines for cloud adoption, making a one-size-fits-all approach to adoption nearly impossible.

The process of adopting cloud technologies is not a linear process, and this is especially true for large enterprises with many different teams. Some teams may be responsible for a large set of existing workloads and have a requirement to modernize them by adding cloud technologies or migrating them completely. Other teams may have the opportunity to innovate by beginning new development from scratch in the cloud. Yet other teams may not be ready to adopt cloud technologies in production but are ready to learn about and experiment with the cloud.

Each of these different teams requires different approaches to adopting the cloud, but the core knowledge necessary to begin the process is common to all.

## Audience

The audience for the Azure Cloud Adoption Guide includes Enterprise Administrators, Finance, IT operations, IT security and compliance, workload development owners, and workload operations owners.

## How to use the Azure Cloud Adoption Guide

If your enterprise is new to Azure, begin with the [overview](#) in the *getting started* section. This document includes prescriptive guidance for your enterprise's digital transformation, walking through each step of the process.

If your enterprise has experience in Azure and is seeking more in-depth guidance and best practices for specific areas, there are sections for [governance](#) and [infrastructure](#).

[Get Started](#)

# Enterprise Cloud Adoption: Getting started

9/11/2018 • 11 minutes to read • [Edit Online](#)

The **digital transformation** to cloud computing represents a shift from operating on-premises to operating in the cloud. This shift includes new ways of doing business - for example, the digital transformation shifts from capital expenditures for software and datacenter hardware to operating expenditures for usage of cloud resources.

## Digital transformation: process

To be successful in adopting the cloud, an enterprise must prepare its organization, people, and processes to be ready for this digital transformation. Every enterprise's organizational structure is different, so there is no one-size-fits-all approach to organizational readiness. This document outlines the high-level steps your enterprise can take to get ready. Your organization will have to spend time developing a detailed plan to accomplish each of the listed steps.

The high-level process for the digital transformation is:

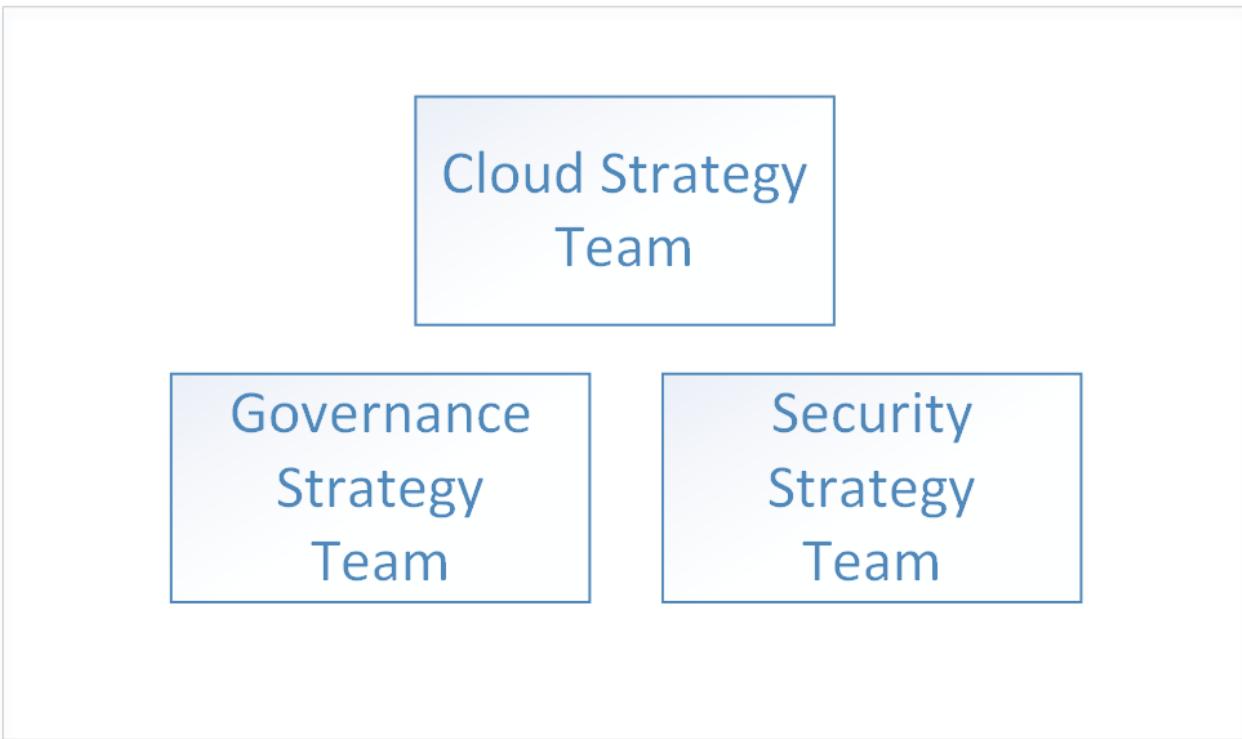
1. Create a cloud strategy team. This team is responsible for leading the digital transformation. It's also important at this stage to form a governance team and a security team for the digital transformation.
2. Members of the cloud strategy team learn what's new and different about cloud technologies.
3. The cloud strategy team prepares the enterprise by building the business case for digital transformation - enumerates all the current gaps in business strategy and determines the high-level solutions to eliminate them.
4. Align high-level solutions with business groups. Identify stakeholders in each business group to own the design and implementation for each solution.
5. Translate existing roles, skills and process to include cloud roles, skills, and process.

### Step 1: create a cloud strategy team

The first step in your enterprise's digital transformation is engaging business leaders from across the organization to create a cloud strategy team (CST). This team consists of business leaders from finance, IT infrastructure, and application groups. These teams can help with the cloud analysis and experimentation phase.

For instance, a Cloud Strategy Team could be driven by the CTO and consist of members of the enterprise architecture team, IT finance, senior technologists from various IT applications groups (HR, finance, and so on), and leaders from the infrastructure, security, and networking teams.

It's also important to form two other high-level teams: a governance team, and a security team. These teams are responsible for designing, implementing, and the ongoing audit of the enterprise's governance and security policies. The governance team requires members that have worked with asset protection, cost management, group policy and related topics. The security team requires members that are well versed in current industry security standards as well as the enterprise's security requirements.

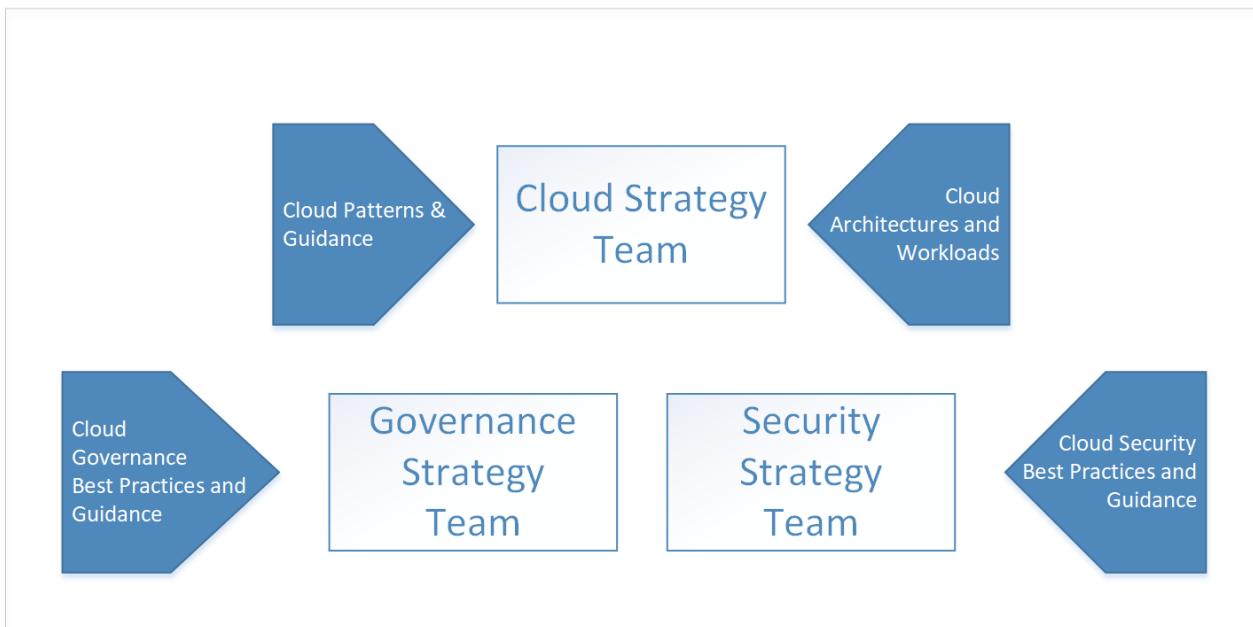


The governance team is responsible for designing and implementing the enterprise's governance model in the cloud, as well as deploying and maintaining the shared infrastructure assets that are part of the digital transformation. These assets include hardware, software, and cloud resources necessary to connect the on-premises network to virtual networking in the cloud.

The security team is responsible for designing and implementing the enterprise's security policy in the cloud, working closely with the governance team. The security team owns the extension of the security boundary of the on-premises network to include virtual networking in the cloud. This may take the form of owning and maintaining the inbound and outbound firewalls on the cloud virtual network as well as ensuring that tools and policy prevent the deployment of unauthorized resources.

## Step 2: learn what's new in the cloud

The next step in your enterprise's digital transformation is for the members of the cloud strategy team to learn about how cloud technology will change the way the enterprise does business. This is preparation and planning for the changes to your business, people, and technology. It's important for the members of the cloud strategy team to understand what's new and different in the cloud as compared to on-premises.



The starting point for understanding the cloud is learning [how Azure works](#) at a high level. Next, learn about the basics of [governance in Azure](#) in preparation for [understanding resource access management](#).

For advanced learning, the governance team should review the concepts and design guides in the governance section of the table of contents. The infrastructure and workloads sections are useful for learning about typical architectures and workloads in the cloud.

## Step 3: identify gaps in business strategy

The next step is for the cloud strategy team to enumerate the business problems that require a digital transformation solution. For example, an enterprise may have an existing on-premises data center with end-of-life hardware that requires replacement. In another example, an enterprise may be experiencing difficulty with time-to-market for new features and services and may be falling behind to competition. These gaps represent the *goals* of your enterprise's digital transformation.

Gaps in business strategy can be classified into the following categories:

| CATEGORY        | DESCRIPTION  |
|-----------------|--|
| Cost management | Represents a gap in the way the Enterprise pays for technology.  |
| Governance      | Represents a gap in the processes used by the Enterprise to protect its assets from improper usage that might result in cost overruns, security issues, or compliance issues.  |
| Compliance      | Represents a gap in the way the enterprise adheres to its own internal processes and policies as well as external laws, regulations, and standards.  |
| Security        | Represents a gap in the way the enterprise protects its technology and data assets from external threats.  |
| Data governance | Represents a gap in the way an enterprise manages its data, especially customer data. For example, new General Data Protection Regulation (GDPR) in the European Union has strict requirements for the protection of customer data that may require new hardware and software. |

Once your enterprise has classified all business strategy gaps into these categories, the next step is to determine a high-level solution for each problem.

The following table illustrates several examples:

| BUSINESS STRATEGY GAP   | CATEGORY        | SOLUTION   |
|---|-----------------|--|
| Services currently hosted on-premises experience issues with availability, resiliency, and scalability during time of peak demand, which is approximately ten percent of usage. Servers in on-premises datacenter are end-of-life. Enterprise IT recommends purchasing new on-premises hardware for datacenter with specifications to handle peak demand. | Cost management | Migrate affected existing on-premises workloads to scalable resources in the cloud, paying for usage only. |
| External data management laws and regulations require the enterprise to adhere to set of standard controls that require encryption of data at rest, requiring new hardware and software.  | Data governance | Move data to Azure storage service encryption for data at rest.  |
| Services hosted in on-premises datacenter been experiencing distributed denial of service (DDoS) attacks on public-facing services. The attacks are difficult to mitigate and require new hardware, software, and security personnel to deal with effectively.  | Security        | Migrate services to Azure, and take advantage of Azure DDoS protection.                                    |

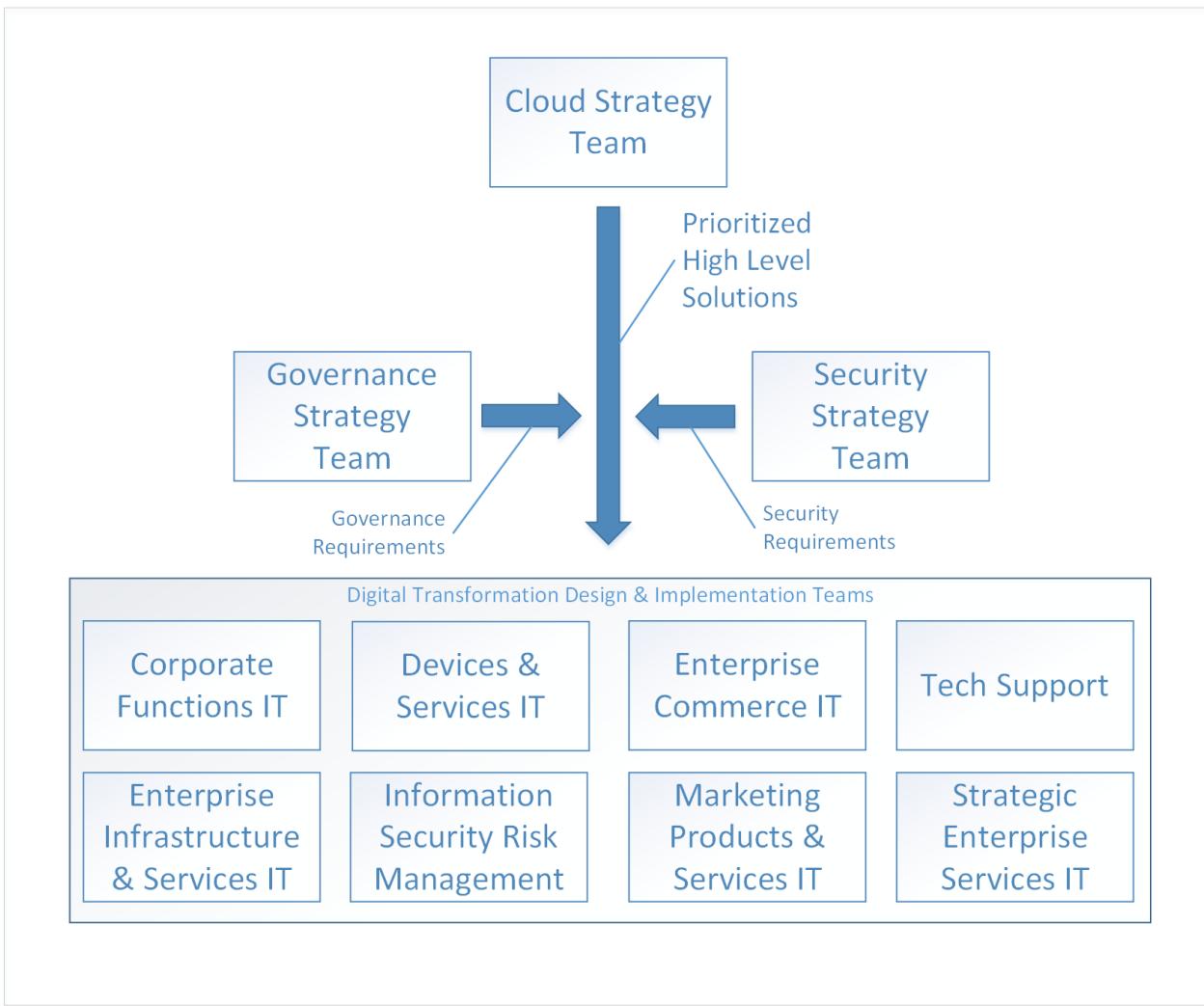
When all of the gaps in business strategy have been enumerated and high-level solutions have been determined, prioritize the list. The list can be prioritized by aligning the business strategy gaps with the enterprise's short and long-term goals in each category. For example, if the enterprise has a short-term goal to reduce IT spend in the next two fiscal quarters, the business gaps in the *cost management* category may be prioritized by the projected cost saving associated with each.

The output of this process is a stack-ranked list of high-level solutions aligned with business categories.

## Step 4: align high-level solutions with business groups to design solutions

Now that the goals of the digital transformation have been enumerated, prioritized, and high-level solutions proposed, the next step is for the cloud strategy team to align each of the high-level solutions with design and implementation teams in each of the business groups.

The teams take the prioritized lists and work through each high-level solution to design each solution. The design process will involve the specification of new infrastructure and new workloads. There may also be changes to the roles of the people and the processes they follow. It's also extremely important at this stage for each of the design teams to include both the governance and security teams for review of each design. Each design must fall within with the policies and procedures defined by the governance and security teams, and these teams must be included in the final sign off of each design.



The design of each solution is a non-trivial task and as designs are created, must be considered in context with other solution designs from other teams. For example, if several of the designs result in a migration of existing on-premises applications and services to the cloud, it may be more efficient to group these together and design an overall migration strategy. In another example, it may not be possible to migrate some existing on-premises applications and services and the solution may be to replace them with either new development or third-party services. In this case, it may be more efficient to group these together and determine the overlap between them to determine if a third-party service can be used for more than one solution.

Once the design of the solution is complete, the team moves on to the implementation phase for each design. The implementation phase for each solution design can be run using standard project management processes.

## Step 5: translate existing roles, skills, and process for the cloud

At each evolutionary phase during the history of the IT industry, the most notable industry changes are often marked by changes in staff roles. During the transition from mainframes to the client/server model, the role of the computer operator largely disappeared, replaced by the system administrator. When the age of virtualization arrived, the requirement for individuals working with physical servers diminished, replaced with a need for virtualization specialists. Similarly, as institutions shift to cloud computing, roles will likely change again. For example, datacenter specialists might be replaced with cloud financial analysts. Even in cases where IT job titles have not changed, the daily work roles have evolved significantly.

IT staff members may feel anxious about their roles and positions as they realize that a different set of skills is needed for the support of cloud solutions. But agile employees who explore and learn new cloud technologies don't need to have that fear. They can lead the adoption of cloud services and help the organization understand and embrace the associated changes.

## Capturing concerns

During the digital transformation, each team should capture any staff concerns as they arise. When capturing concerns, identify the following:

- The type of concern. For example, workers may be resistant to the changes in job duties that accompany the digital transformation.
- The impact of the concern if it is not addressed. For example, resistance to the digital transformation may result in workers being slow to execute the changes necessary.
- The area equipped to address the concern. For example, if workers in the IT department are reluctant to acquire new skills, the IT stakeholder's area is best equipped to address this concern. Identifying the area may be clear for some concerns, and in these cases you may need to escalate to executive leadership.

## Identify gaps

Another aspect of working through the issues with your enterprise's digital transformation is identifying **gaps**. A gap is a role, skill, or process required for your digital transformation that does not currently exist in your enterprise.

Begin by enumerating the new responsibilities that accompany the digital transformation, with an emphasis on new responsibilities and current responsibilities to be retired. Identify the area that is aligned with each responsibility. For new responsibilities, determine how closely aligned it is with the area. Some responsibilities may span several areas, and this represents an opportunity for better alignment that should be captured as a concern. In the case where no area is identified as being responsible, capture this as a gap.

Next, identify the skills necessary to support the responsibility. Determine if your enterprise has existing resources with these skills. If there are no existing resources, determine what training programs or talent acquisition is necessary. Determine the time frame by which the responsibility must be supported to keep your digital transformation on track.

Finally, identify the roles that will execute these skills. Some of your existing workforce will assume parts of the role, and in other cases an entirely new role may be necessary.

## Partner across teams

The skills necessary to fill the gaps in your organization's digital transformation will typically not be confined to a single role, or even a single department. Skills will have relationships and dependencies that can span a single role or multiple roles, and those roles may exist in several departments. For example, a workload owner may require someone in an IT role to provision core resources such as subscriptions and resource groups.

These dependencies represent new processes that your organization implements to manage the workflow between roles. In the above example, there are several different types of process that can support the relationship between the workload owner and the IT role. For example, a workflow tool can be created to manage the process, or, a simple email template can be used.

Track these dependencies and make note of the processes that will support them, and whether or not the process currently exists. For process that require tooling, ensure that the timeline for deploying any tools aligns with the overall digital transformation schedule.

## Next steps

The digital transformation is an iterative process, and with each iteration the teams involved will become more efficient.

[Understand how Azure works](#)

# Enterprise Cloud Adoption: How does Azure work?

9/11/2018 • 2 minutes to read • [Edit Online](#)

Azure is Microsoft's public cloud platform. Azure offers a large collection of services including platform as a service (PaaS), infrastructure as a service (IaaS), database as a service (DBaaS), and many others. But what exactly is Azure, and how does it work?

Azure, like other cloud platforms, relies on a technology known as **virtualization**. Most computer hardware can be emulated in software, because most computer hardware is simply a set of instructions permanently or semi-permanently encoded in silicon. Using an emulation layer that maps software instructions to hardware instructions, virtualized hardware can execute in software as if it were the actual hardware itself.

Essentially, the cloud is a set of physical servers in one or more datacenters that execute virtualized hardware on behalf of customers. So how does the cloud create, start, stop, and delete millions of instances of virtualized hardware for millions of customers simultaneously?

To understand this, let's look at the architecture of the hardware in the datacenter. Within each datacenter is a collection of servers sitting in server racks. Each server rack contains many server **blades** as well as a network switch providing network connectivity and a power distribution unit (PDU) providing power. Racks are sometimes grouped together in larger units known as **clusters**.

Within each rack or cluster, most of the servers are designated to run these virtualized hardware instances on behalf of the user. However, a number of the servers run cloud management software known as a fabric controller. The **fabric controller** is a distributed application with many responsibilities. It allocates services, monitors the health of the server and the services running on it, and heals servers when they fail.

Each instance of the fabric controller is connected to another set of servers running cloud orchestration software, typically known as a **front end**. The front end hosts the web services, RESTful APIs, and internal Azure databases used for all functions the cloud performs.

For example, the front end hosts the services that handle customer requests to allocate Azure resources such as [virtual networks](#), [virtual machines](#), and services like [Cosmos DB](#). First, the front end validates the user and verifies the user is authorized to allocate the requested resources. If so, the front end consults a database to locate a server rack with sufficient capacity, and then instructs the fabric controller on the rack to allocate the resource.

So, very simply, Azure is a huge collection of servers and networking hardware, along with a complex set of distributed applications that orchestrate the configuration and operation of the virtualized hardware and software on those servers. And it is this orchestration that makes Azure so powerful - users are no longer responsible for maintaining and upgrading hardware, Azure does all this behind the scenes.

## Next steps

Now that you understand the internal functioning of Azure, learn about [resource access governance](#).

[Learn about resource governance](#)

# Enterprise Cloud Adoption: What is cloud resource governance?

9/11/2018 • 2 minutes to read • [Edit Online](#)

In [how does Azure work?](#), you learned that Azure is a collection of servers and networking hardware running virtualized hardware and software on behalf of users. Azure enables your organization's development and IT departments to be agile by making it easy to create, read, update, and delete resources as needed.

However, while giving unrestricted resource access to developers can make them very agile, it can also lead to unintended cost consequences. For example, a development team might be approved to deploy a set of resources for testing but forget to delete them when testing is complete. These resources will continue to accrue costs even though their use is no longer approved or necessary.

The solution to this problem is resource access **governance**. Governance refers to the ongoing process of managing, monitoring, and auditing the use of Azure resources to meet the goals and requirements of your organization.

These goals and requirements are unique to each organization so it's not possible to have a one-size-fits-all approach to governance. Rather, Azure implements two primary governance tools, **resource based access control (RBAC)**, and **resource policy**, and it's up to each organization to design their governance model using them.

RBAC defines roles, and roles define the capabilities for a user that is assigned the role. For example, the **owner** role enables all capabilities (create, read, update, and delete) for a resource, while the **reader** role enables only the read capability. Roles can be defined with a broad scope that applies to many resources types, or a narrow scope that applies to a few.

Resource policies define rules for resource creation. For example, a resource policy can limit the SKU of a VM to a particular pre-approved size. Or, a resource policy can enforce the addition of a tag with a cost center when the request is made to create the resource.

When configuring these tools, an important consideration is balancing governance versus organizational agility. That is, the more restrictive your governance policy, the less agile your developers and IT workers become. This is because a restrictive governance policy may require more manual steps, such as requiring a developer to fill out a form or send an email to a person on the governance team to manually create a resource. The governance team has finite capabilities and may become backlogged, resulting in unproductive development teams waiting for their resources to be created and unneeded resources accruing costs while they wait to be deleted.

## Next steps

Now that you understand the concept of cloud resource governance, move on to learn more about [how resource access is managed](#) in Azure in preparation for learning how to design a governance model for a [single team](#) or [multiple teams](#).

[Learn about resource access in Azure](#)

# Enterprise Cloud Adoption: Resource access management in Azure

9/28/2018 • 4 minutes to read • [Edit Online](#)

In [what is resource governance?](#), you learned that governance refers to the ongoing process of managing, monitoring, and auditing the use of Azure resources to meet the goals and requirements of your organization. Before you move on to learn how to design a governance model, it's important to understand the resource access management controls in Azure. The configuration of these resource access management controls forms the basis of your governance model.

Let's begin by taking a closer look at how resources are deployed in Azure.

## What is an Azure resource?

In Azure, the term **resource** refers to an entity managed by Azure. For example, virtual machines, virtual networks, and storage accounts are all referred to as Azure resources.



*Figure 1. A resource.*

## What is an Azure resource group?

Each resource in Azure must belong to a [resource group](#). A resource group is simply a logical construct that groups multiple resources together so they can be managed as a single entity. For example, resources that share a similar lifecycle, such as the resources for an [n-tier application](#) may be created or deleted as a group.

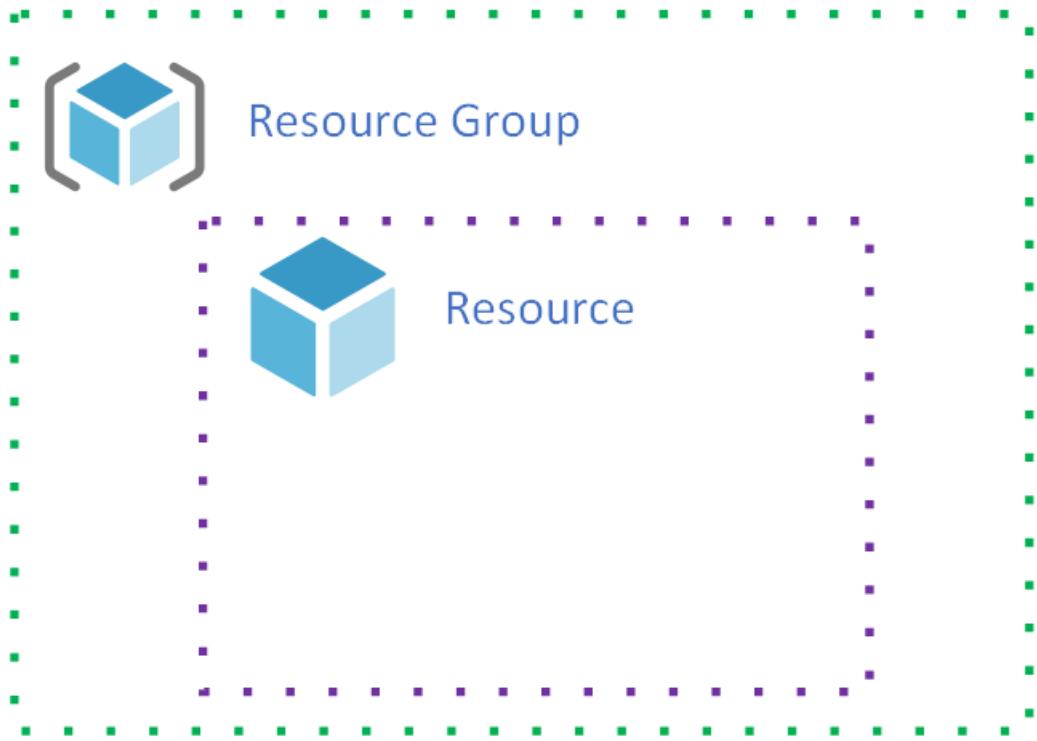


Figure 2. A resource group contains a resource.

Resource groups and the resources they contain are associated with an Azure **subscription**.

## What is an Azure subscription?

An Azure subscription is similar to a resource group in that it's a logical construct that groups together resource groups and their resources. However, an Azure subscription is also associated with the controls used by Azure resource manager. What does this mean? Let's take a closer look at Azure resource manager to learn about the relationship between it and an Azure subscription.

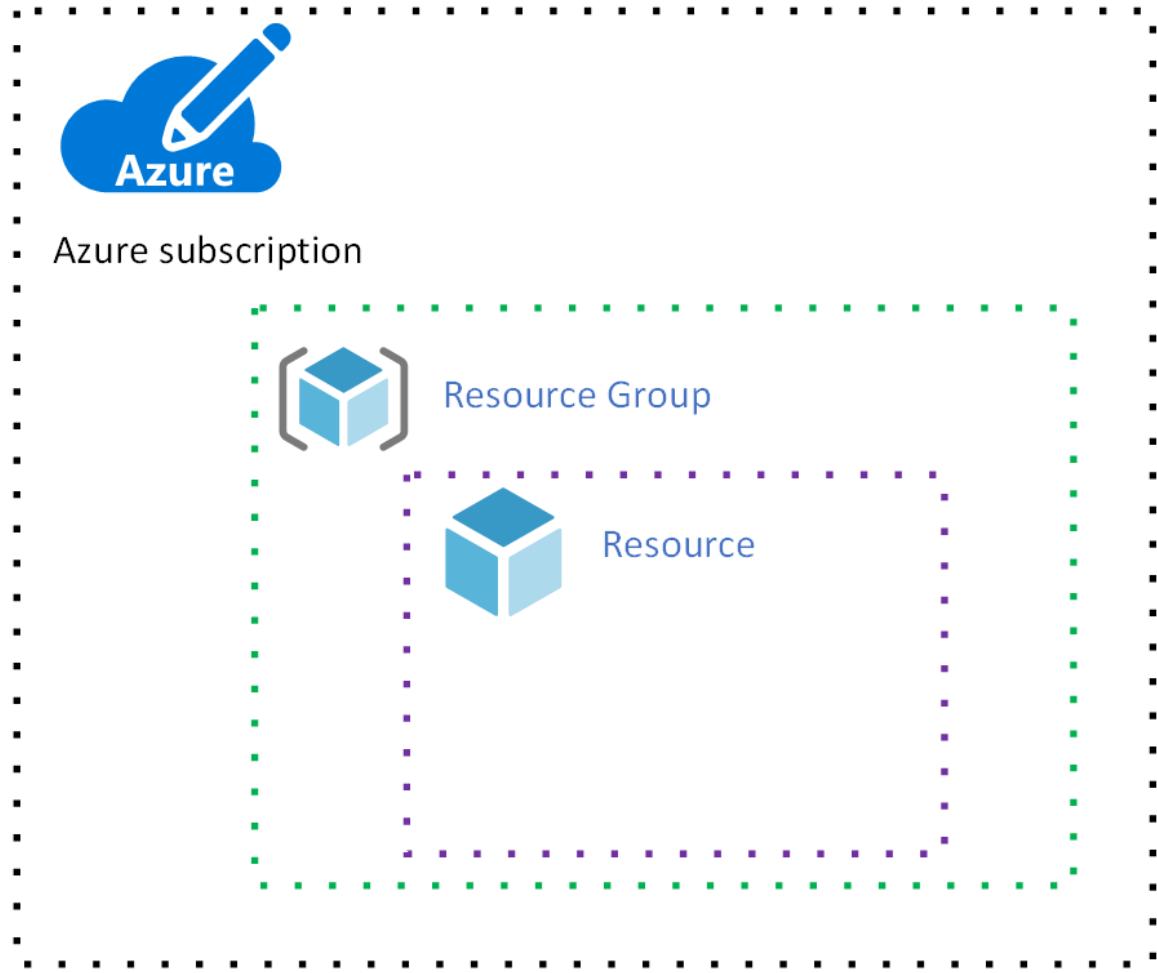


Figure 3. An Azure subscription.

## What is Azure resource manager?

In [how does Azure work?](#) you learned that Azure includes a "front end" with many services that orchestrate all the functions of Azure. One of these services is [Azure resource manager](#), and this service hosts the RESTful API used by clients to manage resources.

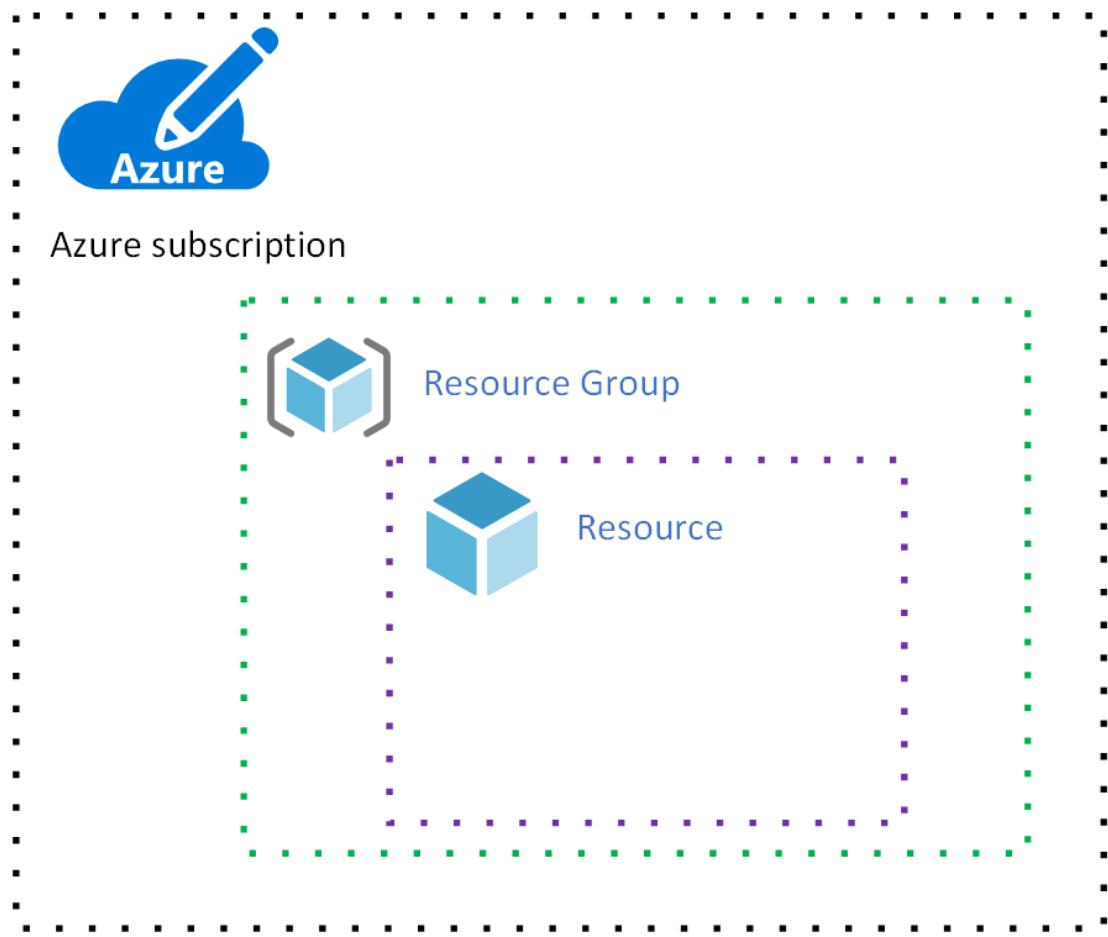
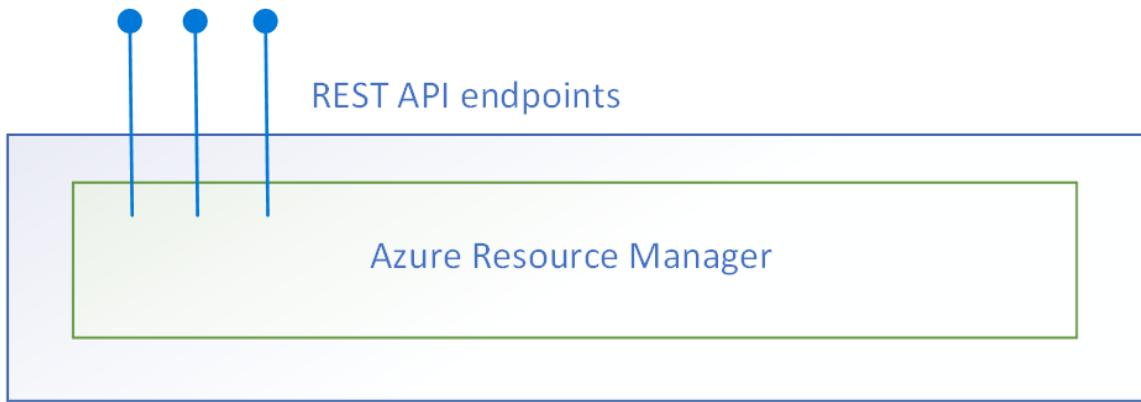
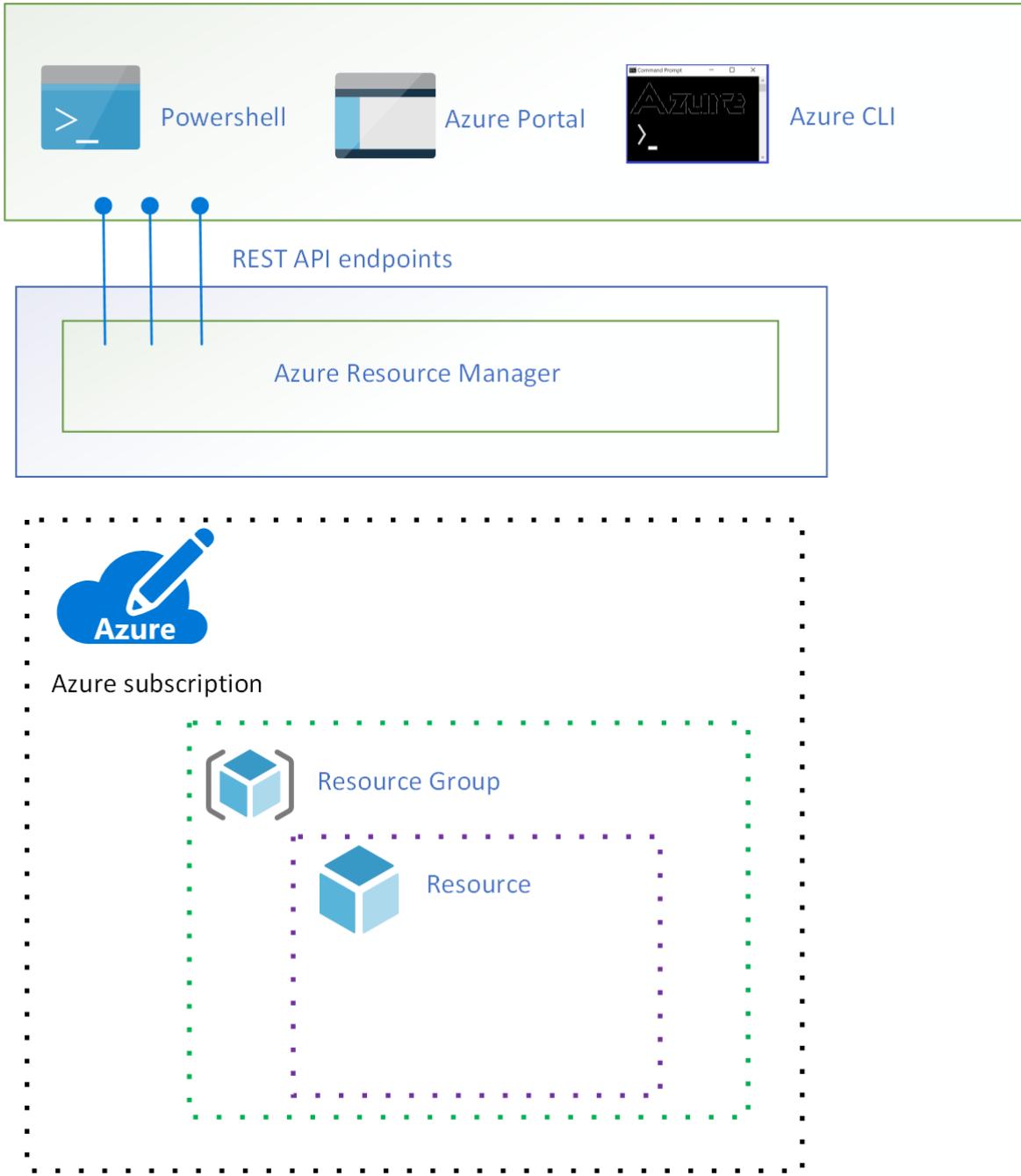


Figure 4. Azure resource manager.

The following figure shows three clients: [Powershell](#), [the Azure portal](#), and the [Azure command line interface \(CLI\)](#):



*Figure 5. Azure clients connect to the Azure resource manager RESTful API.*

While these clients connect to Azure resource manager using the RESTful API, Azure resource manager does not include functionality to manage resources directly. Rather, most resource types in Azure have their own **resource provider**.

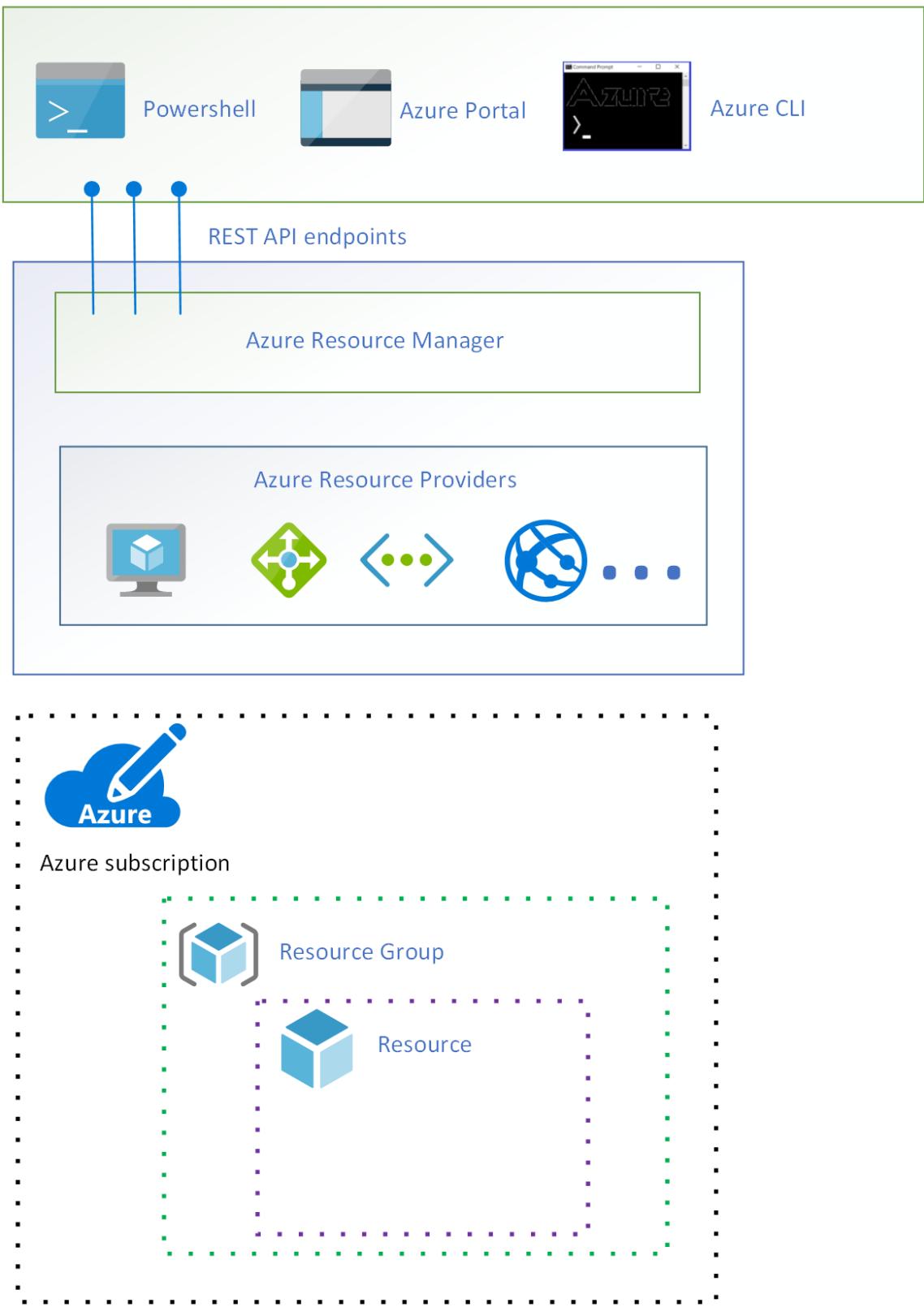


Figure 6. Azure resource providers.

When a client makes a request to manage a specific resource, Azure resource manager connects to the resource provider for that resource type to complete the request. For example, if a client makes a request to manage a virtual machine resource, Azure resource manager connects to the **Microsoft.compute** resource provider.

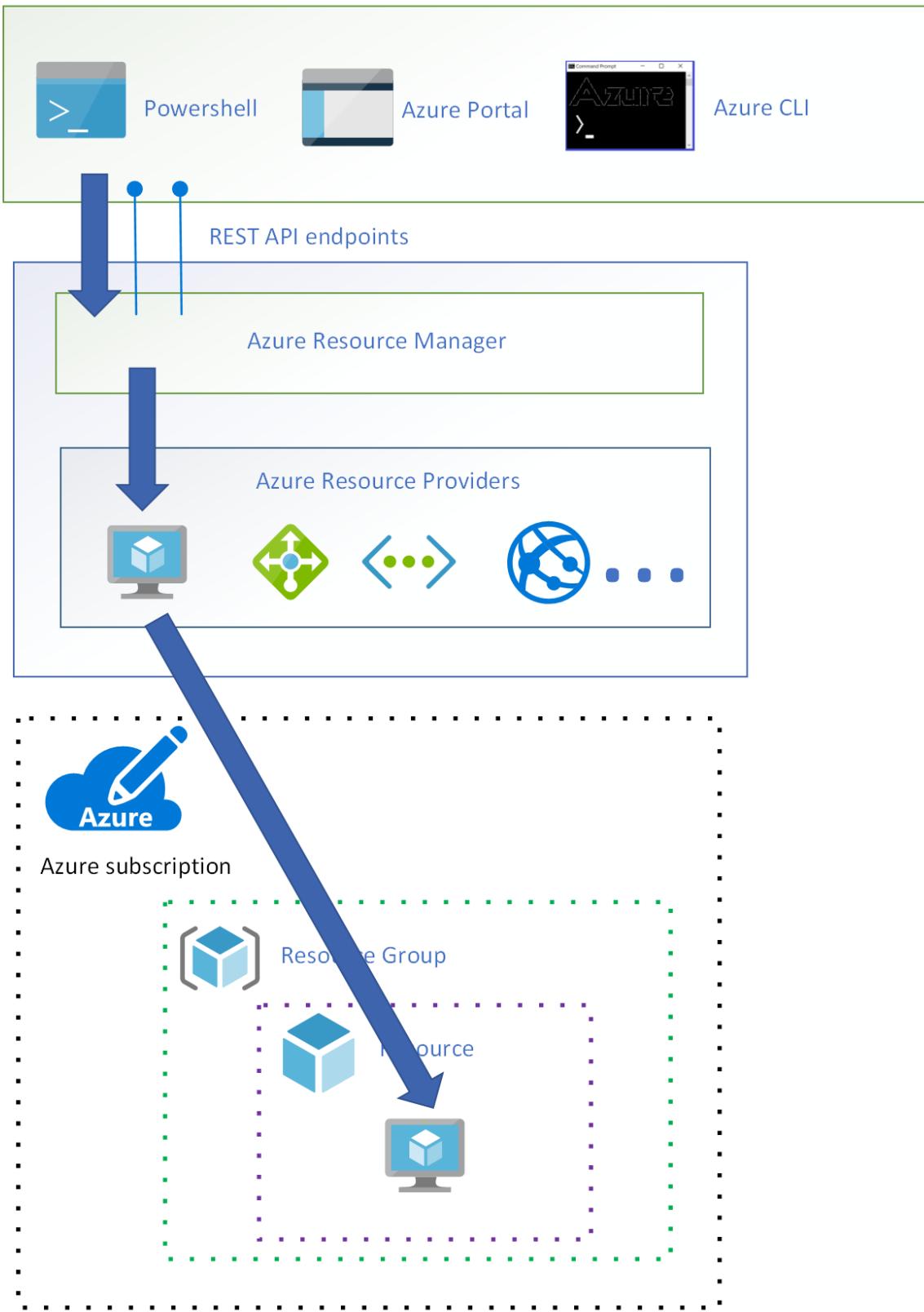


Figure 7. Azure resource manager connects to the **Microsoft.compute** resource provider to manage the resource specified in the client request.

Azure resource manager requires the client to specify an identifier for both the subscription and the resource group in order to manage the virtual machine resource.

Now that you have an understanding of how Azure resource manager works, let's return to our discussion of how an Azure subscription is associated with the controls used by Azure resource manager. Before any resource management request can be executed by Azure resource manager, a set of controls are checked.

The first control is that a request must be made by a validated user, and Azure resource manager has a trusted relationship with [Azure Active Directory \(Azure AD\)](#) to provide user identity functionality.

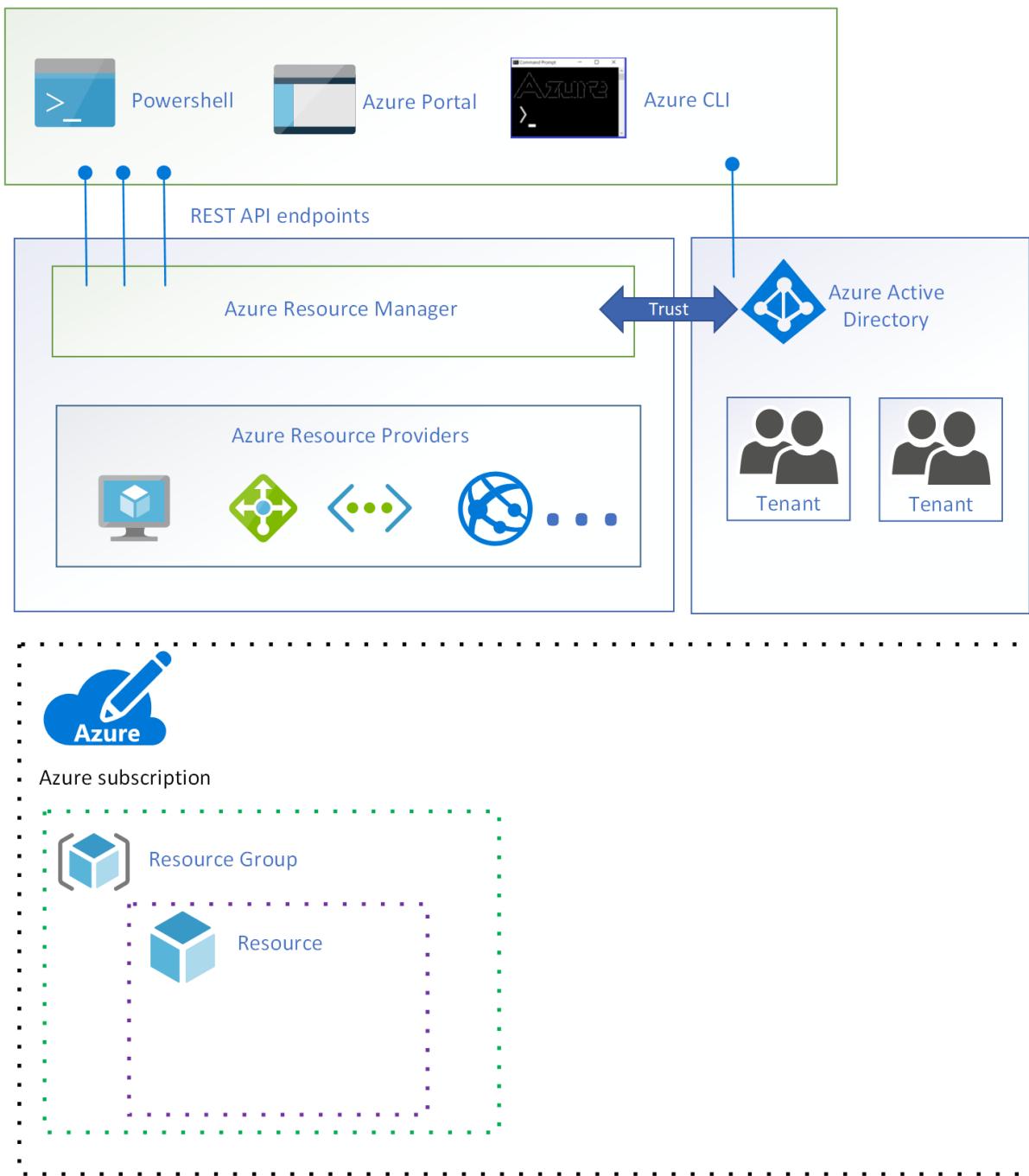


Figure 8. Azure Active Directory.

In Azure AD, users are segmented into **tenants**. A tenant is a logical construct that represents a secure, dedicated instance of Azure AD typically associated with an organization. Each subscription is associated with an Azure AD tenant.

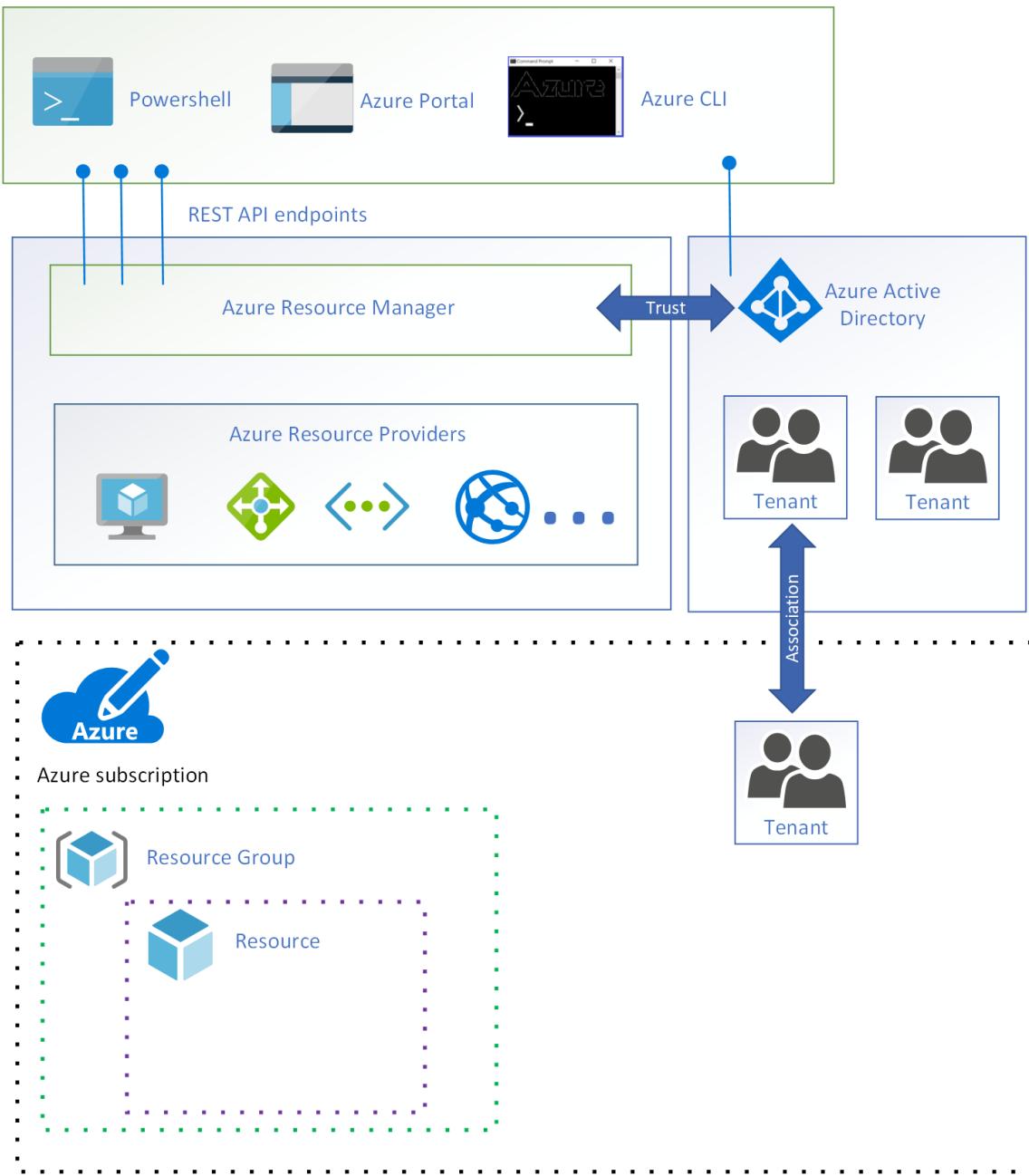


Figure 9. An Azure AD tenant associated with a subscription.

Each client request to manage a resource in a particular subscription requires that the user has an account in the associated Azure AD tenant.

The next control is a check that the user has sufficient permission to make the request. Permissions are assigned to users using [role-based access control \(RBAC\)](#).

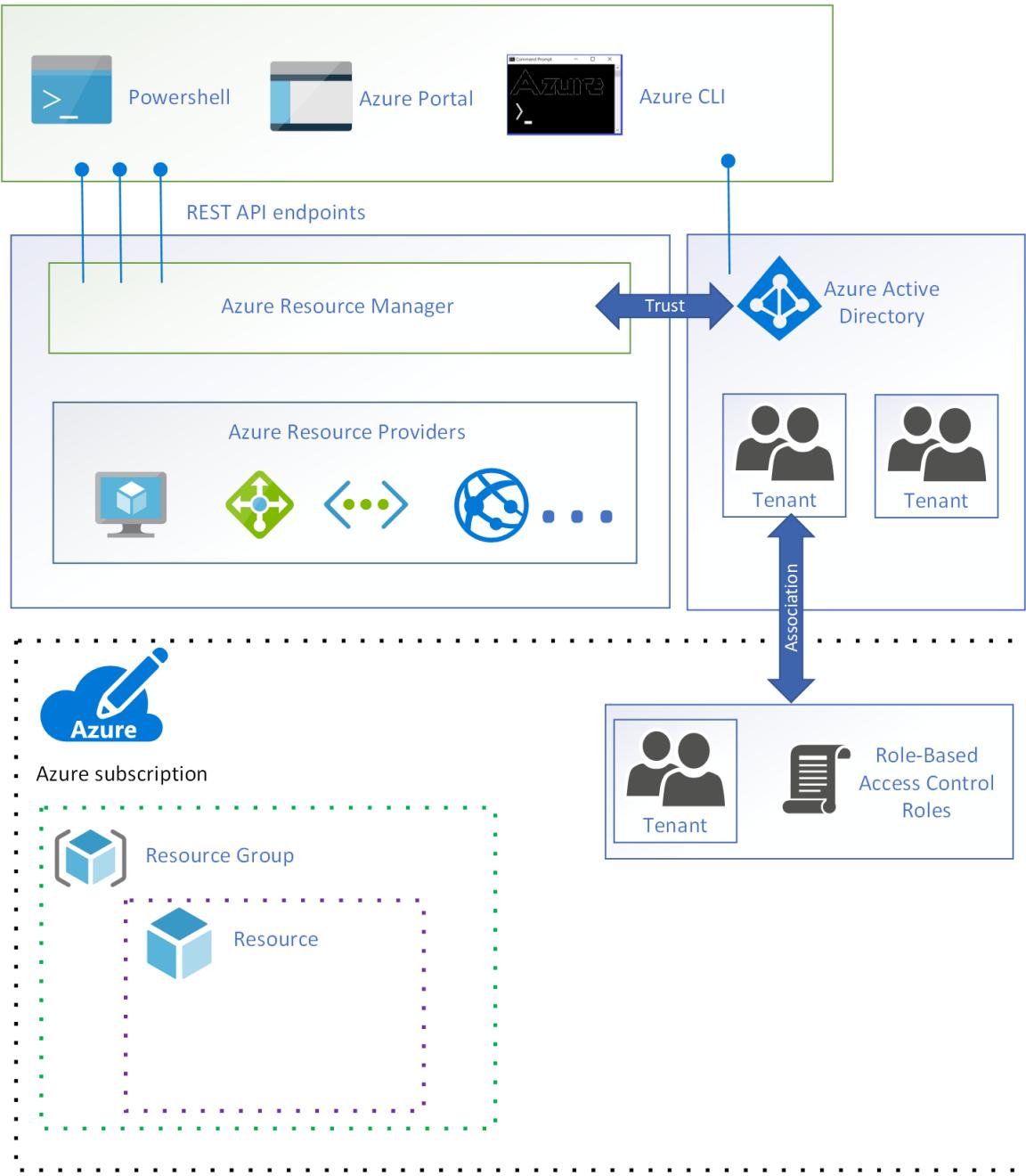


Figure 10. Each user in the tenant is assigned one or more RBAC roles.

An RBAC role specifies a set of permissions a user may take on a specific resource. When the role is assigned to the user, those permissions are applied. For example, the [built-in owner role](#) allows a user to perform any action on a resource.

The next control is a check that the request is allowed under the settings specified for [Azure resource policy](#). Azure resource policies specify the operations allowed for a specific resource. For example, an Azure resource policy can specify that users are only allowed to deploy a specific type of virtual machine.

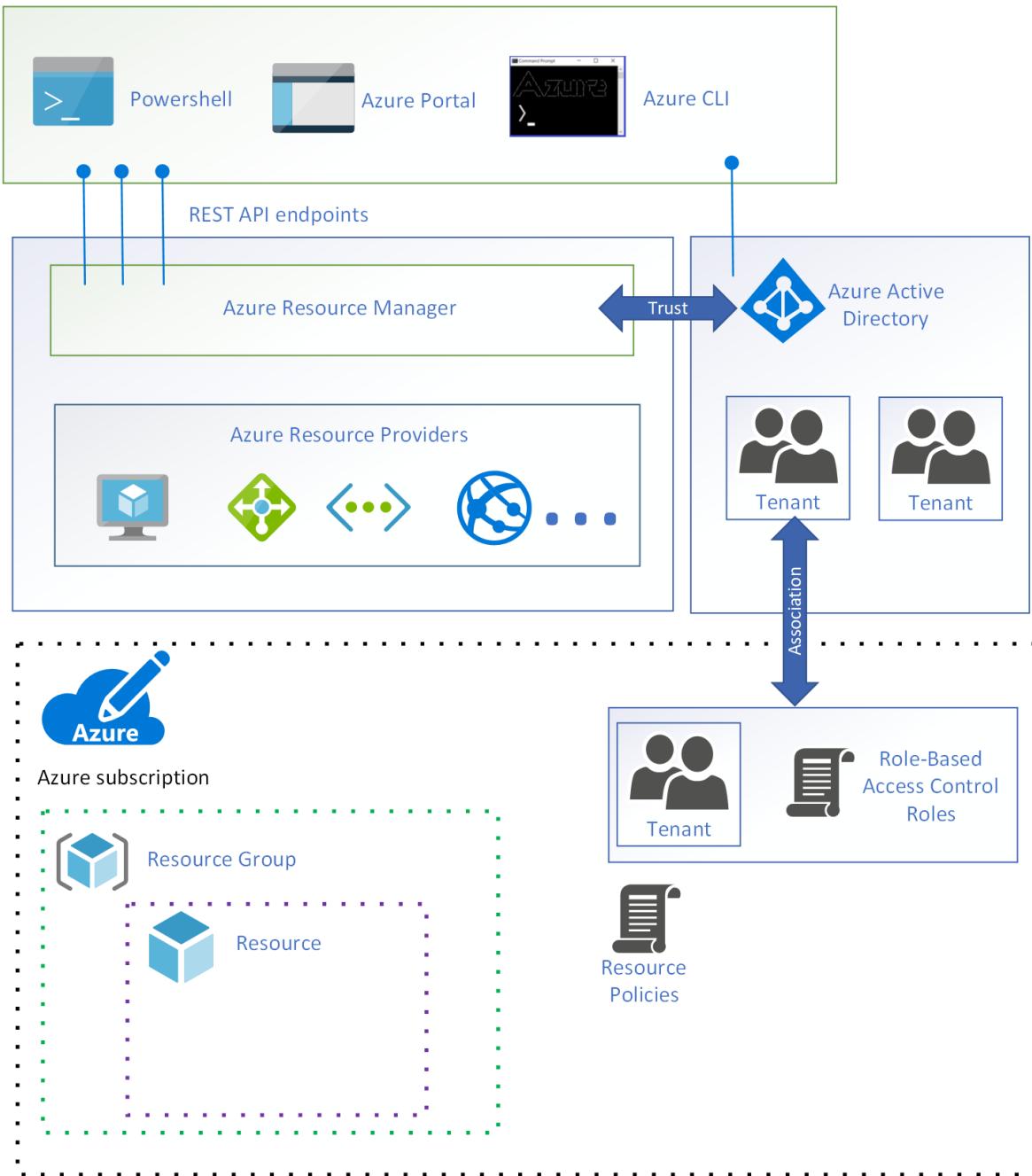


Figure 11. Azure resource policy.

The next control is a check that the request does not exceed an [Azure subscription limit](#). For example, each subscription has a limit of 980 resource groups per subscription. If a request is received to deploy an additional resource group once the limit has been reached, it is denied.

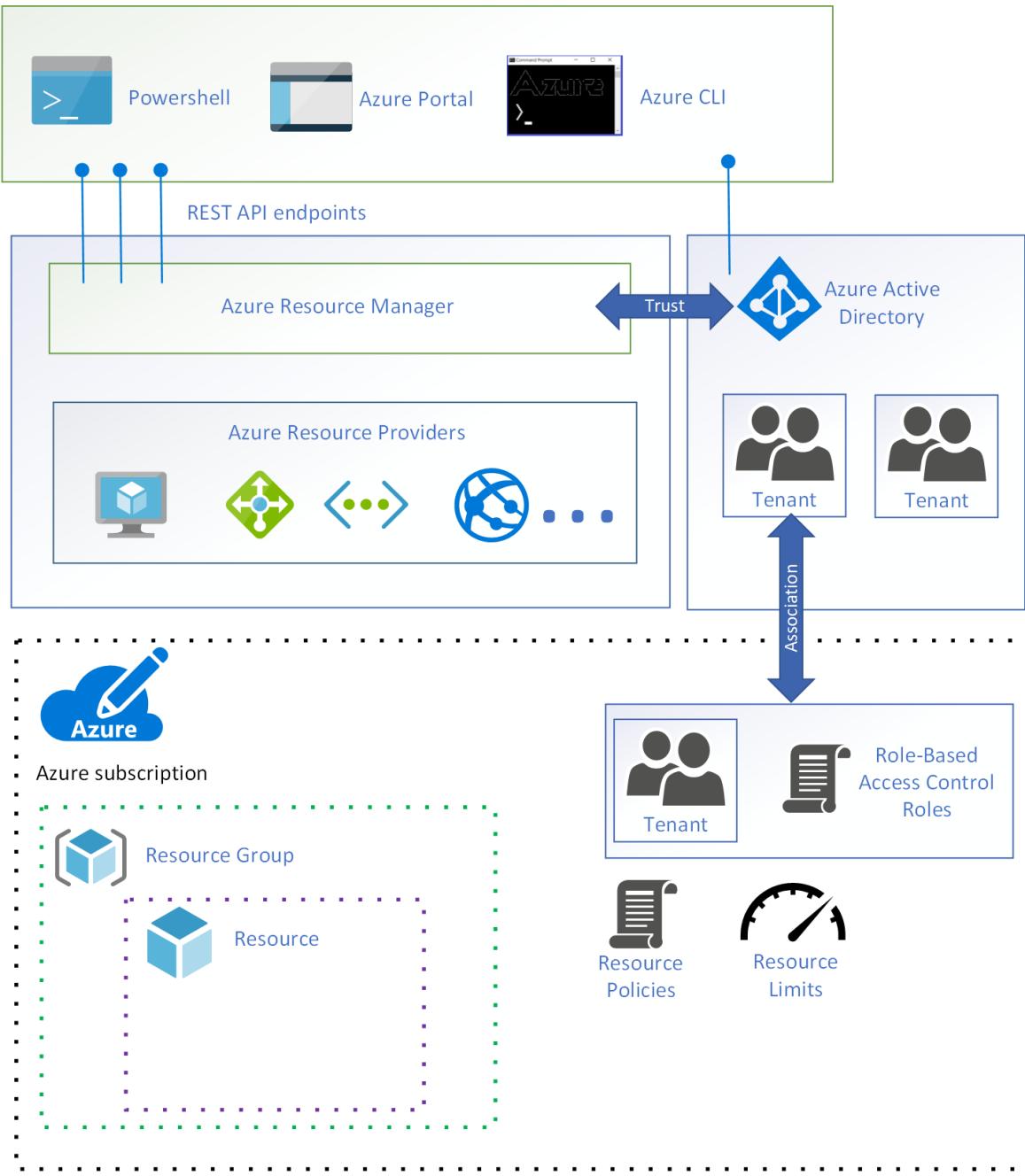


Figure 12. Azure resource limits.

The final control is a check that the request is within the financial commitment associated with the subscription. For example, if the request is to deploy a virtual machine, Azure resource manager verifies that the subscription has sufficient payment information.

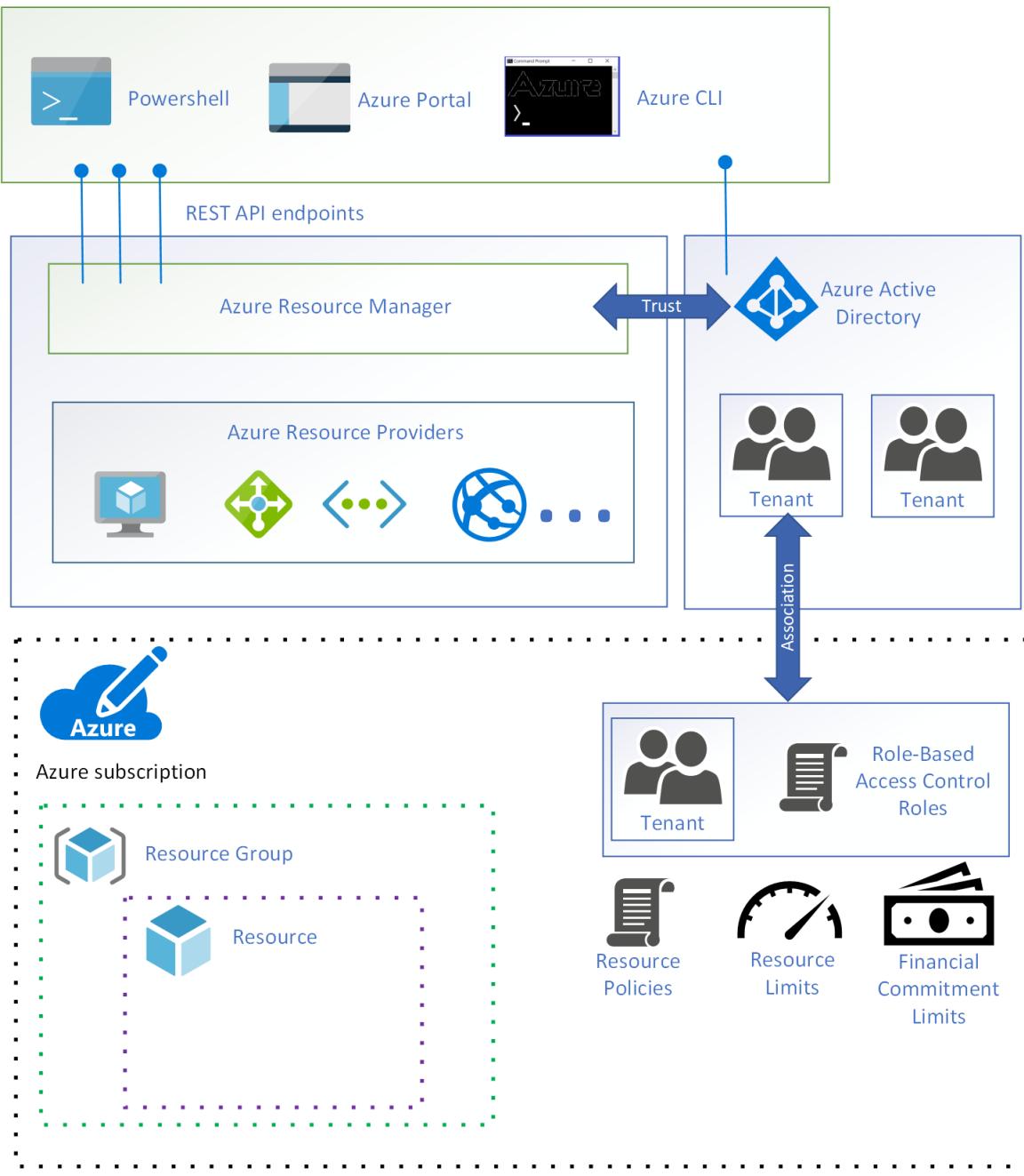


Figure 13. A financial commitment is associated with a subscription.

## Summary

In this article, you learned about how resource access is managed in Azure using Azure resource manager.

## Next steps

Now that you understand how resource access is managed in Azure, move on to learn how to design a governance model [for a single team](#) or [multiple teams](#) using these services.

[An overview of governance](#)

# Enterprise Cloud Adoption: Governance overview

9/11/2018 • 2 minutes to read • [Edit Online](#)

This section of Azure enterprise cloud adoption covers the topic of *governance*. If you are new to the topic of governance in Azure, you can begin with [what is cloud resource governance?](#) and [resource access management in Azure](#) in the [getting started](#) section.

If you are familiar with the concept of governance, this section covers [governance design for a simple workload](#) and [governance design for multiple teams and multiple workloads](#). Both of these documents include an implementation guide.

## Next steps

Once you have learned how to design and implement a governance model in Azure, you can move on to learn how to deploy an [infrastructure](#) to Azure.

[Learn about resource access for a single team](#)

# Enterprise Cloud Adoption: Governance design for a simple workload

9/11/2018 • 6 minutes to read • [Edit Online](#)

The goal of this guidance is to help you learn the process for designing a resource governance model in Azure to support a single team and a simple workload. We'll look at a set of hypothetical governance requirements, then go through several example implementations that satisfy those requirements.

In the foundational adoption stage, our goal is to deploy a simple workload to Azure. This results in the following requirements:

- Identity management for a single **workload owner** who is responsible for deploying and maintaining the simple workload. The workload owner requires permission to create, read, update, and delete resources as well as permission to delegate these rights to other users in the identity management system.
- Manage all resources for the simple workload as a single management unit.

## Licensing Azure

Before we begin designing our governance model, it's important to understand how Azure is licensed. This is because the administrative accounts associated with your Azure license have the highest level of access to all of your Azure resources. These administrative accounts form the basis of your governance model.

### NOTE

If your organization has an existing [Microsoft Enterprise Agreement](#) that does not include Azure, Azure can be added by making an upfront monetary commitment. See [licensing Azure for the enterprise](#) for more information.

When Azure was added to your organization's Enterprise Agreement, your organization was prompted to create an **Azure account**. During the account creation process, an **Azure account owner** was created, as well as an Azure Active Directory (Azure AD) tenant with a **global administrator** account. An Azure AD tenant is a logical construct that represents a secure, dedicated instance of Azure AD.

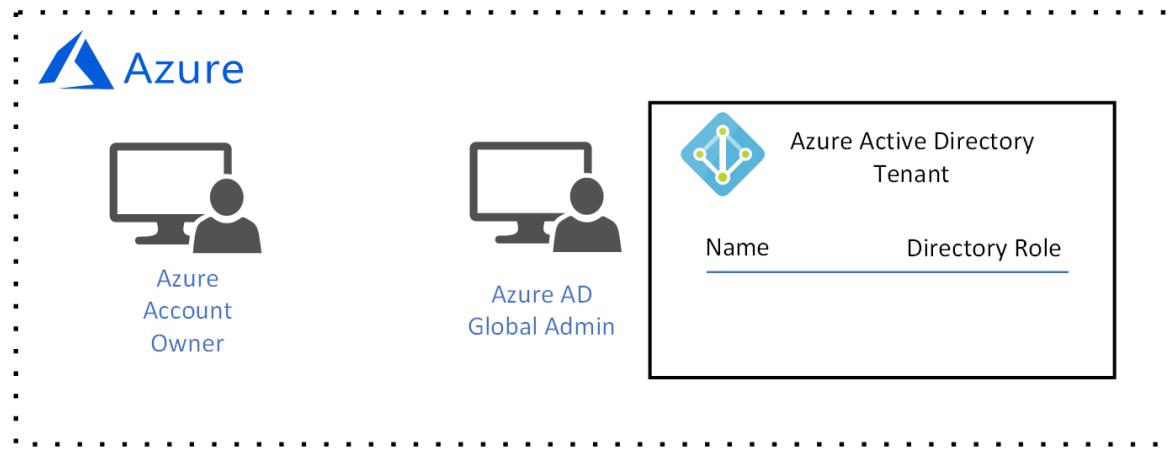


Figure 1. An Azure account with an Account Manager and Azure AD Global Administrator.

## Identity management

Azure only trusts [Azure AD](#) to authenticate users and authorize user access to resources, so Azure AD is our identity management system. The Azure AD global administrator has the highest level of permissions and can perform all actions related to identity, including creating users and assigning permissions.

Our requirement is identity management for a single **workload owner** who is responsible for deploying and maintaining the simple workload. The workload owner requires permission to create, read, update, and delete resources as well as permission to delegate these rights to other users in the identity management system.

Our Azure AD global administrator will create the **workload owner** account for the **workload owner**:

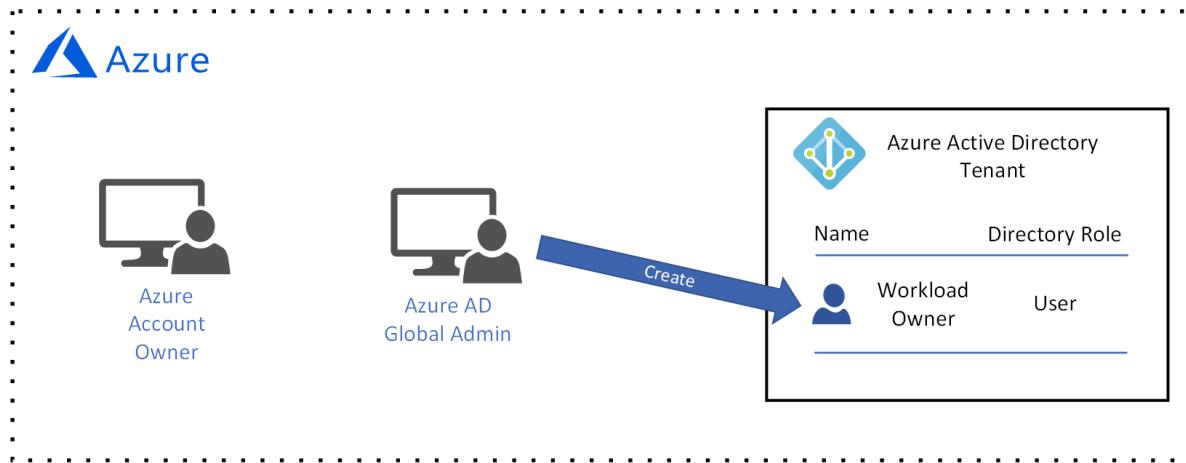


Figure 2. The Azure AD global administrator creates the workload owner user account.

We aren't able to assign resource access permission until this user is added to a **subscription**, so we'll do that in the next two sections.

## Resource management scope

As the number of resources deployed by your organization grows, the complexity of governing those resources grows as well. Azure implements a logical container hierarchy to enable your organization to manage your resources in groups at various levels of granularity, also known as **scope**.

The top level of resource management scope is the **subscription** level. A subscription is created by the Azure **account owner**, who establishes the financial commitment and is responsible for paying for all Azure resources associated with the subscription:

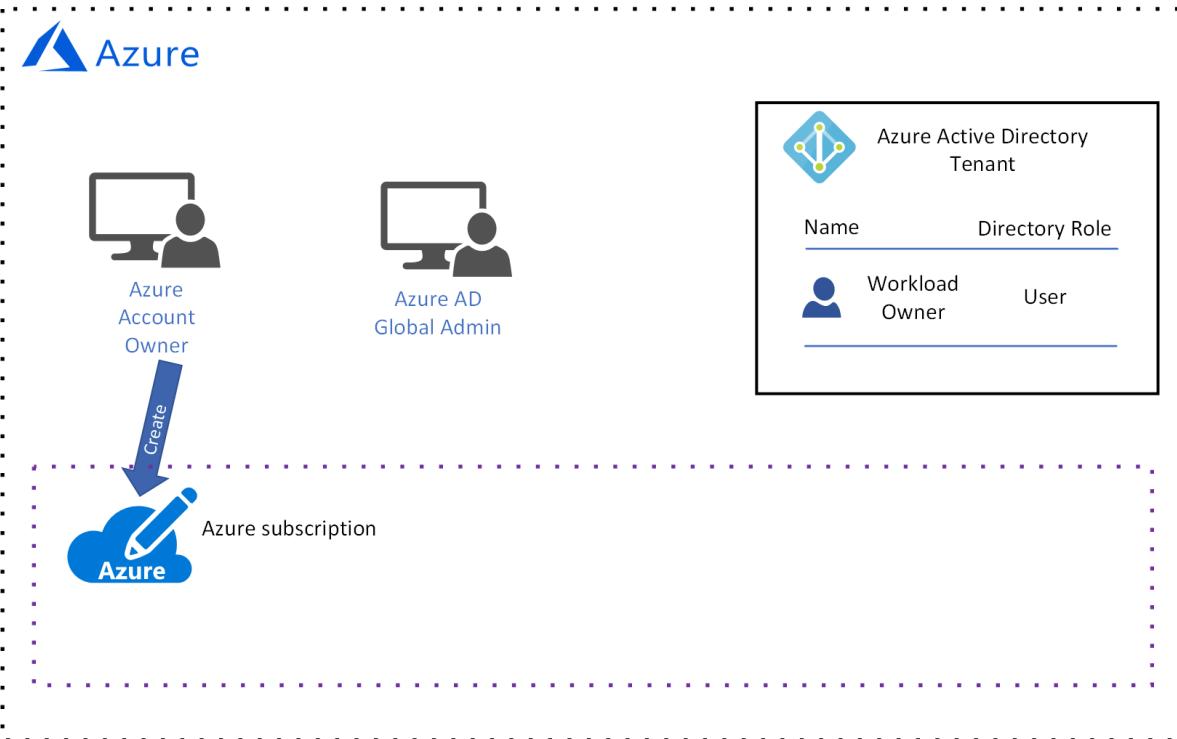


Figure 3. The Azure account owner creates a subscription.

When the subscription is created, the Azure **account owner** associates an Azure AD tenant with the subscription, and this Azure AD tenant is used for authenticating and authorizing users:

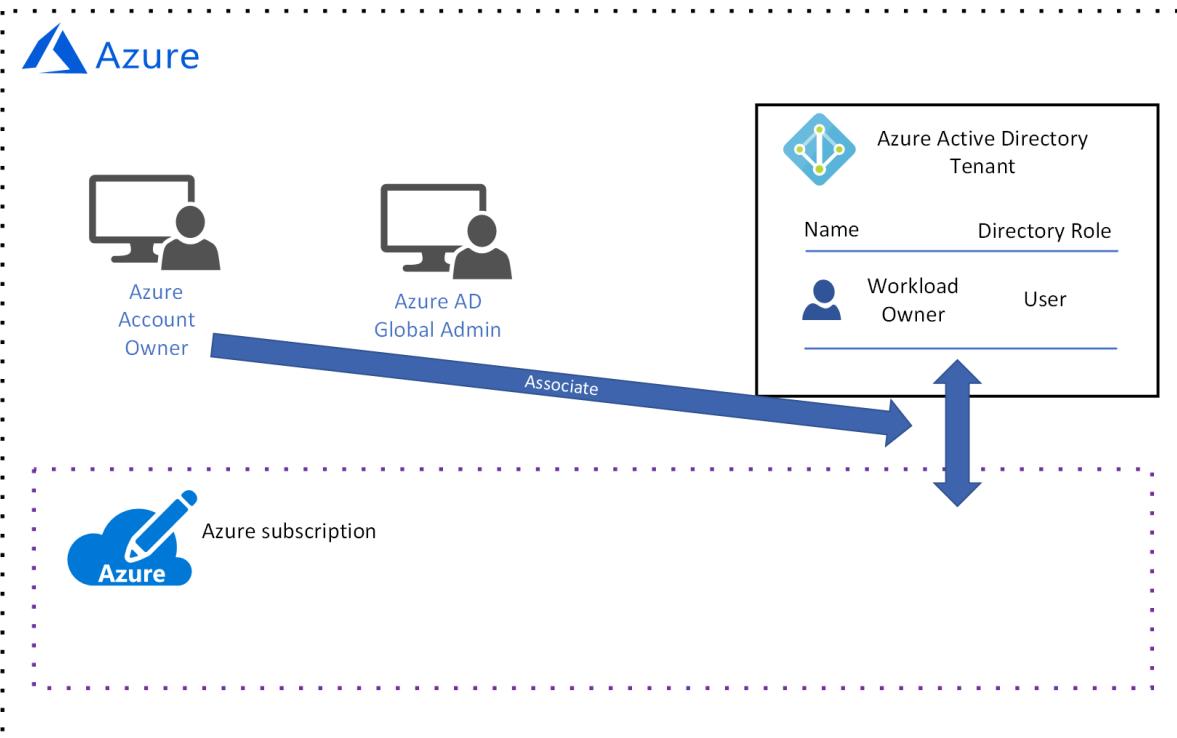


Figure 4. The Azure account owner associates the Azure AD tenant with the subscription.

You may have noticed that there is currently no user associated with the subscription, which means that no one has permission to manage resources. In reality, the **account owner** is the owner of the subscription and has permission to take any action on a resource in the subscription. However, in practical terms the **account owner** is more than likely a finance person in your organization and is not responsible for creating, reading, updating, and deleting resources - those tasks will be performed by the **workload owner**. Therefore, we need to add the

**workload owner** to the subscription and assign permissions.

Since the **account owner** is currently the only user with permission to add the **workload owner** to the subscription, they add the **workload owner** to the subscription:

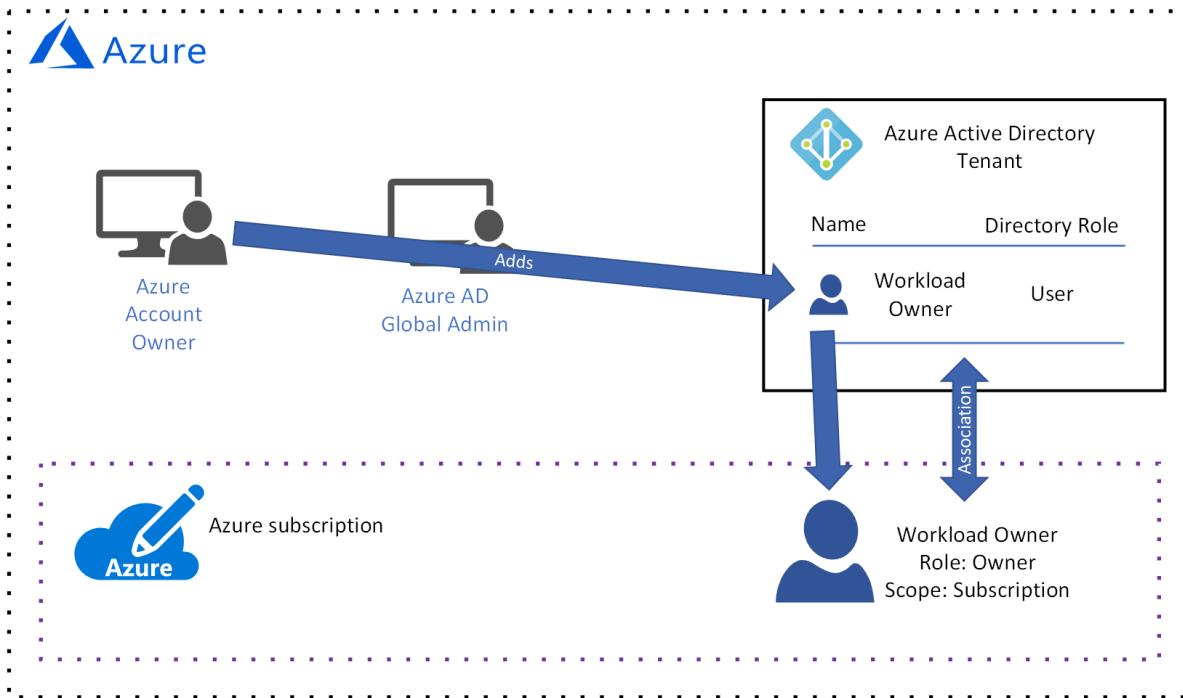


Figure 5. The Azure account owner adds the workload owner to the subscription.

The Azure **account owner** grants permissions to the **workload owner** by assigning a [role-based access control \(RBAC\)](#) role. The RBAC role specifies a set of permissions that the **workload owner** has for an individual resource type or a set of resource types.

Notice that in this example, the **account owner** has assigned the [built-in owner role](#):

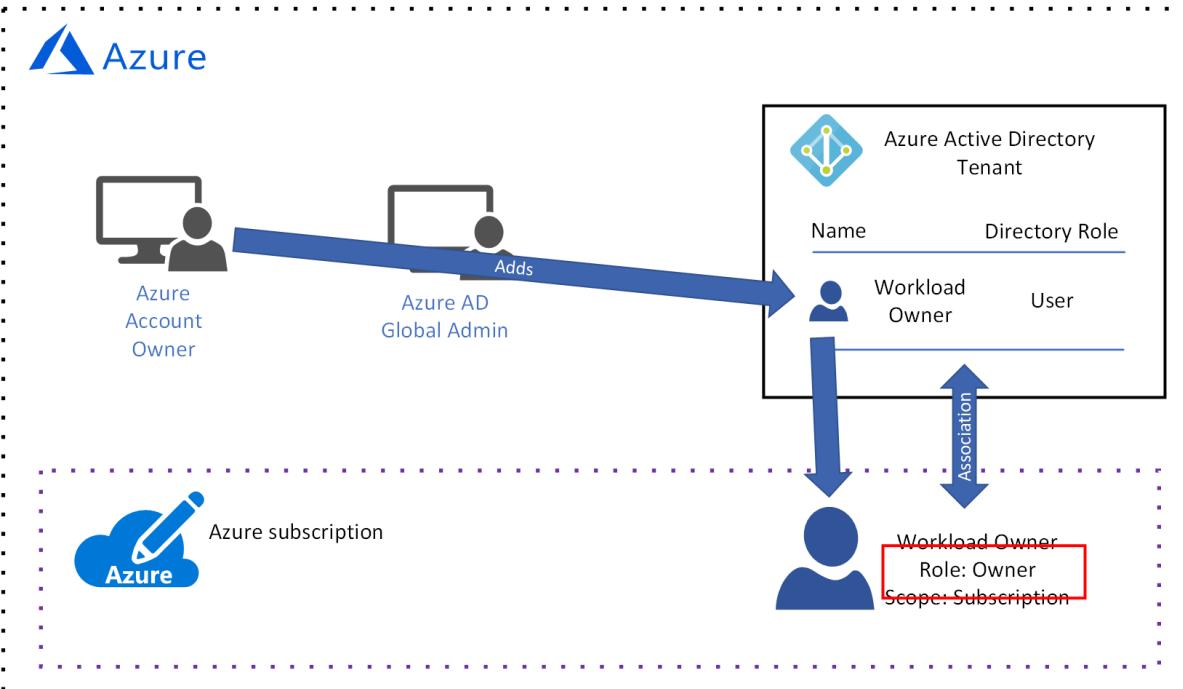


Figure 6. The workload owner was assigned the built-in owner role.

The built-in **owner** role grants all permissions to the **workload owner** at the subscription scope.

#### IMPORTANT

The Azure **account owner** is responsible for the financial commitment associated with the subscription, but the **workload owner** has the same permissions. The **account owner** must trust the **workload owner** to deploy resources that are within the subscription budget.

The next level of management scope is the **resource group** level. A resource group is a logical container for resources. Operations applied at the resource group level apply to all resources in a group. Also, it's important to note that permissions for each user are inherited from the next level up unless they are explicitly changed at that scope.

To illustrate this, let's look at what happens when the **workload owner** creates a resource group:

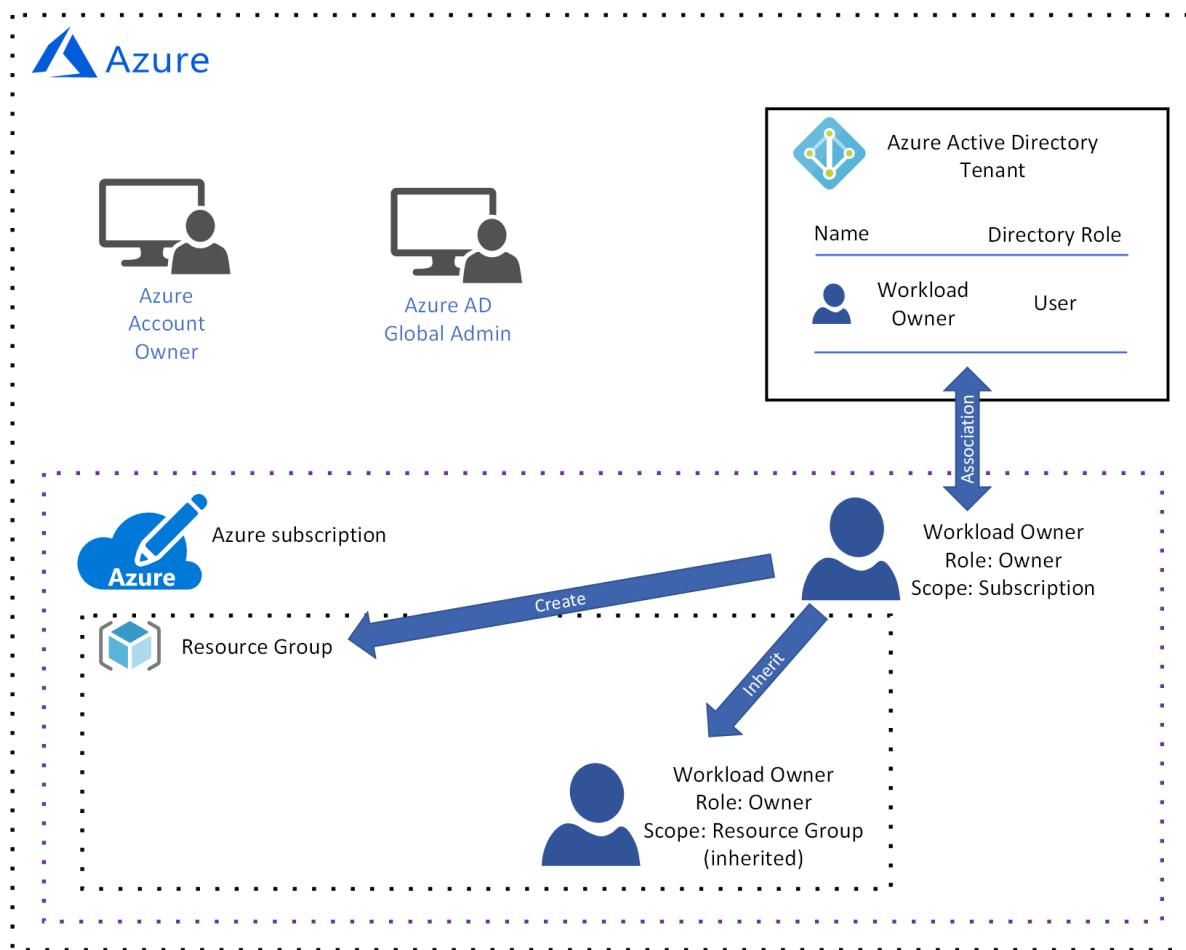


Figure 7. The workload owner creates a resource group and inherits the built-in owner role at the resource group scope.

Again, the built-in **owner** role grants all permissions to the **workload owner** at the resource group scope. As we discussed earlier, this role is inherited from the subscription level. If a different role is assigned to this user at this scope, it applies to this scope only.

The lowest level of management scope is at the **resource** level. Operations applied at the resource level apply only to the resource itself. And once again, permissions at the resource level are inherited from resource group scope. For example, let's look at what happens if the **workload owner** deploys a **virtual network** into the resource group:

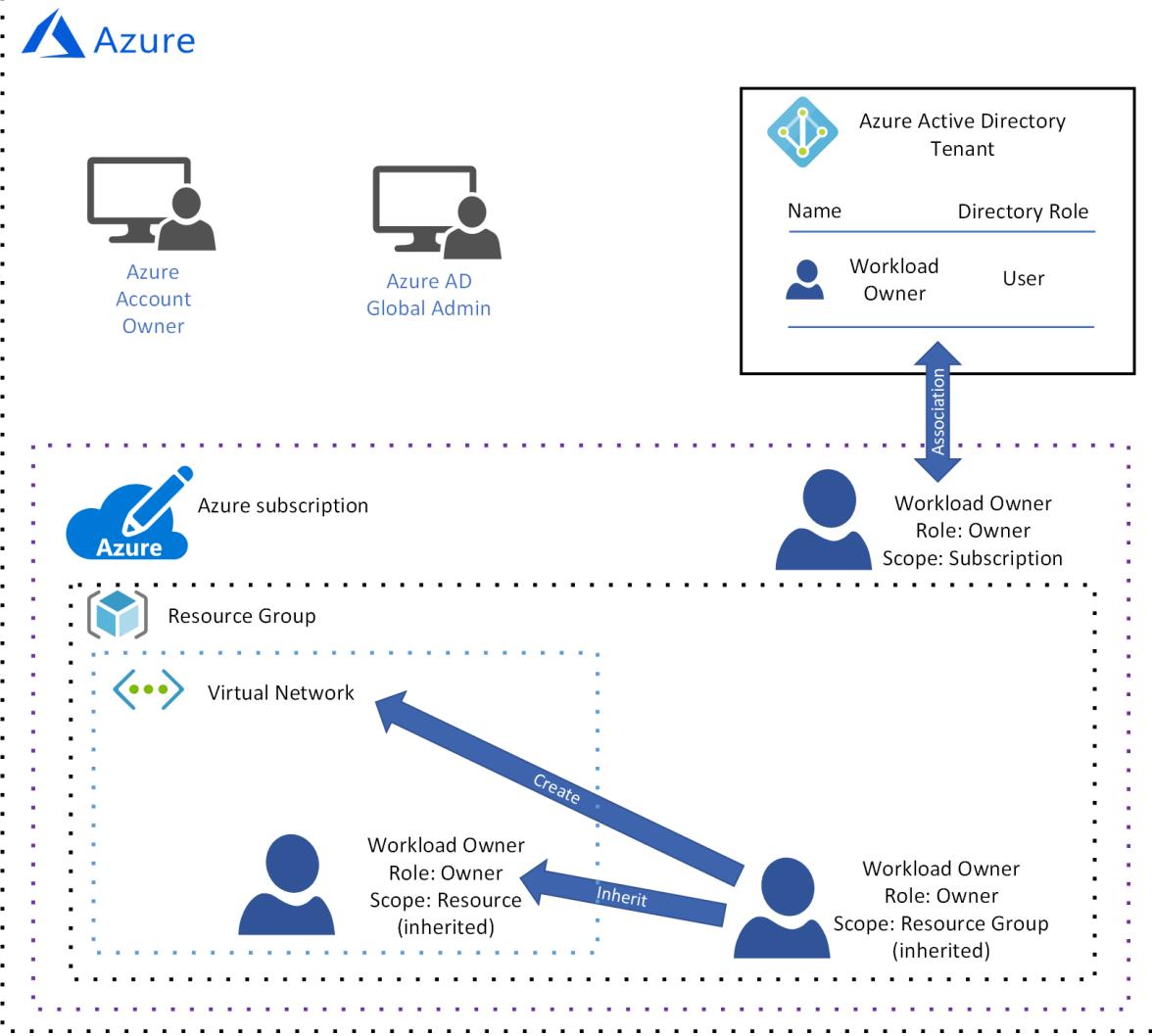


Figure 8. The workload owner creates a resource and inherits the built-in owner role at the resource scope.

The **workload owner** inherits the owner role at the resource scope, which means the workload owner has all permissions for the virtual network.

## Implementing the basic resource access management model

Let's move on to learn how to implement the governance model designed earlier.

To begin, your organization requires an Azure account. If your organization has an existing [Microsoft Enterprise Agreement](#) that does not include Azure, Azure can be added by making an upfront monetary commitment. See [licensing Azure for the enterprise](#) for more information.

When your Azure account is created, you specify a person in your organization to be the Azure **account owner**. An Azure Active Directory (Azure AD) tenant is then created by default. Your Azure **account owner** must [create the user account](#) for the person in your organization who is the **workload owner**.

Next, your Azure **account owner** must [create a subscription](#) and [associate the Azure AD tenant](#) with it.

Finally, now that the subscription is created and your Azure AD tenant is associated with it, you can [add the workload owner to the subscription with the built-in owner role](#).

## Next steps

[Deploy a basic workload to Azure](#)

[Learn about resource access for multiple teams](#)

# Enterprise Cloud Adoption: Governance design for multiple teams

9/11/2018 • 23 minutes to read • [Edit Online](#)

The goal of this guidance is to help you learn the process for designing a resource governance model in Azure to support multiple teams, multiple workloads, and multiple environments. We'll look at a set of hypothetical governance requirements, then go through several example implementations that satisfy those requirements.

The requirements are:

- The enterprise plans to transition new cloud roles and responsibilities to a set of users and therefore requires identity management for multiple teams with different resource access needs in Azure. This identity management system is required to store the identity of the following users:
  1. The individual in your organization responsible for ownership of **subscriptions**.
  2. The individual in your organization responsible for the **shared infrastructure resources** used to connect your on-premises network to an Azure virtual network.
  3. Two individuals in your organization responsible for managing a **workload**.
- Support for multiple **environments**. An environment is a logical grouping of resources, such as virtual machines, virtual networking, and network traffic routing services. These groups of resources have similar management and security requirements and are typically used for a specific purpose such as testing or production. In this example, the requirement is for three environments:
  1. A **shared infrastructure environment** that includes resources shared by workloads in other environments. For example, a virtual network with a gateway subnet that provides connectivity to on-premises.
  2. A **production environment** with the most restrictive security policies. May include internal or external facing workloads.
  3. A **development environment** for proof-of-concept and testing work. This environment has security, compliance, and cost policies that differ from those in the production environment.
- A **permissions model of least privilege** in which users have no permissions by default. The model must support the following:
  - A single trusted user at the subscription scope with permission to assign resource access rights.
  - Each workload owner is denied access to resources by default. Resource access rights are granted explicitly by the single trusted user at the subscription scope.
  - Management access for the shared infrastructure resources limited to the shared infrastructure owner.
  - Management access for each workload restricted to the workload owner.
  - The enterprise does not want to have to manage roles independently in each of the three environments, therefore requires the use of **built-in RBAC roles** only. If the enterprise were to use custom RBAC roles, an additional process is required to synchronize custom roles across the three environments.
- Cost tracking by workload owner name, environment, or both.

## Identity management

Before we can design identity management for your governance model, it's important to understand the four major areas it encompasses:

- Administration: the processes and tools for creating, editing, and deleting user identity.
- Authentication: verifying user identity by validating credentials, such as a user name and password.

- Authorization: determining which resources an authenticated user is allowed to access or what operations they have permission to perform.
- Auditing: periodically reviewing logs and other information to discover security issues related to user identity. This includes reviewing suspicious usage patterns, periodically reviewing user permissions to verify they are accurate, and other functions.

There is only one service trusted by Azure for identity, and that is Azure Active Directory (Azure AD). We'll be adding users to Azure AD and using it for all of the functions listed above. But before we look at how we'll configure Azure AD, it's important to understand the privileged accounts that are used to manage access to these services.

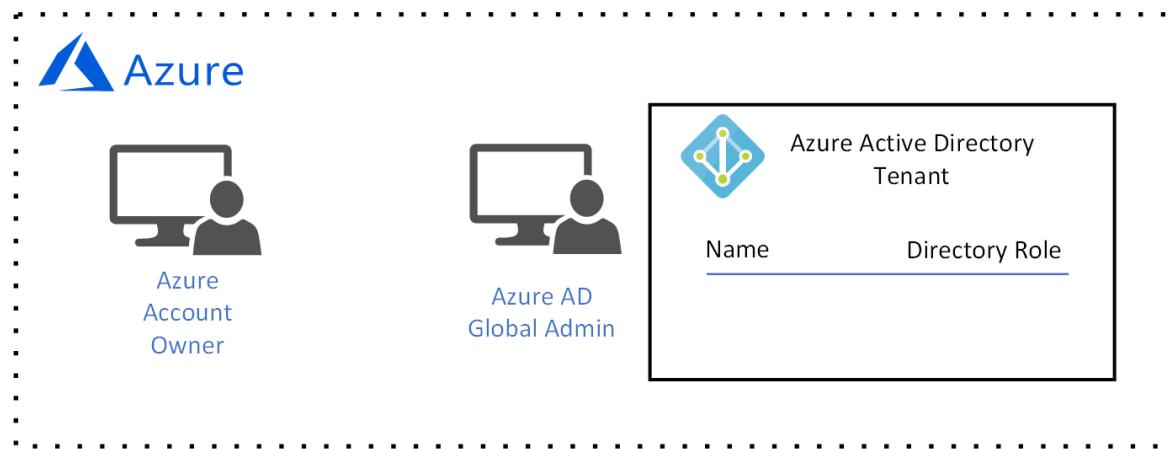
When your organization signed up for an Azure account, at least one Azure **account owner** was assigned. Also, an Azure AD **tenant** was created, unless an existing tenant was already associated with your organization's use of other Microsoft services such as Office 365. A **global administrator** with full permissions on the Azure AD tenant was associated when it was created.

The user identities for both the Azure Account Owner and the Azure AD global administrator are stored in a highly secure identity system that is managed by Microsoft. The Azure Account Owner is authorized to create, update, and delete subscriptions. The Azure AD global administrator is authorized to perform many actions in Azure AD, but for this design guide we'll focus on the creation and deletion of user identity.

#### NOTE

Your organization may already have an existing Azure AD tenant if there's an existing Office 365 or Intune license associated with your account.

The Azure Account Owner has permission to create, update, and delete subscriptions:



*Figure 1. An Azure account with an Account Manager and Azure AD Global Administrator.*

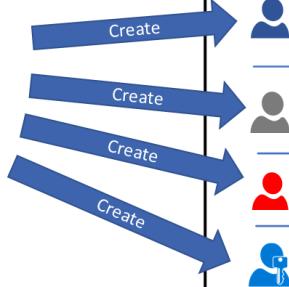
The Azure AD **global administrator** has permission to create user accounts:



Azure  
Account  
Owner



Azure AD  
Global Admin



Azure Active Directory Tenant

| Name                | Directory Role |
|---------------------|----------------|
| App1 Workload Owner | User           |
| App2 Workload Owner | User           |
| Network Operations  | User           |
| Subscription Owner  | User           |

Figure 2. The Azure AD Global Administrator creates the required user accounts in the tenant.

The first two accounts, **App1 Workload Owner** and **App2 Workload Owner** are each associated with an individual in your organization responsible for managing a workload. The **network operations** account is owned by the individual that is responsible for the shared infrastructure resources. Finally, the **subscription owner** account is associated with the individual responsible for ownership of subscriptions.

## Resource access permissions model of least privilege

Now that your identity management system and user accounts have been created, we have to decide how we'll apply role-based access control (RBAC) roles to each account to support a permissions model of least privilege.

There's another requirement stating the resources associated with each workload be isolated from one another such that no one workload owner has management access to any other workload they do not own. There's also a requirement to implement this model using only [built-in roles for Azure RBAC](#).

Each RBAC role is applied at one of three scopes in Azure: **subscription**, **resource group**, then an individual **resource**. Roles are inherited at lower scopes. For example, if a user is assigned the [built-in owner role](#) at the subscription level, that role is also assigned to that user at the resource group and individual resource level unless it's overridden.

Therefore, to create a model of least privilege access we have to decide the actions a particular type of user is allowed to take at each of these three scopes. For example, the requirement is for a workload owner to have permission to manage access to only the resources associated with their workload and no others. If we were to assign the [built-in owner role](#) at the subscription scope, each workload owner would have management access to all workloads.

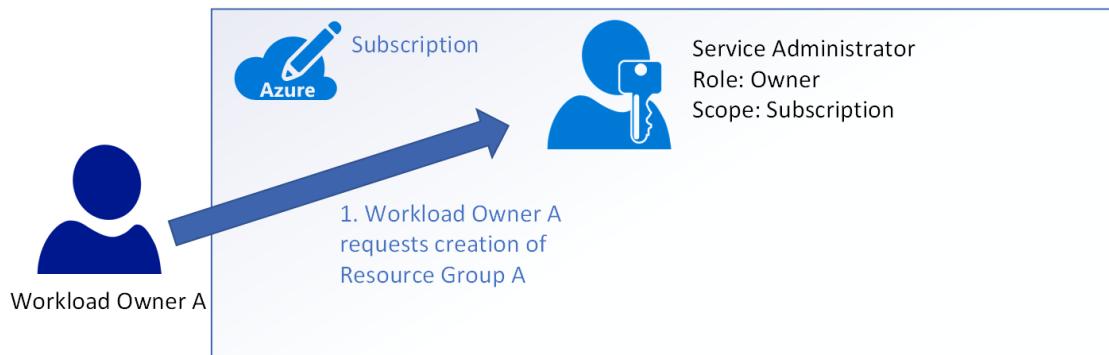
Let's take a look at two example permission models to understand this concept a little better. In the first example, the model trusts only the service administrator to create resource groups. In the second example, the model assigns the built-in owner role to each workload owner at the subscription scope.

In both examples, we have a subscription service administrator that is assigned the [built-in owner role](#) at the subscription scope. Recall that the [built-in owner role](#) grants all permissions including the management of access to resources.

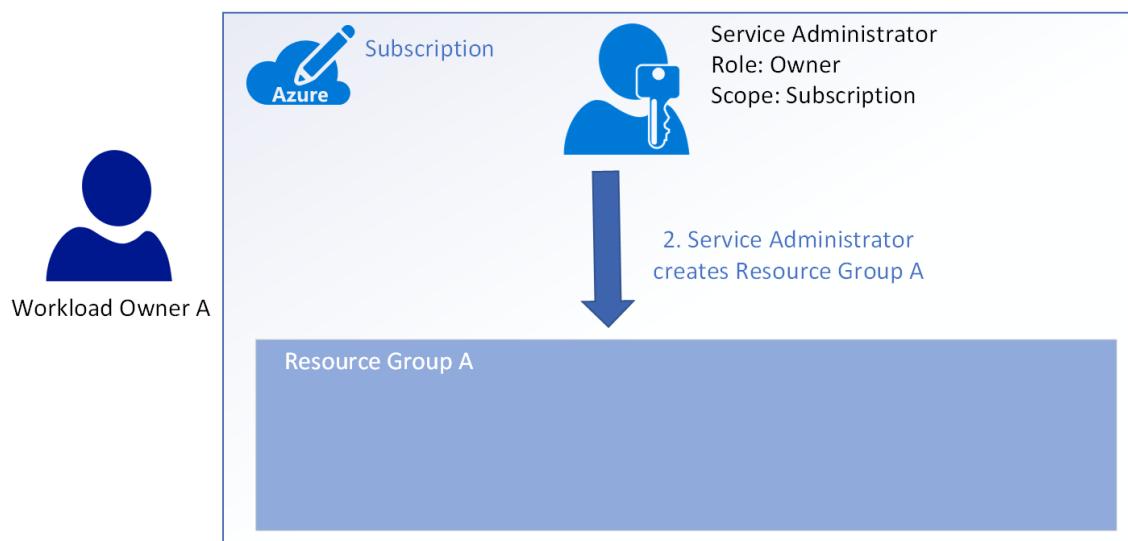


Figure 3. A subscription with a service administrator assigned the built-in owner role.

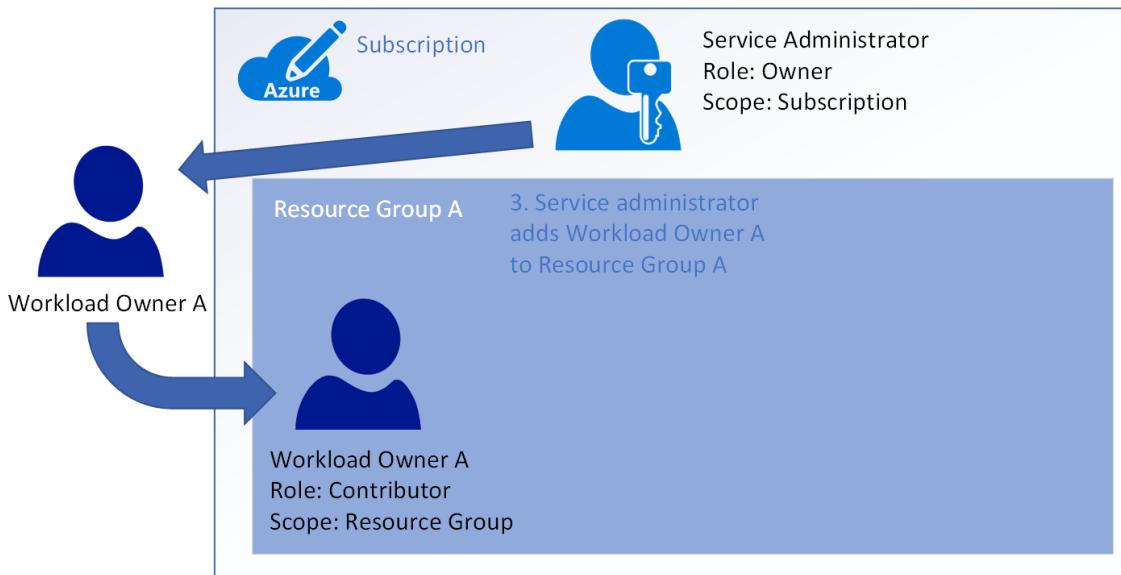
1. In the first example, we have **workload owner A** with no permissions at the subscription scope - they have no resource access management rights by default. This user wants to deploy and manage the resources for their workload. They must contact the **service administrator** to request creation of a resource group.



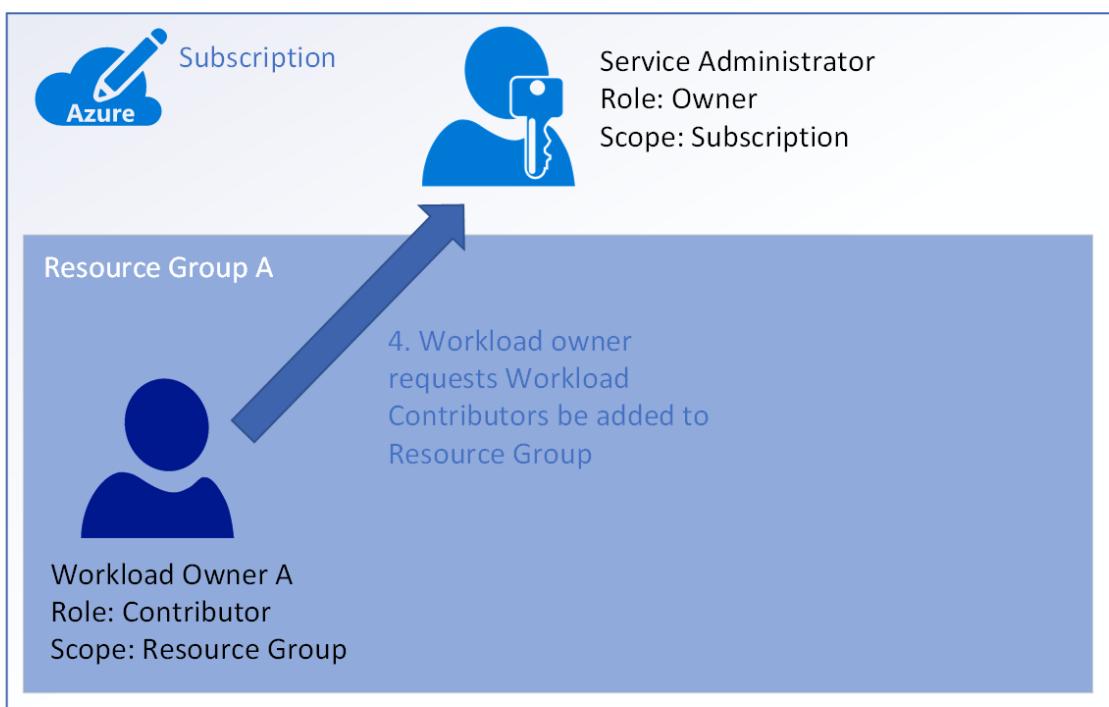
2. The **service administrator** reviews their request and creates **resource group A**. At this point, **workload owner A** still doesn't have permission to do anything.



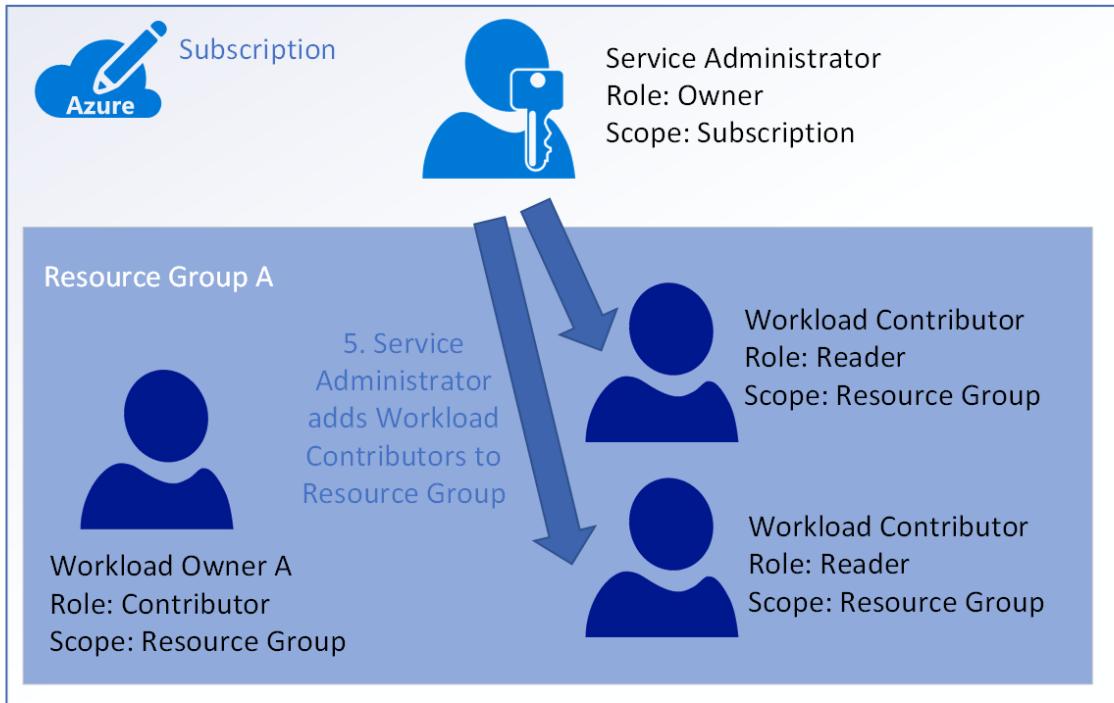
3. The **service administrator** adds **workload owner A** to **resource group A** and assigns the **built-in contributor role**. The contributor role grants all permissions on **resource group A** except managing access permission.



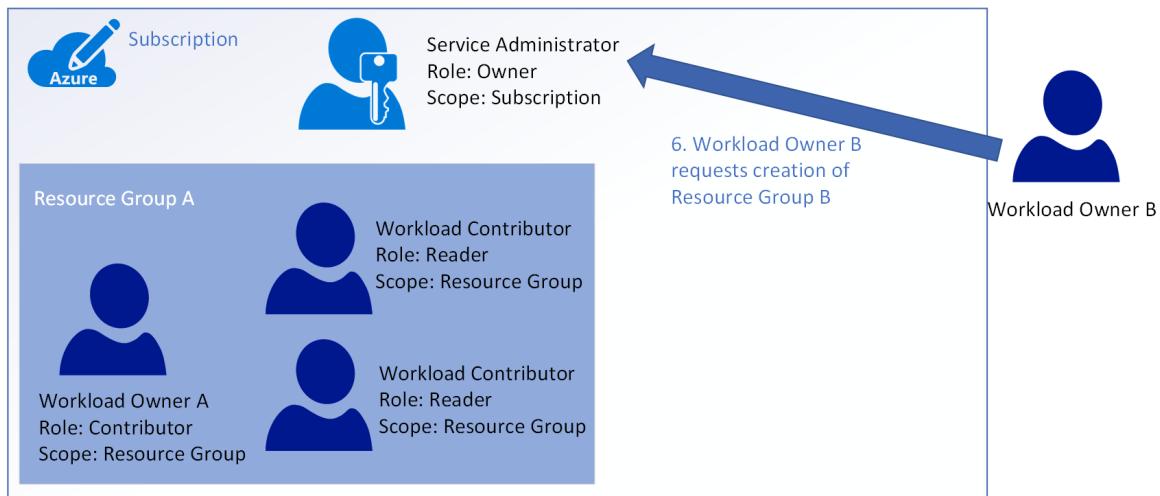
- Let's assume that **workload owner A** has a requirement for a pair of team members to view the CPU and network traffic monitoring data as part of capacity planning for the workload. Because **workload owner A** is assigned the contributor role, they do not have permission to add a user to **resource group A**. They must send this request to the **service administrator**.



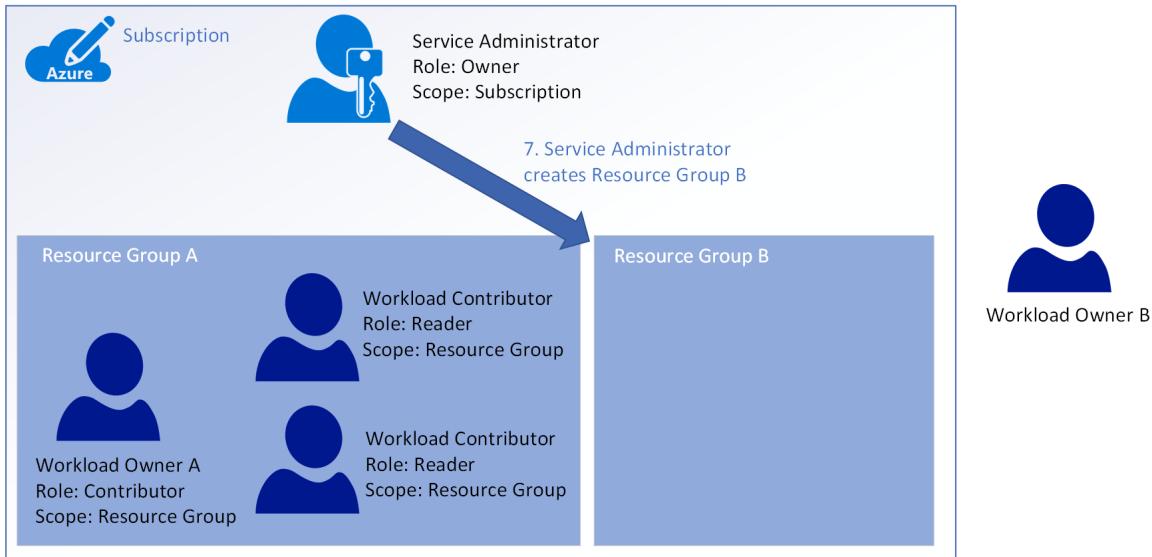
- The **service administrator** reviews the request, and adds the two **workload contributor** users to **resource group A**. Neither of these two users require permission to manage resources, so they are assigned the [built-in reader role](#).



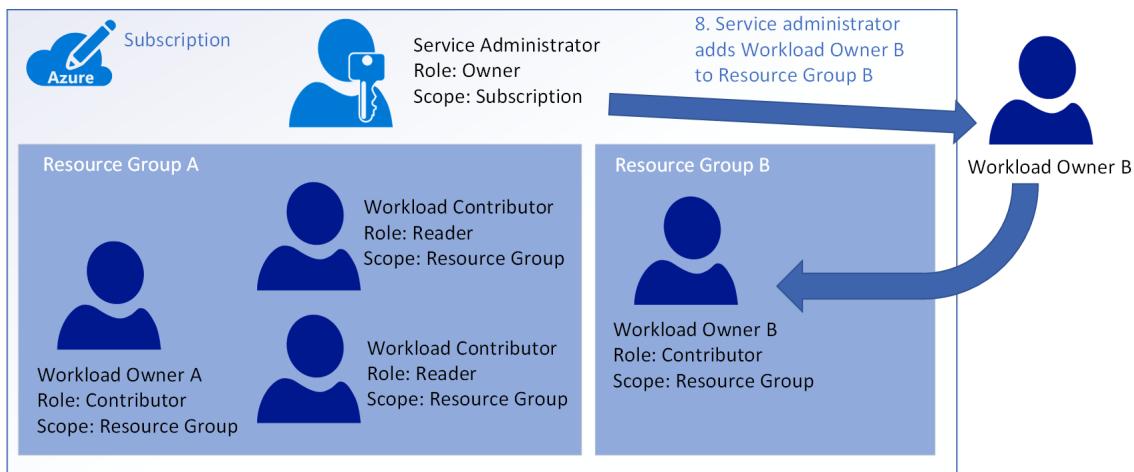
6. Next, **workload owner B** also requires a resource group to contain the resources for their workload. As with **workload owner A**, **workload owner B** initially does not have permission to take any action at the subscription scope so they must send a request to the **service administrator**.



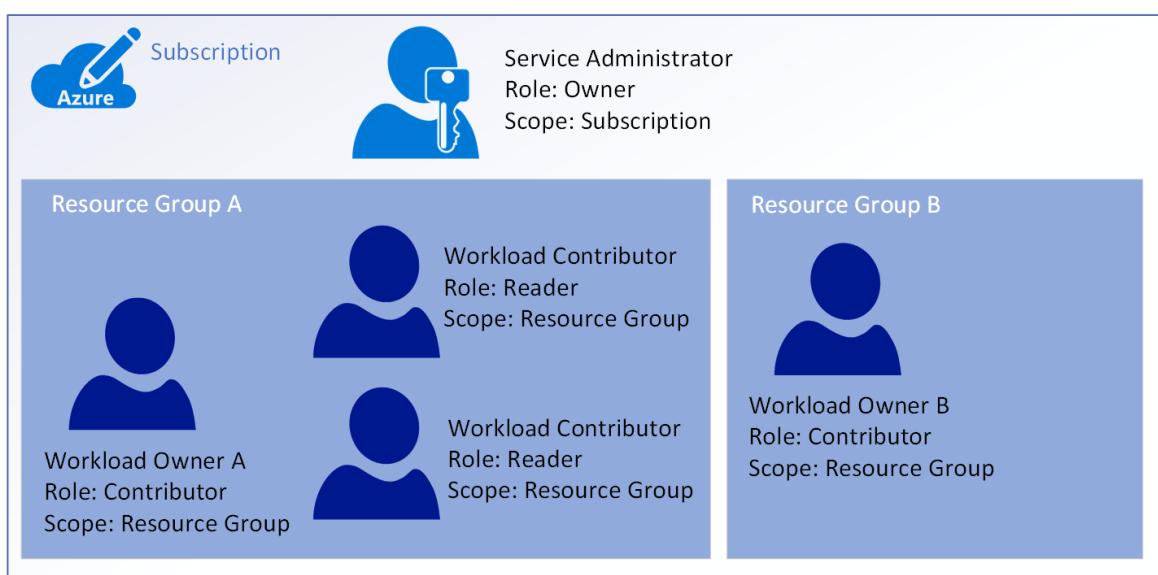
7. The **service administrator** reviews the request and creates **resource group B**.



8. The **service administrator** then adds **workload owner B** to **resource group B** and assigns the [built-in contributor role](#).



At this point, each of the workload owners is isolated in their own resource group. None of the workload owners or their team members have management access to the resources in any other resource group.



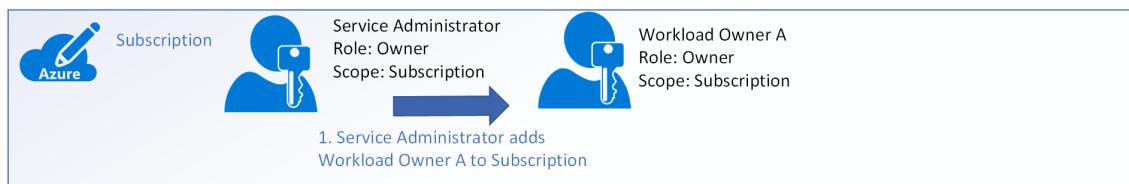
*Figure 4. A subscription with two workload owners isolated with their own resource group.*

This model is a least privilege model - each user is assigned the correct permission at the correct resource management scope.

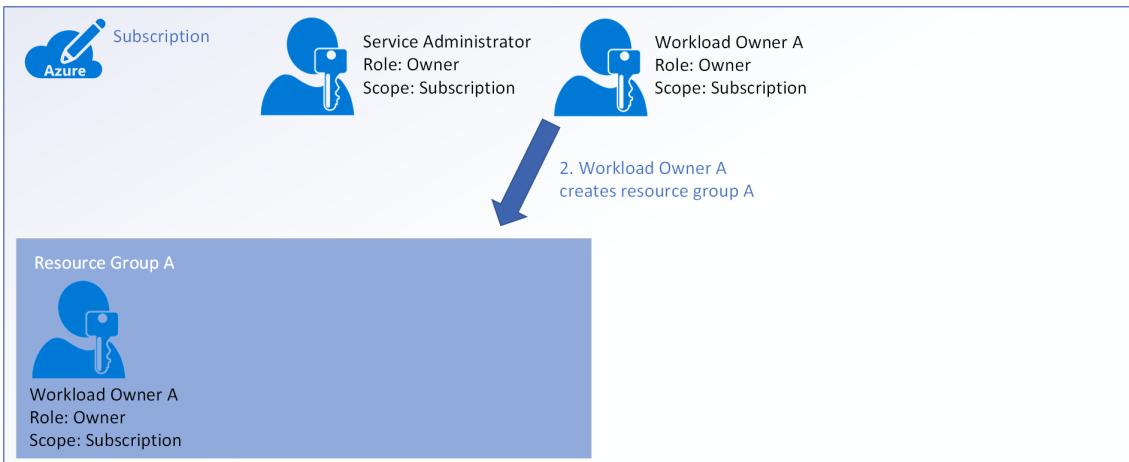
However, consider that every task in this example was performed by the **service administrator**. While this is a simple example and may not appear to be an issue because there were only two workload owners, it's easy to imagine the types of issues that would result for a large organization. For example, the **service administrator** can become a bottleneck with a large backlog of requests that result in delays.

Let's take a look at second example that reduces the number of tasks performed by the **service administrator**.

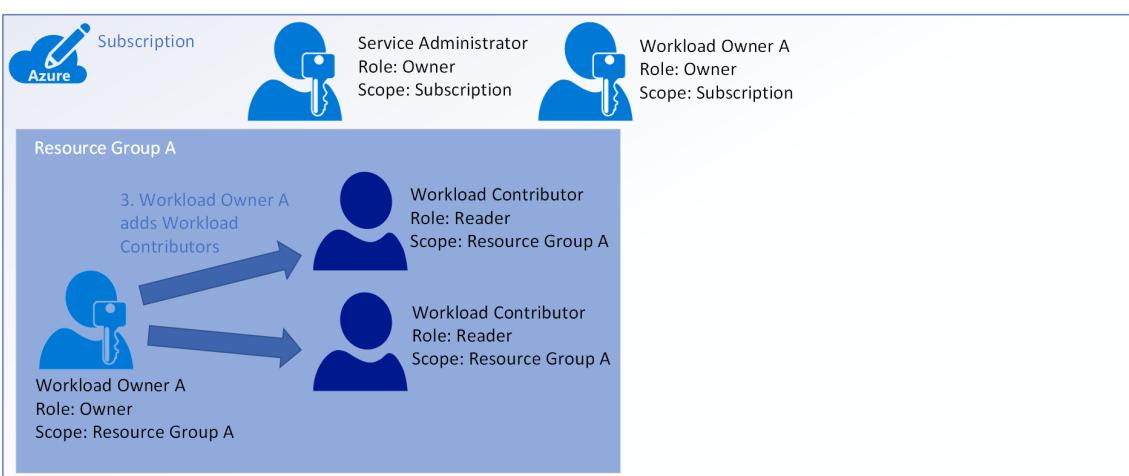
1. In this model, **workload owner A** is assigned the [built-in owner role](#) at the subscription scope, enabling them to create their own resource group: **resource group A**.



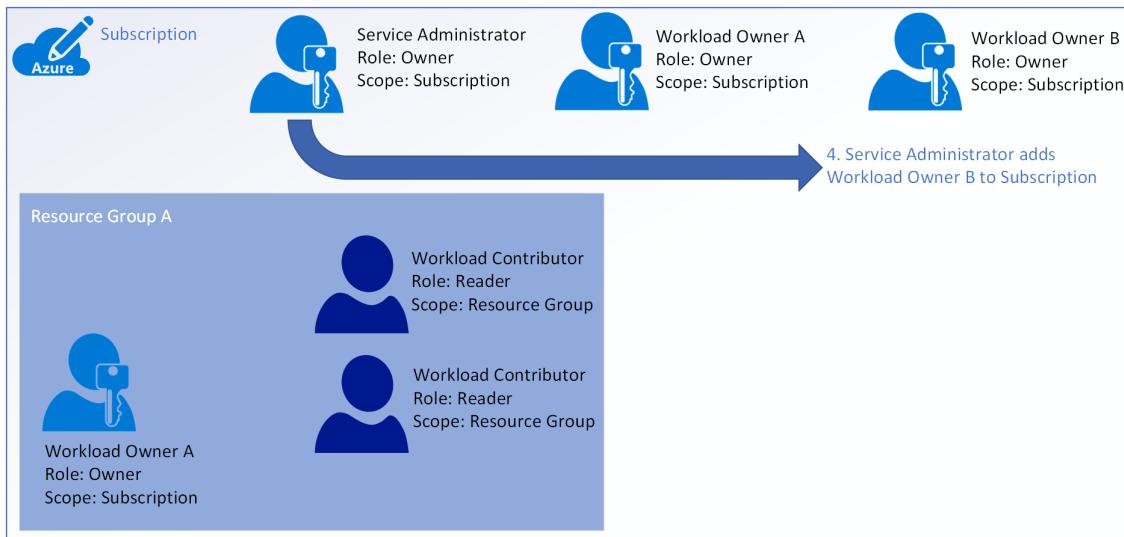
2. When **resource group A** is created, **workload owner A** is added by default and inherits the [built-in owner role](#) from the subscription scope.



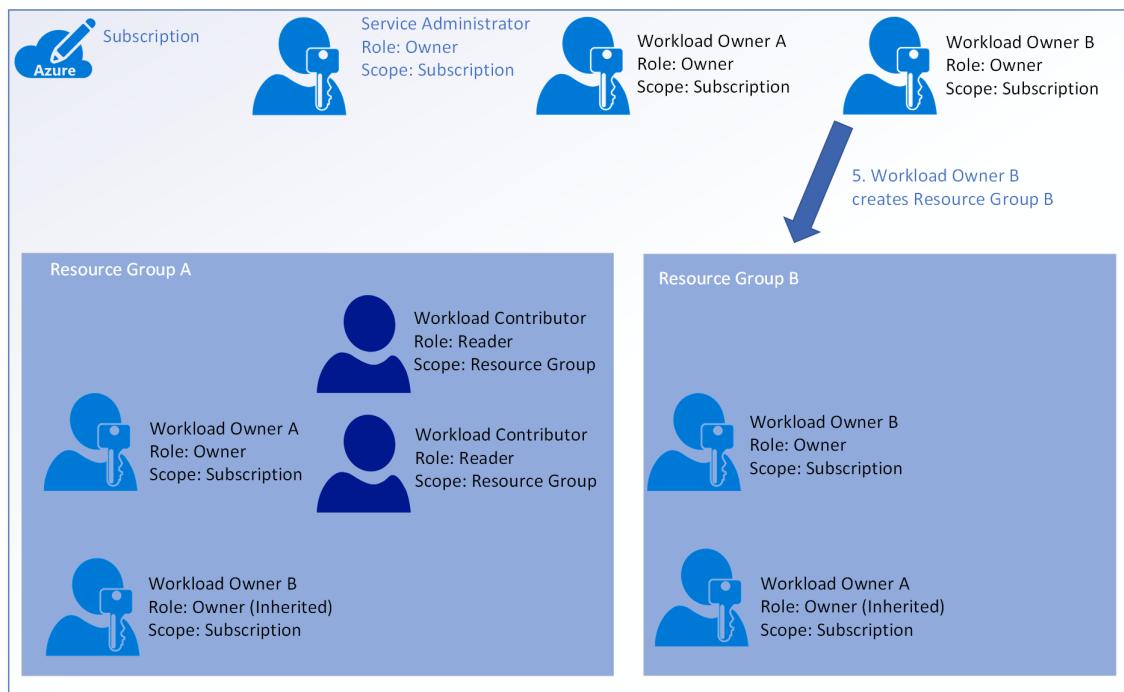
3. The [built-in owner role](#) grants **workload owner A** permission to manage access to the resource group. **Workload owner A** adds two **workload contributors** and assigns the [built-in reader role](#) to each of them.



4. **Service administrator** now adds **workload owner B** to the subscription with the built-in owner role.



5. **Workload owner B** creates **resource group B** and is added by default. Again, **workload owner B** inherits the built-in owner role from the subscription scope.



Note that in this model, the **service administrator** performed fewer actions than they did in the first example due to the delegation of management access to each of the individual workload owners.

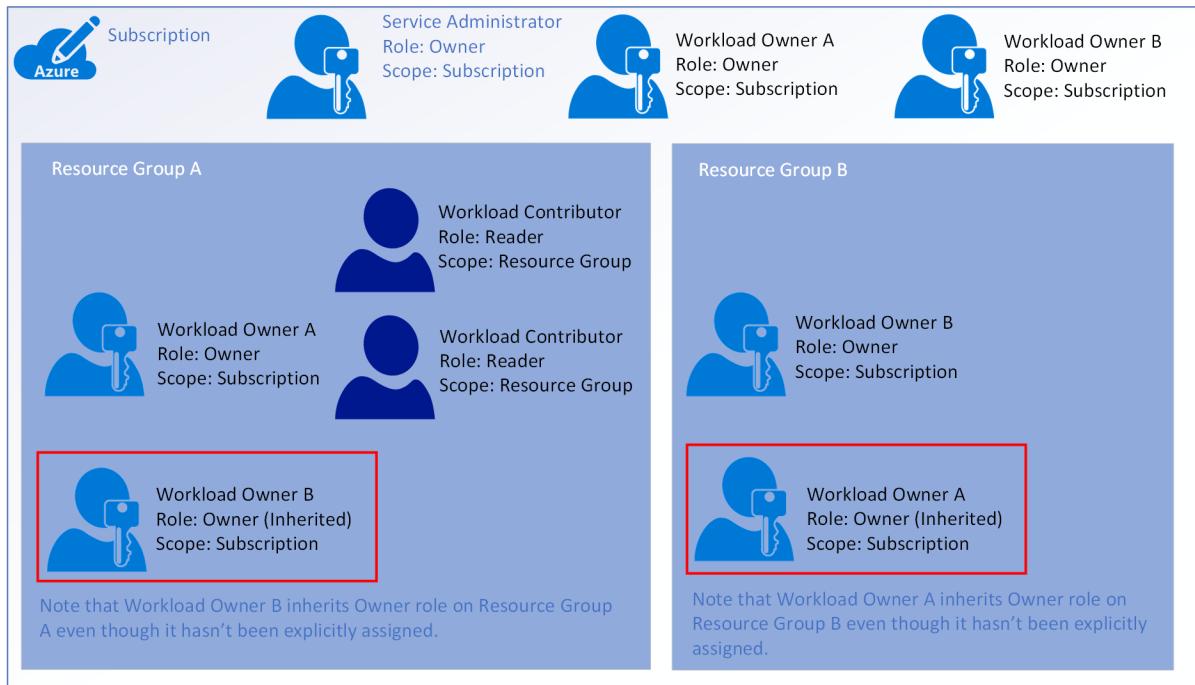


Figure 5. A subscription with a service administrator and two workload owners, all assigned the built-in owner role.

However, because both **workload owner A** and **workload owner B** are assigned the built-in owner role at the subscription scope, they have each inherited the built-in owner role for each other's resource group. This means that not only do they have full access to one another's resources, they are also able to delegate management access to each other's resource groups. For example, **workload owner B** has rights to add any other user to **resource group A** and can assign any role to them, including the built-in owner role.

If we compare each example to the requirements, we see that both examples support a single trusted user at the subscription scope with permission to grant resource access rights to the two workload owners. Each of the two workload owners did not have access to resource management by default and required the **service administrator** to explicitly assign permissions to them. However, only the first example supports the requirement that the resources associated with each workload are isolated from one another such that no workload owner has access to the resources of any other workload.

## Resource management model

Now that we've designed a permissions model of least privilege, let's move on to take a look at some practical applications of these governance models. Recall from the requirements that we must support the following three environments:

1. **Shared infrastructure:** a single group of resources shared by all workloads. These are resources such as network gateways, firewalls, and security services.
2. **Development:** multiple groups of resources representing multiple non-production ready workloads. These resources are used for proof-of-concept, testing, and other developer activities. These resources may have a more relaxed governance model to enable increased developer agility.
3. **Production:** multiple groups of resources representing multiple production workloads. These resources are used to host the private and public facing application artifacts. These resources typically have the tightest governance and security models to protect the resources, application code, and data from unauthorized access.

For each of these three environments, we have a requirement to track cost data by **workload owner**, **environment**, or both. That is, we want to know the ongoing cost of the **shared infrastructure**, the costs incurred by individuals in both the **development** and **production** environments, and finally the overall cost of

## development and production.

You have already learned that resources are scoped to two levels: **subscription** and **resource group**. Therefore, the first decision is how to organize environments by **subscription**. There are only two possibilities: a single subscription, or, multiple subscriptions.

Before we look at examples of each of these models, let's review the management structure for subscriptions in Azure.

Recall from the requirements that we have an individual in the organization who is responsible for subscriptions, and this user owns the **subscription owner** account in the Azure AD tenant. However, this account does not have permission to create subscriptions. Only the **Azure Account Owner** has permission to do this:

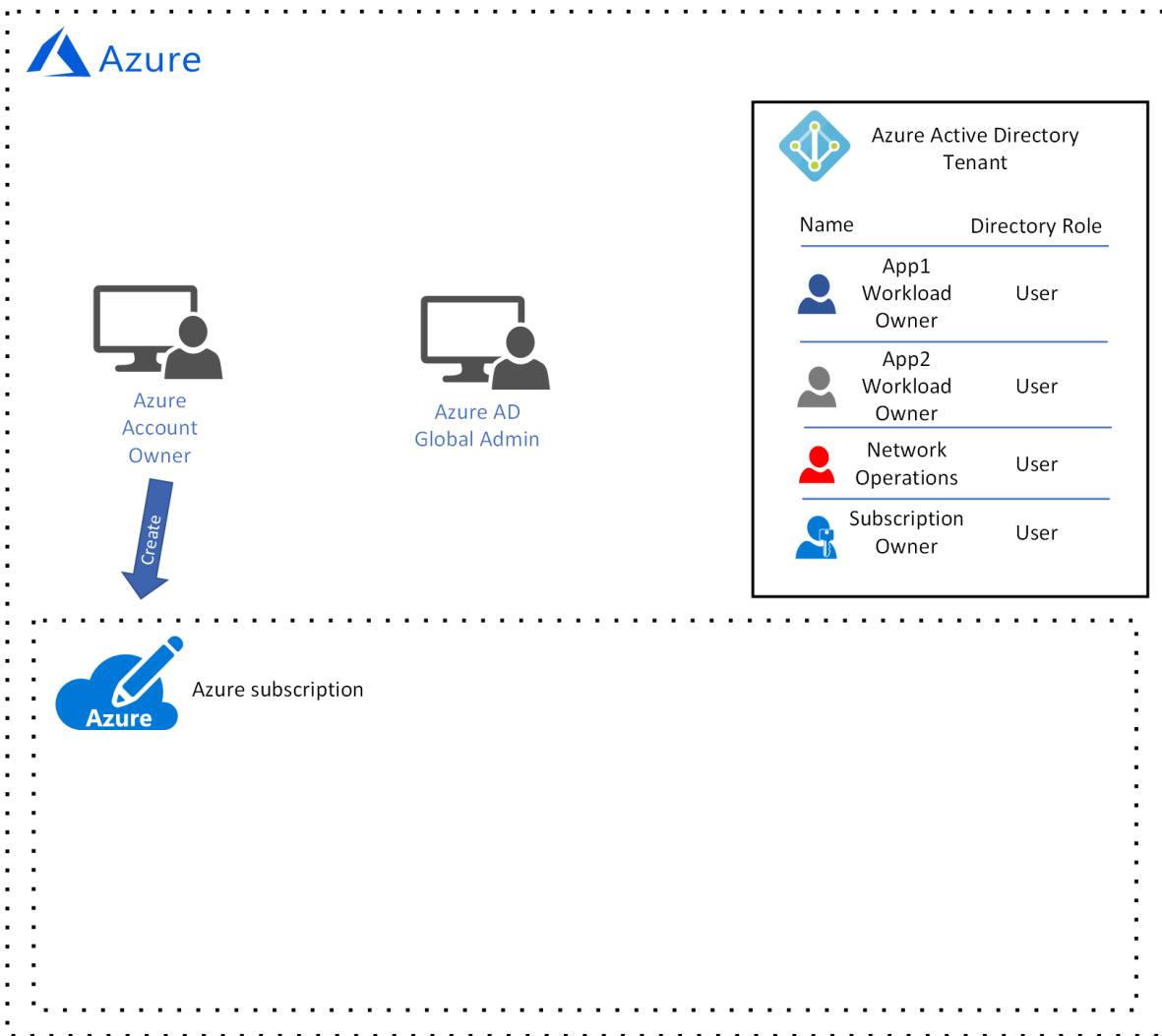


Figure 6. An Azure Account Owner creates a subscription.

Once the subscription has been created, the **Azure Account Owner** can add the **subscription owner** account to the subscription with the **owner** role:

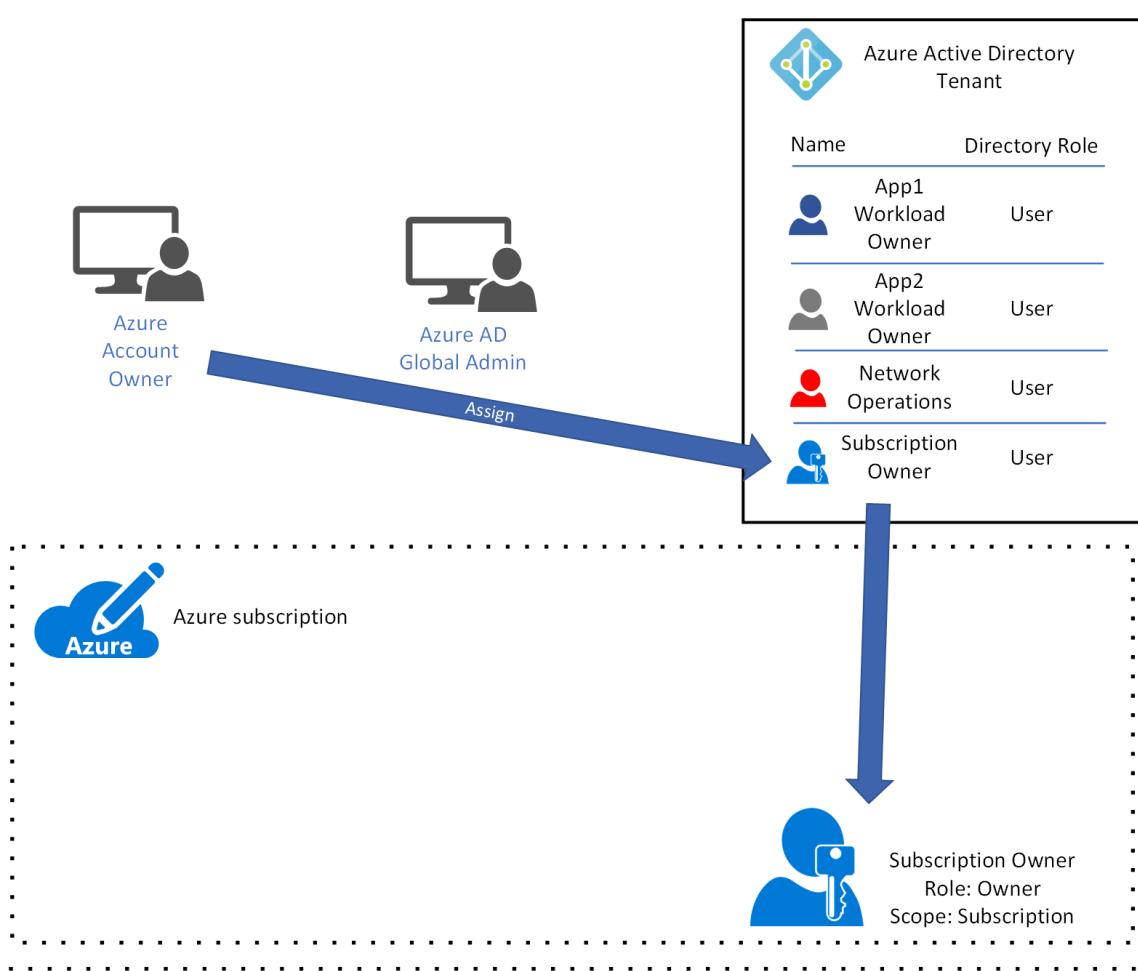


Figure 7. The Azure Account Owner adds the **subscription owner** user account to the subscription with the **owner** role.

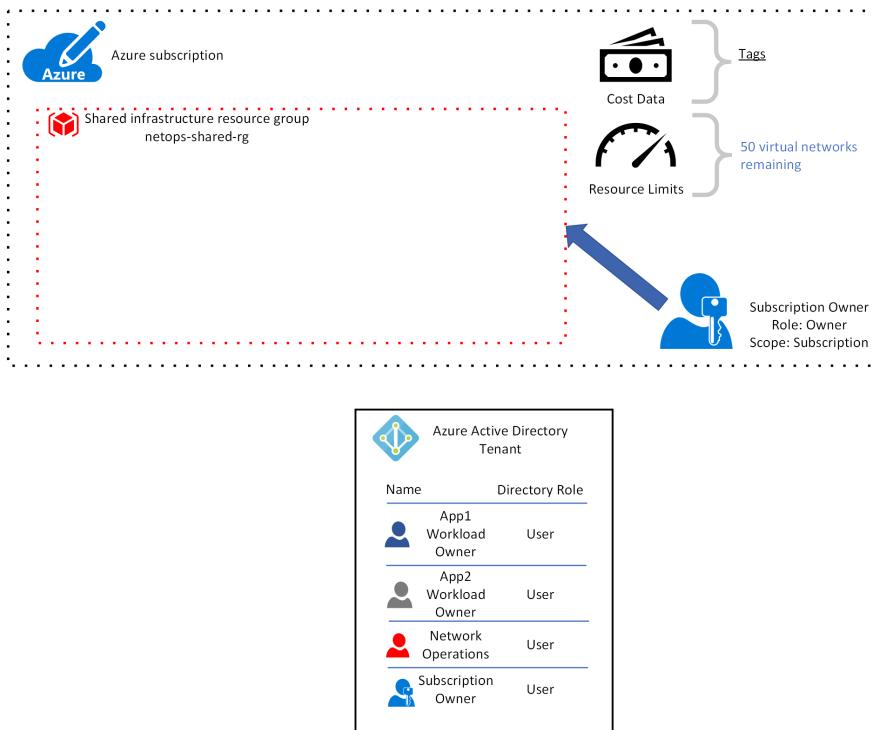
The **subscription owner** can now create **resource groups** and delegate resource access management.

First let's look at an example resource management model using a single subscription. The first decision is how to align resource groups to the three environments. We have two options:

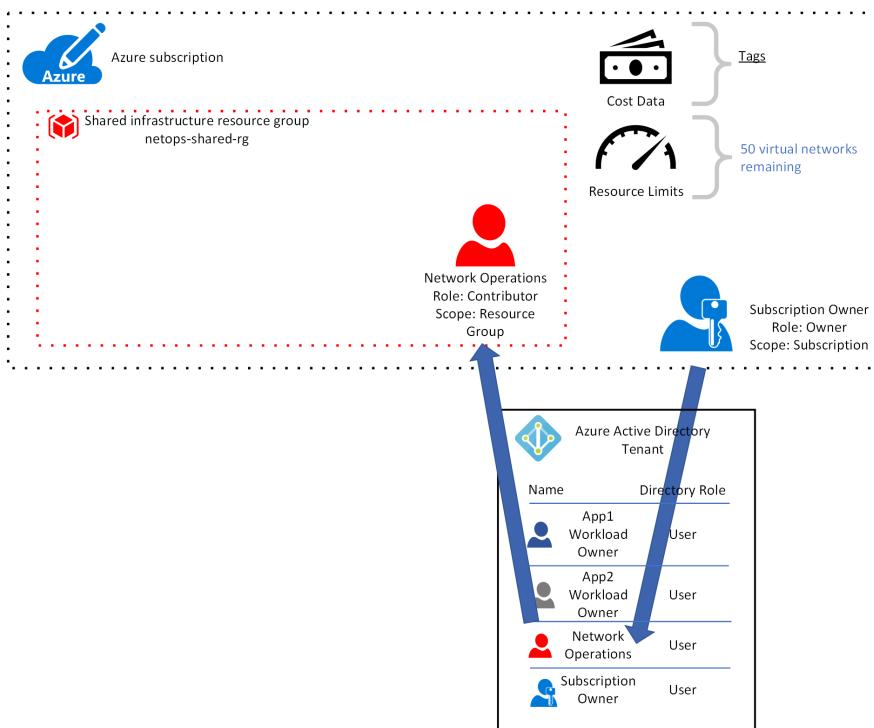
1. Align each environment to a single resource group. All shared infrastructure resources are deployed to a single **shared infrastructure** resource group. All resources associated with development workloads are deployed to a single **development** resource group. All resources associated with production workloads are deployed into a single **production** resource group for the **production** environment.
2. Create separate resource groups for each workload, using a naming convention and tags to align resource groups with each of the three environments.

Let's begin by evaluating the first option. We'll be using the permissions model that we discussed in the previous section, with a single subscription service administrator who creates resource groups and adds users to them with either the built-in **contributor** or **reader** role.

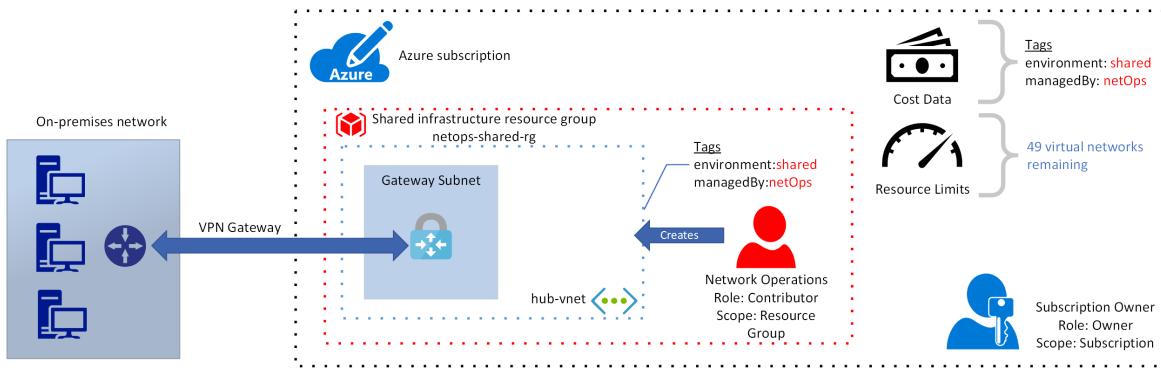
1. The first resource group deployed represents the **shared infrastructure** environment. The **subscription owner** creates a resource group for the shared infrastructure resources named **netops-shared-rg**.



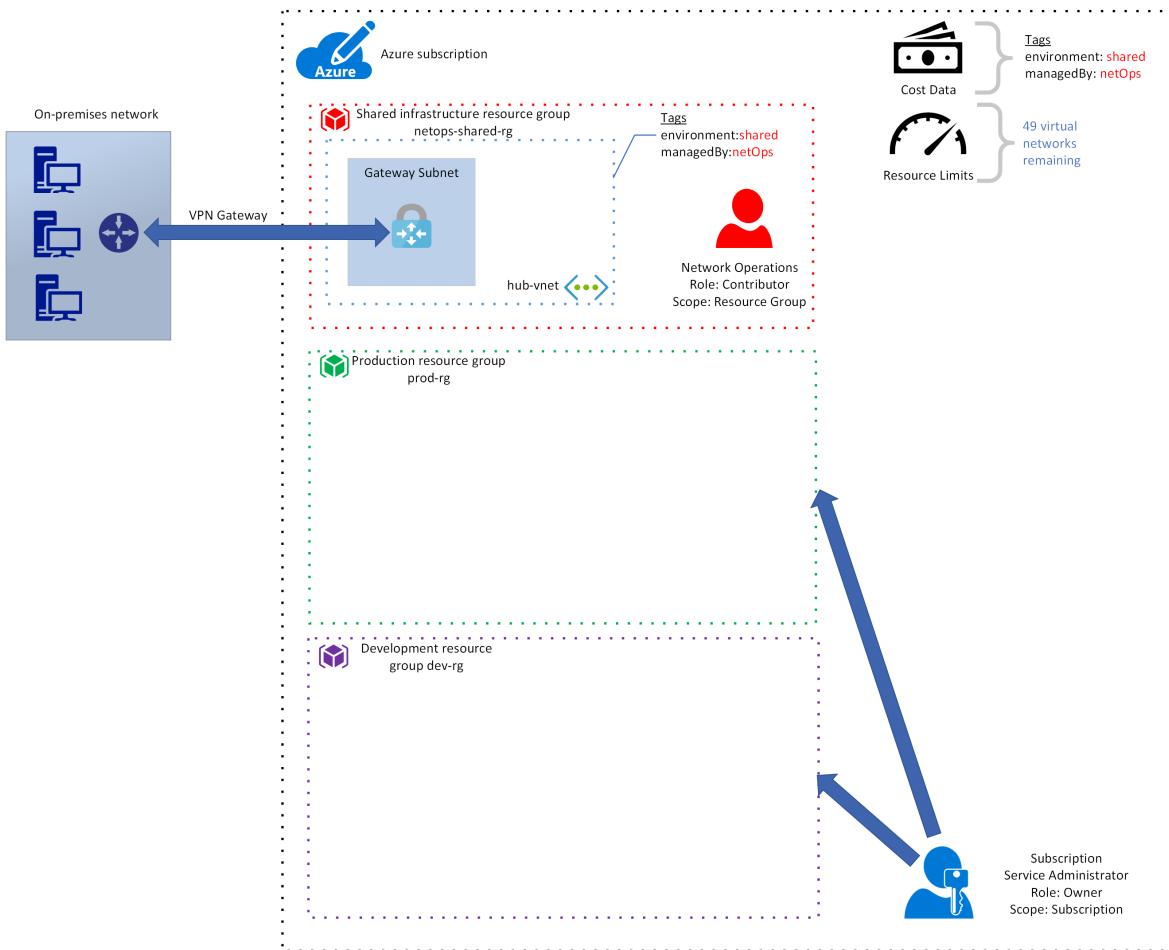
2. The **subscription owner** adds the **network operations user** account to the resource group and assigns the **contributor** role.



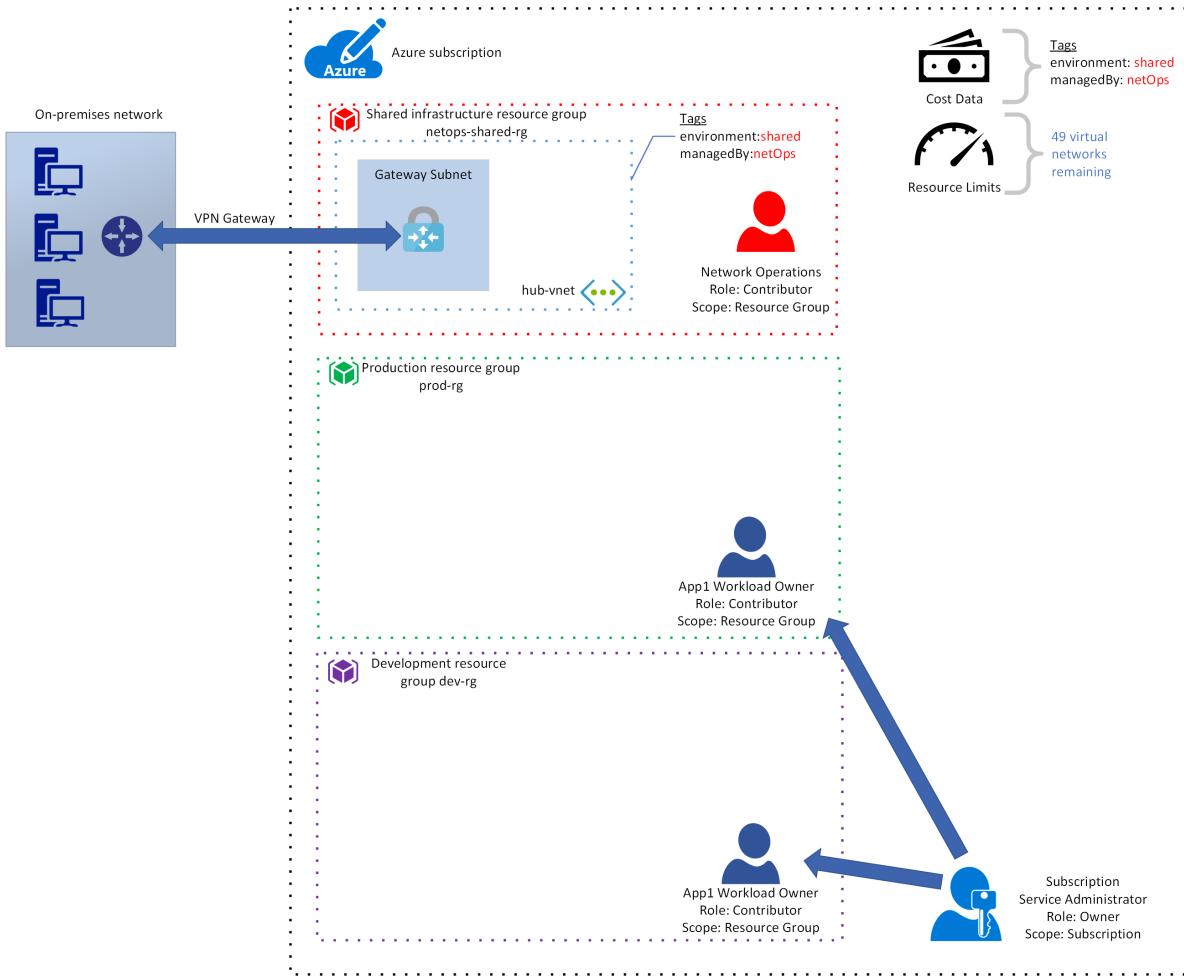
3. The **network operations user** creates a [VPN gateway](#) and configures it to connect to the on-premises VPN appliance. The **network operations** user also applies a pair of [tags](#) to each of the resources: *environment:shared* and *managedBy:netOps*. When the **subscription service administrator** exports a cost report, costs will be aligned with each of these tags. This allows the **subscription service administrator** to pivot costs using the *environment* tag and the *managedBy* tag. Notice the **resource limits** counter at the top right-hand side of the figure. Each Azure subscription has [service limits](#), and to help you understand the effect of these limits we'll follow the virtual network limit for each subscription. There is a default limit of 50 virtual networks per subscription, and after the first virtual network is deployed there are now 49 available.



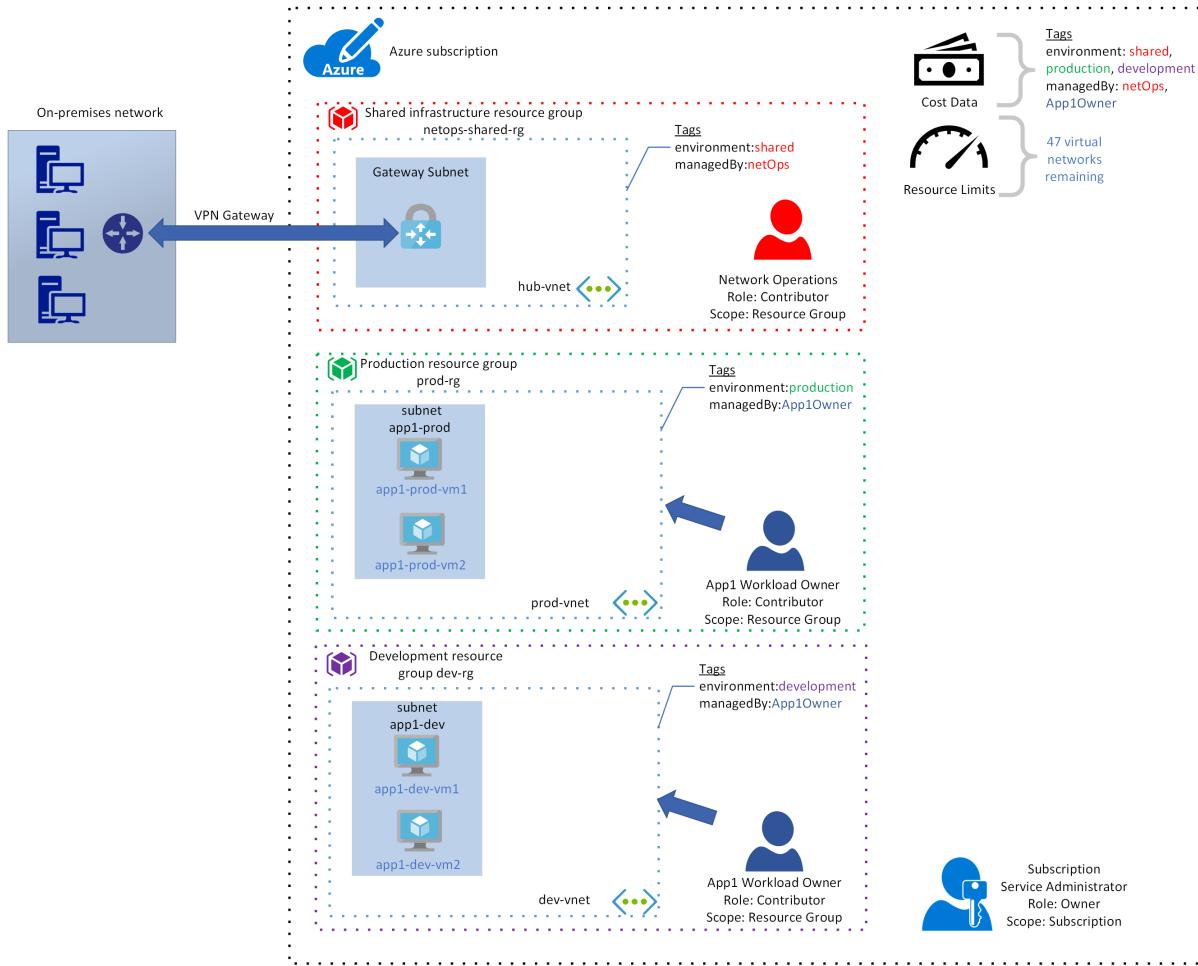
4. Two more resource groups are deployed, the first is named *prod-rg*. This resource group is aligned with the **production** environment. The second is named *dev-rg* and is aligned with the **development** environment. All resources associated with production workloads are deployed to the **production** environment and all resources associated with development workloads are deployed to the **development** environment. In this example we'll only deploy two workloads to each of these two environments so we won't encounter any Azure subscription service limits. However, it's important to consider that each resource group has a limit of 800 resources per resource group. Therefore, if we keep adding workloads to each resource group it is possible that this limit will be reached.



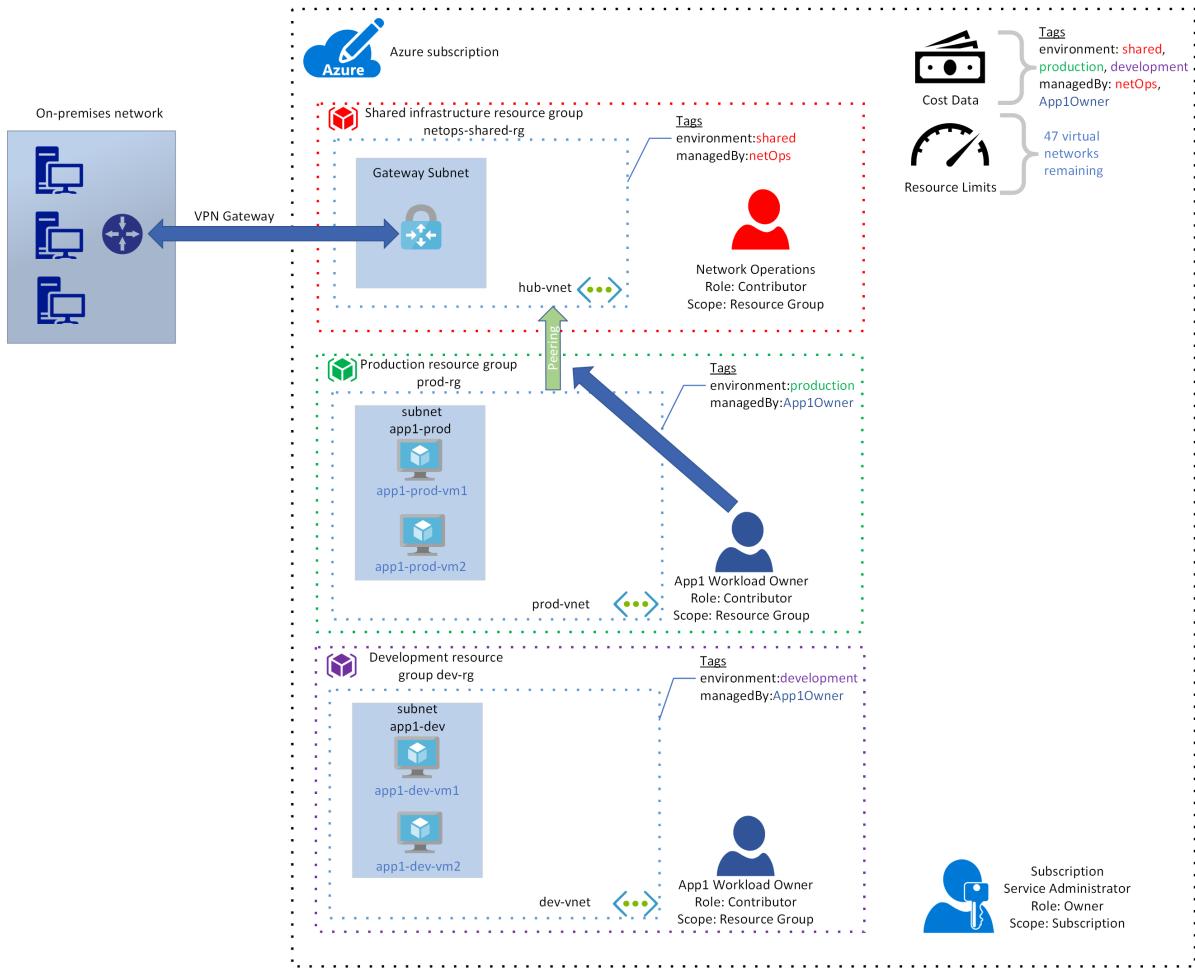
5. The first **workload owner** sends a request to the **subscription service administrator** and is added to each of the **development** and **production** environment resource groups with the **contributor** role. As you learned earlier, the **contributor** role allows the user to perform any operation other than assigning a role to another user. The first **workload owner** can now create the resources associated with their workload.



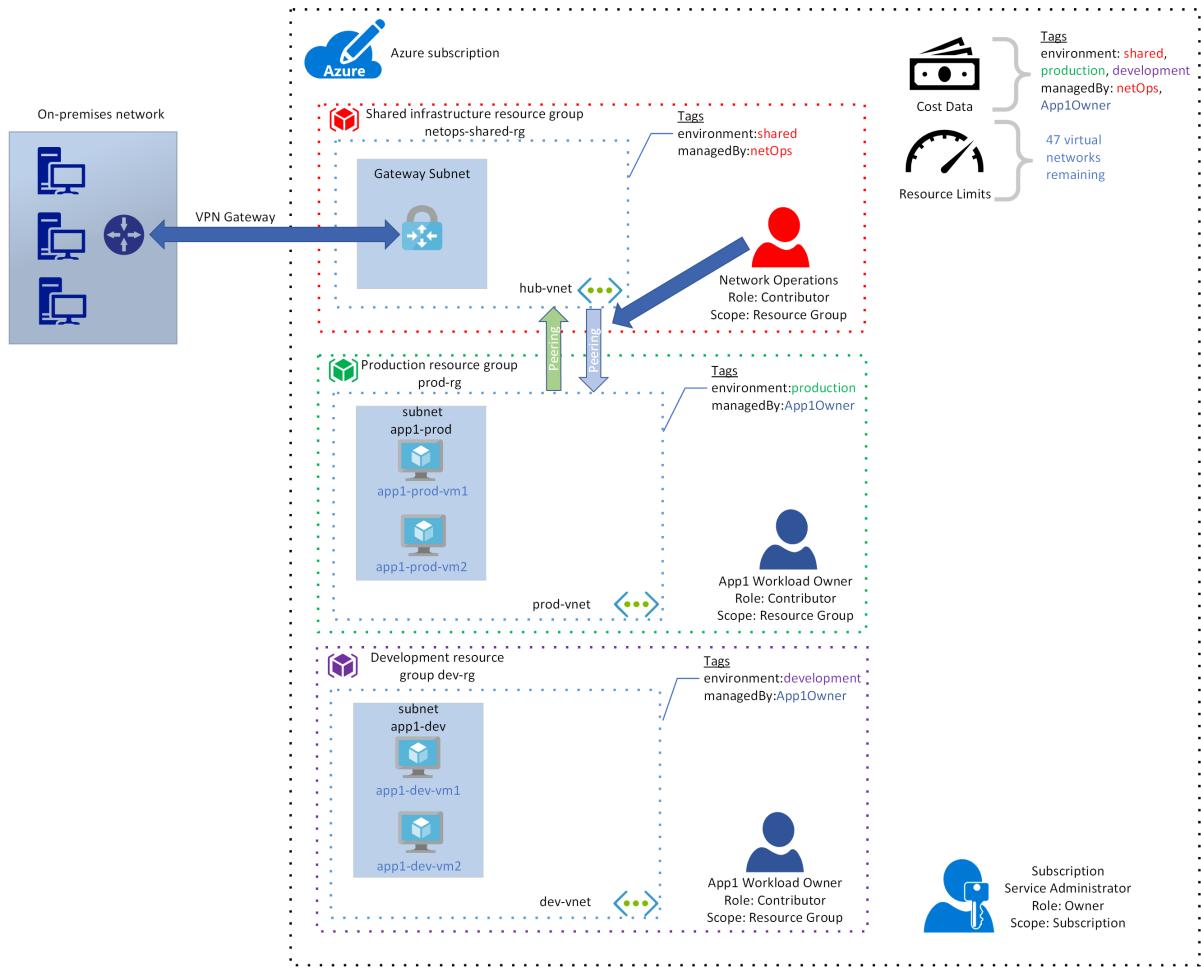
6. The first **workload owner** creates a virtual network in each of the two resource groups with a pair of virtual machines in each. The first **workload owner** applies the *environment* and *managedBy* tags to all resources. Note that the Azure service limit counter is now at 47 virtual networks remaining.



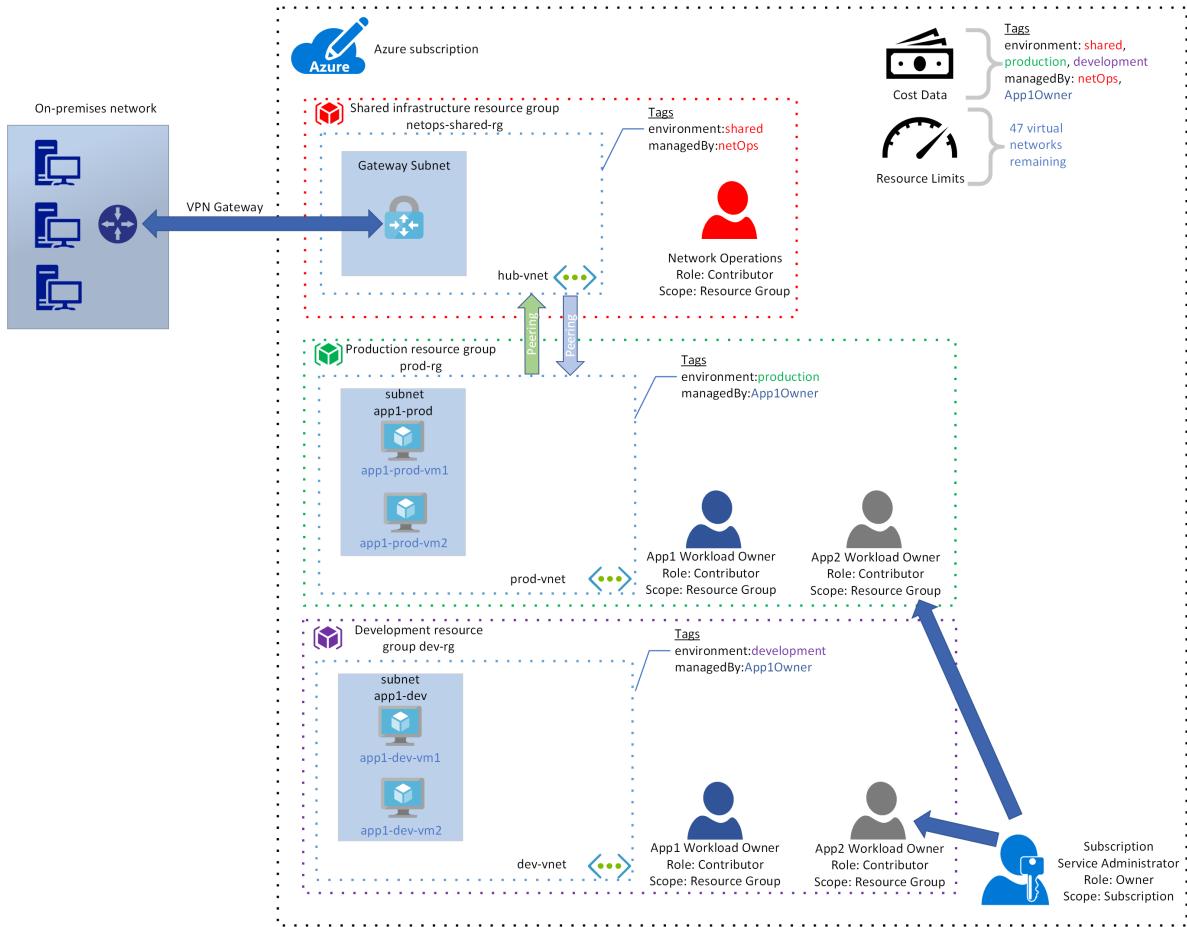
7. Each of the virtual networks does not have connectivity to on-premises when they are created. In this type of architecture, each virtual network must be peered to the **hub-vnet** in the **shared infrastructure** environment. Virtual network peering creates a connection between two separate virtual networks and allows network traffic to travel between them. Note that virtual network peering is not inherently transitive. A peering must be specified in each of the two virtual networks that are connected, and if only one of the virtual networks specifies a peering the connection is incomplete. To illustrate the effect of this, the first **workload owner** specifies a peering between **prod-vnet** and **hub-vnet**. The first peering is created, but no traffic flows because the complementary peering from **hub-vnet** to **prod-vnet** has not yet been specified. The first **workload owner** contacts the **network operations** user and requests this complementary peering connection.



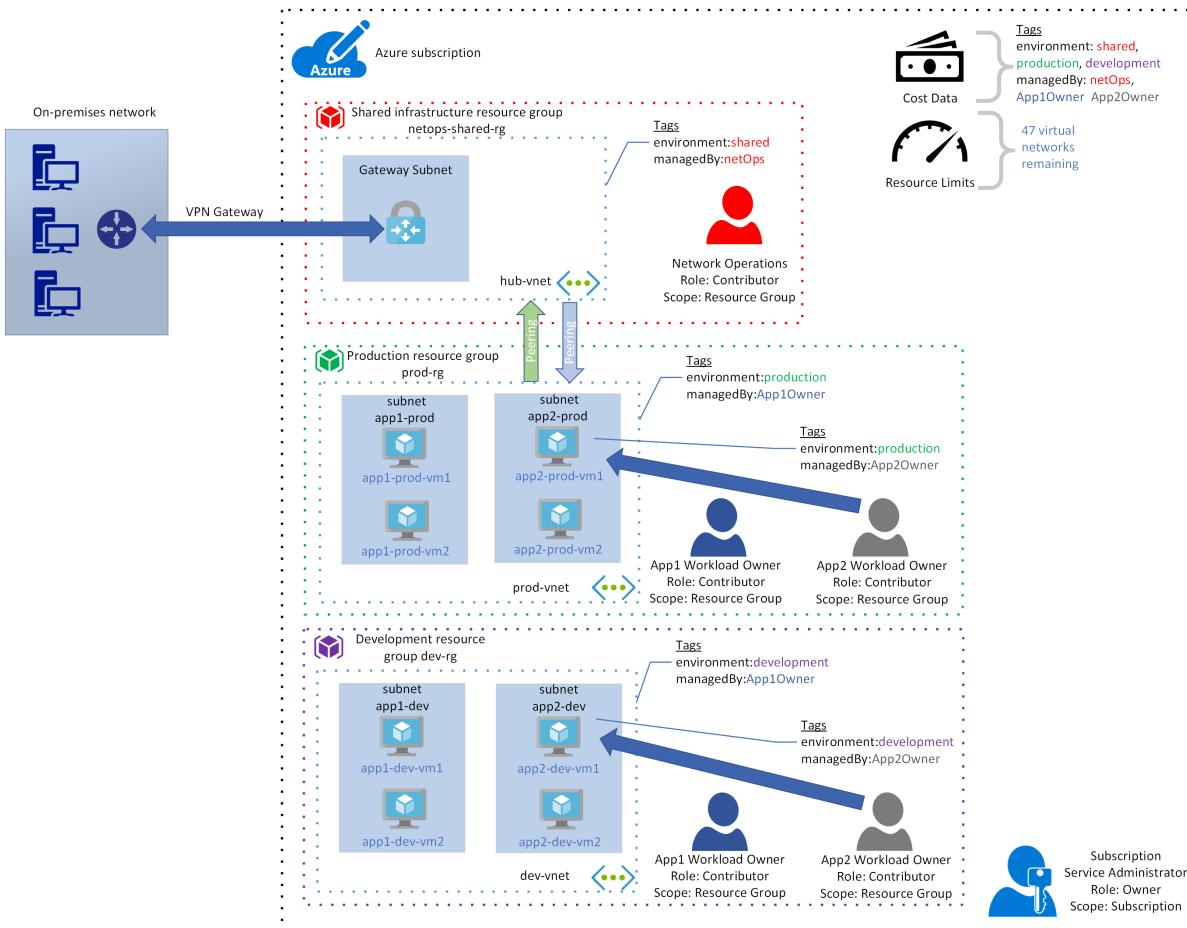
8. The **network operations** user reviews the request, approves it, then specifies the peering in the settings for the **hub-vnet**. The peering connection is now complete and network traffic flows between the two virtual networks.



- Now, a second **workload owner** sends a request to the **subscription service administrator** and is added to the existing **production** and **development** environment resource groups with the **contributor** role. The second **workload owner** has the same permissions on all resources as the first **workload owner** in each resource group.

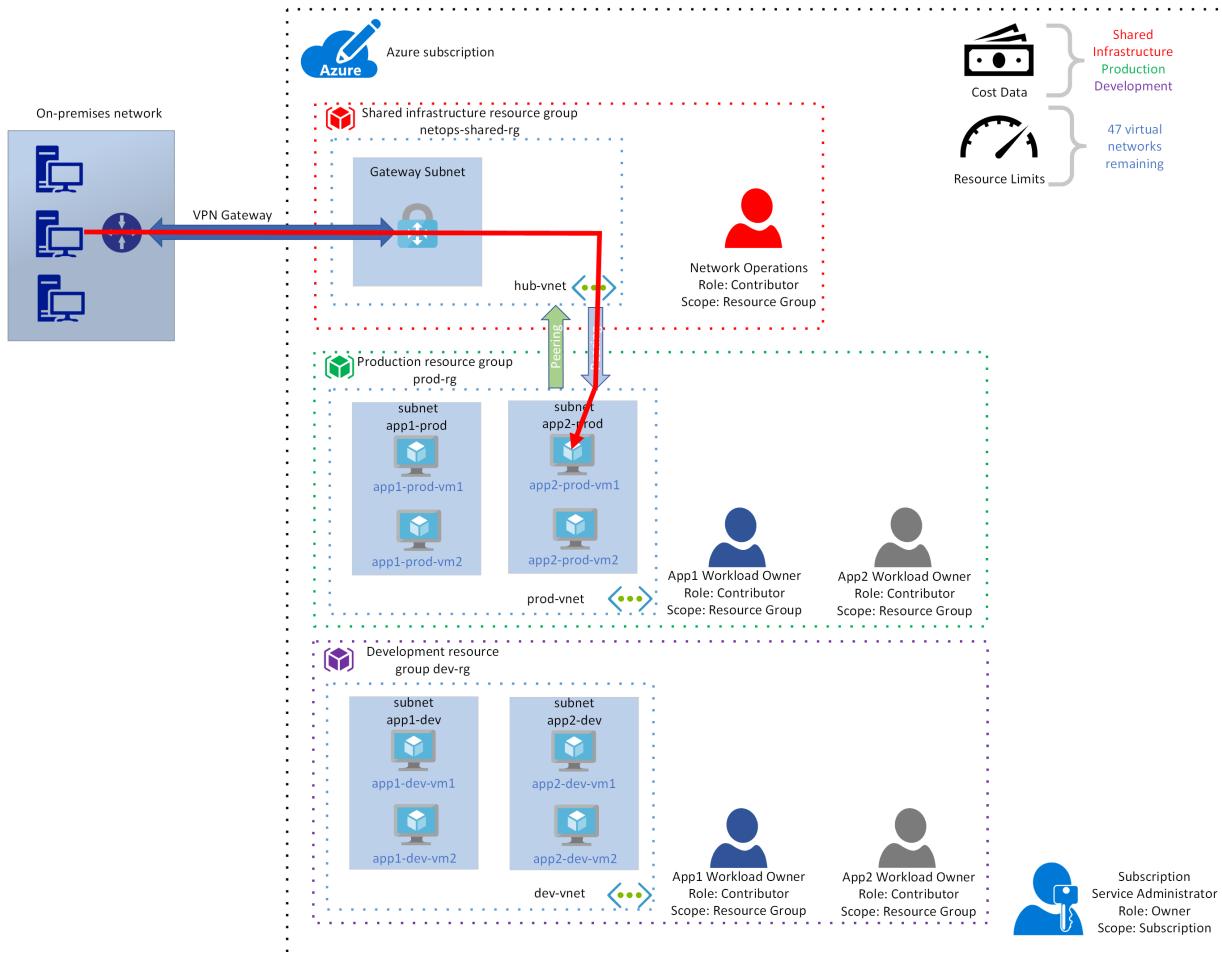


10. The second **workload owner** creates a subnet in the **prod-vnet** virtual network, then adds two virtual machines. The second **workload owner** applies the *environment* and *managedBy* tags to each resource.



This example resource management model enables us to manage resources in the three required environments. The shared infrastructure resources are protected because there's only a single user in the subscription with permission to access those resources. Each of the workload owners is able to utilize the share infrastructure resources without having any permissions on the actual shared resources themselves. However, This management model fails the requirement for workload isolation - each of the two **workload owners** are able to access the resources of the other's workload.

There's another important consideration with this model that may not be immediately obvious. In the example, it was **app1 workload owner** that requested the network peering connection with the **hub-vnet** to provide connectivity to on-premises. The **network operations** user evaluated that request based on the resources deployed with that workload. When the **subscription owner** added **app2 workload owner** with the **contributor** role, that user had management access rights to all resources in the **prod-rg** resource group.

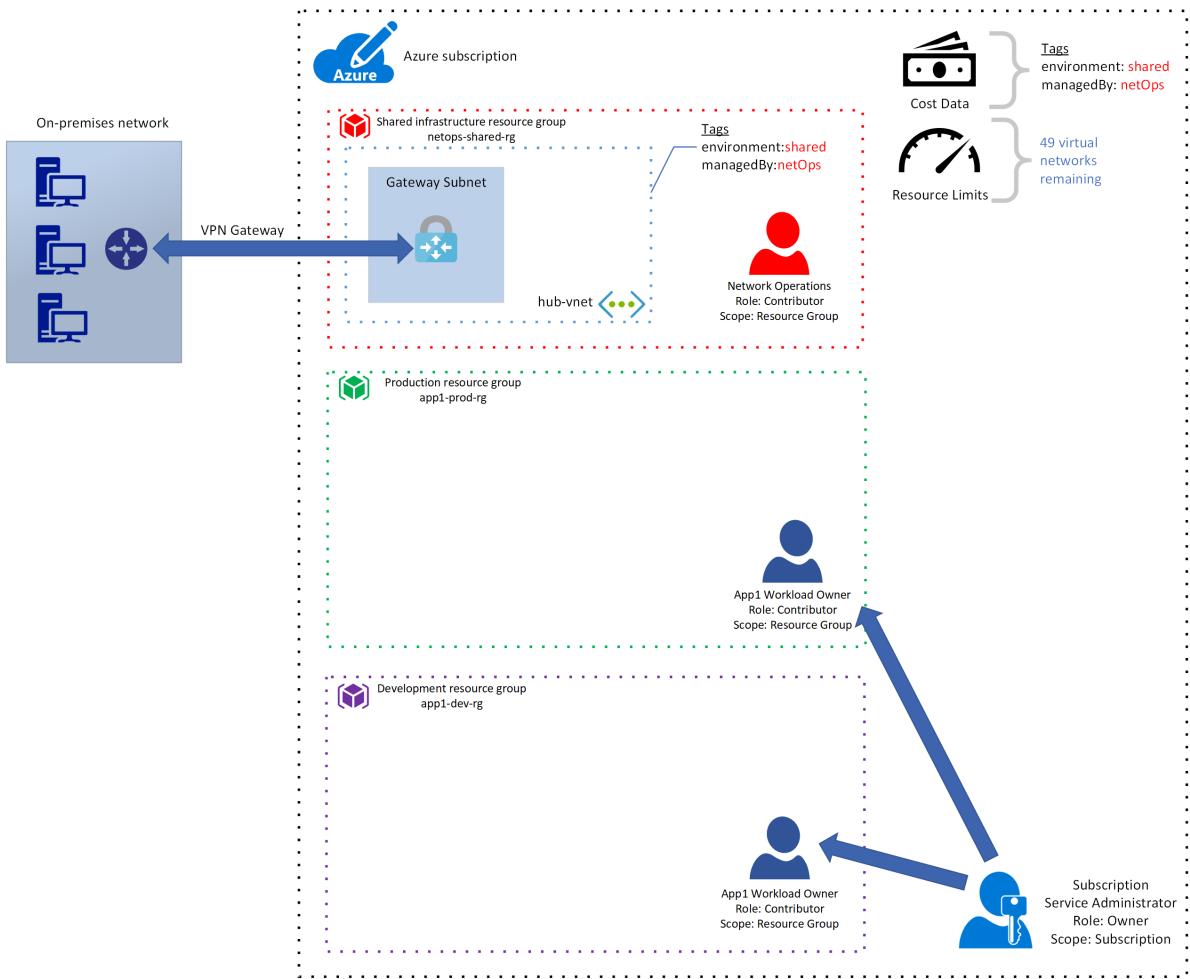


This means **app2 workload owner** had permission to deploy their own subnet with virtual machines in the **prod-vnet** virtual network. By default, those virtual machines now have access to the on-premises network. The **network operations** user is not aware of those machines and did not approve their connectivity to on-premises.

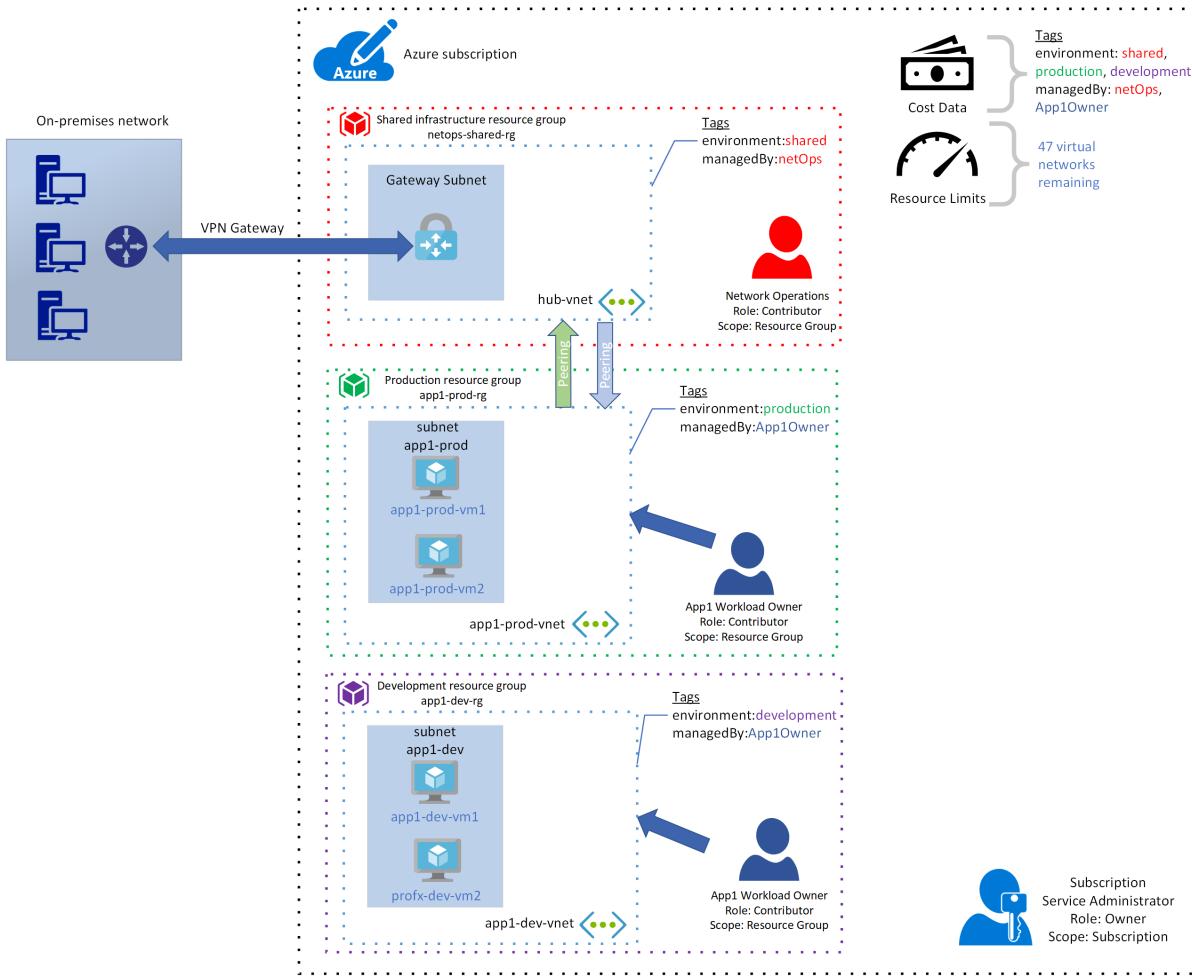
Next, let's look at a single subscription with multiple resources groups for different environments and workloads. Note that in the previous example, the resources for each environment were easily identifiable because they were in the same resource group. Now that we no longer have that grouping, we will have to rely on a resource group naming convention to provide that functionality.

1. The **shared infrastructure** resources will still have a separate resource group in this model, so that remains the same. Each workload requires two resource groups - one for each of the **development** and **production** environments. For the first workload, the **subscription owner** creates two resource groups. The first is named **app1-prod-rg** and the second one is named **app1-dev-rg**. As discussed earlier, this naming convention identifies the resources as being associated with the first workload, **app1**, and either the **dev** or **prod** environment. Again, the **subscription owner** adds **app1 workload owner** to the resource group with the

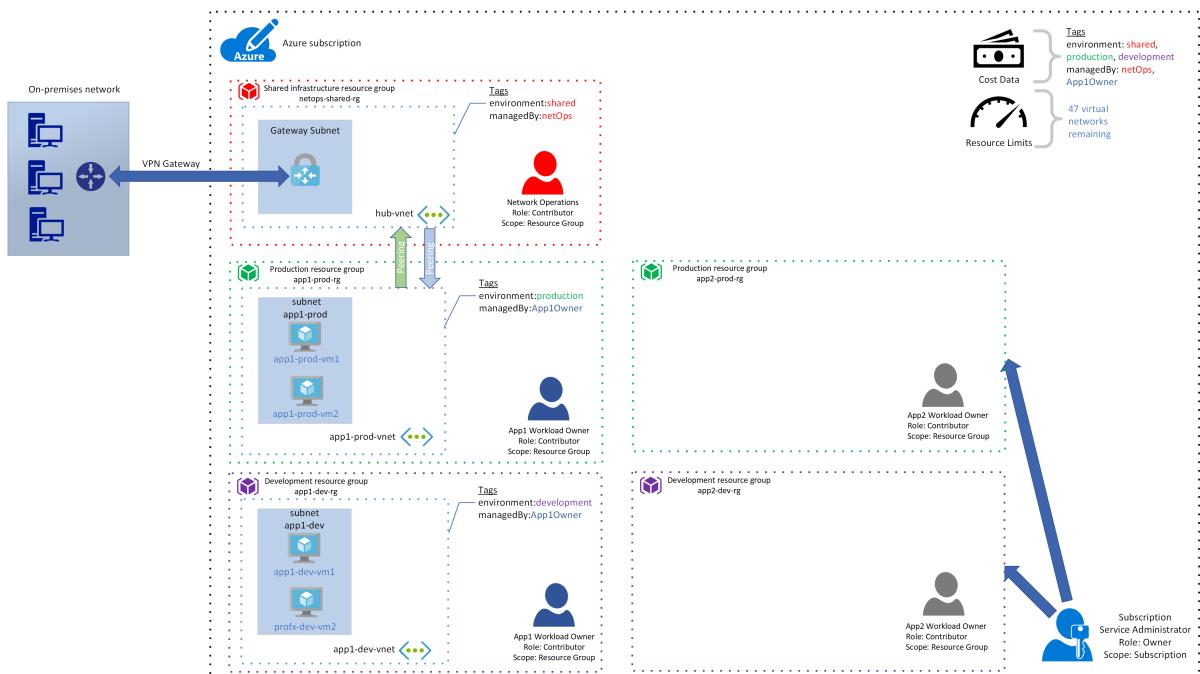
**contributor** role.



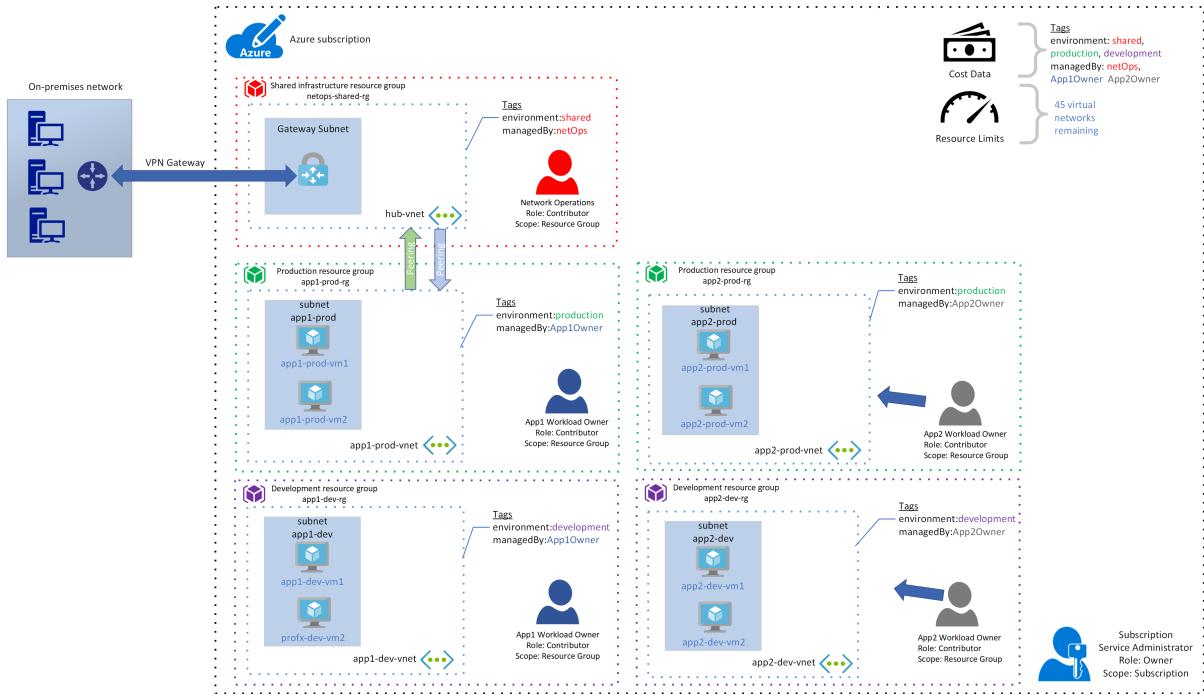
- Similar to the first example, **app1 workload owner** deploys a virtual network named **app1-prod-vnet** to the **production** environment, and another named **app1-dev-vnet** to the **development** environment. Again, **app1 workload owner** sends a request to the **network operations** user to create a peering connection. Note that **app1 workload owner** adds the same tags as in the first example, and the limit counter has been decremented to 47 virtual networks remaining in the subscription.



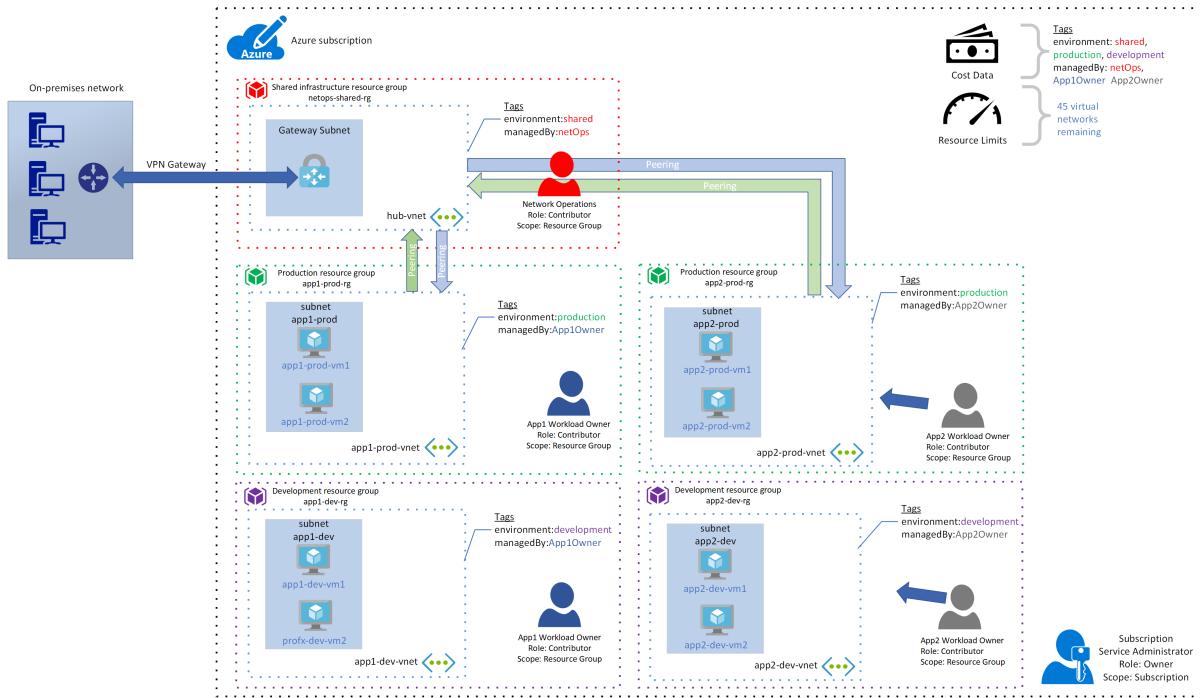
3. The **subscription owner** now creates two resource groups for **app2 workload owner**. Following the same conventions as for **app1 workload owner**, the resource groups are named **app2-prod-rg** and **app2-dev-rg**. The **subscription owner** adds **app2 workload owner** to each of the resource groups with the **contributor** role.



4. *App2 workload owner* deploys virtual networks and virtual machines to the resource groups with the same naming conventions. Tags are added and the limit counter has been decremented to 45 virtual networks remaining in the *subscription*.



5. App2 workload owner sends a request to the *network operations* user to peer the *app2-prod-vnet* with the *hub-vnet*. The *network operations* user creates the peering connection.



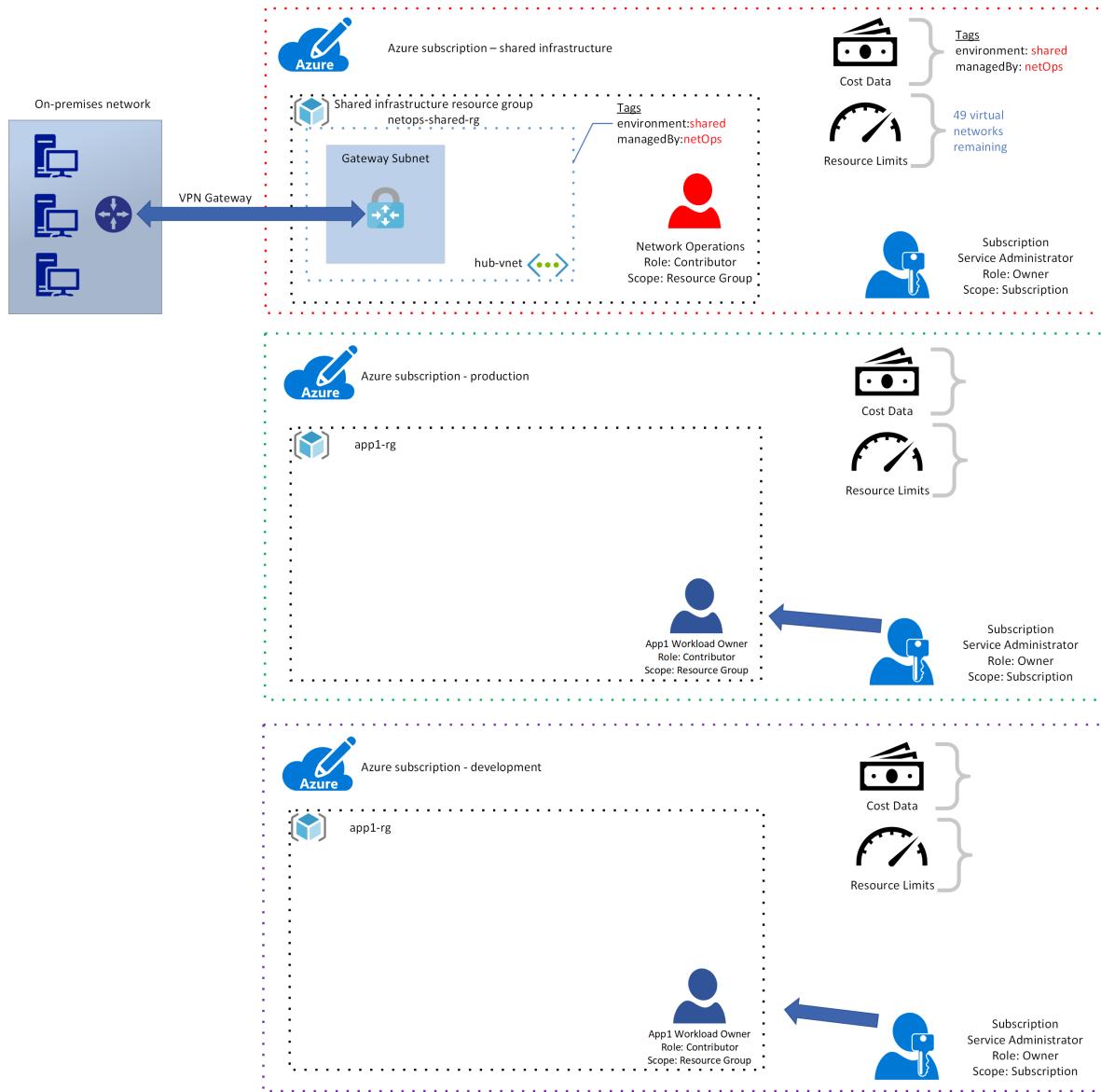
The resulting management model is similar to the first example, with several key differences:

- Each of the two workloads is isolated by workload and by environment.
- This model required two more virtual networks than the first example model. While this is not an important distinction with only two workloads, the theoretical limit on the number of workloads for this model is 24.
- Resources are no longer grouped in a single resource group for each environment. Grouping resources requires an understanding of the naming conventions used for each environment.
- Each of the peered virtual network connections was reviewed and approved by the *network operations* user.

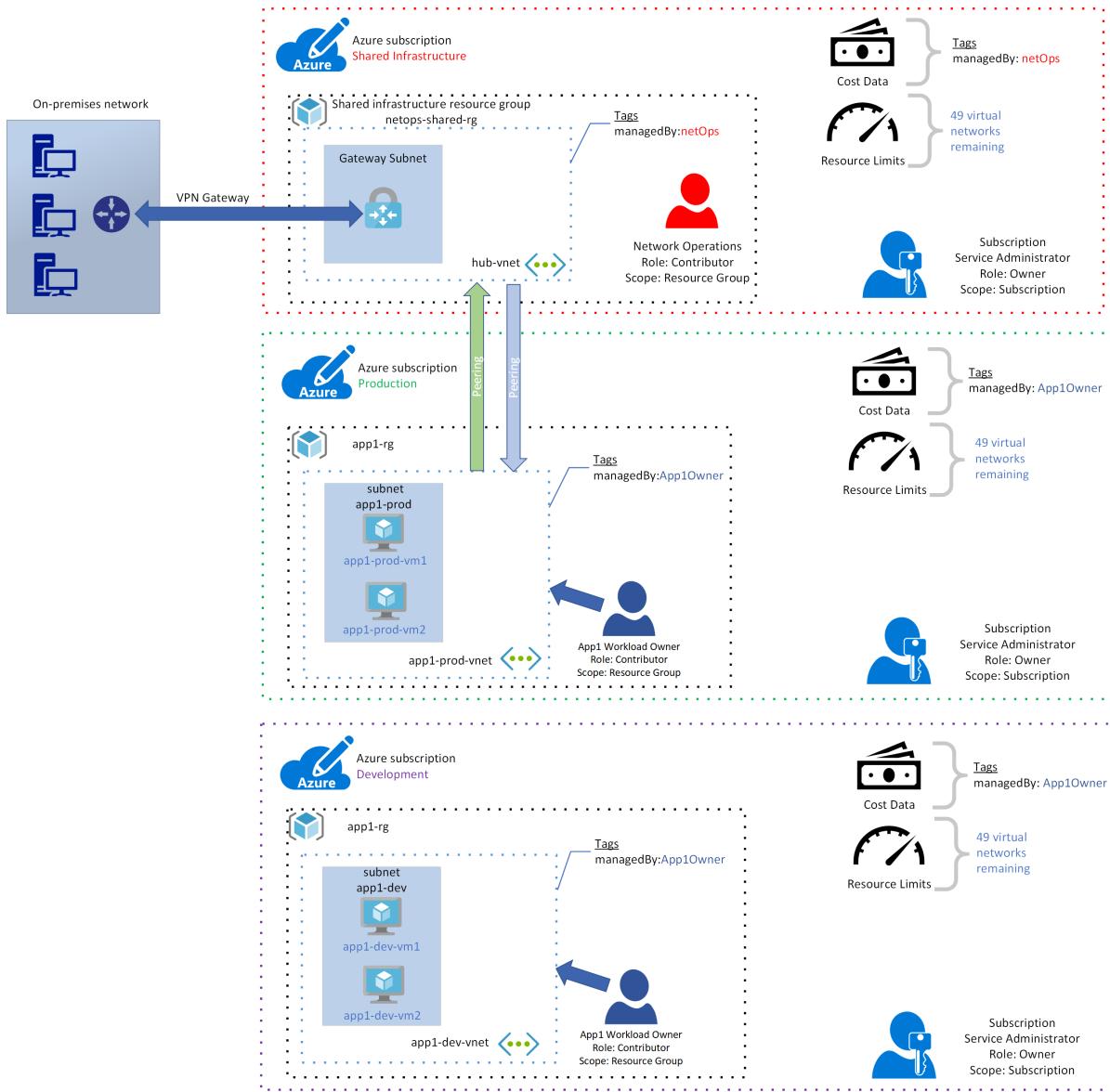
Now let's look at a resource management model using multiple subscriptions. In this model, we'll align each of the three environments to a separate subscription: a **shared services** subscription, **production** subscription, and finally a **development** subscription. The considerations for this model are similar to a model using a single subscription in that we have to decide how to align resource groups to workloads. We've already determined that

creating a resource group for each workload satisfies the workload isolation requirement, so we'll stick with that model in this example.

1. In this model, there are three *subscriptions*: *shared infrastructure*, *production*, and *development*. Each of these three subscriptions requires a *subscription owner*, and in the simple example we'll use the same user account for all three. The *shared infrastructure* resources are managed similarly to the first two examples above, and the first workload is associated with the *app1-rg* in the *production* environment and the same-named resource group in the *development* environment. The *app1 workload owner* is added to each of the resource group with the *contributor* role.



2. As with the earlier examples, *app1 workload owner* creates the resources and requests the peering connection with the *shared infrastructure* virtual network. *App1 workload owner* adds only the *managedBy* tag because there is no longer a need for the *environment* tag. That is, resources are for each environment are now grouped in the same *subscription* and the *environment* tag is redundant. The limit counter is decremented to 49 virtual networks remaining.



- Finally, the *subscription owner* repeats the process for the second workload, adding the resource groups with the *app2 workload owner* in the \*contributor role. The limit counter for each of the environment subscriptions is decremented to 48 virtual networks remaining.

This management model has the benefits of the second example above. However, the key difference is that limits are less of an issue due to the fact that they are spread over two *subscriptions*. The drawback is that the cost data tracked by tags must be aggregated across all three *subscriptions*.

Therefore, you can select any of these two examples resource management models depending on the priority of your requirements. If you anticipate that your organization will not reach the service limits for a single subscription, you can use a single subscription with multiple resource groups. Conversely, if your organization anticipates many workloads, multiple subscriptions for each environment may be better.

## Implementing the resource management model

You've learned about several different models for governing access to Azure resources. Now we'll walk through the steps necessary to implement the resource management model with one subscription for each of the **shared infrastructure**, **production**, and **development** environments from the design guide. We'll have one **subscription owner** for all three environments. Each workload will be isolated in a **resource group** with a **workload owner** added with the **contributor** role.

## NOTE

Read [understanding resource access in Azure](#) to learn more about the relationship between Azure Accounts and subscriptions.

Follow these steps:

1. Create an [Azure account](#) if your organization doesn't already have one. The person who signs up for the Azure account becomes the Azure account administrator, and your organization's leadership must select an individual to assume this role. This individual will be responsible for:
  - Creating subscriptions, and
  - Creating and administering [Azure Active Directory \(AD\)](#) tenants that store user identity for those subscriptions.
2. Your organization's leadership team decides which people are responsible for:
  - Management of user identity; an [Azure AD tenant](#) is created by default when your organization's Azure Account is created, and the account administrator is added as the [Azure AD global administrator](#) by default. Your organization can choose another user to manage user identity by [assigning the Azure AD global administrator role to that user](#).
  - Subscriptions, which means these users:
    - Manage costs associated with resource usage in that subscription,
    - Implement and maintain least permission model for resource access, and
    - Keep track of service limits.
  - Shared infrastructure services (if your organization decides to use this model), which means this user is responsible for:
    - On-premises to Azure network connectivity, and
    - Ownership of network connectivity within Azure through virtual network peering.
  - Workload owners.
3. The Azure AD global administrator [creates the new user accounts](#) for:
  - The person who will be the **subscription owner** for each subscription associated with each environment. Note that this is necessary only if the subscription **service administrator** will not be tasked with managing resource access for each subscription/environment.
  - The person who will be the **network operations user**, and
  - The people who are **workload owner(s)**.
4. The Azure account administrator creates the following three subscriptions using the [Azure account portal](#):
  - A subscription for the **shared infrastructure** environment,
  - A subscription for the **production** environment, and
  - A subscription for the **development** environment.
5. The Azure account administrator [adds the subscription service owner to each subscription](#).
6. Create an approval process for **workload owners** to request the creation of resource groups. The approval process can be implemented in many ways, such as over email, or you can use a process management tool such as [Sharepoint workflows](#). The approval process can follow these steps:
  - The **workload owner** prepares a bill of materials for required Azure resources in either the **development** environment, **production** environment, or both, and submits it to the **subscription owner**.
  - The **subscription owner** reviews the bill of materials and validates the requested resources to ensure that the requested resources are appropriate for their planned use - for example, checking that the requested [virtual machine sizes](#) are correct.
  - If the request is not approved, the **workload owner** is notified. If the request is approved, the **subscription owner** [creates the requested resource group](#) following your organization's [naming](#)

conventions, adds the **workload owner** with the **contributor** role and sends notification to the **workload owner** that the resource group has been created.

7. Create an approval process for workload owners to request a virtual network peering connection from the shared infrastructure owner. As with the previous step, this approval process can be implemented using email or a process management tool.

Now that you've implemented your governance model, you can deploy your shared infrastructure services.

## Next steps

[Learn about deploying a basic infrastructure](#)

# Enterprise Cloud Adoption: Deploy a basic workload

9/25/2018 • 3 minutes to read • [Edit Online](#)

The term **workload** is typically understood to define an arbitrary unit of functionality such as an application or service. We think about a workload in terms of the code artifacts that are deployed to a server, but also in terms of any other services that are necessary. This is a useful definition for an on-premises application or service but in the cloud we need to expand on it.

In the cloud, a workload not only encompasses all the artifacts but also includes the cloud resources as well. We include cloud resources as part of our definition because of a concept known as infrastructure-as-code. As you learned in the [how does Azure work?](#), resources in Azure are deployed by an orchestrator service. The orchestrator service exposes this functionality through a web API, and this web API can be called using several tools such as Powershell, the Azure command line interface (CLI), and the Azure portal. This means that we can specify our resources in a machine-readable file that can be stored along with the code artifacts associated with our application.

This enables us to define a workload in terms of code artifacts and the necessary cloud resources, and this further enables us to isolate our workloads. Workloads may be isolated by the way resources are organized, by network topology, or by other attributes. The goal of workload isolation is to associate a workload's specific resources to a team so the team can independently manage all aspects of those resources. This enables multiple teams to share resource management services in Azure while preventing the unintentional deletion or modification of each other's resources.

This isolation also enables another concept known as DevOps. DevOps includes the software development practices that include both software development and IT operations above, but adds the use of automation as much as possible. One of the principles of DevOps is known as continuous integration and continuous delivery (CI/CD). Continuous integration refers to the automated build processes that are run each time a developer commits a code change, and continuous delivery refers to the automated processes that deploy this code to various environments such as a development environment for testing or a production environment for final deployment.

## Basic workload

A **basic workload** is typically defined as a single web application, or a virtual network (VNet) with virtual machine (VM).

### NOTE

This guide does not cover application development. For more information about developing applications on Azure, see the [Azure Application Architecture Guide](#).

Regardless of whether the workload is a web application or a VM, each of these deployments requires a **resource group**. A user with permission to create a resource group must do that before following the steps below.

## Basic web application (PaaS)

For a basic web application, select one of the 5-minute quickstarts from the [web apps documentation](#) and follow the steps.

**NOTE**

Some of the quickstarts will deploy a resource group by default. In this case, it's not necessary to create a resource group explicitly. Otherwise, deploy the web application to the resource group created above.

Once your simple workload has been deployed, you can learn more about the proven practices for deploying a [basic web application](#) to Azure.

## Single Windows or Linux VM (IaaS)

For a simple workload that runs on a virtual machine, the first step is to deploy a virtual network. All IaaS resources in Azure such as virtual machines, load balancers, and gateways require a virtual network. Learn about [Azure virtual networks](#), and then follow the steps to [deploy a Virtual Network to Azure using the portal](#). When you specify the settings for the virtual network in the Azure portal, specify the name of the resource group created above.

The next step is to decide whether to deploy a single Windows or Linux VM. For a Windows VM, follow the steps to [deploy a Windows VM to Azure with the portal](#). Again, when you specify the settings for the virtual machine in the Azure portal, specify the name of the resource group created above.

Once you've followed the steps and deployed the VM, you can learn about [proven practices for running a Windows VM on Azure](#). For a Linux VM, follow the steps to [deploy a Linux VM to Azure with the portal](#). You can also learn more about [proven practices for running a Linux VM on Azure](#).

# Enterprise Cloud Adoption: Operations overview

9/21/2018 • 2 minutes to read • [Edit Online](#)

This section of Azure enterprise cloud adoption covers the topic of **operations**.

Once your enterprise is engaged in a **digital transformation**, a majority of the work done on the design and implementation teams will revolve around migrating existing workloads from on-premises to Azure, developing and testing new cloud-native applications in Azure, and incorporating new innovative Azure services into existing on-premises workloads. However, these are just the first step in the digital transformation. Once these workloads are up and running in Azure, the next step is to **operate** them in the cloud.

Operating in the cloud refers to the IT processes and non-functional requirements necessary to run workloads in Azure as a business. This includes monitoring of workloads, analyzing dependencies for bottlenecks, developing strategies for disaster recovery, and more.

[Establish an operational fitness review](#)

# Establishing an operational fitness review

9/21/2018 • 9 minutes to read • [Edit Online](#)

As your enterprise begins to operate workloads in Azure, the next step is to establish an **operational fitness review** process to enumerate, implement, and iteratively review the **non-functional** requirements for these workloads. *Non-functional* requirements are related to the expected operational behavior of the service. There are five essential categories of non-functional requirements referred to as the [pillars of software quality](#): scalability, availability, resiliency (including business continuity and disaster recovery), management, and security. The purpose of an operational fitness review process is ensuring that your mission critical workloads meet the expectations of your business with respect to the quality pillars.

For this reason, your enterprise should undertake an operational fitness review process to fully understand the issues that result from running the workload in a production environment, determine how to remediate the issues, then resolve them. This article outlines a high-level operational fitness review process that your enterprise can use to achieve this goal.

## Operational fitness at Microsoft

From the outset, the development of the Azure platform has been a continuous development and integration project undertaken by many teams across Microsoft. It would be very difficult to ensure quality and consistency for a project of Azure's size and complexity without a robust process for enumerating and implementing the fundamental non-functional requirements on a regular basis.

These processes followed by Microsoft form the basis for those outlined in this document.

## Understanding the problem

As you learned in [Getting started](#), the first step in an enterprise's digital transformation is identifying the business problems to be solved by adopting Azure. The next step is to determine a high-level solution to the problem, such as migrating a workload to the cloud, or adapting an existing on-premises service to include cloud functionality. Finally, the solution is designed and implemented.

During this process, the focus is often on the *features* of the service. That is, there are a set of desired *functional* requirements for the service to perform. For example, a product delivery service requires features for determining the source and destination locations of the product, tracking the product during delivery, customer notifications, and others.

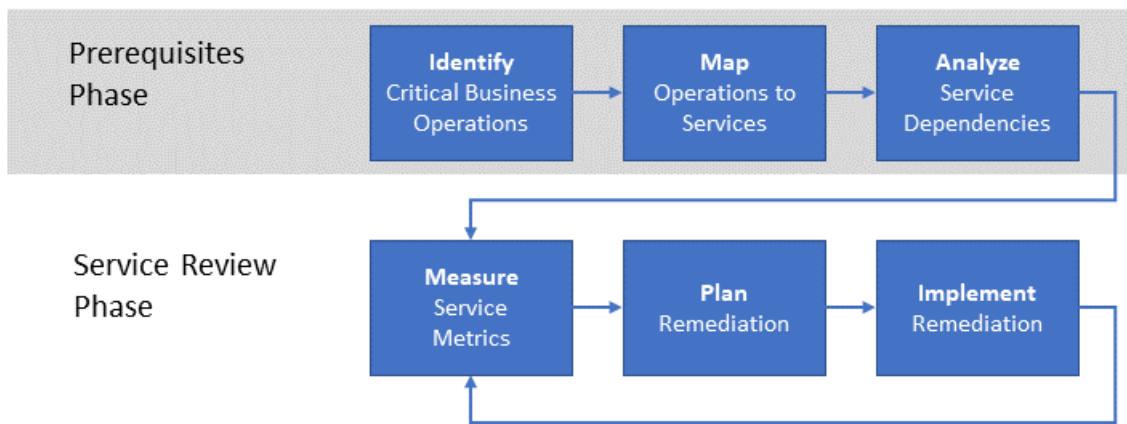
In contrast, the *non-functional* requirements relate to properties such as the service's [availability](#), [resiliency](#), and [scalability](#). These properties differ from the functional requirements because they do not directly affect the final function of any particular feature in the service. However, these non-functional requirements are related to the *performance* and *continuity* of the service.

Some non-functional requirements can be specified in terms of a service level agreement (SLA). For example, with regard to service continuity, an availability requirement for the service can be expressed as a percentage such as **available 99.99% of the time**. Other non-functional requirements may be more difficult to define and may change as production needs evolve. For example, a consumer-facing service might start facing unanticipated throughput requirements after a surge of popularity.

![NOTE] Defining the requirements for resiliency, including explanations of RPO, RTO, SLA, and related concepts, are explored in more depth in [Designing resilient applications for Azure](#).

# Operational fitness review process

The key to maintaining the performance and continuity of an enterprise's services is to implement an *operational fitness review* process.



At a high level, the process has two phases. In the prerequisites phase, the requirements are established and mapped to supporting services. This occurs less frequently; perhaps annually or when new operations are introduced. The output of the prerequisites phase is used in the flow phase. The flow phase occurs more frequently; we recommend monthly.

## Prerequisites phase

The steps in this phase are intended to capture the necessary requirements for conducting a regular review of the important services.

- **Identify critical business operations.** Identify the enterprise's **mission critical** business operations. Business operations are independent from any supporting service functionality. In other words, business operations represent the actual activities that the business needs to perform and are supported by a set of IT services. The term *mission critical*, or alternatively *business critical*, reflects a severe impact to the business if the operation is impeded. For example, an online retailer may have a business operation such as "enable a customer to add an item to a shopping cart" or "process a credit card payment". If either of these operations were to fail, a customer would be unable to complete the transaction and the enterprise would fail to realize sales.
- **Map operations to services.** Map these business operations to the services that support them. In the above shopping cart example, several services may be involved: an inventory stock management service, a shopping cart service, and others. In the credit card payment example above, an on-premises payment service may interact with a third-party payment processing service.
- **Analyze service dependencies.** Most business operations require orchestration between multiple supporting services. It is important to understand the dependences between the services and the flow of mission critical transactions through these services. You should also consider the dependencies between on-premises services and Azure services. In the shopping cart example, the inventory stock management service may be hosted on-premises and ingest data input by employees from a physical warehouse, but it may store data in an Azure service such as [Azure storage](#) or a database such as [Azure Cosmos DB](#).

An output from these activities is a set of **scorecard metrics** for service operations. The metrics are categorized in terms of non-functional criteria such as availability, scalability, and disaster recovery. Scorecard metrics express the criteria that the service is expected to meet operationally. These metrics can be expressed at any level of granularity that is appropriate for the service operation.

The scorecard should be expressed in simple terms to facilitate meaningful discussion between the business owners and engineering. For example, a scalability scorecard metric could be expressed as *green* for performing at

the desired criteria, *yellow* for failing to meet the desired criteria but actively implementing a planned remediation, and *red* for failing to meet the desired criteria with no plan or action.

It is important to emphasize that these metrics should directly reflect business needs.

### Service review phase

The service review phase is core of the operational fitness review process.

- **Measure service metrics.** Using the scorecard metrics, the services should be monitored to ensure that they meet the business expectations. This means that service monitoring is essential. If you are not able to monitor a set of services with respect to the non-functional requirements, then the corresponding scorecard metrics should be considered red. In this case, the first step for remediation is to implement the appropriate service monitoring. For example, if the business expects a service to operate with 99.99% availability, but there is no production telemetry in place to measure the availability, you should assume that you're not meeting the requirement.
- **Plan remediation.** For each service operation with metrics that fall below an acceptable threshold, determine the cost of remediating the service to bring operation to an acceptable metric. If the cost of remediating the service is greater than the expected revenue generation of the service, move on to consider the non-tangible costs such as customer experience. For example, if customers have difficulty placing a successful order using the service, they may choose a competitor instead.
- **Implement remediation.** After the business owners and engineering converge on a plan, it should be implemented. The status of the implementation should be reported whenever scorecard metrics are reviewed.

This process is iterative, and ideally your enterprise should have a team dedicated to owning it. This team should meet regularly to review existing remediation projects, kick off the fundamentals review of new workloads, and track the enterprise's overall scorecard. The team should have the authority to ensure accountability for remediation teams that are behind schedule or fail to meet metrics.

## Structure of the operational fitness review team

The operational fitness review team is composed of the following roles:

1. **Business owner.** This role provides knowledge of the business to identify and prioritize each "mission critical" business operation. This role also compares the mitigation cost to the business impact and drives the final decision on remediation.
2. **Business advocate.** This role is responsible for breaking down business operations into discreet parts and mapping those parts to on-premises and cloud services and infrastructure. The role requires deep knowledge of the technology associated with each business operation.
3. **Engineering owner.** This role is responsible for implementing the services associated with the business operation. These individuals may participate in the design, implementation, and deployment of any solutions for solving non-functional requirement issues uncovered by the operational fitness review team.
4. **Service owner.** This role is responsible for operating the business's applications and services. These individuals collect logging and usage data for these applications and services. This data is used both to identify issues and verify fixes once deployed.

## Operational fitness review meeting

We recommend that your operational fitness review team meet on a regular basis. For example, the could team meet on a monthly cadence and report status and metrics to senior leadership on a quarterly basis.

The details of the process and meeting should be adapted to fit your specific needs. We recommend the following

tasks as a starting point:

1. The business owner and business advocate enumerate and determine the non-functional requirements for each business operation, with input from the engineering and service owners. For business operations that have been previously identified, the priority is reviewed and verified. For new business operations, a priority in the existing list is assigned.
2. The engineering and service owners map the **current state** of business operations to the corresponding on-premises and cloud services. The mapping is composed of a list of the components in each service, oriented as a dependency tree. Once the list and dependency tree are generated, the **critical paths** through the tree are determined.
3. The engineering and service owners review the current state of operational logging and monitoring for the services listed in the previous step. Robust logging and monitoring are critical, in order to identify service components that contribute to failing to meet non-functional requirements. If sufficient logging and monitoring are not in place, a plan must be created and implemented to put them in place.
4. Scorecard metrics are created for new business operation. The scorecard is composed of the list of constituent components for each service identified in step 2, aligned with the non-functional requirements and a metric representing how well the component meets the requirement.
5. For those constituent components that fail to meet non-functional requirements, a high-level solution is designed and an engineering owner is assigned. At this point, the business owner and business advocate should establish a budget for the remediation work, based on the expected revenue of the business operation.
6. Finally, a review is conducted of the ongoing remediation work. Each of the scorecard metrics for work in progress is reviewed against the expected metrics. For constituent components that are meeting metrics, the service owner presents logging and monitoring data to confirm that the metric is met. For those constituent components that are not meeting metrics, each engineering owner explains the issues that are preventing metrics from being reached and any new designs for remediation.

## Recommended resources

- [Pillars of software quality](#). This section of the Azure Application Architecture guide describes the five pillars of software quality: Scalability, availability, resiliency, management, and security.
- [Ten design principles for Azure applications](#). This section of the Azure Application Architecture guide discusses a set of design principles to make your application more scalable, resilient, and manageable.
- [Designing resilient applications for Azure](#). This guide starts with a definition of the term resiliency and related concepts. Then it describes a process for achieving resiliency, using a structured approach over the lifetime of an application, from design and implementation to deployment and operations.
- [Cloud Design Patterns](#). These design patterns are useful for engineering teams when building applications on the pillars of software quality.

# Azure enterprise scaffold: Prescriptive subscription governance

9/28/2018 • 31 minutes to read • [Edit Online](#)

Enterprises are increasingly adopting the public cloud for its agility and flexibility. They utilize the cloud's strengths to generate revenue and optimize resource usage for the business. Microsoft Azure provides a multitude of services and capabilities that enterprises assemble like building blocks to address a wide array of workloads and applications.

Deciding to use Microsoft Azure is only the first step to achieving the benefit of the cloud. The second step is understanding how the enterprise can effectively use Azure and identify the baseline capabilities that need to be in place to address questions like:

- "I'm concerned about data sovereignty; how can I ensure that my data and systems meet our regulatory requirements?"
- "How do I know what every resource is supporting so I can account for it and bill it back accurately?"
- "I want to make sure that everything we deploy or do in the public cloud starts with the mindset of security first, how do I help facilitate that?"

The prospect of an empty subscription with no guard rails is daunting. This blank space can hamper your move to Azure.

This article provides a starting point for technical professionals to address the need for governance and balance it with the need for agility. It introduces the concept of an enterprise scaffold that guides organizations in implementing and managing their Azure environments in a secure way. It provides the framework to develop effective and efficient controls.

## Need for governance

When moving to Azure, you must address the topic of governance early to ensure the successful use of the cloud within the enterprise. Unfortunately, the time and bureaucracy of creating a comprehensive governance system means some business groups go directly to providers without involving enterprise IT. This approach can leave the enterprise open to compromise if the resources are not properly managed. The characteristics of the public cloud - agility, flexibility, and consumption-based pricing - are important to business groups that need to quickly meet the demands of customers (both internal and external). But, enterprise IT needs to ensure that data and systems are effectively protected.

When creating a building, scaffolding is used to create the basis of a structure. The scaffold guides the general outline and provides anchor points for more permanent systems to be mounted. An enterprise scaffold is the same: a set of flexible controls and Azure capabilities that provide structure to the environment, and anchors for services built on the public cloud. It provides the builders (IT and business groups) a foundation to create and attach new services keeping speed of delivery in mind.

The scaffold is based on practices we have gathered from many engagements with clients of various sizes. Those clients range from small organizations developing solutions in the cloud to large multi-national enterprises and independent software vendors who are migrating workloads and developing cloud-native solutions. The enterprise scaffold is "purpose-built" to be flexible to support both traditional IT workloads and agile workloads; such as, developers creating software-as-a-service (SaaS) applications based on Azure platform capabilities.

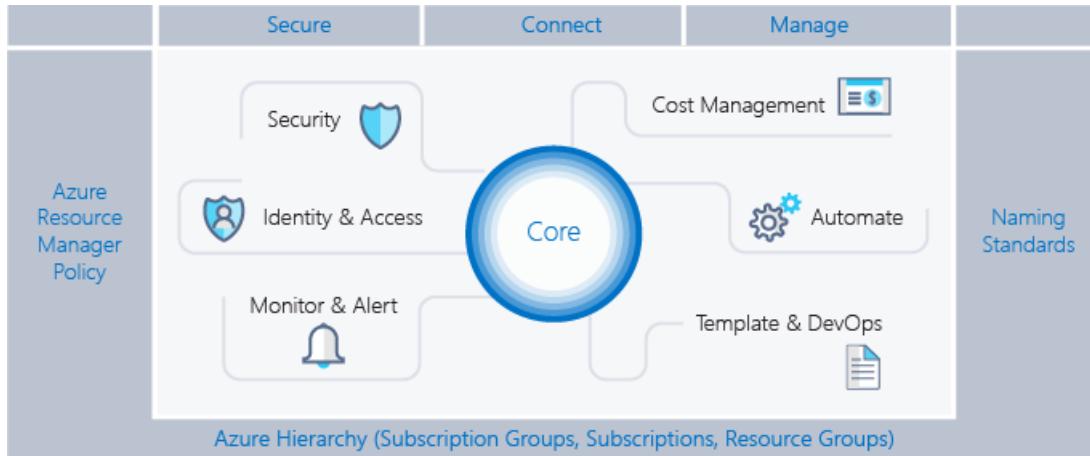
The enterprise scaffold is intended to be the foundation of each new subscription within Azure. It enables administrators to ensure workloads meet the minimum governance requirements of an organization without

preventing business groups and developers from quickly meeting their own goals. Our experience shows that this greatly speeds, rather than impedes, public cloud growth.

#### NOTE

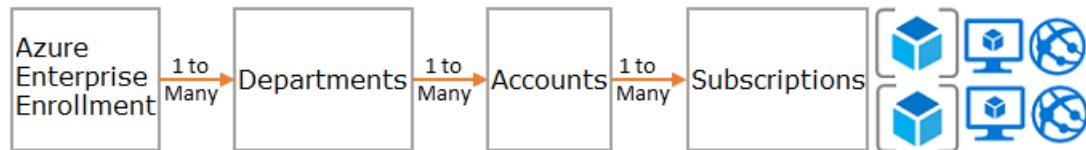
Microsoft has released into preview a new capability called [Azure Blueprints](#) that will enable you to package, manage, and deploy common images, templates, policies, and scripts across subscriptions and management groups. This capability is the bridge between the scaffold's purpose as reference model and deploying that model to your organization.

The following image shows the components of the scaffold. The foundation relies on a solid plan for the management hierarchy and subscriptions. The pillars consist of Resource Manager policies and strong naming standards. The rest of the scaffold are core Azure capabilities and features that enable and connect a secure and manageable environment.



## Define your hierarchy

The foundation of the scaffold is the hierarchy and relationship of the Azure Enterprise Enrollment through to subscriptions and resource groups. The enterprise enrollment defines the shape and use of Azure services within your company from a contractual point of view. Within the enterprise agreement, you can further subdivide the environment into departments, accounts, and finally, subscriptions and resource groups to match your organization's structure.



An Azure subscription is the basic unit where all resources are contained. It also defines several limits within Azure, such as number of cores, virtual networks and other resources. Azure Resource Groups are used to further refine the subscription model and enable a more natural grouping of resources.

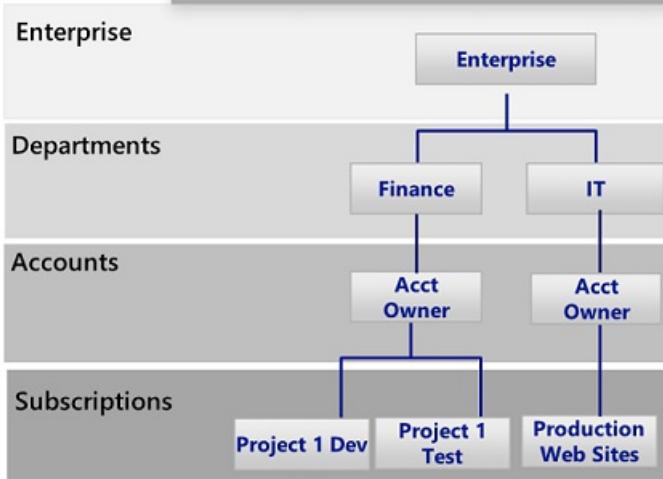
Every enterprise is different and the hierarchy in the above image allows for significant flexibility in how Azure is organized within your company. Modeling your hierarchy to reflect the needs of your company for billing, resource management, and resource access is the first — and most important — decision you make when starting in the public cloud.

### Departments and Accounts

The three common patterns for Azure Enrollments are:

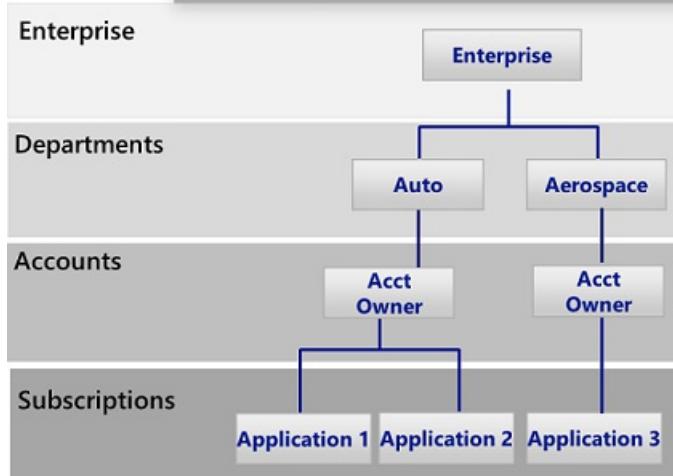
- The **functional** pattern

## Functional



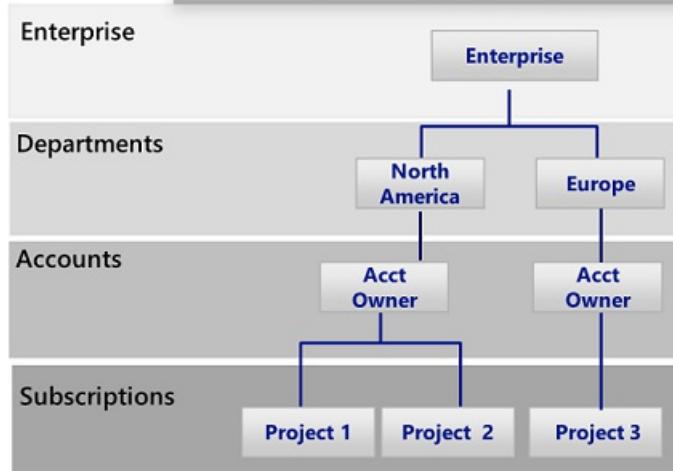
- The **business unit** pattern

## Business Division



- The **geographic** pattern

## Geographic



Though each of these patterns has its place, the **business unit** pattern is increasingly being adopted for its flexibility in modeling an organization's cost model as well as reflecting span of control. Microsoft Core Engineering and Operations group has created a sub-set of the **business unit** pattern that is very effective, modeled on **Federal**, **State**, and **Local**. (For more information, see [Organizing subscriptions and resource groups within the Enterprise](#).)

## Management Groups

Microsoft has recently released a new way of modeling your hierarchy: [Azure management groups](#). Management groups are much more flexible than departments and accounts and can be nested up to six levels. Management groups allow you to create a hierarchy that is separate from your billing hierarchy, solely for efficient management of resources. Management groups can mirror your billing hierarchy and often enterprises start that way. However, the power of management groups is when you use them to model your organization where related subscriptions — regardless where they are in the billing hierarchy — are grouped together and need common roles assigned as well as policies and initiatives. A few examples:

- **Production/Non-Production.** Some enterprises create management groups to identify their production and non-production subscriptions. Management groups allow these customers to more easily manage roles and policies, for example: non-production subscription may allow developers "contributor" access, but in production, they have only "reader" access.
- **Internal Services/External Services.** Much like Production/Non-Production, enterprises often have different requirements, policies and roles for internal services vs external (customer facing) services.

Well thought out management groups are, along with Azure Policy and Initiatives the backbone of efficient governance of Azure.

## Subscriptions

When deciding on your Departments and Accounts (or management groups), you are primarily looking at how you're dividing up your Azure environment to match your organization. Subscriptions, however, are where the real work happens and your decisions here impact security, scalability and billing. Many organizations look at the following patterns as their guides:

- **Application/Service:** Subscriptions represent an application or a service (portfolio of applications)
- **Lifecycle:** Subscriptions represent a lifecycle of a service, such as Production or Development.
- **Department:** Subscriptions represent departments in the organization.

The first two patterns are the most commonly used, and both are highly recommended. The Lifecycle approach is appropriate for most organizations. In this case, the general recommendation is to use two base subscriptions. "Production" and "Non-Production," and then use resource groups to break out the environments further.

## Resource groups

Azure Resource Manager enables you to put resources into meaningful groups for management, billing, or natural affinity. Resource groups are containers of resources that have a common life cycle or share an attribute such as "all SQL servers" or "Application A".

Resource groups can't be nested, and resources can only belong to one resource group. Some actions can act on all resources in a resource group. For example, deleting a resource group removes all resources within the resource group. Like subscriptions, there are common patterns when creating resource groups and will vary from "Traditional IT" workloads to "Agile IT" workloads:

- "Traditional IT" workloads are most commonly grouped by items within the same life cycle, such as an application. Grouping by application allows for individual application management.
- "Agile IT" workloads tend to focus on external customer-facing cloud applications. The resource groups often reflect the layers of deployment (such as Web Tier, App Tier) and management.

### NOTE

Understanding your workload helps you develop a resource group strategy. These patterns can be mixed and matched. For example, a shared services resource group in the same subscription as "Agile" resource groups.

## Naming standards

The first pillar of the scaffold is a consistent naming standard. Well-designed naming standards enable you to identify resources in the portal, on a bill, and within scripts. You likely already have existing naming standards for on-premises infrastructure. When adding Azure to your environment, you should extend those naming standards to your Azure resources.

### TIP

For naming conventions:

- Review and adopt where possible the [Patterns and Practices guidance](#). This guidance helps you decide on a meaningful naming standard and provides extensive examples.
- Using Resource Manager Policies to help enforce naming standards

Remember that it's difficult to change names later, so a few minutes now will save you trouble later.

Concentrate your naming standards on those resources that are more commonly used and searched for. For example, all resource groups should follow a strong standard for clarity.

### Resource Tags

Resource tags are tightly aligned with naming standards. As resources are added to subscriptions, it becomes increasingly important to logically categorize them for billing, management, and operational purposes. For more information, see [Use tags to organize your Azure Resource](#).

### IMPORTANT

Tags can contain personal information and may fall under the regulations of GDPR. Plan for management of your tags carefully. If you're looking for general info about GDPR, see the GDPR section of the [Service Trust Portal](#).

Tags are used in many ways beyond billing and management. They are often used as part of automation (see later section). This can cause conflicts if not considered up front. The recommended practice is to identify all the common tags at the enterprise level (such as ApplicationOwner, CostCenter) and apply them consistently when deploying resources using automation.

## Azure Policy and Initiatives

The second pillar of the scaffold involves using [Azure Policy and Initiatives](#) to manage risk by enforcing rules (with effects) over the resources and services in your subscriptions. Azure Initiatives are collections of policies that are designed to achieve a single goal. Azure policy and initiatives are then assigned to a resource scope to begin enforcement of the particular policies. <IMG of Initiatives/Policies/Assignments>

Azure Policy and Initiatives are even more powerful when used with the management groups mentioned earlier. Management groups enable the assignment of an initiative or policy to an entire set of subscriptions.

### Common uses of Resource Manager policies

Azure policies and initiatives are a powerful tool in the Azure toolkit. Policies allow companies to provide controls for "Traditional IT" workloads that enable the stability that is needed for LOB applications while also allowing "Agile" workloads; such as, developing customer applications without opening up the enterprise to additional risk. The most common patterns we see for policies are:

- **Geo-compliance/data sovereignty.** Azure has an ever-growing list of regions across the world. Enterprises often need to ensure that resources in a particular scope remain in a geographic region to address regulatory requirements.

- **Avoid exposing servers publicly.** Azure policy can prohibit the deployment of certain resources types. A common use is to create a policy to deny the creation of a public IP within a particular scope, avoiding unintended exposure of a server to the internet.
- **Cost Management and Metadata.** Resource tags are often used to add important billing data to resources and resource groups such as CostCenter, Owner and more. These tags are invaluable for accurate billing and management of resources. Policies can enforce the application of resources tags to all deployed resource, making it easier to manage.

### Common uses of initiatives

The introduction of initiatives provided enterprises a way to group logical policies together and track as a whole. Initiatives further support the enterprise to address the needs of both "agile" and "traditional" workloads. We have seen very creative uses of initiatives, but commonly we see:

- **Enable monitoring in Azure Security Center.** This is a default initiative in the Azure Policy and an excellent example of what initiative are. It enables policies that identify un-encrypted SQL databases, VM vulnerabilities and more common security related needs.
- **Regulatory specific initiative.** Enterprises often group policies common to a regulatory requirement (such as HIPAA) so that controls and compliancy to those controls are tracked efficiently.
- **Resource Types & SKUs.** Creating an initiative that restricts the types of resources that can be deployed as well as the SKUs that can be deployed can help to control costs and ensure your organization is only deploying resources that your team have the skillset and procedures to support.

#### TIP

We recommend you always use initiative definitions instead of policy definitions. After assigning an initiative to a scope, such as subscription or management group, you can easily add another policy to the initiative without having to change any assignments. This makes understanding what is applied and tracking compliance far easier.

### Policy and Initiative assignments

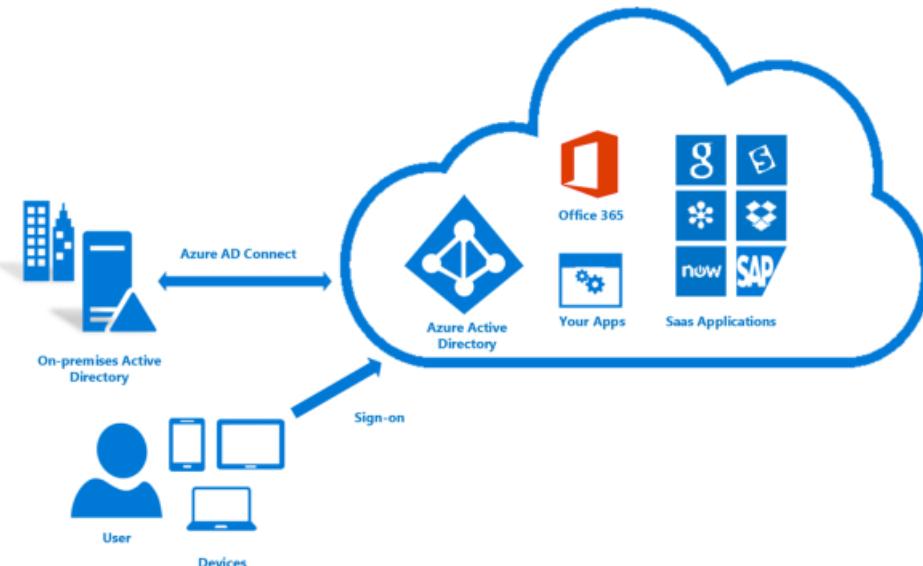
After the creation of policies and grouping them into logical initiatives you must assign the policy to a scope, whether it is a management group, a subscription or even a resource group. Assignments allow you to also exclude a sub-scope from the assignment of a policy. For example, if you deny the creation of public IPs within a subscription, you could create an assignment with an exclusion for a resource group connected to your protected DMZ.

You will find several Policy examples that show how Policy and Initiatives can be applied to various resources within Azure on this [GitHub](#) repository.

## Identity and access management

One of the first, and most crucial, questions you ask yourself when starting with the public cloud is "who should have access to resources?" and "how do I control this access?" Allowing or disallowing access to the Azure portal, and controlling access to resources in the portal is critical to the long term success and safety of your assets in the cloud.

To accomplish the task of securing access to your resources you will first configure your identity provider and then configure Roles and access. Azure Active Directory (Azure AD), connected to your on-premises Active Directory, is the foundation of Azure Identity. That said, Azure AD is *not* Active Directory and it's important to understand what an Azure AD tenant is and how it relates to your Azure enrollment. Review the available [information](#) to gain a solid foundation on Azure AD and AD. To connect and synchronize your Active Directory to Azure AD, install and configure the [AD Connect tool](#) on-premises.



When Azure was initially released, access controls to a subscription were basic: Administrator or Co-Administrator. Access to a subscription in the Classic model implied access to all the resources in the portal. This lack of fine-grained control led to the proliferation of subscriptions to provide a level of reasonable access control for an Azure Enrollment. This proliferation of subscriptions is no longer needed. With role-based access control (RBAC), you can assign users to standard roles that provide common access such as "owner", "contributor" or "reader" or even create your own roles

When implementing role-based access, the following are highly recommended:

- Control the Administrator/Co-Administrator of a subscription as these roles have extensive permissions. You only need to add the Subscription Owner as a Co-administrator if they need to manage Azure Classic deployments.
- Use Management Groups to assign [roles](#) across multiple subscriptions and reduce the burden of managing them at the subscription level.
- Add Azure users to a group (for example, Application X Owners) in Active Directory. Use the synced group to provide group members the appropriate rights to manage the resource group containing the application.
- Follow the principle of granting the **least privilege** required to do the expected work.

#### IMPORTANT

Consider using [Azure AD Privileged Identity Management](#), [Azure Multi-Factor Authentication](#) and [Conditional Access](#) capabilities to provide better security and more visibility to administrative actions across your Azure subscriptions. These capabilities come from a valid Azure AD Premium license (depending on the feature) to further secure and manage your identity. Azure AD PIM enables "Just-in-Time" administrative access with approval workflow, as well as a full audit of administrator activations and activities. Azure MFA is another critical capability and enables two-step verification for login to the Azure portal. When combined with Conditional Access Controls you can effectively manage your risk of compromise.

Planning and preparing for your identity and access controls and following Azure Identity Management best practice ([link](#)) is one of the best risk mitigation strategies that you can employ and should be considered mandatory for every deployment.

## Security

One of the biggest blockers to cloud adoption traditionally has been concerns over security. IT risk managers and

security departments need to ensure that resources in Azure are protected and secure by default. Azure provides a number of capabilities that you can leverage to protect resources and detect/prevent threats against those resources.

## Azure Security Center

The [Azure Security Center](#) provides a unified view of the security status of resources across your environment in addition to advanced threat protection. Azure Security Center is an open platform that enables Microsoft partners to create software that plugs into and enhance its capabilities. The baseline capabilities of Azure Security Center (free tier) provides assessment and recommendations that will enhance your security posture. Its paid tiers enable additional and valuable capabilities such as Just In Time admin access and adaptive application controls (whitelisting).

### TIP

Azure security center is a very powerful tool that is constantly being enhanced and incorporating new capabilities you can leverage to detect threats and protect your enterprise. It is highly recommended to always enable ASC.

## Azure resource locks

As your organization adds core services to subscriptions it becomes increasingly important to avoid business disruption. One type of disruption that we often see is unintended consequences of scripts and tools working against an Azure subscription deleting resources mistakenly. [Resource Locks](#) enable you to restrict operations on high-value resources where modifying or deleting them would have a significant impact. Locks are applied to a subscription, resource group, or even individual resources. The common use case is to apply locks to foundational resources such as virtual networks, gateways, network security groups and key storage accounts.

## Secure DevOps Toolkit

The "Secure DevOps Kit for Azure" (AzSK) is a collection of scripts, tools, extensions, automations, etc. originally created by Microsoft's own IT Team and released in OpenSource via Github ([link](#)). AzSK caters to the end to end Azure subscription and resource security needs for teams using extensive automation and smoothly integrating security into native dev ops workflows helping accomplish secure dev ops with these 6 focus areas:

- Secure the subscription
- Enable secure development
- Integrate security into CICD
- Continuous Assurance
- Alerting & Monitoring
- Cloud Risk Governance



## Secure DevOps Kit for Azure



The AzSK is a rich set of tools, scripts and information that are an important part of a full Azure governance plan and incorporating this into your scaffold is crucial to supporting your organizations risk management goals

### Azure Update Management

One of the key tasks you can do to keep your environment safe is ensure that your servers are patched with the latest updates. While there are many tools to accomplish this, Azure provides the [Azure Update Management](#) solution to address the identification and rollout of critical OS patches. It makes use of Azure Automation (which is covered in the [Automate](#) section, later in this guide).

## Monitor and alerts

Collecting and analyzing telemetry that provides line of sight into the activities, performance metrics, health and availability of the services you are using across all of your Azure subscriptions is critical to proactively manage your applications and infrastructure and is a foundational need of every Azure subscription. Every Azure service emits telemetry in the form of Activity Logs, Metrics and Diagnostic Logs.

- **Activity Logs** describe all operations performed on resources in your subscriptions
- **Metrics** are numerical information emitted from within a resource that describe the performance and health of a resource
- **Diagnostic Logs** are emitted by an Azure service and provide rich, frequent data about the operation of that service.

This information can be viewed and acted upon at multiple levels and are continually being improved. Azure provides **shared**, **core** and **deep** monitoring capabilities of Azure resources through the services outlined in the

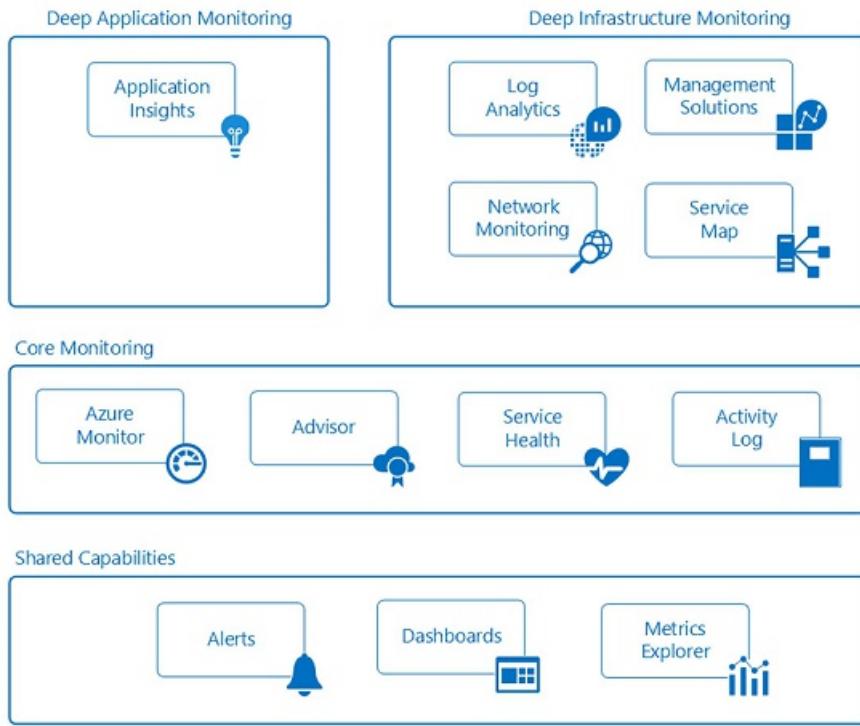


diagram below.

### Shared capabilities

- **Alerts:** You can collect every log, event and metric from Azure resources, but without the ability to be notified of critical conditions and act, this data is only useful for historic purposes and forensics. Azure Alerts proactively notify you of conditions you define across all your applications and infrastructure. You create alert rules across logs, events and metrics that use action groups to notify sets of recipients. Action groups also provide the ability to automate remediation using external actions such as webhooks to run Azure Automation runbooks and Azure Functions.
- **Dashboards:** Dashboards enable you to aggregate monitoring views and combine data across resources and subscriptions to give you an enterprise-wide view into the telemetry of Azure resources. You can create and configure your own views and share them with others. For example, you could create a dashboard consisting of various tiles for DBAs to provide information across all Azure database services, including Azure SQL DB, Azure DB for PostgreSQL and Azure DB for MySQL.
- **Metrics Explorer:** Metrics are numerical values generated by Azure resources (e.g. % CPU, Disk I/O, that provide insight into the operation and performance of your resources. By using Metrics Explorer you can define and send the metrics in which you are interested to Log Analytics for aggregation and analysis.

### Core monitoring

- **Azure Monitor:** Azure Monitor is the core platform service that provides a single source for monitoring Azure resources. The Azure Portal interface of Azure Monitor provides a centralized jump off point for all the monitoring features across Azure including the deep monitoring capabilities of Application Insights, Log Analytics, Network Monitoring, Management Solutions and Service Maps. With Azure Monitor you can visualize, query, route, archive and act on the metrics and logs coming from Azure resources across your entire cloud estate. In addition to the portal you can retrieve data through the Monitor PowerShell Cmdlets, Cross Platform CLI or the Azure Monitor REST APIs.
- **Azure Advisor:** Azure Advisor constantly monitors telemetry across your subscriptions and environments and provides recommendations on best practices on how to optimize your Azure resources to save money and improve performance, security and availability of the resources that make up your applications.
- **Service Health:** Azure Service Health identifies any issues with Azure Services that may impact your applications as well as assists you in planning for scheduled maintenance windows.
- **Activity Log:** The Activity Log describes all operations on resources in your subscriptions. It provides an

audit trail to determine the 'what', 'who', and 'when' of any create, update, delete operation on resources. Activity Log events are stored in the platform and are available to query for 90 days. You can ingest Activity Logs into Log Analytics for longer retention periods and deeper querying and analysis across multiple resources.

## Deep application monitoring

- **Application Insights:** Application Insights enables you to collect application specific telemetry and monitor the performance, availability and usage of applications in the cloud or on-premises. By instrumenting your application with supported SDKs for multiple languages including .NET, JavaScript, JAVA, Node.js, Ruby and Python. Application Insights events are ingested into the same Log Analytics data store that supports infrastructure and security monitoring to enable you to correlate and aggregate events over time through a rich query language.

## Deep infrastructure monitoring

- **Log Analytics:** Log Analytics plays a central role in Azure monitoring by collecting telemetry and other data from a variety of sources and providing a query language and analytics engine that gives you insights into the operation of your applications and resources. You can either interact directly with Log Analytics data through highly performant log searches and views, or you may use analysis tools in other Azure services that store their data in Log Analytics such as Application Insights or Azure Security Center.
- **Network Monitoring:** Azure's network monitoring services enable you to gain insight into network traffic flow, performance, security, connectivity and bottlenecks. A well-planned network design should include configuring Azure network monitoring services such as Network Watcher and ExpressRoute Monitor.
- **Management Solutions:** Management solutions are packaged sets of logic, insights and pre-defined Log Analytics queries for an application or service. They rely on Log Analytics as the foundation to store and analyze event data. Sample management solutions include monitoring containers and Azure SQL Database analytics.
- **Service Map:** Service Map provides a graphical view into your infrastructure components, their processes and interdependencies on other computers and external processes. It integrates events, performance data and management solutions in Log Analytics.

### TIP

Before creating individual alerts, create and maintain a set of shared Action Groups that can be used across Azure Alerts. This will enable you to centrally maintain the lifecycle of your recipient lists, notification delivery methods (email, SMS phone numbers) and webhooks to external actions (Azure Automation runbooks, Azure Functions / Logic Apps, ITSM).

## Cost management

One of the major changes that you will face when you move from on-premises cloud to the public cloud is the switch from capital expenditure (buying hardware) to operating expenditure (paying for service as you use it). This switch from CAPEX to OPEX also brings the need to more carefully manage your costs. The benefit of the cloud is that you can fundamentally and positively impact the cost of a service you use by merely turning it off (or resizing) when it's not needed. Deliberately managing your costs in the cloud is a recommended practice and one that mature customers do daily.

Microsoft provides several tools for you to be able to visual, track and manage your costs. We also provide a full set of APIs to enable you to customize and integrate cost management into your own tools and dashboards. These tools are loosely grouped into: Azure Portal Capabilities and external capabilities

### Azure Portal capabilities

These are tools to provide you instant information on cost as well as the ability to take actions

- **Subscription Resource Cost:** Located in The Portal, the [Azure Cost Analysis](#) view provides a quick look at your costs and information on daily spend by resource or resource group.
- **Azure Cost Management:** This product is the result of the purchase of Cloudyn by Microsoft and allows you to manage and analyze your Azure spend as well what you spend on other Public Cloud providers. There are both free and paid tiers, with a great wealth of capabilities as seen in the [overview](#).
- **Azure Budgets and Action Groups** Knowing what somethings costs and doing something about it until recently has been more of a manual exercise. With the introduction of Azure Budgets and its APIs, it's now possible to create actions (as seen in [this example](#)) when costs hit a threshold. For example, shutting down a "test" resource group when it hits 100% of its budget, or [\[another example\]](#).
- **Azure Advisor** Knowing what something costs is only half the battle; the other half is knowing what to do with that information. [Azure Advisor](#) provides you recommendations on actions to take to save money, improve reliability or even increase security.

#### External cost management tools

- **PowerBI Azure Consumption Insights.** Do you want to create your own visualizations for your organization? If so, then the Azure Consumption Insights content pack for PowerBI is your tool of choice. Using this content pack and PowerBI you can create custom visualizations to represent your organization, do deeper analysis on costs and add in other data sources for further enrichment.
- **Consumption API.** The [consumption APIs](#) give you programmatic access to cost and usage data in addition to information on budgets, reserved instances and marketplace charges. These APIs are accessible only for Enterprise Enrollments and some Web Direct subscriptions however they give you the ability to integrate your cost data into your own tools and data warehouses. You can also access these APIs by using the Azure CLI, seen [here](#).

When we look across customers who have used the cloud for a long time and are "mature" in their use, we see a number of highly recommended practices

- **Actively monitor costs.** Organizations that are mature Azure users constantly monitor costs and take actions when needed. Some organizations even dedicate people to do analysis and suggest changes to usage, and these people more than pay for themselves the first time they find an unused HDInsight cluster that's been running for months.
- **Use Reserved Instances.** Another key tenant for managing costs in the cloud is to use the right tool for the job. If you have an IaaS VM that must stay on 24x7, then using a Reserved Instance will save you significant money. Finding the right balance between automating the shutdown of VMs and using RIs takes experience and analysis.
- **Use automation effectively:** Many workloads do not need to be running every day. Even turning off a VM for a 4-hour period every day can save you 15% of your cost. Automation will pay for itself quickly.
- **Use resource tags for visibility:** As mentioned elsewhere in this document, using resource tags will allow for better analysis of costs.

Cost management is a discipline that is core to the effective and efficient running of a public cloud. Enterprises that achieve success will be able to control their costs and match them to their actual demand as opposed to overbuying and hoping demand comes.

## Automate

One of the many capabilities that differentiates the maturity of organizations using cloud providers is the level of automation that they have incorporated. Automation is a never-ending process and as your organization moves to the cloud it is any area that you need to invest resources and time in building. Automation serves many purposes including consistent rollout of resources (where it ties directly to another core scaffold concept, Templates & DevOps) to the remediation of issues. Automation is the "connective tissue" of the Azure scaffold and links each area together.

There are a number of tools that are available as you build out this capability, from first party tools such as Azure Automation, EventGrid and AzureCLI to an extensive amount of third party tools such as Terraform, Jenkins, Chef, and Puppet (to name a few). Core to your operations team ability to automate are Azure Automation, Event Grid and the Azure Cloud Shell:

- **Azure Automation:** Is a cloud-based capability that allows you to author Runbooks (in either PowerShell or Python) and allows you automate processes, configure resources, and even apply patches. [Azure Automation](#) has an extensive set of cross platform capabilities that are integral to your deployment but are too extensive to be covered in depth here.
- **Event Grid:** this [service](#) is a fully-managed event routing system that lets you react to events within your Azure environment. Like Automation is the connective tissue of mature cloud organizations, Event Grid is the connective tissue of good automation. Using Event Grid, you can create a simple, serverless, action to send an email to an administrator whenever a new resource is created and log that resource in a database. That same Event Grid can notify when a resource is deleted and remove the item from the database.
- **Azure Cloud Shell:** is an interactive, browser-based [shell](#) for managing resources in Azure. It provides a complete environment for either PowerShell or Bash that is launched as needed (and maintained for you) so that you have a consistent environment from which to run your scripts. The Azure Cloud Shell provides access to additional key tools -already installed-- to automate your environment including [Azure CLI](#), [Terraform](#) and a growing list of additional [tools](#) to manage containers, databases (sqlcmd) and more.

Automation is a full-time job and it will rapidly become one of the most important operational tasks within your cloud team. Organizations that take the approach of "automate first" have greater success in using Azure:

- Managing costs: actively seeking opportunities and creating automation to re-size resources, scale-up/down and turn off unused resources.
- Operational flexibility: through the use of automation (along with Templates and DevOps) you gain a level of repeatability that increases availability, increases security and enables your team to focus on solving business problems.

## Templates and DevOps

As highlighted in the Automate section, your goal as an organization should be to provision resources through source-controlled templates and scripts and to minimize interactive configuration of your environments. This approach of "infrastructure as code" along with a disciplined DevOps process for continuous deployment can ensure consistency and reduce drift across your environments. Almost every Azure resource is deployable through [Azure Resource Manager \(ARM\) JSON templates](#) in conjunction with PowerShell or the Azure cross platform CLI and tools such as Terraform from Hashicorp (which has first class support and integrated into the Azure Cloud Shell).

Article such as [this one](#) provide an excellent discussion on best practices and lessons learned in applying a DevOps approach to ARM templates with the [Azure DevOps](#) tool chain. Take the time and effort to develop a core set of templates specific to your organization's requirements, and to develop continuous delivery pipelines with DevOps tool chains (Azure DevOps, Jenkins, Bamboo, Teamcity, Concourse), especially for your production and QA environments. There is a large library of [Azure Quick Start templates](#) on GitHub that you can use as a starting point for templates, and you can quickly create cloud-based delivery pipelines with Azure DevOps.

As a best practice for production subscriptions or resource groups, your goal should be utilizing RBAC security to disallow interactive users by default and utilizing automated continuous delivery pipelines based on service principals to provision all resources and deliver all application code. No admin or developer should touch the Azure Portal to interactively configure resources. This level of DevOps takes a concerted effort and utilizes all the concepts of the Azure scaffold and provides a consistent and more secure environment that will meet your organizations to grow scale.

#### TIP

When designing and developing complex ARM templates, use [linked templates](#) to organize and refactor complex resource relationships from monolithic JSON files. This will enable you to manage resources individually and make your templates more readable, testable and reusable.

Azure is a hyper scale cloud provider and as you move your organization from the world of on-premises servers to the cloud, utilizing the same concepts that cloud providers and SaaS applications use will provide your organization to react to the needs of the business in vastly more efficient way.

## Core network

The final component of the Azure scaffold reference model is core to how your organization accesses Azure, in a secure manner. Access to resources can be either internal (within the corporation's network) or external (through the internet). It is easy for users in your organization to inadvertently put resources in the wrong spot, and potentially open them to malicious access. As with on-premises devices, enterprises must add appropriate controls to ensure that Azure users make the right decisions. For subscription governance, we identify core resources that provide basic control of access. The core resources consist of:

- **Virtual networks** are container objects for subnets. Though not strictly necessary, it is often used when connecting applications to internal corporate resources.
- **User Defined Routes** allow you to manipulate the route table within a subnet enabling you to send traffic through a network virtual appliance or to a remote gateway on a peered virtual network.
- **Virtual Network Peering** enables you to seamlessly connect two or more Azure virtual networks, creating more complex hub & spoke designs or shared services networks.
- **Service Endpoints**. In the past, PaaS services relied on different methods to secure access to those resources from your virtual networks. Service endpoints allow you to secure access to enabled PaaS services from ONLY connected endpoints, increasing overall security.
- **Security groups** are an extensive set of rules which provide you the ability to allow or deny inbound and outbound traffic to/from Azure Resources. [Security Groups](#) consist of Security Rules, which can be augmented with **Service Tags** (which define common Azure services such as AzureKeyVault, Sql and others) and **Application Groups** (which define application structure, such as WebServers, AppServers and such)

#### TIP

Use Service tags and Application groups in your network security groups to not only enhance the readability of your rules - which is crucial to understanding impact- but also to enable effective micro-segmentation within a larger subnet, reducing sprawl and increasing flexibility.

## Virtual Data Center

Azure provides you both internal capabilities and third-party capabilities from our extensive partner network that enable you to have an effective security stance. More importantly, Microsoft provides best practices and guidance in the form of the [Azure Virtual Data Center](#). As you move from a single workload to multiple workloads which leverage hybrid capabilities, the VDC guidance will provide you with "recipe" to enable a flexible, network that will grow as your workloads in Azure grow.

## Next steps

Governance is crucial to the success of Azure. This article targets the technical implementation of an enterprise scaffold but only touches on the broader process and relationships between the components. Policy governance flows from the top down and is determined by what the business wants to achieve. Naturally, the creation of a governance model for Azure includes representatives from IT, but more importantly it should have strong

representation from business group leaders, and security and risk management. In the end, an enterprise scaffold is about mitigating business risk to facilitate an organization's mission and objectives.

Now that you have learned about subscription governance, it's time to see these recommendations in practice. See [Examples of implementing Azure subscription governance](#).

# Examples of implementing Azure enterprise scaffold

10/3/2018 • 9 minutes to read • [Edit Online](#)

This article provides examples of how an enterprise can implement the recommendations for an [Azure enterprise scaffold](#). It uses a fictional company named Contoso to illustrate best practices for common scenarios.

## Background

Contoso is a worldwide company that provides supply chain solutions for customers. They provide everything from a Software as a Service model to a packaged model deployed on-premises. They develop software across the globe with significant development centers in India, the United States, and Canada.

The ISV portion of the company is divided into several independent business units that manage products in a significant business. Each business unit has its own developers, product managers, and architects.

The Enterprise Technology Services (ETS) business unit provides centralized IT capability, and manages several data centers where business units host their applications. Along with managing the data centers, the ETS organization provides and manages centralized collaboration (such as email and websites) and network/telephony services. They also manage customer-facing workloads for smaller business units who don't have operational staff.

The following personas are used in this article:

- Dave is the ETS Azure administrator.
- Alice is Contoso's Director of Development in the supply chain business unit.

Contoso needs to build a line-of-business app and a customer-facing app. It has decided to run the apps on Azure. Dave reads the [prescriptive subscription governance](#) article, and is now ready to implement the recommendations.

## Scenario 1: line-of-business application

Contoso is building a source code management system (BitBucket) to be used by developers across the world. The application uses Infrastructure as a Service (IaaS) for hosting, and consists of web servers and a database server. Developers access servers in their development environments, but they don't need access to the servers in Azure. Contoso ETS wants to allow the application owner and team to manage the application. The application is only available while on Contoso's corporate network. Dave needs to set up the subscription for this application. The subscription will also host other developer-related software in the future.

### Naming standards & resource groups

Dave creates a subscription to support developer tools that are common across all the business units. Dave needs to create meaningful names for the subscription and resource groups (for the application and the networks). He creates the following subscription and resource groups:

| ITEM           | NAME                                  | DESCRIPTION   |
|----------------|---------------------------------------|---|
| Subscription   | Contoso ETS DeveloperTools Production | Supports common developer tools                                   |
| Resource Group | bitbucket-prod-rg                     | Contains the application web server and database server           |
| Resource Group | corenetworks-prod-rg                  | Contains the virtual networks and site-to-site gateway connection |

## Role-based access control

After creating his subscription, Dave wants to ensure that the appropriate teams and application owners can access their resources. Dave recognizes that each team has different requirements. He utilizes the groups that have been synced from Contoso's on-premises Active Directory (AD) to Azure Active Directory, and provides the right level of access to the teams.

Dave assigns the following roles for the subscription:

| ROLE                | ASSIGNED TO                             | DESCRIPTION   |
|---------------------|---|---|
| Owner               | Managed ID from Contoso's AD            | This ID is controlled with Just in Time (JIT) access through Contoso's Identity Management tool and ensures that subscription owner access is fully audited     |
| Security Reader     | Security and risk management department | This role allows users to look at the Azure Security Center and the status of the resources   |
| Network Contributor | Network team                            | This role allows Contoso's network team to manage the Site to Site VPN and the Virtual Networks   |
| Custom role         | Application owner                       | Dave creates a role that grants the ability to modify resources within the resource group. For more information, see <a href="#">Custom Roles in Azure RBAC</a> |

## Policies

Dave has the following requirements for managing resources in the subscription:

- Because the development tools support developers across the world, he doesn't want to block users from creating resources in any region. However, he needs to know where resources are created.
- He is concerned with costs. Therefore, he wants to prevent application owners from creating unnecessarily expensive virtual machines.
- Because this application serves developers in many business units, he wants to tag each resource with the business unit and application owner. By using these tags, ETS can bill the appropriate teams.

He creates the following [Azure policies](#):

| FIELD    | EFFECT | DESCRIPTION   |
|----------|--------|---|
| location | audit  | Audit the creation of the resources in any region                             |
| type     | deny   | Deny creation of G-Series virtual machines                                    |
| tags     | deny   | Require application owner tag   |
| tags     | deny   | Require cost center tag   |
| tags     | append | Append tag name <b>BusinessUnit</b> and tag value <b>ETS</b> to all resources |

## Resource tags

Dave understands that he needs to have specific information on the bill to identify the cost center for the BitBucket implementation. Additionally, Dave wants to know all the resources that ETS owns.

He adds the following [tags](#) to the resource groups and resources.

| TAG NAME         | TAG VALUE   |
|------------------|---|
| ApplicationOwner | The name of the person who manages this application                   |
| CostCenter       | The cost center of the group that is paying for the Azure consumption |
| BusinessUnit     | <b>ETS</b> (the business unit associated with the subscription)       |

## Core network

The Contoso ETS information security and risk management team reviews Dave's proposed plan to move the application to Azure. They want to ensure that the application isn't exposed to the internet. Dave also has developer apps that in the future will be moved to Azure. These apps require public interfaces. To meet these requirements, he provides both internal and external virtual networks, and a network security group to restrict access.

He creates the following resources:

| RESOURCE TYPE          | NAME          | DESCRIPTION   |
|------------------------|---------------|---|
| Virtual Network        | internal-vnet | Used with the BitBucket application and is connected via ExpressRoute to Contoso's corporate network. A subnet ( <code>bitbucket</code> ) provides the application with a specific IP address space |
| Virtual Network        | external-vnet | Available for future applications that require public-facing endpoints  |
| Network Security Group | bitbucket-nsg | Ensures that the attack surface of this workload is minimized by allowing connections only on port 443 for the subnet where the application lives ( <code>bitbucket</code> )                        |

## Resource locks

Dave recognizes that the connectivity from Contoso's corporate network to the internal virtual network must be protected from any wayward script or accidental deletion.

He creates the following [resource lock](#):

| LOCK TYPE           | RESOURCE      | DESCRIPTION   |
|---------------------|---------------|---|
| <b>CanNotDelete</b> | internal-vnet | Prevents users from deleting the virtual network or subnets, but does not prevent the addition of new subnets |

## Azure Automation

Dave has nothing to automate for this application. Although he created an Azure Automation account, he won't initially use it.

## Azure Security Center

Contoso IT service management needs to quickly identify and handle threats. They also want to understand what problems may exist.

To fulfill these requirements, Dave enables the [Azure Security Center](#), and provides access to the Security Reader role.

## Scenario 2: customer-facing app

The business leadership in the supply chain business unit has identified various opportunities to increase engagement with Contoso's customers by using a loyalty card. Alice's team must create this application and decides that Azure increases their ability to meet the business need. Alice works with Dave from ETS to configure two subscriptions for developing and operating this application.

### Azure subscriptions

Dave logs in to the Azure Enterprise Portal and sees that the supply chain department already exists. However, as this project is the first development project for the supply chain team in Azure, Dave recognizes the need for a new account for Alice's development team. He creates the "R&D" account for her team and assigns access to Alice. Alice logs in via the Azure portal and creates two subscriptions: one to hold the development servers and one to hold the production servers. She follows the previously established naming standards when creating the following subscriptions:

| SUBSCRIPTION USE | NAME  |
|------------------|---|
| Development      | Contoso SupplyChain ResearchDevelopment LoyaltyCard Development |
| Production       | Contoso SupplyChain Operations LoyaltyCard Production           |

### Policies

Dave and Alice discuss the application and identify that this application only serves customers in the North American region. Alice and her team plan to use Azure's Application Service Environment and Azure SQL to create the application. They may need to create virtual machines during development. Alice wants to ensure that her developers have the resources they need to explore and examine problems without pulling in ETS.

For the **development subscription**, they create the following policy:

| FIELD    | EFFECT | DESCRIPTION                                       |
|----------|--------|---|
| location | audit  | Audit the creation of the resources in any region |

They don't limit the type of sku a user can create in development, and they don't require tags for any resource groups or resources.

For the **production subscription**, they create the following policies:

| FIELD    | EFFECT | DESCRIPTION   |
|----------|--------|---|
| location | deny   | Deny the creation of any resources outside of the US data centers |
| tags     | deny   | Require application owner tag                                     |

| FIELD | EFFECT | DESCRIPTION   |
|-------|--------|---|
| tags  | deny   | Require department tag  |
| tags  | append | Append tag to each resource group that indicates production environment |

They don't limit the type of sku a user can create in production.

### Resource tags

Dave understands that he needs to have specific information to identify the correct business groups for billing and ownership. He defines resource tags for resource groups and resources.

| TAG NAME         | TAG VALUE  |
|------------------|--|
| ApplicationOwner | The name of the person who manages this application  |
| Department       | The cost center of the group that is paying for the Azure consumption  |
| EnvironmentType  | <b>Production</b> (Even though the subscription includes <b>Production</b> in the name, including this tag enables easy identification when looking at resources in the portal or on the bill) |

### Core networks

The Contoso ETS information security and risk management team reviews Dave's proposed plan to move the application to Azure. They want to ensure that the Loyalty Card application is properly isolated and protected in a DMZ network. To fulfill this requirement, Dave and Alice create an external virtual network and a network security group to isolate the Loyalty Card application from the Contoso corporate network.

For the **development subscription**, they create:

| RESOURCE TYPE   | NAME          | DESCRIPTION  |
|-----------------|---------------|--|
| Virtual Network | internal-vnet | Serves the Contoso Loyalty Card development environment and is connected via ExpressRoute to Contoso's corporate network |

For the **production subscription**, they create:

| RESOURCE TYPE          | NAME            | DESCRIPTION  |
|------------------------|-----------------|--|
| Virtual Network        | external-vnet   | Hosts the Loyalty Card application and is not connected directly to Contoso's ExpressRoute. Code is pushed via their Source Code system directly to the PaaS services                                      |
| Network Security Group | loyaltycard-nsg | Ensures that the attack surface of this workload is minimized by only allowing in-bound communication on TCP 443. Contoso is also investigating using a Web Application Firewall for additional protection |

## Resource locks

Dave and Alice confer and decide to add resource locks on some of the key resources in the environment to prevent accidental deletion during an errant code push.

They create the following lock:

| LOCK TYPE    | RESOURCE      | DESCRIPTION   |
|--------------|---------------|---|
| CanNotDelete | external-vnet | To prevent people from deleting the virtual network or subnets. The lock does not prevent the addition of new subnets |

## Azure Automation

Alice and her development team have extensive runbooks to manage the environment for this application. The runbooks allow for the addition/deletion of nodes for the application and other DevOps tasks.

To use these runbooks, they enable [Automation](#).

## Azure Security Center

Contoso IT service management needs to quickly identify and handle threats. They also want to understand what problems may exist.

To fulfill these requirements, Dave enables Azure Security Center. He ensures that the Azure Security Center is monitoring the resources, and provides access to the DevOps and security teams.

## Next steps

- To learn about creating Resource Manager templates, see [Best practices for creating Azure Resource Manager templates](#).