

Azure Architecture Center

Guidance for architecting solutions on Azure using established patterns and practices.



ARCHITECTURE
Browse Azure architectures



CONCEPT
Explore cloud best practices



WHAT'S NEW
See what's new



HOW - TO ...
Well
Architected
Framework...



HOW - TO ...
Cloud
Adoption
Framework...



TRAINING
Find partner-led workshops ↗

Architecting applications on Azure

Best practices and patterns for building applications on Microsoft Azure



Design for the cloud

- Principles of a well-designed application
- Best practices in cloud applications
- Responsible engineering
- Application design patterns
- Architect multitenant solutions on Azure
- Build microservices on Azure



Optimize your workload

- Guiding tenets for your architecture
- Examine your workload
- Performance tuning
- Performance antipatterns
- Secure your infrastructure



Choose the right technology



Essential scenarios

[Choose a compute service](#)
[Choose a Kubernetes at the edge option](#)
[Choose a data store](#)
[Choose a load-balancing service](#)
[Choose a messaging service](#)
[Choose an IoT solution](#)

[Architecture for startups](#)
[Azure and Power Platform solutions](#)
[Azure and Microsoft 365 solutions](#)
[AWS services comparison](#)
[Google Cloud services comparison](#)

Technology Areas

Explore architectures and guides for different technologies

Popular Articles

-  [AKS Production Baseline](#)
-  [AWS to Azure services comparison](#)
-  [Google Cloud to Azure services comparison](#)
-  [Cloud Design Patterns](#)
-  [CQRS design pattern](#)
-  [Best practices in cloud applications](#)
-  [Web API design](#)
-  [Performance antipatterns for cloud applications](#)
-  [Choose your Azure compute service](#)
-  [Application architecture fundamentals](#)
-  [Hub-spoke network topology](#)
-  [Architect multitenant solutions on Azure](#)

AI & Machine Learning

-  [Artificial intelligence \(AI\) architecture design](#)
-  [Batch scoring of Python models](#)
-  [Conversational bot](#)
-  [Machine learning options](#)
-  [Natural language processing](#)
-  [Cognitive services options](#)
-  [Team Data Science Process](#)

Analytics

-  [Analytics architecture design](#)
-  [Choose an analytical data store in Azure](#)
-  [Choose a data analytics technology in Azure](#)
-  [Analytics end-to-end with Azure Synapse](#)
-  [Automated enterprise BI with Azure Data Factory](#)
-  [Stream processing with Azure Databricks](#)
-  [Databricks Monitoring](#)
-  [Advanced analytics architecture](#)

Databases

-  [Databases architecture design](#)
-  [Big Data architectures](#)
-  [Build a scalable system for massive data](#)
-  [Choose a data store](#)
-  [Extract, transform, and load \(ETL\)](#)
-  [Online analytical processing \(OLAP\)](#)
-  [Online transaction processing \(OLTP\)](#)
-  [Data lakes](#)

- IoT analytics for construction
- Real-time fraud detection
- Mining equipment monitoring
- Predict the length of stay in hospitals

DevOps

- Checklist
- Advanced Azure Resource Manager Templates
- DevOps with Azure DevOps
- DevOps with containers
- Jenkins server

High performance computing (HPC)

- Computational fluid dynamics (CFD)
- Computer-aided engineering
- HPC video rendering
- Image Modeling
- Linux virtual desktops

[Introduction to HPC on Azure >](#)

Enterprise integration

- Basic enterprise integration
- Enterprise BI with SQL Data Warehouse
- Enterprise integration with queues and events

Identity

- Identity in multitenant applications
- Choose an Active Directory integration architecture
- Integrate on-premises AD with Microsoft Entra ID
- Extend AD DS to Azure
- Create an AD DS forest in Azure
- Extend AD FS to Azure

Internet of Things (IoT)

- Internet of Things (IoT) architecture
- Automotive IoT data

Microservices

- Domain analysis
- Tactical DDD
- Identify microservice boundaries
- Design a microservices architecture
- Monitor microservices in Azure Kubernetes Service (AKS)
- CI/CD for microservices
- CI/CD for microservices on Kubernetes
- Migrate from Cloud Services to Service Fabric
- Azure Kubernetes Service (AKS)
- Azure Service Fabric
- Decomposing a monolithic application

Networking

- [!\[\]\(633dd45d48d71eb51a85c6dd83ee51e9_img.jpg\) Choose a hybrid network architecture](#)
- [!\[\]\(bdddf9191a284aa0945448444083c5b0_img.jpg\) ExpressRoute](#)
- [!\[\]\(944943bcf87a12c5b9337bf7ed1ef546_img.jpg\) ExpressRoute with VPN failover](#)
- [!\[\]\(77e1e368d53d3ed6ec2a15bf2432e026_img.jpg\) Troubleshoot a hybrid VPN connection](#)
- [!\[\]\(beb4ee3dc3a91926258601f02c4f4582_img.jpg\) Hub-spoke topology](#)
- [!\[\]\(dc5b06ae612c8367b0d228fe9920a97f_img.jpg\) DMZ between Azure and on-premises](#)
- [!\[\]\(66ee83dc9a723348caa7f40b8aaad75e_img.jpg\) DMZ between Azure and the Internet](#)
- [!\[\]\(74fe510dcc64d0099dee18e363dbb883_img.jpg\) Highly available network virtual appliances](#)
- [!\[\]\(737f95ac934b65b692ba079db1728de4_img.jpg\) Segmenting Virtual Networks](#)

Serverless applications

- [!\[\]\(4c660a3c4ce1da3313488b7854f55083_img.jpg\) Code walkthrough](#)
- [!\[\]\(f01c435bb39e3068a9b4895c9a993158_img.jpg\) Serverless event processing](#)
- [!\[\]\(c5f009707b314589d498a683120545c5_img.jpg\) Serverless web application](#)

[Introduction to Serverless Applications on Azure >](#)

VM workloads

- [!\[\]\(8d139a66f540002704b5c70b7fe6cc7a_img.jpg\) Linux VM deployment](#)
- [!\[\]\(c209541a4bc5f45e44bd7791f9477320_img.jpg\) Windows VM deployment](#)
- [!\[\]\(8fd54d112e752061b5361c5bdf346185_img.jpg\) N-tier application with Cassandra \(Linux\)](#)
- [!\[\]\(3525fd0bd3680f905a850c70520e38c7_img.jpg\) Multi-region N-tier application](#)
- [!\[\]\(3c3fba180f5a473bd1cb3c114e029235_img.jpg\) Highly scalable WordPress](#)

SAP

- [!\[\]\(9bfa69b6b0f097b09744337d04f22d78_img.jpg\) Overview](#)
- [!\[\]\(7d26c345cabf494d35782f002b741ce9_img.jpg\) SAP HANA on Azure \(Large Instances\)](#)
- [!\[\]\(40fb90293499d45782783c449b0d92d0_img.jpg\) SAP HANA Scale-up on Linux](#)
- [!\[\]\(7da84d8385265e3244ec94f60d0fcdb1_img.jpg\) SAP NetWeaver on Windows on Azure](#)
- [!\[\]\(ee4a2ee0ef75789bb6059be6ccb5c98b_img.jpg\) SAP S/4HANA in Linux on Azure](#)
- [!\[\]\(2c00ae2a46e33230d65febabc5ba4024_img.jpg\) SAP BW/4HANA in Linux on Azure](#)
- [!\[\]\(a107e81a5049260c7632ed0b5b7487c2_img.jpg\) SAP NetWeaver on SQL Server](#)
- [!\[\]\(031070279ccc682ce608f5a03bd958c9_img.jpg\) SAP deployment using an Oracle DB](#)
- [!\[\]\(1b7299a79d758422fba186ce2b0638be_img.jpg\) Dev/test for SAP](#)

Web apps

- [!\[\]\(8ba0a8bc08cfb681721719303df69bb8_img.jpg\) Basic web application](#)
- [!\[\]\(33b903febf51b8cea076831f447c997e_img.jpg\) Baseline zone-redundant web application](#)
- [!\[\]\(630eff6382b86f77a4b5cf981ec06d32_img.jpg\) Multi-region deployment](#)
- [!\[\]\(b3e47a113008163b127f99f2c8ca3a9e_img.jpg\) Web application monitoring](#)
- [!\[\]\(57c2cc3eed2966eaef2ebb697d130483_img.jpg\) E-commerce API management](#)
- [!\[\]\(f0afefe92ccf853de00541fb9556fcf4_img.jpg\) E-commerce front-end](#)
- [!\[\]\(fb6208e6c6a328705923be9cb073eb71_img.jpg\) E-commerce product search](#)
- [!\[\]\(6b5347b70a8ca67076036efc887391d4_img.jpg\) Publishing internal APIs externally](#)
- [!\[\]\(6d99486f9581c8663d5489a4acf3101e_img.jpg\) Securely managed web application](#)
- [!\[\]\(322d9ee2829fb2b5800018987bc7de22_img.jpg\) Highly available web application](#)

Build your skills with Microsoft Learn training

[Build great solutions with the Microsoft Azure Well-Architected Framework](#)

[Introduction to the Well-Architected Framework](#)

[Azure Fundamentals](#)

[Security, responsibility, and trust in Azure](#)

[Architect infrastructure operations in Azure](#)

[Tour the N-tier architecture style](#)

Azure architecture icons

Article • 08/28/2023

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

General guidelines

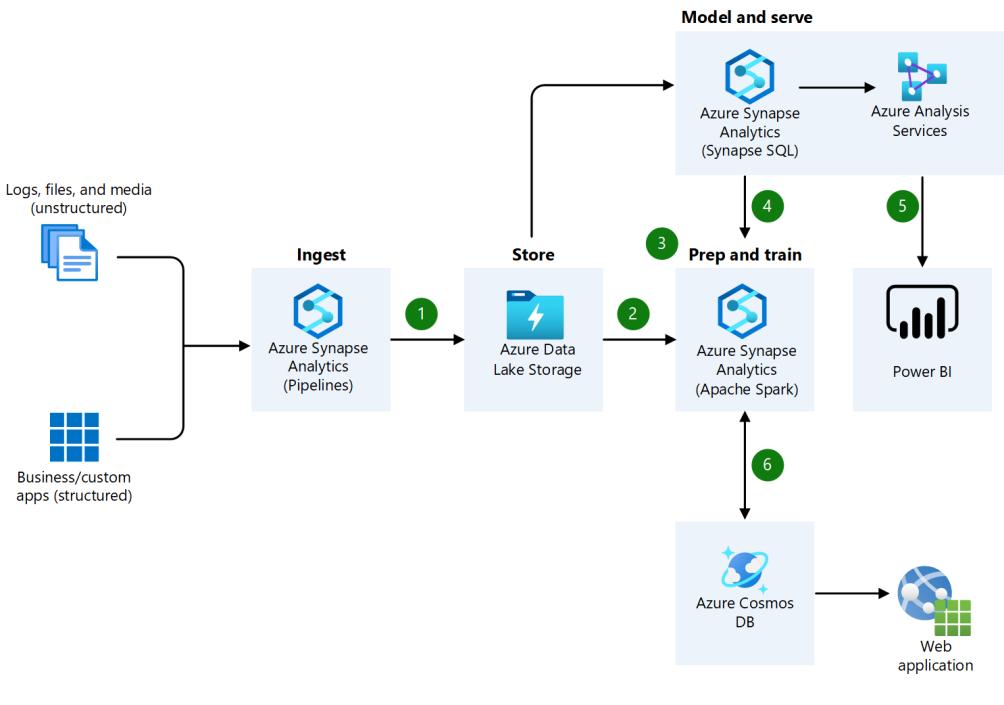
Do's

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

Don'ts

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

Example architecture diagram



[Browse all Azure architectures](#) to view other examples.

Icon updates

Month	Change description
August 2023	Added new Microsoft Entra ID icon. Reorganized folder structure of some icons.
June 2023	General updates.
April 2023	Added 24 Microsoft Defender for IoT icons.
March 2023	Added 13 icons for various services and components.
January 2023	Added ~106 icons for various services and components.
November 2022	Updated icon names from Azure Security Center to Microsoft Defender for Cloud.
January 2021	There are ~26 icons that have been added to the existing set.
November 2020	The folder structure of our collection of Azure architecture icons has changed. The FAQs and Terms of Use PDF files appear in the first level when you download the SVG icons below. The files in the icons folder are the same except there's no longer a CXP folder.

Icon terms

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

I agree to the above terms.

[Download SVG icons](#)

More icon sets from Microsoft

- [Microsoft 365 architecture icons and templates](#)
- [Dynamics 365 icons](#)
- [Microsoft Power Platform icons](#)

What's new in Azure Architecture Center

Download Feed

Article • 01/22/2024

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

<https://www.microsoft.com/en-us/videoplayer/embed/RWxOYN?postJslIMsg=true>

The following new and updated articles have recently been published in the Azure Architecture Center.

January 2024

New articles

- Baseline OpenAI end-to-end chat reference architecture
- Deploy Devito on an Azure virtual machine
- Deploy Ansys HFSS on an Azure virtual machine
- Conceptual planning for IPv6 networking
- Choose an Azure container service
- General considerations for choosing an Azure container service
- Deploy tNavigator on an Azure virtual machine

Updated articles

- AKS day-2 guide: Patch and upgrade guidance (#e6cba7ec73)
- Deploy Altair Radioss on an Azure virtual machine (#274173ea1e)

December 2023

New articles

- AKS Backup and Recovery

Updated articles

- Augment security, observability, and analytics by using Microsoft Sentinel, Azure Monitor, and Azure Data Explorer ([#7f08993a2d ↗](#))
- IBM z/OS online transaction processing on Azure ([#4b3e0077c0 ↗](#))
- Mainframe and midrange data replication to Azure using Qlik ([#4b3e0077c0 ↗](#))
- DR for Azure Data Platform - Deploy this scenario ([#fb09637e3d ↗](#))

November 2023

New articles

- Use Azure Batch to run Financial Service Industry (FSI) workloads
- Deploy Quantum ESPRESSO on an Azure virtual machine
- Mainframe workload migration proof of concept
- Enable real-time sync of MongoDB Atlas data changes to Azure Synapse Analytics
- Deploy M-Star on a virtual machine
- Software-defined vehicle DevOps toolchain
- Architectural approaches for control planes in multitenant solutions
- Deploy CP2K on an Azure virtual machine
- Deploy NAMD on a virtual machine

Updated articles

- Deploy WRF on an Azure virtual machine ([#0280138a15 ↗](#))
- Continuous validation with Azure Load Testing and Azure Chaos Studio ([#4522d8522f ↗](#))
- Deploy GROMACS on an Azure virtual machine ([#7cf6296c95 ↗](#))
- AKS triage—Cluster health ([#b0e359ee7e ↗](#))
- AKS triage—Container registry connectivity ([#b0e359ee7e ↗](#))
- AKS triage—Admission controllers ([#b0e359ee7e ↗](#))
- AKS triage—Workload deployments ([#b0e359ee7e ↗](#))
- AKS triage—Node health ([#b0e359ee7e ↗](#))
- Azure Kubernetes Service (AKS) operations triage ([#b0e359ee7e ↗](#))
- Deploy Autodesk Maya on a virtual machine ([#df60074cf9 ↗](#))
- The journey to SaaS: Dynamics 365 ([#56ce7c6db6 ↗](#))
- Azure landing zones - Bicep modules design considerations ([#399bd77fc2 ↗](#))
- Azure landing zones - Terraform module design considerations ([#399bd77fc2 ↗](#))

October 2023

New articles

- [The journey to SaaS: Dynamics 365](#)
- [Real-time analytics on data with Azure Service Bus and Azure Data Explorer](#)
- [Unisys ClearPath Forward OS 2200 enterprise server virtualization on Azure](#)
- [Deploy OpenRadioss on an Azure virtual machine](#)
- [Windows 365 Azure network connection](#)
- [Automotive connected fleets](#)

Updated articles

- [Microsoft SaaS stories \(#831aeb8915↗\)](#)
- [SWIFT Alliance Connect Virtual on Azure \(#cd5aca549a↗\)](#)
- [Ingest FAA SWIM content to analyze flight data \(#6fa8f0f9cc↗\)](#)
- [Microsoft Entra security for AWS \(#c00d96f670↗\)](#)
- [Baseline architecture for an AKS cluster \(#c00d96f670↗\)](#)
- [Azure VMware Solution networking \(#bd323181bd↗\)](#)

Deploy Azure landing zones

Article • 08/23/2023

This article discusses the options available to you to deploy platform and application landing zones. Platform landing zones provide centralized services used by workloads. Application landing zones are environments deployed for the workloads themselves.

ⓘ Important

For more information about platform versus application landing zones definitions, see [What is an Azure landing zone?](#) in the Cloud Adoption Framework for Azure documentation.

This article covers common roles and responsibilities for differing cloud operating models. It also lists deployment options for platform and application landing zones.

Cloud operating model roles and responsibilities

The Cloud Adoption Framework describes four [common cloud operating models](#). Azure [identity and access for landing zones](#) recommends five role definitions (Roles) to consider if your organization's cloud operating model requires customized role-based access control. If your organization has more decentralized operations, the Azure [built-in roles](#) might be sufficient.

The following table outlines the key roles for each of the cloud operating models.

Role	Decentralized operations	Centralized operations	Enterprise operations	Distributed operations
Azure platform owner (such as the built-in Owner role)	Workload team	Central cloud strategy	Enterprise architect in Cloud Center of Excellence (CCoE)	Based on portfolio analysis. See Business alignment and Business commitments .
Network management (NetOps)	Workload team	Central IT	Central Networking in CCoE	Central Networking for each distributed team + CCoE.
Security operations	Workload team	Security operations	CCoE + SOC	Mixed. See Define a security strategy .

Role	Decentralized operations	Centralized operations	Enterprise operations	Distributed operations
(SecOps)		center (SOC)		
Subscription owner	Workload team	Central IT	Central IT + Application Owners	CCoE + Application Owners.
Application owners (DevOps, AppOps)	Workload team	Workload team	Central IT + Application Owners	CCoE + Application Owners.

Platform

The following options provide an opinionated approach to deploy and operate the [Azure landing zone conceptual architecture](#) as detailed in the Cloud Adoption Framework. Depending upon customizations, the resulting architecture might not be the same for all the options listed here. The differences between the options are how you deploy the architecture. They use differing technologies, take different approaches, and are customized differently.

Deployment option	Description
Azure landing zone Portal accelerator	An Azure portal-based deployment provides a full implementation of the conceptual architecture, along with opinionated configurations for key components, such as management groups and policies.
Azure landing zone Terraform accelerator	This accelerator provides an orchestrator module and also allows you to deploy each capability individually or in part.
Azure landing zone Bicep accelerator	A modular accelerator where each module encapsulates a core capability of the Azure landing zone conceptual architecture . While the modules can be deployed individually, the design proposes the use of orchestrator modules to encapsulate the complexity of deploying different topologies with the modules.

Operate Azure landing zones

After you deploy the landing zone, you need to operate and maintain it. For more information, see the guidance on how to [Keep your Azure landing zone up to date](#).

[Azure Governance Visualizer](#) is intended to help you get a holistic overview on your technical Azure governance implementation by connecting the dots and providing

sophisticated reports.

Alternative platform deployment for policies with Enterprise Policy as Code (EPAC)

[Enterprise Policy as Code \(EPAC\)](#) is an alternative method to deploy, manage, and operate Azure Policy in your environment. You can use EPAC instead of the preceding [platform options](#) to manage the policies in an Azure landing zones environment. For more information on the integration approach, see [Integrate EPAC with Azure landing zones](#).

EPAC is best suited for more advanced and mature DevOps and infrastructure-as-code customers. However, customers of any size can use EPAC if they want to after they assess it. To ensure that you're aligned, see [Who should use EPAC?](#) first.

Note

Evaluate and consider both options carefully. Potentially run through an MVP or proof of concept before you decide on what to use in the long term.

Subscription vending

After the platform landing zone is in place, the next step is to create and operationalize application landing zones for workload owners. Subscription democratization is a [design principle](#) of Azure landing zones that uses subscriptions as units of management and scale. This approach accelerates application migrations and new application development.

[Subscription vending](#) standardizes the process you use to request, deploy, and govern subscriptions. It enables application teams to deploy their workloads faster. To get started, see [Subscription vending implementation guidance](#). Then review the following infrastructure-as-code modules. They provide flexibility to fit your implementation needs.

Deployment option	Description
Bicep subscription vending	The subscription vending Bicep module is designed to accelerate deployment of the individual landing zones (also known as subscriptions) within an Azure Active Directory tenant on Enterprise Agreement (EA), Microsoft Customer Agreement (MCA), and Microsoft Partner Agreement (MPA) billing accounts.

Deployment option	Description
Terraform subscription vending	The subscription vending Terraform module is designed to accelerate deployment of the individual landing zones (also known as subscriptions) within an Azure Active Directory tenant on EA, MCA, and MPA billing accounts

Application

Application landing zones are one or more subscriptions that are deployed as environments for workloads or applications. These workloads can take advantage of services deployed in platform landing zones. The application landing zones can be centrally managed applications, decentralized workloads, or technology platforms such as Azure Kubernetes Service (AKS) that host applications.

You can use the following options to deploy and manage applications or workloads in an application landing zone.

Application	Description
AKS landing zone accelerator	An open-source collection of Azure Resource Manager (ARM), Bicep, and Terraform templates that represent the strategic design path and target technical state for an AKS deployment.
Azure App Service landing zone accelerator	Proven recommendations and considerations across both multitenant and App Service environment use cases with a reference implementation for ASEv3-based deployment.
Azure API Management landing zone accelerator	Proven recommendations and considerations for deploying APIM management with a reference implementation showcasing Azure Application Gateway with an internal APIM instance-backed Azure Functions as back end.
SAP on Azure landing zone accelerator	Terraform and Ansible templates that accelerate SAP workload deployments by using Azure landing zone best practices, including the creation of infrastructure components like compute, networking, storage, monitoring, and build of SAP systems.
HPC landing zone accelerator	An end-to-end HPC cluster solution in Azure that uses tools like Terraform, Ansible, and Packer. It addresses Azure landing zone best practices, including implementing identity, jumpbox access, and autoscale.
Azure VMware Solution landing zone accelerator	ARM, Bicep, and Terraform templates that accelerate VMware deployments, including Azure VMware Solution private cloud, jumpbox, networking, monitoring, and add-ons.

Application	Description
Azure Virtual Desktop landing zone accelerator	ARM, Bicep, and Terraform templates that accelerate Azure Virtual Desktop deployments, including creation of host pools, networking, storage, monitoring, and add-ons.
Azure Red Hat OpenShift landing zone accelerator	An open-source collection of Terraform templates that represent an optimal Azure Red Hat OpenShift deployment that includes Azure and Red Hat resources.
Azure Arc landing zone accelerator for hybrid and multicloud	Azure Arc-enabled servers, Kubernetes, and Azure Arc-enabled SQL Managed Instance. See the Jumpstart ArcBox overview .
Azure Spring Apps landing zone accelerator	Azure Spring Apps landing zone accelerator is intended for an application team that builds and deploys Spring Boot applications in a typical landing enterprise zone design. As the workload owner, use architectural guidance provided in this accelerator to achieve your target technical state with confidence.
Enterprise-scale landing zone for Citrix on Azure	Design guidelines for the Cloud Adoption Framework for Citrix Cloud in an Azure enterprise-scale landing zone cover for many design areas.

Azure landing zones - Bicep modules design considerations

Article • 11/07/2023

This article discusses the design considerations of the modularized [Azure Landing Zones \(ALZ\) - Bicep](#) solution you can use to deploy and manage the core platform capabilities of the [Azure landing zone conceptual architecture](#) as detailed in the Cloud Adoption Framework (CAF).

[Bicep](#) is a domain-specific language (DSL) that uses declarative syntax to deploy Azure resources. It has concise syntax, reliable type safety, and support for code reuse.



An implementation of this architecture is available on [GitHub: Azure Landing Zones \(ALZ\) - Bicep Implementation](#). You can use it as a starting point and configure it as per your needs.

<https://www.youtube-nocookie.com/embed/-pZNrH1GOxs>

ⓘ Note

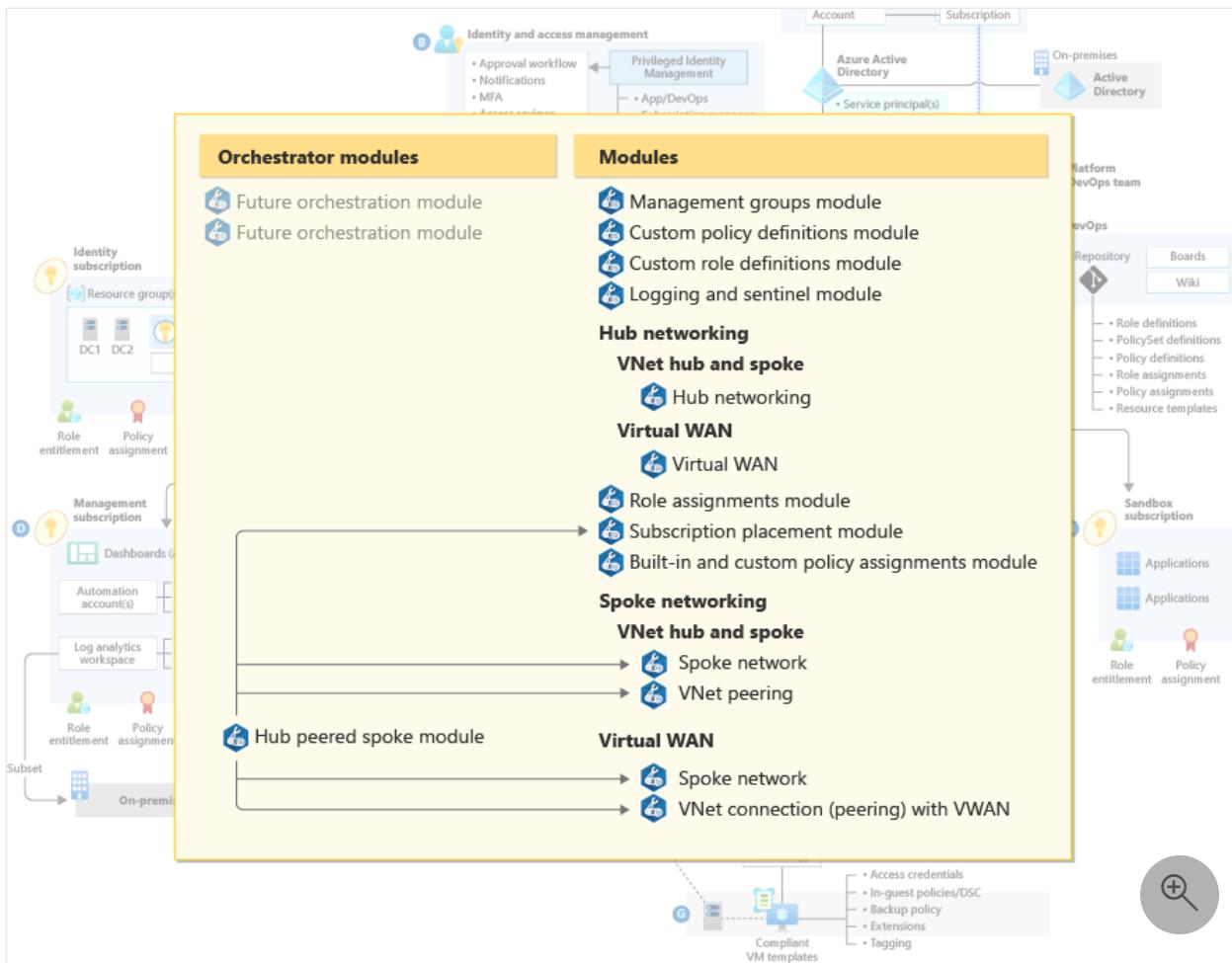
There are **implementations** for several deployment technologies, including portal-based, ARM templates and Terraform modules. The choice of deployment technology should not influence the resulting Azure landing zones deployment.

ALZ Bicep Accelerator

You can find step by step guidance around implementing, automating, and maintaining your ALZ Bicep module with the [ALZ Bicep Accelerator](#).

The ALZ Bicep Accelerator framework was developed to provide end-users support to onboarding and deployment of ALZ Bicep using full-fledged CI/CD pipelines, support for GitHub Actions and Azure DevOps Pipelines, dedicated Framework to stay in-sync with new ALZ Bicep releases and modify or add custom modules, and provides branching strategy guidance and pull request pipelines for linting and validating Bicep modules.

Design



The architecture takes advantage of the modular nature of Azure Bicep and is composed of number of modules. Each module encapsulates a core capability of the Azure Landing Zones conceptual architecture. The modules can be deployed individually, but there are dependencies to be aware of.

The architecture proposes the inclusion of orchestrator modules to simplify the deployment experience. The orchestrator modules could be used to automate the deployment of the modules and to encapsulate differing deployment topologies.

Modules

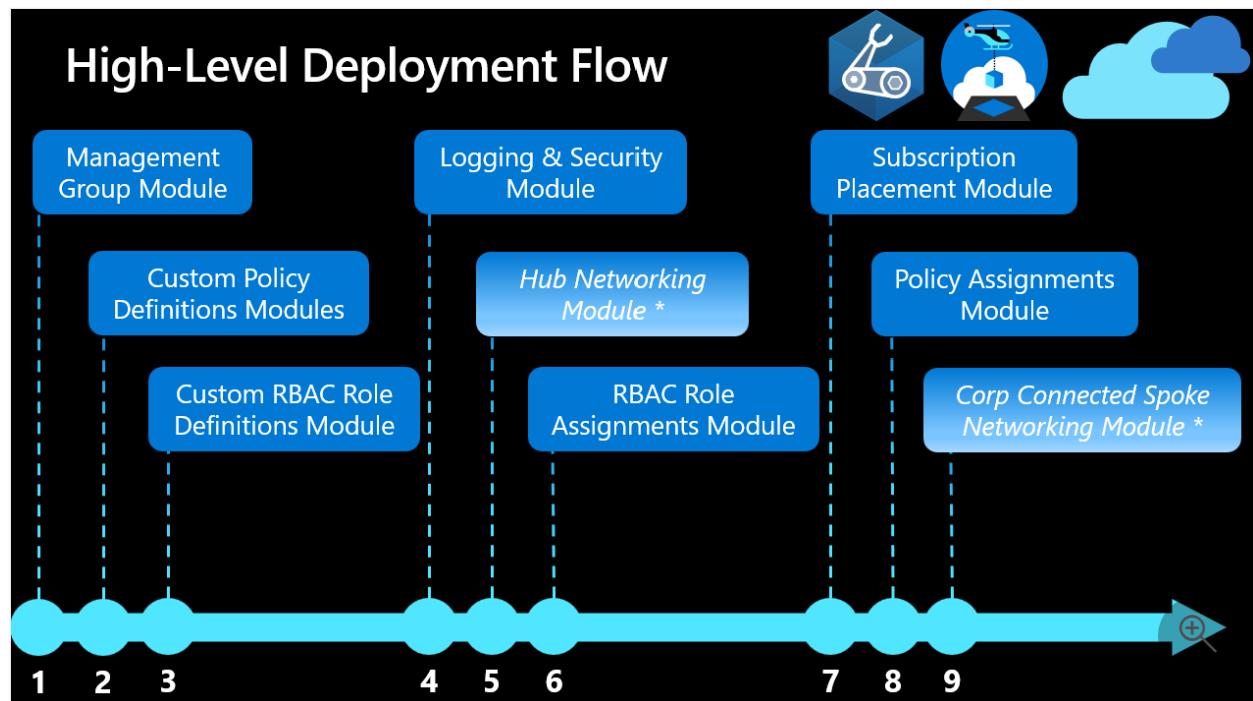
A core concept in Bicep is the use of [modules](#). Modules enable you to organize deployments into logical groupings. With modules, you improve the readability of your Bicep files by encapsulating complex details of your deployment. You can also easily reuse modules for different deployments.

The ability to re-use modules offers a real benefit when defining and deploying landing zones. It enables repeatable, consistent environments in code while reducing the effort required to deploy at scale.

Layers and staging

<https://www.youtube-nocookie.com/embed/FNT0ZtUxYKQ>

In addition to modules, the Bicep landing zone architecture is structured using a concept of layers. Layers are groups of Bicep modules that are intended to be deployed together. Those groups form logical stages of the implementation.



A benefit of this layered approach is the ability to add to your environment incrementally over time. For example, you can start with a small number of the layers. You can add the remaining layers at a subsequent stage when you're ready.

Module descriptions

This section provides a high-level overview of the core modules in this architecture.

Layer	Module	Description	Useful Links
Core	Management Groups	Management groups are the highest level resources in an Azure tenant. Management groups allow you to more easily manage your resources. You can apply policy at the management group level and lower level resources will inherit that policy. Specifically, you can apply the following items at the management group level that will	<ul style="list-style-type: none">• Management groups - Cloud Adoption Framework (CAF) documentation• Module: Management Groups - Reference Implementation

Layer	Module	Description	Useful Links
		<p>be inherited by subscriptions under the management group:</p> <ul style="list-style-type: none"> • Azure Policies • Azure Role Based Access Controls (RBAC) role assignments • Cost controls <p>This module deploys the management group hierarchy as defined in the Azure landing zone conceptual architecture.</p>	
Core	Custom Policy Definitions	<p>DeployIfNotExists (DINE) or Modify policies help ensure the subscriptions and resources that make up landing zones are compliant. The policies also ease the burden of management of landing zones.</p> <p>This module deploys custom policy definitions to management groups. Not all customers are able to use DINE or Modify policies. If that is the case for you, CAF guidance on custom policies provides guidance.</p>	<ul style="list-style-type: none"> • Adopt policy-driven guardrails - CAF documentation • Module: Custom policy definitions - Reference Implementation • Custom policy definitions deployed in reference implementations
Core	Custom Role Definitions	<p>Role-based access control (RBAC) simplifies the management of user rights within a system. Instead of managing the rights of individuals, you determine the rights required for different roles in your system. Azure RBAC has several built-in roles. Custom role definitions allow you to create custom roles for your environment.</p> <p>This module deploys custom role definitions. The module should follow CAF guidance on Azure role-based access control.</p>	<ul style="list-style-type: none"> • Azure role-based access control - CAF documentation • Custom role definitions deployed in reference implementation

Layer	Module	Description	Useful Links
Management	Logging, Automation & Sentinel	<p>Azure Monitor, Azure Automation and Microsoft Sentinel allow you monitor and manage your infrastructure and workloads. Azure Monitor is a solution that allows you to collect, analyze and act on telemetry from your environment.</p> <p>Microsoft Sentinel is a cloud-native security information and event management (SIEM). It allows you to:</p> <ul style="list-style-type: none"> • Collect - Collect data across your entire infrastructure • Detect - Detect threats that were previously undetected • Respond - Respond to legitimate threats with built-in orchestration • Investigate - Investigate threats with artificial intelligence <p>Azure Automation is a cloud-based automation system. It includes:</p> <ul style="list-style-type: none"> • Configuration management - Inventory and track changes for Linux and Windows virtual machines and manage desired state configuration • Update management - Assess Windows and Linux system compliance and create scheduled deployments to meet compliance • Process automation - Automate management tasks 	<ul style="list-style-type: none"> • Workload management and monitoring - CAF documentation • Module: Logging, Automation & Sentinel - Reference Implementation ↗

Layer	Module	Description	Useful Links
		This module deploys the tools necessary to monitor, manage and access threats to your environment. These tools should include Azure Monitor, Azure Automation and Microsoft Sentinel.	
Connectivity	Networking	<p>Network topology is a key consideration in Azure landing zone deployments. CAF focuses on 2 core networking approaches:</p> <ul style="list-style-type: none"> • Topologies based on Azure Virtual WAN • Traditional topologies <p>These modules deploy the network topology that you choose.</p>	<ul style="list-style-type: none"> • Define an Azure network topology - CAF Documentation • Modules: Network Topology Deployment - Reference Implementation ↗
Identity	Role Assignments	<p>Identity and access management (IAM) is the key security boundary in cloud computing. Azure RBAC allows you to perform role assignments of built-in roles or custom role definitions to security principals.</p> <p>This module deploys role assignments to Service Principals, Managed Identities or security groups across management groups and subscriptions. The module should follow CAF guidance on Azure identity and access management.</p>	<ul style="list-style-type: none"> • Azure identity and access management design area - CAF documentation • Module: Role Assignments for Management Groups & Subscriptions - Reference Implementation ↗
Core	Subscription Placement	<p>Subscriptions that are assigned to a management group inherit:</p> <ul style="list-style-type: none"> • Azure Policies • Azure Role Based Access Controls (RBAC) role assignments • Cost controls 	<ul style="list-style-type: none"> • Management groups - Cloud Adoption Framework (CAF) documentation • Module: Subscription Placement ↗

Layer	Module	Description	Useful Links
		This module moves subscriptions under the appropriate management group.	
Core	Built-In and Custom Policy Assignments	This module deploys the default Azure landing zone Azure Policy assignments to management groups. It also creates role assignments for system-assigned Managed Identities created by policies.	<ul style="list-style-type: none"> • Adopt policy-driven guardrails - CAF documentation • Module: ALZ Default Policy Assignments ↗
Management	Orchestrator Modules	Orchestrator modules can greatly improve the deployment experience. These modules encapsulate the deployment of multiple modules in a single module. This hides the complexity from the end user.	<ul style="list-style-type: none"> • Module: Orchestration - hubPeeredSpoke - Spoke network, including peering to Hub (Hub & Spoke or Virtual WAN) ↗

Customizing the Bicep implementation

The [Azure landing zone implementations](#) provided as part of the Cloud Adoption Framework suit a wide variety of requirements and use cases. However, there are often scenarios where customization is required to meet specific business needs.

💡 Tip

See [Tailor the Azure landing zone architecture to meet requirements](#) for further information.

Once the platform landing zone is implemented the next step is to deploy [Application landing zones](#) which enable application teams under the `landing_zones` management group with the guardrails that Central IT or PlatformOps administrators require. The `corp` management group is for corporate connected applications, while the `online` management group is for applications that are primarily publicly facing, but may still connect to corporate applications via hub networks in some scenarios.

[https://www.youtube-nocookie.com/embed/cZ7IN3zGbyM ↗](https://www.youtube-nocookie.com/embed/cZ7IN3zGbyM)

The [Bicep Azure landing zone implementation](#) ↗ can be used as the basis of your customized deployment. It provides you a way to accelerate your implementation by

removing the need to start from scratch because of a specific required change that rules a ready-made option out.



Information on customizing the modules is available in the GitHub repo wiki [GitHub: Azure Landing Zones \(ALZ\) Bicep - Wiki- Consumer Guide](#). You can use it as a starting point and configure it as per your needs.

Azure landing zones - Terraform module design considerations

Article • 03/30/2023

This article discusses important areas to consider when using the [Azure landing zones Terraform module](#). The module provides an opinionated approach to deploy and operate an Azure platform based on the [Azure landing zone conceptual architecture](#) as detailed in the Cloud Adoption Framework (CAF).

Terraform is an open-source Infrastructure as Code (IaC) tool, created by HashiCorp, that uses declarative syntax to deploy infrastructure resources. It is extensible, has cross-platform support and enables immutable infrastructure through state tracking.

<https://www.youtube-nocookie.com/embed/Pqfleth62Yg>

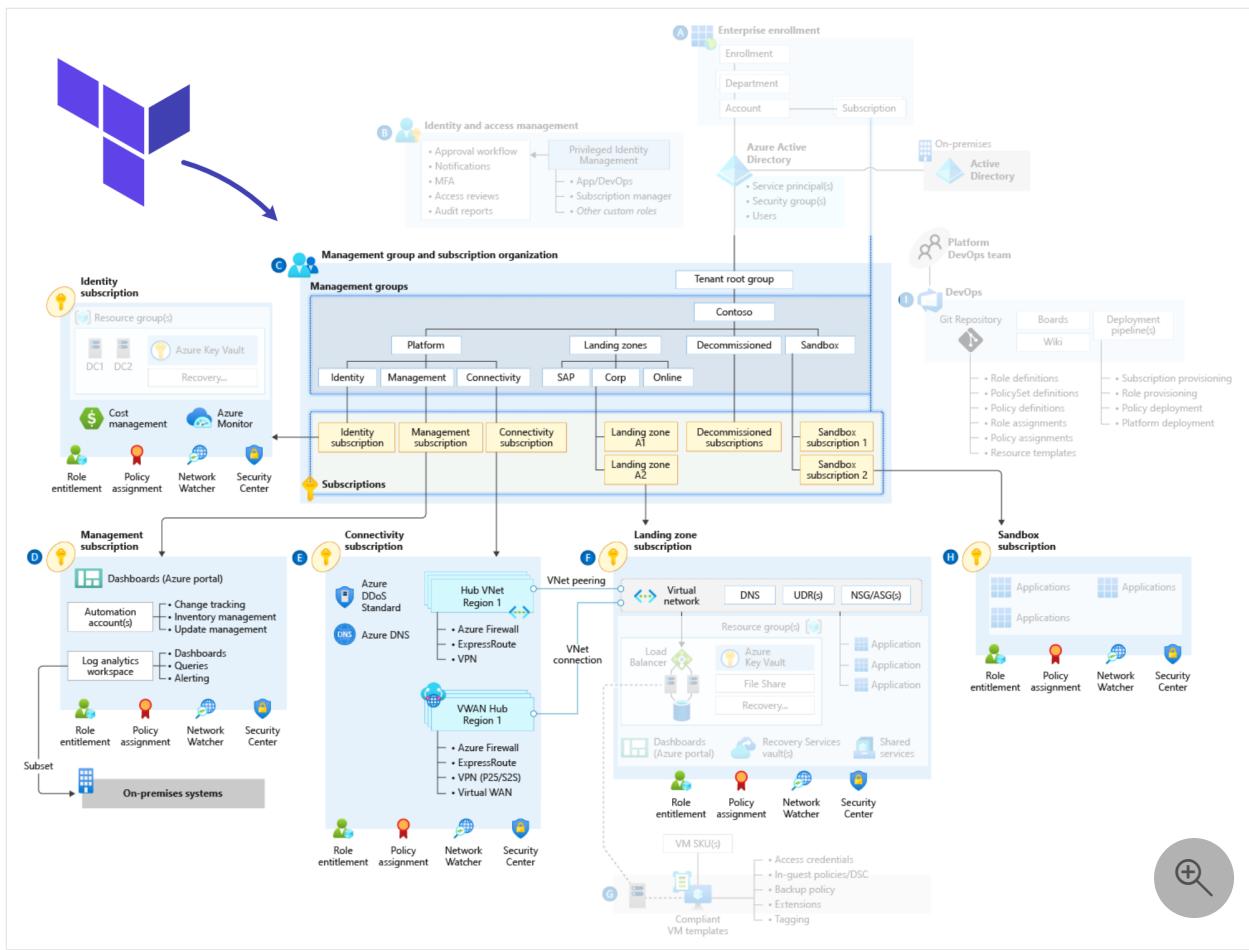
ⓘ Important

The module is available on the [Terraform Registry: Azure landing zones Terraform module](#). You can use it as a starting point and configure it as per your needs.

ⓘ Note

There are [implementations](#) for several deployment technologies, including portal-based, ARM templates and Terraform modules. The choice of deployment technology should not influence the resulting Azure landing zones deployment.

Design



The architecture takes advantage of the configurable nature of Terraform and is composed of a primary orchestration module. This module encapsulates multiple capabilities of the Azure landing zones conceptual architecture. You can deploy each capability individually or in part. For example, you can deploy just a hub network, or just the Azure DDoS Protection, or just the DNS resources. When doing so, you need to take into account that the capabilities have dependencies.

The architecture utilizes an orchestrator approach to simplify the deployment experience. You might prefer to implement each capability using one or more dedicated module instances where each is dedicated to a specific part of the architecture. This is all possible with the correct configuration.

Modules

A core concept in Terraform is the use of modules. Modules enable you to organize deployments into logical groupings. With modules, you improve the readability of your Terraform files by encapsulating complex details of your deployment. You can also easily reuse modules for different deployments.

The ability to re-use modules offers a real benefit when defining and deploying landing zones. It enables repeatable, consistent environments in code while reducing the effort required to deploy at scale.

The Terraform implementation of Azure landing zones is delivered using a single module that acts as an orchestration layer. The orchestration layer allows you to select which resources are deployed and managed using the module. The module can be used multiple times in the same environment to deploy resources independently from each other. This can be useful in organizations where different teams are responsible for the different capabilities, or collections of sub-resources.

Layers and staging

The implementation focuses on the central resource hierarchy of the [Azure landing zone conceptual architecture](#). The design is centered around the following capabilities:

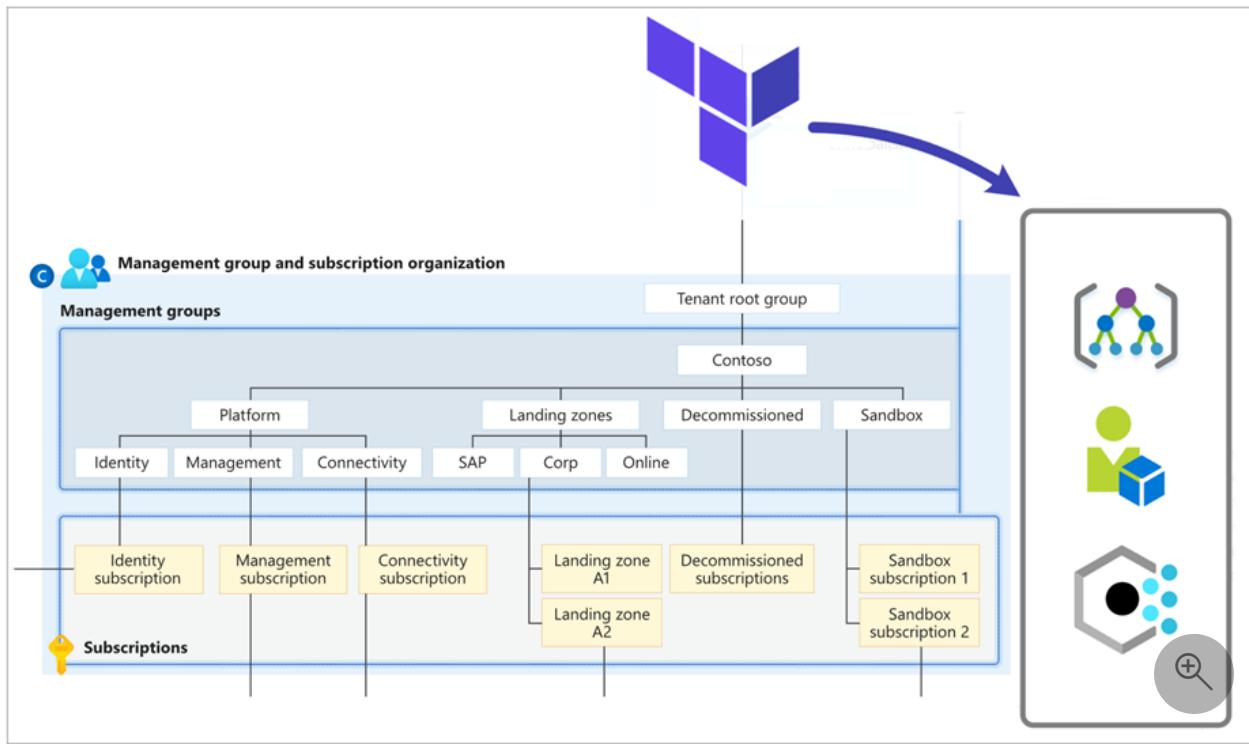
- Core resources
- Management resources
- Connectivity resources
- Identity resources

The module groups resources into these capabilities as they are intended to be deployed together. These groups form logical stages of the implementation.

You control the deployment of each of these capabilities by using feature flags. A benefit of this approach is the ability to add to your environment incrementally over time. For example, you can start with a small number of capabilities. You can add the remaining capabilities at a later stage when you're ready.

Core resources

The [core resources ↗](#) capability of the module aligns to the [resource organization](#) design area of the Cloud Adoption Framework. It deploys the foundational resources of the [conceptual architecture for Azure landing zones](#).



Archetypes

An important concept within the core resources capability is the inclusion of archetypes.

Archetypes provide a reusable, code-based approach to defining which policy definitions, policy set definitions, policy assignments, role definitions and role assignments must be applied at a given scope. In the Terraform implementation, these decisions are encapsulated as [Archetype Definitions](#).

To create a landing zone, management groups are associated with an archetype definition. In the below example for a corp landing zone, the archetype_config has a pointer to the "es_corp" archetype definition. That definition contains all the policy and role configurations which will be added to this management group.

```
Terraform

es_corp_landing_zones = {
  "contoso-corp" = {
    display_name          = "Corp"
    parent_management_group_id = "contoso-landing-zones"
    subscription_ids      = []
    archetype_config      = {
      archetype_id = "es_corp"
      parameters   = {}
      access_control = {}
    }
  }
}
```

When the built-in archetypes don't align to your requirements, the module provides options to either [create new archetypes](#) or [make changes to existing](#).

Management resources

The [management resources](#) capability of the module aligns to the [management](#) design area of the Cloud Adoption Framework. This capability provides the option to deploy management and monitoring resources to the management platform landing zone.

Connectivity resources

The [connectivity resources](#) capability of the module provides the option to deploy the [network topology and connectivity](#) of the [conceptual architecture for Azure landing zones](#).

Identity resources

The [identity resources](#) capability of the module aligns to the [Azure identity and access management design area](#) of the Cloud Adoption Framework. This capability provides the option to configure policies on the Identity platform landing zone.

ⓘ Note

No resources are deployed with this capability. When the `deploy_identity_resources` variable is set to true, Azure Policy assignments are configured that protect resources in the identity platform landing zone subscription.

Module descriptions

This section provides a high-level overview of the resources deployed by this module.

Layer	Resource Type(s)	Description	Useful Links

Layer	Resource Type(s)	Description	Useful Links
Core	Management Groups	<p>Management groups are the highest level resources in an Azure tenant. Management groups allow you to more easily manage your resources. You can apply policy at the management group level and lower level resources will inherit that policy. Specifically, you can apply the following items at the management group level that will be inherited by subscriptions under the management group:</p> <ul style="list-style-type: none"> • Azure Policies • Azure Role Based Access Controls (RBAC) role assignments • Cost controls 	<ul style="list-style-type: none"> • Management groups - Cloud Adoption Framework (CAF) documentation
Core	Policy definitions, policy assignments, and policy set definitions	<p>DeployIfNotExists (DINE) or Modify policies help ensure the subscriptions and resources that make up landing zones are compliant. Policies are assigned to management groups through policy assignments. The policies ease the burden of management of landing zones. Policy set definitions group sets of policies together.</p> <p>Not all customers are able to use DINE or Modify policies. If that is the case for you, CAF guidance on custom policies provides guidance.</p>	<ul style="list-style-type: none"> • Adopt policy-driven guardrails - CAF documentation • Custom policy definitions deployed in reference implementations ↗

Layer	Resource Type(s)	Description	Useful Links
Core	Role definitions and role assignments	<p>Role-based access control (RBAC) simplifies the management of user rights within a system. Instead of managing the rights of individuals, you determine the rights required for different roles in your system. Azure RBAC has several built-in roles. Custom role definitions allow you to create custom roles for your environment.</p> <p>Identity and access management (IAM) is the key security boundary in cloud computing. Azure RBAC allows you to perform role assignments of built-in roles or custom role definitions to Service Principals, Managed Identities or security groups across management groups and subscriptions.</p>	<ul style="list-style-type: none"> • Azure role-based access control - CAF documentation • Azure identity and access management design area - CAF documentation • Custom policy definitions deployed in reference implementations ↗

Layer	Resource Type(s)	Description	Useful Links
Management	Azure Monitor, Azure Automation, and Microsoft Sentinel	<p>Azure Monitor, Azure Automation and Microsoft Sentinel allow you to monitor and manage your infrastructure and workloads. Azure Monitor is a solution that allows you to collect, analyze and act on telemetry from your environment.</p> <p>Microsoft Sentinel is a cloud-native security information and event management (SIEM). It allows you to:</p> <ul style="list-style-type: none"> • Collect - Collect data across your entire infrastructure • Detect - Detect threats that were previously undetected • Respond - Respond to legitimate threats with built-in orchestration • Investigate - Investigate threats with artificial intelligence <p>Azure Automation is a cloud-based automation system. It includes:</p> <ul style="list-style-type: none"> • Configuration management - Inventory and track changes for Linux and Windows virtual machines and manage desired state configuration • Update management - Assess Windows and Linux system compliance and create scheduled deployments to meet compliance • Process automation - Automate management tasks 	<ul style="list-style-type: none"> • Workload management and monitoring - CAF documentation

Layer	Resource Type(s)	Description	Useful Links
Connectivity	Core networking resource types listed here ↗	<p>Network topology is a key consideration in Azure landing zone deployments. CAF focuses on two core networking approaches:</p> <ul style="list-style-type: none"> • Topologies based on Azure Virtual WAN • Traditional topologies 	<ul style="list-style-type: none"> • Define an Azure network topology - CAF Documentation
Connectivity	Azure DDoS Protection	Azure landing zone guidance recommends enabling Azure DDoS Network Protection. This service offers turnkey protection against DDoS attacks.	<ul style="list-style-type: none"> • Azure DDoS Network Protection
Connectivity	DNS Zones, Private DNS Zones, and Private DNS Zone Virtual Network Link	Private DNS zones can be deployed to support the use of private endpoints. A private endpoint is a NIC that is assigned a private IP address from your virtual network. You can use the private IP address to securely communicate to services that supports Azure Private Link. Private DNS zones can be configured to resolve the fully qualified domain name (FQDN) of the service to the private endpoint private IP address.	<ul style="list-style-type: none"> • Azure Private Endpoint DNS configuration

Using the Terraform module

https://www.youtube-nocookie.com/embed/vFO_cyolUW0 ↗

Deploying core resources

By default, the module will deploy the following hierarchy, which is the core set of landing zone management groups:

- Root
 - Platform
 - Identity
 - Management
 - Connectivity

- Landing zones
- Decommissioned
- Sandbox

The SAP, Corp and Online landing zone management groups don't apply to everyone so they aren't deployed by default. The following are ways to deploy these:

1. For demo purposes, you can set the `deploy_demo_landing_zones` variable to true that will deploy SAP, Corp and Online landing zones
2. For production purposes, you can turn on the management groups you want by setting the following variables to true:
 - `deploy_corp_landing_zones`
 - `deploy_online_landing_zones`
 - `deploy_sap_landing_zones`
3. You can deploy your own custom landing zone management groups by creating a [custom landing zone](#) definition

Deploying management resources

To deploy the management resources, the `deploy_management_resources` variable must be set to true and the `subscription_id_management` variable must be set to the ID of the management subscription where the resources are to be deployed.

```
Bash
```

```
deploy_management_resources = true
subscription_id_management = <management subscription id>
```

Deploying connectivity resources

[Deploy Connectivity Resources](#) provides guidance on how to deploy these topologies.

Deploying identity resources

To deploy the identity capability, the `deploy_identity_resources` variable must be set to true and the `subscription_id_identity` variable must be set to the ID of the identity subscription where the policies are to be configured.

```
Bash
```

```
deploy_identity_resources = true  
subscription_id_identity = <identity subscription id>
```

Customizing the Terraform implementation

<https://www.youtube-nocookie.com/embed/ct2KHaA7ekI> ↗

The [Azure landing zone implementations](#) provided as part of the Cloud Adoption Framework suit a wide variety of requirements and use cases. However, there are often scenarios where customization is required to meet specific business needs.

💡 Tip

See [Tailor the Azure landing zone architecture to meet requirements](#) for further information.

The [Azure landing zones Terraform module](#) ↗ can be used as the basis of your customized deployment. It provides you a way to accelerate your implementation by removing the need to start from scratch because of a specific required change that rules a ready-made option out.



Information on customizing the modules is available in the GitHub repo wiki [GitHub: Azure landing zones Terraform module - Wiki](#) ↗. You can use it as a starting point and configure it as per your needs.

Azure Virtual Desktop landing zone design guide

Article • 04/20/2023

This article provides a design-oriented overview of the [enterprise-scale landing zone for Azure Virtual Desktop](#), for architects and technical decision makers. The goal is to help you quickly gain an understanding of the accelerator and how it's designed, allowing you to shorten the time required to complete a successful deployment.

Landing zone concepts

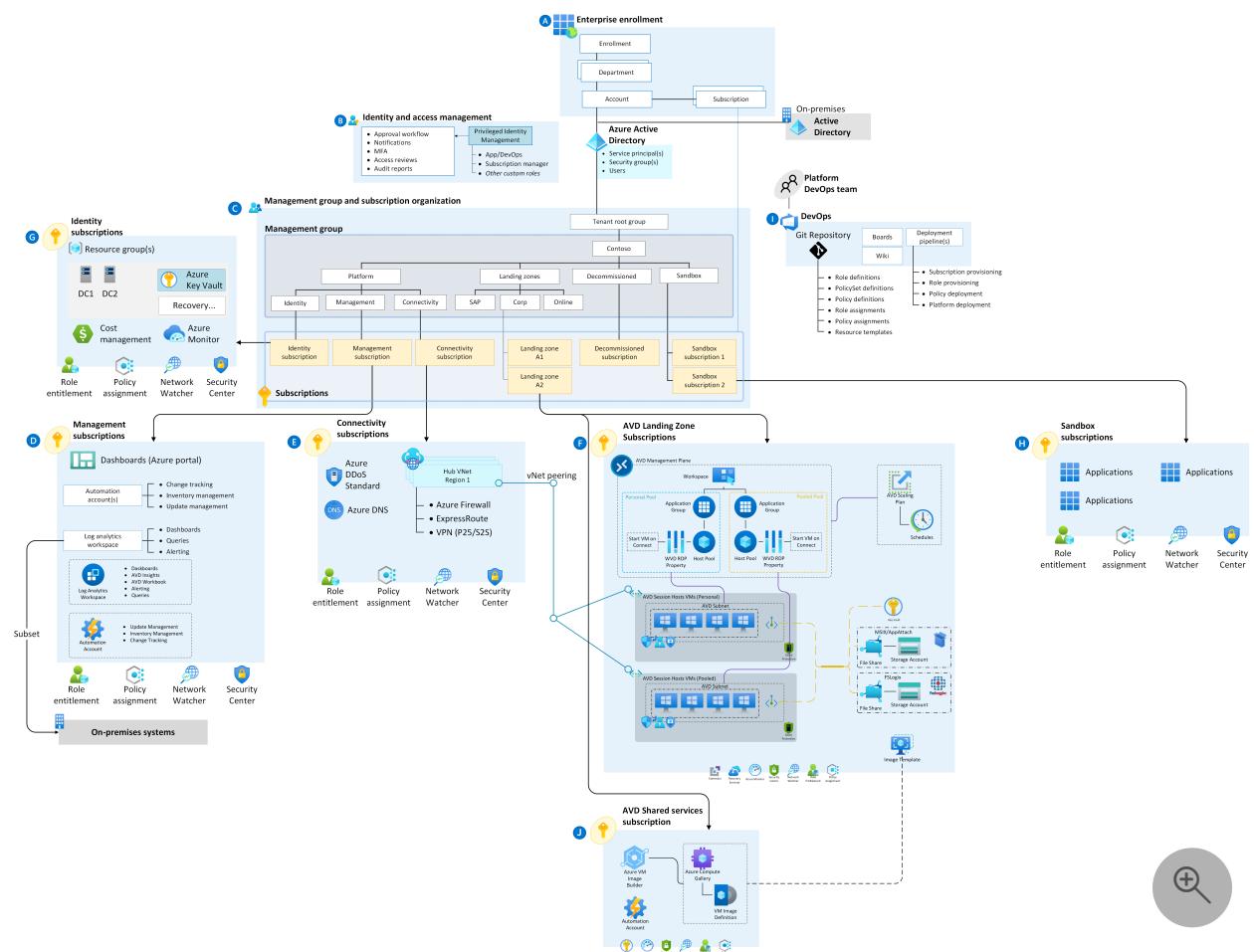
If you understand Azure landing zones, you can skip ahead to the next section. If not, here are some concepts to review before proceeding:

- Abstractly speaking, a ***landing zone*** helps you plan for and design an Azure deployment, by conceptualizing a designated area for placement and integration of resources. There are [two types of landing zones](#):
 - ***platform landing zone***: provides centralized enterprise-scale foundational services for workloads and applications.
 - ***application landing zone***: provides services specific to an application or workload.
- Concretely, a landing zone can be viewed through two lenses:
 - **reference architecture**: a specific design that illustrates resource deployment to one or more Azure subscriptions, which meet the requirements of the landing zone.
 - **reference implementation**: artifacts that deploy Azure resources into the landing zone subscription(s), according to the reference architecture. Many landing zones offer multiple deployment options, but the most common is a ready-made Infrastructure as Code (IaC) template referred to as a ***landing zone accelerator***. Accelerators automate and accelerate the deployment of a reference implementation, using IaC technology such as ARM, Bicep, Terraform, and others.
- A workload deployed to an application landing zone integrates with and is dependent upon services provided by the platform landing zone. These infrastructure services run workloads such as networking, identity access management, policies, and monitoring. This operational foundation enables migration, modernization, and innovation at enterprise-scale in Azure.

In summary, [Azure landing zones](#) provide a destination for cloud workloads, a prescriptive model for managing workload portfolios at scale, and consistency and governance across workload teams.

Reference architecture

The [enterprise-scale landing zone for Azure Virtual Desktop](#) is part of the "Desktop virtualization" scenario article series in the Azure Cloud Adoption Framework. The series provides compatibility requirements, design principles, and deployment guidance for the landing zone. They also serve as the reference architecture for an enterprise-scale implementation, ensuring the environment is capable of hosting desktops and any supporting workloads.



Design principles

Like other landing zones, the enterprise-scale Azure Virtual Desktop landing zone was designed using a core set of [Cloud Adoption Framework design principles](#) and guided by common [design areas](#).

Design areas for the Azure Virtual Desktop landing zone are indicated with letters "A" through "J" in the diagram, to illustrate the hierarchy of resource organization:

Legend	Design area	Objective
A	Enterprise enrollment	Proper tenant creation, enrollment, and billing setup are important early steps.
B, G	Identity and access management	Identity and access management is a primary security boundary in the public cloud. It's the foundation for any secure and fully compliant architecture.
C-H, J	Resource organization	As cloud adoption scales, considerations for subscription design and management group hierarchy have an impact on governance, operations management, and adoption patterns.
C-H, J	Management and monitoring	For stable, ongoing operations in the cloud, a management baseline is required to provide visibility, operations compliance, and protect and recover capabilities.
E, F	Network topology and connectivity	Networking and connectivity decisions are an equally important foundational aspect of any cloud architecture.
G, F, J	Business continuity and disaster recovery	Automate auditing and enforcement of governance policies.
F, J	Security governance and compliance	Implement controls and processes to protect your cloud environments.
I	Platform automation and DevOps	Align the best tools and templates to deploy your landing zones and supporting resources.

Reference implementation

The Azure Virtual Desktop landing zone accelerator deploys resources for an enterprise-scale reference implementation of Azure Virtual Desktop. This implementation is based on the reference architecture discussed in the previous section.

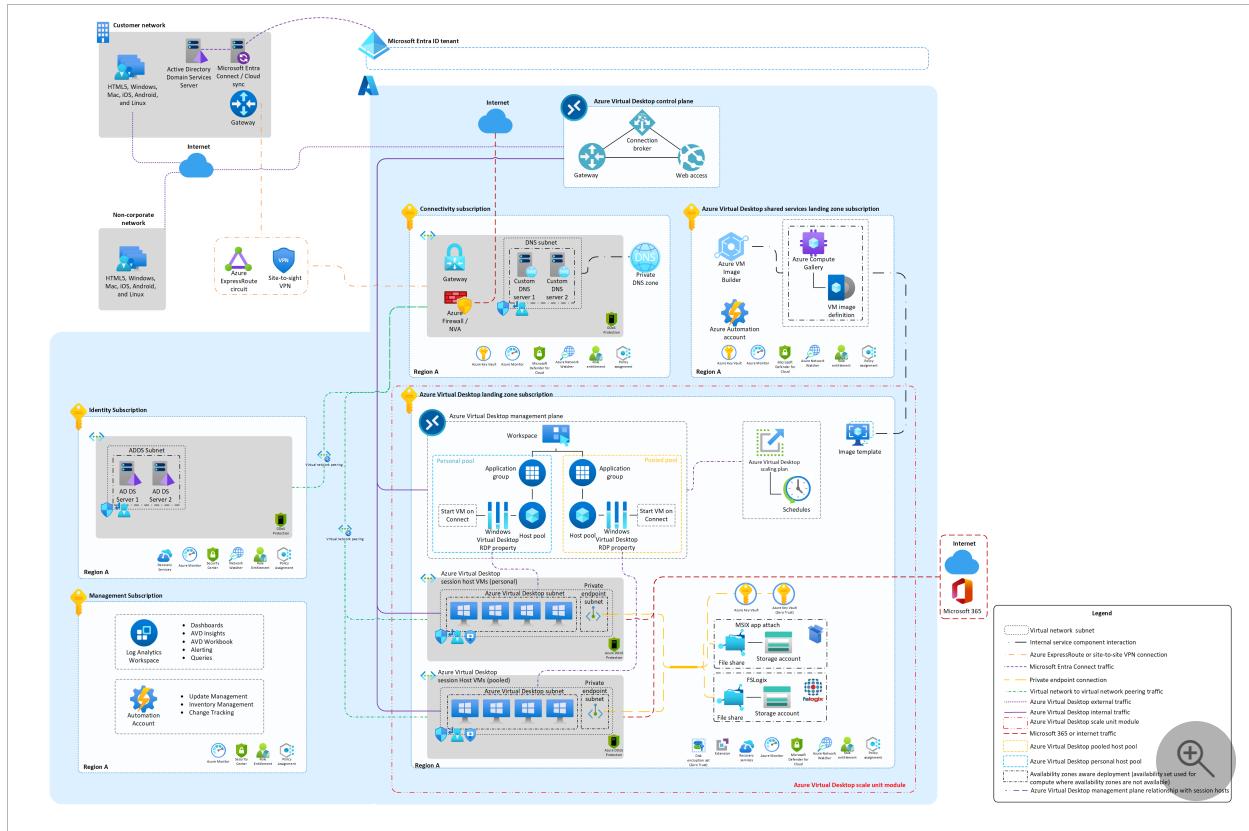
Architecture

ⓘ Important

The accelerator deploys resources into the Azure Virtual Desktop landing zone subscriptions identified in the following architecture diagram: **AVD LZ Subscription**, and **AVD Shared Services LZ Subscription**.

We strongly recommend deployment of the appropriate [Cloud Adoption Framework platform landing zone](#) first, to provide the enterprise-scale foundation services required by the resources deployed by the accelerator. Refer

to the [baseline deployment prerequisites](#) to review the full set of prerequisites and requirements for the accelerator.



[Download a Visio diagram](#) of this architecture

Accelerator overview

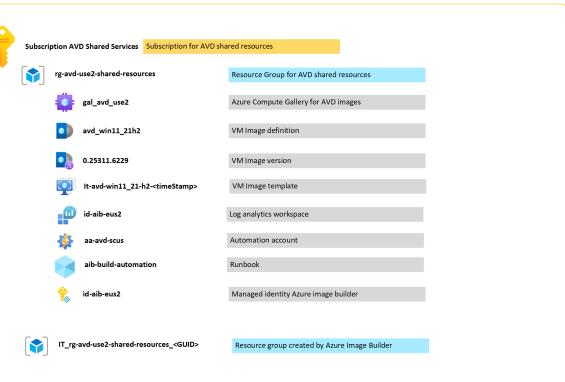
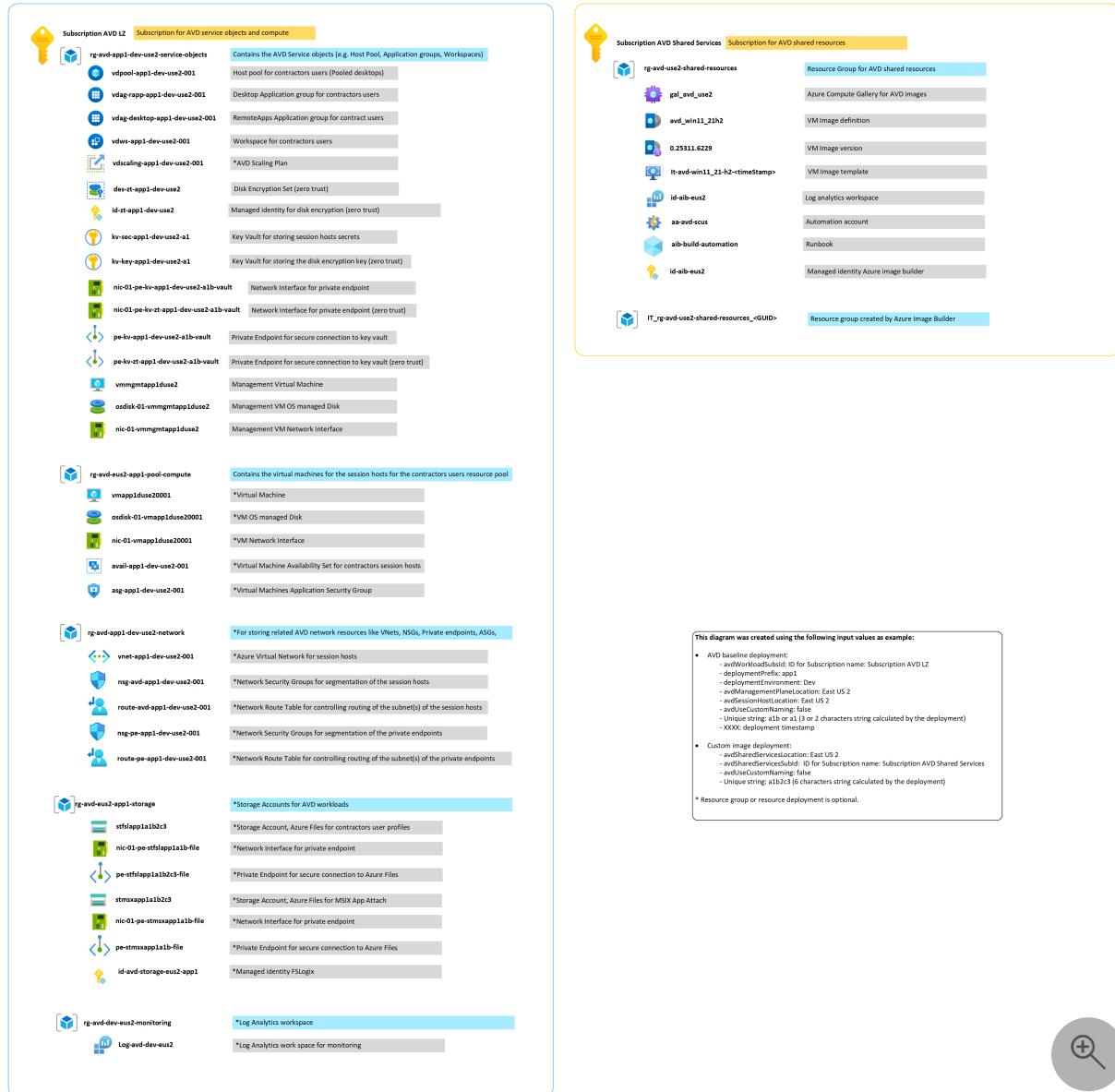
The  [Azure Virtual Desktop landing zone accelerator](#) supports multiple deployment scenarios depending on your requirements. Each deployment scenario supports both greenfield and brownfield deployments, and provides multiple IaC template options:

- Azure portal UI (ARM template)
- Azure CLI or Azure PowerShell (Bicep/ARM template)
- Terraform template

The accelerator uses resource naming automation based on the following recommendations:

- Microsoft Cloud Adoption Framework (CAF) best practices for naming convention
- The [recommended abbreviations for Azure resource types](#)
- The [minimum suggested tags](#).

Before proceeding with the deployment scenarios, familiarize yourself with the Azure resource [naming, tagging, and organization](#) used by the accelerator:



[Download a full-sized image of this diagram](#)

Accelerator deployment

To continue with deployment, choose the following deployment scenario tab that best matches your requirements:

Baseline deployment

The baseline deployment deploys the Azure Virtual Desktop resources and dependent services that allow you to establish an Azure Virtual Desktop baseline.

This deployment scenario includes the following items:

- Azure Virtual Desktop resources, including one workspace, two application groups, a scaling plan, a host pool, and session host virtual machines
- An Azure Files share integrated with your identity service
- Azure Key Vault for secret, key, and certificate management
- Optionally, a new Azure Virtual Network with baseline Network Security Groups (NSG), Application Security Groups (ASG), and route tables

When you're ready for deployment, complete the following steps:

1. Review the [get started](#) document for details on prerequisites, planning information, and a discussion on what is deployed.
2. Optionally, refer to the **Custom image build deployment** tab to build an updated image for your Azure Virtual Desktop host sessions.
3. Continue with the [baseline deployment steps](#). If you created a custom Azure Compute Gallery image in the previous step, be sure to select "Compute gallery" for **OS image source** and select the correct **Image** on the **Session hosts** page:

OS selection

OS image source * ⓘ

Compute Gallery

Image *



Subscription vending implementation guidance

Azure

This article provides implementation guidance for subscription vending automation. Subscription vending standardizes the process for requesting, deploying, and governing subscriptions so that application teams can deploy their workloads faster.

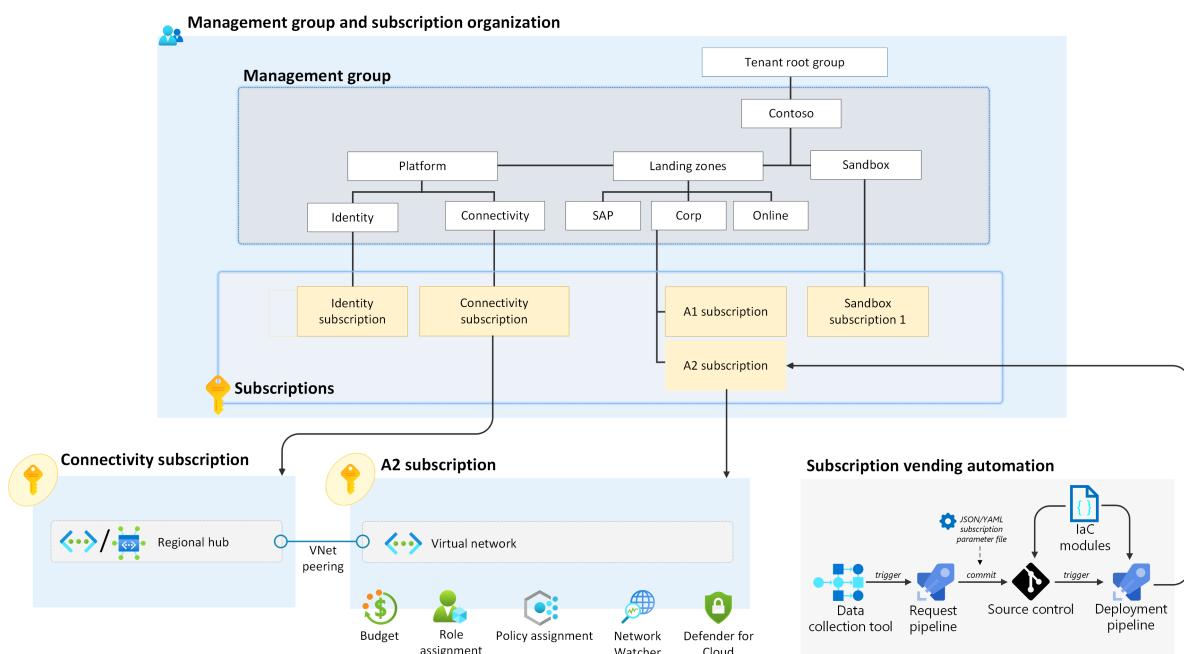


Figure 1. A subscription vending implementation in an example Azure environment.



We created subscription vending [Bicep](#) and [Terraform](#) modules that you should use as a starting point. You should modify the templates to fit your implementation needs. For more information on the subscription vending process, see [Subscription vending overview](#).

https://www.youtube-nocookie.com/embed/OoC_0afxAvg

Architecture

You should architect your subscription vending automation to accomplish three primary tasks. Subscription vending automation should (1) collect subscription request data, (2) initiate platform automation, and (3) create the subscription by using infrastructure-as-code. There are several approaches for implementing subscription vending automation

to accomplish these three tasks. The example implementation (*figure 2*) shows one approach that uses a Gitflow. The Gitflow design aligns with the declarative approach that many platform teams use to manage the platform.

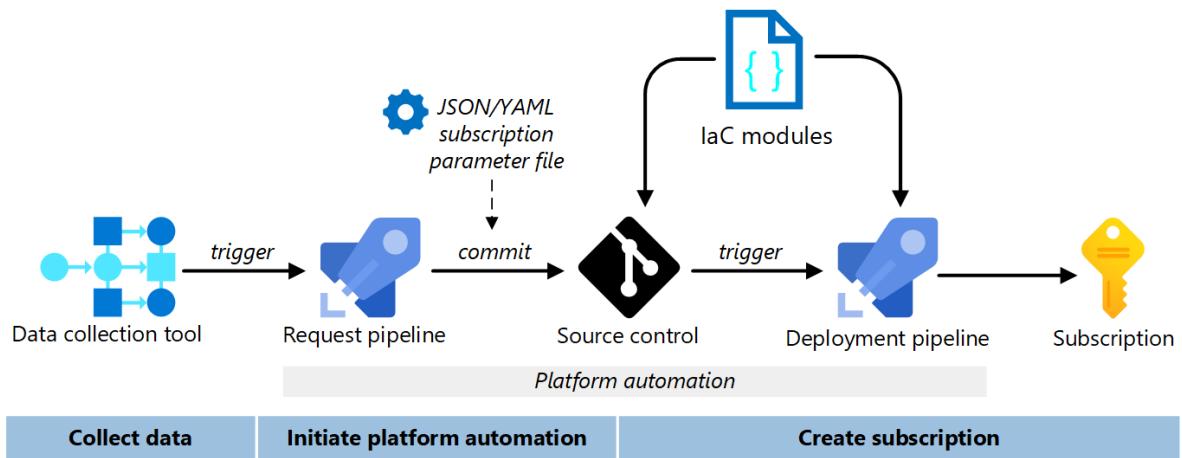


Figure 2. Example implementation of subscription vending automation.

In the example implementation (*figure 2*), the *data collection tool* gathers subscription request data. When the subscription request receives approval, it initiates the platform automation. The *platform automation* consists of the request pipeline, source control, and deployment pipeline. The *request pipeline* creates a JSON or YAML subscription parameter file with the data from the data collection tool. The request pipeline also creates a new branch, commits the subscription parameter file, and opens a pull request in *source control*. The new branch merges with the main branch in source control. The merge triggers the *deployment pipeline* to create the subscription with the infrastructure-as-code modules.

The deployment should place the *subscription* in the correct management group based on the governance requirements (see *figure 1*). The deployment creates a preliminary subscription budget as the foundation for cost management. Based on the needs of the workload, the deployment could create an empty virtual network and configure peering to a regional hub. The platform team should hand off the subscription to the application team after creation and configuration. The application team should update the subscription budget and create the workload resources.

Collect data

The goal of collecting data is to receive business approval and define the values of the JSON/YAML subscription parameter file. You should use a data collection tool to collect the required data when the application team submits the subscription request. The data collection tool should interface with other systems in the subscription vending workflow to initiate the platform automation.

Use a data collection tool. You can use an IT Service Management (ITSM) tool to collect the data or build a customer portal with a low-code or no-code tool like [Microsoft PowerApps](#). The data collection tool should provide business logic to approve or deny the subscription request.

Collect the required data. You need to collect enough data to define the values of the JSON/YAML subscription parameter so that you can automate the deployment. The specific values you collect depend on your needs. You should capture the request authorizer, cost center, and networking requirements (internet or on-premises connectivity). It might be helpful to ask the application team for anticipated workload components (application platform, data requirements), data sensitivity, and number of environments (development, test, preproduction, production).

Validate data. You should validate data during the data collection process. It's harder to address issues later in the platform automation phases.

Create a trackable request. Your data collection tool should create a logged and trackable request for a new subscription (for example, a ticket in an ITSM tool). The request should contain all necessary data to fulfill the requirements of that subscription. You should bind the business logic and authorization tracking to the request.

Interface with other internal systems. Where needed, the data collection tool should interface with other tools or systems in your organization. The goal is to enrich the request with data from other systems. You might need identity, finance, security, and networking data to execute the automation. For example, the automation could interface with an IP address management (IPAM) tool to reserve the right IP address space.

Create a trigger. When the subscription request receives approval, the data transfer should trigger the platform automation. It's best to create a push notification with the necessary data from your data collection tool. You might need a middleware layer, such as Azure Functions or Azure Logic Apps, to initiate the process.

Initiate platform automation

The notification and data from the data collection tool should trigger the platform automation. The goal of platform automation is to create a JSON/YAML subscription parameter file, merge the file to the main branch, and deploy it with the infrastructure-as-code modules to create the subscription. The platform team should own and maintain the platform automation. The platform automation in the example implementation consists of the request pipeline, source control, and deployment pipeline (see *figure 2*).

Use JSON or YAML files. You should use structured data files (JSON or YAML) to store the data to create a subscription. You should document the structure of the file and make it extensible to support future needs. For example, the following JSON code snippet defines the subscription parameter values for one of the Bicep modules in GitHub.

```
JSON

{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "subscriptionDisplayName": {
            "value": "sub-bicep-lz-vending-example-001"
        },
        "subscriptionAliasName": {
            "value": "sub-bicep-lz-vending-example-001"
        },
        "subscriptionBillingScope": {
            "value": "providers/Microsoft.Billing/billingAccounts/1234567/enrollmentAccounts/123456"
        },
        // Insert more parameters here
    }
}
```

See entire file [↗](#). For more examples, see [Bicep examples ↗](#) and [Terraform examples ↗](#)

Use one file per subscription request. The subscription is the unit of deployment in the subscription vending process, so each subscription request should have one dedicated subscription parameter file.

Use a pull request system. The Gitflow process that creates the subscription parameter file should automate the following steps:

1. Create a new branch for each subscription request.
2. Use the data collected to create a single YAML/JSON subscription parameter file for the new subscription in the branch.
3. Create a pull request from your branch into `main`.
4. Update the data collection tool with a state change and reference to this pull request.

The *request pipeline* in the example implementation executes these steps (see *figure 2*). You could also use a code-based solution hosted in Azure if the workflow is complex.

Validate the subscription parameter file. The pull request should trigger a linting process to validate the request data. The goal is to ensure the deployment is successful. It should validate the YAML/JSON subscription parameter file. It could also verify that the IP address range is still available. You might also want to add a manual review gate with human intervention. They could perform the final review and make changes to the subscription parameter file. The output should be a JSON/YAML subscription parameter file with all the data to create a subscription.

Trigger the deployment pipeline. When the pull request merges into the `main` branch, the merge should trigger the deployment pipeline.

Create a subscription

The last task of the subscription vending automation is to create and configure the new subscription. The example implementation uses the *deployment pipeline* to deploy the infrastructure-as-code module with the JSON/YAML subscription parameter file (see figure 2).

Use infrastructure as code. Your deployment should use infrastructure as code to create the subscription. The platform team should create and maintain these templates to ensure proper governance. You should use the subscription vending [Bicep](#) and [Terraform](#) modules and modify them to fit your implementation needs.

Use a deployment pipeline. The deployment pipeline orchestrates the creation and configuration of the new subscription. The pipeline should execute the following tasks:

[] Expand table

Task category	Pipeline task
Identity	<ul style="list-style-type: none">• Create or update Microsoft Entra resources to represent subscription ownership.• Configure privileged workload identities for workload team deployments.
Governance	<ul style="list-style-type: none">• Place in management group hierarchy.• Assign subscription owner.• Configure subscription-level role-based access controls (RBACs) to configured security groups.

Task category	Pipeline task
	<ul style="list-style-type: none"> • Assign subscription-level Azure Policy. • Configure the Microsoft Defender for Cloud enrollment.
Networking	<ul style="list-style-type: none"> • Deploy virtual networks. • Configure virtual network peering to platform resources (regional hub).
Budgets	<ul style="list-style-type: none"> • Create budgets for the subscription owners by using the collected data.
Reporting	<ul style="list-style-type: none"> • Update external systems, such as IPAM, to commit to IP reservations. • Update the data collection tool request with final subscription name and globally unique identifier (GUID). • Notify the application team that the subscription is ready.

You need a commercial agreement to create a subscription programmatically. If you don't have a commercial agreement, you need to introduce a manual process to create the subscription but can still automate all other aspects of subscription configuration.

Establish a workload identity. The deployment pipeline needs permission to perform these operations with all the systems it interfaces with. You should either use managed identity or OpenID Connect (OIDC) to authenticate to Azure.

Post-deployment

The subscription vending automation ends with subscription creation and configuration. The platform team should hand off the new subscription to the application team after creation. The application team should update the subscription budget, create the workload resources, and deploy the workload. The platform team controls the governance of the subscription and manages changes to subscription governance over time.

Enforce cost management. Subscription budgets provide notifications that are critical to cost management. The deployment should create a preliminary subscription budget based on the subscription request data. The application team receives the subscription. They should update the budget to meet the needs of the workload. For more information, see:

- [Create and manage budgets](#)
- [Manage costs with Azure Budgets](#)
- [Cost allocation](#)

- [Track costs across business units, environments, or projects](#)

Manage subscription governance. You should update the subscription as the governance requirements of the workload change. For example, you might need to move a subscription to a different management group. You should build automation for some of these routine operations. For more information, see:

- [Moving management groups and subscription](#)
- [Keep policies and policy initiatives up to date](#)
- [Resource tagging](#)
- [Tailor the Azure landing zone architecture to meet requirements](#)

Next steps

Subscription vending simplifies and standardizes the subscription creation process and places it under the governance of the organization. You should implement subscription vending automation to help your application teams access application landing zones faster and onboard workloads quicker. For more information, see:

- [Bicep modules ↗](#)
- [Terraform modules ↗](#)
- [Subscription vending overview](#)
- [Azure landing zone overview](#)

Azure application architecture fundamentals

Article • 12/16/2022

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

Introduction

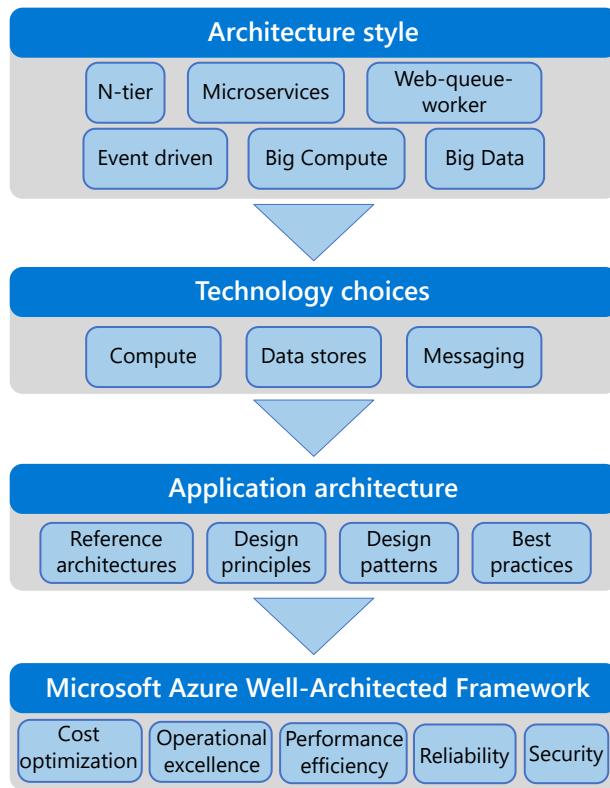
The cloud is changing how applications are designed and secured. Instead of monoliths, applications are decomposed into smaller, decentralized services. These services communicate through APIs or by using asynchronous messaging or eventing. Applications scale horizontally, adding new instances as demand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

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

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

How this guidance is structured

The Azure application architecture fundamentals guidance is organized as a series of steps, from the architecture and design to implementation. For each step, there is supporting guidance that will help you design 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 architectural styles. There are benefits and challenges to each.

Learn more: [Architecture styles](#)

Technology choices

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

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

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

Design the architecture

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

Reference architectures

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

Design principles

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

Design patterns

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

Best practices

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

Security best practices

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

Quality pillars

A successful cloud application will focus on five pillars of software quality: Reliability, Security, Cost Optimization, Operational Excellence, and Performance Efficiency.

Leverage the [Microsoft Azure Well-Architected Framework](#) to assess your architecture across these five pillars.

Next steps

[Architecture styles](#)

Architecture styles

Article • 12/16/2022

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

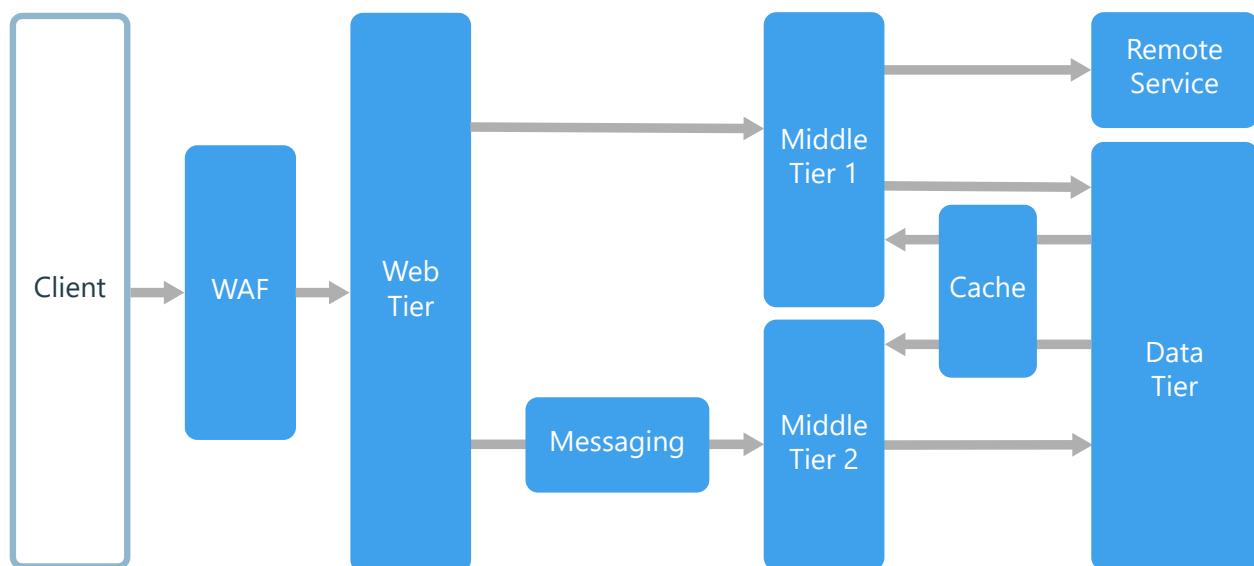
We have identified a set of architecture styles that are commonly found in cloud applications. The article for each style includes:

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

A quick tour of the styles

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

N-tier

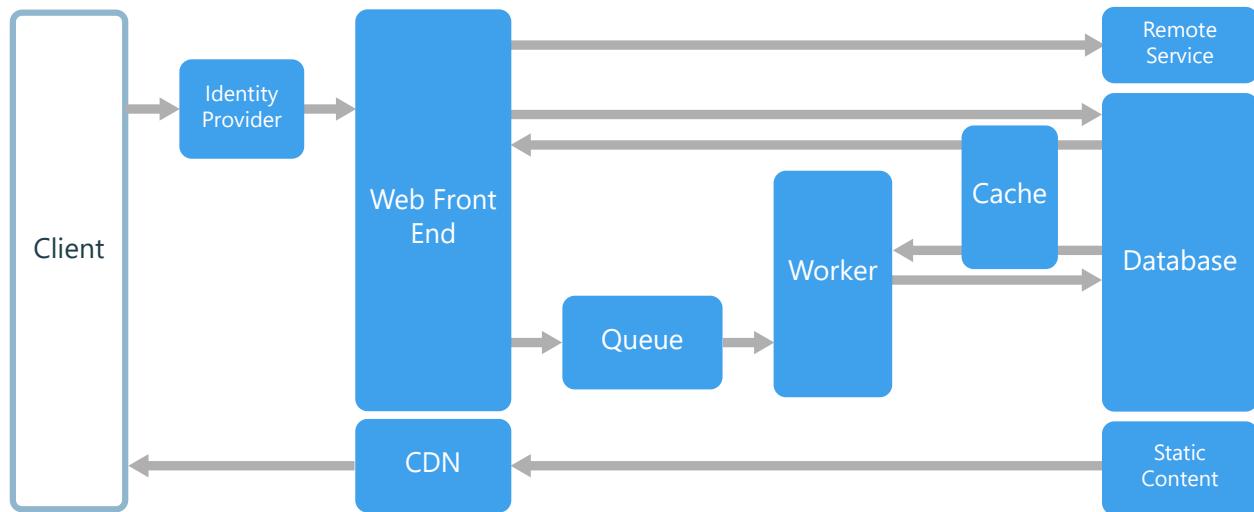


[N-tier](#) is a traditional architecture for enterprise applications. Dependencies are managed by dividing the application into *layers* that perform logical functions, such as presentation, business logic, and data access. A layer can only call into layers that sit below it. However, this horizontal layering can be a liability. It can be hard to introduce

changes in one part of the application without touching the rest of the application. That makes frequent updates a challenge, limiting how quickly new features can be added.

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

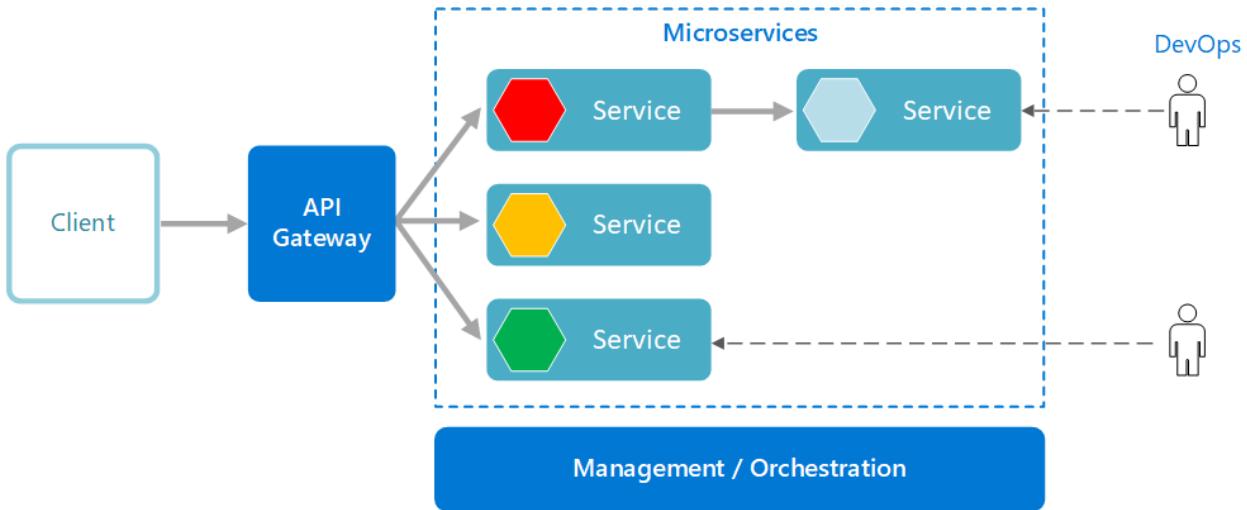
Web-Queue-Worker



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

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

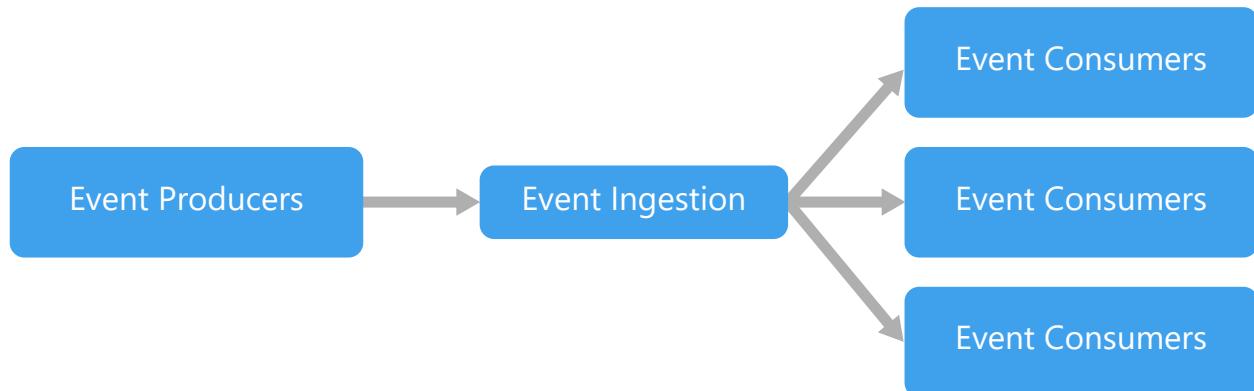
Microservices



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

Each service can be built by a small, focused development team. Individual services can be deployed without a lot of coordination between teams, which encourages frequent updates. A microservice architecture is more complex to build and manage than either N-tier or web-queue-worker. It requires a mature development and DevOps culture. But done right, this style can lead to higher release velocity, faster innovation, and a more resilient architecture.

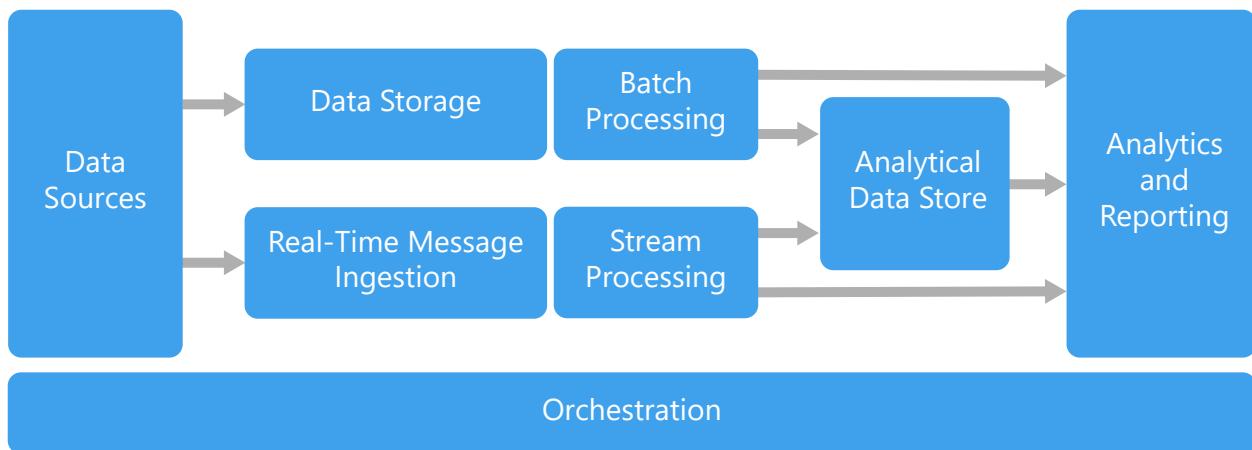
Event-driven architecture



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

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

Big Data, Big Compute



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

Architecture styles as constraints

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

For example, the constraints in microservices include:

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

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

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

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

Architecture style	Dependency management	Domain type
N-tier	Horizontal tiers divided by subnet	Traditional business domain. Frequency of updates is low.
Web-queue-worker	Front and backend jobs, decoupled by async messaging.	Relatively simple domain with some resource intensive tasks.
Microservices	Vertically (functionally) decomposed services that call each other through APIs.	Complicated domain. Frequent updates.
Event-driven architecture	Producer/consumer. Independent view per sub-system.	IoT and real-time systems.
Big data	Divide a huge dataset into small chunks. Parallel processing on local datasets.	Batch and real-time data analysis. Predictive analysis using ML.
Big compute	Data allocation to thousands of cores.	Compute intensive domains such as simulation.

Consider challenges and benefits

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

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

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

unacceptable latency or create network congestion (for example, in a microservices architecture).

- **Manageability.** How hard is it to manage the application, monitor, deploy updates, and so on?

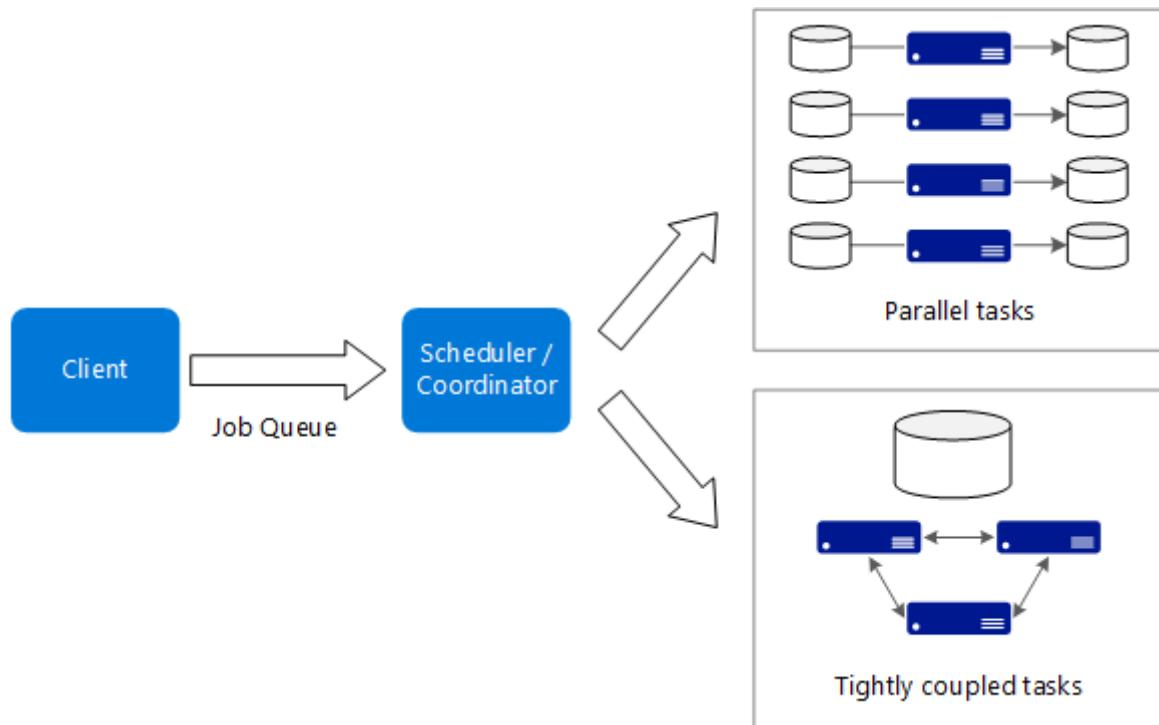
Related resources

- [Ten design principles for Azure applications](#)
- [Build applications on the Microsoft Cloud](#)
- [Best practices in cloud applications](#)
- [Cloud Design Patterns](#)
- [Performance testing and antipatterns for cloud applications](#)
- [Architect multitenant solutions on Azure](#)
- [Mission critical workload architecture on Azure](#)
- [Architecture for startups](#)

Big compute architecture style

Azure Azure Batch

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



Here are some typical characteristics of big compute applications:

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

When to use this architecture

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

Benefits

- High performance with "[embarrassingly parallel](#)" processing.
- Can harness hundreds or thousands of computer cores to solve large problems faster.
- Access to specialized high-performance hardware, with dedicated high-speed InfiniBand networks.
- You can provision VMs as needed to do work, and then tear them down.

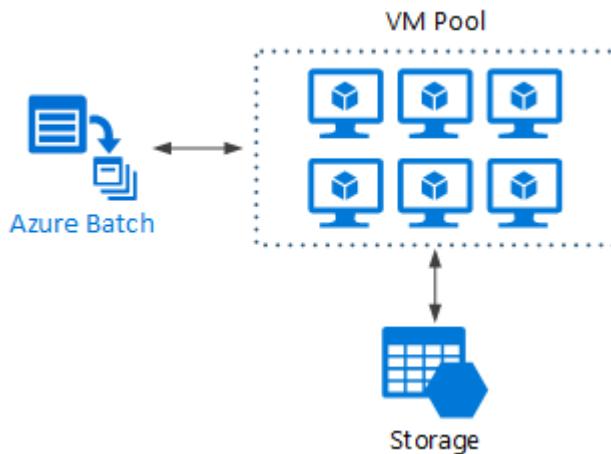
Challenges

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

Big compute using Azure Batch

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

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

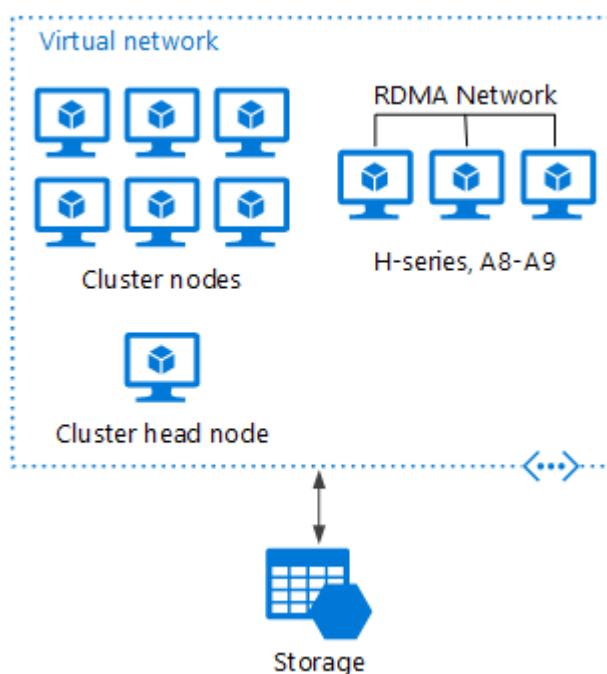


Big compute running on Virtual Machines

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

HPC Pack deployed to Azure

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

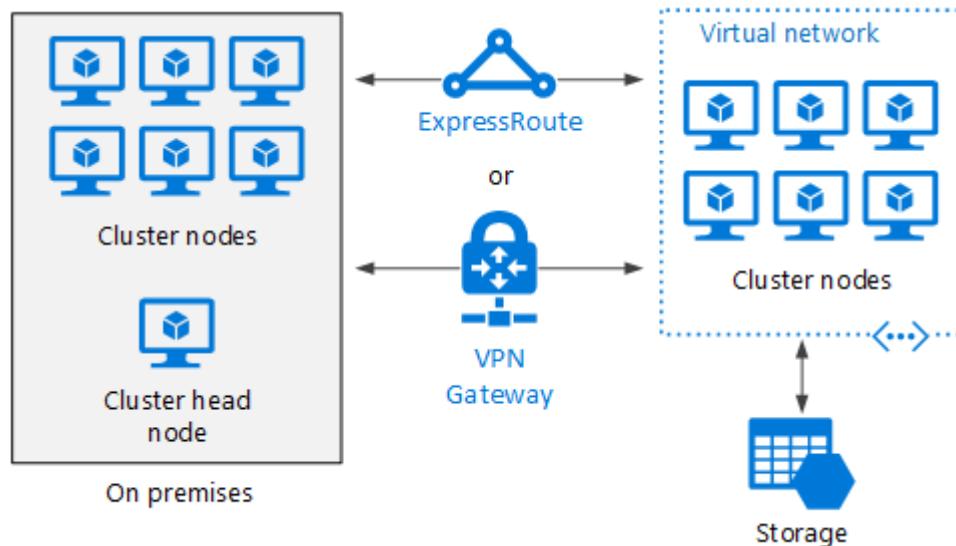


The head node provides management and job scheduling services to the cluster. For tightly coupled tasks, use an RDMA network that provides very high bandwidth, low

latency communication between VMs. For more information, see [Deploy an HPC Pack 2016 cluster in Azure](#).

Burst an HPC cluster to Azure

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



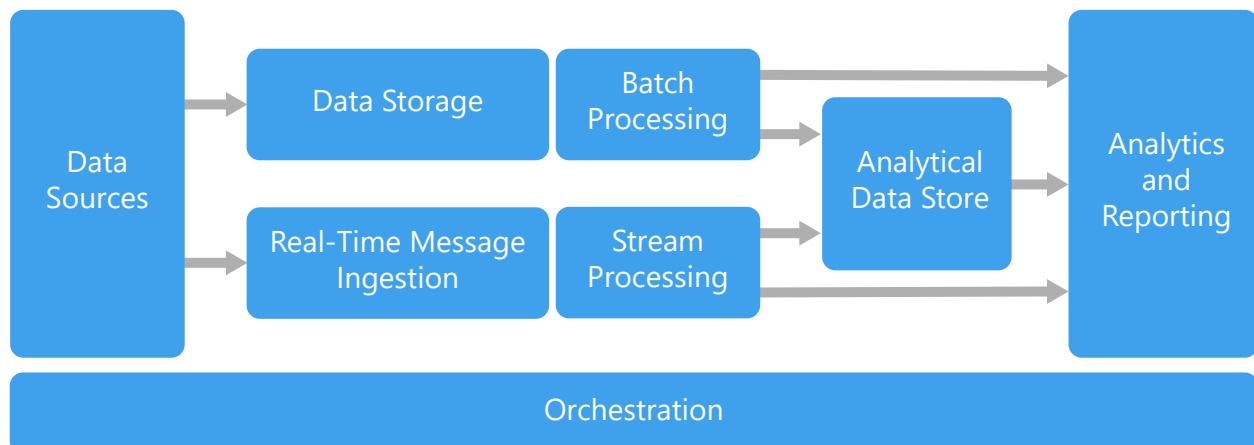
Next steps

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

Big data architecture style

Azure Data Lake Analytics Azure IoT

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 Spark Streaming in an HDInsight cluster.
- **Analytical data store:** Many big data solutions prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools. The analytical data store used to serve these queries can be a Kimball-style relational data warehouse, as seen in most traditional business intelligence (BI) solutions. Alternatively, the data could be presented through a low-latency NoSQL technology such as HBase, or an interactive Hive database that provides a metadata abstraction over data files in the distributed data store. Azure Synapse Analytics provides a managed service for large-scale, cloud-based data warehousing. HDInsight supports Interactive Hive, HBase, and Spark SQL, which can also be used to serve data for analysis.
- **Analysis and reporting:** The goal of most big data solutions is to provide insights into the data through analysis and reporting. To empower users to analyze the data, the architecture may include a data modeling layer, such as a multidimensional OLAP cube or tabular data model in Azure Analysis Services. It might also support self-service BI, using the modeling and visualization technologies in Microsoft Power BI or Microsoft Excel. Analysis and reporting can also take the form of interactive data exploration by data scientists or data analysts. For these scenarios, many Azure services support analytical notebooks, such as Jupyter, enabling these users to leverage their existing skills with Python or R. For large-scale data exploration, you can use Microsoft R Server, either standalone or with Spark.

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

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

- Managed services, including Azure Data Lake Store, Azure Data Lake Analytics, Azure Synapse Analytics, Azure Stream Analytics, Azure Event Hubs, Azure IoT Hub, and Azure Data Factory.
- Open source technologies based on the Apache Hadoop platform, including HDFS, HBase, Hive, Spark, 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 Azure 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.

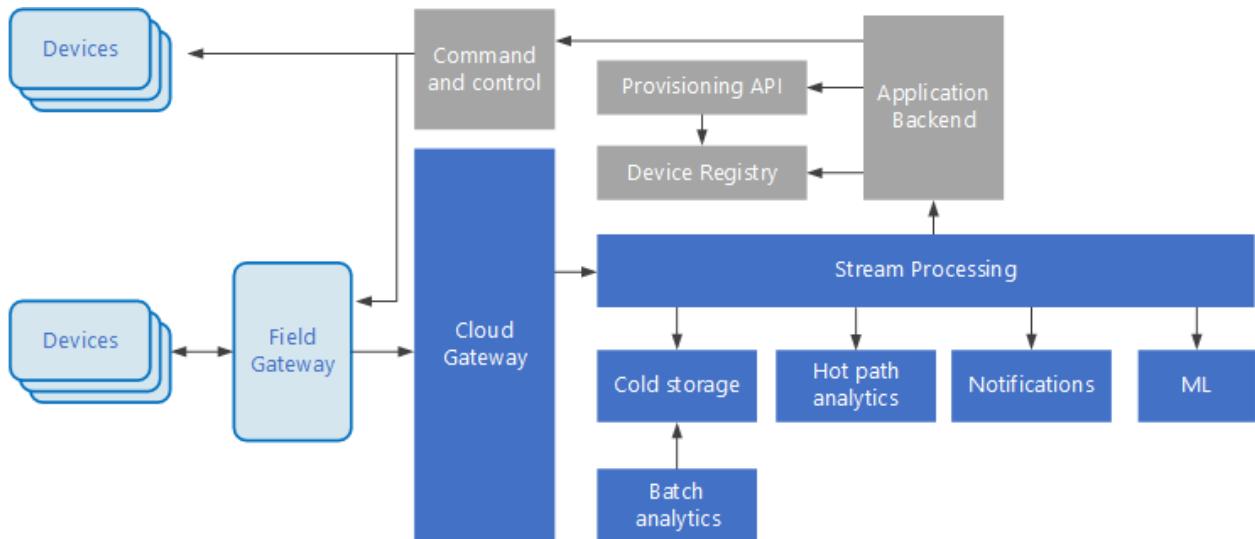
Also, partitioning tables that are used in Hive, U-SQL, or SQL queries can significantly improve query performance.

- **Apply schema-on-read semantics.** Using a data lake lets you combine storage for files in multiple formats, whether structured, semi-structured, or unstructured. Use *schema-on-read* semantics, which project a schema onto the data when the data is processing, not when the data is stored. This builds flexibility into the solution, and prevents bottlenecks during data ingestion caused by data validation and type checking.
- **Process data in-place.** Traditional BI solutions often use an extract, transform, and load (ETL) process to move data into a data warehouse. With larger volumes data, and a greater variety of formats, big data solutions generally use variations of ETL, such as transform, extract, and load (TEL). With this approach, the data is processed within the distributed data store, transforming it to the required structure, before moving the transformed data into an analytical data store.
- **Balance utilization and time costs.** For batch processing jobs, it's important to consider two factors: The per-unit cost of the compute nodes, and the per-minute cost of using those nodes to complete the job. For example, a batch job may take eight hours with four cluster nodes. However, it might turn out that the job uses all four nodes only during the first two hours, and after that, only two nodes are required. In that case, running the entire job on two nodes would increase the total job time, but would not double it, so the total cost would be less. In some business scenarios, a longer processing time may be preferable to the higher cost of using underutilized cluster resources.
- **Separate cluster resources.** When deploying HDInsight clusters, you will normally achieve better performance by provisioning separate cluster resources for each type of workload. For example, although Spark clusters include Hive, if you need to perform extensive processing with both Hive and Spark, you should consider deploying separate dedicated Spark and Hadoop clusters. Similarly, if you are using HBase and Storm for low latency stream processing and Hive for batch processing, consider separate clusters for Storm, HBase, and Hadoop.
- **Orchestrate data ingestion.** In some cases, existing business applications may write data files for batch processing directly into Azure storage blob containers, where they can be consumed by HDInsight or Azure Data Lake Analytics. However, you will often need to orchestrate the ingestion of data from on-premises or external data sources into the data lake. Use an orchestration workflow or pipeline, such as those supported by Azure Data Factory or Oozie, to achieve this in a predictable and centrally manageable fashion.

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

IoT architecture

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



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

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

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

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

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

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

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

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

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

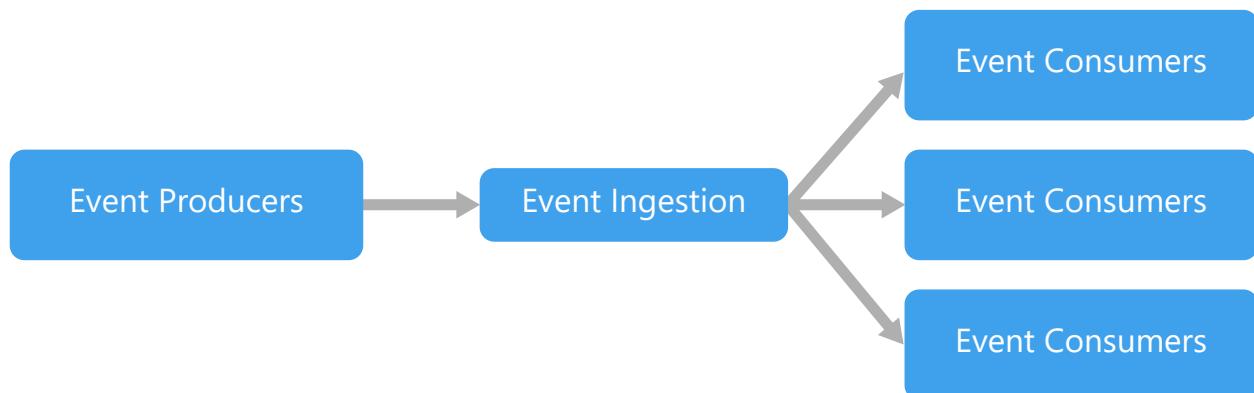
Next steps

- Learn more about [big data architectures](#).
- Learn more about [IoT solutions](#).

Event-driven architecture style

Azure IoT Azure Stream Analytics

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 [publish/subscribe](#) (also called *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 can't be replayed, and new subscribers don't 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.

- **Basic event correlation.** A consumer needs to process a small number of discrete business events, usually correlated by some identifier, persisting some information from earlier events to use when processing later events. This pattern is supported by libraries like [NServiceBus](#) and [MassTransit](#).
- **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. 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 [idempotent message processing](#) logic isn't implemented.
- Coordinating messages across services. Business processes often involve multiple services publishing and subscribing to messages to achieve a consistent outcome across a whole workload. [Workflow patterns](#) such as the [Choreography pattern](#) and [Saga Orchestration](#) can be used to reliably manage message flows across various services.

Additional considerations

- The amount of data to include in an event can be a significant consideration that affects both performance and cost. Putting all the relevant information needed for processing in the event itself can simplify the processing code and save additional lookups. Putting the minimal amount of information in an event, like just a couple of identifiers, will reduce transport time and cost, but requires the processing code to look up any additional information it needs. For more information on this, take a look at [this blog post](#).
- While a request is only visible to the request-handling component, events are often visible to multiple components in a workload, even if those components don't or aren't meant to consume them. Operating with an "assume breach" mindset, be mindful of what information you include in events to prevent unintended information exposure.

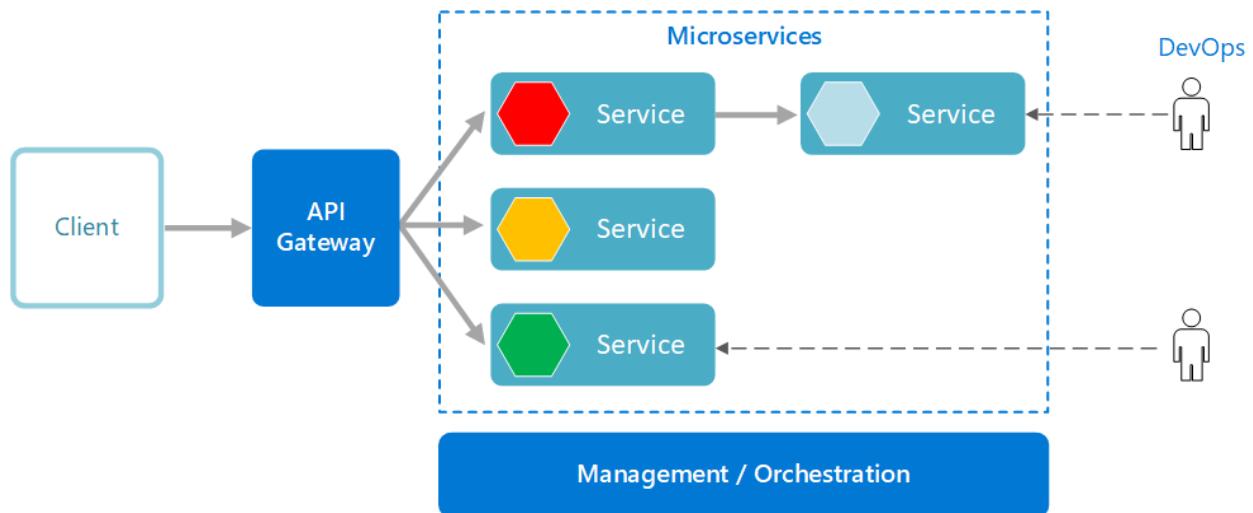
Related resources

- [Community discussion video](#) on the considerations of choosing between choreography and orchestration.

Microservice architecture style

Azure

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



What are microservices?

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

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

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

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

Advantages of using an API gateway include:

- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.
- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.
- Out-of-the-box policies, like for throttling, caching, transformation, or validation.

Benefits

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features might be held up waiting for a bug fix to be integrated, tested, and published.
- **Small, focused teams.** A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend to be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- **Small code base.** In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.

- **Mix of technologies.** Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Fault isolation.** If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly. For example, you can implement the [Circuit Breaker pattern](#), or you can design your solution so that the microservices communicate with each other using [asynchronous messaging patterns](#).
- **Scalability.** Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- **Data isolation.** It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application might all touch the same data, making any alterations to the schema risky.

Challenges

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

- **Complexity.** A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing.** Writing a small service that relies on other dependent services requires a different approach than writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- **Lack of governance.** The decentralized approach to building microservices has advantages, but it can also lead to problems. You might end up with so many different languages and frameworks that the application becomes hard to maintain. It might be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

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

Best practices

- Model services around the business domain.
- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.
- Data storage should be private to the service that owns the data. Use the best storage for each service and data type.
- Services communicate through well-designed APIs. Avoid leaking implementation details. APIs should model the domain, not the internal implementation of the service.
- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.
- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway.
- Keep domain knowledge out of the gateway. The gateway should handle and route client requests without any knowledge of the business rules or domain logic.

Otherwise, the gateway becomes a dependency and can cause coupling between services.

- Services should have loose coupling and high functional cohesion. Functions that are likely to change together should be packaged and deployed together. If they reside in separate services, those services end up being tightly coupled, because a change in one service will require updating the other service. Overly chatty communication between two services may be a symptom of tight coupling and low cohesion.
- Isolate failures. Use resiliency strategies to prevent failures within a service from cascading. See [Resiliency patterns](#) and [Designing reliable applications](#).

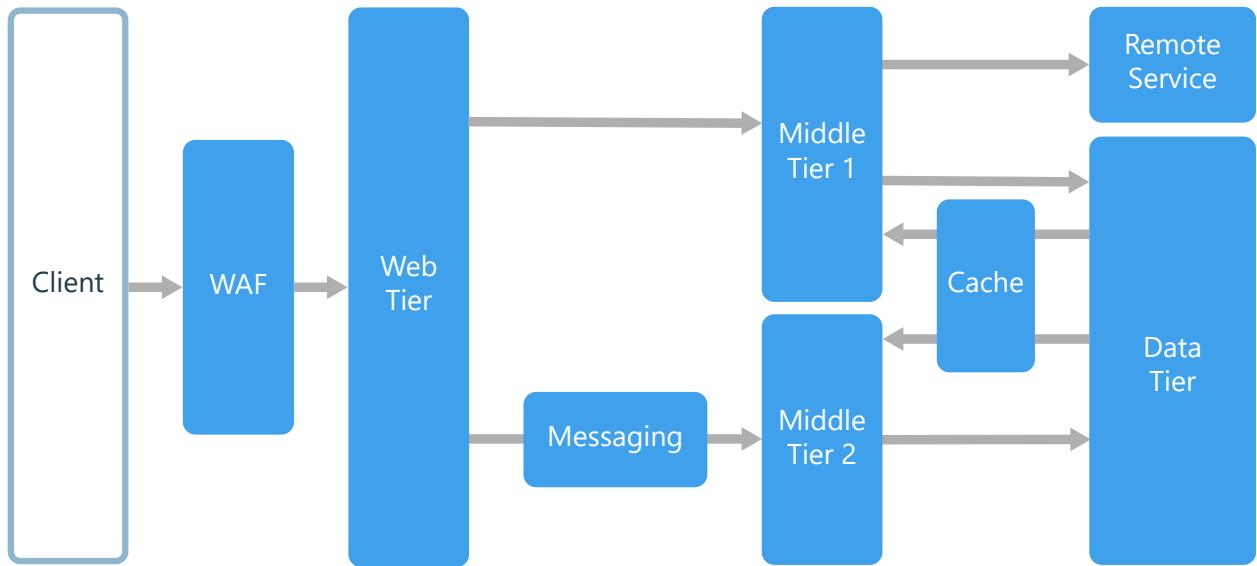
Next steps

For detailed guidance about building a microservices architecture on Azure, see [Designing, building, and operating microservices on Azure](#).

N-tier architecture style

Azure Storage Azure Cloud Services Azure Virtual Machines

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 patterns](#) through a message queue.

Although each layer might be hosted in its own tier, that's not required. Several layers might be hosted on the same tier. Physically separating the tiers improves scalability and resiliency, but also adds latency from the additional network communication.

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

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

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

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

When to use this architecture

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

Consider an N-tier architecture for:

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

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

Benefits

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

Challenges

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

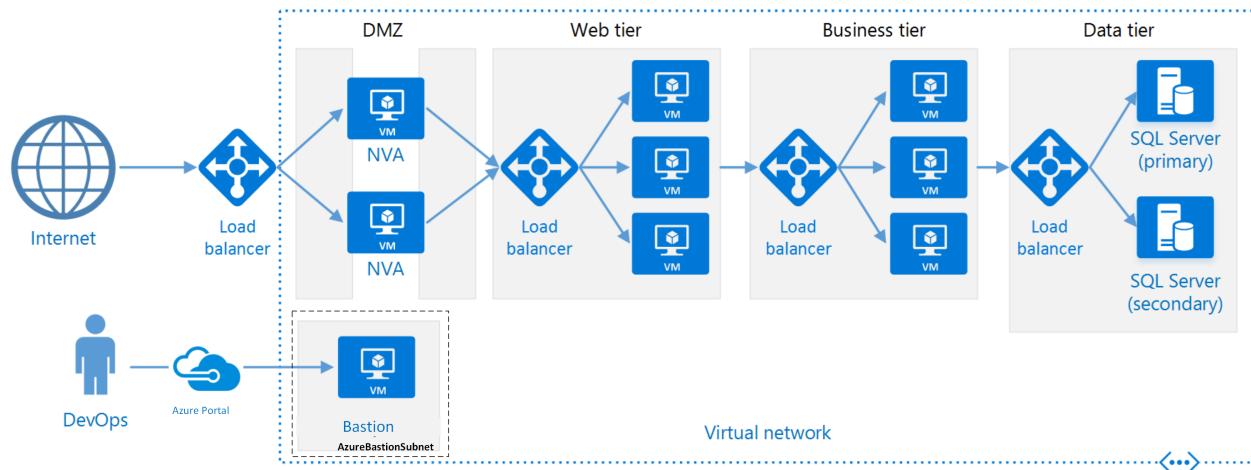
Best practices

- Use autoscaling to handle changes in load. See [Autoscaling best practices](#).
- Use [asynchronous messaging](#) to decouple tiers.
- Cache semistatic data. See [Caching best practices](#).
- Configure the database tier for high availability, using a solution such as [SQL Server Always On availability groups](#).

- Place a web application firewall (WAF) between the front end and the Internet.
- Place each tier in its own subnet, and use subnets as a security boundary.
- Restrict access to the data tier, by allowing requests only from the middle tier(s).

N-tier architecture on virtual machines

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



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

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

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

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

Note

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

application - you could even express that naming in resources you choose to represent that tier (e.g. vmss-appName-business-layer).

Additional considerations

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

- If you need higher availability than the Azure SLA for VMs provides, replicate the application across two regions and use Azure Traffic Manager for failover. For more information, see [Run Windows VMs in multiple regions](#) or [Run Linux VMs in multiple regions](#).

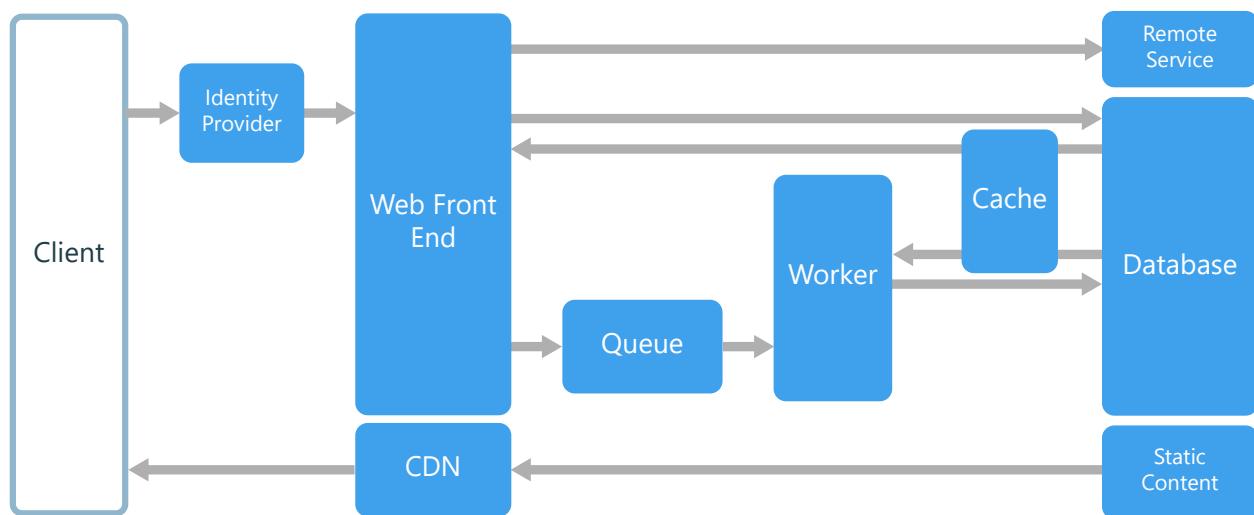
Related Resources

- [N-tier application with Apache Cassandra](#)
- [Windows N-tier application on Azure with SQL Server][n-tier-windows-SQL]
- [Microsoft Learn module: Tour the N-tier architecture style](#)
- [Azure Bastion](#)
- More information on messaging in an N-tier architecture style on [Azure](#) ↗

Web-Queue-Worker architecture style

Azure App Service

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 features 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.
- The web front end can malfunction after successfully persisting to the database but before it emits the messages to the queue. This can result in possible consistency issues as the worker will not perform its part of the logic. Techniques like the [transactional outbox pattern](#) can be used to help mitigate this problem but require changing the routing of outgoing messages to first "loop back" through a separate queue. One library that provides support for this technique is the [NServiceBus Transactional Session](#).

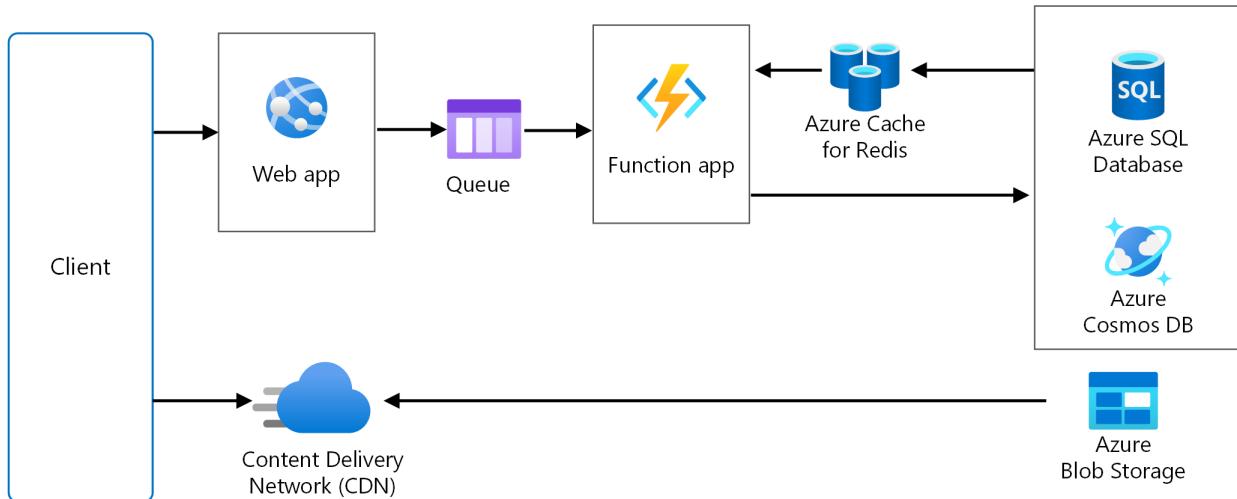
Best practices

- 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][polyglot].
- 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.



[Download a Visio file](#) of this architecture.

Workflow

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

For more information, see the [App Service web application reference architecture](#) and how to [build message-driven business applications with NServiceBus and Azure Service Bus](#).

Other considerations

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

Related resources

- [RESTful web API design](#)
- [Autoscaling](#)
- [Caching guidance](#)
- [CDN guidance](#)
- [Data partitioning guidance](#)
- [Scalable web application](#)
- [Queue-Based Load Leveling pattern](#)

Ten design principles for Azure applications

Article • 11/01/2023

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 an identity service.** Use an identity as a service (IDaaS) platform instead of building or operating your own.
- **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

Article • 07/26/2023

Design your application to be self healing when failures occur

In a distributed system, failures can happen. Hardware can fail. The network can have transient failures. Rarely, an entire service, data center, or even Azure 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 high availability, you might deploy to multiple availability zones in a region. To avoid outages even in the unlikely event of an entire Azure region experiencing disruption, you can automatically fail over to a secondary region during a regional outage. However, that will incur a higher cost and potentially lower performance than a single-region deployment.

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

Recommendations

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

Protect failing remote services (Circuit Breaker). It's good to retry after a transient failure, but if the failure persists, you can end up with too many callers hammering a failing service. This can lead to cascading failures, as requests back up. Use the [Circuit](#)

[Breaker pattern](#) to fail fast (without making the remote call) when an operation is likely to fail.

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

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

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

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

Checkpoint long-running transactions. Checkpoints can provide resiliency if a long-running operation fails. When the operation restarts (for example, it is picked up by another VM), it can be resumed from the last checkpoint. Consider implementing a mechanism that records state information about the task at regular intervals, and save this state in durable storage that can be accessed by any instance of the process running the task. In this way, if the process is shut down, the work that it was performing can be resumed from the last checkpoint by using another instance. There are libraries that provide this functionality, such as [NServiceBus](#) and [MassTransit](#). They transparently persist state, where the intervals are aligned with the processing of messages from queues in Azure Service Bus.

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

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

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

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

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

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

Consider using availability zones. Many Azure regions provide [availability zones](#), which are isolated sets of data centers within the region. Some Azure services can be deployed *zonally*, which ensures they are placed in a specific zone, and can help to reduce latency in communicating between components in the same workload. Alternatively, some services can be deployed with *zone redundancy*, which means that Azure automatically replicates the resource across zones for high availability. Consider which approach makes provides the best set of tradeoffs for your solution.

For a structured approach to making your applications self healing, see [Design reliable applications for Azure](#).

Make all things redundant

Article • 09/22/2023

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

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

Recommendations

Consider business requirements. The amount of redundancy built into a system can affect both cost and complexity. Your architecture should be informed by your business requirements, such as recovery time objective (RTO) and recovery point objective (RPO). You should also consider your performance requirements, and your team's ability to manage complex sets of resources.

Consider multi-zone and multi-region architectures. Ensure that you understand how [availability zones and regions](#) provide resiliency and different sets of architectural tradeoffs.

Azure availability zones are isolated sets of data centers within a region. By using availability zones, you can be resilient to failures of a single data center or an entire availability zone. You can use availability zones to make tradeoffs between cost, risk mitigation, performance, and recoverability. For example, when you use zone redundant services in your architecture, Azure provides automatic data replication and failover between geographically separated instances, which mitigates many different types of risks.

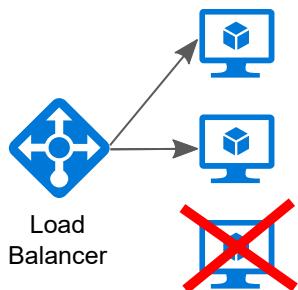
If you have a mission-critical workload and need to mitigate the risk of a region-wide outage, consider a multi-region deployment. While multi-region deployments insulate you against regional disasters, they come at a cost. Multi-region deployments are more expensive than a single-region deployment, and are more complicated to manage. You'll need operational procedures to handle failover and fallback. Depending on your RPO requirements, you might need to accept slightly lower performance to enable cross-region data replication. The additional cost and complexity might be justified for some business scenarios.

Tip

For many workloads, a zone-redundant architecture provides the best set of tradeoffs. Consider a multi-region architecture if your business requirements indicate that you need to mitigate the unlikely risk of a region-wide outage, and if you're prepared to accept the tradeoffs involved in such an approach.

To learn more about how to design your solution to use availability zones and regions, see [Recommendations for using availability zones and regions](#).

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](#).



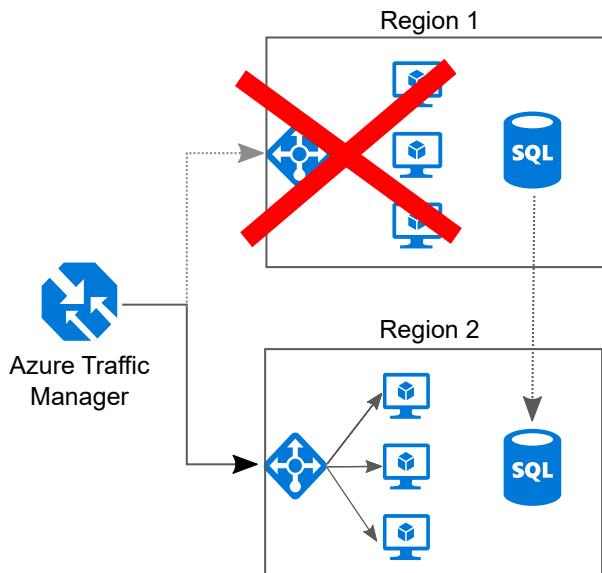
Replicate databases. Azure SQL Database and Azure Cosmos DB automatically replicate the data within a region, and can be configured to replicate across availability zones for higher resiliency. You can also choose to enable geo-replication across regions. Geo-replication for [Azure SQL Database](#) and [Azure Cosmos DB](#) creates secondary readable replicas of your data in one or more secondary regions. If an outage occurs in the primary region, the database can fail over to the secondary region for writes. Depending on the replication configuration you might experience some data loss from unreplicated transactions.

If you use an IaaS database solution, choose one that supports replication and failover, such as [SQL Server Always On availability groups](#).

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.

Multi-region solutions

The following diagram shows a multi-region application that uses Azure Traffic Manager to handle failover.



If you use Traffic Manager in a multi-region solution, consider the following recommendations:

Synchronize front and backend failover. Use 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 backend components and database solution, you may need to coordinate failing over your backend services and databases.

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.

To achieve this, disable the primary endpoint of Traffic Manager after failover. Note that if the monitoring interval of probes is short and the tolerated number of failures is small, failover as well as fallback will take place in a short time. In some cases, disabling won't be completed in time. To avoid unconfirmed fallback, consider also implementing a health endpoint that can verify that all subsystems are healthy. See the [Health Endpoint Monitoring pattern](#).

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.

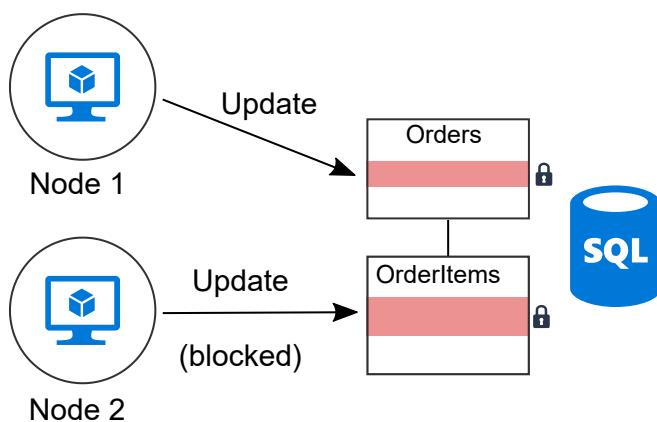
Minimize coordination

Azure Storage Azure SQL Database Azure Cosmos DB

Minimize coordination between application services to achieve scalability

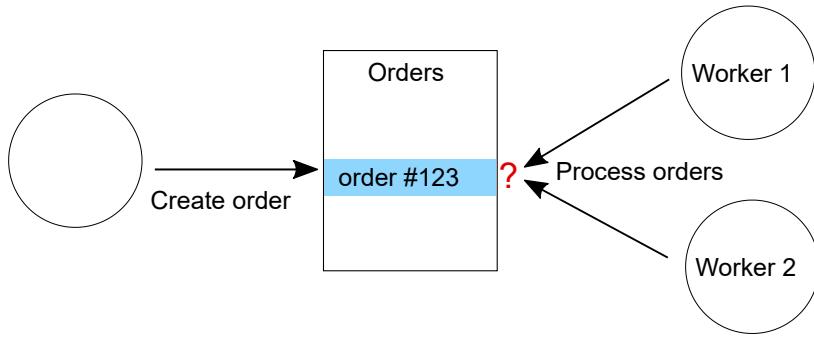
Most cloud applications consist of multiple application services — web front ends, databases, business processes, reporting and analysis, and so on. To achieve scalability and reliability, each of those services should run on multiple instances.

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



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

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



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

Recommendations

Embrace eventual consistency. When data is distributed, it takes coordination to enforce strong consistency guarantees. For example, suppose an operation updates two databases. Instead of putting it into a single transaction scope, it's better if the system can accommodate eventual consistency, perhaps by using the [Compensating Transaction](#) pattern to logically roll back after a failure.

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

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

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

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

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

Design idempotent operations. When possible, design operations to be idempotent. That way, they can be handled using at-least-once semantics. For example, you can put work items on a queue. If a worker crashes in the middle of an operation, another worker simply picks up the work item. If the worker needs to update data as well as emit other messages as a part of its logic, the [idempotent message processing pattern](#) should be used.

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

Article • 10/20/2023

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 naturally asynchronous tasks. Tasks like sending emails, actions where the user doesn't need an immediate response, and integration with other systems are all good places to make use of [asynchronous messaging patterns](#).

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.

Consider scaling for redundancy. Scaling out can improve your application's reliability. For example, consider scaling out across multiple [availability zones](#), such as by using zone-redundant services. This approach can improve your application's throughput as well as provide resiliency if one zone experiences an outage.

Related resources

- [Autoscaling](#)
- [Asynchronous messaging patterns](#)
- [Background jobs](#)
- [Pipes and Filters pattern](#)

Partition around limits

Article • 12/16/2022

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

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

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

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

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

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

For more detailed guidance, see [Data partitioning](#).

Recommendations

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

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

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

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

Design for operations

Article • 12/16/2022

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

The cloud has dramatically changed the role of the operations team. They are no longer responsible for managing the hardware and infrastructure that hosts the application.

That said, operations is still a critical part of running a successful cloud application.

Some of the important functions of the operations team include:

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

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

Recommendations

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

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

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

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

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

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

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

Use platform as a service (PaaS) options

Article • 10/10/2023

Infrastructure as a service (IaaS) and platform as a service (PaaS) are cloud service models.

IaaS offers access to computing resources like servers, storage, and networks. The IaaS provider hosts and manages this infrastructure. Customers use the internet to access the hardware and resources.

In contrast, PaaS provides a framework for developing and running apps. As with IaaS, the PaaS provider hosts and maintains the platform's servers, networks, storage, and other computing resources. But PaaS also includes tools, services, and systems that support the web application lifecycle. Developers use the platform to build apps without having to manage backups, security solutions, upgrades, and other administrative tasks.

Advantages of PaaS over IaaS

When it's possible, use PaaS instead of IaaS. IaaS is like having a box of parts. You can build anything, but you have to assemble it yourself. PaaS options are easier to configure and administer. You don't need to set up virtual machines (VMs) or virtual networks. You also don't have to handle maintenance tasks, such as installing patches and updates.

For example, suppose your application needs a message queue. You can set up your own messaging service on a VM by using something like RabbitMQ. But Azure Service Bus provides a reliable messaging service, and it's simpler to set up. You can create a Service Bus namespace as part of a deployment script. Then you can use a client SDK to call Service Bus.

PaaS alternatives to IaaS solutions

Your application might have specific requirements that make IaaS a more suitable approach than PaaS. But you can still look for places to incorporate PaaS options. A few examples include caches, queues, and data storage. The following table provides other examples.

Instead of running ...	Consider using ...
Active Directory	Microsoft Entra ID

Instead of running ...	Consider using ...
Elasticsearch	Azure Cognitive Search
Hadoop	Azure HDInsight
IIS	Azure App Service
MongoDB	Azure Cosmos DB
Redis	Azure Cache for Redis
SQL Server	Azure SQL Database
File share	Azure NetApp Files

This list isn't exhaustive. There are many ways that you can exchange IaaS technologies for related PaaS solutions.

Use a fully managed identity service platform

Article • 10/10/2023

Almost every cloud application needs to work with user identities. Identity is the foundation of modern security practices like [zero trust](#), and user identity for applications is a critical part of your solution's architecture.

For most solutions, we strongly recommend using an identity as a service (IDaaS) platform, a fully managed identity solution, instead of building or operating your own. In this article, we describe the challenges of building or running your own identity system.

Recommendations

Important

By using an IDaaS, like Microsoft Entra ID, Azure AD B2C, or another similar system, you can mitigate many of the issues that are described in this article. We recommend this approach wherever possible.

Your solution requirements might lead you to use a framework or off-the-shelf identity solution that you host and run yourself. While using a prebuilt identity platform mitigates some of the issues that are described in this article, handling many of these issues is still your responsibility with such a solution.

You should avoid using an identity system that you build from scratch.

Avoid storing credentials

When you run your own identity system, you have to store a database of credentials. You should never store credentials in clear text, or even as encrypted data.

Instead, you might consider cryptographically hashing and salting credentials before storing them, which makes them more difficult to attack. However, even hashed and salted credentials are vulnerable to several types of attack.

Regardless of how you protect the individual credentials, maintaining a database of credentials makes you a target for attacks. Recent years have shown that both large and small organizations have had their credentials databases targeted for attack.

Consider credential storage to be a liability, not an asset. By using an IDaaS, you outsource the problem of credential storage to experts who can invest the time and resources in securely managing credentials.

Implement identity and federation protocols

Modern identity protocols are complex. Industry experts have designed OAuth 2, OpenID Connect, and other protocols to ensure that they mitigate real-world attacks and vulnerabilities. The protocols also evolve to adapt to changes in technologies, attack strategies, and user expectations. Identity specialists, with expertise in the protocols and how they're used, are in the best position to implement and validate systems that follow these protocols. For more information about the protocols and the platform, see [OAuth 2.0 and OpenID Connect \(OIDC\) in the Microsoft identity platform](#).

It's also common to federate identity systems. Identity federation protocols are complex to establish, manage, and maintain, and they require specialist knowledge and experience. For more information, see [Federated identity pattern](#).

Adopt modern identity features

Users expect an identity system to have a range of advanced features, including:

- Passwordless authentication, which uses secure approaches to sign in that don't require users to enter credentials.
- Single sign-on (SSO), which allows users to sign in by using an identity from their employer, school, or another organization.
- Multifactor authentication (MFA), which prompts users to authenticate themselves in multiple ways. For example, a user might sign in by using a password and also by using an authenticator app on a mobile device or a code that's sent by email.
- Auditing, which tracks every event that happens in the identity platform, including successful, failed, and aborted sign-in attempts. To forensically analyze a sign-in attempt later might require a detailed log.
- Conditional access, which creates a risk profile around a sign-in attempt that's based on various factors. The factors might include the user's identity, the location of the sign-in attempt, previous sign-in activity, and the sensitivity of the data or application.
- Just-in-time access control, which temporarily allows users to sign in based on an approval process, and then removes the authorization automatically.

If you're building an identity component yourself as part of your business solution, it's unlikely you'll be able to justify the work involved in implementing these features—and in maintaining them. Some of these features also require extra work, such as integration with messaging providers to send MFA codes, and storing and retaining audit logs for a sufficient time period.

IDaaS platforms can also provide an improved set of security features that are based on the volume of sign-in requests that they receive. For example, the following features work best when there's a large number of customers who use a single identity platform:

- Detection of risky sign-in events, such as sign-in attempts from botnets
- Detection of [impossible travel](#) between a user's activities
- Detection of common credentials, such as passwords that are frequently used by other users, which are therefore subject to a heightened risk of compromise
- Use of machine learning techniques to classify sign-in attempts as valid or invalid
- Monitoring of the so-called *dark web* for leaked credentials and preventing their exploitation
- Ongoing monitoring of the threat landscape and the current vectors that attackers use

If you build or run your own identity system, you can't take advantage of these features.

Use a reliable, high-performance identity system

Because identity systems are such a key part of modern cloud applications, they must be reliable. If your identity system is unavailable, then the rest of your solution might be affected and either operate in a degraded fashion or fail to operate at all. By using an IDaaS with a service level agreement, you can increase your confidence that your identity system will remain operational when you need it. For example, Microsoft Entra ID offers an SLA for uptime for the Basic and Premium service tiers, which covers both the sign-in and token issuing processes. For more information, see [SLA for Microsoft Entra ID](#).

Similarly, an identity system must perform well and be able to scale to the level of growth that your system might experience. Depending on your application architecture, it's possible that every request might require interaction with your identity system, and any performance issues will be apparent to your users. IDaaS systems are incentivized to scale to large user loads. They're designed to absorb large volumes of traffic, including traffic generated by different forms of attacks.

Test your security and apply tight controls

If you run an identity system, it's your responsibility to keep it secured. Examples of the controls you need to consider implementing include:

- Periodic penetration testing, which requires specialized expertise.
- Vetting of employees and anybody else with access to the system.
- Tight control of all changes to your solution with all changes reviewed by experts.

These controls are often expensive and difficult to implement.

Use cloud-native security controls

When you use Microsoft Entra ID as your solution's identity provider, you can take advantage of cloud-native security features like [managed identities for Azure resources](#).

If you choose to use a separate identity platform, you need to consider how your application can take advantage of managed identities and other Microsoft Entra features while simultaneously integrating with your own identity platform.

Focus on your core value

It's expensive and complex to maintain a secure, reliable, and responsive identity platform. In most situations, an identity system isn't a component that adds value to your solution, or that differentiates you from your competitors. It's good to outsource your identity requirements to a system that's built by experts. That way, you can focus on designing and building the components of your solution that add business value for your customers.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [John Downs](#) | Senior Customer Engineer, FastTrack for Azure

Other contributors:

- [Jelle Druyts](#) | Principal Customer Engineer, FastTrack for Azure
- [LaBrina Loving](#) | Principal Customer Engineering Manager, FastTrack for Azure
- [Gary Moore](#) | Programmer/Writer
- [Arsen Vladimirsksiy](#) | Principal Customer Engineer, FastTrack for Azure

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [What is Microsoft Entra ID?](#)
- [What is Azure Active Directory B2C?](#)
- [Explore identity and Microsoft Entra ID](#)
- [Design an identity security strategy](#)
- [Implement Microsoft identity](#)
- [Manage identity and access in Microsoft Entra ID](#)

Related resources

- [Authenticate using Microsoft Entra ID and OpenID Connect](#)
- [Federated identity pattern](#)
- [Hybrid identity](#)
- [Identity architecture design](#)
- [Resilient identity and access management with Microsoft Entra ID](#)

Design for evolution

Article • 03/22/2023

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, where all the related functions reside in one service. 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 business needs

Article • 07/26/2023

Every design decision must be justified by a business requirement. This design principle might seem obvious, but is crucial to keep in mind when designing Azure applications.

Must your application support millions of users, or a few thousand? Are there large traffic bursts, or a steady workload? What level of application outage is acceptable? Ultimately, business requirements drive these design considerations.

The following recommendations help you design and build solutions to meet business requirements:

- **Define business objectives** such as recovery time objective (RTO), recovery point objective (RPO), and maximum tolerable outage (MTO). These numbers should inform decisions about the architecture.

For example, suppose your business requires a very low RTO and a very low RPO. You might choose to use a zone-redundant architecture to meet these requirements. If your business can tolerate a higher RTO and RPO, adding redundancy might add extra cost for no business benefit.

- **Consider the failure risks you need to mitigate.** Follow the [Design for self-healing guidance](#) to design your solution to be resilient to many types of common failure modes. Consider whether you need to account for less likely situations, like a geographic area experiencing a major natural disaster that might affect all of the availability zones in the region. Mitigating these uncommon risks is generally more expensive and involves significant tradeoffs, so have a clear understanding of the business's tolerance for risk.
- **Document service level agreements (SLAs)** and service level objectives (SLOs), including availability and performance metrics. For example, a proposed solution might deliver 99.95% availability. Whether that SLO meets the SLA is a business decision.
- **Model applications** for your business domain. Analyze the business requirements, and use these requirements to model the solution. Consider using a domain-driven design (DDD) approach to create domain models that reflect your business processes and use cases.
- **Define functional and nonfunctional requirements.** Functional requirements determine whether an application performs its task. Nonfunctional requirements determine how well the application performs. Make sure you understand

nonfunctional requirements like scalability, availability, and latency. These requirements influence design decisions and technology choices.

- **Decompose workloads.** Workload in this context means a discrete capability or computing task that can logically be separated from other tasks. Different workloads might have different requirements for availability, scalability, data consistency, and disaster recovery.
- **Plan for growth.** A solution might support current needs for number of users, transaction volume, and data storage, but it also needs to handle growth without major architectural changes. Also consider that your business model and business requirements might change over time. It's hard to evolve a solution for new use cases and scenarios if the application's service model and data models are too rigid.
- **Manage costs.** In a traditional on-premises application, you pay up front for hardware as a capital expenditure. In a cloud application, you pay for the resources you consume. Make sure that you understand your services' pricing model. Total costs might include network bandwidth usage, storage, IP addresses, and service consumption.

Also consider operations costs. In the cloud, you don't have to manage hardware or infrastructure, but you still need to manage application DevOps, incident response, and disaster recovery.

Next steps

- [Domain Model ↗](#)
- [Azure pricing ↗](#)

Related resources

- [Design to scale out](#)
- [Partition around limits](#)
- [Design for evolution](#)

Resiliency checklist for specific Azure services

Article • 07/26/2023

Resiliency is the ability of a system to recover from failures and continue to function. 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. For more information about designing resilient applications, see [Design reliable Azure applications](#).

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.

Deploy zone-redundant App Service plans. In supported regions, App Service plans can be deployed as zone redundant, which means that the instances are automatically distributed across availability zones. App Service automatically distributes traffic between the zones, and handles failover if a zone experiences an outage. For more information, see [Migrate App Service to availability zone support](#).

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.

Azure Load Balancer

Select Standard SKU. Standard Load Balancer provides a dimension of reliability that Basic does not - that of availability zones and zone resiliency. This means when a zone goes down, your zone-redundant Standard Load Balancer will not be impacted. This

ensures your deployments can withstand zone failures within a region. In addition, Standard Load Balancer supports global load balancing ensuring your application is not impacted by region failures either.

Provision at least two instances. Deploy Azure LB with at least two instances in the backend. A single instance could result in a single point of failure. In order to build for scale, you might want to pair LB with Virtual Machine Scale Sets.

Use outbound rules. Outbound rules ensure that you are not faced with connection failures as a result of SNAT port exhaustion. [Learn more about outbound connectivity](#). While outbound rules will help improve the solution for small to mid size deployments, for production workloads, we recommend coupling Standard Load Balancer or any subnet deployment with [VNet NAT](#).

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.

Azure Cosmos DB

Configure zone redundancy. When you use zone redundancy, Azure Cosmos DB synchronously replicates all writes across availability zones. It automatically fails over in the event of a zone outage. For more information, see [Achieve high availability with Azure Cosmos DB](#).

Replicate the database across regions. Azure Cosmos DB allows you to associate any number of Azure regions with an Azure Cosmos DB database account. An Azure 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 nontransient 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.

Configure zone redundancy. When zone redundancy is enabled on your namespace, Event Hubs automatically replicates changes between multiple availability zones. If one availability zone fails, failover happens automatically. For more information, see [Availability zones](#).

Implement disaster recovery by failing over to a secondary Event Hubs namespace. For more information, see [Azure Event Hubs Geo-disaster recovery](#).

Azure Cache for Redis

Configure zone redundancy. When zone redundancy is enabled on your cache, Azure Cache for Redis spreads the virtual machines that host your cache across multiple availability zones. Zone redundancy provides high availability and fault tolerance in the event of a data center outage. For more information, see [Enable zone redundancy for Azure Cache for Redis](#).

Configure Geo-replication. Geo-replication provides a mechanism for linking two Premium-tier Azure Cache for Redis 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 Cache for Redis](#)

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-tier Azure Cache for Redis](#)

If you are using Azure Cache for Redis as a temporary data cache and not as a persistent store, these recommendations may not apply.

Cognitive Search

Provision more than one replica. Use at least two replicas for read high-availability, or three for read-write high-availability.

Use zone redundancy. You can deploy Cognitive Search replicas across multiple availability zones. This approach helps your service to remain operational even when data center outages occur. For more information, see [Reliability in Azure Cognitive Search](#)

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 Cognitive 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 Cognitive 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 Cognitive 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 Cognitive 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 zone redundancy. When zone redundancy is enabled on your namespace, Service Bus automatically replicates changes between multiple availability zones. If one availability zone fails, failover happens automatically. For more information, see [Best practices for insulating applications against Service Bus outages and disasters](#).

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

Use zone-redundant storage. Zone-redundant storage (ZRS) copies your data synchronously across three Azure availability zones in the primary region. If an availability zone experiences an outage, Azure Storage automatically fails over to an alternative zone. For more information, see [Azure Storage redundancy](#).

When using geo-redundancy, configure read-access. If you use a multi-region architecture, use a suitable storage tier for global redundancy. With RA-GRS or RA-GZRS, your data is replicated to a secondary region. RA-GRS uses locally redundant storage (LRS) in the primary region, while RA-GZRS uses zone-redundant storage (ZRS)

in the primary region. Both configurations provide read-only access to your data in the secondary region. If there is a storage outage in the primary region, the application can read the data from the secondary region if you have designed it for this possibility. For more information, see [Azure Storage redundancy](#).

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 an Azure Storage outage, the application can use the backup queue, until the primary region becomes available again. That way, the application can continue to process new requests during the outage.

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](#).

Azure Synapse Analytics

Do not disable geo-backup. By default, Synapse Analytics 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 datacenters. 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 [virtual machine 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.

Replicate VMs using Azure Site Recovery. When you replicate Azure VMs using [Site Recovery](#), all the VM disks are continuously replicated to the target region asynchronously. The recovery points are created every few minutes. This gives you a Recovery Point Objective (RPO) in the order of minutes. You can conduct disaster recovery drills as many times as you want, without affecting the production application or the ongoing replication. For more information, see [Run a disaster recovery drill to Azure](#).

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. Include basic health metrics, infrastructure logs, and [boot diagnostics](#). Boot diagnostics can help you diagnose a boot failure if your VM gets into a nonbootable state. For more information, see [Overview of Azure Diagnostic Logs](#).

Configure Azure Monitor. Collect and analyze monitoring data from Azure virtual machines including the guest operating system and the workloads that run in it, see [Azure Monitor](#) and [Quickstart: Azure Monitor](#).

Virtual Network

To allow or block public IP addresses, add a network security group 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 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](#).

Failure mode analysis for Azure applications

Article • 07/26/2023

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 mitigation steps 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 mitigation steps. 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](#).

Microsoft Entra ID

OpenID Connect authentication fails.

Detection. Possible failure modes include:

1. Microsoft Entra ID is not available, or cannot be reached due to a network problem. Redirection to the authentication endpoint fails, and the OpenID Connect middleware throws an exception.
2. Microsoft Entra tenant does not exist. Redirection to the authentication endpoint returns an HTTP error code, and the OpenID Connect middleware throws an exception.
3. User cannot authenticate. No detection strategy is necessary; Microsoft Entra ID 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 nontransient 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 nontransient 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 nontransient 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).

Azure 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 Azure 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.
- Enable zone redundancy when you work with a region that supports availability zones. When you use zone redundancy, Azure Cosmos DB automatically fails over in the event of a zone outage. For more information, see [Achieve high availability with Azure Cosmos DB](#).
- If you're designing a multi-region solution, replicate the Azure 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 set up global distribution in Azure Cosmos DB for NoSQL](#).

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 Azure 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.
- Enable zone redundancy when you work with a region that supports availability zones. When you use zone redundancy, Azure Cosmos DB synchronously replicates all writes across availability zones. For more information, see [Achieve high availability with Azure Cosmos DB](#).
- If you're designing a multi-region solution, replicate the Azure 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 set up global distribution in Azure Cosmos DB for NoSQL](#).
- As a fallback, persist the document to a backup queue, and process the queue later.

Diagnostics. Log all errors on the client side.

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.

Azure Cache for Redis

Reading from the cache fails.

Detection. Catch `StackExchange.Redis.RedisConnectionException`.

Recovery:

1. Retry on transient failures. Azure Cache for Redis supports built-in retry. For more information, see [Azure Cache for Redis retry guidelines](#).
2. Treat nontransient failures as a cache miss, and fall back to the original data source.

Diagnostics. Use [Azure Cache for Redis diagnostics](#).

Writing to the cache fails.

Detection. Catch `StackExchange.Redis.RedisConnectionException`.

Recovery:

1. Retry on transient failures. Azure Cache for Redis supports built-in retry. For more information, see [Azure Cache for Redis retry guidelines](#).

2. If the error is nontransient, ignore it and let other transactions write to the cache later.

Diagnostics. Use [Azure Cache for Redis diagnostics](#).

SQL Database

Cannot connect to the database in the primary region.

Detection. Connection fails.

Recovery:

- **Enable zone redundancy.** By enabling zone redundancy, Azure SQL Database automatically replicates your writes across multiple Azure availability zones within supported regions. For more information, see [Zone-redundant availability](#).
- **Enable geo-replication.** If you're designing a multi-region solution, consider enabling SQL Database active geo-replication.

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:

- 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](#).
- 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.
- 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.
- Use zone redundancy to automatically replicate changes between multiple availability zones. If one availability zone fails, failover happens automatically. For more information, see [Best practices for insulating applications against Service Bus outages and disasters](#).
 - 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.
- If you're designing a multi-region solution, 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 can't 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.
2. 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](#).
3. 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 nontransient 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.



A sample is available [here](#). It uses [Polly](#) for these exceptions:

- 429 - Throttling
- 408 - Timeout
- 401 - Unauthorized
- 503 or 5xx - Service unavailable

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

See [Resiliency and dependencies](#) in the Azure Well-Architected Framework. Building failure recovery into the system should be part of the architecture and design phases from the beginning to avoid the risk of failure.

Technology choices for Azure solutions

Article • 12/16/2022

This article provides a list of resources that you can use to make informed decisions about the technologies that you choose for your Azure solutions. Explore comparison matrices, flowcharts, and decision trees to ensure that you find the best matches for your scenario.

Choose a compute service

The term *compute* refers to the hosting model for the computing resources that your application runs on. The following articles can help you choose the right technologies:

Article	Summary
Choose an Azure compute service	Decide which compute service best suits your application.
High availability and disaster recovery scenarios for IaaS apps	Learn about high availability (HA) and disaster recovery (DR) options for multitier infrastructure as a service (IaaS) apps in Azure.
Choose an Azure compute option for microservices	Learn about two compute options for microservices: service orchestrator and serverless architecture.
Choose between traditional web apps and SPAs	Learn how to choose between traditional web apps and single-page applications (SPAs).
Choose an Azure multiparty computing service	Decide which multiparty computing services to use for your solution.

Choose a container option

There are many ways to build and deploy cloud-native and containerized applications in Azure. Review these articles to learn more:

Article	Summary
Compare Container Apps with other Azure container options	Understand when to use Azure Container Apps and how it compares to other container options, including Azure Container Instances, Azure App Service, Azure Functions, and Azure Kubernetes Service (AKS).

Article	Summary
Choose a Kubernetes at the edge compute option	Learn about the pros and cons of various options for extending compute at the edge.
Choose a bare-metal Kubernetes at the edge platform option	Find the best option, given a specific use case, for configuring Kubernetes clusters at the edge.

Choose a hybrid option

Many organizations need a hybrid approach to analytics, automation, and services because their data is hosted both on-premises and in the cloud. The following articles can help you choose the best technologies for your scenario:

Article	Summary
Azure hybrid options	Learn about Azure hybrid solutions, including alternatives to deploy and host hybrid services on-premises, at the edge, in Azure, and in other clouds.
Compare Azure Stack Hub to Azure	Learn the differences between Azure and Azure Stack Hub.
Compare Azure, Azure Stack Hub, and Azure Stack HCI	Learn the differences between Azure, Azure Stack Hub, and Azure Stack HCI.
Compare Azure Stack HCI to Azure Stack Hub	Determine whether Azure Stack HCI or Azure Stack Hub is right for your organization.
Compare Azure Stack HCI to Windows Server	Determine whether Azure Stack HCI or Windows Server is right for your organization.
Choose drives for Azure Stack HCI and Windows Server clusters	Learn how to choose drives for Azure Stack HCI and Windows Server clusters to meet performance and capacity requirements.

Choose an identity service

Identity solutions help you protect your data and resources. These articles can help you choose an Azure identity service:

Article	Summary
Active Directory services	Compare the identity services that are provided by Active Directory Domain Services, Azure Active Directory (Azure AD), and Azure Active Directory Domain Services.
Hybrid identity authentication methods	Choose an authentication method for an Azure AD hybrid identity solution in a medium-sized to large organization.

Choose a storage service

The Azure Storage platform is the Microsoft cloud storage solution for modern data storage scenarios. Review these articles to determine the best solution for your use case:

Article	Summary
Review your storage options	Review the storage options for Azure workloads.
Azure managed disk types	Learn about the disk types that are available for Azure virtual machines, including Ultra disks, Premium SSDs v2 (preview), Premium SSDs, Standard SSDs, and Standard HDDs.
Choose an Azure solution for data transfer	Choose an Azure solution for data transfer, based on the amount of data and the available network bandwidth in your environment.

Choose a data store

The cloud is changing the way applications are designed, including how data is processed and stored. These articles can help you choose a data solution:

Article	Summary
Understand data store models	Learn about the high-level differences between the various data storage models in Azure data services.
Choose an Azure data store for your application	Use a flowchart to choose an Azure data store.
Criteria for choosing a data store	Review some general considerations for choosing a data store.

Article	Summary
Choose a big data storage technology in Azure	Compare big data storage options in Azure. View key selection criteria and a capability matrix.
OLAP solutions	Learn about online analytical processing (OLAP) solutions for organizing large databases and supporting complex analysis without affecting transactional systems.
OLTP solutions	Learn about atomicity, consistency, and other features of online transaction processing (OLTP), which manages transactional data and supports querying.
Data warehousing	Learn about data warehousing in Azure. A data warehouse is a repository of integrated data from disparate sources that's used for reporting and analysis of the data.
Data lakes	Learn about data lake storage repositories, which can hold terabytes or petabytes of data in a native, raw format.
Non-relational data and NoSQL	Learn about non-relational databases that store data as key/value pairs, graphs, time series, objects, and other storage models.
Choose a data pipeline orchestration technology	Choose an Azure data pipeline orchestration technology to automate pipeline orchestration, control flow, and data movement workflows.
Choose a search data store	Learn about the capabilities of search data stores in Azure and the key criteria for choosing one that best matches your needs.
Transfer data to and from Azure	Learn about Azure data transfer options like Azure Import/Export, Azure Data Box, Azure Data Factory, and command-line and graphical interface tools.

Choose an analytics solution

With the exponential growth in data, organizations rely on the limitless compute, storage, and analytical power of Azure. Review these articles to learn about the available analytics solutions:

Article	Summary
Choose an analytical data store	Evaluate analytical data store options for big data in Azure.

Article	Summary
Choose a data analytics and reporting technology	Evaluate big data analytics technology options for Azure.
Choose a batch processing technology	Compare technology choices for big data batch processing in Azure.
Choose a stream processing technology	Compare options for real-time message stream processing in Azure.

Choose an AI / machine learning service

AI is the capability of a computer to imitate intelligent human behavior. Through AI, machines can analyze images, comprehend speech, interact in natural ways, and make predictions based on data. Review these articles to learn about the AI and machine learning technology choices that are available in Azure:

Article	Summary
Choose an Azure Cognitive Services technology	Learn about cognitive services that you can use in AI applications and data flows.
Natural language processing technology	Choose a natural language processing service for sentiment analysis, topic and language detection, key phrase extraction, and document categorization.
Compare machine learning products and technologies	Compare options for building, deploying, and managing your machine learning models. Decide which products to use for your solution.
Azure Machine Learning guide for tool selection	Choose the best services for building an end-to-end machine learning pipeline, from experimentation to deployment.
MLflow and Azure Machine Learning	Learn about how Azure Machine Learning uses MLflow to log metrics and artifacts from machine learning models and deploy your machine learning models to an endpoint.

Choose a networking service

These articles can help you explore the networking technologies that are available in Azure:

Article	Summary
Load balancing options	Learn about Azure load balancing services and how you can use them to distribute your workloads across multiple computing resources.
Choose between virtual network peering and VPN gateways	Review the differences between virtual network peering and VPN gateways, which are two ways to connect virtual networks in Azure.

Choose a messaging service

Learn about the services that Azure provides to help you deliver events or messages throughout your solution:

Article	Summary
Compare messaging services	Learn about the three Azure messaging services: Azure Event Grid, Azure Event Hubs, and Azure Service Bus. Choose the best service for your scenario.
Asynchronous messaging options	Learn about asynchronous messaging options in Azure, including the various types of messages and the entities that participate in a messaging infrastructure.
Choose a real-time message ingestion technology	Choose an Azure message ingestion store to support message buffering, scale-out processing, reliable delivery, and queuing semantics.

Choose an IoT option

IoT solutions use a combination of technologies to connect devices, events, and actions through cloud applications. Review these articles to learn more about the IoT technology choices that Azure provides:

Article	Summary
Choose an IoT solution	Use Azure IoT Central or individual Azure platform as a service (PaaS) components to build, deploy, and manage IoT solutions.
Compare IoT Hub and Event Hubs	Review a comparison between Azure IoT Hub and Event Hubs that highlights functional differences and use cases. The comparison includes supported protocols, device management, monitoring, and file uploads.

Choose a mobile development framework

Article	Summary
Choose a mobile development framework	Learn about the supported native and cross-platform languages for building client applications.

Choose a mixed reality engine

Article	Summary
Choose a mixed reality engine	Learn about the engine choices for mixed reality development for HoloLens and virtual reality.

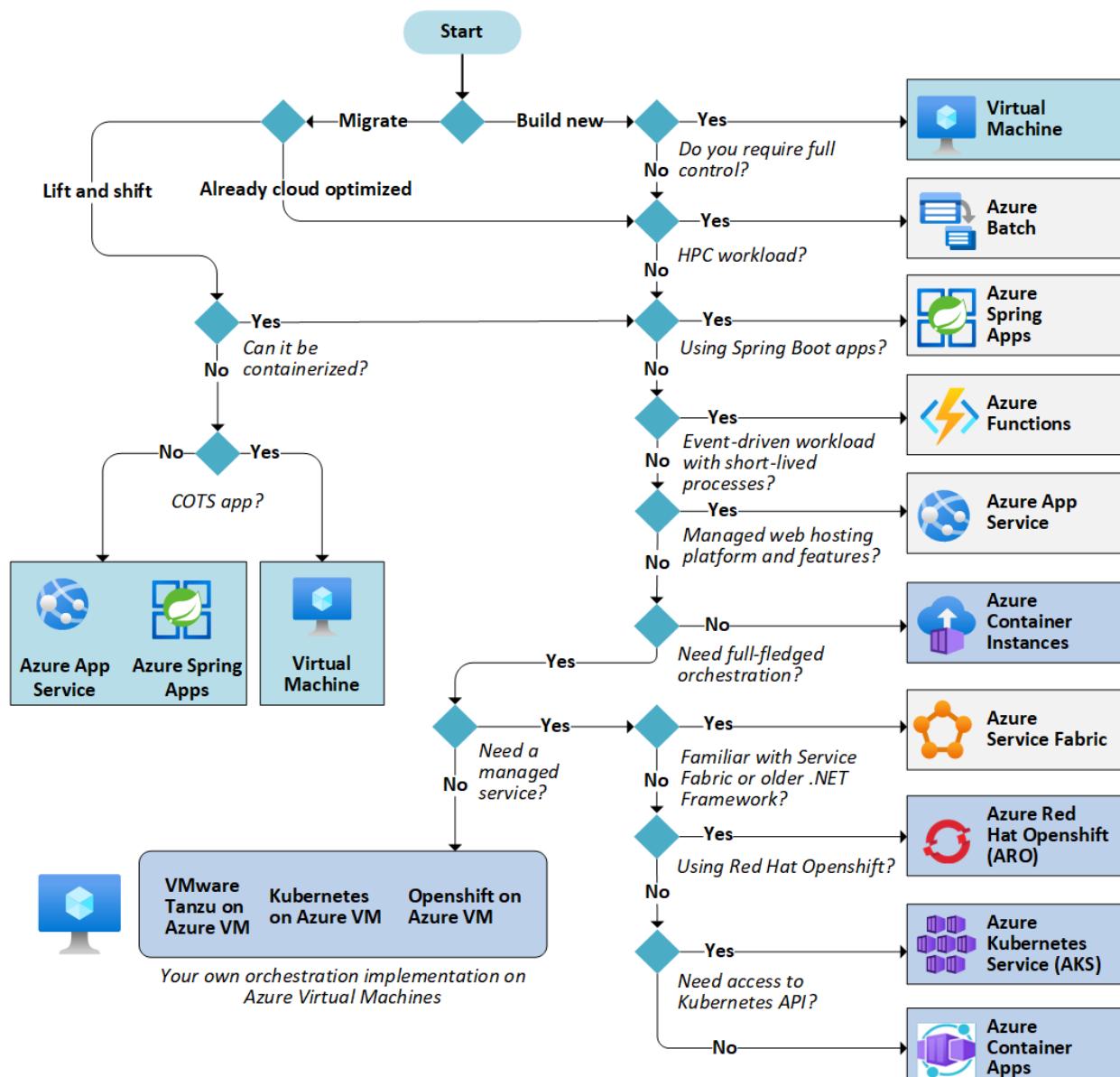
Choose an Azure compute service

Azure App Service Azure Kubernetes Service (AKS)

Azure offers many ways to host your application code. The term *compute* refers to the hosting model for the resources that your application runs on. This article helps choose a compute service for your application.

Choose a candidate service

Use the following flowchart to select a candidate compute service.



Container exclusive services		
- Azure Batch	- Azure Spring Apps	- VMware Tanzu on Azure VM
- Azure Functions	- Azure Service Fabric	- OpenShift on Azure VM
- Azure App Service	- Kubernetes on Azure VM	

- Container compatible services
 - Azure Batch
 - Azure Functions
 - Azure Service Fabric
 - Azure Spring Apps
 - Azure App Service

This diagram refers to two migration strategies:

- **Lift and shift:** A strategy for migrating a workload to the cloud without redesigning the application or making code changes. It's also called *rehosting*. For more information, see [Azure migration and modernization center](#).
- **Cloud optimized:** A strategy for migrating to the cloud by refactoring an application to take advantage of cloud-native features and capabilities.

The output from this flowchart is your starting point. Next, evaluate the service to see if it meets your needs.

This article includes several tables that can help you choose a service. The initial candidate from the flowchart might be unsuitable for your application or workload. In that case, expand your analysis to include other compute services.

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

Understand the basic features

If you're not familiar with the Azure service selected in the previous section, see this overview documentation:

- [Azure Virtual Machines](#): A service where you deploy and manage virtual machines (VMs) inside an Azure virtual network.
- [Azure App Service](#): A managed service for hosting web apps, mobile app back ends, RESTful APIs, or automated business processes.
- [Azure Functions](#): A managed function as a service.
- [Azure Kubernetes Service \(AKS\)](#): A managed Kubernetes service for running containerized applications.
- [Azure Container Apps](#): A managed service built on Kubernetes, which simplifies the deployment of containerized applications in a serverless environment.
- [Azure Container Instances](#): This service is a fast and simple way to run a container in Azure. You don't have to provision any VMs or adopt a higher-level service.
- [Azure Red Hat OpenShift](#): A fully managed OpenShift cluster for running containers in production with Kubernetes.
- [Azure Spring Apps](#): A managed service designed and optimized for hosting Spring Boot apps.
- [Azure Service Fabric](#): A distributed systems platform that can run in many environments, including Azure or on-premises.
- [Azure Batch](#): A managed service for running large-scale parallel and high-performance computing (HPC) applications.

Understand the hosting models

For hosting models, cloud services fall into three categories:

- **Infrastructure as a service (IaaS)**: Lets you provision VMs along with the associated networking and storage components. Then you can deploy whatever software and applications you want onto those VMs. This model is the closest to a traditional on-premises environment. Microsoft manages the infrastructure. You still manage the VMs.
- **Platform as a service (PaaS)**: Provides a managed hosting environment where you can deploy your application without needing to manage VMs or networking resources. Azure App Service and Azure Container Apps are PaaS services.
- **Functions as a service (FaaS)**: Lets you deploy your code to the service, which automatically runs it. Azure Functions is a FaaS service.

ⓘ Note

Azure Functions is an [Azure serverless](#) compute offering. To see how this service compares with other Azure serverless offerings, such as Logic Apps, which provides serverless workflows, see [Choose the right integration and automation services in Azure](#).

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

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

[+] [Expand table](#)

Service	Application composition	Density	Minimum number of nodes	State management	Web hosting
Azure Virtual	Agnostic	Agnostic	1 ²	Stateless or stateful	Agnostic

Service	Application composition	Density	Minimum number of nodes	State management	Web hosting
Machines					
Azure App Service	Applications, containers	Multiple apps per instance by using App Service plan	1	Stateless	Built in
Azure Functions	Functions, containers	Serverless ¹	Serverless ¹	Stateless or stateful ⁶	Not applicable
Azure Kubernetes Service	Containers	Multiple containers per node	3 ³	Stateless or stateful	Agnostic
Azure Container Apps	Containers	Serverless	Serverless	Stateless or stateful	Agnostic
Azure Container Instances	Containers	No dedicated instances	No dedicated nodes	Stateless	Agnostic
Azure Red Hat OpenShift	Containers	Multiple containers per node	6 ⁵	Stateless or stateful	Agnostic
Azure Spring Apps	Applications, microservices	Multiple apps per service instance	2	Stateless	Built in
Azure Service Fabric	Services, guest executables, containers	Multiple services per VM	5 ³	Stateless or stateful	Agnostic

Service	Application composition	Density	Minimum number of nodes	State management	Web hosting
Azure Batch	Scheduled jobs	Multiple apps per VM	1 ⁴	Stateless	No

Notes

1. If you're using a Consumption plan. For an 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 service-level agreement (SLA) with two or more instances.
3. Recommended for production environments.
4. Can scale down to zero after job completes.
5. Three for primary nodes and three for worker nodes.
6. When using [Durable Functions](#).

Networking

[\[+\] Expand table](#)

Service	Virtual network integration	Hybrid connectivity
Azure Virtual Machines	Supported	Supported
Azure App Service	Supported ¹	Supported ²
Azure Functions	Supported ¹	Supported ³
Azure Kubernetes Service	Supported	Supported
Azure Container Apps	Supported	Supported
Azure Container Instances	Supported	Supported
Azure Red Hat OpenShift	Supported	Supported

Service	Virtual network integration	Hybrid connectivity
Azure Spring Apps	Supported	Supported
Azure Service Fabric	Supported	Supported
Azure Batch	Supported	Supported

Notes

1. Requires App Service Environment.
2. Use [Azure App Service Hybrid Connections](#).
3. Requires App Service plan or [Azure Functions Premium plan](#).

DevOps

[] Expand table

Service	Local debugging	Programming model	Application update
Azure Virtual Machines	Agnostic	Agnostic	No built-in support
Azure App Service	IIS Express, others ¹	Web and API applications, WebJobs for background tasks	Deployment slots
Azure Functions	Visual Studio or Azure Functions CLI	Serverless, event-driven	Deployment slots
Azure Kubernetes Service	Minikube, Docker, others	Agnostic	Rolling update
Azure Container Apps	Local container runtime	Agnostic	Revision management

Service	Local debugging	Programming model	Application update
Azure Container Instances	Local container runtime	Agnostic	Not applicable
Azure Red Hat OpenShift	Minikube, Docker, others	Agnostic	Rolling update
Azure Spring Apps	Visual Studio Code, IntelliJ, Eclipse	Spring Boot, Steeltoe	Rolling upgrade, blue-green deployment
Azure Service Fabric	Local node cluster	Guest executable, Service model, Actor model, Containers	Rolling upgrade (per service)
Azure Batch	Not supported	Command-line application	Not applicable

Notes

- Options include IIS Express for ASP.NET or node.js (`iisnode`), PHP web server, Azure Toolkit for IntelliJ, and Azure Toolkit for Eclipse. App Service also supports remote debugging of deployed web app.

Scalability

[\[\]](#) Expand table

Service	Autoscaling	Load balancer	Scale limit³
Azure Virtual Machines	Virtual machine scale sets	Azure Load Balancer	Platform image: 1,000 nodes per scale set, Custom image: 600 nodes per scale set
Azure App Service	Built-in service	Integrated	30 instances, 100 with App Service Environment

Service	Autoscaling	Load balancer	Scale limit³
Azure Functions	Built-in service	Integrated	200 instances per function app
Azure Kubernetes Service	Pod autoscaling ¹ , cluster autoscaling ²	Azure Load Balancer or Azure Application Gateway	5,000 nodes when using Uptime SLA
Azure Container Apps	Scaling rules ⁴	Integrated	5 environments per region, 20 container apps per environment, 30 replicas per container app
Azure Container Instances	Not supported	No built-in support	20 container groups per subscription (default limit)
Azure Red Hat OpenShift	Pod autoscaling, cluster autoscaling	Azure Load Balancer or Azure Application Gateway	60 nodes per cluster (default limit)
Azure Spring Apps	Built-in service	Integrated	500 app instances in Standard
Azure Service Fabric	Virtual machine scale sets	Azure Load Balancer	100 nodes per virtual machine scale set
Azure Batch	Not applicable	Azure Load Balancer	20 core limit (default limit)

Notes

1. See [Autoscale pods](#).
2. See [Automatically scale a cluster to meet application demands on Azure Kubernetes Service](#).
3. See [Azure subscription and service limits, quotas, and constraints](#).
4. See [Set scaling rules in Azure Container Apps](#).

Availability

 [Expand table](#)

Service	SLA	Multiregion failover
Azure Virtual Machines	SLA for Virtual Machines	Azure Traffic Manager, Azure Front Door, and cross-region Azure Load Balancer
Azure App Service	SLA for App Service	Azure Traffic Manager and Azure Front Door
Azure Functions	SLA for Functions	Azure Traffic Manager and Azure Front Door
Azure Kubernetes Service	SLA for AKS	Azure Traffic Manager, Azure Front Door, and Multiregion Cluster
Azure Container Apps	SLA for Container Apps	Azure Traffic Manager and Azure Front Door
Azure Container Instances	SLA for Container Instances	Azure Traffic Manager and Azure Front Door
Azure Red Hat OpenShift	SLA for Azure Red Hat OpenShift	Azure Traffic Manager and Azure Front Door
Azure Spring Apps	SLA for Azure Spring Apps	Azure Traffic Manager, Azure Front Door, and Multiregion Cluster
Azure Service Fabric	SLA for Service Fabric	Azure Traffic Manager, Azure Front Door, and cross-region Azure Load Balancer
Azure Batch	SLA for Batch	Not applicable

For guided learning on service guarantees, see [Core Cloud Services - Azure architecture and service guarantees](#).

Security

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

- Azure Windows virtual machine
- Azure Linux virtual machine
- Azure App Service
- Azure Functions
- Azure Kubernetes Service
- Azure Container Instances
- Azure Spring Apps
- Azure Service Fabric
- Azure Batch

Other criteria

[] [Expand table](#)

Service	TLS	Cost	Suitable architecture styles
Azure Virtual Machines	Configured in VM	Windows ↗, Linux ↗	N-tier, big compute (HPC)
Azure App Service	Supported	App Service pricing ↗	Web-queue-worker
Azure Functions	Supported	Functions pricing ↗	Microservices, event-driven architecture
Azure Kubernetes Service	Ingress controller	AKS pricing ↗	Microservices, event-driven architecture
Azure Container Apps	Ingress controller	Container Apps pricing ↗	Microservices, event-driven architecture
Azure Container Instances	Use sidecar container	Container Instances pricing ↗	Microservices, task automation, batch jobs

Service	TLS	Cost	Suitable architecture styles
Azure Red Hat OpenShift	Supported	Azure Red Hat OpenShift pricing ↗	Microservices, event-driven architecture
Azure Spring Apps	Supported	Azure Spring Apps pricing ↗	Spring Boot, microservices
Azure Service Fabric	Supported	Service Fabric pricing ↗	Microservices, event-driven architecture
Azure Batch	Supported	Batch pricing ↗	Big compute (HPC)

Consider limits and cost

Along with the previous comparison tables, do a more detailed evaluation of the following aspects of the candidate service:

- [Service limits](#)
- [Cost ↗](#)
- [SLA ↗](#)
- [Regional availability ↗](#)

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors:

- [Ayobami Ayodeji ↗](#) | Senior Program Manager
- [Jelle Druyts ↗](#) | Principal Service Engineer
- [Martin Gjoshevski ↗](#) | Senior Service Engineer
- [Phil Huang ↗](#) | Senior Cloud Solution Architect
- [Julie Ng ↗](#) | Senior Service Engineer
- [Paolo Salvatori ↗](#) | Principal Service Engineer

To see nonpublic LinkedIn profiles, sign in to LinkedIn.

Next steps

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

Related resources

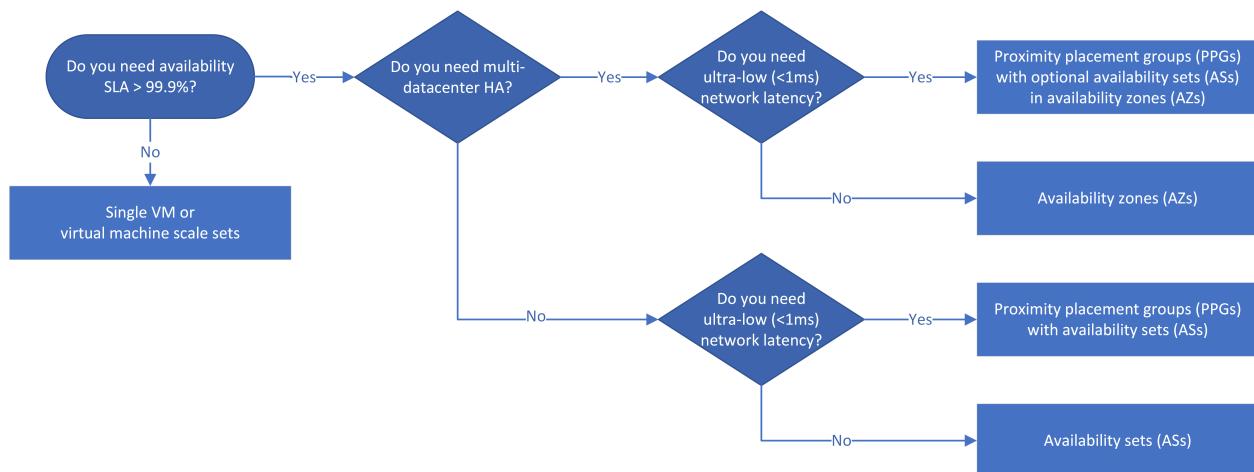
- Choose an Azure compute option for microservices
- Lift and shift to containers with Azure App Service
- Technology choices for Azure solutions

High availability and disaster recovery scenarios for IaaS apps

Azure Site Recovery Azure Virtual Machines Azure Disk Storage

This article presents a decision tree and examples of high-availability (HA) and disaster recovery (DR) options when deploying multitier infrastructure-as-a-service (IaaS) apps to Azure.

Architecture



Workflow

[Availability sets](#) (ASs) provide VM redundancy and availability within a datacenter by distributing VMs across multiple isolated hardware nodes. A subset of VMs keeps running during planned or unplanned downtime, so the entire app remains available and operational.

[Availability zones](#) (AZs) are unique physical locations that span datacenters within an Azure region. Each AZ accesses one or more datacenters that have independent power, cooling, and networking, and each AZ-enabled Azure region has a minimum of three separate AZs. The physical separation of AZs within a region protects deployed VMs from datacenter failure.

The decision flowchart reflects the principle that HA apps should use AZs if possible. Cross-zone, and therefore cross-datacenter, HA provides > 99.99% SLA because of resilience to datacenter failure.

ASs and AZs for different app tiers aren't guaranteed to be within the same datacenters. If app latency is a primary concern, you should colocate services in a single datacenter by using [proximity placement groups](#) (PPGs) with AZs and ASs.

Components

- [Azure Site Recovery](#)
- [Azure Virtual Machines](#)
- [Azure Disk Storage](#)

Alternatives

- As an alternative to regional DR using Azure Site Recovery, if the app can replicate data natively, you can implement *multi-region DR* using hot/cold standby servers, such as a stretched cluster for DR only. This alternative isn't specifically detailed in the examples, but could be added to any of the solutions. Note that replication between regions is asynchronous, and some data loss is expected.

Alternatively, if you have your own data replication technology, you can use it to create a secondary in-region zone for DR. Depending on the region of your workloads, it might also be possible to use Azure Site Recovery to replicate items to an alternative zone, you can check regional availability and read more about this feature at [Enable Zone to Zone Disaster Recovery for Azure virtual machines](#).

- Multi-region HA is possible, but requires a global load balancer such as Front Door or Traffic Manager. For more information, see [Run an N-tier application in multiple Azure regions for high availability](#).

Scenario details

Multitier or [n-tier](#) architectures are common in traditional on-premises apps, so they're a natural choice for migrating on-premises apps to the cloud, or when developing apps for both on-premises and the cloud. N-tier architectures are typically implemented as IaaS apps divided into logical layers and physical tiers, with a top web or presentation tier, a middle business tier, and a data tier.

In an IaaS n-tier app, each tier runs on a separate set of VMs. The web and business tiers are stateless, meaning any VM in the tier can handle any request for that tier. The data tier is a replicated database, object storage, or file storage. Multiple VMs in each tier provide resiliency if one VM fails, and load balancers distribute requests across the VMs.

You can scale out tiers by adding more VMs to the pools, and use [virtual machine scale sets](#) to automatically scale out identical VMs. Because you use load balancers, you can scale out tiers without affecting app uptime.

If the service-level agreement (SLA) for an IaaS app requires > 99% availability, you can place VMs in *availability sets*, *availability zones*, and *proximity placement groups* to configure high availability for the app. The HA and DR solutions you choose depend on the required SLA, latency considerations, and regional DR requirements.

Potential use cases

- Migrate an n-tier app from on-premises to the cloud.
- Deploy an n-tier app both on-premises and to the cloud.
- Configure high availability and disaster recovery for an IaaS app.

This solution can be used for any industry, including the following scenarios:

- Public sector applications
- Banking (finance industry)
- Healthcare

Considerations

- AZs aren't available in all [Azure regions](#).
- Decide which deployment option you want to use before you build the solution. Although possible, it's not easy to move from one option to another post-deployment. You would have to delete the VMs and recreate them from the underlying managed disks, which is an involved process.
- Make sure you can map your application against the selected solution. Many app layer resiliency patterns and designs are outside the scope of this decision tree.
- Three scenarios can lead to Azure VM reboots: unplanned hardware maintenance, unexpected downtime, and planned maintenance. For more information about these events and HA best practices to reduce their impact, see [Understand VM reboots, maintenance vs. downtime](#).

Single VMs

If an app doesn't require > 99.9% availability, you don't need to configure it for HA, and can deploy single VMs. You can use virtual machine scale sets to automatically scale out

identical VMs. Deploy single VMs without specifying a zone, so they're distributed throughout a region. These apps have an SLA of 99.9% if you use Azure Premium SSD disks.

Single VMs use the default service healing functionality built into all Azure datacenters. For predictable failures, this functionality typically uses live migration, but during unpredictable events, VMs might be rebooted or made unavailable.

High availability

If the app requires an SLA of > 99.9%, design the app for HA. Use AZs if possible, because they provide datacenter fault tolerance. You can use ASs instead of AZs, but using ASs reduces availability from 99.99% to 99.95%, because ASs can't tolerate datacenter failure.

AZs are suitable for many clustered app scenarios, including [AlwaysOn SQL clusters](#), using *active-active*, *active-passive*, or a combination of both HA levels at each tier with fast failover. Synchronous replication is possible between any Database Management System (DBMS) nodes, because of the low latency of the cross-zonal network. You can also run a *stretched-cluster* configuration across zones, which has higher latency and supports asynchronous replication.

If you want to use a VM-based *cluster arbiter*, for example a *file-share witness*, place it in the third AZ, to ensure quorum isn't lost if any one zone fails. Alternatively, you might be able to use a cloud-based witness in another region.

All VMs in an AZ are in a single *fault domain* (FD) and *update domain* (UD), meaning they share a common power source and network switch, and can all be rebooted at the same time. If you create VMs across different AZs, your VMs are effectively distributed across different FDs and UDs, so they won't all fail or be rebooted at the same time. If you want to have redundant in-zone VMs as well as cross-zone VMs, you should place the in-zone VMs in ASs in PPGs to ensure they won't all be rebooted at once. Even for single-instance VM workloads that aren't redundant today, you can still optionally use ASs in the PPGs to allow for future growth and flexibility.

For deploying virtual machine scale sets across AZs, consider using [Orchestration mode](#), currently in public preview, which allows combining FDs and AZs.

AZs with in-zone PPGs allow for one of the lowest network latencies in Azure, and an SLA of at least 99.99% because of multi-datacenter resiliency. Use [accelerated networking](#) on the VMs where possible.

This solution might present a scenario where a service running on a VM in one zone needs to interact with a service in another zone. For example, there might be an active-active web tier and an active-passive database tier across zones. Some requests will cross zones, which introduces latency. While cross-zone latency is still very low, if you need to ensure the lowest possible latency, keep all network communications between app tiers within a zone.

Latency considerations

Network latency depends, among other things, on the physical distance between deployed VMs. If an app requires very low latency between tiers, you can deploy it in a single datacenter, using a PPG with ASs for each tier. If possible, use accelerated networking on the VMs. This scenario allows for one of the lowest network latencies in Azure, and an SLA of 99.95%.

You can use the following tools to gain better insight into latency conditions for a variety of scenarios:

- To test the latency between VMs, see [Test VM network latency](#).
- To test latency between zones, use the [AvZone-Latency-Test](#). This test can help you determine which logical zones have the lowest latency for your subscription.
- To test latency between Azure regions, use <http://www.azurespeed.com/>. This regularly updated tool can be useful when considering asynchronous replication between regions.

Disaster recovery

DR considerations include *availability*, the ability of the app to keep running in a healthy state, and *data durability*, the preservation of data if a disaster happens.

HA failover should be fast, with no data loss, and have a very limited effect on service. In contrast, a traditional DR failover might have a longer associated *Recovery Time Objective (RTO)* and *Recovery Point Objective (RPO)*, and is asynchronous, with potential data loss.

You can take advantage of AZs for both HA and DR by using a different AZ for your DR solution. However, using a different AZ doesn't guarantee that the datacenters in each AZ will be located physically far apart.

[Azure Site Recovery](#) lets you replicate VMs to another Azure region for regional disaster recovery and business continuity. You can use Azure Site Recovery to recover your apps

in the event of source region outages, or to conduct periodic disaster recovery drills to ensure you meet compliance requirements.

If your app supports Azure Site Recovery, you can provide a regional DR solution for increased protection, if the criticality of the app demands it. However, cross-zone, cross-datacenter HA alone might be sufficient protection, because if an app is fully resilient to datacenter failure, there should be no downtime or data loss.

Cost optimization

There's no additional cost for VMs deployed in AZs. There might be additional inter-AZ VM-to-VM data transfer charges. For more information, see the [Bandwidth pricing page](#).

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Shaun Croucher](#) | Senior Consultant

Next steps

- [Availability sets](#)
- [Availability zones](#)
- [Virtual machine scale sets](#)
- [Enable Zone to Zone Disaster Recovery for Azure virtual machines](#)

Related resources

- [N-tier architecture style](#)
- [Multitier web application built for high availability and disaster recovery on Azure](#)
- [Run a zone-redundant web application for high availability](#)
- [Run a web application in multiple Azure regions for high availability](#)
- [Run an N-tier application in multiple Azure regions for high availability](#)

Choose an Azure compute option for microservices

Article • 06/30/2023

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."
- [Azure Container Apps](#) is a managed service built on Kubernetes that abstracts the complexities of container orchestration and other management tasks. Container Apps simplifies the deployment and management of containerized applications and microservices in a serverless environment while providing the features of Kubernetes.
- [Service Fabric](#) is a distributed systems platform for packaging, deploying, and managing microservices. Microservices can be deployed to Service Fabric as containers, as binary executables, or as [Reliable Services](#). Using the Reliable Services programming model, services can directly use Service Fabric programming

APIs to query the system, report health, receive notifications about configuration and code changes, and discover other services. A key differentiation with Service Fabric is its strong focus on building stateful services using [Reliable Collections](#).

- Other options such as Docker Enterprise Edition can run in an IaaS environment on Azure. You can find deployment templates on [Azure Marketplace](#) ↗.

Containers

Sometimes people talk about containers and microservices as if they were the same thing. While that's not true — you don't need containers to build microservices — containers do have some benefits that are particularly relevant to microservices, such as:

- **Portability.** A container image is a standalone package that runs without needing to install libraries or other dependencies. That makes them easy to deploy. Containers can be started and stopped quickly, so you can spin up new instances to handle more load or to recover from node failures.
- **Density.** Containers are lightweight compared with running a virtual machine, because they share OS resources. That makes it possible to pack multiple containers onto a single node, which is especially useful when the application consists of many small services.
- **Resource isolation.** You can limit the amount of memory and CPU that is available to a container, which can help to ensure that a runaway process doesn't exhaust the host resources. See the [Bulkhead pattern](#) for more information.

Serverless (Functions as a Service)

With a [serverless](#) ↗ architecture, you don't manage the VMs or the virtual network infrastructure. Instead, you deploy code and the hosting service handles putting that code onto a VM and executing it. This approach tends to favor small granular functions that are coordinated using event-based triggers. For example, a message being placed onto a queue might trigger a function that reads from the queue and processes the message.

[Azure Functions](#) is a serverless compute service that supports various function triggers, including HTTP requests, Service Bus queues, and Event Hubs events. For a complete list, see [Azure Functions triggers and bindings concepts](#). Also consider [Azure Event Grid](#), which is a managed event routing service in Azure.

Orchestrator or serverless?

Here are some factors to consider when choosing between an orchestrator approach and a serverless approach.

Manageability A serverless application is easy to manage, because the platform manages all compute resources for you. While an orchestrator abstracts some aspects of managing and configuring a cluster, it does not completely hide the underlying VMs. With an orchestrator, you will need to think about issues such as load balancing, CPU and memory usage, and networking.

Flexibility and control. An orchestrator gives you a great deal of control over configuring and managing your services and the cluster. The tradeoff is additional complexity. With a serverless architecture, you give up some degree of control because these details are abstracted.

Portability. All of the orchestrators listed here (Kubernetes, DC/OS, Docker Swarm, and Service Fabric) can run on-premises or in multiple public clouds.

Application integration. It can be challenging to build a complex application using a serverless architecture, due to the need to coordinate, deploy, and manage many small independent functions. One option in Azure is to use [Azure Logic Apps](#) to coordinate a set of Azure Functions. For an example of this approach, see [Create a function that integrates with Azure Logic Apps](#).

Cost. With an orchestrator, you pay for the VMs that are running in the cluster. With a serverless application, you pay only for the actual compute resources consumed. In both cases, you need to factor in the cost of any additional services, such as storage, databases, and messaging services.

Scalability. Azure Functions scales automatically to meet demand, based on the number of incoming events. With an orchestrator, you can scale out by increasing the number of service instances running in the cluster. You can also scale by adding additional VMs to the cluster.

Our reference implementation primarily uses Kubernetes, but we did use Azure Functions for one service, namely the Delivery History service. Azure Functions was a good fit for this particular service, because it's an event-driven workload. By using an Event Hubs trigger to invoke the function, the service needed a minimal amount of code. Also, the Delivery History service is not part of the main workflow, so running it outside of the Kubernetes cluster doesn't affect the end-to-end latency of user-initiated operations.

Next steps

Interservice communication

Related resources

- Using domain analysis to model microservices
- Design a microservices architecture
- Expose Azure Spring Apps through a reverse proxy
- Design APIs for microservices

Choose Between Traditional Web Apps and Single Page Apps (SPAs)

Article • 02/25/2023

💡 Tip

This content is an excerpt from the eBook, *Architect Modern Web Applications with ASP.NET Core and Azure*, available on [.NET Docs](#) or as a free downloadable PDF that can be read offline.

[Download PDF](#)



"Atwood's Law: Any application that can be written in JavaScript, will eventually be written in JavaScript."

- *Jeff Atwood*

There are two general approaches to building web applications today: traditional web applications that perform most of the application logic on the server, and single-page applications (SPAs) that perform most of the user interface logic in a web browser, communicating with the web server primarily using web APIs. A hybrid approach is also possible, the simplest being host one or more rich SPA-like subapplications within a larger traditional web application.

Use traditional web applications when:

- Your application's client-side requirements are simple or even read-only.
- Your application needs to function in browsers without JavaScript support.
- Your application is public-facing and benefits from search engine discovery and referrals.

Use a SPA when:

- Your application must expose a rich user interface with many features.
- Your team is familiar with JavaScript, TypeScript, or Blazor WebAssembly development.
- Your application must already expose an API for other (internal or public) clients.

Additionally, SPA frameworks require greater architectural and security expertise. They experience greater churn due to frequent updates and new client frameworks than traditional web applications. Configuring automated build and deployment processes and utilizing deployment options like containers may be more difficult with SPA applications than traditional web apps.

Improvements in user experience made possible by the SPA approach must be weighed against these considerations.

Blazor

ASP.NET Core includes a model for building rich, interactive, and composable user interfaces called Blazor. Blazor server-side allows developers to build UI with C# and Razor on the server and for the UI to be interactively connected to the browser in real-time using a persistent SignalR connection. Blazor WebAssembly introduces another option for Blazor apps, allowing them to run in the browser using WebAssembly. Because it's real .NET code running on WebAssembly, you can reuse code and libraries from server-side parts of your application.

Blazor provides a new, third option to consider when evaluating whether to build a purely server-rendered web application or a SPA. You can build rich, SPA-like client-side behaviors using Blazor, without the need for significant JavaScript development. Blazor applications can call APIs to request data or perform server-side operations. They can interoperate with JavaScript where necessary to take advantage of JavaScript libraries and frameworks.

Consider building your web application with Blazor when:

- Your application must expose a rich user interface
- Your team is more comfortable with .NET development than JavaScript or TypeScript development

If you have an existing web forms application you're considering migrating to .NET Core or the latest .NET, you may wish to review the free e-book, [Blazor for Web Forms](#)

Developers to see whether it makes sense to consider migrating it to Blazor.

For more information about Blazor, see [Get started with Blazor](#).

When to choose traditional web apps

The following section is a more detailed explanation of the previously stated reasons for picking traditional web applications.

Your application has simple, possibly read-only, client-side requirements

Many web applications are primarily consumed in a read-only fashion by the vast majority of their users. Read-only (or read-mostly) applications tend to be much simpler than those applications that maintain and manipulate a great deal of state. For example, a search engine might consist of a single entry point with a textbox and a second page for displaying search results. Anonymous users can easily make requests, and there is little need for client-side logic. Likewise, a blog or content management system's public-facing application usually consists mainly of content with little client-side behavior. Such applications are easily built as traditional server-based web applications, which perform logic on the web server and render HTML to be displayed in the browser. The fact that each unique page of the site has its own URL that can be bookmarked and indexed by search engines (by default, without having to add this functionality as a separate feature of the application) is also a clear benefit in such scenarios.

Your application needs to function in browsers without JavaScript support

Web applications that need to function in browsers with limited or no JavaScript support should be written using traditional web app workflows (or at least be able to fall back to such behavior). SPAs require client-side JavaScript in order to function; if it's not available, SPAs are not a good choice.

Your team is unfamiliar with JavaScript or TypeScript development techniques

If your team is unfamiliar with JavaScript or TypeScript, but is familiar with server-side web application development, then they will probably be able to deliver a traditional web app more quickly than a SPA. Unless learning to program SPAs is a goal, or the user experience afforded by a SPA is required, traditional web apps are a more productive choice for teams who are already familiar with building them.

When to choose SPAs

The following section is a more detailed explanation of when to choose a Single Page Applications style of development for your web app.

Your application must expose a rich user interface with many features

SPAs can support rich client-side functionality that doesn't require reloading the page as users take actions or navigate between areas of the app. SPAs can load more quickly, fetching data in the background, and individual user actions are more responsive since full page reloads are rare. SPAs can support incremental updates, saving partially completed forms or documents without the user having to click a button to submit a form. SPAs can support rich client-side behaviors, such as drag-and-drop, much more readily than traditional applications. SPAs can be designed to run in a disconnected mode, making updates to a client-side model that are eventually synchronized back to the server once a connection is re-established. Choose a SPA-style application if your app's requirements include rich functionality that goes beyond what typical HTML forms offer.

Frequently, SPAs need to implement features that are built into traditional web apps, such as displaying a meaningful URL in the address bar reflecting the current operation (and allowing users to bookmark or deep link to this URL to return to it). SPAs also should allow users to use the browser's back and forward buttons with results that won't surprise them.

Your team is familiar with JavaScript and/or TypeScript development

Writing SPAs requires familiarity with JavaScript and/or TypeScript and client-side programming techniques and libraries. Your team should be competent in writing modern JavaScript using a SPA framework like Angular.

References – SPA Frameworks

- **Angular:** <https://angular.io> ↗
- **React:** <https://reactjs.org/> ↗
- **Vue.js:** <https://vuejs.org/> ↗

Your application must already expose an API for other (internal or public) clients

If you're already supporting a web API for use by other clients, it may require less effort to create a SPA implementation that leverages these APIs rather than reproducing the logic in server-side form. SPAs make extensive use of web APIs to query and update data as users interact with the application.

When to choose Blazor

The following section is a more detailed explanation of when to choose Blazor for your web app.

Your application must expose a rich user interface

Like JavaScript-based SPAs, Blazor applications can support rich client behavior without page reloads. These applications are more responsive to users, fetching only the data (or HTML) required to respond to a given user interaction. Designed properly, server-side Blazor apps can be configured to run as client-side Blazor apps with minimal changes once this feature is supported.

Your team is more comfortable with .NET development than JavaScript or TypeScript development

Many developers are more productive with .NET and Razor than with client-side languages like JavaScript or TypeScript. Since the server-side of the application is already being developed with .NET, using Blazor ensures every .NET developer on the team can understand and potentially build the behavior of the front end of the application.

Decision table

The following decision table summarizes some of the basic factors to consider when choosing between a traditional web application, a SPA, or a Blazor app.

Factor	Traditional Web App	Single Page Application	Blazor App
Required Team Familiarity with JavaScript/TypeScript	Minimal	Required	Minimal
Support Browsers without Scripting	Supported	Not Supported	Supported
Minimal Client-Side Application Behavior	Well-Suited	Overkill	Viable
Rich, Complex User Interface Requirements	Limited	Well-Suited	Well-Suited

Other considerations

Traditional Web Apps tend to be simpler and have better Search Engine Optimization (SEO) criteria than SPA apps. Search engine bots can easily consume content from

traditional apps, while support for indexing SPAs may be lacking or limited. If your app benefits from public discovery by search engines, this is often an important consideration.

In addition, unless you've built a management tool for your SPA's content, it may require developers to make changes. Traditional Web Apps are often easier for non-developers to make changes to, since much of the content is simply HTML and may not require a build process to preview or even deploy. If non-developers in your organization are likely to need to maintain the content of the app, a traditional web app may be a better choice.

SPAs shine when the app has complex interactive forms or other user interaction features. For complex apps that require authentication to use, and thus aren't accessible by public search engine spiders, SPAs are a great option in many cases.

[Previous](#)

[Next](#)

Choose an Azure multiparty computing service

Azure Blob Storage

Azure Kubernetes Service (AKS)

Azure SQL Database

Multiparty computing or privacy-preserving computation allows parties in a business relationship to share data, do computations, and arrive at a mutual result without divulging their private data. Azure services can help you build a multiparty computing solution. The solution can include cloud-based and on-premises resources.

Multiparty computing has the following attributes:

- More than one company or organization is involved.
- The parties are independent.
- The parties don't trust one another with all their data.
- All parties access a common computing and data storage platform.
- Some processes must be private for some of the parties involved.

Azure multiparty computing

This section describes multiparty computing options that are available by using Azure services.

Blockchain with Azure Virtual Machines

You can run ledger software by using Azure Virtual Machines. Create as many virtual machines as you need, and connect them in a blockchain network.

Deploying your own virtual machines allows you to customize your solution. The approach includes management overhead, such as updates, high availability, and business continuity requirements. You might have multiple organizations and multiple cloud accounts. Connecting the individual nodes can be complicated.

Deployment templates are available on Azure for most blockchain ledgers for virtual machines.

Blockchain on Kubernetes

Because most blockchain ledgers support deploying into Docker containers, you can use Kubernetes to manage the containers. Azure has a managed Kubernetes offering called Azure Kubernetes Service (AKS) that you can use to deploy and configure your blockchain nodes.

AKS implementations come with a managed service for the virtual machines that power the AKS cluster. Your organization must still manage your AKS clusters and any networking or storage options in your architecture.

Deployment templates are available on Azure for most blockchain ledgers for AKS.

Blockchain as a service

Azure supports third-party services that run ledger software on Azure. The service provider manages the infrastructure. They handle maintenance and updates. High availability and consortium management are included in the service.

ConsenSys offers Quorum on Azure. Quorum is an open-source protocol layer that supports Ethereum-based applications.

Other offerings might be available in the future.

Azure confidential ledger

Azure confidential ledger is a managed service built on the Confidential Consortium Framework. It implements a permissioned blockchain network of nodes within Azure confidential computing. Confidential ledger builds on existing encryption.

- Existing encryption:
 - **Data at rest.** Encrypt inactive data when stored in blob storage or a database.
 - **Data in transit.** Encrypt data that's flowing between public or private networks.
- Confidential computing:
 - **Data in use.** Encrypt data that's in use, while in memory and during computation.

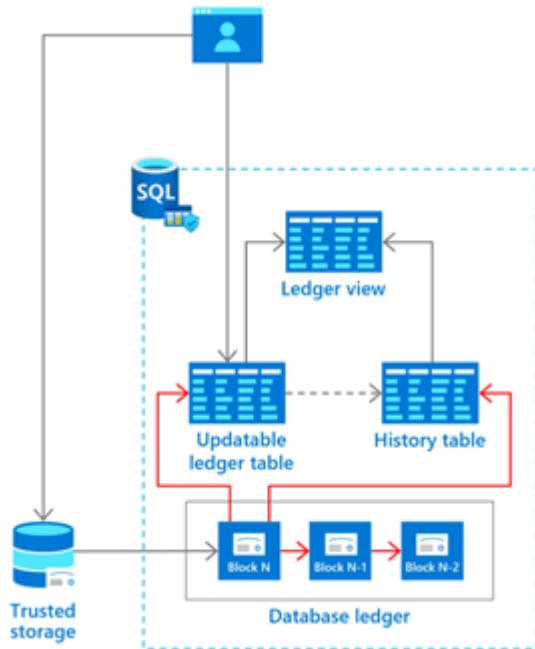
Confidential computing allows encryption of data in the main memory. Confidential computing lets you process data from multiple sources without exposing the input data to other parties. This type of secure computation supports multiparty computing scenarios where data protection is mandatory in every step. Examples might be money laundering detection, fraud detection, and secure analysis of healthcare data.

Data stored in confidential ledger is immutable and tamper-proof in the append-only ledger. The ledger is also independently verifiable. Confidential ledger uses secure

enclaves for a decentralized blockchain network and requires a minimal trusted computing base.

Azure SQL Database ledger

Azure SQL Database ledger allows participants to verify the data integrity of centrally housed data without the network consensus of a blockchain network. For some centralized solutions, trust is important, but decentralized infrastructure isn't necessary. This approach avoids complexity and performance implications of such an infrastructure.



Ledger provides tamper-evidence capabilities for your database. These capabilities allow you to cryptographically attest that your data hasn't been tampered with.

Ledger helps protect data from any attacker or high-privileged user, including database, system, and cloud administrators. Historical data is preserved. If a row is updated in the database, its previous value is maintained in a history table. This capability offers protection without any application changes.

Ledger is a feature of SQL Database. It can be enabled in any existing SQL Database.

Compare options

Use the following tables to compare options so that you can make informed decisions.

Confidential ledger and SQL Database ledger

This table compares confidential ledger with SQL Database ledger.

[Expand table](#)

Capabilities	SQL Database ledger	Confidential ledger
Centralized system that requires tamper evidence	Yes	No
Decentralized system that requires data to be tamper proof	No	Yes
Protects relational data from tampering	Yes	No
Protects unstructured data from tampering	No	Yes
Secure off-chain store of chain data in a blockchain	Yes	No
Secure off-chain store for files referenced to from a blockchain	No	Yes
Relational data is queryable	Yes	No
Unstructured stored data is queryable	No	Yes

Confidential ledger and Azure Blob Storage

The immutable storage feature of Azure Blob Storage ensures that data written to it can be read but never changed. This table compares that technology with confidential ledger.

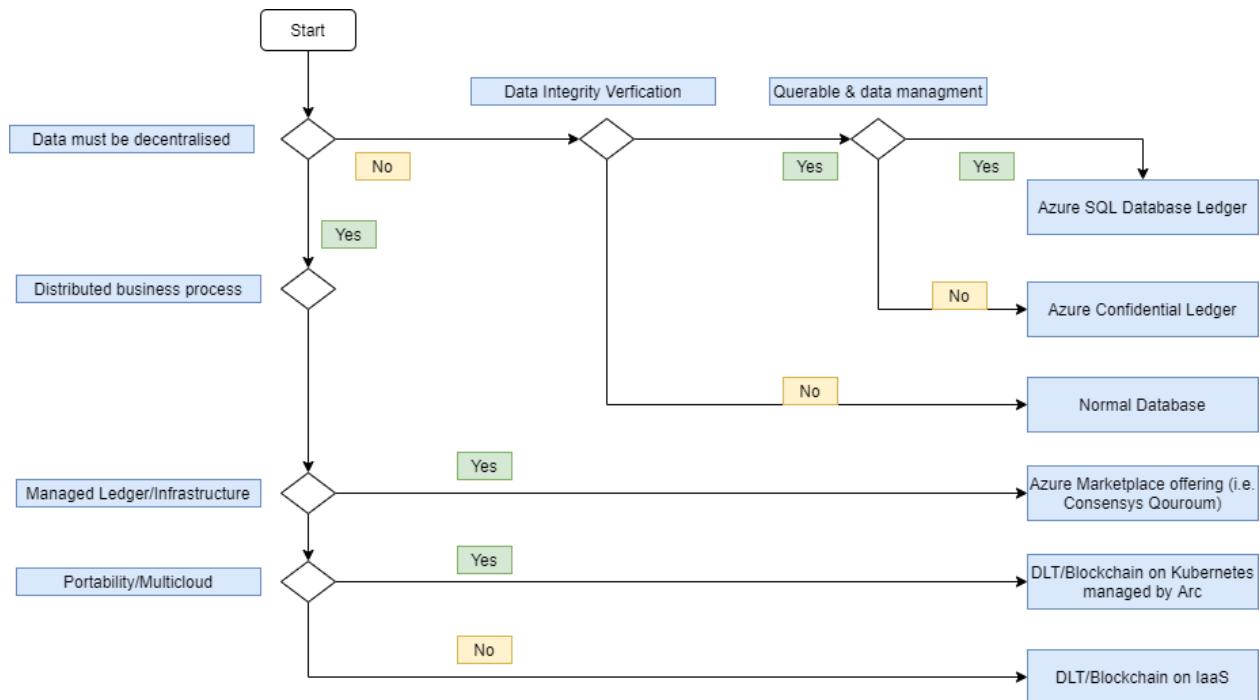
[Expand table](#)

Capabilities	Confidential ledger	Immutable storage
Confidential hardware enclaves	Yes	No
Append-only data integrity	Yes	Yes, limited to intervals

Capabilities	Confidential ledger	Immutable storage
In-use data encryption	Yes	No
Blockchain ledger proof	Yes	No

Multiparty computing decision

This diagram summarizes options for multiparty computing with Azure services.



Next steps

- [Azure confidential ledger ↗](#)
- [Azure SQL Database ledger](#)
- [Azure Virtual Machines ↗](#)
- [Azure Kubernetes Service ↗](#)
- [Azure SQL Database ↗](#)
- [Authenticate Azure confidential ledger nodes](#)
- [Azure confidential ledger architecture](#)

Related resources

- [Supply chain track and trace](#)
- [Blockchain workflow application ↗](#)

- Multicloud blockchain DLT
- Decentralized trust between banks

Choose an Azure container service

Article • 01/08/2024

Azure offers a range of container hosting services that are designed to accommodate various workloads, architectures, and business requirements. This container service selection guide can help you understand which Azure container service is best suited to your workload scenarios and requirements.

ⓘ Note

In this guide, the term *workload* refers to a collection of application resources that support a business goal or the execution of a business process. A workload uses multiple services, like APIs and data stores, that work together to deliver specific end-to-end functionality.

How to use this guide

This guide includes two articles: this introduction article and another article about [considerations that are shared](#) across all workload types.

ⓘ Note

If you aren't yet committed to containerization, see [Choose an Azure compute service](#) for information about other compute options that you can use to host your workload.

This introduction article describes the Azure container services that are in scope for this guide and how the service models compare in terms of tradeoffs between control and ease of use, such as customer-managed versus Microsoft-managed approaches. After you identify candidate services based on your service model preferences, the next step is to evaluate the options against your workload requirements by reviewing the article on [shared considerations](#) for networking, security, operations, and reliability.

This guide takes into consideration tradeoffs that you might need to make, based on the technical requirements, size, and complexity of your workload and the expertise of your workload's team.

Azure container services in scope for this guide

This guide focuses on a subset of the container services that Azure currently offers. This subset provides a mature feature set for web applications and APIs, networking, observability, developer tools, and operations. These container services are compared:



[Azure Container Apps](#) is a fully managed Kubernetes-based application platform that helps you deploy HTTP and non-HTTP apps from code or containers without orchestrating complex infrastructure. For more information, see [Azure Container Apps documentation](#).



[Azure Kubernetes Service \(AKS\)](#) is a managed Kubernetes service for running containerized applications. With AKS, you can take advantage of managed [add-ons](#) and [extensions](#) for additional capabilities while preserving the broadest level of flexibility and control. For more information, see [AKS documentation](#).



[Web App for Containers](#) is a feature of Azure App Service, a fully managed service for hosting HTTP-based web apps with built-in infrastructure maintenance, security patching, scaling, and diagnostic tooling. For more information, see [App Service documentation](#).

For a complete list of all Azure container services, see [the container services product category page](#).

Service model considerations

The service model provides the broadest insight into the level of flexibility and control that any Azure container service provides, in exchange for its overall simplicity and ease of use.

For a general introduction into the terminology and concepts around service models, including infrastructure as a service (IaaS) and platform as a service (PaaS), see [Shared responsibility in the cloud](#).

Comparing the service models of Azure container solutions

AKS

As a hybrid of IaaS and PaaS, AKS prioritizes control over simplicity. Though AKS streamlines the management of the underlying core infrastructure, this VM-based platform is still exposed to your applications and requires appropriate guardrails and processes, like patching, to ensure security and business continuity. The compute infrastructure is supported by additional Azure resources that are hosted directly in your subscription, like Azure load balancers.

AKS also provides access to the Kubernetes API server, which enables you to customize the container orchestration and thus deploy projects from the Cloud Native Computing Foundation (CNCF). Consequently, there's a significant learning curve for workload teams that are new to Kubernetes. If you're new to containerized solutions, this learning curve might be a deterrent. The following PaaS solutions offer a lower barrier to entry. You can move to Kubernetes when your requirements dictate that move.

Azure Container Apps

Container Apps, a PaaS offering, balances control with simplicity. It offers both serverless and dedicated compute options, which abstract away the need to patch the operating system or build guardrails around applications, relative to the operating system. Container Apps also completely abstracts away the container orchestration API and provides a subset of its key functionality through Azure APIs that your team might already be familiar with. Additionally, Layer 7 ingress, traffic splitting, A/B testing, and application lifecycle management are all fully available out of the box.

Web App for Containers

Web App for Containers is also a PaaS offering, but it provides more simplicity, and less control, than Container Apps. It abstracts away container orchestration but still provides appropriate scaling, application lifecycle management, traffic splitting, network integration, and observability.

Hosting model considerations

You can use Azure resources, like AKS clusters, to host multiple workloads. Doing so can help you streamline operations and thereby reduce overall cost. If you choose this path, here are a few important considerations:

- **AKS** is commonly used to host multiple workloads or disparate workload components. You can isolate these workloads and components by using Kubernetes native functionality, like namespaces, access controls, and network controls, to meet security requirements.

You can also use AKS in single-workload scenarios if you need the additional functionality that the Kubernetes API provides and your workload team has enough experience to operate a Kubernetes cluster. Teams with less Kubernetes experience can still successfully operate their own clusters by taking advantage of Azure managed [add-ons](#) and features, like [cluster auto-upgrade](#), to reduce operational overhead.

- **Container Apps** should be used to host a single workload with a shared security boundary. Container Apps has a single top-level logical boundary called a *Container Apps environment*, which also serves as an enhanced-security boundary. There are no mechanisms for additional granular access control. For example, intra-environment communication is unrestricted, and all applications share a single Log Analytics workspace.

If the workload has multiple components and multiple security boundaries, deploy multiple Container Apps environments, or consider AKS instead.

- **Web App for Containers** is a feature of App Service. App Service groups applications into a billing boundary called an *App Service plan*. Because you can scope role-based access control (RBAC) at the application level, it might be tempting to host multiple workloads in a single plan. However, we recommend that you host a single workload per plan to avoid the Noisy Neighbor problem. All apps in a single App Service plan share the same allocated compute, memory, and storage.

When you consider hardware isolation, you need to be aware that App Service plans generally run on infrastructure that's shared with other Azure customers. You can choose Dedicated tiers for dedicated VMs or Isolated tiers for dedicated VMs in a dedicated virtual network.

In general, all Azure container services can host multiple applications that have multiple components. However, Container Apps and Web App for Containers are better suited

for a single-workload component or multiple highly related workload components that share a similar lifecycle, where a single team owns and runs the applications.

If you need to host disparate, potentially unrelated application components or workloads on one host, consider AKS.

The tradeoff between control and ease of use

AKS provides the most configurability, but this configurability comes at the cost of increased operational overhead, as compared to the other services. Although Container Apps and Web App for Containers are both PaaS services that have similar levels of Microsoft-managed features, Web App for Containers emphasizes simplicity to cater to its target audience: existing Azure PaaS customers, who find the interface familiar.

Rule of thumb

Generally, services that offer more simplicity tend to suit customers who prefer to focus more on feature development and less on infrastructure. Services that offer more control tend to suit customers who need more configurability and have the skills, resources, and business justification necessary to manage their own infrastructure.

Shared considerations across all workloads

Although a workload team might prefer a particular service model, that model might not meet the requirements of the organization as a whole. For example, developers might prefer less operational overhead, but security teams might consider this type of overhead necessary to meet compliance requirements. Teams need to collaborate to make the appropriate tradeoffs.

Be aware that shared considerations are broad. Only a subset might be relevant to you, depending not just on the workload type but also on your role within the organization.

The following table provides a high-level overview of considerations, including service feature comparisons. Review the considerations in each category and compare them against your workload's requirements.

Expand table

Category	Overview
Networking considerations	Networking in Azure container services varies depending on your preference for simplicity versus configurability. AKS is highly configurable, providing

Category	Overview
	<p>extensive control over network flow, but it requires more operational effort. Container Apps offers Azure-managed networking features. It's a middle ground between AKS and Web App for Containers, which is tailored to customers who are familiar with App Service.</p> <p>Crucially, network design decisions can have long-term consequences because of the challenges of changing them without re-deploying workloads. Several factors, like IP address planning, load balancing responsibilities, service discovery methods, and private networking capabilities, differ across these services. You should carefully review how the services meet specific networking requirements.</p>
Security considerations	<p>Container Apps, AKS, and Web App for Containers all provide integration with key Azure security offerings like Azure Key Vault and managed identities. AKS offers additional features like runtime threat protection and network policies. Although it might seem that PaaS services like Container Apps offer fewer security features, that's partly because more of the underlying infrastructure components are managed by Azure and not exposed to customers, which reduces risk.</p>
Operational considerations	<p>Although AKS offers the most customization, it demands greater operational input. In contrast, PaaS solutions like Container Apps and Web App for Containers let Azure handle tasks like OS updates. Scalability and hardware SKU flexibility are crucial. AKS provides flexible hardware options, whereas Container Apps and Web App for Containers provide set configurations. Application scalability in AKS is the sole responsibility of the customer. Container Apps and Web App for Containers offer more streamlined approaches.</p>
Reliability considerations	<p>Web App for Containers and Container Apps health probe configurations are more streamlined than those of AKS, given that they use the familiar Azure Resource Manager API. AKS requires the use of the Kubernetes API. It also requires you to take on the additional responsibility of managing Kubernetes node pool scalability and availability in order to properly schedule application instances. These requirements result in additional overhead for AKS.</p> <p>Moreover, SLAs for Container Apps and Web App for Containers are more straightforward than those of AKS, for which the control plane and node pools each have their own SLAs and need to be compounded accordingly. All services offer zone redundancy in datacenters that offer it.</p>

After reviewing the preceding considerations, you still might not have found the perfect fit. That's perfectly normal.

Evaluating tradeoffs

Choosing a cloud service isn't a straightforward exercise. Given the complexity of cloud computing, the collaboration between many teams, and resource constraints involving people, budgets, and time, every solution has tradeoffs.

Be aware that, for any given workload, some requirements might be more critical than others. For example, an application team might prefer a PaaS solution like Container Apps but choose AKS because their security team requires deny-by-default network controls between colocated workload components, which is an AKS-only feature that uses Kubernetes network policies.

Finally, note that the preceding shared considerations include the most common requirements but aren't comprehensive. It's the workload team's responsibility to investigate every requirement against the preferred service's feature set before confirming a decision.

Conclusion

This guide describes the most common considerations that you face when you choose an Azure container service. It's designed to guide workload teams in making informed decisions. The process starts with choosing a cloud service model, which involves determining the desired level of control. Control comes at the expense of simplicity. In other words, it's a process of finding the right balance between a self-managed infrastructure and one that's managed by Microsoft.

Many workload teams can choose an Azure container service solely based on the preferred service model: PaaS versus IaaS. Other teams need to investigate further to determine how service-specific features address workload or organizational requirements.

All workload teams should use this guide in addition to incorporating due diligence to avoid difficult-to-reverse decisions. Be aware, however, that the decision isn't confirmed until developers try the service and decide based on experience rather than theory.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Andre Dewes](#) | Senior Customer Engineer
- [Marcos Martinez](#) | Senior Service Engineer

- [Julie Ng](#) | Senior Engineer

Other contributors:

- [Mick Alberts](#) | Technical Writer
- [Martin Gjoshevski](#) | Senior Customer Engineer
- [Don High](#) | Principal Customer Engineer
- [Nelly Kiboi](#) | Service Engineer
- [Xuhong Liu](#) | Senior Service Engineer
- [Faisal Mustafa](#) | Senior Customer Engineer
- [Walter Myers](#) | Principal Customer Engineering Manager
- [Sonalika Roy](#) | Senior Customer Engineer
- [Paolo Salvatori](#) | Principal Customer Engineer
- [Victor Santana](#) | Principal Customer Engineer

Next step

Learn more about shared architectural considerations for the services mentioned in this article.

[Shared architectural considerations](#)

Preread Recommendations

Article • 11/15/2023

This document is designed to help you guide through the process of selecting a container offering on Azure Confidential Computing that best suits your workload requirements and security posture. To make the most of the guide, we recommend the following prereads.

Azure Compute Decision Matrix

Familiarize yourself with the overall [Azure Compute offerings](#) to understand the broader context in which Azure Confidential Computing operates.

Introduction to Azure Confidential Computing

Azure Confidential Computing offers solutions to enable isolation of your sensitive data while it's being processed in the cloud. You can read more about confidential computing [Azure confidential computing](#).

Attestation

Attestation is a process that provides assurances regarding the integrity and identity of the hardware and software environments in which applications run. In Confidential Computing, attestation allows you to verify that your applications are running on trusted hardware and in a trusted execution environment.

Learn more about attestation and Microsoft Azure Attestation service at [Attestation in Azure](#)

Definition of memory isolation

In confidential computing, memory isolation is a critical feature that safeguards data during processing. The Confidential Computing Consortium defines memory isolation as:

"Memory isolation is the ability to prevent unauthorized access to data in memory, even if the attacker has compromised the operating system or other privileged software. This is achieved by using hardware-based features to create a secure and isolated environment for confidential workload."

Choosing a Container offering on Azure Confidential Computing

Azure Confidential Computing offers various solutions for container deployment and management, each tailored for different levels of isolation and attestation capabilities.

Your current setup and operational needs dictate the most relevant path through this document. If you're already utilizing Azure Kubernetes Service (AKS) or have dependencies on Kubernetes APIs, we recommend following the AKS paths. On the other hand, if you're transitioning from a Virtual Machine setup and are interested in exploring serverless containers, the ACI (Azure Container Instances) path should be of interest.

Azure Kubernetes Service (AKS)

Confidential VM Worker Nodes

- **Guest Attestation:** Ability to verify that you're operating on a confidential virtual machine provided by Azure.
- **Memory Isolation:** VM level isolation with unique memory encryption key per VM.
- **Programming model:** Zero to minimal changes for containerized applications. Support is limited to containers that are Linux based (containers using a Linux base image for the container).

You can find more information on [Getting started with CVM worker nodes with a lift and shift workload to CVM node pool](#).

Confidential Containers on AKS

- **Full Guest Attestation:** Enables attestation of the full confidential computing environment including the workload.
- **Memory Isolation:** Node level isolation with a unique memory encryption key per VM.
- **Programming model:** Zero to minimal changes for containerized applications (containers using a Linux base image for the container).
- **Ideal Workloads:** Applications with sensitive data processing, multi-party computations, and regulatory compliance requirements.

You can find more information on [Getting started with CVM worker nodes with a lift and shift workload to CVM node pool](#).

Confidential Computing Nodes with Intel SGX

- **Application enclave Attestation:** Enables attestation of the container running, in scenarios where the VM isn't trusted, but only the application is trusted, ensuring a heightened level of security and trust in the application's execution environment.
- **Isolation:** Process level isolation.
- **Programming model:** Requires the use of open-source library OS or vendor solutions to run existing containerized applications. Support is limited to containers that are Linux based (containers using a Linux base image for the container).
- **Ideal Workloads:** High-security applications such as key management systems.

You can find more information about the offering and our partner solutions [here](#).

Serverless

Confidential Containers on Azure Container Instances (ACI)

- **Full Guest Attestation:** Enables attestation of the full confidential computing environment including the workload.
- **Isolation:** Container group level isolation with a unique memory encryption key per container group.
- **Programming model:** Zero to minimal changes for containerized applications. Support is limited to containers that are Linux based (containers using a Linux base image for the container).
- **Ideal Workloads:** Rapid development and deployment of simple containerized workloads without orchestration. Support for bursting from AKS using Virtual Nodes.

You can find more details at [Getting started with Confidential Containers on ACI](#).

Learn more

| [Intel SGX Confidential Virtual Machines on Azure](#) [Confidential Containers on Azure](#)

Choose a Kubernetes at the edge compute option

Article • 12/16/2022

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

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

Assumptions

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

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

Kubernetes choices at a glance

	Operational cost	Ease of configuration	Flexibility	Mixed node	Summary
Bare-metal Kubernetes	High**	Difficult**	High**	Yes	A ground-up configuration on any available infrastructure at location with the option to use Azure Arc for added Azure capabilities.

	Operational cost	Ease of configuration	Flexibility	Mixed node	Summary
K8s on Azure Stack Edge Pro	Low	Easy	Low	Linux only	Kubernetes deployed on Azure Stack Edge appliance deployed at location.
AKS hybrid	Low	Easy	Medium	Yes	AKS deployed on Azure Stack HCI or Windows Server 2019.

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

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

Bare-metal Kubernetes

Ground-up configuration of Kubernetes using tools like [kubeadm](#) on any underlying infrastructure.

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

Scenario

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

- This option can be especially difficult for those who are new to Kubernetes. This isn't uncommon for organizations looking to run edge clusters. Options like [MicroK8s](#) or [k3s](#) aim to flatten that learning curve.
- It's important to understand any underlying infrastructure and any integration that is expected to take place up front. This will help to narrow down viable options and to identify any gaps with the open-source tooling and/or plugins.

- Enabling clusters with [Azure Arc](#) presents a simple way to manage your cluster from Azure alongside other resources. This also brings other Azure capabilities to your cluster, including [Azure Policy](#), [Azure Monitor](#), [Microsoft Defender for Cloud](#), and other services.
- Because cluster configuration isn't trivial, it's especially important to be mindful of CI/CD. Tracking and acting on upstream changes of various plugins, and making sure those changes don't affect the health of your cluster, becomes a direct responsibility. It's important for you to have a strong CI/CD solution, strong testing, and monitoring in place.

Tooling options

Cluster bootstrap:

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

Storage:

- Explore available [CSI drivers](#): Many options are available to fit your requirements from cloud to local file shares.

Networking:

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

Considerations

Operational cost:

- Without the support that comes with managed services, it's up to the organization to maintain and operate the cluster as a whole (storage, networking, upgrades, observability, application management). The operational cost is considered high.

Ease of configuration:

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

Flexibility:

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

Kubernetes on Azure Stack Edge

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

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

Scenario

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

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

- Support for two node clusters isn't currently available. This effectively means that this option is *not* a highly available (HA) solution.

Considerations

Operational cost:

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

Ease of configuration:

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

Flexibility:

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

AKS hybrid

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

Scenario

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

Considerations

Operational cost:

- Microsoft-supported cluster minimizes operational costs.

Ease of configuration:

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

Flexibility:

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

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Prabhjot Kaur](#) | Principal Cloud Solution Architect

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

For more information, see the following articles:

- [What is Azure IoT Edge](#)
- [Kubernetes on your Azure Stack Edge Pro GPU device](#)
- [Use IoT Edge module to run a Kubernetes stateless application on your Azure Stack Edge Pro GPU device](#)
- [Deploy a Kubernetes stateless application via kubectl on your Azure Stack Edge Pro GPU device](#)
- [Use Kubernetes dashboard to monitor your Azure Stack Edge Pro GPU device](#)

Related resources

- [AI at the edge with Azure Stack Hub](#)
- [Building a CI/CD pipeline for microservices on Kubernetes](#)

Choose a bare-metal Kubernetes at the edge platform option

Azure Kubernetes Service (AKS)

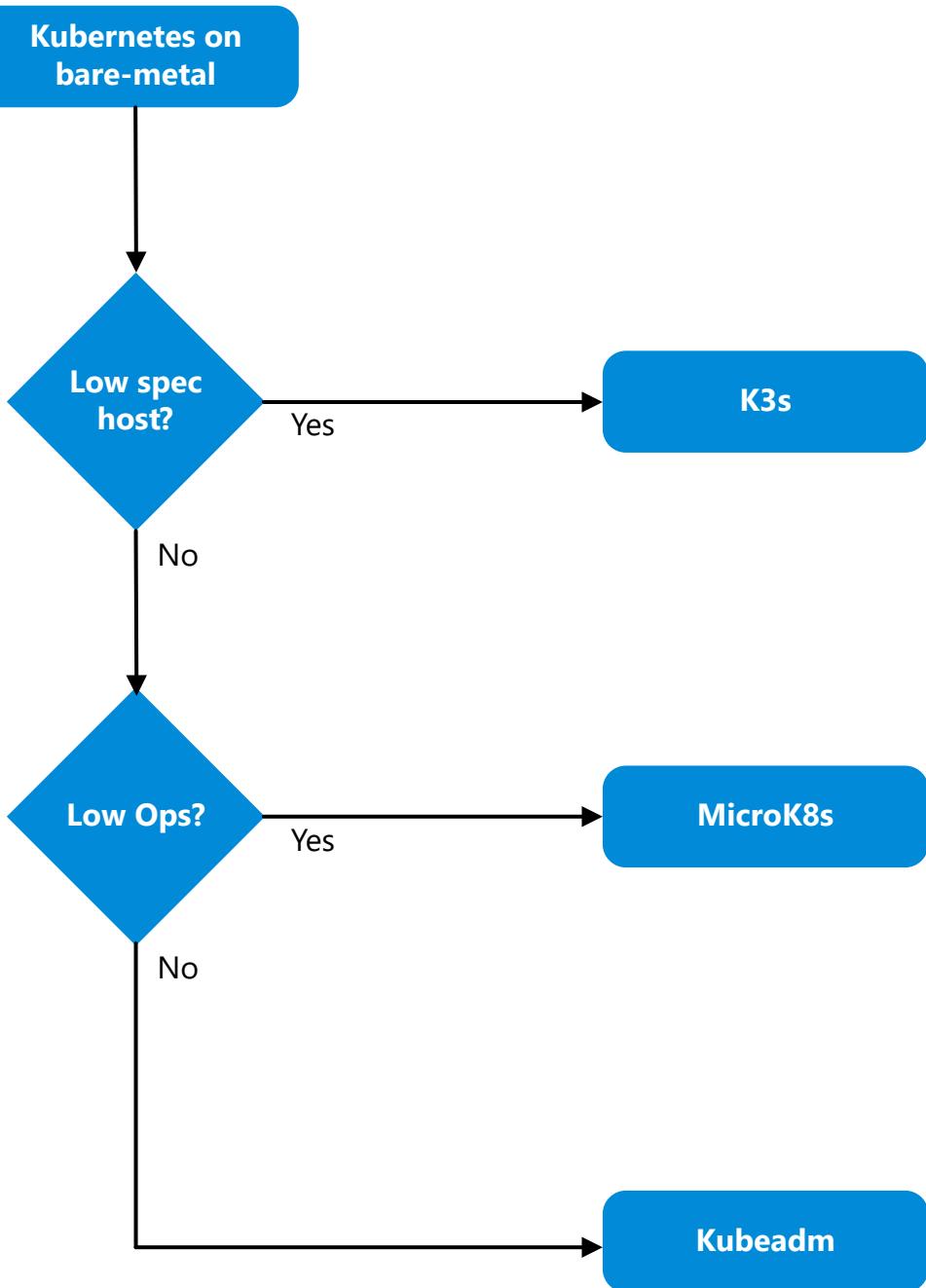
If you're looking to run Kubernetes at the edge and notice that managed solutions aren't quite meeting your requirements, you may be [exploring bare-metal as an option](#). This document helps you find the best available option for your use case when configuring Kubernetes clusters at the edge.

ⓘ Note

This article is not an exhaustive comparison; rather, it presents potential paths for making decisions based on major qualifiers between common options.

Decision tree for bare-metal Kubernetes at the edge

Reference the following tree when deciding between the options presented below for bare-metal Kubernetes at the edge.



[Download a Visio file ↗ of this flowchart](#)

- [MicroK8s ↗](#): Conformant "Low Ops" Kubernetes by Canonical
- [K3s ↗](#): Certified Kubernetes distribution built for IoT and edge computing
- [kubeadm ↗](#): Kubernetes tool for creating ground-up Kubernetes clusters; good for standard compute (Linux/Windows)

Note

Low Ops refers to the decreased cost of operations when some operational tasks are abstracted or made easier, like auto updates or simplified upgrades.

MicroK8s by Canonical

MicroK8s is delivered as a single **snap** package that can be easily installed on Linux machines with snap support. Alternative installs are available for Windows, macOS, and raspberry PI/ARM. When installed, MicroK8s creates a single-node cluster, which can be managed with the MicroK8s tooling. It's packaged with its own kubectl, and certain addons may be enabled (for example, helm, dns, ingress, metallb, and [more ↗](#)). Multinode, Windows nodes, and high-availability (HA) scenarios are also supported.

Considerations:

- There are various resource requirements depending on where you want to run MicroK8s. Reference the [product docs ↗](#) for minimum resource requirements. For example:
 - Ubuntu: 4-GB RAM, 20-GB disk space
 - Windows: 4-GB RAM, 40-GB disk space
- Windows workloads are only supported for MicroK8s clusters with Calico CNI.
- Each node on a MicroK8s multinode cluster requires its own environment to work in, whether that is a separate VM or container on a single machine or a different machine on the same network.
- Difficulties may crop up when running MicroK8s on some ARM hardware. Reference the [docs ↗](#) for potential remedies.

K3s by Rancher

K3s is a lightweight distribution of Kubernetes. K3s is deployed as a single binary and comes with embedded tools such as kubectl and ctr, similar to MicroK8s.

Considerations:

- The binary is less than 100 MB, but there are still minimum resource requirements depending on your scenario. Reference the [docs](#) for minimum resource requirements.
- SQLite3 is the default storage system, though [other options](#) are supported.
- Windows nodes aren't currently supported for K3s.
- HA can be achieved with either an [external database](#) or an [embedded database](#). K3s has added full support for embedded etcd as of release v1.19.5+k3s1.

kubeadm

Kubeadm is a plain vanilla installation of Kubernetes from the ground up.

Considerations:

- Requires 2 GiB (gibibytes) or more of RAM per machine.
- Requires at least 2 CPUs on control-plane node.
- The control-plane node must be a machine running a deb/rpm-compatible Linux OS.
- [The Kubernetes version and version skew support policy](#) applies to *kubeadm* and to Kubernetes overall. Check that policy to learn about what versions of Kubernetes and kubeadm are supported.

Management/Automation

When it comes to automation and management of the provisioning of bare-metal clusters, there are a couple of options to explore: Ansible and Metal3.

[Ansible](#) provides an easy way to manage remote resources and therefore is a prime candidate to manage and join remote nodes to a Kubernetes cluster. All you need is the Ansible binary, running on a Linux machine, and SSH on remote machines. This method provides a flexible mechanism to run arbitrary scripts on target machines, which means you could use Ansible with any of the tools mentioned above.

[Metal3](#) takes a different approach to solve this problem by utilizing similar concepts to [Cluster API](#). You'll need to instantiate an ephemeral cluster to provision and manage bare-metal clusters using native Kubernetes objects. At the time of writing,

Metal3 uses kubeadm and therefore doesn't support lightweight Kubernetes distributions.

For management beyond cluster provisioning, consider learning about [Azure Arc](#)–enabled clusters to manage your clusters in Azure.

Next steps

For more information, see the following articles:

- [Reference implementation ↗](#)
- [What is Azure IoT Edge](#)
- [Introduction to Kubernetes on Azure](#)
- [Kubernetes on your Azure Stack Edge Pro GPU device](#)

Related resources

- [Choosing a Kubernetes at the edge compute option](#)
- [Baseline architecture for an Azure Kubernetes Service \(AKS\) cluster](#)
- [Microservices architecture on Azure Kubernetes Service \(AKS\)](#)
- [AI at the edge with Azure Stack Hub](#)
- [Building a CI/CD pipeline for microservices on Kubernetes](#)

Azure hybrid options

Azure Arc Azure IoT Edge Azure Stack Edge Azure Stack HCI Azure Stack Hub

Azure offers several hybrid solutions that can host applications and workloads, extend Azure services, and provide security and operational tooling for hybrid environments. Azure hybrid services range from virtualized hardware that hosts traditional IT apps and databases to integrated platform as a service (PaaS) solutions for on-premises, edge, and multicloud scenarios. This guide helps you choose a hybrid solution that meets your business requirements.

Hybrid concepts

Hybrid environments include the following types of hosting locations and infrastructure:

- **Hybrid cloud:** These environments combine public cloud services with on-premises infrastructure. This hybrid strategy is common for organizations that have strict data sovereignty regulations, low latency requirements, or crucial resiliency and business continuity needs.
- **Edge:** These environments host devices that provide on-premises computing and data storage. This approach is common for organizations and applications that need to remain close to the data, reduce latency, or compute data in near real time.
- **Multicloud:** These environments use multiple cloud computing services and providers. This strategy provides flexibility, can reduce risk, and lets organizations investigate and use different providers for specific applications. But this approach often requires cloud-specific knowledge and adds complexity to management, operations, and security.

Hybrid solutions encompass a system's [control plane and data plane](#).

- **Control plane:** This plane refers to resource management operations, such as creating Azure virtual machines (VMs). Azure uses [Azure Resource Manager](#) to handle the control plane.
- **Data plane:** This plane uses the capabilities of resource instances that the control plane creates, such as accessing Azure VMs over remote desktop protocol (RDP).

Azure hybrid solutions can extend Azure control plane operations outside of Azure datacenters, or run dedicated control plane instances, to provide data plane capabilities.

Hybrid considerations

To make a hybrid solution decision, you must consider hardware, hosting and deployment, and application or workload requirements and constraints. Hybrid solutions must also support developer operations (DevOps) and comply with organizational and industry standards and regulations.

Hardware

Depending on workload type, you might need traditional datacenter hardware that can run VMs, containers, and databases. For other scenarios, like IoT deployments, restricted hardware devices are a better fit and can run on rack, portable, or ruggedized servers.

Consider whether to refresh, repurpose, or replace existing hardware. Brownfield scenarios use existing hardware in modern hybrid workload approaches. Greenfield scenarios acquire new hardware or use hardware as a service with a monthly fee.

Hosting and deployment

Consider whether to use on-premises datacenter, edge, Azure cloud, or multicloud hosting with a consistent cloud-native technology approach. Business, compliance, cost, or security requirements might determine hosting location.

A large-scale application deployment is different from smaller-scale implementations. A traditional IT deployment to VMs and databases is different from deployments to containers or distributed devices.

Distributed, complex, large-scale deployments must be able to massively scale service implementation, and might address concerns like business continuity differently than traditional IT.

Application or workload

Consider whether applications or workloads are distributed, containerized, or traditional IT hosted on VMs or databases. [Azure IoT Hub](#), [Azure Kubernetes Service \(AKS\)](#) clusters, or PaaS solutions outside Azure datacenters can host hybrid workloads.

Traditional applications that run on VMs benefit from hyperconverged infrastructure (HCI) and Azure operational, security, and management tooling for day-two operations. Cloud-native applications are better suited to run on container orchestrators like AKS and use Azure PaaS solutions.

If you need to deploy models built and trained in the cloud and run them on-premises, monitor IoT devices at scale, or provide Azure data transfer options, consider edge deployments and solutions.

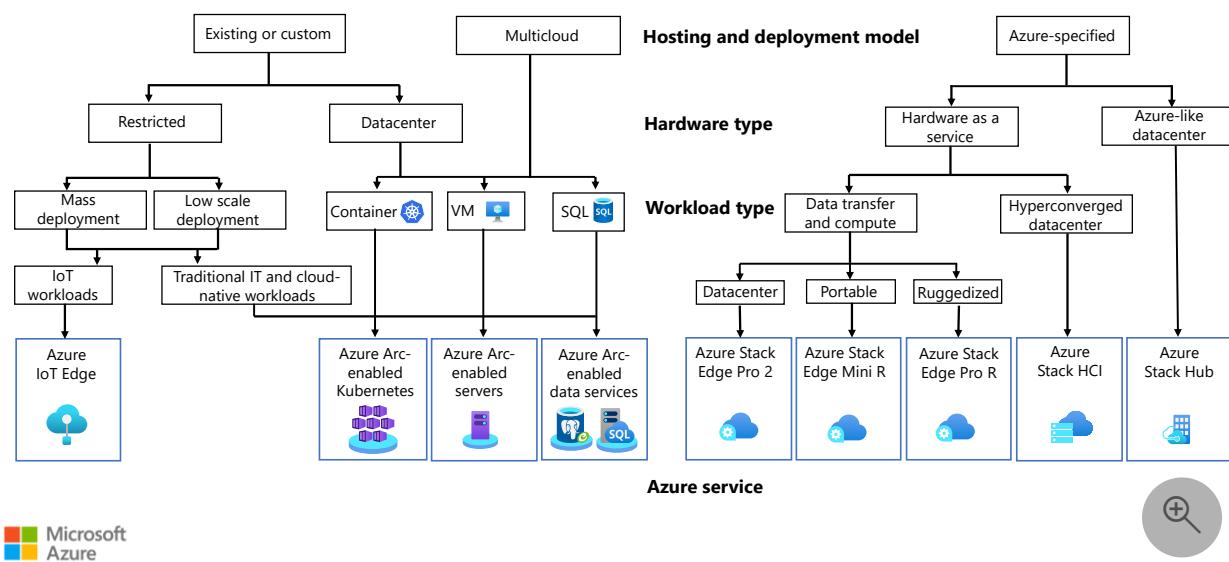
Choose a hybrid solution

All the preceding factors are important for the final solution, but depending on requirements, background, and expertise, organizations might approach solution evaluation from different perspectives. Organizations might start with their hardware and hosting requirements and constraints, or by investigating Azure services from an application and workload perspective. DevOps teams might focus on mass deployments and restricted or purpose-built hardware, while systems administrators might emphasize hosting location or hardware and hypervisor usage.

The following sections present a hybrid solution decision tree based on deployment model and an Azure hybrid service matrix describing supported workloads, hardware types, and deployment models. Work through these illustrations to choose a candidate solution. Then, carry out a detailed evaluation of the candidate services to see if they meet your needs.

Hybrid solution decision tree

The following decision tree starts with choosing an existing or custom, multicloud, or Azure-specified hybrid solution. The tree proceeds through decision points to select an appropriate Azure hybrid service.



Download a [PowerPoint file](#) of all diagrams in this article.

For *existing or custom* deployments:

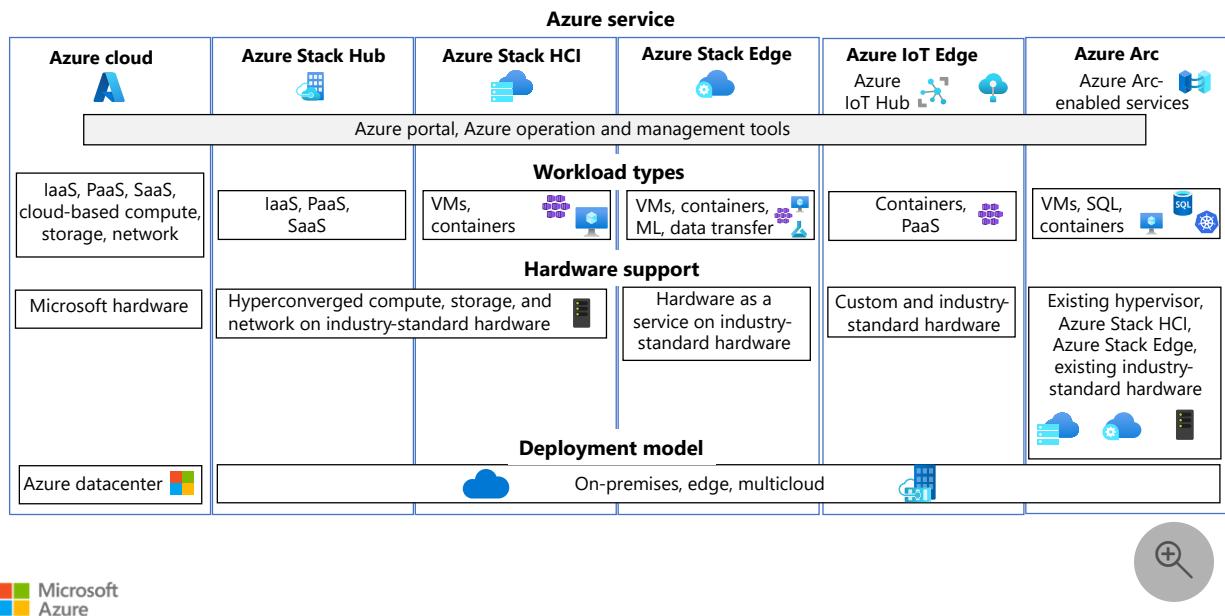
1. Decide whether the hardware is **restricted** or deployed in a **datacenter**.
2. For **restricted** hardware, decide whether the deployment is **mass** or **low scale**.
3. For **datacenter** and **multicloud** deployments, determine whether the workload type uses **containers** or traditional IT deployment in **VMs** or **SQL** databases.
4. Existing and custom **IoT workloads** can use [Azure IoT Edge](#). Existing and custom traditional, database, and cloud-native deployments can use [Azure Arc](#)-enabled servers and services.
5. **Container-based** deployments can use Azure Arc-enabled Kubernetes. **VM-based** deployments can use Azure Arc-enabled servers. **SQL** database deployments can use Azure Arc-enabled data services.

For *Azure-specified* deployments:

1. Decide whether you want **hardware as a service** or **Azure datacenter-like** deployments. Azure **datacenter-like** deployments can use [Azure Stack Hub](#).
2. For **hardware as a service**, decide whether your workload type uses **data transfer and compute** or a [hyperconverged](#) infrastructure (HCI). For a [hyperconverged](#) solution, you can use [Azure Stack HCI](#).
3. **Data transfer and compute** workloads can use [Azure Stack Edge](#). **Datacenter** deployments can use [Azure Stack Edge Pro 2](#). **Portable** deployments can use [Azure Stack Edge Mini R](#). **Ruggedized** deployments can use [Azure Stack Edge Pro R](#).

Azure hybrid services matrix

The following decision matrix presents supported workloads, hardware capabilities, and deployment models for several Azure hybrid services. All Azure services include the Azure portal and other Azure operations and management tools.



Microsoft Azure



Download a [PowerPoint file](#) of all diagrams in this article.

- The *Azure cloud* provides cloud-based software as a service (SaaS), infrastructure as a service (IaaS), and PaaS compute, storage, and network services. The services run on Microsoft hardware in Azure datacenters.
- **Azure Stack** is a family of products and solutions that extend Azure to the edge or to on-premises datacenters. Azure Stack provides several solutions for various use cases.
 - [Azure Stack Hub](#) extends Azure to run apps in on-premises environments. Azure Stack Hub provides SaaS, IaaS, and PaaS hyperconverged compute, storage, and network services and runs on industry-standard hardware on-premises or in multicloud datacenters. Azure Stack Hub delivers Azure services to datacenters with integrated systems and can run on connected or disconnected environments.
 - [Azure Stack HCI](#) is a hyperconverged solution that uses validated hardware to run virtualized and containerized workloads on-premises. Azure Stack HCI provides VM-based and AKS-based hyperconverged compute, storage, and network services and runs on industry-standard hardware on-premises or in multicloud datacenters. Azure Stack HCI connects workloads to Azure for cloud services and management.
 - [Azure Stack Edge](#) delivers Azure capabilities such as compute, storage, networking, and hardware-accelerated machine learning to edge locations. Azure Stack Edge provides VM-based, AKS-based, machine learning, and data transfer services on industry-standard hardware as a service and runs on-premises or in multicloud datacenters.

- [Azure IoT Edge](#) and [IoT Hub](#) deploy custom functionality to mass devices. IoT Edge natively integrates with IoT Hub to provide DevOps, PaaS, and containerized services on custom and industry-standard hardware and runs on-premises or in multicloud datacenters.
- [Azure Arc](#) provides application delivery and management by using Azure Arc-enabled services for VMs, SQL databases, and Kubernetes. Azure Arc projects existing bare metal, VM, and Kubernetes infrastructure resources into Azure to handle operations with Azure management and security tools. Azure Arc simplifies governance and management by delivering a consistent multicloud and on-premises management platform for Azure services.

Azure Arc runs on existing industry-standard hardware, hypervisors, Azure Stack HCI, or Azure Stack Edge, on-premises or in multicloud datacenters. Azure Arc includes the following capabilities:

- [Azure Arc-enabled servers](#)
- [SQL Server on Azure Arc-enabled servers](#)
- [Azure Arc-enabled Kubernetes](#)
- [Azure Arc-enabled vSphere](#)
- [Arc-enabled System Center Virtual Machine Manager](#)
- [Azure Arc-enabled VMs on Azure Stack HCI](#)

Azure Arc-enabled services let you create on-premises and multicloud applications with Azure PaaS and data services such as [Azure App Service](#), [Azure Functions](#), [Azure Logic Apps](#), [Azure SQL Managed Instance](#), [PostgreSQL Hyperscale](#), and [Azure Machine Learning](#). You can run these services anywhere and use existing infrastructure.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors:

- [Robert Eichenseer](#) | Sr Service Engineer
- [Laura Nicolas](#) | Sr Software Engineer

To see nonpublic LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Overview of a hybrid workload](#)
- [Azure hybrid and multicloud patterns and solutions documentation](#)

- Introduction to hybrid and multicloud
- Introduction to Azure hybrid cloud services (Learn module)

Related resources

- Hybrid architecture design
- Run containers in a hybrid environment
- Implement a secure hybrid network

Differences between Azure Stack Hub and Azure when using services and building apps

Article • 03/26/2021

Before you use services or build apps for Azure Stack Hub, it's important to understand the differences between Azure Stack Hub and global Azure. This article identifies different features and key considerations when using Azure Stack Hub as your hybrid cloud development environment.

Overview

Azure Stack Hub is a hybrid cloud platform that enables you to use Azure services from your company or service provider datacenter. You can build an app on Azure Stack Hub and then deploy it to Azure Stack Hub, to Azure, or to your Azure hybrid cloud.

Your Azure Stack Hub operator tells you which services are available for you to use, and how to get support. They offer these services through their customized plans and offers.

The [Azure technical documentation content](#) assumes that apps are being developed for an Azure service and not for Azure Stack Hub. When you build and deploy apps to Azure Stack Hub, you must understand some key differences, such as:

- Azure Stack Hub delivers a subset of the services and features that are available in Azure.
- Your company or service provider can choose which services they want to offer. The available options might include customized services or applications. They may offer their own customized documentation.
- Use the correct Azure Stack Hub-specific endpoints (for example, the URLs for the portal address and the Azure Resource Manager endpoint).
- You must use PowerShell and API versions that are supported by Azure Stack Hub. Using supported versions ensures that your apps work in both Azure Stack Hub and Azure.

High-level differences

The following table describes the high-level differences between Azure Stack Hub and global Azure. Note these differences when you develop for Azure Stack Hub or use Azure Stack Hub services:

Area	Azure (global)	Azure Stack Hub
Who operates it?	Microsoft	Your organization or service provider.
Who do you contact for support?	Microsoft	<p>For an integrated system, contact your Azure Stack Hub operator (at your organization or service provider) for support.</p> <p>For Azure Stack Development Kit (ASDK) support, visit the Microsoft forums. Because the development kit is an evaluation environment, there's no official support offered through Microsoft Support.</p>
Available services	See the list of Azure services . Available services vary by Azure region.	Azure Stack Hub supports a subset of Azure services. Actual services will vary based on what your organization or service provider chooses to offer.
Azure Resource Manager endpoint*	https://management.azure.com	<p>For an Azure Stack Hub integrated system, use the endpoint that your Azure Stack Hub operator provides.</p> <p>For the ASDK, use: https://management.local.azurestack.external.</p>
Portal URL*	https://portal.azure.com	<p>For an Azure Stack Hub integrated system, use the URL that your Azure Stack Hub operator provides.</p> <p>For the ASDK, use: https://portal.local.azurestack.external.</p>
Region	You can select which region you want to deploy to.	<p>For an Azure Stack Hub integrated system, use the region that's available on your system.</p> <p>For the Azure Stack Development Kit (ASDK), the region is always local.</p>
Resource groups	A resource group can span regions.	For both integrated systems and the development kit, there's only one region.
Supported namespaces, resource types, and API versions	The latest (or earlier versions that aren't yet deprecated).	Azure Stack Hub supports specific versions. See the Version requirements section of this article.

Area	Azure (global)	Azure Stack Hub

*If you're an Azure Stack Hub operator, for more information see [Using the administrator portal](#) and [Administration basics](#).

Helpful tools and best practices

Microsoft provides tools and guidance that help you develop for Azure Stack Hub.

[\[+\] Expand table](#)

Recommendation	References
Install the correct tools on your developer workstation.	<ul style="list-style-type: none"> - Install PowerShell - Download tools - Configure PowerShell - Install Visual Studio
Review information about the following items: - Azure Resource Manager template considerations. - How to find quickstart templates. - Use a policy module to help you use Azure to develop for Azure Stack Hub.	Develop for Azure Stack Hub
Review and follow the best practices for templates.	Resource Manager Quickstart Templates

Version requirements

Azure Stack Hub supports specific versions of Azure PowerShell and Azure service APIs. Use supported versions to ensure that your app can deploy to both Azure Stack Hub and to global Azure.

To make sure that you use a correct version of Azure PowerShell, use [API version profiles](#). To determine the latest API version profile that you can use, determine the build of Azure Stack Hub you're using. You can get this information from your Azure Stack Hub administrator.

Note

If you're using the Azure Stack Development Kit, and you have administrative access, see the [Determine the current version](#) section to determine the Azure Stack

Hub build.

For other APIs, run the following PowerShell command to output the namespaces, resource types, and API versions that are supported in your Azure Stack Hub subscription. There may still be differences at a property level. For this command to work, you must have already [installed](#) and [configured](#) PowerShell for an Azure Stack Hub environment. You must also have a subscription to an Azure Stack Hub offer.

Az modules

```
Get-AzResourceProvider | Select ProviderNamespace -Expand ResourceTypes  
| Select * -Expand ApiVersions | `  
Select ProviderNamespace, ResourceType, @{Name="ApiVersion";  
Expression={$_.}}
```

Example output (truncated):

ProviderNamespace	ResourceType	ApiVersion
Microsoft.Compute	availabilitySets	2016-03-30
Microsoft.Compute	locations	2016-03-30
Microsoft.Compute	locations	2015-06-15
Microsoft.Compute	locations/operations	2016-03-30
Microsoft.Compute	locations/operations	2015-06-15
Microsoft.Compute	locations/publishers	2016-03-30
Microsoft.Compute	locations/publishers	2015-06-15
Microsoft.Compute	locations/usages	2016-03-30
Microsoft.Compute	locations/usages	2015-06-15
Microsoft.Compute	locations/vmSizes	2016-03-30
Microsoft.Compute	operations	2015-06-15
Microsoft.Compute	operations	2016-03-30
Microsoft.Compute	virtualMachines	2016-03-30
Microsoft.Compute	virtualMachines	2015-06-15
Microsoft.Compute	virtualMachines/extensions	2016-03-30
Microsoft.Compute	virtualMachines/extensions	2015-06-15
Microsoft.Compute	virtualMachineScaleSets	2016-03-30
Microsoft.Compute	virtualMachineScaleSets	2015-06-15

Next steps

For more detailed information about differences at a service level, see:

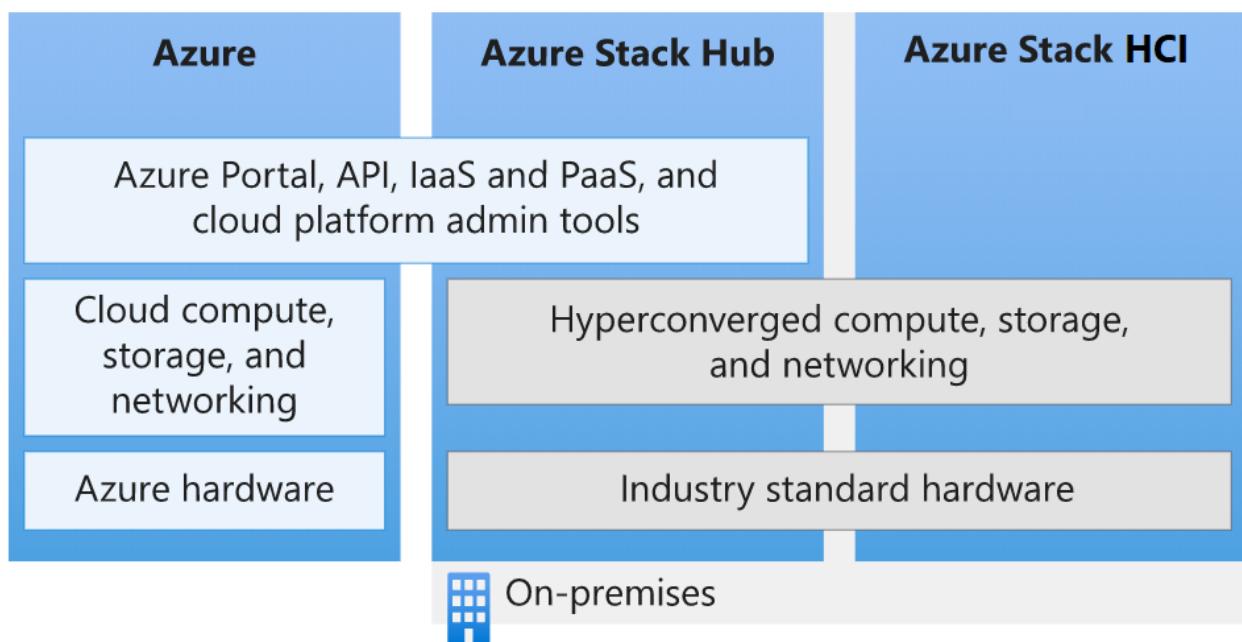
- [Considerations for VMs in Azure Stack Hub](#)
- [Considerations for Storage in Azure Stack Hub](#)
- [Considerations for Azure Stack Hub networking](#)
- [Considerations for Azure Stack Hub SQL resource provider](#)

Differences between global Azure, Azure Stack Hub, and Azure Stack HCI

Article • 10/24/2022

Microsoft provides Azure and the Azure Stack Hub family of services in one Azure ecosystem. Use the same application model, self-service portals, and APIs with Azure Resource Manager to deliver cloud-based capabilities whether your business uses global Azure or on-premises resources.

This article describes the differences between global Azure, Azure Stack Hub, and Azure Stack HCI capabilities. It provides common scenario recommendations to help you make the best choice for delivering Microsoft cloud-based services for your organization.



Global Azure

Microsoft Azure is an ever-expanding set of cloud services to help your organization meet your business challenges. It's the freedom to build, manage, and deploy apps on a massive, global network using your favorite tools and frameworks.

Global Azure offers more than 100 services available in 54 regions around the globe. For the most current list of global Azure services, see the [Products available by region](#). The services available in Azure are listed by category and also by whether they're generally available or available through preview.

For more information about global Azure services, see [Get started with Azure](#).

Azure Stack Hub

Azure Stack Hub is an extension of Azure that brings the agility and innovation of cloud computing to your on-premises environment. Deployed on-premises, Azure Stack Hub can be used to provide Azure consistent services either connected to the internet (and Azure) or in disconnected environments with no internet connectivity. Azure Stack Hub uses the same underlying technologies as global Azure, which includes the core components of Infrastructure-as-a-Service (IaaS), Software-as-a-Service (SaaS), and optional Platform-as-a-Service (PaaS) capabilities. These capabilities include:

- Azure VMs for Windows and Linux
- Azure Web Apps and Functions
- Azure Key Vault
- Azure Resource Manager
- Azure Marketplace
- Containers
- Admin tools (Plans, offers, RBAC, and so on)

The PaaS capabilities of Azure Stack Hub are optional because Azure Stack Hub isn't operated by Microsoft--it's operated by our customers. This means you can offer whatever PaaS service you want to end users if you're prepared to abstract the underlying infrastructure and processes away from the end user. However, Azure Stack Hub does include several optional PaaS service providers including App Service, SQL databases, and MySQL databases. These are delivered as resource providers so they're multi-tenant ready, updated over time with standard Azure Stack Hub updates, visible in the Azure Stack Hub portal, and well integrated with Azure Stack Hub.

In addition to the resource providers described above, there are additional PaaS services available and tested as [Azure Resource Manager template-based solutions](#) that run in IaaS. As an Azure Stack Hub operator, you can offer them as PaaS services to your users including:

- Service Fabric
- Kubernetes Container Service
- Ethereum Blockchain
- Cloud Foundry

Example use cases for Azure Stack Hub

- Financial modeling
- Clinical and claims data
- IoT device analytics

- Retail assortment optimization
- Supply-chain optimization
- Industrial IoT
- Predictive maintenance
- Smart city
- Citizen engagement

Learn more about Azure Stack Hub at [What is Azure Stack Hub](#).

Azure Stack HCI

[Azure Stack HCI](#) is a hyperconverged cluster that uses validated hardware to run virtualized Windows and Linux workloads on-premises and easily connect to Azure for cloud-based backup, recovery, and monitoring. Initially based on Windows Server 2019, Azure Stack HCI is now delivered as an Azure service with a subscription-based licensing model and hybrid capabilities built-in. Although Azure Stack HCI is based on the same core operating system components as Windows Server, it's an entirely new product line focused on being the best virtualization host.

Azure Stack HCI uses Microsoft-validated hardware from an OEM partner to ensure optimal performance and reliability. The solutions include support for technologies such as NVMe drives, persistent memory, and remote-direct memory access (RDMA) networking.

Example use cases for Azure Stack HCI

- Remote or branch office systems
- Datacenter consolidation
- Virtual desktop infrastructure
- Business-critical infrastructure
- Lower-cost storage
- High availability and disaster recovery in the cloud
- Virtualizing enterprise apps like SQL Server
- Run containers with [Azure Kubernetes Service \(AKS\)](#) on Azure Stack HCI
- Run Azure Arc enabled services such as [Azure data services](#), which includes SQL Managed Instance and PostgreSQL Hyperscale, and [Azure enabled application services \(preview\)](#), which includes App Service, Functions, Logic Apps, API Management, and Event Grid.

Visit the [Azure Stack HCI website](#) to view 70+ Azure Stack HCI solutions currently available from Microsoft partners.

Next steps

[Azure Stack Hub administration basics](#)

[Quickstart: use the Azure Stack Hub administration portal](#)

Azure Stack HCI solution overview

Article • 01/31/2024

Applies to: Azure Stack HCI, versions 23H2 and 22H2

Azure Stack HCI is a hyperconverged infrastructure (HCI) solution that hosts Windows and Linux VM or containerized workloads and their storage. It's a hybrid product that connects the on-premises system to Azure for cloud-based services, monitoring, and management.

Overview

An Azure Stack HCI system consists of a server or a cluster of servers running the Azure Stack HCI operating system and connected to Azure. You can use the Azure portal to monitor and manage individual Azure Stack HCI systems as well as view all of your Azure Stack HCI deployments. You can also manage with your existing tools, including Windows Admin Center and PowerShell.

Azure Stack HCI is available for download from the Azure portal with a free 60-day trial ([Download Azure Stack HCI](#)).

To acquire the servers to run Azure Stack HCI, you can purchase Azure Stack HCI integrated systems from a Microsoft hardware partner with the operating system pre-installed, or buy validated nodes and install the operating system yourself. See the [Azure Stack HCI Catalog](#) ↗ for hardware options and use the Azure Stack HCI sizing tool to estimate hardware requirements.

Azure Stack HCI features and architecture

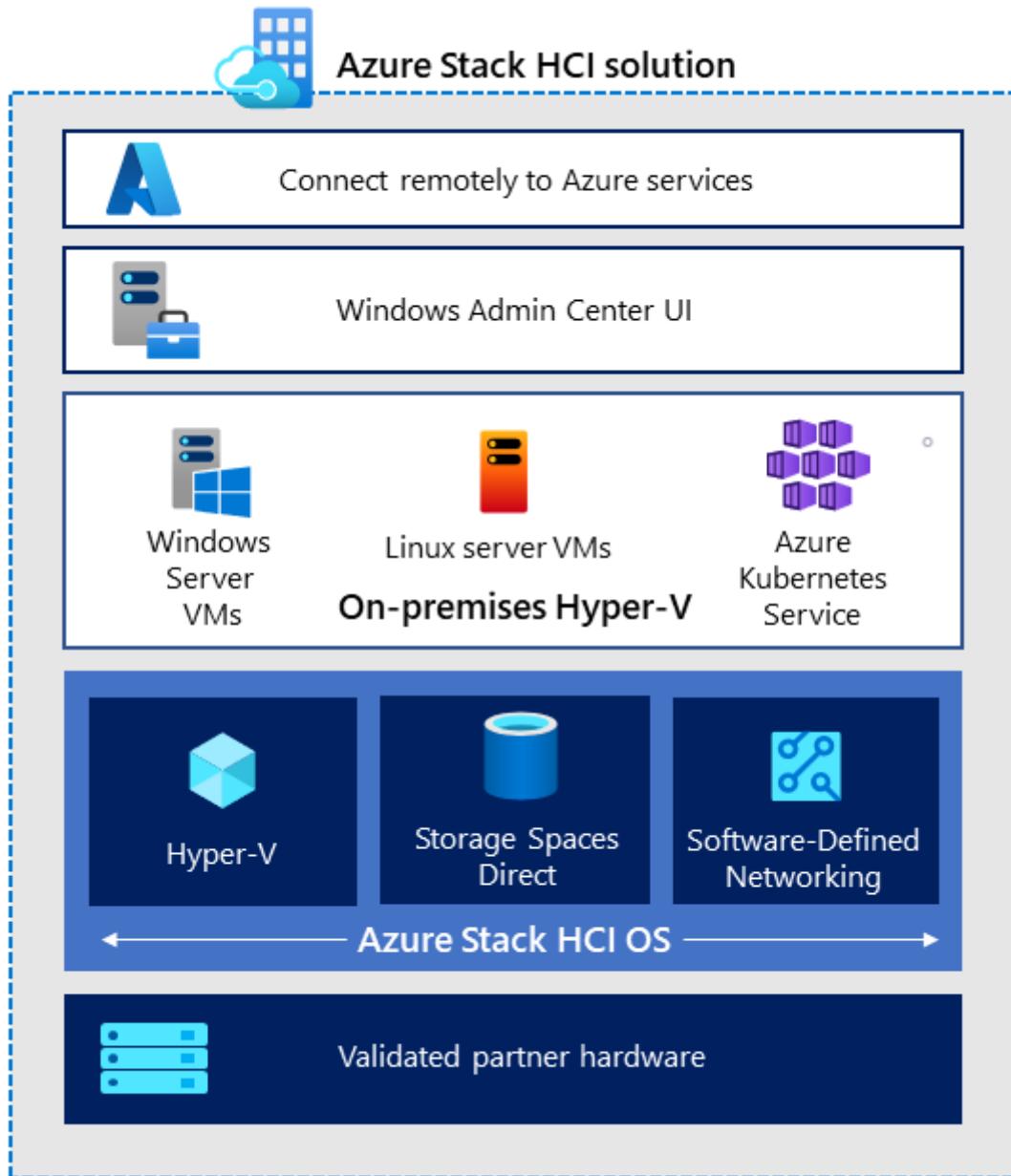
Azure Stack HCI is built on proven technologies including Hyper-V, Storage Spaces Direct, Azure Kubernetes Service (AKS), and Azure-inspired software defined networking (SDN).

Each Azure Stack HCI system consists of between 1 and 16 physical servers. All servers share common configurations and resources by leveraging the Windows Server Failover Clustering feature.

Azure Stack HCI combines the following:

- Azure Stack HCI operating system

- Validated hardware from a hardware partner
- Azure services including monitoring and management
- Windows Admin Center for management via Azure and on-premises
- Hyper-V-based compute resources
- Storage Spaces Direct-based virtualized storage
- SDN-based virtualized networking using Network Controller (optional)
- Azure Kubernetes Service (AKS) hybrid (optional)



See [What's new in Azure Stack HCI](#) for details on the latest enhancements.

Why Azure Stack HCI?

There are many reasons customers choose Azure Stack HCI, including:

- It provides industry-leading virtualization performance and value.

- You pay for the software monthly via an Azure subscription instead of when buying the hardware.
- It's familiar for Hyper-V and server admins, allowing them to leverage existing virtualization and storage concepts and skills.
- It can be monitored and managed from the Azure portal or using on-premises tools such as Microsoft System Center, Active Directory, Group Policy, and PowerShell scripting.
- It works with popular third-party backup, security, and monitoring tools.
- Flexible hardware choices allow customers to choose the vendor with the best service and support in their geography.
- Joint support between Microsoft and the hardware vendor improves the customer experience.
- Solution updates make it easy to keep the entire solution up-to-date.

Common use cases for Azure Stack HCI

Customers often choose Azure Stack HCI in the following scenarios.

[] Expand table

Use case	Description
Azure Virtual Desktop (AVD)	Azure Virtual Desktop for Azure Stack HCI lets you deploy Azure Virtual Desktop session hosts on your on-premises Azure Stack HCI infrastructure. You manage your session hosts from the Azure portal. To learn more, see Azure Virtual Desktop for Azure Stack HCI .
Azure Kubernetes Service (AKS) hybrid	You can leverage Azure Stack HCI to host container-based deployments, which increases workload density and resource usage efficiency. Azure Stack HCI also further enhances the agility and resiliency inherent to Azure Kubernetes deployments. Azure Stack HCI manages automatic failover of VMs serving as Kubernetes cluster nodes in case of a localized failure of the underlying physical components. This supplements the high availability built into Kubernetes, which automatically restarts failed containers on either the same or another VM. To learn more, see Azure Kubernetes Service on Azure Stack HCI and Windows Server .
Run Azure Arc services on-premises	Azure Arc allows you to run Azure services anywhere. This allows you to build consistent hybrid and multicloud application architectures by using Azure services that can run in Azure, on-premises, at the edge, or at other cloud providers. Azure Arc enabled services allow you to run Arc VMs, Azure data services and Azure application services such as Azure App Service, Functions, Logic Apps, Event Grid, and API Management anywhere to support hybrid workloads. To learn more, see Azure Arc overview .

Use case	Description
Highly performant SQL Server	Azure Stack HCI provides an additional layer of resiliency to highly available, mission-critical Always On availability groups-based deployments of SQL Server. This approach also offers extra benefits associated with the single-vendor approach, including simplified support and performance optimizations built into the underlying platform. To learn more, see Deploy SQL Server on Azure Stack HCI .
Trusted enterprise virtualization	Azure Stack HCI satisfies the trusted enterprise virtualization requirements through its built-in support for Virtualization-based Security (VBS). VBS relies on Hyper-V to implement the mechanism referred to as virtual secure mode, which forms a dedicated, isolated memory region within its guest VMs. By using programming techniques, it's possible to perform designated, security-sensitive operations in this dedicated memory region while blocking access to it from the host OS. This considerably limits potential vulnerability to kernel-based exploits. To learn more, see Deploy Trusted Enterprise Virtualization on Azure Stack HCI .
Scale-out storage	Storage Spaces Direct is a core technology of Azure Stack HCI that uses industry-standard servers with locally attached drives to offer high availability, performance, and scalability. Using Storage Spaces Direct results in significant cost reductions compared with competing offers based on storage area network (SAN) or network-attached storage (NAS) technologies. These benefits result from an innovative design and a wide range of enhancements, such as persistent read/write cache drives, mirror-accelerated parity, nested resiliency, and deduplication.
Disaster recovery for virtualized workloads	An Azure Stack HCI stretched cluster provides automatic failover of virtualized workloads to a secondary site following a primary site failure. Synchronous replication ensures crash consistency of VM disks.
Data center consolidation and modernization	Refreshing and consolidating aging virtualization hosts with Azure Stack HCI can improve scalability and make your environment easier to manage and secure. It's also an opportunity to retire legacy SAN storage to reduce footprint and total cost of ownership. Operations and systems administration are simplified with unified tools and interfaces and a single point of support.
Branch office and edge	For branch office and edge workloads, you can minimize infrastructure costs by deploying two-node clusters with inexpensive witness options, such as Cloud Witness or a USB drive-based file share witness. Another factor that contributes to the lower cost of two-node clusters is support for switchless networking, which relies on crossover cable between cluster nodes instead of more expensive high-speed switches. Customers can also centrally view remote Azure Stack HCI deployments in the Azure portal. To learn more, see Deploy branch office and edge on Azure Stack HCI .

Demo of using Microsoft Azure with Azure Stack HCI

For an end-to-end example of using Microsoft Azure to manage apps and infrastructure at the Edge using Azure Arc, Azure Kubernetes Service, and Azure Stack HCI, see the [Retail edge transformation with Azure hybrid demo](#).

Using a fictional customer, inspired directly by real customers, you will see how to deploy Kubernetes, set up GitOps, deploy VMs, use Azure Monitor and drill into a hardware failure, all without leaving the Azure portal.

<https://www.youtube-nocookie.com/embed/t81MNUjAnEQ>

This video includes preview functionality which shows real product functionality, but in a closely controlled environment.

Azure integration benefits

Azure Stack HCI allows you to take advantage of cloud and on-premises resources working together and natively monitor, secure, and back up to the cloud.

You can use the Azure portal for an increasing number of tasks including:

- **Monitoring:** View all of your Azure Stack HCI systems in a single, global view where you can group them by resource group and tag them.
- **Billing:** Pay for Azure Stack HCI through your Azure subscription.

You can also subscribe to additional Azure hybrid services.

For more details on the cloud service components of Azure Stack HCI, see [Azure Stack HCI hybrid capabilities with Azure services](#).

What you need for Azure Stack HCI

To get started, you'll need:

- One or more servers from the [Azure Stack HCI Catalog](#), purchased from your preferred Microsoft hardware partner.
- An [Azure subscription](#).
- Operating system licenses for your workload VMs – for example, Windows Server. See [Activate Windows Server VMs](#).
- An internet connection for each server in the cluster that can connect via HTTPS outbound traffic to well-known Azure endpoints at least every 30 days. See [Azure connectivity requirements](#) for more information.
- For clusters stretched across sites:
 - At least four servers (two in each site)

- At least one 1 Gb connection between sites (a 25 Gb RDMA connection is preferred)
- An average latency of 5 ms round trip between sites if you want to do synchronous replication where writes occur simultaneously in both sites.
- If you plan to use SDN, you'll need a virtual hard disk (VHD) for the Azure Stack HCI operating system to create Network Controller VMs (see [Plan to deploy Network Controller](#)).

Make sure your hardware meets the [System requirements](#) and that your network meets the [physical network](#) and [host network](#) requirements for Azure Stack HCI.

For Azure Kubernetes Service on Azure Stack HCI and Windows Server requirements, see [AKS requirements on Azure Stack HCI](#).

Azure Stack HCI is priced on a per core basis on your on-premises servers. For current pricing, see [Azure Stack HCI pricing](#).

Hardware and software partners

Microsoft recommends purchasing Integrated Systems built by our hardware partners and validated by Microsoft to provide the best experience running Azure Stack HCI. You can also run Azure Stack HCI on Validated Nodes, which offer a basic building block for HCI systems to give customers more hardware choices. Microsoft partners also offer a single point of contact for implementation and support services.

Browse the [Azure Stack HCI Catalog](#) to view Azure Stack HCI solutions from Microsoft partners such as ASUS, Blue Chip, DataON, Dell EMC, Fujitsu, HPE, Hitachi, Lenovo, NEC, primeLine Solutions, QCT, SecureGUARD, and Supermicro.

Some Microsoft partners are developing software that extends the capabilities of Azure Stack HCI while allowing IT admins to use familiar tools. To learn more, see [Utility applications for Azure Stack HCI](#).

Next steps

- Learn more about [Azure Stack HCI, version 23H2 deployment](#).

Compare Azure Stack HCI to Windows Server

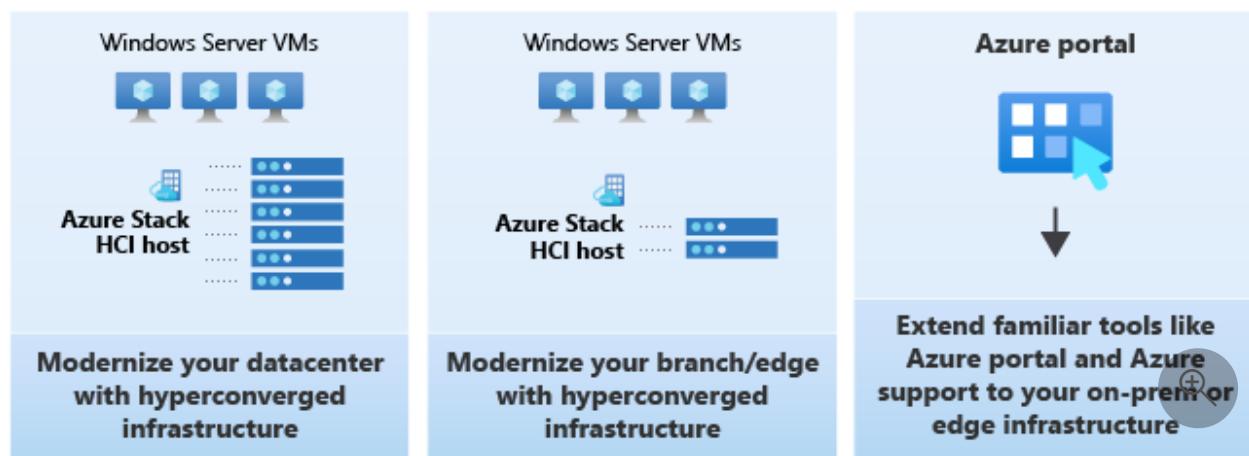
Article • 01/20/2023

Applies to: Azure Stack HCI, versions 22H2 and 21H2; Windows Server 2022

This article explains key differences between Azure Stack HCI and Windows Server and provides guidance about when to use each. Both products are actively supported and maintained by Microsoft. Many organizations choose to deploy both as they are intended for different and complementary purposes.

When to use Azure Stack HCI

Azure Stack HCI is Microsoft's premier hyperconverged infrastructure platform for running VMs or virtual desktops on-premises with connections to Azure hybrid services. Azure Stack HCI can help to modernize and secure your datacenters and branch offices, and achieve industry-best performance with low latency and data sovereignty.



Use Azure Stack HCI for:

- The best virtualization host to modernize your infrastructure, either for existing workloads in your core datacenter or emerging requirements for branch office and edge locations.
- Easy extensibility to the cloud, with a regular stream of innovations from your Azure subscription and a consistent set of tools and experiences.
- All the benefits of hyperconverged infrastructure: a simpler, more consolidated datacenter architecture with high-speed storage and networking.

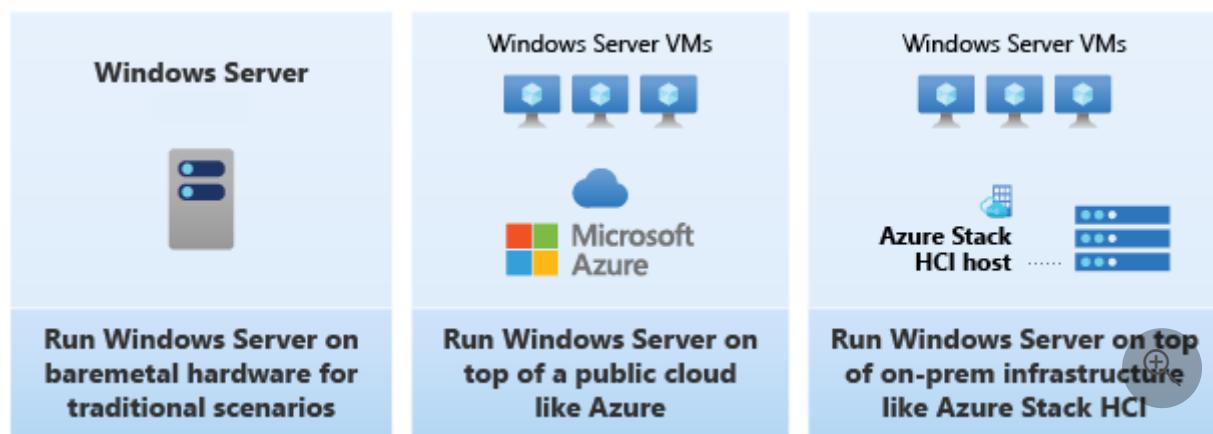
(!) Note

When using Azure Stack HCI, run all of your workloads inside virtual machines or containers, not directly on the cluster. Azure Stack HCI isn't licensed for clients to connect directly to it using Client Access Licenses (CALs).

For information about licensing Windows Server VMs running on an Azure Stack HCI cluster, see [Activate Windows Server VMs](#).

When to use Windows Server

Windows Server is a highly versatile, multi-purpose operating system with dozens of roles and hundreds of features and includes the right for clients to connect directly with appropriate CALs. Windows Server machines can be in the cloud or on-premises, including virtualized on top of Azure Stack HCI.



Use Windows Server for:

- A guest operating system inside of virtual machines (VMs) or containers
- As the runtime server for a Windows application
- To use one or more of the built-in server roles such as Active Directory, file services, DNS, DHCP, or Internet Information Services (IIS)
- As a traditional server, such as a bare-metal domain controller or SQL Server installation
- For traditional infrastructure, such as VMs connected to Fibre Channel SAN storage

Compare product positioning

The following table shows the high-level product packaging for Azure Stack HCI and Windows Server.

Attribute	Azure Stack HCI	Windows Server
Product type	Cloud service that includes an operating system and more	Operating system
Legal	Covered under your Microsoft customer agreement or online subscription agreement	Has its own end-user license agreement
Licensing	Billed to your Azure subscription	Has its own paid license
Support	Covered under Azure support	Can be covered by different support agreements, including Microsoft Premier Support
Where to get it	Download from the Azure portal or comes preinstalled on integrated systems	Microsoft Volume Licensing Service Center or Evaluation Center
Runs in VMs	For evaluation only; intended as a host operating system	Yes, in the cloud or on premises
Hardware	Runs on any of more than 200 pre-validated solutions from the Azure Stack HCI Catalog	Runs on any hardware with the "Certified for Windows Server" logo. See the WindowsServerCatalog
Sizing	Azure Stack HCI sizing tool	None
Lifecycle policy	Always up to date with the latest features. You have up to six months to install updates.	Use this option of the Windows Server servicing channels : Long-Term Servicing Channel (LTSC)

Compare workloads and benefits

The following table compares the workloads and benefits of Azure Stack HCI and Windows Server.

Attribute	Azure Stack HCI	Windows Server
Azure Kubernetes Service (AKS)	Yes	Yes
Azure Arc-Enabled PaaS Services	Yes	Yes
Windows Server 2022 Azure Edition	Yes	No
Windows Server subscription add-on (Dec. 2021)	Yes	No
Free Extended Security Updates (ESUs) for Windows Server and	Yes	No ¹

Attribute	Azure Stack HCI	Windows Server
SQL 2008/R2 and 2012/R2		

¹ Requires purchasing an Extended Security Updates (ESU) license key and manually applying it to every VM.

Compare technical features

The following table compares the technical features of Azure Stack HCI and Windows Server 2022.

Attribute	Azure Stack HCI	Windows Server 2022
Hyper-V	Yes	Yes
Storage Spaces Direct	Yes	Yes
Software-Defined Networking	Yes	Yes
Adjustable storage repair speed	Yes	Yes
Secured-core Server	Yes	Yes
Stronger, faster network encryption	Yes	Yes
4-5x faster Storage Spaces repairs	Yes	Yes
Stretch clustering for disaster recovery with Storage Spaces Direct	Yes	No
High availability for GPU workload	Yes	No
Restart up to 10x faster with kernel-only restarts	Yes	No
Simplified host networking with Network ATC	Yes	No
Storage Spaces Direct on a single server	Yes	No
Storage Spaces Direct thin provisioning	Yes	No
Dynamic processor compatibility mode	Yes	No
Cluster-Aware OS feature update	Yes	No
Integrated driver and firmware updates	Yes (Integrated Systems only)	No

For more information, see [What's New in Azure Stack HCI, version 22H2](#) and [Using Azure Stack HCI on a single server](#).

Compare management options

The following table compares the management options for Azure Stack HCI and Windows Server. Both products are designed for remote management and can be managed with many of the same tools.

Attribute	Azure Stack HCI	Windows Server
Windows Admin Center	Yes	Yes
Microsoft System Center	Yes (sold separately)	Yes (sold separately)
Third-party tools	Yes	Yes
Azure Backup and Azure Site Recovery support	Yes	Yes
Azure portal	Yes (natively)	Requires Azure Arc agent
Azure portal > Extensions and Arc-enabled host	Yes	Manual ¹
Azure portal > Windows Admin Center integration (preview)	Yes	Azure VMs only ¹
Azure portal > Multi-cluster monitoring for Azure Stack HCI (preview)	Yes	No
Azure portal > Azure Resource Manager integration for clusters	Yes	No
Azure portal > Arc VM management (preview)	Yes	No
Desktop experience	No	Yes

¹ Requires manually installing the Arc-git statusConnected Machine agent on every machine.

Compare product pricing

The table below compares the product pricing for Azure Stack HCI and Windows Server. For details, see [Azure Stack HCI pricing](#).

Attribute	Azure Stack HCI	Windows Server
Price type	Subscription service	Varies: most often a one-time license
Price structure	Per core, per month	Varies: usually per core
Price	Per core, per month	See Pricing and licensing for Windows Server 2022
Evaluation/trial period	60-day free trial once registered	180-day evaluation copy
Channels	Enterprise agreement, cloud service provider, or direct	Enterprise agreement/volume licensing, OEM, services provider license agreement (SPLA)

Next steps

- [Compare Azure Stack HCI to Azure Stack Hub](#)

Choose drives for Azure Stack HCI and Windows Server clusters

Article • 04/18/2023

Applies to: Azure Stack HCI, versions 22H2 and 21H2; Windows Server 2022, Windows Server 2019

This article provides guidance on how to choose drives to meet your performance and capacity requirements.

Drive types

Storage Spaces Direct, the underlying storage virtualization technology behind Azure Stack HCI and Windows Server currently works with four types of drives:

Type of drive	Description
 PMEM	PMem refers to persistent memory, a new type of low latency, high-performance storage.
 NVMe	NVMe (Non-Volatile Memory Express) refers to solid-state drives that sit directly on the PCIe bus. Common form factors are 2.5" U.2, PCIe Add-In-Card (AIC), and M.2. NVMe offers higher IOPS and IO throughput with lower latency than any other type of drive we support today except PMem.
 SSD	SSD refers to solid-state drives, which connect via conventional SATA or SAS.
 HDD	HDD refers to rotational, magnetic hard disk drives, which offer vast storage capacity.

Note

This article covers choosing drive configurations with NVMe, SSD, and HDD. For more information on PMem, see [Understand and deploy persistent memory](#).

ⓘ Note

Storage Bus Layer (SBL) cache is not supported in single server configuration. All flat single storage type configurations (for example all-NVMe or all-SSD) is the only supported storage type for single server.

Built-in cache

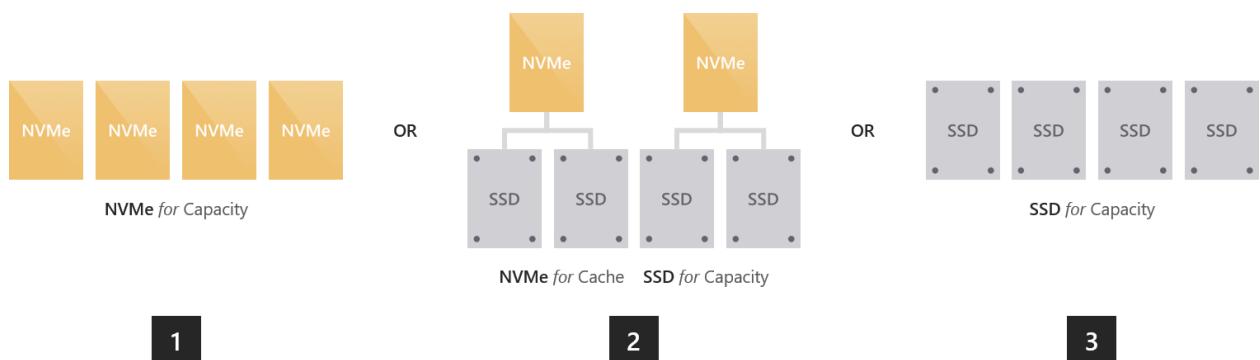
Storage Spaces Direct features a built-in server-side cache. It is a large, persistent, real-time read and write cache. In deployments with multiple types of drives, it is configured automatically to use all drives of the "fastest" type. The remaining drives are used for capacity.

For more information, check out [Understanding the storage pool cache](#).

Option 1 – Maximizing performance

To achieve predictable and uniform submillisecond latency across random reads and writes to any data, or to achieve extremely high IOPS (we've done [over 13 million ↗!](#)) or IO throughput (we've done over 500 GB/sec reads), you should go "all-flash."

There are multiple ways to do so:



- 1. All NVMe.** Using all NVMe provides unmatched performance, including the most predictable low latency. If all your drives are the same model, there is no cache. You can also mix higher-endurance and lower-endurance NVMe models, and configure the former to cache writes for the latter ([requires set-up](#)).
- 2. NVMe + SSD.** Using NVMe together with SSDs, the NVMe will automatically cache writes to the SSDs. This allows writes to coalesce in cache and be de-staged only as needed, to reduce wear on the SSDs. This provides NVMe-like write characteristics, while reads are served directly from the also-fast SSDs.

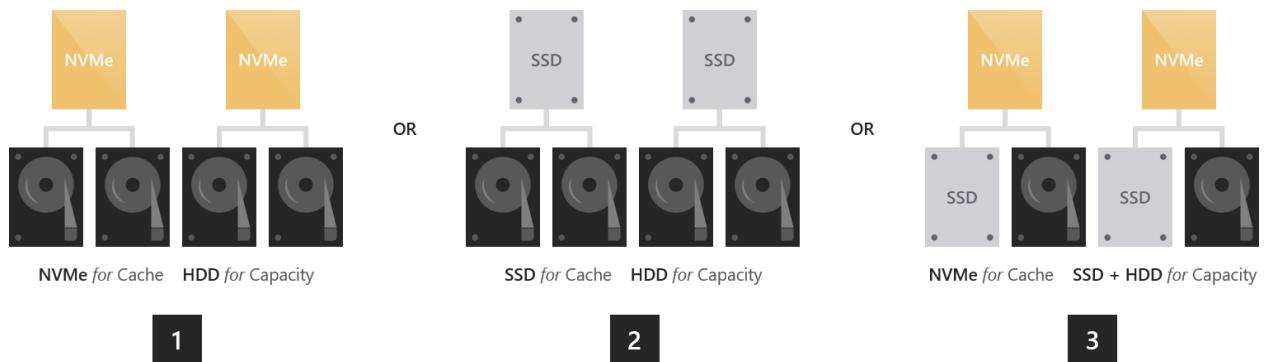
3. All SSD. As with All-NVMe, there is no cache if all your drives are the same model. If you mix higher-endurance and lower-endurance models, you can configure the former to cache writes for the latter ([requires set-up](#)).

! Note

An advantage to using all-NVMe or all-SSD with no cache is that you get usable storage capacity from every drive. There is no capacity "spent" on caching, which may be appealing at smaller scale.

Option 2 – Balancing performance and capacity

For environments with a variety of applications and workloads, some with stringent performance requirements and others requiring considerable storage capacity, you should go "hybrid" with either NVMe or SSDs caching for larger HDDs.



1. NVMe + HDD.

The NVMe drives will accelerate reads and writes by caching both. Caching reads allows the HDDs to focus on writes. Caching writes absorbs bursts and allows writes to coalesce and be de-staged only as needed, in an artificially serialized manner that maximizes HDD IOPS and IO throughput. This provides NVMe-like write characteristics, and for frequently or recently read data, NVMe-like read characteristics too.

2. SSD + HDD.

Similar to the above, the SSDs will accelerate reads and writes by caching both. This provides SSD-like write characteristics, and SSD-like read characteristics for frequently or recently read data.

There is one additional, rather exotic option: to use drives of *all three* types.

3. NVMe + SSD + HDD.

With drives of all three types, the NVMe drives cache for both the SSDs and HDDs. The appeal is that you can create volumes on the SSDs, and volumes on the HDDs, side by side in the same cluster, all accelerated by NVMe. The former are exactly as in an "all-flash" deployment, and the latter are

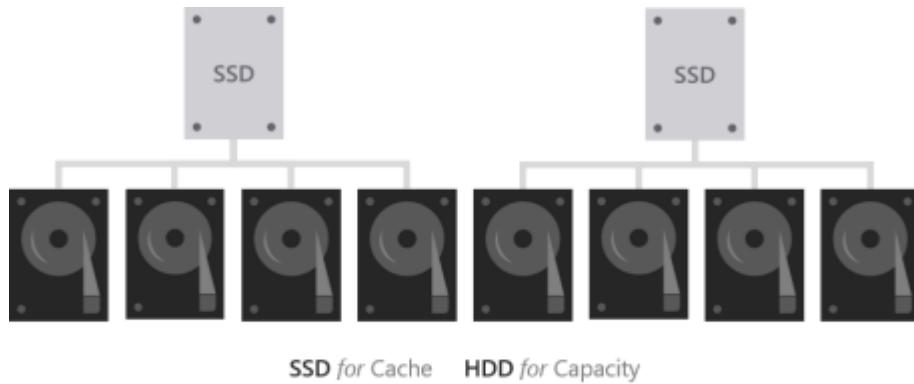
exactly as in the "hybrid" deployments described above. This is conceptually like having two pools, with largely independent capacity management, failure and repair cycles, and so on.

ⓘ Important

We recommend using the SSD tier to place your most performance-sensitive workloads on all-flash.

Option 3 – Maximizing capacity

For workloads that require vast capacity and write infrequently, such as archival, backup targets, data warehouses or "cold" storage, you should combine a few SSDs for caching with many larger HDDs for capacity.



1. **SSD + HDD.** The SSDs will cache reads and writes, to absorb bursts and provide SSD-like write performance, with optimized de-staging later to the HDDs.

ⓘ Important

Configuration with HDDs only is not supported. High endurance SSDs caching to low endurance SSDs is not advised.

Sizing considerations

Cache

Every server must have at least two cache drives (the minimum required for redundancy). We recommend making the number of capacity drives a multiple of the number of cache drives. For example, if you have 4 cache drives, you will experience more consistent performance with 8 capacity drives (1:2 ratio) than with 7 or 9.

The cache should be sized to accommodate the working set of your applications and workloads, that is, all the data they are actively reading and writing at any given time. There is no cache size requirement beyond that. For deployments with HDDs, a fair starting place is 10 percent of capacity – for example, if each server has $4 \times 4\text{ TB HDD} = 16\text{ TB}$ of capacity, then $2 \times 800\text{ GB SSD} = 1.6\text{ TB}$ of cache per server. For all-flash deployments, especially with very [high endurance](#) SSDs, it may be fair to start closer to 5 percent of capacity – for example, if each server has $24 \times 1.2\text{ TB SSD} = 28.8\text{ TB}$ of capacity, then $2 \times 750\text{ GB NVMe} = 1.5\text{ TB}$ of cache per server. You can always add or remove cache drives later to adjust.

General

We recommend limiting the total storage capacity per server to approximately 400 terabytes (TB). The more storage capacity per server, the longer the time required to resync data after downtime or rebooting, such when applying software updates. The current maximum size per storage pool is 4 petabytes (PB) (4,000 TB) (1 PB for Windows Server 2016).

Next steps

For more information, see also:

- [Understand the storage pool cache](#)
- [System requirements](#)
- [Drive symmetry considerations](#)
- [Plan volumes](#)
- [Fault tolerance and storage efficiency](#)
- [Understand and deploy persistent memory](#)

Storage options for FSLogix profile containers in Azure Virtual Desktop

Article • 03/12/2023

Azure offers multiple storage solutions that you can use to store your FSLogix profile container. This article compares storage solutions that Azure offers for Azure Virtual Desktop FSLogix user profile containers. We recommend storing FSLogix profile containers on Azure Files for most of our customers.

Azure Virtual Desktop offers FSLogix profile containers as the recommended user profile solution. FSLogix is designed to roam profiles in remote computing environments, such as Azure Virtual Desktop. At sign-in, this container is dynamically attached to the computing environment using a natively supported Virtual Hard Disk (VHD) and a Hyper-V Virtual Hard Disk (VHDX). The user profile is immediately available and appears in the system exactly like a native user profile.

The following tables compare the storage solutions Azure Storage offers for Azure Virtual Desktop FSLogix profile container user profiles.

Azure platform details

Features	Azure Files	Azure NetApp Files	Storage Spaces Direct
Use case	General purpose	General purpose to enterprise scale	Cross-platform
Platform service	Yes, Azure-native solution	Yes, Azure-native solution	No, self-managed
Regional availability	All regions	Select regions 	All regions
Redundancy	Locally redundant/zone-redundant/geo-redundant/geo-zone-redundant	Locally redundant/zone-redundant with cross-zone replication /geo-redundant with cross-region replication	Locally redundant/zone-redundant/geo-redundant

Features	Azure Files	Azure NetApp Files	Storage Spaces Direct
Tiers and performance	Standard (Transaction optimized) Premium Up to max 100K IOPS per share with 10 GBps per share at about 3-ms latency	Standard Premium Ultra Up to max 460K IOPS per volume with 4.5 GBps per volume at about 1 ms latency. For IOPS and performance details, see Azure NetApp Files performance considerations and the FAQ .	Standard HDD: up to 500 IOPS per-disk limits Standard SSD: up to 4k IOPS per-disk limits Premium SSD: up to 20k IOPS per-disk limits We recommend Premium disks for Storage Spaces Direct
Capacity	100 TiB per share, Up to 5 PiB per general purpose account	100 TiB per volume, up to 12.5 PiB per NetApp account	Maximum 32 TiB per disk
Required infrastructure	Minimum share size 1 GiB	Minimum capacity pool 2 TiB, min volume size 100 GiB	Two VMs on Azure IaaS (+ Cloud Witness) or at least three VMs without and costs for disks
Protocols	SMB 3.0/2.1, NFSv4.1 (preview), REST	NFSv3, NFSv4.1, SMB 3.x/2.x, dual-protocol	NFSv3, NFSv4.1, SMB 3.1

Azure management details

Features	Azure Files	Azure NetApp Files	Storage Spaces Direct
Access	Cloud, on-premises and hybrid (Azure file sync)	Cloud, on-premises	Cloud, on-premises
Backup	Azure backup snapshot integration	Azure NetApp Files snapshots Azure NetApp Files backup	Azure backup snapshot integration
Security and compliance	All Azure supported certificates	Azure supported certificates	All Azure supported certificates

Features	Azure Files	Azure NetApp Files	Storage Spaces Direct
Azure Active Directory integration	Native Active Directory and Azure Active Directory Domain Services	Azure Active Directory Domain Services and Native Active Directory	Native Active Directory or Azure Active Directory Domain Services support only

Once you've chosen your storage method, check out [Azure Virtual Desktop pricing ↗](#) for information about our pricing plans.

Azure Files tiers

Azure Files offers two different tiers of storage: premium and standard. These tiers let you tailor the performance and cost of your file shares to meet your scenario's requirements.

- Premium file shares are backed by solid-state drives (SSDs) and are deployed in the FileStorage storage account type. Premium file shares provide consistent high performance and low latency for input and output (IO) intensive workloads. Premium file shares use a provisioned billing model, where you pay for the amount of storage you would like your file share to have, regardless of how much you use.
- Standard file shares are backed by hard disk drives (HDDs) and are deployed in the general purpose version 2 (GPv2) storage account type. Standard file shares provide reliable performance for IO workloads that are less sensitive to performance variability, such as general-purpose file shares and dev/test environments. Standard file shares use a pay-as-you-go billing model, where you pay based on storage usage, including data stored and transactions.

To learn more about how billing works in Azure Files, see [Understand Azure Files billing](#).

The following table lists our recommendations for which performance tier to use based on your workload. These recommendations will help you select the performance tier that meets your performance targets, budget, and regional considerations. We've based these recommendations on the example scenarios from [Remote Desktop workload types](#).

Workload type	Recommended file tier
Light (fewer than 200 users)	Standard file shares
Light (more than 200 users)	Premium file shares or standard with multiple file shares
Medium	Premium file shares

Workload type	Recommended file tier
Heavy	Premium file shares
Power	Premium file shares

For more information about Azure Files performance, see [File share and file scale targets](#). For more information about pricing, see [Azure Files pricing ↗](#).

Azure NetApp Files tiers

Azure NetApp Files volumes are organized in capacity pools. Volume performance is defined by the service level of the hosting capacity pool. Three performance levels are offered, ultra, premium and standard. For more information, see [Storage hierarchy of Azure NetApp Files](#). Azure NetApp Files performance is [a function of tier times capacity](#). More provisioned capacity leads to higher performance budget, which likely results in a lower tier requirement, providing a more optimal TCO.

The following table lists our recommendations for which performance tier to use based on workload defaults.

Workload	Example Users	Azure NetApp Files
Light	Users doing basic data entry tasks	Standard tier
Medium	Consultants and market researchers	Premium tier: small-medium user count Standard tier: large user count
Heavy	Software engineers, content creators	Premium tier: small-medium user count Standard tier: large user count
Power	Graphic designers, 3D model makers, machines learning researchers	Ultra tier: small user count Premium tier: medium user count Standard tier: large user count

In order to provision the optimal tier and volume size, consider using [this calculator ↗](#) for guidance.

Next steps

To learn more about FSLogix profile containers, user profile disks, and other user profile technologies, see the table in [FSLogix profile containers and Azure Files](#).

If you're ready to create your own FSLogix profile containers, get started with one of these tutorials:

- [Set up FSLogix Profile Container with Azure Files and Active Directory](#)
- [Set up FSLogix Profile Container with Azure NetApp Files](#)

Confidential computing deployment models

Article • 07/02/2023

Azure confidential computing supports multiple deployment models. These different models support the wide variety of customer security requirements for modern cloud computing.

Infrastructure as a Service (IaaS)

Under Infrastructure as a Service (IaaS) deployment model, you can use confidential virtual machines (VMs) in confidential computing. You can use VMs based on [AMD Secure Encrypted Virtualization Secure Nested Paging \(SEV-SNP\)](#), [Intel Trust Domain Extensions \(TDX\)](#) or [Intel Software Guard Extensions \(SGX\)](#) application enclaves.

Infrastructure as a Service (IaaS) is a cloud computing deployment model that grants access to scalable computing resources, such as servers, storage, networking, and virtualization, on demand. By adopting IaaS deployment model, organizations can forego the process of procuring, configuring, and managing their own infrastructure, instead only paying for the resources they utilize. This makes it a cost-effective solution.

In the domain of cloud computing, IaaS deployment model enables businesses to rent individual services from cloud service providers like Azure. Azure assumes responsibility for managing and maintaining the infrastructure, empowering organizations to concentrate on installing, configuring, and managing their software. Azure also offers supplementary services such as comprehensive billing management, logging, monitoring, storage resiliency, and security.

Scalability constitutes another significant advantage of IaaS deployment model in cloud computing. Enterprises can swiftly scale their resources up and down according to their requirements. This flexibility facilitates faster development life cycles, accelerating time to market for new products and ideas. Additionally, IaaS deployment model ensures reliability by eliminating single points of failure. Even in the event of a hardware component failure, the service remains accessible.

In conclusion, IaaS deployment model in combination with Azure Confidential Computing offers benefits, including cost savings, increased efficiency, innovation opportunities, reliability, high scalability, and all secured by a robust and comprehensive security solution designed specifically to protect highly sensitive data.

Platform as a Service (PaaS)

For Platform as a Service (PaaS), you can use [confidential containers](#) in confidential computing. This offering includes enclave-aware containers in Azure Kubernetes Service (AKS).

Choosing the right deployment model depends on many factors. You might need to consider the existence of legacy applications, operating system capabilities, and migration from on-premises networks.

While there are still many reasons to use VMs, containers provide extra flexibility for the many software environments of modern IT.

Containers can support apps that:

- Run on multiple clouds.
- Connect to microservices.
- Use various programming languages and frameworks.
- Use automation and Azure Pipelines, including continuous integration and continuous deployment (CI/CD) implementation.

Containers also increase portability of applications, and improve resource usage, by applying the elasticity of the Azure cloud.

Normally, you might deploy your solution on confidential VMs if:

- You've got legacy applications that cannot be modified or containerized. However, you still need to introduce protection of data in memory, while the data is being processed.
- You're running multiple applications requiring different operating systems (OS) on a single piece of infrastructure.
- You want to emulate an entire computing environment, including all OS resources.
- You're migrating your existing VMs from on-premises to Azure.

You might opt for a confidential container-based approach when:

- You're concerned about cost and resource allocation. However, you need a more agile platform for deployment of your proprietary apps and datasets.
- You're building a modern cloud-native solution. You also have full control of source code and the deployment process.
- You need multi-cloud support.

Both options offer the highest security level for Azure services.

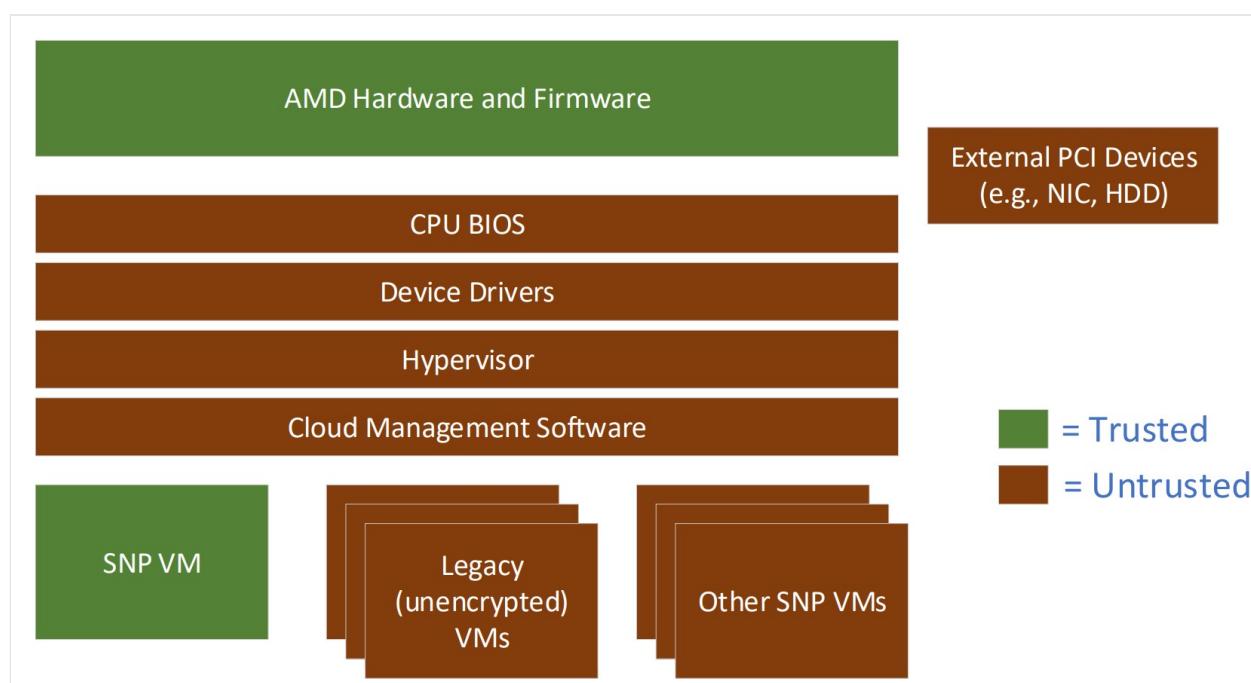
There are some differences in the security postures of [confidential VMs](#) and [confidential containers](#) as follows.

Confidential VMs on AMD SEV-SNP

Confidential VMs on AMD SEV-SNP offer hardware-encrypted protection of the entire VM from unauthorized access by the host administrator. This level typically includes the hypervisor, which the cloud service provider (CSP) manages. You can use this type of confidential VM to prevent the CSP accessing data and code executed within the VM.

VM admins or any other app or service running inside the VM, operate beyond the protected boundaries. These users and services can access data and code within the VM.

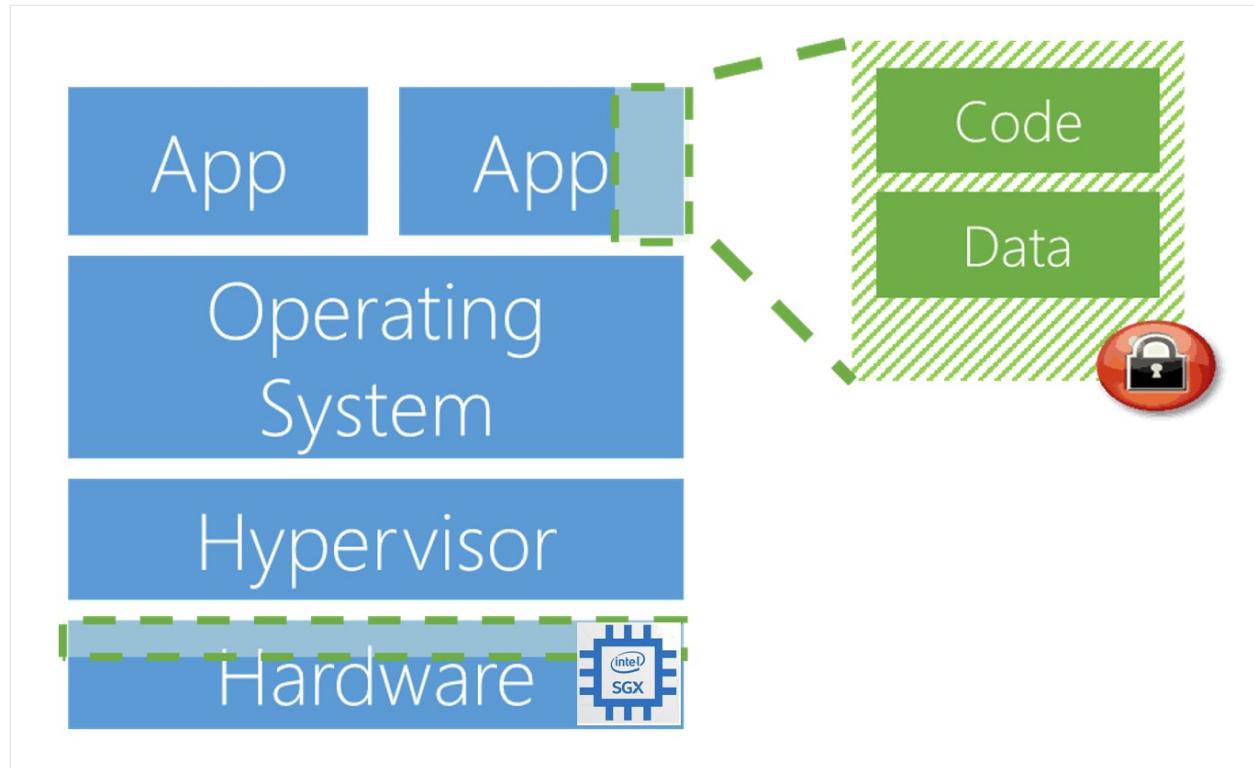
AMD SEV-SNP technology provides VM isolation from the hypervisor. The hardware-based memory integrity protection helps prevent malicious hypervisor-based attacks. The SEV-SNP model trusts the AMD Secure Processor and the VM. The model doesn't trust any other hardware and software components. Untrusted components include the BIOS, and the hypervisor on the host system.



Secure enclaves on Intel SGX

Secure enclaves on Intel SGX protect memory spaces inside a VM with hardware-based encryption. The security boundary of application enclaves is more restricted than confidential VMs on AMD SEV-SNP. For Intel SGX, the security boundary applies to portions of memory within a VM. Users, apps, and services running inside the Intel SGX-powered VM can't access any data and code in execution inside the enclave.

Intel SGX helps protect data in use by application isolation. By protecting selected code and data from modification, developers can partition their application into hardened enclaves or trusted execution modules to help increase application security. Entities outside the enclave can't read or write the enclave memory, whatever their permissions levels. The hypervisor or the operating system also can't obtain this access through normal OS-level calls. To call an enclave function, you have to use a new set of instructions in the Intel SGX CPUs. This process includes several protection checks.



Next steps

- Learn about confidential containers
- Learn about confidential computing enclaves

Preread Recommendations

Article • 11/15/2023

This document is designed to help you guide through the process of selecting a container offering on Azure Confidential Computing that best suits your workload requirements and security posture. To make the most of the guide, we recommend the following prereads.

Azure Compute Decision Matrix

Familiarize yourself with the overall [Azure Compute offerings](#) to understand the broader context in which Azure Confidential Computing operates.

Introduction to Azure Confidential Computing

Azure Confidential Computing offers solutions to enable isolation of your sensitive data while it's being processed in the cloud. You can read more about confidential computing [Azure confidential computing](#).

Attestation

Attestation is a process that provides assurances regarding the integrity and identity of the hardware and software environments in which applications run. In Confidential Computing, attestation allows you to verify that your applications are running on trusted hardware and in a trusted execution environment.

Learn more about attestation and Microsoft Azure Attestation service at [Attestation in Azure](#)

Definition of memory isolation

In confidential computing, memory isolation is a critical feature that safeguards data during processing. The Confidential Computing Consortium defines memory isolation as:

"Memory isolation is the ability to prevent unauthorized access to data in memory, even if the attacker has compromised the operating system or other privileged software. This is achieved by using hardware-based features to create a secure and isolated environment for confidential workload."

Choosing a Container offering on Azure Confidential Computing

Azure Confidential Computing offers various solutions for container deployment and management, each tailored for different levels of isolation and attestation capabilities.

Your current setup and operational needs dictate the most relevant path through this document. If you're already utilizing Azure Kubernetes Service (AKS) or have dependencies on Kubernetes APIs, we recommend following the AKS paths. On the other hand, if you're transitioning from a Virtual Machine setup and are interested in exploring serverless containers, the ACI (Azure Container Instances) path should be of interest.

Azure Kubernetes Service (AKS)

Confidential VM Worker Nodes

- **Guest Attestation:** Ability to verify that you're operating on a confidential virtual machine provided by Azure.
- **Memory Isolation:** VM level isolation with unique memory encryption key per VM.
- **Programming model:** Zero to minimal changes for containerized applications. Support is limited to containers that are Linux based (containers using a Linux base image for the container).

You can find more information on [Getting started with CVM worker nodes with a lift and shift workload to CVM node pool](#).

Confidential Containers on AKS

- **Full Guest Attestation:** Enables attestation of the full confidential computing environment including the workload.
- **Memory Isolation:** Node level isolation with a unique memory encryption key per VM.
- **Programming model:** Zero to minimal changes for containerized applications (containers using a Linux base image for the container).
- **Ideal Workloads:** Applications with sensitive data processing, multi-party computations, and regulatory compliance requirements.

You can find more information on [Getting started with CVM worker nodes with a lift and shift workload to CVM node pool](#).

Confidential Computing Nodes with Intel SGX

- **Application enclave Attestation:** Enables attestation of the container running, in scenarios where the VM isn't trusted, but only the application is trusted, ensuring a heightened level of security and trust in the application's execution environment.
- **Isolation:** Process level isolation.
- **Programming model:** Requires the use of open-source library OS or vendor solutions to run existing containerized applications. Support is limited to containers that are Linux based (containers using a Linux base image for the container).
- **Ideal Workloads:** High-security applications such as key management systems.

You can find more information about the offering and our partner solutions [here](#).

Serverless

Confidential Containers on Azure Container Instances (ACI)

- **Full Guest Attestation:** Enables attestation of the full confidential computing environment including the workload.
- **Isolation:** Container group level isolation with a unique memory encryption key per container group.
- **Programming model:** Zero to minimal changes for containerized applications. Support is limited to containers that are Linux based (containers using a Linux base image for the container).
- **Ideal Workloads:** Rapid development and deployment of simple containerized workloads without orchestration. Support for bursting from AKS using Virtual Nodes.

You can find more details at [Getting started with Confidential Containers on ACI](#).

Learn more

| [Intel SGX Confidential Virtual Machines on Azure](#) [Confidential Containers on Azure](#)

Compare self-managed Active Directory Domain Services, Microsoft Entra ID, and managed Microsoft Entra Domain Services

Article • 10/06/2023

To provide applications, services, or devices access to a central identity, there are three common ways to use Active Directory-based services in Azure. This choice in identity solutions gives you the flexibility to use the most appropriate directory for your organization's needs. For example, if you mostly manage cloud-only users that run mobile devices, it may not make sense to build and run your own Active Directory Domain Services (AD DS) identity solution. Instead, you could just use Microsoft Entra ID.

Although the three Active Directory-based identity solutions share a common name and technology, they're designed to provide services that meet different customer demands. At high level, these identity solutions and feature sets are:

- **Active Directory Domain Services (AD DS)** - Enterprise-ready lightweight directory access protocol (LDAP) server that provides key features such as identity and authentication, computer object management, group policy, and trusts.
 - AD DS is a central component in many organizations with an on-premises IT environment, and provides core user account authentication and computer management features.
 - For more information, see [Active Directory Domain Services overview in the Windows Server documentation](#).
- **Microsoft Entra ID** - Cloud-based identity and mobile device management that provides user account and authentication services for resources such as Microsoft 365, the Microsoft Entra admin center, or SaaS applications.
 - Microsoft Entra ID can be synchronized with an on-premises AD DS environment to provide a single identity to users that works natively in the cloud.
 - For more information about Microsoft Entra ID, see [What is Microsoft Entra ID?](#)
- **Microsoft Entra Domain Services** - Provides managed domain services with a subset of fully compatible traditional AD DS features such as domain join, group policy, LDAP, and Kerberos / NTLM authentication.
 - Domain Services integrates with Microsoft Entra ID, which itself can synchronize with an on-premises AD DS environment. This ability extends central identity

use cases to traditional web applications that run in Azure as part of a lift-and-shift strategy.

- To learn more about synchronization with Microsoft Entra ID and on-premises, see [How objects and credentials are synchronized in a managed domain](#).

This overview article compares and contrasts how these identity solutions can work together, or would be used independently, depending on the needs of your organization.

To get started, create a Domain Services managed domain using the Microsoft Entra admin center

Domain Services and self-managed AD DS

If you have applications and services that need access to traditional authentication mechanisms such as Kerberos or NTLM, there are two ways to provide Active Directory Domain Services in the cloud:

- A *managed domain* that you create using Microsoft Entra Domain Services. Microsoft creates and manages the required resources.
- A *self-managed* domain that you create and configure using traditional resources such as virtual machines (VMs), Windows Server guest OS, and Active Directory Domain Services (AD DS). You then continue to administer these resources.

With Domain Services, the core service components are deployed and maintained for you by Microsoft as a *managed* domain experience. You don't deploy, manage, patch, and secure the AD DS infrastructure for components like the VMs, Windows Server OS, or domain controllers (DCs).

Domain Services provides a smaller subset of features to traditional self-managed AD DS environment, which reduces some of the design and management complexity. For example, there are no AD forests, domain, sites, and replication links to design and maintain. You can still [create forest trusts between Domain Services and on-premises environments](#).

For applications and services that run in the cloud and need access to traditional authentication mechanisms such as Kerberos or NTLM, Domain Services provides a managed domain experience with the minimal amount of administrative overhead. For more information, see [Management concepts for user accounts, passwords, and administration in Domain Services](#).

When you deploy and run a self-managed AD DS environment, you have to maintain all of the associated infrastructure and directory components. There's additional maintenance overhead with a self-managed AD DS environment, but you're then able to do additional tasks such as extend the schema or create forest trusts.

Common deployment models for a self-managed AD DS environment that provides identity to applications and services in the cloud include the following:

- **Standalone cloud-only AD DS** - Azure VMs are configured as domain controllers and a separate, cloud-only AD DS environment is created. This AD DS environment doesn't integrate with an on-premises AD DS environment. A different set of credentials is used to sign in and administer VMs in the cloud.
- **Extend on-premises domain to Azure** - An Azure virtual network connects to an on-premises network using a VPN / ExpressRoute connection. Azure VMs connect to this Azure virtual network, which lets them domain-join to the on-premises AD DS environment.
 - An alternative is to create Azure VMs and promote them as replica domain controllers from the on-premises AD DS domain. These domain controllers replicate over a VPN / ExpressRoute connection to the on-premises AD DS environment. The on-premises AD DS domain is effectively extended into Azure.

The following table outlines some of the features you may need for your organization, and the differences between a managed domain or a self-managed AD DS domain:

Feature	Managed domain	Self-managed AD DS
Managed service	✓	✗
Secure deployments	✓	Administrator secures the deployment
DNS server	✓ (managed service)	✓
Domain or Enterprise administrator privileges	✗	✓
Domain join	✓	✓
Domain authentication using NTLM and Kerberos	✓	✓
Kerberos constrained delegation	Resource-based	Resource-based & account-based
Custom OU structure	✓	✓
Group Policy	✓	✓

Feature	Managed domain	Self-managed AD DS
Schema extensions	✗	✓
AD domain / forest trusts	✓ (one-way outbound forest trusts only)	✓
Secure LDAP (LDAPS)	✓	✓
LDAP read	✓	✓
LDAP write	✓ (within the managed domain)	✓
Geo-distributed deployments	✓	✓

Domain Services and Microsoft Entra ID

Microsoft Entra ID lets you manage the identity of devices used by the organization and control access to corporate resources from those devices. Users can also register their personal device (a bring-your-own (BYO) model) with Microsoft Entra ID, which provides the device with an identity. Microsoft Entra ID then authenticates the device when a user signs in to Microsoft Entra ID and uses the device to access secured resources. The device can be managed using Mobile Device Management (MDM) software like Microsoft Intune. This management ability lets you restrict access to sensitive resources to managed and policy-compliant devices.

Traditional computers and laptops can also join to Microsoft Entra ID. This mechanism offers the same benefits of registering a personal device with Microsoft Entra ID, such as to allow users to sign in to the device using their corporate credentials.

Microsoft Entra joined devices give you the following benefits:

- Single-sign-on (SSO) to applications secured by Microsoft Entra ID.
- Enterprise policy-compliant roaming of user settings across devices.
- Access to the Windows Store for Business using corporate credentials.
- Windows Hello for Business.
- Restricted access to apps and resources from devices compliant with corporate policy.

Devices can be joined to Microsoft Entra ID with or without a hybrid deployment that includes an on-premises AD DS environment. The following table outlines common device ownership models and how they would typically be joined to a domain:

Type of device	Device platforms	Mechanism
Personal devices	Windows 10, iOS, Android, macOS	Microsoft Entra registered
Organization-owned device not joined to on-premises AD DS	Windows 10	Microsoft Entra joined
Organization-owned device joined to an on-premises AD DS	Windows 10	Microsoft Entra hybrid joined

On a Microsoft Entra joined or registered device, user authentication happens using modern OAuth / OpenID Connect based protocols. These protocols are designed to work over the internet, so are great for mobile scenarios where users access corporate resources from anywhere.

With Domain Services-joined devices, applications can use the Kerberos and NTLM protocols for authentication, so can support legacy applications migrated to run on Azure VMs as part of a lift-and-shift strategy. The following table outlines differences in how the devices are represented and can authenticate themselves against the directory:

Aspect	Microsoft Entra joined	Domain Services-joined
Device controlled by	Microsoft Entra ID	Domain Services managed domain
Representation in the directory	Device objects in the Microsoft Entra directory	Computer objects in the Domain Services managed domain
Authentication	OAuth / OpenID Connect based protocols	Kerberos and NTLM protocols
Management	Mobile Device Management (MDM) software like Intune	Group Policy
Networking	Works over the internet	Must be connected to, or peered with, the virtual network where the managed domain is deployed
Great for...	End-user mobile or desktop devices	Server VMs deployed in Azure

If on-premises AD DS and Microsoft Entra ID are configured for federated authentication using AD FS, then there's no (current/valid) password hash available in Azure DS. Microsoft Entra user accounts created before fed auth was implemented might have an old password hash but this likely doesn't match a hash of their on-premises password. As a result, Domain Services won't be able to validate the users credentials

Next steps

To get started with using Domain Services, [create a Domain Services managed domain using the Microsoft Entra admin center](#).

You can also learn more about [management concepts for user accounts, passwords, and administration in Domain Services](#) and [how objects and credentials are synchronized in a managed domain](#).

Choose the right authentication method for your Microsoft Entra hybrid identity solution

Article • 11/06/2023

Choosing the correct authentication method is the first concern for organizations wanting to move their apps to the cloud. Don't take this decision lightly, for the following reasons:

1. It's the first decision for an organization that wants to move to the cloud.
2. The authentication method is a critical component of an organization's presence in the cloud. It controls access to all cloud data and resources.
3. It's the foundation of all the other advanced security and user experience features in Microsoft Entra ID.

Identity is the new control plane of IT security, so authentication is an organization's access guard to the new cloud world. Organizations need an identity control plane that strengthens their security and keeps their cloud apps safe from intruders.

ⓘ Note

Changing your authentication method requires planning, testing, and potentially downtime. **Staged rollout** is a great way to test users' migration from federation to cloud authentication.

Out of scope

Organizations that don't have an existing on-premises directory footprint aren't the focus of this article. Typically, those businesses create identities only in the cloud, which doesn't require a hybrid identity solution. Cloud-only identities exist solely in the cloud and aren't associated with corresponding on-premises identities.

Authentication methods

When the Microsoft Entra hybrid identity solution is your new control plane, authentication is the foundation of cloud access. Choosing the correct authentication method is a crucial first decision in setting up a Microsoft Entra hybrid identity solution.

The authentication method you choose, is configured by using Microsoft Entra Connect, which also provisions users in the cloud.

To choose an authentication method, you need to consider the time, existing infrastructure, complexity, and cost of implementing your choice. These factors are different for every organization and might change over time.

<https://www.youtube-nocookie.com/embed/YtW2cmVqSEw>

Microsoft Entra ID supports the following authentication methods for hybrid identity solutions.

Cloud authentication

When you choose this authentication method, Microsoft Entra ID handles users' sign-in process. Coupled with single sign-on (SSO), users can sign in to cloud apps without having to reenter their credentials. With cloud authentication, you can choose from two options:

Microsoft Entra password hash synchronization. The simplest way to enable authentication for on-premises directory objects in Microsoft Entra ID. Users can use the same username and password that they use on-premises without having to deploy any other infrastructure. Some premium features of Microsoft Entra ID, like Identity Protection and [Microsoft Entra Domain Services](#), require password hash synchronization, no matter which authentication method you choose.

Note

Passwords are never stored in clear text or encrypted with a reversible algorithm in Microsoft Entra ID. For more information on the actual process of password hash synchronization, see [Implement password hash synchronization with Microsoft Entra Connect Sync](#).

Microsoft Entra pass-through authentication. Provides a simple password validation for Microsoft Entra authentication services by using a software agent that runs on one or more on-premises servers. The servers validate the users directly with your on-premises Active Directory, which ensures that the password validation doesn't happen in the cloud.

Companies with a security requirement to immediately enforce on-premises user account states, password policies, and sign-in hours might use this authentication method. For more information on the actual pass-through authentication process, see [User sign-in with Microsoft Entra pass-through authentication](#).

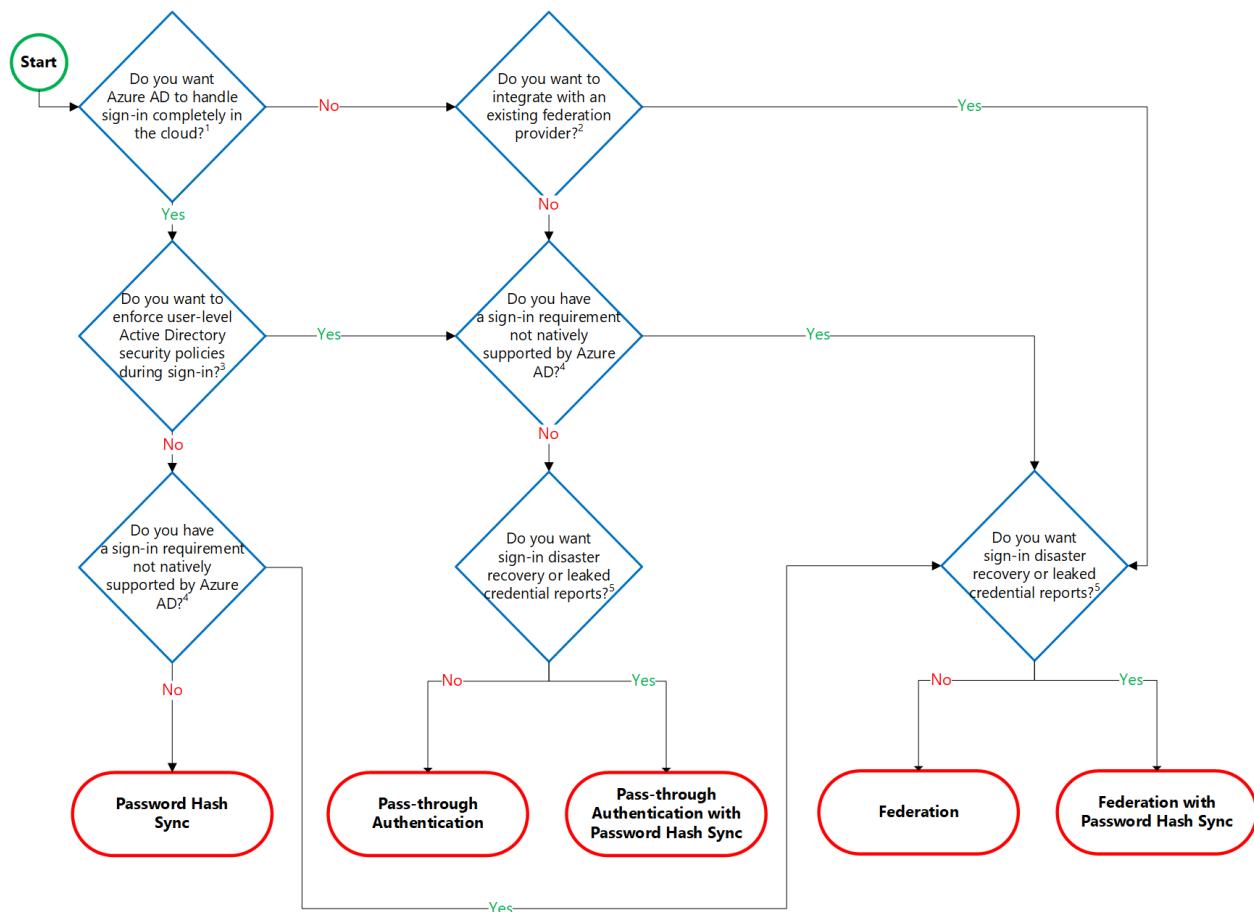
Federated authentication

When you choose this authentication method, Microsoft Entra ID hands off the authentication process to a separate trusted authentication system, such as on-premises Active Directory Federation Services (AD FS), to validate the user's password.

The authentication system can provide other advanced authentication requirements, for example, third-party multifactor authentication.

The following section helps you decide which authentication method is right for you by using a decision tree. It helps you determine whether to deploy cloud or federated authentication for your Microsoft Entra hybrid identity solution.

Decision tree



Details on decision questions:

1. Microsoft Entra ID can handle sign-in for users without relying on on-premises components to verify passwords.
2. Microsoft Entra ID can hand off user sign-in to a trusted authentication provider such as Microsoft's AD FS.
3. If you need to apply, user-level Active Directory security policies such as account expired, disabled account, password expired, account locked out, and sign-in hours

on each user sign-in, Microsoft Entra ID requires some on-premises components.

4. Sign-in features not natively supported by Microsoft Entra ID:

- Sign-in using third-party authentication solution.
- Multi-site on-premises authentication solution.

5. Microsoft Entra ID Protection requires Password Hash Sync regardless of which sign-in method you choose, to provide the *Users with leaked credentials* report. Organizations can fail over to Password Hash Sync if their primary sign-in method fails and it was configured before the failure event.

Note

Microsoft Entra ID Protection require [Microsoft Entra ID P2](#) licenses.

Detailed considerations

Cloud authentication: Password hash synchronization

- **Effort.** Password hash synchronization requires the least effort regarding deployment, maintenance, and infrastructure. This level of effort typically applies to organizations that only need their users to sign in to Microsoft 365, SaaS apps, and other Microsoft Entra ID-based resources. When turned on, password hash synchronization is part of the Microsoft Entra Connect Sync process and runs every two minutes.
- **User experience.** To improve users' sign-in experience, use [Microsoft Entra joined devices](#) or [Microsoft Entra hybrid joined devices](#). If you can't join your Windows devices to Microsoft Entra ID, we recommend deploying seamless SSO with password hash synchronization. Seamless SSO eliminates unnecessary prompts when users are signed in.
- **Advanced scenarios.** If organizations choose to, it's possible to use insights from identities with Microsoft Entra ID Protection reports with Microsoft Entra ID P2. An example is the leaked credentials report. Windows Hello for Business has [specific requirements when you use password hash synchronization](#). [Microsoft Entra Domain Services](#) requires password hash synchronization to provision users with their corporate credentials in the managed domain.

Organizations that require multifactor authentication with password hash synchronization must use Microsoft Entra multifactor authentication or [Conditional](#)

[Access custom controls](#). Those organizations can't use third-party or on-premises multifactor authentication methods that rely on federation.

 **Note**

Microsoft Entra Conditional Access require Microsoft Entra ID P1  licenses.

- **Business continuity.** Using password hash synchronization with cloud authentication is highly available as a cloud service that scales to all Microsoft datacenters. To make sure password hash synchronization doesn't go down for extended periods, deploy a second Microsoft Entra Connect server in staging mode in a standby configuration.
- **Considerations.** Currently, password hash synchronization doesn't immediately enforce changes in on-premises account states. In this situation, a user has access to cloud apps until the user account state is synchronized to Microsoft Entra ID. Organizations might want to overcome this limitation by running a new synchronization cycle after administrators do bulk updates to on-premises user account states. An example is disabling accounts.

 **Note**

The password expired and account locked-out states aren't currently synced to Microsoft Entra ID with Microsoft Entra Connect. When you change a user's password and set the *user must change password at next logon* flag, the password hash will not be synced to Microsoft Entra ID with Microsoft Entra Connect until the user changes their password.

Refer to [implementing password hash synchronization](#) for deployment steps.

Cloud authentication: Pass-through Authentication

- **Effort.** For pass-through authentication, you need one or more (we recommend three) lightweight agents installed on existing servers. These agents must have access to your on-premises Active Directory Domain Services, including your on-premises AD domain controllers. They need outbound access to the Internet and access to your domain controllers. For this reason, it's not supported to deploy the agents in a perimeter network.

Pass-through Authentication requires unconstrained network access to domain controllers. All network traffic is encrypted and limited to authentication requests.

For more information on this process, see the [security deep dive](#) on pass-through authentication.

- **User experience.** To improve users' sign-in experience, use [Microsoft Entra joined devices](#) or [Microsoft Entra hybrid joined devices](#). If you can't join your Windows devices to Microsoft Entra ID, we recommend deploying seamless SSO with password hash synchronization. Seamless SSO eliminates unnecessary prompts when users are signed in.
- **Advanced scenarios.** Pass-through Authentication enforces the on-premises account policy at the time of sign-in. For example, access is denied when an on-premises user's account state is disabled, locked out, or their [password expires](#) or the logon attempt falls outside the hours when the user is allowed to sign in.

Organizations that require multifactor authentication with pass-through authentication must use Microsoft Entra multifactor authentication or [Conditional Access custom controls](#). Those organizations can't use a third-party or on-premises multifactor authentication method that relies on federation. Advanced features require that password hash synchronization is deployed whether or not you choose pass-through authentication. An example is the leaked credentials report of Identity Protection.

- **Business continuity.** We recommend that you deploy two extra pass-through authentication agents. These extras are in addition to the first agent on the Microsoft Entra Connect server. This other deployment ensures high availability of authentication requests. When you have three agents deployed, one agent can still fail when another agent is down for maintenance.

There's another benefit to deploying password hash synchronization in addition to pass-through authentication. It acts as a backup authentication method when the primary authentication method is no longer available.

- **Considerations.** You can use password hash synchronization as a backup authentication method for pass-through authentication, when the agents can't validate a user's credentials due to a significant on-premises failure. Fail over to password hash synchronization doesn't happen automatically and you must use Microsoft Entra Connect to switch the sign-on method manually.

For other considerations on Pass-through Authentication, including Alternate ID support, see [frequently asked questions](#).

Refer to [implementing pass-through authentication](#) for deployment steps.

Federated authentication

- **Effort.** A federated authentication system relies on an external trusted system to authenticate users. Some companies want to reuse their existing federated system investment with their Microsoft Entra hybrid identity solution. The maintenance and management of the federated system falls outside the control of Microsoft Entra ID. It's up to the organization by using the federated system to make sure it's deployed securely and can handle the authentication load.
- **User experience.** The user experience of federated authentication depends on the implementation of the features, topology, and configuration of the federation farm. Some organizations need this flexibility to adapt and configure the access to the federation farm to suit their security requirements. For example, it's possible to configure internally connected users and devices to sign in users automatically, without prompting them for credentials. This configuration works because they already signed in to their devices. If necessary, some advanced security features make users' sign-in process more difficult.
- **Advanced scenarios.** A federated authentication solution is required when customers have an authentication requirement that Microsoft Entra ID doesn't support natively. See detailed information to help you [choose the right sign-in option](#). Consider the following common requirements:
 - Third-party multifactor providers requiring a federated identity provider.
 - Authentication by using third-party authentication solutions. See the [Microsoft Entra federation compatibility list](#).
 - Sign in that requires a sAMAccountName, for example DOMAIN\username, instead of a User Principal Name (UPN), for example, user@domain.com.
- **Business continuity.** Federated systems typically require a load-balanced array of servers, known as a farm. This farm is configured in an internal network and perimeter network topology to ensure high availability for authentication requests.

Deploy password hash synchronization along with federated authentication as a backup authentication method when the primary authentication method is no longer available. An example is when the on-premises servers aren't available. Some large enterprise organizations require a federation solution to support multiple Internet ingress points configured with geo-DNS for low-latency authentication requests.

- **Considerations.** Federated systems typically require a more significant investment in on-premises infrastructure. Most organizations choose this option if they already have an on-premises federation investment. And if it's a strong business

requirement to use a single-identity provider. Federation is more complex to operate and troubleshoot compared to cloud authentication solutions.

For a nonroutable domain that can't be verified in Microsoft Entra ID, you need extra configuration to implement user ID sign in. This requirement is known as Alternate login ID support. See [Configuring Alternate Login ID](#) for limitations and requirements. If you choose to use a third-party multifactor authentication provider with federation, ensure the provider supports WS-Trust to allow devices to join Microsoft Entra ID.

Refer to [Deploying Federation Servers](#) for deployment steps.

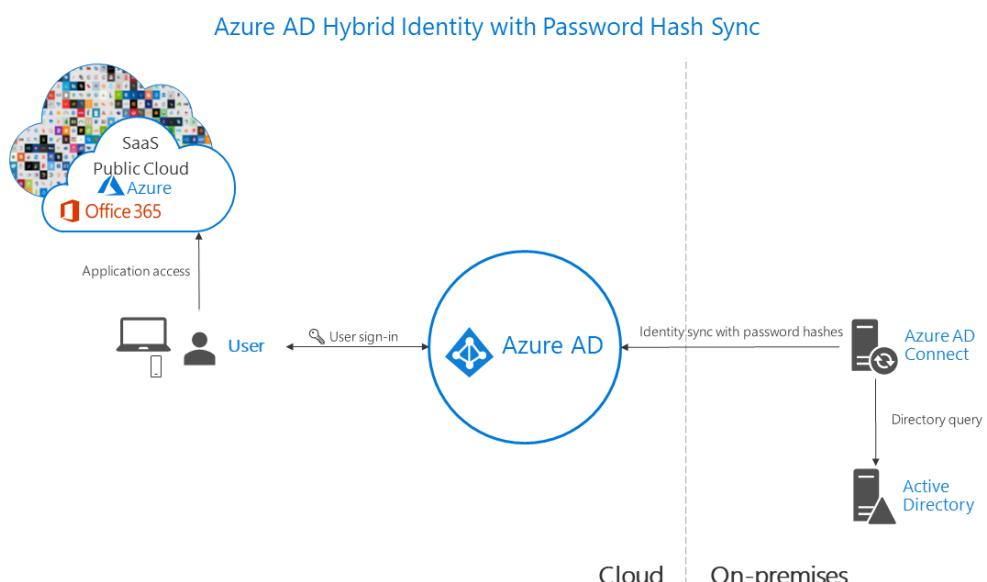
ⓘ Note

When you deploy your Microsoft Entra hybrid identity solution, you must implement one of the supported topologies of Microsoft Entra Connect. Learn more about supported and unsupported configurations at [Topologies for Microsoft Entra Connect](#).

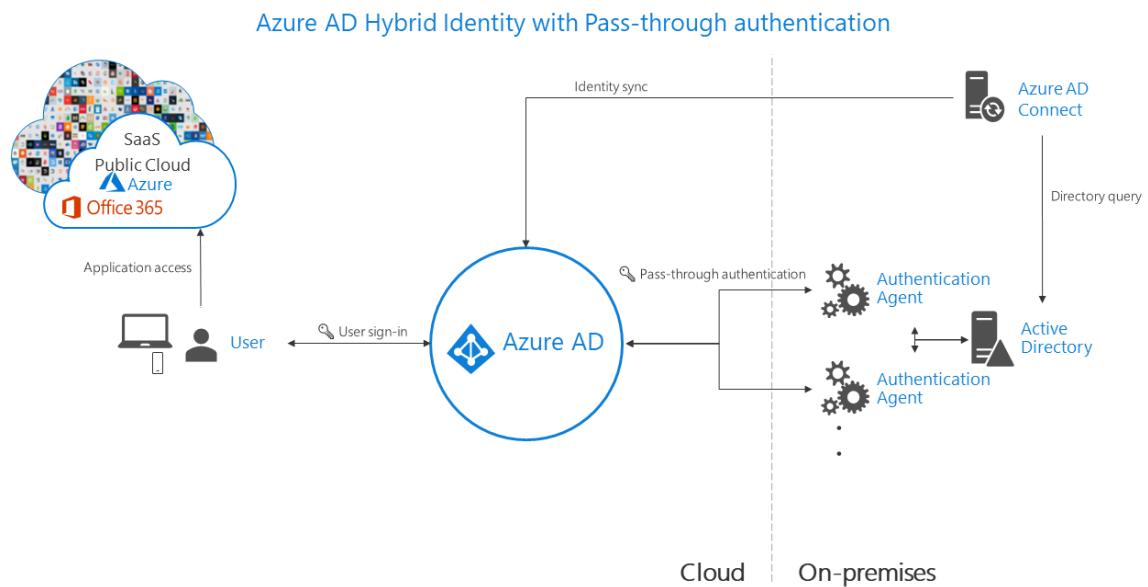
Architecture diagrams

The following diagrams outline the high-level architecture components required for each authentication method you can use with your Microsoft Entra hybrid identity solution. They provide an overview to help you compare the differences between the solutions.

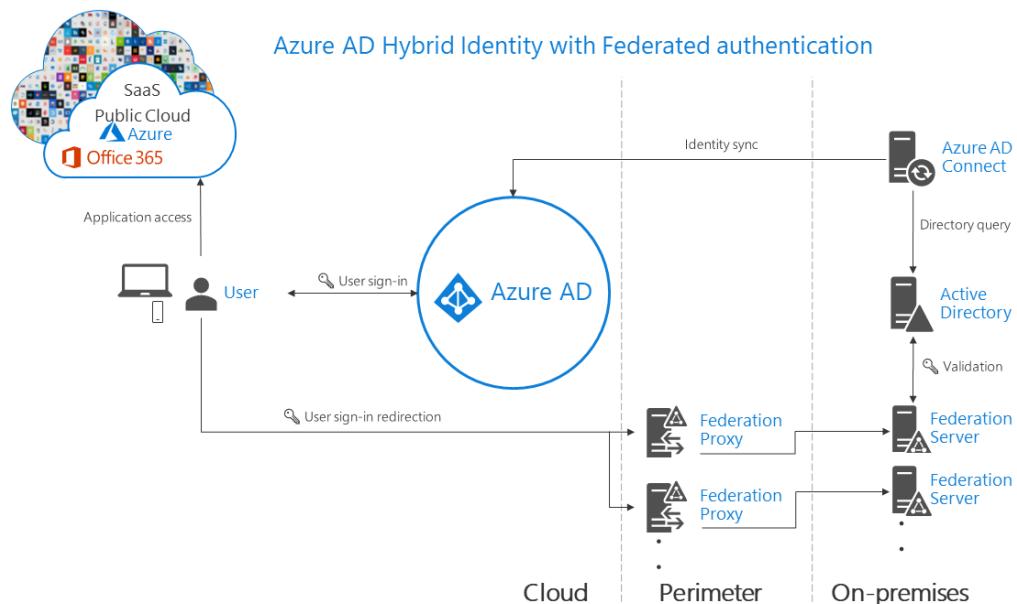
- Simplicity of a password hash synchronization solution:



- Agent requirements of pass-through authentication, using two agents for redundancy:



- Components required for federation in your perimeter and internal network of your organization:



Comparing methods

Consideration	Password hash synchronization	Pass-through Authentication	Federation with AD FS
Where does authentication happen?	In the cloud	In the cloud, after a secure password verification exchange	On-premises

Consideration	Password hash synchronization	Pass-through Authentication	Federation with AD FS
		with the on-premises authentication agent	
What are the on-premises server requirements beyond the provisioning system: Microsoft Entra Connect?	None	One server for each additional authentication agent	Two or more AD FS servers Two or more WAP servers in the perimeter/DMZ network
What are the requirements for on-premises Internet and networking beyond the provisioning system?	None	Outbound Internet access from the servers running authentication agents	Inbound Internet access to WAP servers in the perimeter Inbound network access to AD FS servers from WAP servers in the perimeter Network load balancing
Is there a TLS/SSL certificate requirement?	No	No	Yes
Is there a health monitoring solution?	Not required	Agent status provided by the [Microsoft Entra admin center](tshoot-connect-pass-through-authentication.md)	Microsoft Entra Connect Health
Do users get single sign-on to cloud resources from domain-joined devices within the company network?	Yes with Microsoft Entra joined devices , Microsoft Entra hybrid joined devices , the Microsoft Enterprise SSO plug-in for Apple devices , or Seamless SSO	Yes with Microsoft Entra joined devices , Microsoft Entra hybrid joined devices , the Microsoft Enterprise SSO plug-in for Apple devices , or Seamless SSO	Yes
What sign-in types are supported?	UserPrincipalName + password Windows-Integrated	UserPrincipalName + password Windows-Integrated	UserPrincipalName + password sAMAccountName +

Consideration	Password hash synchronization	Pass-through Authentication	Federation with AD FS
	Authentication by using Seamless SSO Alternate login ID Microsoft Entra joined Devices Microsoft Entra hybrid joined devices Certificate and smart card authentication	Authentication by using Seamless SSO Alternate login ID Microsoft Entra joined Devices Microsoft Entra hybrid joined devices Certificate and smart card authentication	password Windows-Integrated Authentication Certificate and smart card authentication Alternate login ID
Is Windows Hello for Business supported?	Key trust model Hybrid Cloud Trust <i>Both require Windows Server 2016 Domain functional level</i>	Key trust model Hybrid Cloud Trust <i>Both require Windows Server 2016 Domain functional level</i>	Key trust model Hybrid Cloud Trust Certificate trust model
What are the multifactor authentication options?	Microsoft Entra multifactor authentication Custom Controls with Conditional Access*	Microsoft Entra multifactor authentication Custom Controls with Conditional Access*	Microsoft Entra multifactor authentication Third-party MFA Custom Controls with Conditional Access*
What user account states are supported?	Disabled accounts (up to 30-minute delay) Account locked out Account expired Password expired Sign-in hours	Disabled accounts Account locked out Account expired Password expired Sign-in hours	Disabled accounts Account locked out Account expired Password expired Sign-in hours
What are the Conditional Access options?	Microsoft Entra Conditional Access, with Microsoft Entra ID P1 or P2	Microsoft Entra Conditional Access, with Microsoft Entra ID P1 or P2	Microsoft Entra Conditional Access, with Microsoft Entra ID P1 or P2

Consideration	Password hash synchronization	Pass-through Authentication	Federation with AD FS
			AD FS claim rules
Is blocking legacy protocols supported?	Yes	Yes	Yes
Can you customize the logo, image, and description on the sign-in pages?	Yes, with Microsoft Entra ID P1 or P2	Yes, with Microsoft Entra ID P1 or P2	Yes
What advanced scenarios are supported?	Smart password lockout Leaked credentials reports, with Microsoft Entra ID P2	Smart password lockout	Multisite low-latency authentication system AD FS extranet lockout Integration with third-party identity systems

ⓘ Note

Custom controls in Microsoft Entra Conditional Access do not currently support device registration.

Recommendations

Your identity system ensures your users' access to apps that you migrate and make available in the cloud. Use or enable password hash synchronization with whichever authentication method you choose, for the following reasons:

- 1. High availability and disaster recovery.** Pass-through Authentication and federation rely on on-premises infrastructure. For pass-through authentication, the on-premises footprint includes the server hardware and networking the Pass-through Authentication agents require. For federation, the on-premises footprint is even larger. It requires servers in your perimeter network to proxy authentication requests and the internal federation servers.

To avoid single points of failure, deploy redundant servers. Then authentication requests will always be serviced if any component fails. Both pass-through authentication and federation also rely on domain controllers to respond to authentication requests, which can also fail. Many of these components need maintenance to stay healthy. Outages are more likely when maintenance isn't planned and implemented correctly.

2. **On-premises outage survival.** The consequences of an on-premises outage due to a cyber-attack or disaster can be substantial, ranging from reputational brand damage to a paralyzed organization unable to deal with the attack. Recently, many organizations were victims of malware attacks, including targeted ransomware, which caused their on-premises servers to go down. When Microsoft helps customers deal with these kinds of attacks, it sees two categories of organizations:

- Organizations that previously also turned on password hash synchronization on top of federated or pass-through authentication changed their primary authentication method to then use password hash synchronization. They were back online in a matter of hours. By using access to email via Microsoft 365, they worked to resolve issues and access other cloud-based workloads.
- Organizations that didn't previously enable password hash synchronization had to resort to untrusted external consumer email systems for communications to resolve issues. In those cases, it took them weeks to restore their on-premises identity infrastructure, before users were able to sign in to cloud-based apps again.

3. **Identity protection.** One of the best ways to protect users in the cloud is Microsoft Entra ID Protection with Microsoft Entra ID P2. Microsoft continually scans the Internet for user and password lists that bad actors sell and make available on the dark web. Microsoft Entra ID can use this information to verify if any of the usernames and passwords in your organization are compromised. Therefore, it's critical to enable password hash synchronization no matter which authentication method you use, whether it's federated or pass-through authentication. Leaked credentials are presented as a report. Use this information to block or force users to change their passwords when they try to sign in with leaked passwords.

Conclusion

This article outlines various authentication options that organizations can configure and deploy to support access to cloud apps. To meet various business, security, and technical requirements, organizations can choose between password hash synchronization, Pass-through Authentication, and federation.

Consider each authentication method. Does the effort to deploy the solution, and the user's experience of the sign-in process address your business requirements? Evaluate whether your organization needs the advanced scenarios and business continuity features of each authentication method. Finally, evaluate the considerations of each authentication method. Do any of them prevent you from implementing your choice?

Next steps

In today's world, threats are present 24 hours a day and come from everywhere. Implement the correct authentication method, and it will mitigate your security risks and protect your identities.

[Get started](#) with Microsoft Entra ID and deploy the right authentication solution for your organization.

If you're thinking about migrating from federated to cloud authentication, learn more about [changing the sign-in method](#). To help you plan and implement the migration, use [these project deployment plans](#), or consider using the new [Staged Rollout](#) feature to migrate federated users to using cloud authentication in a staged approach.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Review your storage options

Article • 03/30/2023

Storage capabilities are critical for supporting workloads and services that are hosted in the cloud. As you prepare for your cloud adoption, review this information to plan for your storage needs.

Select storage tools and services to support your workloads

Azure Storage is the Azure platform's managed service for providing cloud storage. Azure Storage is composed of several core services and supporting features. Storage in Azure is highly available, secure, durable, scalable, and redundant. Use these scenarios and considerations to choose Azure services and architectures. For more information, see [Azure Storage documentation](#).

Key questions

Answer the following questions about your workloads to help make decisions about your storage needs:

- **Do your workloads require disk storage to support the deployment of infrastructure as a service (IaaS) virtual machines?** [Azure managed disks](#) provide virtual disk capabilities for IaaS virtual machines.
- **Will you need to provide downloadable images, documents, or other media as part of your workloads?** [Azure Blob Storage](#) hosts static files, which are then accessible for download over the internet. For more information, see [Static website hosting in Azure Storage](#).
- **Will you need a location to store virtual machine logs, application logs, and analytics data?** You can use Blob Storage to store Azure Monitor log data. See [Azure Storage Analytics](#).
- **Will you need to provide a location for backup, disaster recovery, or archiving workload-related data?** Blob Storage provides backup and disaster recovery capabilities. For more information, see [Backup and disaster recovery for Azure IaaS disks](#).

You can also use Blob Storage to back up other resources, like on-premises or IaaS virtual machine-hosted SQL Server data. See [SQL Server Backup and Restore](#).

- **Will you need to support big data analytics workloads?** [Azure Data Lake Storage Gen2](#) is built on Azure Blob Storage. Data Lake Storage Gen2 supports large-enterprise data lake functionality. It also can handle storing petabytes of information while sustaining hundreds of gigabits of throughput.
- **Will you need to provide cloud-native file shares?** Azure has two services that provide cloud-hosted file shares:
 - [Azure NetApp Files](#) provides high-performance NFS and SMB shares, with advanced data management features such as snapshots and cloning, that are well suited to common enterprise workloads like SAP.
 - [Azure Files](#) provides file shares accessible over SMB 3.0 and HTTPS.
- **Will you need to support hybrid cloud storage for on-premises high-performance computing (HPC) workloads?** [Avere vFXT for Azure](#) is a hybrid caching solution. You can expand your on-premises storage capabilities by using cloud-based storage. Avere vFXT for Azure is optimized for read-heavy HPC workloads that involve 1,000 to 40,000 CPU cores. Avere vFXT for Azure can integrate with on-premises hardware network attached storage (NAS), Blob Storage, or both.
- **Will you need to perform large-scale archiving and syncing of your on-premises data?** [Azure Data Box](#) products are designed to help you move large amounts of data from your on-premises environment to the cloud.
 - [Azure Data Box Gateway](#) is a virtual device that's on-premises. Data Box Gateway helps you manage large-scale data migration to the cloud.
 - [Azure Stack Edge](#) accelerates processing and the secure transfer of data to Azure. If you need to analyze, transform, or filter data before you move it to the cloud, use Azure Data Box.
- **Do you want to expand an existing on-premises file share to use cloud storage?** [Azure File Sync](#) lets you use the Azure Files service as an extension of file shares that are hosted on your on-premises Windows Server computers. The syncing service transforms Windows Server into a quick cache of your Azure file share. It allows your on-premises computers that access the share to use any protocol that's available on Windows Server.

Common storage scenarios

Azure offers multiple products and services for different storage capabilities. The following table describes potential storage scenarios and the recommended Azure services.

Block storage scenarios

Scenario	Suggested Azure services	Considerations for suggested services
I have bare-metal servers or virtual machines (Hyper-V or VMware) with direct attached storage running line-of-business applications.	Azure Premium SSD	For production services, Premium SSD option provides consistent low-latency coupled with high input/output operations per second (IOPS) and throughput.
I have servers that will host web and mobile apps.	Azure Standard SSD	Standard SSD IOPS and throughput might be sufficient at a lower cost than Premium SSD for CPU-bound web and application servers in production.
I have an enterprise SAN or all-flash array.	Premium SSD or Azure Ultra Disk Storage or Azure NetApp Files	Ultra Disk Storage is NVMe-based and offers submillisecond latency with high IOPS and bandwidth. Ultra Disk Storage is scalable up to 64 TB. The choice of Premium SSD or Ultra Disk Storage depends on peak latency, IOPS, and scalability requirements.
I have high-availability clustered servers, such as SQL Server FCI or Windows Server failover clustering.	Azure Files or Premium SSD or Ultra Disk Storage	Clustered workloads require multiple nodes to mount the same underlying shared storage for failover or high availability. Premium file shares offer shared storage that's mountable by using SMB. Shared block storage also can be configured on Premium SSD or Ultra Disk Storage by using partner solutions. See SIOS DataKeeper Cluster Edition .
I have a relational database or data warehouse workload, such as	Premium SSD or Ultra Disk Storage	The choice of Premium SSD or Ultra Disk Storage depends on peak latency, IOPS, and scalability requirements. Ultra Disk Storage also

Scenario	Suggested Azure services	Considerations for suggested services
SQL Server or Oracle.		reduces complexity by removing the need for storage pool configuration for scalability. See Mission critical performance .
I have a NoSQL cluster such as Cassandra or MongoDB.	Premium SSD	Azure disk storage Premium SSD provides consistent low-latency coupled with high IOPS and throughput.
I have containers with persistent volumes.	Azure Files or Standard SSD , Premium SSD , or Ultra Disk Storage	File (RWX) and block (RWO) volumes driver options are available for both Azure Kubernetes Service and custom Kubernetes deployments. Persistent volumes can map to either an Azure disk storage disk or a managed Azure Files share. Choose premium versus standard options based on workload requirements for persistent volumes.
I have a data lake such as a Hadoop cluster for HDFS data.	Data Lake Storage Gen2 or Standard SSD or Premium SSD	The Data Lake Storage Gen2 feature of Blob Storage provides server-side HDFS compatibility and petabyte scale for parallel analytics. It also offers high availability and reliability. Software like Cloudera can use Premium SSD or Standard SSD on controller/worker nodes, if needed.
I have an SAP or SAP HANA deployment.	Premium SSD or Ultra Disk Storage	Ultra Disk Storage is optimized to offer submillisecond latency for tier-1 SAP workloads. Premium SSD, coupled with M-series virtual machines, offers a general-availability option.
I have a disaster recovery site with strict RPO/RTO that syncs from my primary servers.	Azure page blobs	Page blobs are used by replication software to enable low-cost replication to Azure without the need for compute virtual machines until failover occurs. For more information, see Backup and disaster recovery for Azure IaaS disks . Note: Page blobs support a maximum of 8 TB.

File and object storage scenarios

Scenario	Suggested Azure services	Considerations for suggested services
I use Windows file server.	Azure Files or Azure File Sync	With Azure File Sync, you can store rarely used data on cloud-based file shares while caching your most frequently used files on-premises. You can also keep files in sync across multiple servers. If you plan to migrate your workloads to a cloud-only deployment, Azure Files might be sufficient.
I have an enterprise network attached storage such as NetApp or Dell-EMC Isilon.	Azure NetApp Files or Azure Files (premium)	If you have an on-premises deployment of NetApp, consider using Azure NetApp Files to migrate your deployment to Azure. If you're using or migrating to a Windows or Linux server, consider using Azure Files. Also, if you have basic file-share needs, consider using Azure Files. For continued on-premises access, use Azure File Sync to sync file shares with on-premises file shares by using a cloud-tiering mechanism. Cloud tiering uses your server as a cache for the relevant files while scaling cold data in Azure file shares.
I have an SMB or NFS file share.	Azure Files or Azure NetApp Files	The choice of premium or standard Azure Files tiers depends on IOPS, throughput, and your need for latency consistency. If you have an on-premises deployment of NetApp, consider using Azure NetApp Files. If you need to migrate your access control lists and timestamps to the cloud, Azure File Sync can bring these settings to your Azure file shares.
I have an on-premises object storage system for petabytes of data, such as Dell-EMC ECS.	Blob Storage	Azure Blob Storage provides premium, hot, cool, and archive tiers to match your workload performance and cost needs.
I have a Distributed File	Azure Files or Azure File Sync	Azure File Sync offers multisite sync for multiple servers and native Azure file

Scenario	Suggested Azure services	Considerations for suggested services
System Replication deployment or another way of handling branch offices.	Sync	shares. Move to a fixed storage footprint on-premises by using cloud tiering.
I have a tape library for backup and disaster recovery or long-term data retention.	Blob Storage	A Blob Storage archive tier has the lowest possible cost. It might require hours to copy the offline data to a cool, hot, or Premium tier to allow access. Cool tiers provide instantaneous access at low cost.
I have file or object storage configured to receive my backups.	Blob Storage or Azure File Sync	To back up data for long-term retention with lowest-cost storage, move data to Blob Storage and use cool and archive tiers. To enable fast disaster recovery for file data on a server, sync shares to individual Azure file shares by using Azure File Sync. With Azure file share snapshots, you can restore earlier versions. Sync them back to connected servers or access them natively in the Azure file share.
I run data replication to a disaster recovery site.	Azure Files, Azure NetApp Files or Azure File Sync	Azure File Sync removes the need for a disaster recovery server and stores files in native Azure SMB shares. Fast disaster recovery rebuilds any data on a failed on-premises server quickly. You can even keep multiple server locations in sync or use cloud tiering to store only relevant data on-premises.
		Azure NetApp Files provides a storage based feature called Cross region replication which can be used to replicate data to other Azure regions which does not use any VM or application server resources, and is highly optimized to replicate only changed data blocks between updates.
I manage data transfer in	Azure Stack Edge or Data Box Gateway	Using Data Stack Edge or Data Box Gateway, you can copy data in

Scenario	Suggested Azure services	Considerations for suggested services
disconnected scenarios.	Box Gateway	disconnected scenarios. When the gateway is offline, it saves all files you copy in the cache, then uploads them when you're connected.
I manage an ongoing data pipeline to the cloud.	Azure Stack Edge or Data Box Gateway	Move data to the cloud from systems that are constantly generating data by having them copy that data to the storage gateway.
I have bursts of data that arrive at the same time.	Azure Stack Edge or Data Box Gateway	Manage large quantities of data that arrive at the same time. Some examples are when an autonomous car pulls into the garage or a gene sequencing machine finishes its analysis. Copy all that data to Data Box Gateway at fast local speeds. Then, let the gateway upload it as your network allows.

Plan based on data workloads

Scenario	Suggested Azure services	Considerations for suggested services
I want to develop a new cloud-native application that needs to persist unstructured data.	Blob Storage	
I need to migrate data from an on-premises NetApp instance to Azure.	Azure NetApp Files	
I need to migrate data from on-premises Windows file server instances to Azure.	Azure Files	
I need to move file data to the cloud but continue to primarily access the data from on-premises.	Azure Files or Azure File Sync	
I need to support burst compute. I have NFS/SMB read-heavy, file-based workloads with data assets that are on-premises while computation runs in the cloud.	Avere vFXT for Azure	IaaS scale-out NFS/SMB file caching.
I need to move an on-premises	Azure disk	

Scenario	Suggested Azure services	Considerations for suggested services
application that uses a local disk or iSCSI.	storage	
I need to migrate a container-based application that has persistent volumes.	Azure disk storage or Azure Files	
I need to move file shares that aren't on Windows Server or NetApp to the cloud.	Azure Files or Azure NetApp Files	Protocol supports regional availability performance requirements snapshot and clone capabilities price sensitivity.
I need to transfer terabytes to petabytes of data from on-premises to Azure.	Azure Stack Edge	
I need to process data before transferring it to Azure.	Azure Stack Edge	
I need to support continuous data ingestion in an automated way by using local cache.	Data Box Gateway	

Learn more about Azure storage services

After you identify the Azure tools that best match your requirements, use this detailed documentation to learn more about these services:

Service	Description
Azure Blob Storage	<p>Blob Storage is an object storage solution for the cloud. Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that doesn't adhere to a specific data model or definition, such as text or binary data.</p> <p>Use Blob Storage for the following needs:</p> <ul style="list-style-type: none"> - Serving images or documents directly to a browser. - Storing files for distributed access. - Streaming video and audio. - Writing to log files. - Storing data for backup and restore, disaster recovery, and archiving. - Storing data for analysis by an on-premises or Azure-hosted service.
Data Lake Storage Gen2	<p>Blob Storage supports Data Lake Storage Gen2, Microsoft's enterprise big data analytics solution for the cloud. Data Lake Storage Gen2 offers a hierarchical file system, with the advantages of Blob Storage. It also includes low-cost tiered storage, high availability, strong consistency, and disaster recovery capabilities.</p>

Service	Description
Azure disk storage	Azure disk storage offers persistent, high-performance block storage to power Azure Virtual Machines. Azure disks are highly durable, secure, and offer the industry's only single-instance, service-level agreement (SLA) for virtual machines that use Azure Premium SSD or Azure Ultra Disk Storage . Azure disks provide high availability with availability sets and availability zones for your Azure Virtual Machines fault domains. Azure manages disks as a top-level resource. Azure Resource Manager capabilities are provided, such as Azure role-based access control (Azure RBAC), policy, and tagging by default.
Azure Files	Azure Files provides fully managed, native SMB file shares, without the need to run a virtual machine. You can mount an Azure Files share as a network drive to any Azure virtual machine or on-premises computer.
Azure File Sync	Use Azure File Sync to centralize your file shares in Azure Files. Azure File Sync offers the flexibility, performance, and compatibility of an on-premises file server.
Azure NetApp Files	The Azure NetApp Files service is an enterprise-class, high-performance, metered file storage service. Azure NetApp Files supports any workload type and is highly available by default. You can select service and performance levels and set up snapshots through the service.
Azure Stack Edge	Azure Stack Edge is an on-premises network device that moves data into and out of Azure. Data Stack Edge has AI-enabled edge compute to pre-process data during upload. Data Box Gateway is a virtual version of the device but with the same data transfer capabilities.
Data Box Gateway	Data Box Gateway is a storage solution that enables you to seamlessly send data to Azure. It's a virtual device based on a virtual machine provisioned in your virtualized environment or hypervisor. The virtual device is on-premises and you write data to it by using the NFS and SMB protocols. The device then transfers your data to Azure block blobs, page blobs, or to Azure Files.
Avere vFXT for Azure	Avere vFXT for Azure is a filesystem caching solution for data-intensive HPC tasks. Take advantage of cloud computing's scalability to make your data accessible, even for data that's stored in your own on-premises hardware.

Data redundancy and availability

Azure Storage has various redundancy options to help ensure durability and high availability based on your needs.

- Locally redundant storage
- Zone-redundant storage
- Geo-redundant storage (GRS)
- Read-access GRS (RA-GRS)
- Geo-zone-redundant storage (GZRS)

To learn more about these capabilities and how to decide on the best redundancy option for your use cases, see [Azure Storage redundancy](#).

SLAs for storage services provide financially backed guarantees. For more information, see [SLA for managed disks](#), [SLA for virtual machines](#), and [SLA for storage accounts](#).

For help with planning the right solution for Azure disks, see [Backup and disaster recovery for Azure disk storage](#).

Security

To help protect your data in the cloud, Azure Storage offers several best practices for data security and encryption:

- Secure the storage account by using Azure RBAC and Microsoft Entra ID.
- Secure data in transit between an application and Azure by using client-side encryption, HTTPS, or SMB 3.0.
- Set data to be encrypted when it's written to Azure Storage by using Azure Storage encryption.
- Grant delegated access to the data objects in Azure Storage by using shared access signatures.
- Use analytics to track the authentication method that someone is using when they access storage in Azure.

These security features apply to Azure Blob Storage (block and page) and to Azure Files. For more information, see [Security recommendations for Blob Storage](#).

Azure Storage provides encryption at rest and safeguards your data. Azure Storage encryption is enabled by default for managed disks, snapshots, and images in all the Azure regions. All new managed disks, snapshots, images, and new data written to existing managed disks are encrypted at rest using keys managed by Microsoft. For more information, see [Azure Storage encryption](#) and [Managed disks and storage service encryption](#).

Azure Disk Encryption lets you encrypt managed disks that are attached to IaaS virtual machines at rest and in transit. [Azure Key Vault](#) stores your keys. For Windows, encrypt the drives by using industry-standard [BitLocker](#) encryption technology. For Linux, encrypt the disks by using the [dm-crypt](#) subsystem. The encryption process integrates with Azure Key Vault so you can control and manage the disk encryption keys. For more information, see [Azure Disk Encryption for virtual machines and virtual machine scale sets](#).

Regional availability

You can use Azure to deliver scaled services to reach your customers and partners wherever they are. Checking the regional availability of a service beforehand can help you make the right decision for your workload and customer needs. To check availability, see [Managed disks available by region](#) and [Azure Storage available by region](#).

Managed disks are available in all Azure regions that have Azure Premium SSD and Standard SSD offerings. Azure Ultra Disk Storage is offered in several availability zones. Verify the regional availability when you plan mission-critical, top-tier workloads that require Ultra Disk Storage.

Hot and cool Blob Storage, Data Lake Storage Gen2, and Azure Files storage are available in all Azure regions. Archival blob storage, premium file shares, and premium lock Blob Storage are limited to certain regions. Refer to the regions page to check the current status.

To learn more about Azure global infrastructure, see [Azure geographies](#). Consult [Products available by region](#) for storage options available in each Azure region.

Data residency and compliance requirements

Legal and contractual requirements that are related to data storage often apply to your workloads. These requirements depend on the location of your organization, the jurisdiction of the physical assets that host your data stores, and your business sector. Consider data classification, data location, and the respective responsibilities for data protection under the shared responsibility model. For more information, see [Enabling Data Residency and Data Protection in Microsoft Azure Regions](#).

Part of your compliance efforts might include controlling where your database resources are physically located. Azure regions are organized into groups called geographies. An Azure geography ensures that data residency, sovereignty, compliance, and resiliency requirements are honored within geographical and political boundaries. If your workloads are subject to data sovereignty or other compliance requirements, deploy your storage resources to regions that are in a compliant Azure geography. For more information, see [Azure geographies](#).

Next steps

[Review your data options](#)

Azure managed disk types

Article • 01/10/2024

Applies to: ✓ Linux VMs ✓ Windows VMs ✓ Flexible scale sets ✓ Uniform scale sets

Azure managed disks currently offers five disk types, each intended to address a specific customer scenario:

- Ultra disks
- Premium SSD v2
- Premium SSDs (solid-state drives)
- Standard SSDs
- Standard HDDs (hard disk drives)

Disk type comparison

The following table provides a comparison of the five disk types to help you decide which to use.

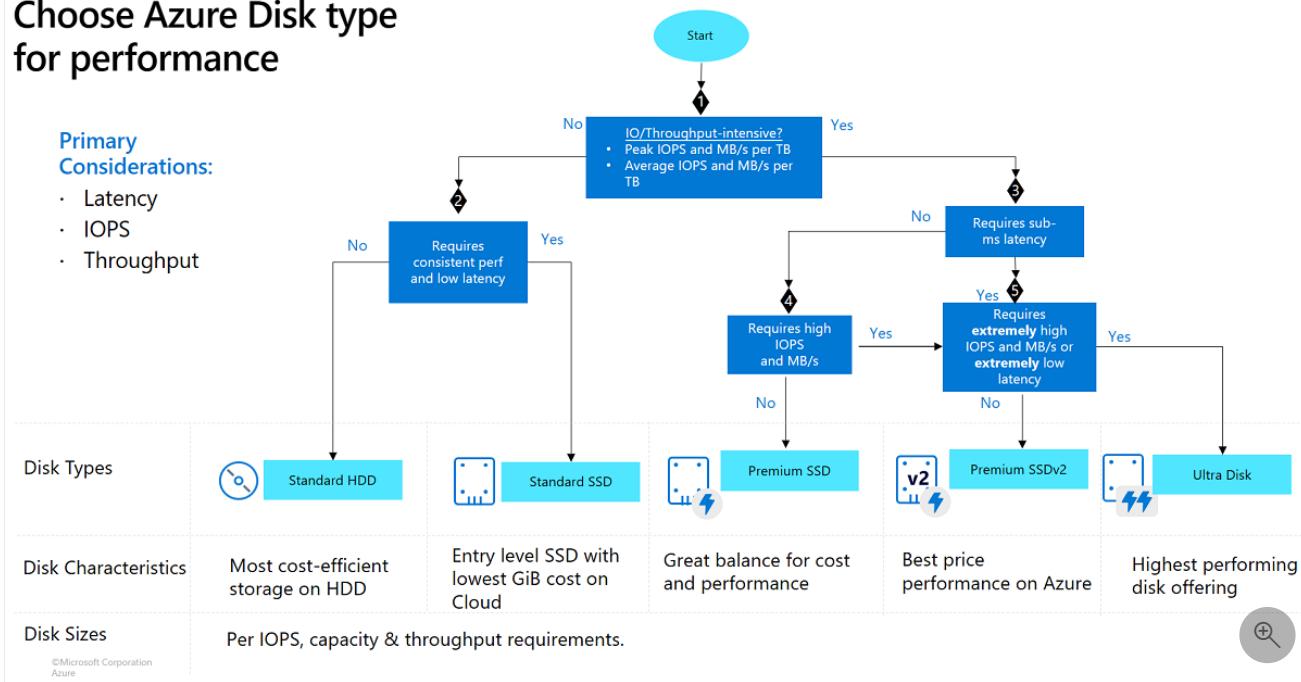
 Expand table

	Ultra disk	Premium SSD v2	Premium SSD	Standard SSD	Standard HDD
Disk type	SSD	SSD	SSD	SSD	HDD
Scenario	IO-intensive workloads such as SAP HANA, top tier databases (for example, SQL, Oracle), and other transaction-heavy workloads.	Production and performance-sensitive workloads that consistently require low latency and high IOPS and throughput	Production and performance sensitive workloads	Web servers, lightly used enterprise applications and dev/test	Backup, non-critical, infrequent access
Max disk size	65,536 GiB	65,536 GiB	32,767 GiB	32,767 GiB	32,767 GiB
Max throughput	4,000 MB/s	1,200 MB/s	900 MB/s	750 MB/s	500 MB/s
Max IOPS	160,000	80,000	20,000	6,000	2,000, 3,000*
Usable as OS Disk?	No	No	Yes	Yes	Yes

* Only applies to disks with performance plus (preview) enabled.

For more help deciding which disk type suits your needs, this decision tree should help with typical scenarios:

Choose Azure Disk type for performance



For a video that covers some high level differences for the different disk types, as well as some ways for determining what impacts your workload requirements, see [Block storage options with Azure Disk Storage and Elastic SAN](#).

Ultra disks

Azure ultra disks are the highest-performing storage option for Azure virtual machines (VMs). You can change the performance parameters of an ultra disk without having to restart your VMs. Ultra disks are suited for data-intensive workloads such as SAP HANA, top-tier databases, and transaction-heavy workloads.

Ultra disks must be used as data disks and can only be created as empty disks. You should use Premium solid-state drives (SSDs) as operating system (OS) disks.

Ultra disk size

Azure ultra disks offer up to 32-TiB per region per subscription by default, but ultra disks support higher capacity by request. To request an increase in capacity, request a quota increase or contact Azure Support.

The following table provides a comparison of disk sizes and performance caps to help you decide which to use.

[Expand table](#)

Disk Size (GiB)	IOPS Cap	Throughput Cap (MB/s)
4	1,200	300
8	2,400	600
16	4,800	1,200
32	9,600	2,400
64	19,200	4,000
128	38,400	4,000
256	76,800	4,000
512	153,600	4,000
1,024-65,536 (sizes in this range increasing in increments of 1 TiB)	160,000	4,000

Ultra disk performance

Ultra disks are designed to provide low sub millisecond latencies and provisioned IOPS and throughput 99.99% of the time. Ultra disks also feature a flexible performance configuration model that allows you to independently configure IOPS and throughput, before and after you provision the disk. Ultra disks come in several fixed sizes, ranging from 4 GiB up to 64 TiB.

Ultra disk IOPS

Ultra disks support IOPS limits of 300 IOPS/GiB, up to a maximum of 160,000 IOPS per disk. To achieve the target IOPS for the disk, ensure that the selected disk IOPS are less than the VM IOPS limit.

The current maximum limit for IOPS for a single VM in generally available sizes is 80,000. Ultra disks with greater IOPS can be used as shared disks to support multiple VMs.

The minimum guaranteed IOPS per disk are 1 IOPS/GiB, with an overall baseline minimum of 100 IOPS. For example, if you provisioned a 4-GiB ultra disk, the minimum IOPS for that disk is 100, instead of four.

For more information about IOPS, see [Virtual machine and disk performance](#).

Ultra disk throughput

The throughput limit of a single ultra disk is 256-kB/s for each provisioned IOPS, up to a maximum of 4000 MB/s per disk (where MB/s = 10^6 Bytes per second). The minimum guaranteed throughput per disk is 4kB/s for each provisioned IOPS, with an overall baseline minimum of 1 MB/s.

You can adjust ultra disk IOPS and throughput performance at runtime without detaching the disk from the virtual machine. After a performance resize operation has been issued on a disk, it can take up to an hour for the change to take effect. Up to four performance resize operations are permitted during a 24-hour window.

It's possible for a performance resize operation to fail because of a lack of performance bandwidth capacity.

Ultra disk limitations

Ultra disks can't be used as OS disks, they can only be created as empty data disks. Ultra disks also can't be used with some features and functionality, including disk export, changing disk type, VM images, availability sets, or Azure disk encryption. The size of an Ultra Disk can't be expanded without either deallocating the VM or detaching the disk. Azure Site Recovery doesn't support ultra disks. In addition, only un-cached reads and un-cached writes are supported. Snapshots for ultra disks are available but have additional limitations. See [Incremental snapshots of Premium SSD v2 and Ultra Disks](#) for details. Azure Backup support for VMs with Ultra Disks is currently in [public preview](#).

Ultra disks support a 4k physical sector size by default. A 512E sector size is available as a generally available offering with no sign-up required. While most applications are compatible with 4k sector sizes, some require 512 byte sector sizes. Oracle Database, for example, requires release 12.2 or later in order to support 4k native disks. For older versions of Oracle DB, 512 byte sector size is required.

The only infrastructure redundancy options currently available to ultra disks are availability zones. VMs using any other redundancy options cannot attach an ultra disk.

The following table outlines the regions ultra disks are available in, as well as their corresponding availability options.

Note

If a region in the following list lacks availability zones that support ultra disks, then a VM in that region must be deployed without infrastructure redundancy in order to attach an ultra disk.

 Expand table

Redundancy options	Regions
Single VMs	Australia Central Brazil South

Redundancy options	Regions
	Central India East Asia Germany West Central Korea Central Korea South UK West North Central US, South Central US, West US US Gov Arizona, US Gov Texas, US Gov Virginia
One availability zone	Brazil Southeast Poland Central UAE North
Two availability zones	South Africa North China North 3 France Central Qatar Central Switzerland North
Three availability zones	Australia East Canada Central North Europe, West Europe Japan East Southeast Asia Sweden Central UK South Central US, East US, East US 2, West US 2, West US 3

Not every VM size is available in every supported region with ultra disks. The following table lists VM series which are compatible with ultra disks.

[Expand table](#)

VM Type	Sizes	Description
General purpose	DSv3-series , DdsV4-series , Dsv4-series , Dasv4-series , Dsv5-series , DdsV5-series , Dasv5-series	Balanced CPU-to-memory ratio. Ideal for testing and development, small to medium databases, and low to medium traffic web servers.
Compute optimized	FSv2-series	High CPU-to-memory ratio. Good for medium traffic web servers, network appliances, batch processes, and application servers.
Memory optimized	ESv3-series , Easv4-series , Edsv4-series , Esv4-series , Esv5-series , Edsv5-series , Easv5-series , Ebsv5 series , EbdsV5 series , M-series , Mv2-series , Msv2/MdsV2-series	High memory-to-CPU ratio. Great for relational database servers, medium to large caches, and in-memory analytics.
Storage optimized	Lsv2-series , Lsv3-series , Lasv3-series	High disk throughput and IO ideal for Big Data, SQL, NoSQL databases, data warehousing and large transactional databases.
GPU optimized	NCv2-series , NCv3-series , NCasT4_v3-series , ND-series , NDv2-series , NVv3-series , NVv4-series , NVadsA10 v5-series	Specialized virtual machines targeted for heavy graphic rendering and video editing, as well as model training and inferencing (ND) with deep learning. Available with single or multiple GPUs.
Performance optimized	HB-series , HC-series , HBv2-series	The fastest and most powerful CPU virtual machines with optional high-throughput network interfaces (RDMA).

If you would like to start using ultra disks, see the article on [using Azure Ultra Disks](#).

Premium SSD v2

Premium SSD v2 offers higher performance than Premium SSDs while also generally being less costly. You can individually tweak the performance (capacity, throughput, and IOPS) of Premium SSD v2 disks at any time, allowing workloads to be cost efficient while meeting shifting performance needs. For example, a transaction-intensive database may need a large amount of IOPS at a small size, or a gaming application may need a large amount of IOPS but only during peak hours. Because of this, for most general purpose workloads, Premium SSD v2 can provide the best price performance.

Premium SSD v2 is suited for a broad range of workloads such as SQL server, Oracle, MariaDB, SAP, Cassandra, Mongo DB, big data/analytics, and gaming, on virtual machines or stateful containers.

Premium SSD v2 support a 4k physical sector size by default, but can be configured to use a 512E sector size as well. While most applications are compatible with 4k sector sizes, some require 512 byte sector sizes. Oracle Database, for example, requires release 12.2 or later in order to support 4k native disks.

Differences between Premium SSD and Premium SSD v2

Unlike Premium SSDs, Premium SSD v2 doesn't have dedicated sizes. You can set a Premium SSD v2 to any supported size you prefer, and make granular adjustments to the performance without downtime. Premium SSD v2 doesn't support host caching but, benefits significantly from lower latency, which addresses some of the same core problems host caching addresses. The ability to adjust IOPS, throughput, and size at any time also means you can avoid the maintenance overhead of having to stripe disks to meet your needs.

Premium SSD v2 limitations

- Premium SSD v2 disks can't be used as an OS disk.
- Currently, Premium SSD v2 disks can only be attached to zonal VMs.
- Encryption at host is supported on Premium SSD v2 disks with some limitations and in select regions. For more information, see [Encryption at host](#).
- Azure Disk Encryption (guest VM encryption via Bitlocker/DM-Crypt) isn't supported for VMs with Premium SSD v2 disks. We recommend you to use encryption at rest with platform-managed or customer-managed keys, which is supported for Premium SSD v2.
- Currently, Premium SSD v2 disks can't be attached to VMs in Availability Sets.
- Azure Site Recovery isn't supported for VMs with Premium SSD v2 disks.
- Azure Backup support for VMs with Premium SSD v2 disks is currently in [public preview](#).
- The size of a Premium SSD v2 can't be expanded without either deallocating the VM or detaching the disk.
- Premium SSDv2 does NOT support host caching.

Regional availability

Currently only available in the following regions:

- Australia East (Three availability zones)
- Brazil South (Two availability zones)
- Canada Central (Three availability zones)
- Central India (Three availability zones)
- Central US (One availability zone)
- China North 3 (Three availability zones)
- East Asia (Three availability zones)
- East US (Three availability zones)
- East US 2 (Three availability zones)
- France Central (Three availability zones)
- Germany West Central (Two availability zones)
- Israel Central (Two availability zones)
- Japan East (Three availability zones)
- Korea Central (Three availability zones)
- North Europe (Three availability zones)
- Norway East (Three availability zones)
- Poland Central (Three availability zones)
- South Africa North (Three availability zones)
- South Central US (Three availability zones)
- Southeast Asia (Two availability zones)
- Sweden Central (Three availability zones)
- Switzerland North (Three availability zones)
- UAE North (Three availability zones)
- UK South (Three availability zones)
- US Gov Virginia (Three availability zones)
- West Europe (Three availability zones)
- West US 2 (Three availability zones)

- West US 3 (Three availability zones)

To learn when support for particular regions was added, see either [Azure Updates](#) or [What's new for Azure Disk Storage](#).

Premium SSD v2 performance

Premium SSD v2 disks are designed to provide sub millisecond latencies and provisioned IOPS and throughput 99.9% of the time. With Premium SSD v2 disks, you can individually set the capacity, throughput, and IOPS of a disk based on your workload needs, providing you with more flexibility and reduced costs. Each of these values determines the cost of your disk.

Premium SSD v2 capacities

Premium SSD v2 capacities range from 1 GiB to 64 TiBs, in 1-GiB increments. You're billed on a per GiB ratio, see the [pricing page](#) for details.

Premium SSD v2 offers up to 32 TiBs per region per subscription by default, but supports higher capacity by request. To request an increase in capacity, request a quota increase or contact Azure Support.

Premium SSD v2 IOPS

All Premium SSD v2 disks have a baseline IOPS of 3000 that is free of charge. After 6 GiB, the maximum IOPS a disk can have increases at a rate of 500 per GiB, up to 80,000 IOPS. So an 8 GiB disk can have up to 4,000 IOPS, and a 10 GiB can have up to 5,000 IOPS. To be able to set 80,000 IOPS on a disk, that disk must have at least 160 GiBs. Increasing your IOPS beyond 3000 increases the price of your disk.

Premium SSD v2 throughput

All Premium SSD v2 disks have a baseline throughput of 125 MB/s that is free of charge. After 6 GiB, the maximum throughput that can be set increases by 0.25 MB/s per set IOPS. If a disk has 3,000 IOPS, the max throughput it can set is 750 MB/s. To raise the throughput for this disk beyond 750 MB/s, its IOPS must be increased. For example, if you increased the IOPS to 4,000, then the max throughput that can be set is 1,000. 1,200 MB/s is the maximum throughput supported for disks that have 5,000 IOPS or more. Increasing your throughput beyond 125 increases the price of your disk.

Premium SSD v2 Sector Sizes

Premium SSD v2 supports a 4k physical sector size by default. A 512E sector size is also supported. While most applications are compatible with 4k sector sizes, some require 512-byte sector sizes. Oracle Database, for example, requires release 12.2 or later in order to support 4k native disks.

Summary

The following table provides a comparison of disk capacities and performance maximums to help you decide which to use.

[Expand table](#)

Disk Size	Maximum available IOPS	Maximum available throughput (MB/s)
1 GiB-64 TiBs	3,000-80,000 (Increases by 500 IOPS per GiB)	125-1,200 (increases by 0.25 MB/s per set IOPS)

To deploy a Premium SSD v2, see [Deploy a Premium SSD v2](#).

Premium SSDs

Azure Premium SSDs deliver high-performance and low-latency disk support for virtual machines (VMs) with input/output (IO)-intensive workloads. To take advantage of the speed and performance of Premium SSDs, you can migrate existing VM disks to Premium SSDs. Premium SSDs are suitable for mission-critical production applications, but you can use them only with compatible VM series. Premium SSDs support the [512E sector size](#).

To learn more about individual Azure VM types and sizes for Windows or Linux, including size compatibility for premium storage, see [Sizes for virtual machines in Azure](#). You'll need to check each individual VM size article to determine if it's premium storage-compatible.

Premium SSD size

[Expand table](#)

Premium SSD sizes	P1	P2	P3	P4	P6	P10	P15	P20	P30	P40	P50	P60	P70	P80
Disk size in GiB	4	8	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,767
Base provisioned IOPS per disk	120	120	120	120	240	500	1,100	2,300	5,000	7,500	7,500	16,000	18,000	20,000
**Expanded provisioned IOPS per disk	N/A	8,000	16,000	20,000	20,000	20,000	20,000							
Base provisioned throughput per disk	25 MB/s	25 MB/s	25 MB/s	25 MB/s	50 MB/s	100 MB/s	125 MB/s	150 MB/s	200 MB/s	250 MB/s	250 MB/s	500 MB/s	750 MB/s	900 MB/s
**Expanded provisioned throughput per disk	N/A	300 MB/s	600 MB/s	900 MB/s	900 MB/s	900 MB/s	900 MB/s							
Max burst IOPS per disk	3,500	3,500	3,500	3,500	3,500	3,500	3,500	3,500	30,000*	30,000*	30,000*	30,000*	30,000*	30,000*
Max burst throughput per disk	170 MB/s	1,000 MB/s*												
Max burst duration	30 min	Unlimited*	Unlimited*	Unlimited*	Unlimited*	Unlimited*	Unlimited*							
Eligible for reservation	No	Yes, up to one year												

*Applies only to disks with on-demand bursting enabled.

** Only applies to disks with performance plus (preview) enabled.

Capacity, IOPS, and throughput are guaranteed when a premium storage disk is provisioned. For example, if you create a P50 disk, Azure provisions 4,095-GB storage capacity, 7,500 IOPS, and 250-MB/s throughput for that disk. Your application can use all or part of the capacity and performance. Premium SSDs are designed to provide the single-digit millisecond latencies, target IOPS, and throughput described in the preceding table 99.9% of the time.

Premium SSD bursting

Premium SSDs offer disk bursting, which provides better tolerance on unpredictable changes of IO patterns. Disk bursting is especially useful during OS disk boot and for applications with spiky traffic. To learn more about how bursting for Azure disks works, see [Disk-level bursting](#).

Premium SSD transactions

For Premium SSDs, each I/O operation less than or equal to 256 kB of throughput is considered a single I/O operation. I/O operations larger than 256 kB of throughput are considered multiple I/Os of size 256 kB.

Standard SSDs

Azure standard SSDs are optimized for workloads that need consistent performance at lower IOPS levels. They're an especially good choice for customers with varying workloads supported by on-premises hard disk drive (HDD) solutions. Compared to standard HDDs, standard SSDs deliver better availability, consistency, reliability, and latency. Standard SSDs are suitable for web servers, low IOPS application servers, lightly used enterprise applications, and non-production workloads. Like standard HDDs, standard SSDs are available on all Azure VMs. Standard SSDs support the [512E sector size](#).

Standard SSD size

[Expand table](#)

Standard SSD sizes	E1	E2	E3	E4	E6	E10	E15	E20	E30	E40	E50	E60	E70	E80
Disk size in GiB	4	8	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,767
Base IOPS per disk	Up to 500	Up to 500	Up to 2,000	Up to 4,000	Up to 6,000									
*Expanded IOPS per disk	N/A	Up to 1,500	Up to 3,000	Up to 6,000	Up to 6,000	Up to 6,000	Up to 6,000							
Base throughput per disk	Up to 60 MB/s	Up to 60 MB/s	Up to 400 MB/s	Up to 600 MB/s	Up to 750 MB/s									
*Expanded throughput per disk	N/A	Up to 150 MB/s	Up to 300 MB/s	Up to 600 MB/s	Up to 750 MB/s	Up to 750 MB/s	Up to 750 MB/s							
Max burst IOPS per disk	600	600	600	600	600	600	600	600	1000					
Max burst throughput per disk	150 MB/s	250 MB/s												
Max burst duration	30 min	30 min												

* Only applies to disks with performance plus (preview) enabled.

Standard SSDs are designed to provide single-digit millisecond latencies and the IOPS and throughput up to the limits described in the preceding table 99% of the time. Actual IOPS and throughput may vary sometimes depending on the traffic patterns. Standard SSDs provide more consistent performance than the HDD disks with the lower latency.

Standard SSD transactions

For standard SSDs, each I/O operation less than or equal to 256 kB of throughput is considered a single I/O operation. I/O operations larger than 256 kB of throughput are considered multiple I/Os of size 256 kB. These transactions incur a billable cost but, there's an hourly limit on the number of transactions that can incur a billable cost. If that hourly limit is reached, additional transactions during that hour no longer incur a cost. For details, see the [blog post](#).

Standard SSD Bursting

Standard SSDs offer disk bursting, which provides better tolerance for the unpredictable IO pattern changes. OS boot disks and applications prone to traffic spikes will both benefit from disk bursting. To learn more about how bursting for Azure disks works, see [Disk-level bursting](#).

Standard HDDs

Azure standard HDDs deliver reliable, low-cost disk support for VMs running latency-tolerant workloads. With standard storage, your data is stored on HDDs, and performance may vary more widely than that of SSD-based disks. Standard HDDs are designed to deliver write latencies of less than 10 ms and read latencies of less than 20 ms for most IO operations. Actual performance may vary depending on IO size and workload pattern, however. When working with VMs, you can use standard HDD disks for dev/test scenarios and less critical workloads. Standard HDDs are available in all Azure regions and can be used with all Azure VMs. Standard HDDs support the [512E sector size](#).

Standard HDD size

[Expand table](#)

Standard Disk Type	S4	S6	S10	S15	S20	S30	S40	S50	S60	S70	S80
Disk size in GiB	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,767
Base IOPS per disk	Up to 500	Up to 500	Up to 1,300	Up to 2,000	Up to 2,000	Up to 2,000					
*Expanded IOPS per disk	N/A	N/A	N/A	N/A	N/A	Up to 1,500	Up to 3,000				
Base throughput per disk	Up to 60 MB/s	Up to 60 MB/s	Up to 60 MB/s	Up to 300 MB/s	Up to 500 MB/s	Up to 500 MB/s					
*Expanded throughput per disk	N/A	N/A	N/A	N/A	N/A	Up to 150 MB/s	Up to 300 MB/s	Up to 500 MB/s			

* Only applies to disks with performance plus (preview) enabled.

Standard HDD Transactions

For Standard HDDs, each I/O operation is considered as a single transaction, whatever the I/O size. These transactions have a billing impact.

Billing

When using managed disks, the following billing considerations apply:

- Disk type
- Managed disk Size
- Snapshots
- Outbound data transfers
- Number of transactions

Managed disk size: Managed disks are billed according to their provisioned size. Azure maps the provisioned size (rounded up) to the nearest offered disk size. For details of the disk sizes offered, see the previous tables. Each disk maps to a supported provisioned disk-size offering and is billed accordingly. For example, if you provisioned a 200-GiB standard SSD, it maps to the disk size offer of E15 (256 GiB). Billing for any provisioned disk is prorated hourly by using the monthly price for the storage offering. For example, you provision an E10 disk and delete it after 20 hours of use. In this case, you're billed for the E10 offering prorated to 20 hours, regardless of the amount of data written to the disk.

Snapshots: Snapshots are billed based on the size used. For example, you create a snapshot of a managed disk with provisioned capacity of 64 GiB and actual used data size of 10 GiB. In this case, the snapshot is billed only for the used data size of 10 GiB.

For more information on snapshots, see the section on snapshots in the [managed disk overview](#).

Outbound data transfers: [Outbound data transfers](#) (data going out of Azure data centers) incur billing for bandwidth usage.

Transactions: You're billed for the number of transactions performed on a standard managed disk. For standard SSDs, each I/O operation less than or equal to 256 kB of throughput is considered a single I/O operation. I/O operations larger than 256 kB of

throughput are considered multiple I/Os of size 256 kB. For Standard HDDs, each IO operation is considered a single transaction, whatever the I/O size.

For detailed information on pricing for managed disks (including transaction costs), see [Managed Disks Pricing](#).

Ultra disks VM reservation fee

Azure VMs have the capability to indicate if they're compatible with ultra disks. An ultra disk-compatible VM allocates dedicated bandwidth capacity between the compute VM instance and the block storage scale unit to optimize the performance and reduce latency. When you add this capability on the VM, it results in a reservation charge. The reservation charge is only imposed if you enabled ultra disk capability on the VM without an attached ultra disk. When an ultra disk is attached to the ultra disk compatible VM, the reservation charge wouldn't be applied. This charge is per vCPU provisioned on the VM.

 Note

For constrained core VM sizes, the reservation fee is based on the actual number of vCPUs and not the constrained cores. For Standard_E32-8s_v3, the reservation fee will be based on 32 cores.

Refer to the [Azure Disks pricing page](#) for ultra disk pricing details.

Azure disk reservation

Disk reservation provides you with a discount on the advance purchase of one year's of disk storage, reducing your total cost. When you purchase a disk reservation, you select a specific disk SKU in a target region. For example, you may choose five P30 (1 TiB) Premium SSDs in the Central US region for a one year term. The disk reservation experience is similar to Azure reserved VM instances. You can bundle VM and Disk reservations to maximize your savings. For now, Azure Disks Reservation offers one year commitment plan for Premium SSD SKUs from P30 (1 TiB) to P80 (32 TiB) in all production regions. For more information about reserved disks pricing, see [Azure Disks pricing page](#).

Next steps

See [Managed Disks pricing](#) to get started.

Choose an Azure solution for data transfer

Article • 10/26/2023

This article provides an overview of some of the common Azure data transfer solutions. The article also links out to recommended options depending on the network bandwidth in your environment and the size of the data you intend to transfer.

Types of data movement

Data transfer can be offline or over the network connection. Choose your solution depending on your:

- **Data size** - Size of the data intended for transfer,
- **Transfer frequency** - One-time or periodic data ingestion, and
- **Network** – Bandwidth available for data transfer in your environment.

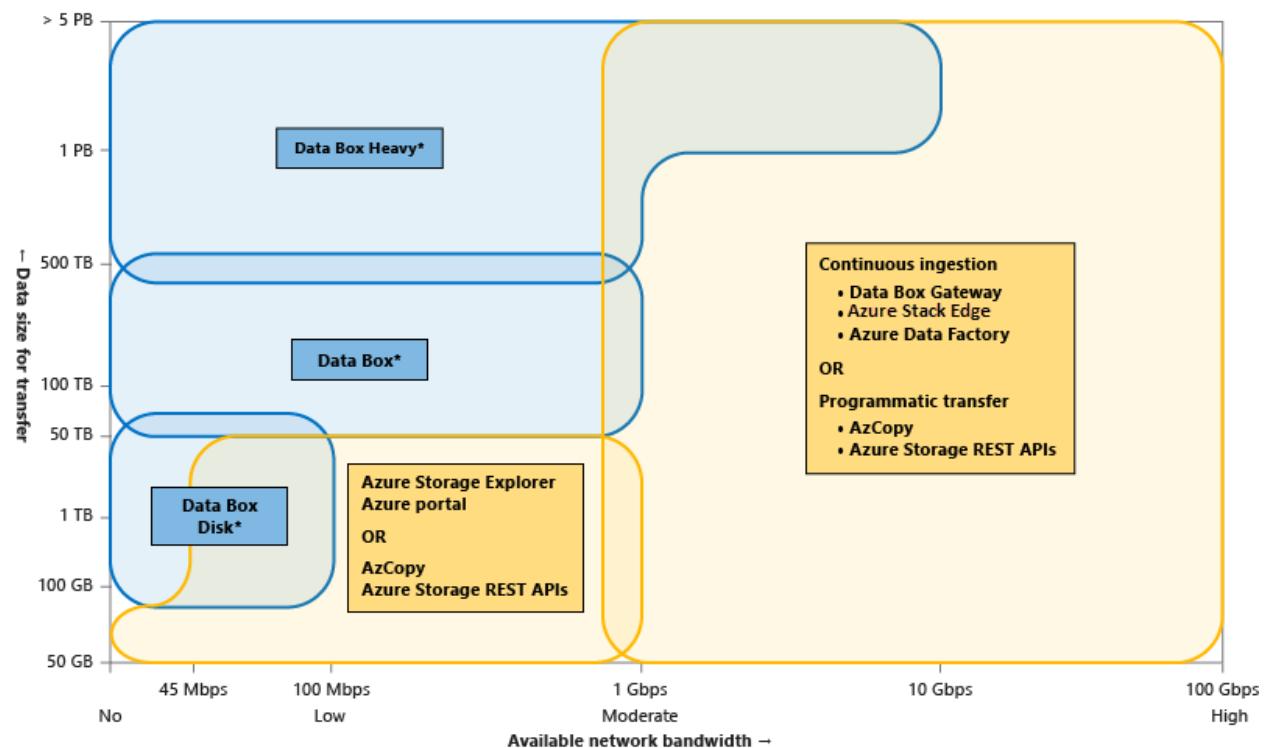
The data movement can be of the following types:

- **Offline transfer using shippable devices** - Use physical shippable devices when you want to do offline one-time bulk data transfer. This use case involves copying data to either a disk or specialized device, and then shipping it to a secure Microsoft facility where the data is uploaded. You can purchase and ship your own disks, or you order a Microsoft-supplied disk or device. Microsoft-supplied solutions for offline transfer include Azure [Data Box](#), [Data Box Disk](#), and [Data Box Heavy](#).
- **Network Transfer** - You transfer your data to Azure over your network connection. This transfer can be done in many ways.
 - **Hybrid migration service** - [Azure Storage Mover](#) is a new, fully managed migration service that enables you to migrate your files and folders to Azure Storage while minimizing downtime for your workload. Azure Storage Mover is a hybrid cloud service consisting of a cloud service component and an on-premises migration agent virtual machine (VM). Storage Mover is used for migration scenarios such as *lift-and-shift*, and for cloud migrations that you repeat occasionally.
 - **On-premises devices** - We supply you a physical or virtual device that resides in your datacenter and optimizes data transfer over the network. These devices also provide a local cache of frequently used files. The physical device is the

Azure Stack Edge and the virtual device is the Data Box Gateway. Both run permanently in your premises and connect to Azure over the network.

- **Graphical interface** - If you occasionally transfer just a few files and don't need to automate the data transfer, you can choose a graphical interface tool such as Azure Storage Explorer or a web-based exploration tool in Azure portal.
- **Scripted or programmatic transfer** - You can use optimized software tools that we provide or call our REST APIs/SDKs directly. The available scriptable tools are AzCopy, Azure PowerShell, and Azure CLI. For programmatic interface, use one of the SDKs for .NET, Java, Python, Node/JS, C++, Go, PHP or Ruby.
- **Managed data pipeline** - You can set up a cloud pipeline to regularly transfer files between several Azure services, on-premises or a combination of two. Use Azure Data Factory to set up and manage data pipelines, and move and transform data for analysis.

The following visual illustrates the guidelines to choose the various Azure data transfer tools depending upon the network bandwidth available for transfer, data size intended for transfer, and frequency of the transfer.



*The upper limits of the offline transfer devices - Data Box Disk, Data Box, and Data Box Heavy can be extended by placing multiple orders of a device type.

Selecting a data transfer solution

Answer the following questions to help select a data transfer solution:

- Is your available network bandwidth limited or nonexistent, and you want to transfer large datasets?

If yes, see: [Scenario 1: Transfer large datasets with no or low network bandwidth](#).

- Do you want to transfer large datasets over network and you have a moderate to high network bandwidth?

If yes, see: [Scenario 2: Transfer large datasets with moderate to high network bandwidth](#).

- Do you want to occasionally transfer just a few files over the network?

If yes, see [Scenario 3: Transfer small datasets with limited to moderate network bandwidth](#).

- Are you looking for point-in-time data transfer at regular intervals?

If yes, use the scripted/programmatic options outlined in [Scenario 4: Periodic data transfers](#).

- Are you looking for on-going, continuous data transfer?

If yes, use the options in [Scenario 4: Periodic data transfers](#).

Data transfer feature in Azure portal

You can also provide information specific to your scenario and review a list of optimal data transfer solutions. To view the list, navigate to your Azure Storage account within the Azure portal and select the **Data transfer** feature. After providing the network bandwidth in your environment, the size of the data you want to transfer, and the frequency of data transfer, you're shown a list of solutions corresponding to the information that you have provided.

Next steps

- Get an introduction to [Azure Storage Explorer](#).
- Read an overview of [AzCopy](#).
- Quickstart: [Upload, download, and list blobs with PowerShell](#)
- Quickstart: [Create, download, and list blobs with Azure CLI](#)
- Learn about:

- [Azure Storage Mover](#), a hybrid migration service.
- [Cloud migration using Azure Storage Mover](#).
- Learn about:
 - [Azure Data Box](#), [Azure Data Box Disk](#), and [Azure Data Box Heavy](#) for offline transfers.
 - [Azure Data Box Gateway](#) and [Azure Stack Edge](#) for online transfers.
- [Learn about Azure Data Factory](#).
- Use the REST APIs to transfer data
 - [In .NET](#)
 - [In Java](#)

StorSimple 8000 series: a hybrid cloud storage solution

Article • 07/10/2023

⊗ Caution

Action Required: StorSimple Data Manager, StorSimple Device Manager, StorSimple 1200, and StorSimple 8000 have reached their end of support. We're no longer updating this content. Check the Microsoft Product Lifecycle for information about how this product, service, technology, or API is supported. StorSimple management services have been decommissioned and removed from the Azure platform.

The following resources are available to help you migrate backup files or to copy live data to your own environment, and to provide documentation to decommission a StorSimple appliance.

[] Expand table

Resource	Description
Azure StorSimple 8000 Series Copy Utility	Microsoft is providing a read-only data copy utility to recover and migrate your backup files from StorSimple cloud snapshots. The StorSimple 8000 Series Copy Utility is designed to run in your environment. You can install and configure the Utility, and then use your Service Encryption Key to authenticate and download your metadata from the cloud.
Azure StorSimple 8000 Series Copy Utility documentation	Instructions for use of the Copy Utility.
StorSimple archived documentation	Archived StorSimple articles from Microsoft technical documentation.

Copy data and then decommission your appliance

Use the following steps to copy data to your environment and then decommission your StorSimple 8000 appliance. If your data has already been migrated to your own

environment, you can proceed with decommissioning your appliance.

Step 1: Copy backup files or live data to your own environment.

- **Backup files.** If you have backup files, use the Azure StorSimple 8000 Series Copy Utility to migrate backup files to your environment. For more information, see [Copy Utility documentation](#).
- **Live data.** If you have live data to copy, you can access and copy live data to your environment via iSCSI.

Step 2: Decommission your device.

After you complete your data migration, use the following steps to decommission the device. Before you decommission your device, make sure to copy all data from your appliance, using either local host copy operations or using the Utility.

Decommission operations can't be undone. We recommend that you complete your data migration as soon as possible.

1. Disconnect the iSCSI session on the host – that is, the iSCSI Initiators.
2. Reset the device to factory default:

This procedure describes how to reset your Azure StorSimple device to factory default settings using Windows PowerShell for StorSimple. Resetting a device removes all data and settings from the entire cluster by default.

Use the following steps to reset your device to factory default settings:

- a. Access the device through its serial console. Check the banner message to ensure that you're connected to the **Active controller**.
- b. In the serial console menu, choose option 1: **Log in with full access**.
- c. At the prompt, type the following command to reset the entire cluster, removing all data, metadata, and controller settings:

Azure PowerShell

Reset-HcsFactoryDefault

To instead reset a single controller, use the [Reset-HcsFactoryDefault cmdlet](#) with the `-scope` parameter.

The system reboots multiple times. You're notified when the reset has successfully completed. Depending on the system model, it can take 45-60 minutes for an 8100 device and 60-90 minutes for an 8600 to finish this process.

Step 3: Shut down the device.

This section explains how to shut down a running or a failed StorSimple device from a remote computer. A device is turned off after both the device controllers are shut down. A device shutdown is complete when the device is physically moved or is taken out of service.

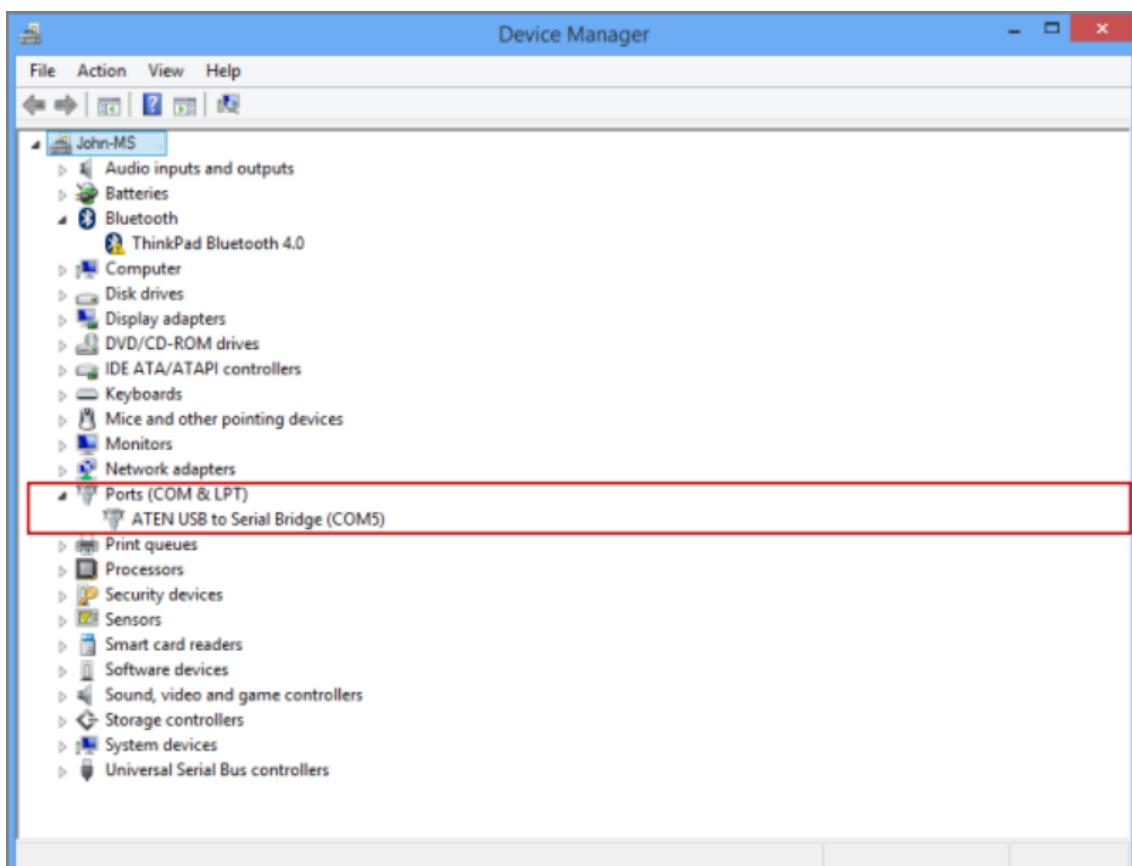
Step 3a: Use the following steps to identify and shut down the passive controller on your device. Perform this operation in Windows PowerShell for StorSimple.

1. Access the device via the serial console or a telnet session from a remote computer. To connect to Controller 0 or Controller 1, follow these steps to use PuTTY to connect to the device serial console.

To connect to Windows PowerShell for StorSimple, you need to use terminal emulation software such as PuTTY. You can use PuTTY when you access the device directly through the serial console or by opening a telnet session from a remote computer.

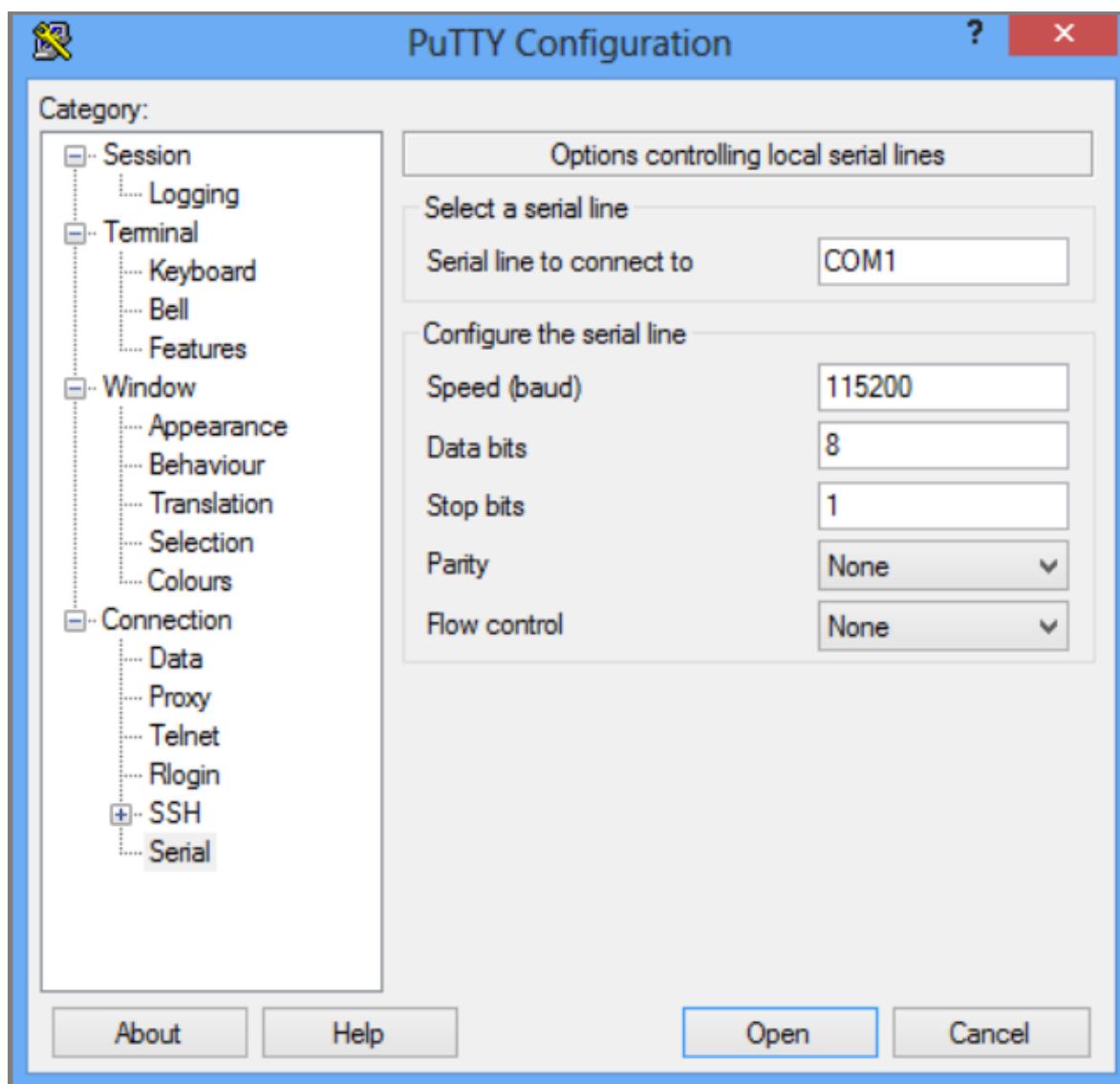
- a. To connect through the serial console, connect your serial cable to the device, directly or through a USB-serial adapter.
- b. Open Control Panel and then open Device Manager.

- c. Identify the COM port as shown in the following illustration.



- d. Start PuTTY.
- e. In the right pane, change **Connection type** to **Serial**.
- f. In the right pane, specify the appropriate COM port. Make sure that the serial configuration parameters are set as follows:
- Speed: 115200
 - Data bits: 8
 - Stop bits: 1
 - Parity: None
 - Flow control: None

These settings are shown in the following illustration.



If the default flow control setting doesn't work, try setting Flow control to XON/XOFF.

g. Select **Open** to start a serial session.

2. In the serial console menu, select option 1: **Log in with full access**.

3. In the banner message, make a note of the controller you're connected to, Controller 0 or Controller 1, and whether it's the active or the passive (standby) controller.
 - a. Run the following command to shut down a single controller:

```
Azure PowerShell  
  
Stop-HcsController
```

This shuts down the controller you're connected to. When you stop the active controller, the device fails over to the passive controller.

- b. To restart a controller, at the prompt, run the following command:

```
Azure PowerShell  
  
Restart-HcsController
```

This restarts the controller you're connected to. When you restart the active controller, it fails over to the passive controller before the restart.

Step 3b: Repeat the previous step to shut down the active controller.

Step 3c: You must now look at the back plane of the device. After the two controllers are shut down, the status LEDs on both the controllers should be blinking red. To turn off the device completely at this time, flip the power switches on both Power and Cooling Modules (PCMs) to the OFF position. This turns off the device.

Create a support request

Use the following steps to create a support ticket for StorSimple data copy, data migration, and device decommission operations.

1. In Azure portal, type **help** in the search bar and then select **Help + Support**.

The screenshot shows the 'Create a resource' page in the Azure portal. On the left, there's a sidebar with 'Get Started' and 'Recently created' sections, followed by a 'Categories' section with various service links. The main area is titled 'Services' and contains a list of items. One item, 'Help + support', is highlighted with a red box. Other items include 'Load balancing - help me choose', 'Multi Approval Process for Apps', 'HelpDesk as a Service', 'Multi Approval Process', 'Zoho Desk', 'GoAnywhere Managed File Transfer for Windows', 'ALVAO Service Desk', 'Intelligent IT Service Desk Platform for MS Teams', 'Gitlab Runner Helper packaged by Bitnami', 'Get support in Partner Center - Partner Center', 'Help users protect files by using Azure RMS - AIP', 'Which support portal should I use? - Partner Center', 'Discovery Solution - REST API (Azure Help)', 'Find commands and Get-Help in Windows PowerShell - Training', 'Help identity issue with automatic checks - adding domains', 'Azure Active Directory' (with several user and group entries), and 'Continue searching in Azure Active Directory'. At the bottom, it says 'Searching 2 of 67 subscriptions. Change' and 'Give feedback'.

2. On the **Help + Support** page, select **Create a support request**.

The screenshot shows the 'Help + support' page. At the top, there's a search bar and two buttons: 'Create a support request' (highlighted with a red box) and 'Choose the right support plan'. Below the search bar, there's a 'Overview' section and a 'Support' section with links for 'All support requests', 'Support Plans', and 'Service Health'. To the right, there's a 'Health Events' section showing a warning message: 'West Europe - Networking issues - Applying mitigation'. It details the status as 'Active' and the last update as 'We have concluded our investigation, confirmed that Virtual Machines outside of West Europe should not have been impacted by this event. We apologize for any inconvenience caused by the notification sent during the event noting impact to Virtual Machines in...'. There's also a 'Create an alert rule' button.

3. On the **New support request** page, provide the required information:

- Provide a brief **Summary** of the issue.
- Specify **Technical** as the **Issue type**.
- Specify the affected **Subscription**.
- Specify **All services**. You must specify **All services** because **StorSimple Manager Service** is no longer available.
- For **Service type**, specify **Azure StorSimple 8000 Series**.

- For Problem type, Specify StorSimple Migration Utility.
- To continue, select **Next**.

Home > Help + support >

New support request

1. Problem description 2. Recommended solution 3. Additional details 4. Review + create

Tell us your issue, and we'll help you resolve it.

Provide information about your billing, subscription, quota management, or technical issue (including requests for technical advice).

Summary *	Copy Utility Issue	✓
Issue type *	Technical	▼
Subscription *	▼	
Can't find your subscription? Show more ⓘ		
Service	<input type="radio"/> My services <input checked="" type="radio"/> All services	
Service type *	Azure StorSimple 8000 Series	
Problem type *	StorSimple Migration Utility	
Azure Active Directory Privileged Identity Management - Advisory <p>Active: An Azure service issue (Tracking ID FMJ8-PS0) impacted resources in your subscription.</p> <p>Engineers are investigating an alert for Azure Active Directory Privileged Identity Management. More information will be provided in 60 minutes or as events warrant.</p> <p>Please use the links below to view the latest information about the issue in Azure Service Health and be sure to create a Service Health alert ☰ to be automatically notified should an Azure event affect you.</p> <p>See latest update RCA In Progress</p>		
Was this helpful? Yes No		
Next		

4. If the Solutions page appears, select **Return to support request** and then select **Next**.
5. On the Additional details tab, provide additional details and contact information:
 - Specify the time when the problem started, a description, and upload relevant files, if applicable.
 - Specify **Yes** or **No** for Advanced diagnostic information collection.
 - Your support plan will be generated based on your subscription. Specify severity, your preferred contact method, and language.
 - Specify **Contact information:** First name, Last name, Email, Phone, and Country/region.

- To continue, select **Next**.

Home > Help + support >

New support request

1. Problem description 2. Recommended solution **3. Additional details** 4. Review + create

Tell us a little more information.

Providing detailed, accurate information helps us resolve your issue faster.

Problem details

When did the problem start? *

(UTC-05:00) Eastern Time (US & Canada)

Not sure, use current time

Description *

File upload

Advanced diagnostic information

To enable faster resolution, we recommend allowing Microsoft support to access your Azure resources to collect advanced diagnostic information. Access is read-only and is removed when your support request is closed. [Learn more](#)

Allow collection of advanced diagnostic information? * Yes (Recommended) No

Support method

Support plan

Severity *

Preferred contact method * Email
A Support engineer will contact you over email.

Phone
A Support engineer will contact you over the phone.

Your availability

Business Hours

Support language *

Contact info

First name *

- On the **Review + create** tab, review the summary of your case. To continue, select **Create**.

New support request

1. Problem description 2. Recommended solution 3. Additional details **4. Review + create**

Review the information you provided before creating your support request.

Basics

Issue type	Technical
Subscription	
Service type	Azure StorSimple 8000 Series
Problem type	StorSimple Migration Utility
Summary	Copy Utility Issue

Terms, conditions, and privacy policy

By clicking "Create" you accept the [terms and conditions](#).

View our [privacy policy](#).

Details

When did the problem start?	Thu, Jun 22, 2023, 5:05 PM (UTC-05:00) Eastern Time (US & Canada)
Description	Issues with StorSimple Data Copy Utility
Advanced diagnostic information	Yes

Support method

Severity	C - Minimal impact
Support plan	
Your availability	Business Hours
Support language	English
Contact method	Email

Contact info

Contact name	
Email	
Country/region	United States

[Previous](#)

Create

Microsoft Support will use this information to reach out to you for additional details and diagnosis. A Support engineer will contact you as soon as possible to proceed with your request.

Next steps

- StorSimple 8000 series copy utility documentation .

Data transfer for large datasets with low or no network bandwidth

Article • 05/03/2023

This article provides an overview of the data transfer solutions when you have limited to no network bandwidth in your environment and you are planning to transfer large data sets. The article also describes the recommended data transfer options and the respective key capability matrix for this scenario.

To understand an overview of all the available data transfer options, go to [Choose an Azure data transfer solution](#).

Offline transfer or network transfer

Large datasets imply that you have few TBs to few PBs of data. You have limited to no network bandwidth, your network is slow, or it is unreliable. Also:

- You are limited by costs of network transfer from your Internet Service Providers (ISPs).
- Security or organizational policies do not allow outbound connections when dealing with sensitive data.

In all the above instances, use a physical device to do a one-time bulk data transfer. Choose from Data Box Disk, Data Box, Data Box Heavy devices which are supplied by Microsoft, or Import/Export using your own disks.

To confirm whether a physical device is the right option, use the following table. It shows the projected time for network data transfer, for various available bandwidths (assuming 90% utilization). If network transfer is projected to be too slow, you should use a physical device.

Data size ↓ Network bandwidth →	45 Mbps (T3)	100 Mbps	1 Gbps	10 Gbps
1 TB	2 days	1 day	3 hours	15 minutes
10 TB	23 days	11 days	1 day	3 hours
35 TB	82 days	37 days	4 days	9 hours
80 TB	187 days	84 days	8 days	20 hours
100 TB	234 days	105 days	11 days	1 day
200 TB	1 year	211 days	21 days	2 days
500 TB	3 years	1 year	53 days	5 days
1 PB	7 years	3 years	108 days	11 days
2 PB	13 years	6 years	216 days	22 days
5 PB	33 years	15 years	1 year	54 days

Key:
Use a Data Box Disk instead
Use a Data Box instead
Use a Data Box Heavy instead
Use the network

Recommended options

The options available in this scenario are devices for Azure Data Box offline transfer or Azure Import/Export.

- **Azure Data Box family for offline transfers** – Use devices from Microsoft-supplied Data Box devices to move large amounts of data to Azure when you're limited by time, network availability, or costs. Copy on-premises data using tools such as Robocopy. Depending on the data size intended for transfer, you can choose from Data Box Disk, Data Box, or Data Box Heavy.
- **Azure Import/Export** – Use Azure Import/Export service by shipping your own disk drives to securely import large amounts of data to Azure Blob storage and Azure Files. This service can also be used to transfer data from Azure Blob storage to disk drives and ship to your on-premises sites.

Comparison of key capabilities

The following table summarizes the differences in key capabilities.

	Data Box Disk	Data Box	Data Box Heavy	Import/Export
Data size	Up to 35 TBs	Up to 80 TBs per device	Up to 800 TB per device	Variable
Data type	Azure Blobs Azure Files*	Azure Blobs Azure Files	Azure Blobs Azure Files	Azure Blobs Azure Files
Form factor	5 SSDs per order	1 X 50-lbs. desktop-sized device per order	1 X ~500-lbs. large device per order	Up to 10 HDDs/SSDs per order

	Data Box Disk	Data Box	Data Box Heavy	Import/Export
Initial setup time	Low (15 mins)	Low to moderate (<30 mins)	Moderate (1-2 hours)	Moderate to difficult (variable)
Send data to Azure	Yes	Yes	Yes	Yes
Export data from Azure	No	No	No	Yes
Encryption	AES 128-bit	AES 256-bit	AES 256-bit	AES 128-bit
Hardware	Microsoft supplied	Microsoft supplied	Microsoft supplied	Customer supplied
Network interface	USB 3.1/SATA	RJ 45, SFP+	RJ45, QSFP+	SATA II/SATA III
Partner integration	Some	High ↗	High ↗	Some
Shipping	Microsoft managed	Microsoft managed	Microsoft managed	Customer managed
Use when data moves	Within a commerce boundary	Within a commerce boundary	Within a commerce boundary	Across geographic boundaries, e.g. US to EU
Pricing	Pricing ↗	Pricing ↗	Pricing ↗	Pricing ↗

* Data Box Disk does not support Large File Shares and does not preserve file metadata.

Next steps

- Understand how to
 - [Transfer data with Data Box Disk.](#)
 - [Transfer data with Data Box.](#)
 - [Transfer data with Import/Export.](#)

Data transfer for large datasets with moderate to high network bandwidth

Article • 05/03/2023

This article provides an overview of the data transfer solutions when you have moderate to high network bandwidth in your environment and you are planning to transfer large datasets. The article also describes the recommended data transfer options and the respective key capability matrix for this scenario.

To understand an overview of all the available data transfer options, go to [Choose an Azure data transfer solution](#).

Scenario description

Large datasets refer to data sizes in the order of TBs to PBs. Moderate to high network bandwidth refers to 100 Mbps to 10 Gbps.

Recommended options

The options recommended in this scenario depend on whether you have moderate network bandwidth or high network bandwidth.

Moderate network bandwidth (100 Mbps - 1 Gbps)

With moderate network bandwidth, you need to project the time for data transfer over the network.

Use the following table to estimate the time and based on that, choose between an offline transfer or over the network transfer. The table shows the projected time for network data transfer, for various available network bandwidths (assuming 90% utilization).

Data size ↓ Network bandwidth →	45 Mbps (T3)	100 Mbps	1 Gbps	10 Gbps
1 TB	2 days	1 day	3 hours	15 minutes
10 TB	23 days	11 days	1 day	3 hours
35 TB	82 days	37 days	4 days	9 hours
80 TB	187 days	84 days	8 days	20 hours
100 TB	234 days	105 days	11 days	1 day
200 TB	1 year	211 days	21 days	2 days
500 TB	3 years	1 year	53 days	5 days
1 PB	7 years	3 years	108 days	11 days
2 PB	13 years	6 years	216 days	22 days
5 PB	33 years	15 years	1 year	54 days

Key:
Use a Data Box Disk instead
Use a Data Box instead
Use a Data Box Heavy instead
Use the network

- If the network transfer is projected to be too slow, you should use a physical device. The recommended options in this case are the offline transfer devices from Azure Data Box family or Azure Import/Export using your own disks.
 - **Azure Data Box family for offline transfers** – Use devices from Microsoft-supplied Data Box devices to move large amounts of data to Azure when you're limited by time, network availability, or costs. Copy on-premises data using tools such as Robocopy. Depending on the data size intended for transfer, you can choose from Data Box Disk, Data Box, or Data Box Heavy.
 - **Azure Import/Export** – Use Azure Import/Export service by shipping your own disk drives to securely import large amounts of data to Azure Blob storage and Azure Files. This service can also be used to transfer data from Azure Blob storage to disk drives and ship to your on-premises sites.
- If the network transfer is projected to be reasonable, then you can use any of the following tools detailed in [High network bandwidth](#).

High network bandwidth (1 Gbps - 100 Gbps)

If the available network bandwidth is high, use one of the following tools.

- **AzCopy** - Use this command-line tool to easily copy data to and from Azure Blobs, Files, and Table storage with optimal performance. AzCopy supports concurrency and parallelism, and the ability to resume copy operations when interrupted.
- **Azure Storage REST APIs/SDKs** – When building an application, you can develop the application against Azure Storage REST APIs and use the Azure SDKs offered in multiple languages.
- **Azure Data Box family for online transfers** – Azure Stack Edge and Data Box Gateway are online network devices that can move data into and out of Azure. Use Azure Stack Edge physical device when there is a simultaneous need for continuous ingestion and pre-processing of the data prior to upload. Data Box

Gateway is a virtual version of the device with the same data transfer capabilities.

In each case, the data transfer is managed by the device.

- **Azure Data Factory** – Data Factory should be used to scale out a transfer operation, and if there is a need for orchestration and enterprise grade monitoring capabilities. Use Data Factory to regularly transfer files between several Azure services, on-premises, or a combination of the two. With Data Factory, you can create and schedule data-driven workflows (called pipelines) that ingest data from disparate data stores and automate data movement and data transformation.

Comparison of key capabilities

The following tables summarize the differences in key capabilities for the recommended options.

Moderate network bandwidth

If using offline data transfer, use the following table to understand the differences in key capabilities.

	Data Box Disk	Data Box	Data Box Heavy	Import/Export
Data size	Up to 35 TBs	Up to 80 TBs per device	Up to 800 TB per device	Variable
Data type	Azure Blobs Azure Files*	Azure Blobs Azure Files	Azure Blobs Azure Files	Azure Blobs Azure Files
Form factor	5 SSDs per order	1 X 50-lbs. desktop-sized device per order	1 X ~500-lbs. large device per order	Up to 10 HDDs/SSDs per order
Initial setup time	Low (15 mins)	Low to moderate (<30 mins)	Moderate (1-2 hours)	Moderate to difficult (variable)
Send data to Azure	Yes	Yes	Yes	Yes
Export data from Azure	No	No	No	Yes
Encryption	AES 128-bit	AES 256-bit	AES 256-bit	AES 128-bit
Hardware	Microsoft supplied	Microsoft supplied	Microsoft supplied	Customer supplied

	Data Box Disk	Data Box	Data Box Heavy	Import/Export
Network interface	USB 3.1/SATA	RJ 45, SFP+	RJ45, QSFP+	SATA II/SATA III
Partner integration	Some	High ↗	High ↗	Some
Shipping	Microsoft managed	Microsoft managed	Microsoft managed	Customer managed
Use when data moves	Within a commerce boundary	Within a commerce boundary	Within a commerce boundary	Across geographic boundaries, e.g. US to EU
Pricing	Pricing ↗	Pricing ↗	Pricing ↗	Pricing ↗

* Data Box Disk does not support Large File Shares and does not preserve file metadata

If using online data transfer, use the table in the following section for high network bandwidth.

High network bandwidth

	Tools AzCopy, Azure PowerShell, Azure CLI	Azure Storage REST APIs, SDKs	Data Box Gateway or Azure Stack Edge	Azure Data Factory
Data type	Azure Blobs, Azure Files, Azure Tables	Azure Blobs, Azure Files, Azure Tables	Azure Blobs, Azure Files	Supports 70+ data connectors for data stores and formats
Form factor	Command-line tools	Programmatic interface	Microsoft supplies a virtual or physical device	Service in Azure portal
Initial one-time setup	Easy	Moderate	Easy (<30 minutes) to moderate (1-2 hours)	Extensive
Data pre-processing	No	No	Yes (With Edge compute)	Yes
Transfer from other clouds	No	No	No	Yes

	Tools AzCopy, Azure PowerShell, Azure CLI	Azure Storage REST APIs, SDKs	Data Box Gateway or Azure Stack Edge	Azure Data Factory
User type	IT Pro or dev	Dev	IT Pro	IT Pro
Pricing	Free, data egress charges apply	Free, data egress charges apply	Azure Stack Edge pricing ↗ Data Box Gateway pricing ↗	Pricing ↗

Next steps

- Learn how to transfer data with Import/Export.
- Understand how to:
 - Transfer data with Data Box Disk.
 - Transfer data with Data Box.
 - Transfer data with AzCopy.
 - Transfer data with Data Box Gateway.
 - Transform data with Azure Stack Edge before sending to Azure.
- Learn how to transfer data with Azure Data Factory.
- Use the REST APIs to transfer data:
 - In .NET
 - In Java

Data transfer for small datasets with low to moderate network bandwidth

Article • 12/02/2022

This article provides an overview of the data transfer solutions when you have low to moderate network bandwidth in your environment and you are planning to transfer small datasets. The article also describes the recommended data transfer options and the respective key capability matrix for this scenario.

To understand an overview of all the available data transfer options, go to [Choose an Azure data transfer solution](#).

Scenario description

Small datasets refer to data sizes in the order of GBs to a few TBs. Low to moderate network bandwidth implies 45 Mbps (T3 connection in datacenter) to 1 Gbps.

- If you are transferring only a handful of files and you don't need to automate data transfer, consider the tools with a graphical interface.
- If you are comfortable with system administration, consider command line or programmatic/scripting tools.

Recommended options

The options recommended in this scenario are:

- **Graphical interface tools** such as Azure Storage Explorer and Azure Storage in Azure portal. These provide an easy way to view your data and quickly transfer a few files.
 - **Azure Storage Explorer** - This cross-platform tool 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** titleSuffix: Azure Storage in Azure portal provides a web-based interface to explore files and upload 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.

- Scripting/programmatic tools such as AzCopy/PowerShell/Azure CLI and Azure Storage REST APIs.
 - **AzCopy** - Use this command-line tool to easily copy data to and from Azure Blobs, Files, and Table storage with optimal performance. AzCopy supports concurrency and parallelism, and the ability to resume copy operations when interrupted.
 - **Azure PowerShell** - For users comfortable with system administration, use the Azure Storage module in Azure PowerShell to transfer data.
 - **Azure CLI** - Use this cross-platform tool to manage Azure services and upload data to Azure Storage.
 - **Azure Storage REST APIs/SDKs** – When building an application, you can develop the application against Azure Storage REST APIs/SDKs and use the Azure client libraries offered in multiple languages.

Comparison of key capabilities

The following table summarizes the differences in key capabilities.

Feature	Azure Storage Explorer	Azure portal	AzCopy Azure PowerShell Azure CLI	Azure Storage REST APIs or SDKs
Availability	Download and install Standalone tool	Web-based exploration tools in Azure portal	Command line tool	Programmable interfaces in .NET, Java, Python, JavaScript, C++, Go, Ruby and PHP
Graphical interface	Yes	Yes	No	No
Supported platforms	Windows, Mac, Linux	Web-based	Windows, Mac, Linux	All platforms
Allowed Blob storage operations for blobs and folders	Upload Download Manage	Upload Download Manage	Upload Download Manage	Yes, customizable
Allowed Data Lake Gen1 storage operations for files and folders	Upload Download Manage	No	Upload Download Manage	No

Feature	Azure Storage Explorer	Azure portal	AzCopy	Azure Storage REST APIs or SDKs
			PowerShell	
			Azure CLI	
Allowed File storage operations for files and directories	Upload Download Manage	Upload Download Manage	Upload Download Manage	Yes, customizable
Allowed Table storage operations for tables	Manage	No	Table support in AzCopy v7	Yes, customizable
Allowed Queue storage	Manage	No	No	Yes, is customizable

Next steps

- Learn how to [transfer data with Azure Storage Explorer](#).
- [Transfer data with AzCopy](#)

Solutions for periodic data transfer

Article • 12/02/2022

This article provides an overview of the data transfer solutions when you are transferring data periodically. Periodic data transfer over the network can be categorized as recurring at regular intervals or continuous data movement. The article also describes the recommended data transfer options and the respective key capability matrix for this scenario.

To understand an overview of all the available data transfer options, go to [Choose an Azure data transfer solution](#).

Recommended options

The recommended options for periodic data transfer fall into two categories depending on whether the transfer is recurring or continuous.

- **Scripted/programmatic tools** – For data transfer that occurs at regular intervals, use the scripted and programmatic tools such as AzCopy and Azure Storage REST APIs. These tools are targeted towards IT professionals and developers.
 - **AzCopy** - Use this command-line tool to easily copy data to and from Azure Blobs, Files, and Table storage with optimal performance. AzCopy supports concurrency and parallelism, and the ability to resume copy operations when interrupted.
 - **Azure Storage REST APIs/SDKs** – When building an application, you can develop the application against Azure Storage REST APIs and use the Azure SDKs offered in multiple languages. The REST APIs can also leverage the Azure Storage Data Movement Library designed especially for the high-performance copying of data to and from Azure.
- **Continuous data ingestion tools** – For continuous, ongoing data ingestion, you can select one of the following options.
 - **Object replication** - Object replication asynchronously copies block blobs between containers in a source and destination storage account. Use object replication as a solution to keep containers in two different storage accounts in sync.
 - **Azure Data Factory** – Data Factory should be used to scale out a transfer operation, and if there is a need for orchestration and enterprise grade monitoring capabilities. Use Azure Data Factory to set up a cloud pipeline that regularly transfers files between several Azure services, on-premises, or a

combination of the two. Azure Data Factory lets you orchestrate data-driven workflows that ingest data from disparate data stores and automate data movement and data transformation.

- **Azure Data Box family for online transfers** - Data Box Edge and Data Box Gateway are online network devices that can move data into and out of Azure. Data Box Edge uses artificial intelligence (AI)-enabled Edge compute to pre-process data before upload. Data Box Gateway is a virtual version of the device with the same data transfer capabilities.

Data Box online transfer device or Azure Data Factory are set up by IT professionals and can transparently automate data transfer.

Comparison of key capabilities

The following table summarizes the differences in key capabilities.

Scripted/Programmatic network data transfer

Capability	AzCopy	Azure Storage REST APIs
Form factor	Command-line tool from Microsoft	Customers develop against Storage REST APIs using Azure client libraries
Initial one-time setup	Minimal	Moderate, variable development effort
Data Format	Azure Blobs, Azure Files, Azure Tables	Azure Blobs, Azure Files, Azure Tables
Performance	Already optimized	Optimize as you develop
Pricing	Free, data egress charges apply	Free, data egress charges apply

Continuous data ingestion over network

Feature	Data Box Gateway	Data Box Edge	Azure Data Factory
Form factor	Virtual device	Physical device	Service in Azure portal, agent on-premises
Hardware	Your hypervisor	Supplied by Microsoft	NA

Feature	Data Box Gateway	Data Box Edge	Azure Data Factory
Initial setup effort	Low (<30 mins.)	Moderate (~couple hours)	Large (~days)
Data Format	Azure Blobs, Azure Files	Azure Blobs, Azure Files	Supports 70+ data connectors for data stores and formats
Data pre-processing	No	Yes, via Edge compute	Yes
Local cache (to store on-premises data)	Yes	Yes	No
Transfer from other clouds	No	No	Yes
Pricing	Pricing ↗	Pricing ↗	Pricing ↗

Next steps

- [Transfer data with AzCopy](#).
- [More information on data transfer with Storage REST APIs](#).
- Understand how to:
 - [Transfer data with Data Box Gateway](#).
 - [Transform data with Data Box Edge before sending to Azure](#).
- Learn how to transfer data with [Azure Data Factory](#).

Understand data store models

Article • 12/09/2022

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

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

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

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

ⓘ Note

Learn more about identifying and reviewing your data service requirements for cloud adoption, in the [Microsoft Cloud Adoption Framework for Azure](#). Likewise, you can also learn about [selecting storage tools and services](#).

Relational database management systems

Relational databases organize data as a series of two-dimensional tables with rows and columns. Most vendors provide a dialect of the Structured Query Language (SQL) for

retrieving and managing data. An RDBMS typically implements a transactionally consistent mechanism that conforms to the ACID (Atomic, Consistent, Isolated, Durable) model for updating information.

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

This model is very useful when strong consistency guarantees are important — where all changes are atomic, and transactions always leave the data in a consistent state.

However, an RDBMS generally can't scale out horizontally without sharding the data in some way. Also, the data in an RDBMS must be normalized, which isn't appropriate for every data set.

Azure services

- [Azure SQL Database ↗ | \(Security Baseline\)](#)
- [Azure Database for MySQL ↗ | \(Security Baseline\)](#)
- [Azure Database for PostgreSQL ↗ | \(Security Baseline\)](#)
- [Azure Database for MariaDB ↗ | \(Security Baseline\)](#)

Workload

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

Data type

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

Examples

- Inventory management
- Order management
- Reporting database
- Accounting

Key/value stores

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

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

The diagram shows a table with four rows, each containing a key and its corresponding binary value. A callout box labeled "Opaque to data store" points to the second row's value.

Key	Value
AAAAA	1101001111010100110101111...
AABAB	1001100001011001101011110...
DFA766	0000000000101010110101010...
FABCC4	1110110110101010100101101...

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

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

Azure services

- Azure Cosmos DB for Table and Azure Cosmos DB for NoSQL | [\(Azure Cosmos DB Security Baseline\)](#)
- Azure Cache for Redis ↗ | [\(Security Baseline\)](#)
- Azure Table Storage | [\(Security Baseline\)](#)

Workload

- Data is accessed using a single key, like a dictionary.

- No joins, lock, or unions are required.
- No aggregation mechanisms are used.
- Secondary indexes are generally not used.

Data type

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

Examples

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

Document databases

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

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

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

Azure service

- [Azure Cosmos DB for NoSQL | \(Azure Cosmos DB Security Baseline\)](#)

Workload

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

Data type

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

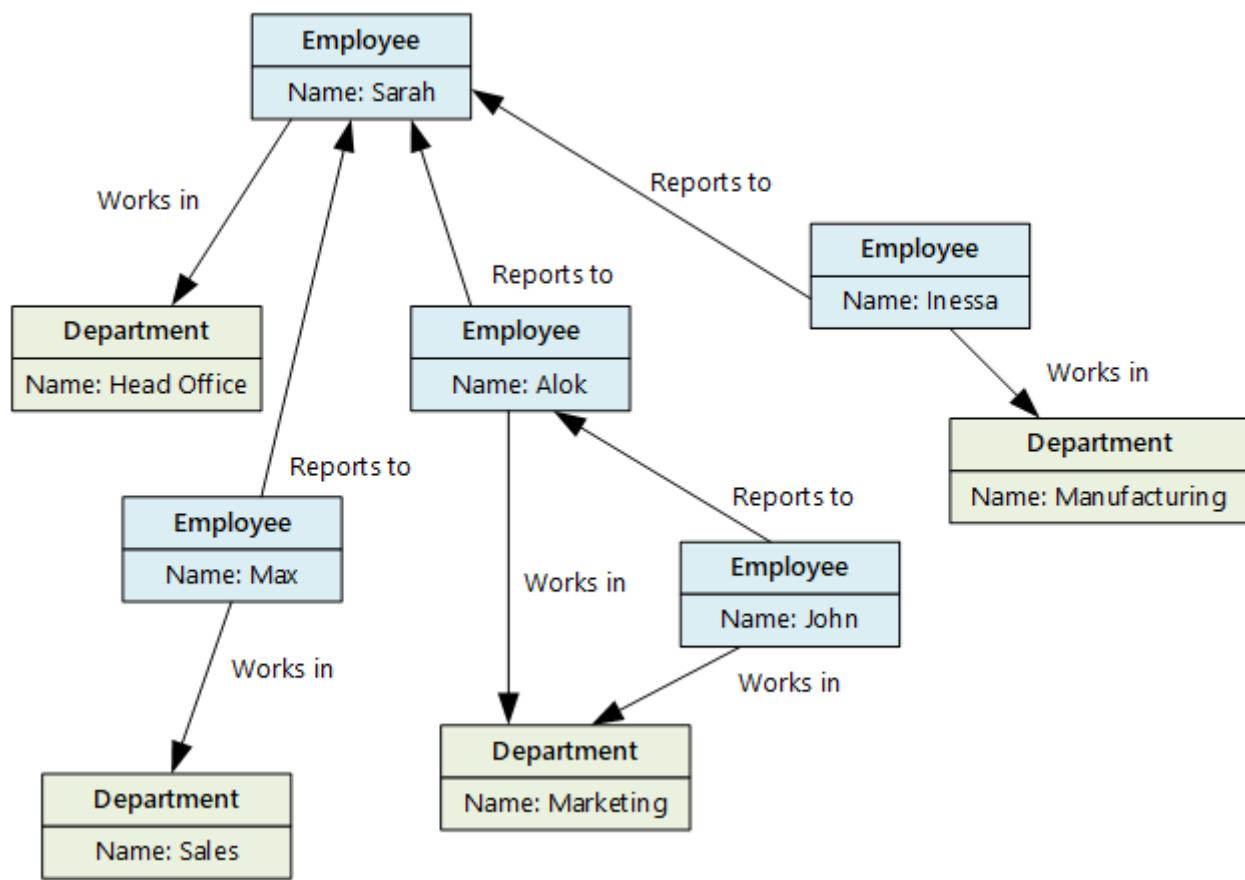
Examples

- Product catalog
- Content management
- Inventory management

Graph databases

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

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



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

Azure services

- Azure Cosmos DB for Apache Gremlin | [\(Security Baseline\)](#)
- SQL Server | [\(Security Baseline\)](#)

Workload

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

Data type

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

Examples

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

Data analytics

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

Azure services

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

Workload

- Data analytics
- Enterprise BI

Data type

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

Examples

- Enterprise data warehouse

Column-family databases

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

You can think of a column-family database as holding tabular data with rows and columns, but the columns are divided into groups known as *column families*. Each column family holds a set of columns that are logically related together and are typically retrieved or manipulated as a unit. Other data that is accessed separately can be stored in separate column families. Within a column family, new columns can be added

dynamically, and rows can be sparse (that is, a row doesn't need to have a value for every column).

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

CustomerID	Column Family: Identity	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 store data in key order, rather than by computing a hash. Many implementations allow you to create indexes over specific columns in a column-family. Indexes let you retrieve data by columns value, rather than row key.

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

Azure services

- [Azure Cosmos DB for Apache Cassandra | \(Security Baseline\)](#)
- [HBase in HDInsight | \(Security Baseline\)](#)

Workload

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

Data type

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

Examples

- Recommendations
- Personalization
- Sensor data
- Telemetry
- Messaging
- Social media analytics
- Web analytics
- Activity monitoring
- Weather and other time-series data

Search Engine Databases

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

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

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

Azure service

- [Azure Search](#) | (Security Baseline)

Workload

- Data indexes from multiple sources and services.
- Queries are ad-hoc and can be complex.

- Full text search is required.
- Ad hoc self-service query is required.

Data type

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

Examples

- Product catalogs
- Site search
- Logging

Time series databases

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

Azure service

- [Azure Time Series Insights](#) ↗

Workload

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

Data type

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

Examples

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

Object storage

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

Azure service

- [Azure Blob Storage](#) | [\(Security Baseline\)](#)
- [Azure Data Lake Storage Gen2](#) | [\(Security Baseline\)](#)

Workload

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

Data type

- Data size is large.
- Value is opaque.

Examples

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

Shared files

Sometimes, using simple flat files can be the most effective means of storing and retrieving information. Using file shares enables files to be accessed across a network.

Given appropriate security and concurrent access control mechanisms, sharing data in this way can enable distributed services to provide highly scalable data access for performing basic, low-level operations such as simple read and write requests.

Azure service

- [Azure Files ↗ | \(Security Baseline\)](#)

Workload

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

Data type

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

Examples

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

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

Next steps

- [Azure Cloud Storage Solutions and Services ↗](#)
- [Review your storage options](#)
- [Introduction to Azure Storage](#)
- [Introduction to Azure Data Explorer](#)

Related resources

- [Databases architecture design](#)
- [Big data architectures](#)
- [Choose a data storage technology](#)
- [Data store decision tree](#)

Non-relational data and NoSQL

Azure Cosmos DB Azure Blob Storage Azure Data Lake

A *non-relational database* is a database that does not use the tabular schema of rows and columns found in most traditional database systems. Instead, non-relational databases use a storage model that is optimized for the specific requirements of the type of data being stored. For example, data may be stored as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.

What all of these data stores have in common is that they don't use a [relational model](#). Also, they tend to be more specific in the type of data they support and how data can be queried. For example, time series data stores are optimized for queries over time-based sequences of data. However, graph data stores are optimized for exploring weighted relationships between entities. Neither format would generalize well to the task of managing transactional data.

The term *NoSQL* refers to data stores that do not use SQL for queries. Instead, the data stores use other programming languages and constructs to query the data. In practice, "NoSQL" means "non-relational database," even though many of these databases do support SQL-compatible queries. However, the underlying query execution strategy is usually very different from the way a traditional RDBMS would execute the same SQL query.

The following sections describe the major categories of non-relational or NoSQL database.

Document data stores

A document data store manages a set of named string fields and object data values in an entity that's referred to as a *document*. These data stores typically store data in the form of JSON documents. Each field value could be a scalar item, such as a number, or a compound element, such as a list or a parent-child collection. The data in the fields of a document can be encoded in various ways, including XML, YAML, JSON, BSON, or even stored as plain text. The fields within documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

Typically, a document contains the entire data for an entity. What items constitute an entity are application-specific. For example, an entity could contain the details of a

customer, an order, or a combination of both. A single document might contain information that would be spread across several relational tables in a relational database management system (RDBMS). A document store does not require that all documents have the same structure. This free-form approach provides a great deal of flexibility. For example, applications can store different data in documents in response to a change in business requirements.

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

The application can retrieve documents by using the document key. The key is a unique identifier for the document, which is often hashed, to help distribute data evenly. Some document databases create the document key automatically. Others enable you to specify an attribute of the document to use as the key. The application can also query documents based on the value of one or more fields. Some document databases support indexing to facilitate fast lookup of documents based on one or more indexed fields.

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

Relevant Azure service:

- [Azure Cosmos DB ↗](#)

Columnar data stores

A columnar or column-family data store organizes data into columns and rows. In its simplest form, a column-family data store can appear very similar to a relational

database, at least conceptually. The real power of a column-family database lies in its denormalized approach to structuring sparse data, which stems from the column-oriented approach to storing data.

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

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

CustomerID	Column Family: Identity	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: Adamcyz 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 specific number of rows in each file. With large data sets, this approach creates a performance benefit by reducing the amount of data that needs to be read from disk when only a few columns are queried together at a time.

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

Relevant Azure service:

- Azure Cosmos DB for Apache Cassandra
- HBase in HDInsight

Key/value data stores

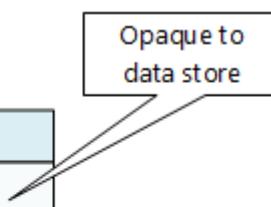
A key/value store is essentially a large hash table. You associate each data value with a unique key, and the key/value store uses this key to store the data by using an appropriate hashing function. The hashing function is selected to provide an even distribution of hashed keys across the data storage.

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...

Opaque to data store



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.

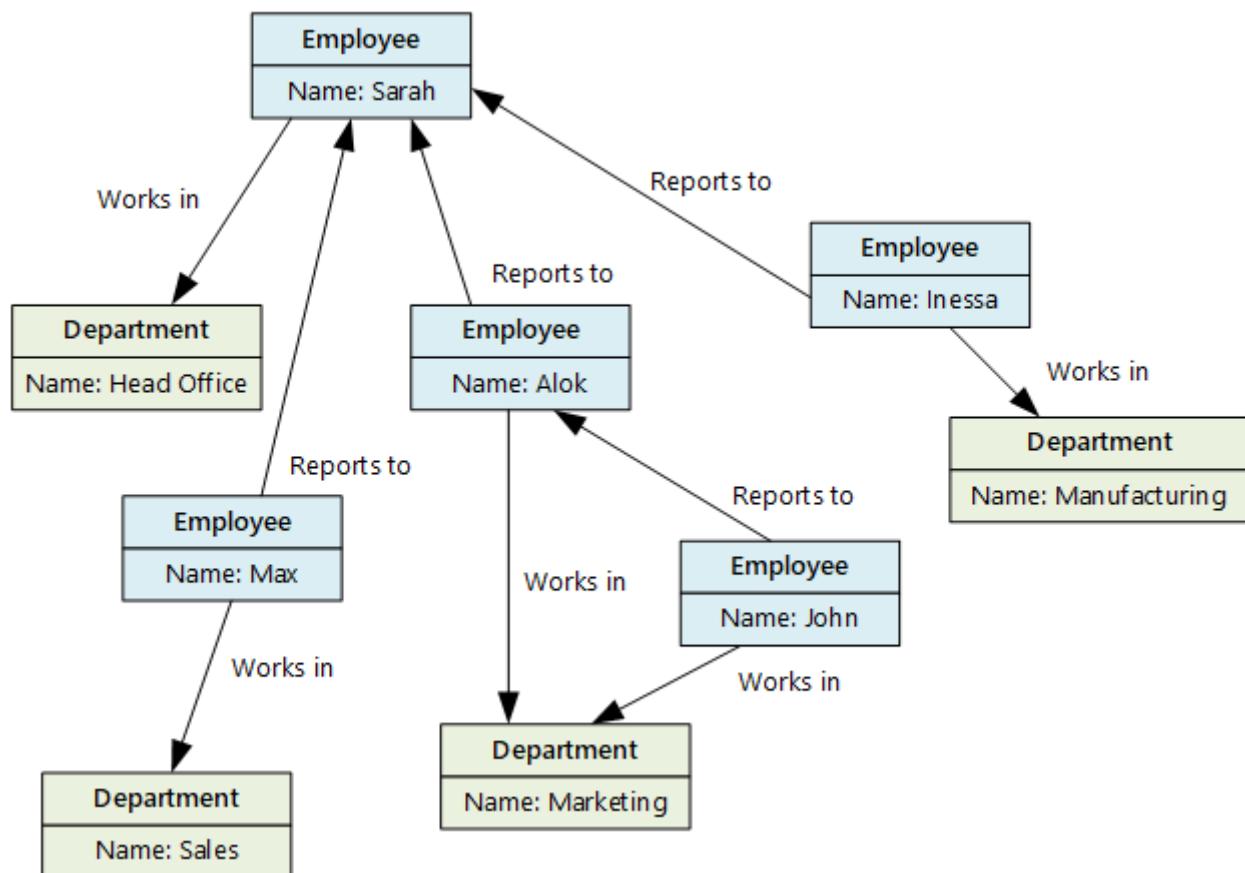
Relevant Azure services:

- [Azure Cosmos DB for Table](#)
- [Azure Cache for Redis ↗](#)
- [Azure Table Storage ↗](#)

Graph data stores

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

The purpose of a graph data store is to allow an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. The following diagram shows an organization's personnel data structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.



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

Relevant Azure service:

- [Azure Cosmos DB Graph API](#)

Time series data stores

Time series data is a set of values organized by time, and a time series data store is optimized for this type of data. Time series data stores must support a very high number of writes, as they typically collect large amounts of data in real time from a large number of sources. Time series data stores are optimized for storing telemetry data. Scenarios include IoT sensors or application/system counters. Updates are rare, and deletes are often done as bulk operations.

timestamp	deviceid	value
2017-01-05T08:00:00.123	1	90.0
2017-01-05T08:00:01.225	2	75.0
2017-01-05T08:01:01.525	2	78.0

Although the records written to a time series database are generally small, there are often a large number of records, and total data size can grow rapidly. Time series data stores also handle out-of-order and late-arriving data, automatic indexing of data points, and optimizations for queries described in terms of windows of time. This last feature enables queries to run across millions of data points and multiple data streams quickly, in order to support time series visualizations, which is a common way that time series data is consumed.

Relevant Azure services:

- [Azure Time Series Insights ↗](#)
- [OpenTSDB with HBase on HDInsight](#)

Object data stores

Object data stores are optimized for storing and retrieving large binary objects or blobs such as images, text files, video and audio streams, large application data objects and documents, and virtual machine disk images. An object consists of the stored data, some metadata, and a unique ID for accessing the object. Object stores are designed to support files that are individually very large, as well provide large amounts of total storage to manage all files.

path	blob	metadata
/delays/2017/06/01/flights.csv	0XAABBCDDDEEF...	{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 process, in turn, enables the scale-out querying of data contained in large files, because multiple processes, typically running on different servers, can each query the large data file simultaneously.

One special case of object data stores is the network file share. Using file shares enables files to be accessed across a network using standard networking protocols like server message block (SMB). Given appropriate security and concurrent access control mechanisms, sharing data in this way can enable distributed services to provide highly scalable data access for basic, low-level operations such as simple read and write requests.

Relevant Azure services:

- [Azure Blob Storage](#)
- [Azure Data Lake Store](#)
- [Azure File Storage](#)

External index data stores

External index data stores provide the ability to search for information held in other data stores and services. An external index acts as a secondary index for any data store, and can be used to index massive volumes of data and provide near real-time access to these indexes.

For example, you might have text files stored in a file system. Finding a file by its file path is quick, but searching based on the contents of the file would require a scan of all

of the files, which is slow. An external index lets you create secondary search indexes and then quickly find the path to the files that match your criteria. Another example application of an external index is with key/value stores that only index by the key. You can build a secondary index based on the values in the data, and quickly look up the key that uniquely identifies each matched item.

id	search-document
233358	{"name": "Pacific Crest National Scenic Trail", "county": "San Diego", "elevation":1294, "location": {"type": "Point", "coordinates": [-120.802102,49.00021]}}
801970	{"name": "Lewis and Clark National Historic Trail", "county": "Richland", "elevation":584, "location": {"type": "Point", "coordinates": [-104.8546903,48.1264084]}}
1144102	{"name": "Intake Trail", "county": "Umatilla", "elevation":1076, "location": {"type": "Point", "coordinates": [-118.0468873,45.9981939]}}

The indexes are created by running an indexing process. This can be performed using a pull model, triggered by the data store, or using a push model, initiated by application code. Indexes can be multidimensional and may support free-text searches across large volumes of text data.

External index data stores are often used to support full text and web-based search. In these cases, searching can be exact or fuzzy. A fuzzy search finds documents that match a set of terms and calculates how closely they match. Some external indexes also support linguistic analysis that can return matches based on synonyms, genre expansions (for example, matching "dogs" to "pets"), and stemming (for example, searching for "run" also matches "ran" and "running").

Relevant Azure service:

- [Azure Search](#) ↗

Typical requirements

Non-relational data stores often use a different storage architecture from that used by relational databases. Specifically, they tend toward having no fixed schema. Also, they tend not to support transactions, or else restrict the scope of transactions, and they generally don't include secondary indexes for scalability reasons.

The following compares the requirements for each of the non-relational data stores:

[Expand table](#)

Requirement	Document data	Column-family data	Key/value data	Graph data
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
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

Requirement	Document data	Column-family data	Key/value data	Graph data
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)

[\[+\]](#) Expand table

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

Requirement	Time series data	Object data	External index data
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)

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [Relational vs. NoSQL data](#)
- [Understand distributed NoSQL databases](#)
- [Microsoft Azure Data Fundamentals: Explore non-relational data in Azure](#)
- [Implement a non-relational data model](#)

Related resources

- [Understand data store models](#)
- [Scalable order processing](#)
- [Near real-time lakehouse data processing](#)

Review your data options

Article • 07/21/2023

When you prepare your landing zone environment for your cloud adoption, you need to determine the data requirements for hosting your workloads. Azure database products and services support various data storage scenarios and capabilities. How you configure your landing zone environment to support your data requirements depends on your workload governance, technical, and business requirements.

Identify data services requirements

As part of your landing zone evaluation and preparation, you need to identify the data stores that your landing zone needs to support. The process involves assessing each of the applications and services that make up your workloads to determine their data storage and access requirements. After you identify and document these requirements, you can create policies for your landing zone to control allowed resource types based on your workload needs.

For each application or service you deploy to your landing zone environment, use the following information as a starting point to help you determine the appropriate data store services to use.

Key questions

Answer the following questions about your workloads to help you make decisions based on the Azure database services decision tree:

- **What is the level of control of the OS and database engine required?** Some scenarios require you to have a high degree of control or ownership of the software configuration and host servers for your database workloads. In these scenarios, you can deploy custom infrastructure as a service (IaaS) virtual machines to fully control the deployment and configuration of data services. You might not require this level of control, but maybe you're not ready to move to a full platform as a service (PaaS) solution. In that case, a managed instance can provide higher compatibility with your on-premises database engine while offering the benefits of a fully managed platform.
- **Will your workloads use a relational database technology?** If so, what technology do you plan to use? Azure provides managed PaaS database capabilities for [Azure SQL Database](#), [MySQL](#), [PostgreSQL](#), and [MariaDB](#).
 - Azure Cosmos DB supports [MongoDB](#) and [PostgreSQL](#) APIs to take advantage of the many benefits that Azure Cosmos DB offers, including automatic high availability and instantaneous scalability.
- **Will your workloads use SQL Server?** In Azure, you can have your workloads running in IaaS-based [SQL Server on Azure Virtual Machines](#) or on the PaaS-based [Azure SQL Database hosted service](#). Choosing which option to use is primarily a question of whether you want to manage your database, apply patches, and take backups, or if you want to delegate these operations to Azure. In some scenarios, compatibility issues might require the use of IaaS-hosted SQL Server. For more information about how to choose the correct option for your workloads, see [Choose the right SQL Server option in Azure](#).
- **Will your workloads use key/value database storage?** [Azure Cache for Redis](#) offers a high-performance cached key/value data storage solution that can power fast, scalable applications. [Azure Cosmos DB](#) also provides general-purpose key/value storage capabilities.
- **Will your workloads use document or graph data?** [Azure Cosmos DB](#) is a multimodel database service that supports various data types and APIs. Azure Cosmos DB also provides document and graph database capabilities.
 - [MongoDB](#) and [Apache Gremlin](#) are document and graph APIs that are supported by Azure Cosmos DB.
- **Will your workloads use column-family data?** [Azure Managed Instance for Apache Cassandra](#) offers a fully managed Apache Cassandra cluster that can extend your existing datacenters into Azure or act as a cloud-only cluster and datacenter.
 - [Apache Cassandra](#) API is also supported by Azure Cosmos DB. See the [product comparison](#) documentation to help guide your decision on the best fit for your workload.
- **Will your workloads require high-capacity data analytics capabilities?** You can use [Azure Synapse Analytics](#) to effectively store and query structured petabyte-scale data. For unstructured big data workloads, you can use [Azure Data Lake](#) to store and analyze petabyte-size files and trillions of objects.
- **Will your workloads require search engine capabilities?** You can use [Azure Cognitive Search](#) to build AI-enhanced cloud-based search indexes that you can integrate into your applications.

- Will your workloads use time series data? [Azure Time Series Insights](#) is built to store, visualize, and query large amounts of time series data, such as data generated by IoT devices.

ⓘ Note

Learn more about how to assess database options for each of your applications or services in the [Azure application architecture guide](#).

Common database scenarios

The following table lists common use-scenario requirements and the recommended database services for handling them.

If you want to	Use this database service
Build apps that scale with a managed and intelligent SQL database in the cloud.	Azure SQL Database
Modernize SQL Server applications with a managed, always-up-to-date SQL instance in the cloud.	Azure SQL Managed Instance
Migrate your SQL workloads to Azure while maintaining complete SQL Server compatibility and operating system-level access.	SQL Server on Azure Virtual Machines
Build scalable, secure, and fully managed enterprise-ready apps on open-source PostgreSQL, scale out single-node PostgreSQL with high performance, or migrate PostgreSQL and Oracle workloads to the cloud.	Azure Database for PostgreSQL
Deliver high availability and elastic scaling to open-source mobile and web apps with a managed community MySQL database service, or migrate MySQL workloads to the cloud.	Azure Database for MySQL
Deliver high availability and elastic scaling to open-source mobile and web apps with a managed community MariaDB database service.	Azure Database for MariaDB
Build applications with guaranteed low latency and high availability anywhere, at any scale, or migrate Cassandra, MongoDB, Gremlin, and other NoSQL workloads to the cloud.	Azure Cosmos DB
Modernize existing Cassandra data clusters and apps, and enjoy flexibility and freedom with managed instance service.	Azure Managed Instance for Apache Cassandra
Build a fully managed elastic data warehouse that has security at every level of scale at no extra cost.	Azure Synapse Analytics
Power fast, scalable applications with an open-source-compatible in-memory data store.	Azure Cache for Redis

Database feature comparison

The following table lists features available in Azure database services.

Feature	Azure SQL Database	Azure SQL Managed Instance	Azure Database for PostgreSQL	Azure Database for MySQL	Azure Database for MariaDB	Azure Managed Instance for Apache Cassandra	Azure Cosmos DB	Azure Cache for Redis	Azure Cosmos DB for MongoDB	Azure Cosmos DB for Gremlin
Database type	Relational	Relational	Relational	Relational	Relational	NoSQL	NoSQL	In-memory	NoSQL	Graph
Data model	Relational	Relational	Relational	Relational	Relational	Multimodel: Document, Wide-column, Key-value, Graph	Wide-column	Key-value	Document	Graph
Distributed multimaster writes	No	No	No	No	No	Yes	Yes	Yes (Enterprise and Flash tiers only)	Yes	Yes

Feature	Azure SQL Database	Azure SQL Managed Instance	Azure Database for PostgreSQL	Azure Database for MySQL	Azure Database for MariaDB	Azure Managed Instance for Apache Cassandra	Azure Cosmos DB	Azure Cache for Redis	Azure Cosmos DB for MongoDB	Azure Cosmos DB for Gremlin
Virtual network connectivity support	Virtual network service endpoint	Native virtual network implementation	Virtual network injection (flexible server only)	Virtual network injection (flexible server only)	Virtual network service endpoint	Native virtual network implementation	Virtual network service endpoint	Virtual network injection (Premium, Enterprise, and Flash tiers only)	Virtual network service endpoint	Virtual network service endpoint

ⓘ Note

Private link service simplifies networking design to allow Azure services to communicate over private networking. It's supported for all Azure database services. In the case of Managed Instance database services, these instances are deployed in virtual networks, which negates the need to deploy private endpoints for them.

Regional availability

Azure lets you deliver services at the scale you need to reach your customers and partners *wherever they are*. A key factor in planning your cloud deployment is to determine what Azure region will host your workload resources.

Most database services are generally available in most Azure regions. A few regions support only a subset of these products, but they mostly target governmental customers. Before you decide which regions you'll deploy your database resources to, see [Products available by region](#) to check the latest status of regional availability.

To learn more about Azure global infrastructure, see [Azure geographies](#). For specific details about the overall services that are available in each Azure region, see [Products available by region](#).

Data residency and compliance requirements

Legal and contractual requirements that are related to data storage often apply to your workloads. These requirements might vary based on the location of your organization, the jurisdiction of the physical assets that host your data stores, and your applicable business sector. Components of data obligations to consider include:

- Data classification.
- Data location.
- Responsibilities for data protection under the shared responsibility model.

For help with understanding these requirements, see [Achieving compliant data residency and security with Azure](#).

Part of your compliance efforts might include controlling where your database resources are physically located. Azure regions are organized into groups called geographies. An [Azure geography](#) ensures that data residency, sovereignty, compliance, and resiliency requirements are honored within geographical and political boundaries. If your workloads are subject to data sovereignty or other compliance requirements, you must deploy your storage resources to regions in a compliant Azure geography.

Establish controls for database services

When you prepare your landing zone environment, you can establish controls that limit what data stores that users can deploy. Controls can help you manage costs and limit security risks. Developers and IT teams will still be able to deploy and configure resources that are needed to support your workloads.

After you identify and document your landing zone's requirements, you can use [Azure Policy](#) to control the database resources that you allow users to create. Controls can take the form of allowing or denying the creation of [database resource types](#).

For example, you might restrict users to creating only Azure SQL Database resources. You can also use policies to control the allowable options when a resource is created. For example, you can restrict what SQL Database SKUs can be provisioned by allowing only specific versions of SQL Server to be installed on an IaaS VM. For more information, see [Azure Policy built-in policy definitions](#).

Policies can be scoped to resources, resource groups, subscriptions, and management groups. You can include your policies in [Azure Blueprints](#) definitions and apply them repeatedly throughout your cloud estate.

Next steps

- Review [database security best practices](#).
- Review a comparison of [Azure SQL deployment options](#).

Criteria for choosing a data store

Article • 07/21/2023

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

General considerations

Keep the following considerations in mind when you make your selection.

Functional requirements

- **Data format:** What type of data are you intending to store? Common types include transactional data, JSON objects, telemetry, search indexes, or flat files.
- **Data size:** How large are the entities you need to store? Will these entities need to be maintained as a single document, or can they be split across multiple documents, tables, and collections?
- **Scale and structure:** What's 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 you update and synchronize data? Will the application perform many updates that could potentially conflict? If so, you might 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 multiversion 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?

Nonfunctional 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 after it's ingested? How large will you need the data store to scale up? Is your workload more read-heavy or write-heavy?
- **Reliability:** What overall service-level agreement 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 infrastructure as a service (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's 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 programming languages, operating systems, or other technology that your team is 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?

Next steps

- [Azure cloud storage solutions and services](#) ↗
- [Review your storage options](#)
- [Introduction to Azure Storage](#)

Related resources

- [Data store decision tree](#)
- [Understand data store models](#)
- [Choose a data storage technology](#)

Choose a big data storage technology in Azure

Article • 10/09/2023

ⓘ Note

On **Feb 29, 2024** Azure Data Lake Storage Gen1 will be retired. For more information, see the [official announcement](#). If you use Azure Data Lake Storage Gen1, make sure to migrate to Azure Data Lake Storage Gen2 prior to that date. To learn how, see [Migrate Azure Data Lake Storage from Gen1 to Gen2 by using the Azure portal](#).

Unless you already have an Azure Data Lake Storage Gen1 account, you cannot create new ones.

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 Storage Gen1](#)

NoSQL databases:

- [Azure Cosmos DB](#)
- [HBase on HDInsight](#)

Analytical databases:

[Azure Data Explorer](#)

Azure Storage blobs

Azure Storage is a managed storage service that is highly available, secure, durable, scalable, and redundant. Microsoft takes care of maintenance and handles critical problems for you. Azure Storage is the most ubiquitous storage solution Azure provides, due to the number of services and tools that can be used with it.

There are various Azure Storage services you can use to store data. The most flexible option for storing blobs from many data sources is [Blob storage](#). Blobs are basically files. They store pictures, documents, HTML files, virtual hard disks (VHDs), big data such as logs, database backups—pretty much anything. Blobs are stored in containers, which are similar to folders. A container provides a grouping of a set of blobs. A storage account can contain an unlimited number of containers, and a container can store an unlimited number of blobs.

Azure Storage is a good choice for big data and analytics solutions, because of its flexibility, high availability, and low cost. It provides hot, cool, and archive storage tiers for different use cases. For more information, see [Azure Blob Storage: Hot, cool, and archive storage tiers](#).

Azure Blob storage can be accessed from Hadoop (available through HDInsight). HDInsight can use a blob container in Azure Storage as the default file system for the cluster. Through a Hadoop distributed file system (HDFS) interface provided by a WASB driver, the full set of components in HDInsight can operate directly on structured or unstructured data stored as blobs. Azure Blob storage can also be accessed via Azure Synapse Analytics using its PolyBase feature.

Other features that make Azure Storage a good choice are:

- [Multiple concurrency strategies](#).
- [Disaster recovery and high availability options](#).
- [Encryption at rest](#).
- [Azure role-based access control \(Azure RBAC\)](#) to control access using Microsoft Entra users and groups.

Azure Data Lake Storage Gen1

[Azure Data Lake Storage Gen1](#) is an enterprise-wide hyperscale repository for big data analytic workloads. Data Lake enables you to capture data of any size, type, and ingestion speed in one single [secure](#) location for operational and exploratory analytics.

Azure Data Lake Storage Gen1 doesn't 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's 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.

Azure Data Lake Storage Gen1 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 Azure Data Lake Storage Gen1 as your primary storage for an HDInsight cluster, with specific guidelines for [Spark](#), [Hive](#), and [MapReduce](#). Also, be sure to check Azure Data Lake Storage Gen1's [regional availability](#), because it isn't 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, Azure Data Lake Storage Gen1 is designed to enable analytics on the stored data and is tuned for performance for data analytics scenarios. Azure Data Lake Storage Gen1 can also be accessed via Azure Synapse using its PolyBase feature.

Azure Cosmos DB

[Azure Cosmos DB](#) is Microsoft's globally distributed multi-model database. 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.

Azure Cosmos DB is schema-agnostic. It automatically indexes all the data without requiring you to deal with schema and index management. It's also multi-model, natively supporting document, key-value, graph, and column-family data models.

Azure Cosmos DB features:

- [Geo-replication](#)
- [Elastic scaling of throughput and storage](#) worldwide
- [Five well-defined consistency levels](#)

HBase on HDInsight

[Apache HBase](#) is an open-source, NoSQL database that is built on Hadoop and modeled after Google BigTable. HBase provides random access and strong consistency

for large amounts of unstructured and semi-structured data in a schemaless database organized by column families.

Data is stored in the rows of a table, and data within a row is grouped by column family. HBase is schemaless in the sense that neither the columns nor the type of data stored in them need to be defined before using them. The open-source code scales linearly to handle petabytes of data on thousands of nodes. It can rely on data redundancy, batch processing, and other features that are provided by distributed applications in the Hadoop ecosystem.

The [HDInsight implementation](#) leverages the scale-out architecture of HBase to provide automatic sharding of tables, strong consistency for reads and writes, and automatic failover. Performance is enhanced by in-memory caching for reads and high-throughput streaming for writes. In most cases, you'll want to [create the HBase cluster inside a virtual network](#) so other HDInsight clusters and applications can directly access the tables.

Azure Data Explorer

[Azure Data Explorer](#) is a fast and highly scalable data exploration service for log and telemetry data. It helps you handle the many data streams emitted by modern software so you can collect, store, and analyze data. Azure Data Explorer is ideal for analyzing large volumes of diverse data from any data source, such as websites, applications, IoT devices, and more. This data is used for diagnostics, monitoring, reporting, machine learning, and additional analytics capabilities. Azure Data Explorer makes it simple to ingest this data and enables you to do complex ad hoc queries on the data in seconds.

Azure Data Explorer can be linearly [scaled out](#) for increasing ingestion and query processing throughput. An Azure Data Explorer cluster can be [deployed to a Virtual Network](#) for enabling private networks.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need managed, high-speed, cloud-based storage for any type of text or binary data? If yes, then select one of the file storage or analytics options.
- Do you need file storage that is optimized for parallel analytics workloads and high throughput/IOPS? If yes, then choose an option that is tuned to analytics workload performance.

- Do you need to store unstructured or semi-structured data in a schemaless database? If so, select one of the non-relational or analytics options. Compare options for indexing and database models. Depending on the type of data you need to store, the primary database models may be the largest factor.
- Can you use the service in your region? Check the regional availability for each Azure service. See [Products available by region](#).

Capability matrix

The following tables summarize the key differences in capabilities.

File storage capabilities

Capability	Azure Data Lake Storage Gen1	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 Microsoft Entra identities	Based on shared secrets Account Access Keys and Shared Access Signature Keys , and Azure role-based access control (Azure RBAC)
Authentication protocol	OAuth 2.0. Calls must contain a valid JWT (JSON web token) issued by Microsoft Entra ID	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 Microsoft Entra identities can be set file and folder level.	For account-level authorization use Account Access Keys . For account, container, or blob authorization use Shared Access Signature Keys .
Auditing	Available.	Available
Encryption at rest	Transparent, server side	Transparent, server side; Client-side encryption
Developer SDKs	.NET, Java, Python, Node.js	.NET, Java, Python, Node.js, C++, Ruby

Capability	Azure Data Lake Storage Gen1	Azure Blob Storage containers
Analytics workload performance	Optimized performance for parallel analytics workloads, High Throughput and IOPS	Not optimized for analytics workloads
Size limits	No limits on account sizes, file sizes or number of files	Specific limits documented here
Geo-redundancy	Locally-redundant (LRS), globally redundant (GRS), read-access globally redundant (RA-GRS), zone-redundant (ZRS).	Locally redundant (LRS), globally redundant (GRS), read-access globally redundant (RA-GRS), zone-redundant (ZRS). See here for more information

NoSQL database capabilities

Capability	Azure Cosmos DB	HBase on HDInsight
Primary database model	Document store, graph, key-value store, wide column store	Wide column store
Secondary indexes	Yes	No
SQL language support	Yes	Yes (using the Phoenix JDBC driver)
Consistency	Strong, bounded-staleness, session, consistent prefix, eventual	Strong
Native Azure Functions integration	Yes	No
Automatic global distribution	Yes	No HBase cluster replication can be configured across regions with eventual consistency
Pricing model	Elastically scalable request units (RUs) charged per-second as needed, elastically scalable storage	Per-minute pricing for HDInsight cluster (horizontal scaling of nodes), storage

Analytical database capabilities

Capability	Azure Data Explorer
Primary database model	Relational (column store), telemetry, and time series store

Capability	Azure Data Explorer
SQL language support	Yes
Pricing model	Elastically scalable cluster instances
Authentication	Based on Microsoft Entra identities
Encryption at rest	Supported, customer managed keys
Analytics workload performance	Optimized performance for parallel analytics workloads
Size limits	Linearly scalable

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [Azure Cloud Storage Solutions and Services](#)
- [Review your storage options](#)
- [Introduction to Azure Storage](#)
- [Introduction to Azure Data Explorer](#)

Related resources

- [Big data architectures](#)
- [Big data architecture style](#)
- [Understand data store models](#)
- [Real-time analytics on big data architecture](#)

Understanding distributed NoSQL databases

Article • 01/09/2023

APPLIES TO: NoSQL MongoDB Cassandra Gremlin Table

Azure Cosmos DB is a globally distributed database platform for both NoSQL and relational databases of any scale. This article explores distributed NoSQL databases in the context of Azure Cosmos DB's various NoSQL API options.

For more information about other data storage options in Azure, see [choosing the right data store in the Azure Architecture Center](#).

Challenges

One of the challenges when maintaining a database system is that many database engines apply locks and latches to enforce strict [ACID semantics](#). This approach is beneficial in scenarios where databases require high consistency of the state of the data no matter how it's accessed. While this approach promises high consistency, it makes heavy trade-offs with respect to concurrency, latency, and availability. This restriction is fundamentally an architectural restriction and will force any team with a high transactional workload to find workarounds like manually distributing, or sharding, data across many different databases or database nodes. These workarounds can be time consuming and challenging to implement.

NoSQL databases

[NoSQL databases](#) refer to databases that were designed to simplify horizontal scaling by adjusting consistency to minimize the trade-offs to concurrency, latency, and availability. NoSQL databases offered configurable levels of consistency so that data can scale across many nodes and offer speed or availability that better mapped to the needs of your application.

Distributed databases

[Distributed databases](#) refer to databases that scale across many different instances or locations. While many NoSQL databases are designed for scale, not all are necessarily distributed databases. Even more, many NoSQL databases require time and effort to distribute across redundant nodes for local-redundancy or globally for geo-redundancy.

The planning, implementation, and networking requirements for a globally distributed database can be complex.

Azure Cosmos DB

With a distributed database that is also a NoSQL database, high transactional workloads suddenly became easier to build and manage. [Azure Cosmos DB](#) is a database platform that offers distributed data APIs in both NoSQL and relational variants. Specifically, many of the NoSQL APIs offer various consistency options that allow you to fine tune the level of consistency or availability that meets your real-world application requirements. Your database could be configured to offer high consistency with tradeoffs to speed and availability. Similarly, your database could be configured to offer the best performance with predictable tradeoffs to consistency and latency of your replicated data. Azure Cosmos DB will automatically and dynamically distribute your data across local instances or globally. Azure Cosmos DB can also provide ACID guarantees and scale throughput to map to your application's requirements.

Next steps

Understanding distributed relational databases

Want to get started with Azure Cosmos DB?

- [Learn about the various APIs](#)
- [Get started with the API for NoSQL](#)
- [Get started with the API for MongoDB](#)
- [Get started with the API for Apache Cassandra](#)
- [Get started with the API for Apache Gremlin](#)
- [Get started with the API for Table](#)

Choose an API in Azure Cosmos DB

Article • 12/05/2022

APPLIES TO: NoSQL MongoDB Cassandra Gremlin Table PostgreSQL

Azure Cosmos DB is a fully managed NoSQL database for modern app development. Azure Cosmos DB takes database administration off your hands with automatic management, updates, and patching. It also handles capacity management with cost-effective serverless and automatic scaling options that respond to application needs to match capacity with demand.

APIs in Azure Cosmos DB

Azure Cosmos DB offers multiple database APIs, which include NoSQL, MongoDB, PostgreSQL, Cassandra, Gremlin, and Table. By using these APIs, you can model real world data using documents, key-value, graph, and column-family data models. These APIs allow your applications to treat Azure Cosmos DB as if it were various other databases technologies, without the overhead of management, and scaling approaches. Azure Cosmos DB helps you to use the ecosystems, tools, and skills you already have for data modeling and querying with its various APIs.

All the APIs offer automatic scaling of storage and throughput, flexibility, and performance guarantees. There's no one best API, and you may choose any one of the APIs to build your application. This article will help you choose an API based on your workload and team requirements.

Considerations when choosing an API

API for NoSQL is native to Azure Cosmos DB.

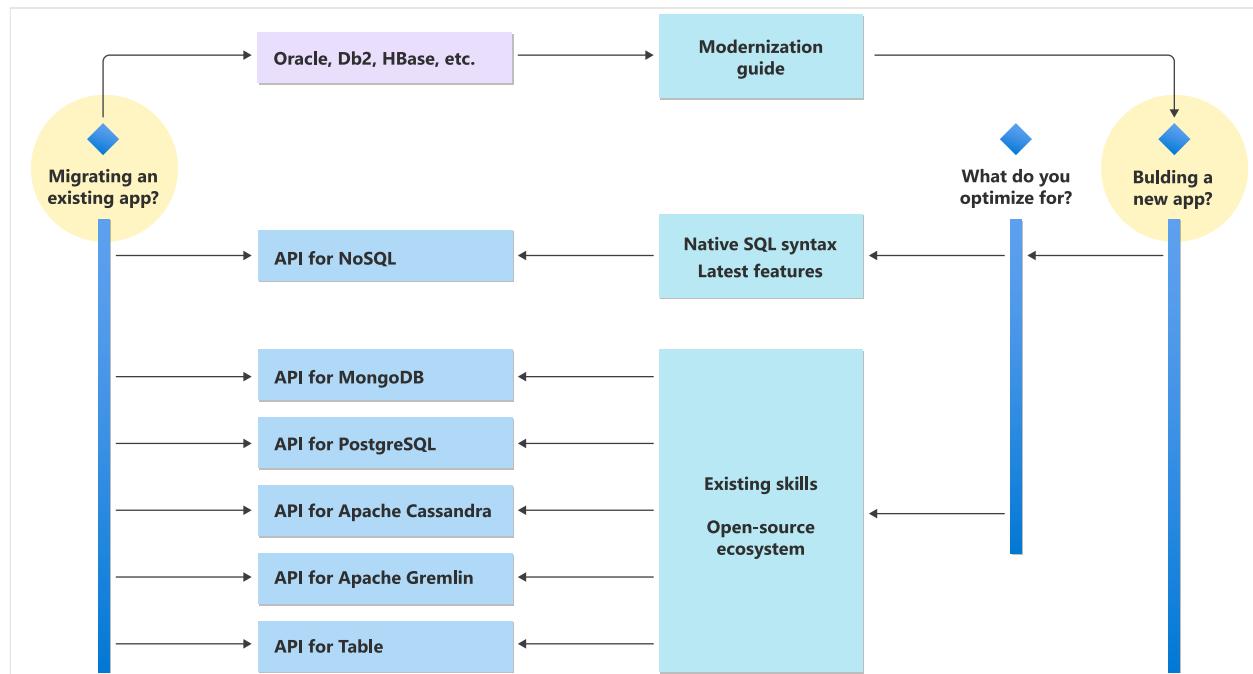
API for MongoDB, PostgreSQL, Cassandra, Gremlin, and Table implement the wire protocol of open-source database engines. These APIs are best suited if the following conditions are true:

- If you have existing MongoDB, PostgreSQL, Cassandra, or Gremlin applications
- If you don't want to rewrite your entire data access layer
- If you want to use the open-source developer ecosystem, client-drivers, expertise, and resources for your database
- If you want to use the Azure Cosmos DB core features such as:

- Global distribution
- Elastic scaling of storage and throughput
- High performance at scale
- Low latency
- Ability to run transactional and analytical workloads
- Fully managed platform
- If you're developing modernized apps on a multicloud environment

You can build new applications with these APIs or migrate your existing data. To run the migrated apps, change the connection string of your application and continue to run as before. When migrating existing apps, make sure to evaluate the feature support of these APIs.

Based on your workload, you must choose the API that fits your requirement. The following image shows a flow chart on how to choose the right API when building new apps or migrating existing apps to Azure Cosmos DB:



API for NoSQL

The Azure Cosmos DB API for NoSQL stores data in document format. It offers the best end-to-end experience as we have full control over the interface, service, and the SDK client libraries. Any new feature that is rolled out to Azure Cosmos DB is first available on API for NoSQL accounts. NoSQL accounts provide support for querying items using the Structured Query Language (SQL) syntax, one of the most familiar and popular query languages to query JSON objects. To learn more, see the [Azure Cosmos DB API for NoSQL](#) training module and [getting started with SQL queries](#) article.

If you're migrating from other databases such as Oracle, DynamoDB, HBase etc. and if you want to use the modernized technologies to build your apps, API for NoSQL is the recommended option. API for NoSQL supports analytics and offers performance isolation between operational and analytical workloads.

API for MongoDB

The Azure Cosmos DB API for MongoDB stores data in a document structure, via BSON format. It's compatible with MongoDB wire protocol; however, it doesn't use any native MongoDB related code. The API for MongoDB is a great choice if you want to use the broader MongoDB ecosystem and skills, without compromising on using Azure Cosmos DB features.

The features that Azure Cosmos DB provides, that you don't have to compromise on includes:

- Scaling
- High availability
- Geo-replication
- Multiple write locations
- Automatic and transparent shard management
- Transparent replication between operational and analytical stores

You can use your existing MongoDB apps with API for MongoDB by just changing the connection string. You can move any existing data using native MongoDB tools such as mongodump & mongorestore or using our Azure Database Migration tool. Tools, such as the MongoDB shell, [MongoDB Compass](#), and [Robo3T](#), can run queries and work with data as they do with native MongoDB. To learn more, see [API for MongoDB](#) article.

API for PostgreSQL

Azure Cosmos DB for PostgreSQL is a managed service for running PostgreSQL at any scale, with the [Citus open source](#) superpower of distributed tables. It stores data either on a single node, or distributed in a multi-node configuration.

Azure Cosmos DB for PostgreSQL is built on native PostgreSQL--rather than a PostgreSQL fork--and lets you choose any major database versions supported by the PostgreSQL community. It's ideal for starting on a single-node database with rich indexing, geospatial capabilities, and JSONB support. Later, if need more performance, you can add nodes to the cluster with zero downtime.

If you're looking for a managed open source relational database with high performance and geo-replication, Azure Cosmos DB for PostgreSQL is the recommended choice. To learn more, see the [Azure Cosmos DB for PostgreSQL introduction](#).

API for Apache Cassandra

The Azure Cosmos DB API for Cassandra stores data in column-oriented schema. Apache Cassandra offers a highly distributed, horizontally scaling approach to storing large volumes of data while offering a flexible approach to a column-oriented schema. API for Cassandra in Azure Cosmos DB aligns with this philosophy to approaching distributed NoSQL databases. This API for Cassandra is wire protocol compatible with native Apache Cassandra. You should consider API for Cassandra if you want to benefit from the elasticity and fully managed nature of Azure Cosmos DB and still use most of the native Apache Cassandra features, tools, and ecosystem. This fully managed nature means on API for Cassandra you don't need to manage the OS, Java VM, garbage collector, read/write performance, nodes, clusters, etc.

You can use Apache Cassandra client drivers to connect to the API for Cassandra. The API for Cassandra enables you to interact with data using the Cassandra Query Language (CQL), and tools like CQL shell, Cassandra client drivers that you're already familiar with. API for Cassandra currently only supports OLTP scenarios. Using API for Cassandra, you can also use the unique features of Azure Cosmos DB such as [change feed](#). To learn more, see [API for Cassandra](#) article. For more information if you're already familiar with Apache Cassandra, but are new to Azure Cosmos DB, see [how to adapt to API for Cassandra](#).

API for Apache Gremlin

The Azure Cosmos DB API for Gremlin allows users to make graph queries and stores data as edges and vertices.

Use the API for Gremlin for scenarios:

- Involving dynamic data
- Involving data with complex relations
- Involving data that is too complex to be modeled with relational databases
- If you want to use the existing Gremlin ecosystem and skills

The API for Gremlin combines the power of graph database algorithms with highly scalable, managed infrastructure. This API provides a unique and flexible solution to

common data problems associated with lack of flexibility or relational approaches. API for Gremlin currently only supports OLTP scenarios.

The API for Gremlin is based on the [Apache TinkerPop](#) graph computing framework. API for Gremlin uses the same Graph query language to ingest and query data. It uses the Azure Cosmos DB partition strategy to do the read/write operations from the Graph database engine. API for Gremlin has a wire protocol support with the open-source Gremlin, so you can use the open-source Gremlin SDKs to build your application. API for Gremlin also works with Apache Spark and [GraphFrames](#) for complex analytical graph scenarios. To learn more, see [API for Gremlin](#) article.

API for Table

The Azure Cosmos DB API for Table stores data in key/value format. If you're currently using Azure Table storage, you may see some limitations in latency, scaling, throughput, global distribution, index management, low query performance. API for Table overcomes these limitations and it's recommended to migrate your app if you want to use the benefits of Azure Cosmos DB. API for Table only supports OLTP scenarios.

Applications written for Azure Table storage can migrate to the API for Table with little code changes and take advantage of premium capabilities. To learn more, see [API for Table](#) article.

Capacity planning when migrating data

Trying to do capacity planning for a migration to Azure Cosmos DB for NoSQL or MongoDB from an existing database cluster? You can use information about your existing database cluster for capacity planning.

- For more information about estimating request units if all you know is the number of vCores and servers in your existing sharded and replicated database cluster, see [estimating request units using vCores or vCPUs](#).
- For more information about estimating request units if you know typical request rates for your current database workload, see [capacity planner for API for NoSQL](#) and [API for MongoDB](#)

Next steps

- [Get started with Azure Cosmos DB for NoSQL](#)
- [Get started with Azure Cosmos DB for MongoDB](#)
- [Get started with Azure Cosmos DB for PostgreSQL](#)

- Get started with Azure Cosmos DB for Cassandra
- Get started with Azure Cosmos DB for Gremlin
- Get started with Azure Cosmos DB for Table
- Trying to do capacity planning for a migration to Azure Cosmos DB? You can use information about your existing database cluster for capacity planning.
 - If all you know is the number of vCores and servers in your existing database cluster, read about [estimating request units using vCores or vCPUs](#)
 - If you know typical request rates for your current database workload, read about [estimating request units using Azure Cosmos DB capacity planner](#)

Choose the right Azure Database for PostgreSQL - Flexible Server hosting option in Azure

Article • 01/25/2024

APPLIES TO:  Azure Database for PostgreSQL - Flexible Server

Important

Azure Database for PostgreSQL - Single Server is on the retirement path. We strongly recommend for you to upgrade to Azure Database for PostgreSQL - Flexible Server. For more information about migrating to Azure Database for PostgreSQL - Flexible Server, see [What's happening to Azure Database for PostgreSQL Single Server?](#)

With Azure, your PostgreSQL workloads can run in a hosted virtual machine infrastructure as a service (IaaS) or as a hosted platform as a service (PaaS). PaaS has multiple deployment options, each with multiple service tiers. When you choose between IaaS and PaaS, you must decide if you want to manage your database, apply patches, and make backups, or if you want to delegate these operations to Azure.

When making your decision, consider the following option in PaaS or alternatively running on Azure VMs (IaaS)

- [Azure Database for PostgreSQL - Flexible Server](#)

PostgreSQL on Azure VMs option falls into the industry category of IaaS. With this service, you can run a PostgreSQL server inside a fully managed virtual machine on the Azure cloud platform. All recent versions and editions of PostgreSQL can be installed on an IaaS virtual machine. In the most significant difference from Azure Database for PostgreSQL flexible server, PostgreSQL on Azure VMs offers control over the database engine. However, this control comes at the cost of responsibility to manage the VMs and many database administration (DBA) tasks. These tasks include maintaining and patching database servers, database recovery, and high-availability design.

The main differences between these options are listed in the following table:

 Expand table

Attribute	Postgres on Azure VMs	Azure Database for PostgreSQL flexible server as PaaS
Availability SLA	- Virtual Machine SLA	- Azure Database for PostgreSQL flexible server
OS and PostgreSQL patching	- Customer managed	Automatic with optional customer managed window
High availability	- Customers architect, implement, test, and maintain high availability. Capabilities might include clustering, replication etc.	Built-in
Zone Redundancy	- Azure VMs can be set up to run in different availability zones. For an on-premises solution, customers must create, manage, and maintain their own secondary data center.	Yes
Hybrid Scenario	- Customer managed	Supported
Backup and Restore	- Customer Managed	Built-in with user configuration on zone-redundant storage
Monitoring Database Operations	- Customer Managed	All offer customers the ability to set alerts on the database operation and act upon reaching thresholds
Advanced Threat Protection	- Customers must build this protection for themselves.	Not available during Preview
Disaster Recovery	- Customer Managed	Supported
Intelligent Performance	- Customer Managed	Supported

Total cost of ownership (TCO)

TCO is often the primary consideration that determines the best solution for hosting your databases. This is true whether you're a startup with little cash or a team in an established company that operates under tight budget constraints. This section describes billing and licensing basics in Azure as they apply to Azure Database for PostgreSQL flexible server and PostgreSQL on Azure VMs.

Billing

Azure Database for PostgreSQL flexible server is currently available as a service in several tiers with different prices for resources. All resources are billed hourly at a fixed rate. For the latest information on the currently supported service tiers, compute sizes, and storage amounts, see [pricing page](#). You can dynamically adjust service tiers and compute sizes to match your application's varied throughput needs. You're billed for outgoing Internet traffic at regular [data transfer rates](#).

With Azure Database for PostgreSQL flexible server, Microsoft automatically configures, patches, and upgrades the database software. These automated actions reduce your administration costs. Also, Azure Database for PostgreSQL flexible server has [automated backup-link](#) capabilities. These capabilities help you achieve significant cost savings, especially when you have a large number of databases. In contrast, with PostgreSQL on Azure VMs you can choose and run any PostgreSQL version. However, you need to pay for the provisioned VM, storage cost associated with the data, backup, monitoring data and log storage and the costs for the specific PostgreSQL license type used (if any).

Azure Database for PostgreSQL flexible server provides built-in high availability at the zonal-level (within an AZ) for any kind of node-level interruption while still maintaining the [SLA guarantee](#) for the service. Azure Database for PostgreSQL flexible server provides [uptime SLAs](#) with and without zone-redundant configuration. However, for database high availability within VMs, you use the high availability options like [Streaming Replication](#) that are available on a PostgreSQL database. Using a supported high availability option doesn't provide another SLA. But it does let you achieve greater than 99.99% database availability at more cost and administrative overhead.

For more information on pricing, see the following articles:

- [Azure Database for PostgreSQL flexible server pricing](#)
- [Virtual machine pricing](#)
- [Azure pricing calculator](#)

Administration

For many businesses, the decision to transition to a cloud service is as much about offloading complexity of administration as it is about cost.

With IaaS, Microsoft:

- Administers the underlying infrastructure.
- Provides automated patching for underlying hardware and OS

With PaaS, Microsoft:

- Administers the underlying infrastructure.
- Provides automated patching for underlying hardware, OS and database engine.
- Manages high availability of the database.
- Automatically performs backups and replicates all data to provide disaster recovery.
- Encrypts the data at rest and in motion by default.
- Monitors your server and provides features for query performance insights and performance recommendations.

With Azure Database for PostgreSQL flexible server, you can continue to administer your database. But you no longer need to manage the database engine, the operating system, or the hardware. Examples of items you can continue to administer include:

- Databases
- Sign-in
- Index tuning
- Query tuning
- Auditing
- Security

Additionally, configuring high availability to another data center requires minimal to no configuration or administration.

- With PostgreSQL on Azure VMs, you have full control over the operating system and the PostgreSQL server instance configuration. With a VM, you decide when to update or upgrade the operating system and database software and what patches to apply. You also decide when to install any other software such as an antivirus application. Some automated features are provided to greatly simplify patching, backup, and high availability. You can control the size of the VM, the number of disks, and their storage configurations. For more information, see [Virtual machine and cloud service sizes for Azure](#).

Time to move to Azure Database for PostgreSQL flexible server (PaaS)

- Azure Database for PostgreSQL flexible server is the right solution for cloud-designed applications when developer productivity and fast time to market for new solutions are critical. With programmatic functionality that is like DBA, the service is suitable for cloud architects and developers because it lowers the need for managing the underlying operating system and database.

- When you want to avoid the time and expense of acquiring new on-premises hardware, PostgreSQL on Azure VMs is the right solution for applications that require a granular control and customization of PostgreSQL engine not supported by the service or requiring access of the underlying OS.

Next steps

- See [Azure Database for PostgreSQL flexible server pricing ↗](#).
- Get started by creating your first server.

Choose a search data store in Azure

Article • 06/01/2023

This article compares technology choices for search data stores in Azure. A search data store is used to create and store specialized indexes for performing searches on free-form text. The text that is indexed may reside in a separate data store, such as blob storage. An application submits a query to the search data store, and the result is a list of matching documents. For more information about this scenario, see [Processing free-form text for search](#).

What are your options when choosing a search data store?

In Azure, all of the following data stores will meet the core requirements for search against free-form text data by providing a search index:

- [Azure Cognitive Search](#)
- [Elasticsearch ↗](#)
- [Azure SQL Database with full text search](#)

Key selection criteria

For search scenarios, begin choosing the appropriate search data store for your needs by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Can you specify your index schema at design time? If not, choose an option that supports updateable schemas.
- Do you need an index only for full-text search, or do you also need rapid aggregation of numeric data and other analytics? If you need functionality beyond full-text search, consider options that support additional analytics.
- Do you need a search index for log analytics, with support for log collection, aggregation, and visualizations on indexed data? If so, consider Elasticsearch, which is part of a log analytics stack.
- Do you need to index data in common document formats such as PDF, Word, PowerPoint, and Excel? If yes, choose an option that provides document indexers.

- Does your database have specific security needs? If yes, consider the security features listed below.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

Capability	Cognitive Search	Elasticsearch	SQL Database
Is managed service	Yes	No	Yes
REST API	Yes	Yes	No
Programmability	.NET, Java, Python, JavaScript	Java	T-SQL
Document indexers for common file types (PDF, DOCX, TXT, and so on)	Yes	No	No

Manageability capabilities

Capability	Cognitive Search	Elasticsearch	SQL Database
Updateable schema	Yes	Yes	Yes
Supports scale out	Yes	Yes	No

Analytic workload capabilities

Capability	Cognitive Search	Elasticsearch	SQL Database
Supports analytics beyond full text search	No	Yes	Yes
Part of a log analytics stack	No	Yes (ELK)	No
Supports semantic search	Yes (find similar documents only)	Yes	Yes

Security capabilities

Capability	Cognitive Search	Elasticsearch	SQL Database
Row-level security	Partial (requires application query to filter by group ID)	Partial (requires application query to filter by group ID)	Yes
Transparent data encryption	No	No	Yes
Restrict access to specific IP addresses	Yes	Yes	Yes
Restrict access to allow virtual network access only	Yes	Yes	Yes
Active Directory authentication (integrated authentication)	No	No	Yes

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [What is Azure Cognitive Search?](#)
- [Full-Text Search in SQL Server and Azure SQL Database](#)
- [Elastic Cloud \(Elasticsearch Service\)](#)

Related resources

- [Process free-form text for search](#)
- [Choose a search data store in Azure](#)
- [Natural language processing technology](#)

Select an Azure data store for your application

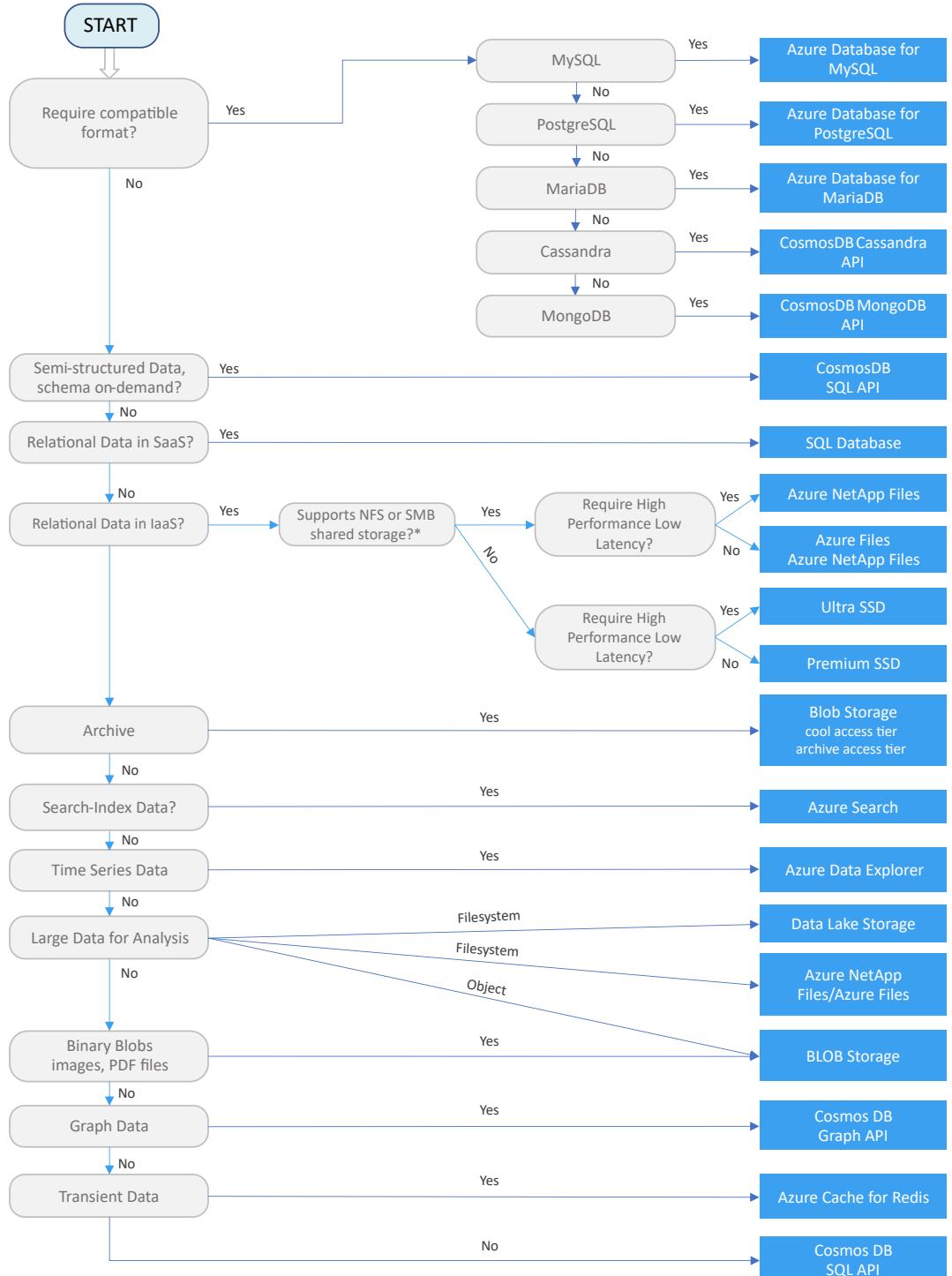
Article • 12/09/2022

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

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

Select a candidate

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



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

Choose specialized storage

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

your database's requirements. For more information about selecting a storage solution, see [Review your storage options](#).

Next steps

- [Azure Cloud Storage Solutions and Services ↗](#)
- [Review your storage options](#)
- [Introduction to Azure Storage](#)

Related resources

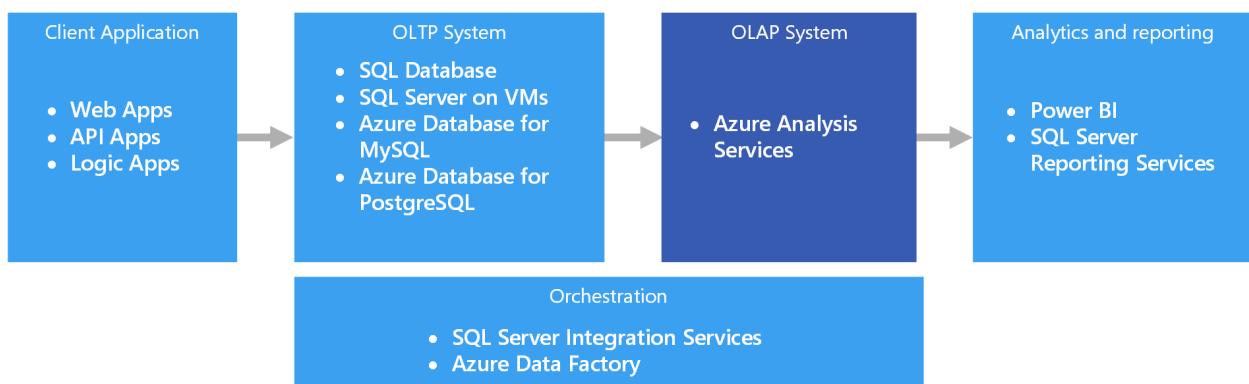
- [Choose a data storage technology](#)
- [Criteria for choosing a data store](#)
- [Understand data store models](#)

Online analytical processing (OLAP)

Azure Analysis Services

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.

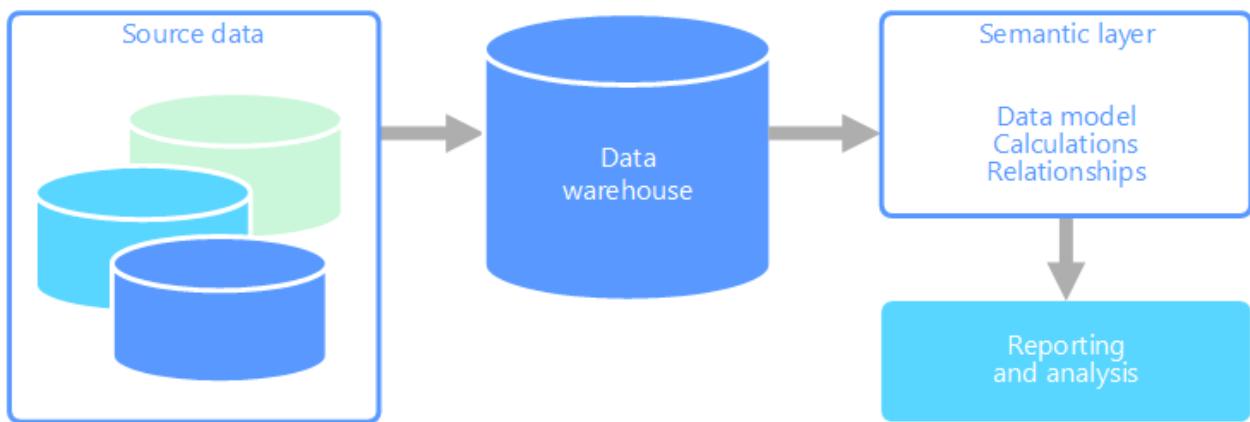
Semantic modeling provides a level of abstraction over the database schema, so that users don't need to know the underlying data structures. This makes it easier for end users to query data without performing aggregates and joins over the underlying

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:

[] Expand table

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
Indexing	Multidimensional indexing
Datum size	Small to medium sized
Model	Multidimensional

Requirement	Description
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 Microsoft Entra ID?
- 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

[] Expand table

Capability	Azure Analysis Services	SQL Server Analysis Services	SQL Server with Columnstore Indexes	Azure SQL Database with Columnstore Indexes
Is managed service	Yes	No	No	Yes

Capability	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 ¹	No ¹
Supports real-time analytics	No	No	Yes	Yes
Requires process to copy data from source(s)	Yes	Yes	No	No
Microsoft Entra integration	Yes	No	No ²	Yes

[1] Although SQL Server and Azure SQL Database cannot be used to query from and integrate multiple external data sources, you can still build a pipeline that does this for you using [SSIS](#) or [Azure Data Factory](#). SQL Server hosted in an Azure VM has additional options, such as linked servers and [PolyBase](#). For more information, see [Pipeline orchestration, control flow, and data movement](#).

[2] Connecting to SQL Server running on an Azure Virtual Machine is not supported using a Microsoft Entra account. Use a domain Active Directory account instead.

Scalability Capabilities

[] [Expand table](#)

Capability	Azure Analysis Services	SQL Server Analysis Services	SQL Server with Columnstore Indexes	Azure SQL Database with Columnstore Indexes
Redundant regional servers for high availability	Yes	No	Yes	Yes
Supports query scale out	Yes	No	Yes	Yes
Dynamic scalability (scale up)	Yes	No	Yes	Yes

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [Columnstore indexes: Overview](#)
- [Create an Analysis Services server](#)
- [What is Azure Data Factory?](#)
- [What is Power BI?](#)

Related resources

- [Big data architecture style](#)
- [Online analytical processing \(OLAP\)](#)

Online transaction processing (OLTP)

Article • 02/16/2023

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
-------------	-------------

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
Appendable	Yes
Workload	Heavy writes, moderate reads
Indexing	Primary and secondary indexes
Datum size	Small to medium sized
Model	Relational
Data shape	Tabular
Query flexibility	Highly flexible
Scale	Small (MBs) to Large (a few TBs)

When to use this solution

Choose OLTP when you need to efficiently process and store business transactions and immediately make them available to client applications in a consistent way. Use this architecture when any tangible delay in processing would have a negative impact on the day-to-day operations of the business.

OLTP systems are designed to efficiently process and store transactions, as well as query transactional data. The goal of efficiently processing and storing individual transactions by an OLTP system is partly accomplished by data normalization — that is, breaking the data up into smaller chunks that are less redundant. This supports efficiency because it enables the OLTP system to process large numbers of transactions independently, and avoids extra processing needed to maintain data integrity in the presence of redundant data.

Challenges

Implementing and using an OLTP system can create a few challenges:

- OLTP systems are not always good for handling aggregates over large amounts of data, although there are exceptions, such as a well-planned SQL Server-based solution. Analytics against the data, that rely on aggregate calculations over millions of individual transactions, are very resource intensive for an OLTP system. They can be slow to execute and can cause a slow-down by blocking other transactions in the database.
- When conducting analytics and reporting on data that is highly normalized, the queries tend to be complex, because most queries need to de-normalize the data by using joins. Also, naming conventions for database objects in OLTP systems tend to be terse and succinct. The increased normalization coupled with terse naming conventions makes OLTP systems difficult for business users to query, without the help of a DBA or data developer.
- Storing the history of transactions indefinitely and storing too much data in any one table can lead to slow query performance, depending on the number of transactions stored. The common solution is to maintain a relevant window of time (such as the current fiscal year) in the OLTP system and offload historical data to other systems, such as a data mart or [data warehouse](#).

OLTP in Azure

Applications such as websites hosted in [App Service Web Apps](#), REST APIs running in App Service, or mobile or desktop applications communicate with the OLTP system, typically via a REST API intermediary.

In practice, most workloads are not purely OLTP. There tends to be an analytical component as well. In addition, there is an increasing demand for real-time reporting, such as running reports against the operational system. This is also referred to as HTAP (Hybrid Transactional and Analytical Processing). For more information, see [Online Analytical Processing \(OLAP\)](#).

In Azure, all of the following data stores will meet the core requirements for OLTP and the management of transaction data:

- [Azure SQL Database](#)
- [SQL Server in an Azure virtual machine](#)
- [Azure Database for MySQL](#)
- [Azure Database for PostgreSQL](#)

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Does your solution have specific dependencies for Microsoft SQL Server, MySQL or PostgreSQL compatibility? Your application may limit the data stores you can choose based on the drivers it supports for communicating with the data store, or the assumptions it makes about which database is used.
- Are your write throughput requirements particularly high? If yes, choose an option that provides in-memory tables.
- Is your solution multitenant? If so, consider options that support capacity pools, where multiple database instances draw from an elastic pool of resources, instead of fixed resources per database. This can help you better distribute capacity across all database instances, and can make your solution more cost effective.
- Does your data need to be readable with low latency in multiple regions? If yes, choose an option that supports readable secondary replicas.
- Does your database need to be highly available across geo-graphic regions? If yes, choose an option that supports geographic replication. Also consider the options that support automatic failover from the primary replica to a secondary replica.
- Does your database have specific security needs? If yes, examine the options that provide capabilities like row level security, data masking, and transparent data encryption.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Is Managed Service	Yes	No	Yes	Yes

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Runs on Platform	N/A	Windows, Linux, Docker	N/A	N/A
Programmability [1]	T-SQL, .NET, R	T-SQL, .NET, R, Python	SQL	SQL, PL/pgSQL, PL/JavaScript (v8)

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

Scalability capabilities

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Maximum database instance size	4 TB	256 TB	16 TB	16 TB
Supports capacity pools	Yes	Yes	No	No
Supports clusters scale out	No	Yes	No	No
Dynamic scalability (scale up)	Yes	No	Yes	Yes

Analytic workload capabilities

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Temporal tables	Yes	Yes	No	No
In-memory (memory-optimized) tables	Yes	Yes	No	No
Columnstore support	Yes	Yes	No	No

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Adaptive query processing	Yes	Yes	No	No

Availability capabilities

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Readable secondaries	Yes	Yes	Yes	Yes
Geographic replication	Yes	Yes	Yes	Yes
Automatic failover to secondary	Yes	No	No	No
Point-in-time restore	Yes	Yes	Yes	Yes

Security capabilities

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Row level security	Yes	Yes	Yes	Yes
Data masking	Yes	Yes	No	No
Transparent data encryption	Yes	Yes	Yes	Yes
Restrict access to specific IP addresses	Yes	Yes	Yes	Yes
Restrict access to allow VNet access only	Yes	Yes	Yes	Yes

Capability	Azure SQL Database	SQL Server in an Azure virtual machine	Azure Database for MySQL	Azure Database for PostgreSQL
Azure Active Directory authentication	Yes	No	Yes	Yes
Active Directory authentication	No	Yes	No	No
Multi-factor authentication	Yes	No	Yes	Yes
Supports Always Encrypted	Yes	Yes	No	No
Private IP	No	Yes	No	No

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [Introduction to Memory-Optimized Tables](#)
- [In-Memory OLTP overview and usage scenarios](#)
- [Optimize performance by using in-memory technologies in Azure SQL Database and Azure SQL Managed Instance](#)
- [Distributed transactions across cloud databases](#)

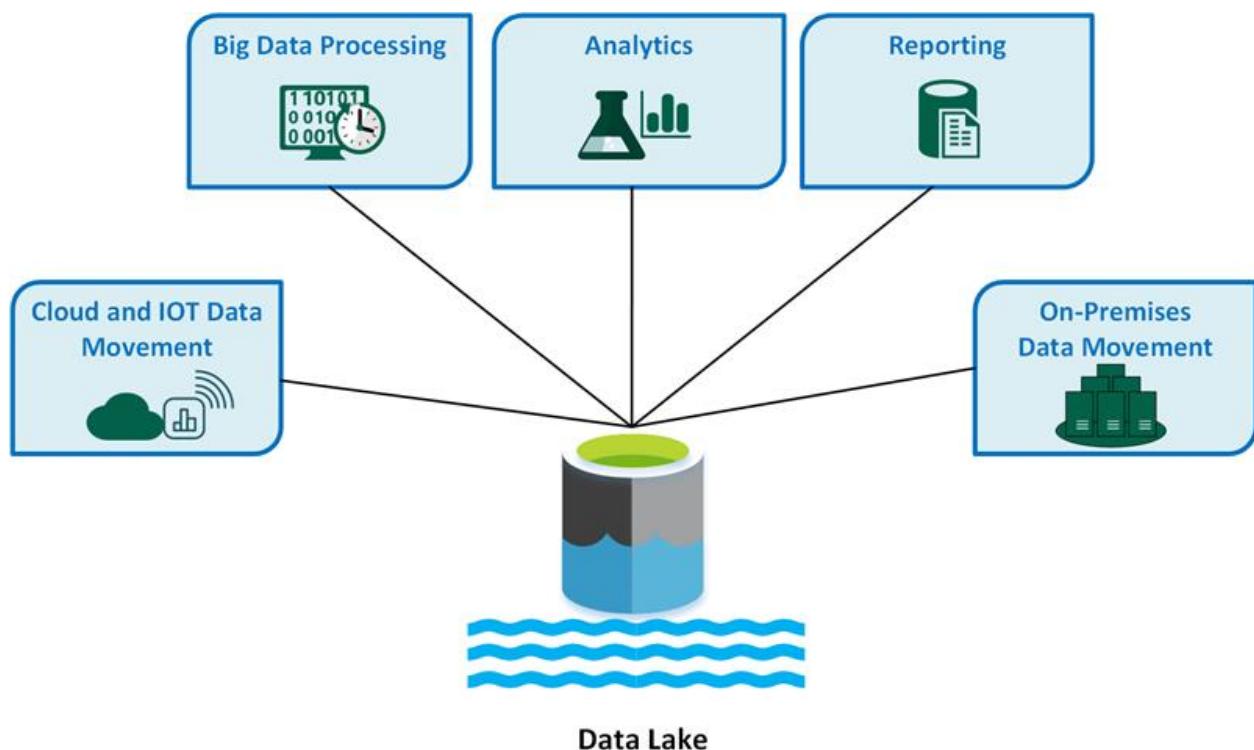
Related resources

- [Azure Data Architecture Guide](#)
- [Databases architecture design](#)
- [Scalable order processing](#)
- [IBM z/OS online transaction processing on Azure](#)

What is a data lake?

Article • 12/09/2022

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.



The following are key data lake use cases:

- Cloud and IoT data movement
- Big data processing
- Analytics
- Reporting
- On-premises data movement

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.

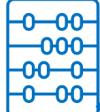
When to use a data lake

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.

The following table compares data lakes and data warehouses:

AREA	DATA LAKE	DATA WAREHOUSE
Data Store  10101 01010 00100	<p>It can capture and retain unstructured, semi-structured, and structured data in its raw format. A Data Lake stores all types of data, irrespective of the source and structure.</p>	<p>It can capture and retain only structured data. A Data Warehouse stores data in quantitative metrics with their attributes. Data is transformed and cleansed.</p>
Schema Definition 	<p>Typically, the schema is defined after data is stored. This offers high agility and data capture quite easily, but it requires work at the end of the process (schema-on-read).</p>	<p>Typically, a schema is defined prior to when data is stored. It requires work at the start of the process, but it offers performance, security, and integration (schema-on-write).</p>
Data Quality 	<p>Any data that may or may not be curated (such as raw data).</p>	<p>Highly curated data that serves as the central version of the truth.</p>
Users 	<p>A Data Lake is ideal for the users who indulge in deep analysis, like Data Scientists, Data Engineers, and Data Analysts.</p>	<p>A Data Warehouse is ideal for operational users like Business Analysts because of being well structured and easy to use and understand.</p>
Price & Performance 	<p>The storage cost is relatively low, compared to a Data Warehouse, and querying results is better.</p>	<p>The storage cost is high, and querying results is time consuming.</p>
Accessibility 	<p>A Data Lake has few constraints and is easily accessible. Data can be changed and updated quickly.</p>	<p>A Data Warehouse is structured by design, which makes it difficult to access and manipulate.</p>

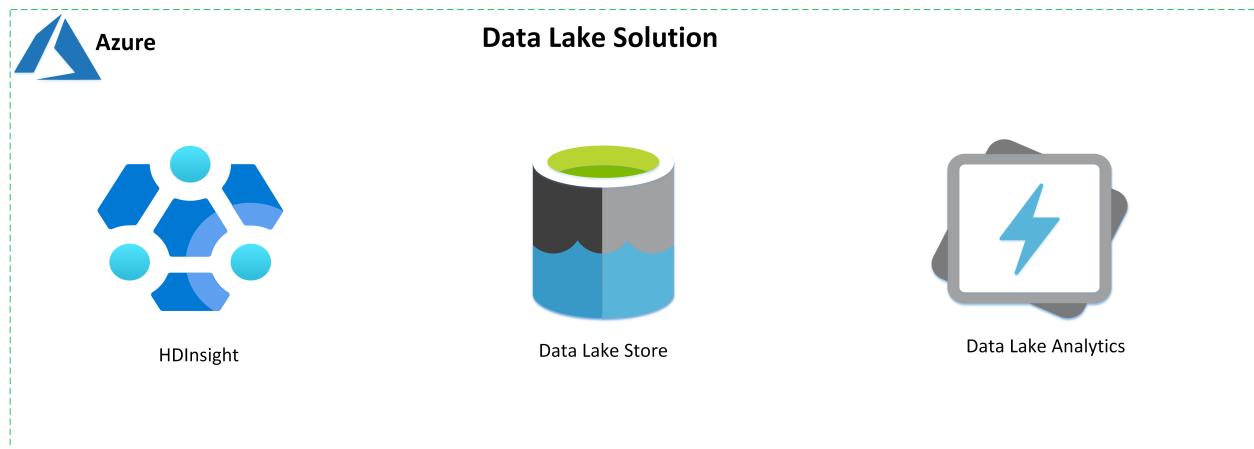
Challenges

- 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.

Technology choices

Build data lake solutions using the following services offered by Azure:



- [Azure HD Insight](#) is a managed, full-spectrum, open-source analytics service in the cloud for enterprises.
- [Azure Data Lake Store](#) is a hyperscale, Hadoop-compatible repository.
- [Azure Data Lake Analytics](#) is an on-demand analytics job service to simplify big data analytics.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Avijit Prasad](#) | Cloud Consultant

Next steps

- [What is Azure HDInsight?](#)
- [Introduction to Azure Data Lake Storage](#)
- [Azure Data Lake Analytics Documentation](#)
- [Introduction to Azure Data Lake Storage \(training module\)](#)
- [What is a Data Lake?](#)

Related resources

- Choose an analytical data store in Azure
- Query a data lake or lakehouse by using Azure Synapse serverless
- Optimized storage – time based with Data Lake
- Data management across Azure Data Lake with Microsoft Purview
- Modern data warehouse for small and medium business

Choose a data pipeline orchestration technology in Azure

Article • 12/16/2022

Most big data solutions consist of repeated data processing operations, encapsulated in workflows. A pipeline orchestrator is a tool that helps to automate these workflows. An orchestrator can schedule jobs, execute workflows, and coordinate dependencies among tasks.

What are your options for data pipeline orchestration?

In Azure, the following services and tools will meet the core requirements for pipeline orchestration, control flow, and data movement:

- [Azure Data Factory](#)
- [Oozie on HDInsight](#)
- [SQL Server Integration Services \(SSIS\)](#)

These services and tools can be used independently from one another, or used together to create a hybrid solution. For example, the Integration Runtime (IR) in Azure Data Factory V2 can natively execute SSIS packages in a managed Azure compute environment. While there is some overlap in functionality between these services, there are a few key differences.

Key Selection Criteria

To narrow the choices, start by answering these questions:

- Do you need big data capabilities for moving and transforming your data? Usually this means multi-gigabytes to terabytes of data. If yes, then narrow your options to those that best suited for big data.
- Do you require a managed service that can operate at scale? If yes, select one of the cloud-based services that aren't limited by your local processing power.
- Are some of your data sources located on-premises? If yes, look for options that can work with both cloud and on-premises data sources or destinations.

- Is your source data stored in Blob storage on an HDFS filesystem? If so, choose an option that supports Hive queries.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

Capability	Azure Data Factory	SQL Server Integration Services (SSIS)	Oozie on HDInsight
Managed	Yes	No	Yes
Cloud-based	Yes	No (local)	Yes
Prerequisite	Azure Subscription	SQL Server	Azure Subscription, HDInsight cluster
Management tools	Azure Portal, PowerShell, CLI, .NET SDK	SSMS, PowerShell	Bash shell, Oozie REST API, Oozie web UI
Pricing	Pay per usage	Licensing / pay for features	No additional charge on top of running the HDInsight cluster

Pipeline capabilities

Capability	Azure Data Factory	SQL Server Integration Services (SSIS)	Oozie on HDInsight
Copy data	Yes	Yes	Yes
Custom transformations	Yes	Yes	Yes (MapReduce, Pig, and Hive jobs)
Azure Machine Learning scoring	Yes	Yes (with scripting)	No
HDInsight On-Demand	Yes	No	No
Azure Batch	Yes	No	No
Pig, Hive, MapReduce	Yes	No	Yes

Capability	Azure Data Factory	SQL Server Integration Services (SSIS)	Oozie on HDInsight
Spark	Yes	No	No
Execute SSIS Package	Yes	Yes	No
Control flow	Yes	Yes	Yes
Access on-premises data	Yes	Yes	No

Scalability capabilities

Capability	Azure Data Factory	SQL Server Integration Services (SSIS)	Oozie on HDInsight
Scale up	Yes	No	No
Scale out	Yes	No	Yes (by adding worker nodes to cluster)
Optimized for big data	Yes	No	Yes

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [Pipelines and activities in Azure Data Factory and Azure Synapse Analytics](#)
- [Provision the Azure-SSIS integration runtime in Azure Data Factory](#)
- [Oozie on HDInsight](#)

Related resources

- [Move data from a SQL Server database to SQL Database with Azure Data Factory](#)
- [Data management patterns](#)

- DataOps for the modern data warehouse

Transfer data to and from Azure

Article • 12/16/2022

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 more network bandwidth is cost-prohibitive.
- Security or organizational policies don't allow outbound connections when dealing with sensitive data.

If your primary concern is how long it takes to transfer your data, you might want to run a test to verify whether network transfer is slower than physical transport.

There are two main options for physically transporting data to Azure:

The Azure Import/Export service

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 the drives shipped to you for loading on-premises.

Azure Data Box

[Azure Data Box](#) is a Microsoft-provided appliance that works much like the Import/Export service. With Data Box, 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 Data Box service is ease of use. You don't need to purchase several hard drives, prepare them, and transfer files to each one. Data Box is supported by many industry-leading Azure partners to make it easier to seamlessly use offline transport to the cloud from their products.

Command-line tools and APIs

Consider these options when you want scripted and programmatic data transfer:

- The [Azure CLI](#) is a cross-platform tool that allows you to manage Azure services and upload data to Storage.
- **AzCopy**. Use AzCopy from a [Windows](#) or [Linux](#) command line to easily copy data to and from Blob Storage, Azure File Storage, and Azure Table Storage with optimal performance. AzCopy supports concurrency and parallelism, and the ability to resume copy operations when interrupted. You can also use AzCopy to copy data from AWS to Azure. For programmatic access, the [Microsoft Azure Storage Data Movement Library](#) is the core framework that powers AzCopy. It's provided as a .NET Core library.
- With [PowerShell](#), the [Start-AzureStorageBlobCopy](#) PowerShell cmdlet is an option for Windows administrators who are used to PowerShell.
- [AdlCopy](#) enables you to copy data from Blob Storage into Azure Data Lake Storage. It can also be used to copy data between two Data Lake Storage accounts. However, it can't be used to copy data from Data Lake Storage to Blob Storage.
- [Distcp](#) is used to copy data to and from an HDInsight cluster storage (WASB) into a Data Lake Storage account.
- [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 tools. Sqoop works with HDInsight clusters by using either Blob Storage or Data Lake Storage attached storage.
- [PolyBase](#) is a technology that accesses data outside a database through the T-SQL language. In SQL Server 2016, it allows you to run queries on external data in Hadoop or to import or export data from Blob Storage. In Azure Synapse Analytics, you can import or export data from Blob Storage and Data Lake Storage. Currently, PolyBase is the fastest method of importing data into Azure Synapse Analytics.
- Use the [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 Blob Storage or Data Lake Storage. 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're only transferring a few files or data objects and don't need to automate the process.

- [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, and upload and download blobs between your local file system and Blob Storage, or between storage accounts.
- **Azure portal.** Both Blob Storage and Data Lake Storage provide a web-based interface for exploring files and uploading new files. This option is a good one if you don't want to install tools or issue commands to quickly explore your files, or if you want to upload a handful of new ones.

Data sync and pipelines

- [Azure Data Factory](#) is a managed service best suited for regularly transferring files between many Azure services, on-premises systems, or a combination of the two. By using Data Factory, you can create and schedule data-driven workflows called pipelines that ingest data from disparate data stores. Data Factory can process and transform the data by using compute services such as Azure HDInsight Hadoop, Spark, Azure Data Lake Analytics, and Azure Machine Learning. You can create data-driven workflows for [orchestrating](#) and automating data movement and data transformation.
- [Pipelines and activities](#) in Data Factory and Azure Synapse Analytics can be used to construct end-to-end data-driven workflows for your data movement and data processing scenarios. Additionally, the [Azure Data Factory integration runtime](#) is used to provide data integration capabilities across different network environments.
- [Azure Data Box Gateway](#) transfers data to and from Azure, but it's a virtual appliance, not a hard drive. Virtual machines residing in your on-premises network write data to Data Box Gateway by using the NFS and SMB protocols. The device then transfers your data to Azure.

Key selection criteria

For data transfer scenarios, choose the appropriate system for your needs by answering these questions:

- Do you need to transfer 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're reusable? If so, select one of the command-line options or Data Factory.
- Do you need to transfer a large amount of data over a network connection? If so, select an option that's 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. Some of these options also require a Hadoop cluster.
- Do you need an automated data pipeline or workflow orchestration? If yes, consider Data Factory.

Capability matrix

The following tables summarize the key differences in capabilities.

Physical transfer

Capability	The Import/Export service	Data Box
Form factor	Internal SATA HDDs or SSDs	Secure, tamper-proof, single hardware appliance
Microsoft manages shipping logistics	No	Yes
Integrates with partner products	No	Yes
Custom appliance	No	Yes

Command-line tools

Hadoop/HDInsight:

Capability	Distcp	Sqoop	Hadoop CLI
Optimized for big data	Yes	Yes	Yes

Capability	Distcp	Sqoop	Hadoop CLI
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 Storage	Yes	Yes	Yes
Copy from Data Lake Storage	Yes	Yes	No

Other:

Capability	Azure CLI	AzCopy	PowerShell	AdlCopy	PolyBase
Compatible platforms	Linux, OS X, Windows	Linux, Windows	Windows	Linux, OS X, Windows	SQL Server, Azure Synapse Analytics
Optimized for big data	No	Yes	No	Yes ¹	Yes ²
Copy to relational database	No	No	No	No	Yes
Copy from relational database	No	No	No	No	Yes
Copy to Blob Storage	Yes	Yes	Yes	No	Yes
Copy from Blob Storage	Yes	Yes	Yes	Yes	Yes
Copy to Data Lake Storage	No	Yes	Yes	Yes	Yes
Copy from Data Lake Storage	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 interfaces, data sync, and data pipelines

Capability	Azure Storage Explorer	Azure portal *	Data Factory	Data Box Gateway
Optimized for big data	No	No	Yes	Yes
Copy to relational database	No	No	Yes	No
Copy from relational database	No	No	Yes	No
Copy to Blob Storage	Yes	No	Yes	Yes
Copy from Blob Storage	Yes	No	Yes	No
Copy to Data Lake Storage	No	No	Yes	No
Copy from Data Lake Storage	No	No	Yes	No
Upload to Blob Storage	Yes	Yes	Yes	Yes
Upload to Data Lake Storage	Yes	Yes	Yes	Yes
Orchestrate data transfers	No	No	Yes	No
Custom data transformations	No	No	Yes	No
Pricing model	Free	Free	Pay per usage	Pay per unit

* Azure portal in this case represents the web-based exploration tools for Blob Storage and Data Lake Storage.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [What is Azure Import/Export service?](#)
- [What is Azure Data Box?](#)
- [What is the Azure CLI?](#)
- [Get started with AzCopy](#)
- [Get started with Storage Explorer](#)
- [What is Azure Data Factory?](#)
- [What is Azure Data Box Gateway?](#)

Related resources

- [Move archive data from mainframe systems to Azure](#)
- [Mainframe file replication and sync on Azure](#)
- [Replicate and sync mainframe data in Azure](#)

Choose an analytical data store in Azure

Article • 10/09/2023

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

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

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

What are your options when choosing an analytical data store?

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

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

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

- [Key/value](#) databases hold a single serialized object for each key value. They're good for storing large volumes of data where you want to get one item for a given key value and you don't have to query based on other properties of the item.
- [Document](#) databases are key/value databases in which the values are *documents*. A "document" in this context is a collection of named fields and values. The database typically stores the data in a format such as XML, YAML, JSON, or BSON, but may use plain text.

Document databases can query on non-key fields and define secondary indexes to make querying more efficient. This makes a document database more suitable for applications that need to retrieve data based on criteria more complex than the value of the document key. For example, you could query on fields such as product ID, customer ID, or customer name.

- **Column store** databases are key/value data stores that store each column separately on disk. A *wide column store* database is a type of column store database that stores *column families*, not just single columns. For example, a census database might have a column family for a person's name (first, middle, last), a family for the person's address, and a family for the person's profile information (date of birth, gender). The database can store each column family in a separate partition, while keeping all the data for one person related to the same key. An application can read a single column family without reading through all of the data for an entity.
- **Graph** databases store information as a collection of objects and relationships. A graph database can efficiently perform queries that traverse the network of objects and the relationships between them. For example, the objects might be employees in a human resources database, and you might want to facilitate queries such as "find all employees who directly or indirectly work for Scott."
- Telemetry and time series databases are an append-only collection of objects. Telemetry databases efficiently index data in a variety of column stores and in-memory structures, making them the optimal choice for storing and analyzing vast quantities of telemetry and time series data.

Key selection criteria

To narrow the choices, start by answering these questions:

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

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

Capability	SQL Database	Azure Synapse SQL pool	Azure Synapse Spark pool	Azure Data Explorer	HBase/Phoenix on HDInsight	Hive LLAP on HDInsight	Azure Analysis Services	Azure Cosmos DB
Is managed service	Yes	Yes	Yes	Yes	Yes ¹	Yes ¹	Yes	Yes
Primary database model	Relational (column store format when using columnstore indexes)	Relational tables with column storage	Wide column store	Relational (column store), telemetry, and time series store	Wide column store	Hive/In-Memory	Tabular semantic models	Document store, graph, key-value store, wide column store
SQL language support	Yes	Yes	Yes	Yes	Yes (using Phoenix ² JDBC driver)	Yes	No	Yes
Optimized for speed serving layer	Yes ²	Yes ³	Yes	Yes	Yes	Yes	No	Yes

[1] With manual configuration and scaling.

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

[3] Supported as an Azure Stream Analytics output.

Scalability capabilities

Capability	SQL Database	Azure Synapse SQL pool	Azure Synapse Spark pool	Azure Data Explorer	HBase/Phoenix on HDInsight	Hive LLAP on HDInsight	Azure Analysis Services	Azure Cosmos DB
Redundant regional servers for high availability	Yes	No	No	Yes	Yes	No	Yes	Yes
Supports query scale-out	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic scalability (scale up)	Yes	Yes	Yes	Yes	No	No	Yes	Yes

Capability	SQL Database	Azure Synapse SQL pool	Azure Synapse Spark pool	Azure Data Explorer	HBase/Phoenix on HDInsight	Hive LLAP on HDInsight	Azure Analysis Services	Azure Cosmos DB
Supports in-memory caching of data	Yes	Yes	Yes	Yes	No	Yes	Yes	No

Security capabilities

Capability	SQL Database	Azure Synapse	Azure Data Explorer	HBase/Phoenix on HDInsight	Hive LLAP on HDInsight	Azure Analysis Services	Azure Cosmos DB
Authentication	SQL / Microsoft Entra ID	SQL / Microsoft Entra ID	Microsoft Entra ID	local / Microsoft Entra ID ¹	local / Microsoft Entra ID ¹	Microsoft Entra ID	database users / Microsoft Entra ID via access control (IAM)
Data encryption at rest	Yes ²	Yes ²	Yes	Yes ¹	Yes ¹	Yes	Yes
Row-level security	Yes	Yes ³	Yes	Yes ¹	Yes ¹	Yes	No
Supports firewalls	Yes	Yes	Yes	Yes ⁴	Yes ⁴	Yes	Yes
Dynamic data masking	Yes	Yes	Yes	Yes ¹	Yes	No	No

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

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

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

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

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Zoiner Tejada ✉ | CEO and Architect

Next steps

- Analyze data in a relational data warehouse
- Create a single database - Azure SQL Database
- Create an Azure Databricks workspace
- Create Apache Spark cluster in Azure HDInsight using Azure portal
- Creating a Synapse workspace
- Explore Azure data services for modern analytics
- Explore Azure database and analytics services
- Query Azure Cosmos DB by using the API for NoSQL

Related resources

- Technology choices for Azure solutions
- Advanced analytics architecture
- Analyze operational data on MongoDB Atlas using Azure Synapse Analytics
- Non-relational data and NoSQL

Choose a data analytics and reporting technology in Azure

Article • 03/14/2023

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 ↗](#)
- [Jupyter Notebooks in VS Code ↗](#)

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.

Jupyter Notebooks in VS Code

VS Code is a free code editor and development platform that you can use locally or connected to remote compute. Combined with the Jupyter extension, it offers a full environment for Jupyter development that can be enhanced with additional language extensions. If you want a best-in-class, free Jupyter experience with the ability to leverage your compute of choice, this is a great option. Using VS Code, you can develop and run notebooks against remotes and containers. To make the transition easier from Azure Notebooks, we have made the container image available so it can be used with VS Code too.

Jupyter (formerly IPython Notebook) is an open-source project that lets you easily combine Markdown text and executable Python source code on one canvas called a notebook. Visual Studio Code supports working with Jupyter Notebooks natively, and through Python code files.

Key selection criteria

To narrow the choices, start by answering these questions:

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

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

Capability	Power BI	Jupyter Notebooks	Zeppelin Notebooks	Jupyter Notebooks in VS Code
Connect to big data cluster for advanced processing	Yes	Yes	Yes	No
Managed service	Yes	Yes ¹	Yes ¹	Yes
Connect to 100s of data sources	Yes	No	No	No
Offline capabilities	Yes ²	No	No	No
Embedding capabilities	Yes	No	No	No
Automatic data refresh	Yes	No	No	No
Access to numerous open source packages	No	Yes ³	Yes ³	Yes ⁴
Data transformation/cleansing options	Power Query ² , R	40 languages, including Python, R, Julia, and Scala	20+ interpreters, including Python, JDBC, and R	Python, F#, R

Capability	Power BI	Jupyter Notebooks	Zeppelin Notebooks	Jupyter Notebooks in VS Code
Pricing	Free for Power BI Desktop (authoring), see pricing ↗ for hosting options	Free	Free	Free
Multiuser collaboration	Yes	Yes (through sharing or with a multiuser server like JupyterHub ↗)	Yes	Yes (through sharing)

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

[2] With the use of Power BI Desktop.

[2] You can search the [Maven repository ↗](#) for community-contributed packages.

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

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada ↗](#) | CEO and Architect

Next steps

- [Get started with Jupyter notebooks for Python](#)
- [Notebooks](#)
- [Run Azure Databricks Notebooks with Azure Data Factory](#)
- [Run Jupyter notebooks in your workspace](#)
- [What is Power BI?](#)

Related resources

- Advanced analytics architecture
- Data analysis and visualization in an Azure industrial IoT analytics solution
- Technology choices for Azure solutions

Advanced analytics

Article • 12/16/2022

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](#)

Choose a batch processing technology in Azure

Article • 10/09/2023

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 fundamental requirement of such batch processing engines is to scale out computations to handle a large volume of data. Unlike real-time processing, batch processing is expected to have latencies (the time between data ingestion and computing a result) that measure in minutes to hours.

Technology choices for batch processing

Azure Synapse Analytics

[Azure Synapse](#) is a distributed system designed to perform analytics on large data. It supports massive parallel processing (MPP), which makes it suitable for running high-performance analytics. Consider Azure Synapse when you have large amounts of data (more than 1 TB) and are running an analytics workload that will benefit from parallelism.

Azure Data Lake Analytics

[Data Lake Analytics](#) is an on-demand analytics job service. It's optimized for distributed processing of large data sets stored in Azure Data Lake Store.

- Languages: [U-SQL](#) (including Python, R, and C# extensions).
- Integrates with Azure Data Lake Store, Azure Storage blobs, Azure SQL Database, and Azure Synapse.
- Pricing model is per-job.

HDInsight

HDInsight is a managed Hadoop service. Use it to deploy and manage Hadoop clusters in Azure. For batch processing, you can use [Spark](#), [Hive](#), [Hive LLAP](#), [MapReduce](#).

- Languages: R, Python, Java, Scala, SQL
- Kerberos authentication with Active Directory, Apache Ranger-based access control
- Gives you complete control of the Hadoop cluster

Azure Databricks

[Azure Databricks](#) is an Apache Spark-based analytics platform. You can think of it as "Spark as a service." It's the easiest way to use Spark on the Azure platform.

- Languages: R, Python, Java, Scala, Spark SQL
- Fast cluster start times, autotermination, autoscaling.
- Manages the Spark cluster for you.
- Built-in integration with Azure Blob Storage, Azure Data Lake Storage (ADLS), Azure Synapse, and other services. See [Data Sources](#).
- User authentication with Microsoft Entra ID.
- Web-based [notebooks](#) for collaboration and data exploration.
- Supports [GPU-enabled clusters](#)

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Do you want to author batch processing logic declaratively or imperatively?
- Will you perform batch processing in bursts? If yes, consider options that let you auto-terminate the cluster or whose pricing model is per batch job.
- Do you need to query relational data stores along with your batch processing, for example, to look up reference data? If yes, consider the options that enable the querying of external relational stores.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

Capability	Azure Data Lake Analytics	Azure Synapse	HDIInsight	Azure Databricks
Is managed service	Yes	Yes	Yes ¹	Yes
Relational data store	Yes	Yes	No	No
Pricing model	Per batch job	By cluster hour	By cluster hour	Databricks Unit ² + cluster hour

[1] With manual configuration.

[2] A Databricks Unit (DBU) is a unit of processing capability per hour.

Capabilities

Capability	Azure Data Lake Analytics	Azure Synapse	HDIInsight with Spark	HDIInsight with Hive	HDIInsight with Hive LLAP	Azure Databricks
Autoscaling	No	No	Yes	Yes	Yes	Yes
Scale-out granularity	Per job	Per cluster	Per cluster	Per cluster	Per cluster	Per cluster
In-memory caching of data	No	Yes	Yes	No	Yes	Yes
Query from external relational stores	Yes	No	Yes	No	No	Yes
Authentication	Microsoft Entra ID	SQL / Microsoft Entra ID	No	Microsoft Entra ID ¹	Microsoft Entra ID ¹	Microsoft Entra ID
Auditing	Yes	Yes	No	Yes ¹	Yes ¹	Yes
Row-level security	No	Yes ²	No	Yes ¹	Yes ¹	No
Supports firewalls	Yes	Yes	Yes	Yes ³	Yes ³	No
Dynamic data masking	No	Yes	No	Yes ¹	Yes ¹	No

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

[2] Filter predicates only. See [Row-Level Security](#)

[3] Supported when [used within an Azure Virtual Network](#).

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- Create a lake database in Azure Synapse Analytics
- Create an Azure Databricks workspace
- Explore Azure Databricks
- Get started with Azure Data Lake Analytics using the Azure portal
- Introduction to Azure Synapse Analytics
- What is Azure Databricks?
- What is Azure Synapse Analytics?

Related resources

- [Analytics architecture design](#)
- [Choose an analytical data store in Azure](#)
- [Choose a data analytics technology in Azure](#)
- [Analytics end-to-end with Azure Synapse](#)
- [Batch processing](#)

Choose a stream processing technology in Azure

Article • 02/02/2023

This article compares technology choices for real-time stream processing in Azure.

Real-time stream processing consumes messages from either queue or file-based storage, processes the messages, and forwards the result to another message queue, file store, or database. Processing may include querying, filtering, and aggregating messages. Stream processing engines must be able to consume endless streams of data and produce results with minimal latency. For more information, see [Real time processing](#).

What are your options when choosing a technology for real-time processing?

In Azure, all of the following data stores will meet the core requirements supporting real-time processing:

- [Azure Stream Analytics](#)
- [HDInsight with Spark Streaming](#)
- [Apache Spark in Azure Databricks](#)
- [HDInsight with Storm](#)
- [Azure Functions](#)
- [Azure App Service WebJobs](#)
- [Apache Kafka streams API](#)

Key Selection Criteria

For real-time processing scenarios, begin choosing the appropriate service for your needs by answering these questions:

- Do you prefer a declarative or imperative approach to authoring stream processing logic?
- Do you need built-in support for temporal processing or windowing?
- Does your data arrive in formats besides Avro, JSON, or CSV? If yes, consider options that support any format using custom code.

- Do you need to scale your processing beyond 1 GB/s? If yes, consider the options that scale with the cluster size.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

Capability	Azure Stream Analytics	HDInsight with Spark Streaming	Apache Spark in Azure Streaming	HDInsight with Storm	Azure Functions	Azure App Service WebJobs
Programmability	SQL, JavaScript	C#/F# ↗ , Java, Python, Scala	C#/F# ↗ , Java, Python, R, Scala	C#, Java	C#, F#, Java, Node.js, Python	C#, Java, Node.js, PHP, Python
Programming paradigm	Declarative	Mixture of declarative and imperative	Mixture of declarative and imperative	Imperative	Imperative	Imperative
Pricing model	Streaming units ↗	Per cluster hour	Databricks units ↗	Per cluster hour	Per function execution and resource consumption	Per app service plan hour

Integration capabilities

Capability	Azure Stream Analytics	HDInsight with Spark Streaming	Apache Spark in Azure Streaming	HDInsight with Storm	Azure Functions	Azure App Service WebJobs

Capability	Azure Stream Analytics	HDInsight with Spark Streaming	Apache Spark in Azure Databricks	HDInsight with Storm	Azure Functions	Azure App Service WebJobs
Inputs	Azure Event Hubs, Azure IoT Hub, Azure Blob storage/ADLS Gen2	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, Azure Cosmos DB, Files
Sinks	Azure Data Lake Storage Gen 1, Azure Data Explorer, Azure Database for PostgreSQL, Azure SQL Database, Azure Synapse Analytics, Blob storage and Azure Data Lake Gen 2, Azure Event Hubs, Power BI, Azure Table storage, Azure Service Bus queues, Azure Service Bus topics, Azure Cosmos DB, Azure Functions	HDFS, Kafka, Storage Blobs, Azure Data Lake Store, Store, Azure Cosmos DB	HDFS, Kafka, Storage Blobs, Azure Data Lake Store, Azure	Event Hubs, Service Bus, Kafka	Supported bindings	Service Bus, Storage Queues, Storage Blobs, Event Hubs, WebHooks, Azure Cosmos DB, Files

Processing capabilities

Capability	Azure Stream Analytics	HDInsight with Spark Streaming	Apache Spark in Azure Databricks	HDInsight with Storm	Azure Functions	Azure App Service WebJobs
Built-in temporal/windowing support	Yes	Yes	Yes	Yes	No	No
Input data formats	Avro, JSON or CSV, UTF-8 encoded	Any format using custom code	Any format using custom code	Any format using custom code	Any format using custom code	Any format using custom code
Scalability	Query partitions	Bounded by cluster size	Bounded by Databricks cluster scale configuration	Bounded by cluster size	Up to 200 function app instances processing in parallel	Bounded by app service plan capacity
Late arrival and out of order event handling support	Yes	Yes	Yes	Yes	No	No

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- [App Service overview](#)
- [Explore Azure Functions](#)
- [Get started with Azure Stream Analytics](#)
- [Perform advanced streaming data transformations](#)
- [Set up clusters in HDInsight](#)
- [Use Apache Spark in Azure Databricks](#)

Related resources

- Choose a real-time message ingestion technology
- Real time processing
- Stream processing with Azure Stream Analytics

Choose a real-time analytics and streaming processing technology on Azure

Article • 05/24/2022

There are several services available for real-time analytics and streaming processing on Azure. This article provides the information you need to decide which technology is the best fit for your application.

When to use Azure Stream Analytics

Azure Stream Analytics is the recommended service for stream analytics on Azure. It's meant for a wide range of scenarios that include but aren't limited to:

- Dashboards for data visualization
- Real-time [alerts](#) from temporal and spatial patterns or anomalies
- Extract, Transform, Load (ETL)
- [Event Sourcing pattern](#)
- [IoT Edge](#)

Adding an Azure Stream Analytics job to your application is the fastest way to get streaming analytics up and running in Azure, using the SQL language you already know. Azure Stream Analytics is a job service, so you don't have to spend time managing clusters, and you don't have to worry about downtime with a 99.9% SLA at the job level. Billing is also done at the job level making startup costs low (one Streaming Unit), but scalable (up to 396 Streaming Units). It's much more cost effective to run a few Stream Analytics jobs than it is to run and maintain a cluster.

Azure Stream Analytics has a rich out-of-the-box experience. You can immediately take advantage of the following features without any additional setup:

- Built-in temporal operators, such as [windowed aggregates](#), temporal joins, and temporal analytic functions.
- Native Azure [input](#) and [output](#) adapters
- Support for slow changing [reference data](#) (also known as a lookup tables), including joining with geospatial reference data for geofencing.
- Integrated solutions, such as [Anomaly Detection](#)
- Multiple time windows in the same query
- Ability to compose multiple temporal operators in arbitrary sequences.

- Under 100-ms end-to-end latency from input arriving at Event Hubs, to output landing in Event Hubs, including the network delay from and to Event Hubs, at sustained high throughput

When to use other technologies

You want to write UDFs, UDAs, and custom deserializers in a language other than JavaScript or C#

Azure Stream Analytics supports user-defined functions (UDF) or user-defined aggregates (UDA) in JavaScript for cloud jobs and C# for IoT Edge jobs. C# user-defined deserializers are also supported. If you want to implement a deserializer, a UDF, or a UDA in other languages, such as Java or Python, you can use Spark Structured Streaming. You can also run the Event Hubs `EventProcessorHost` on your own virtual machines to do arbitrary streaming processing.

Your solution is in a multi-cloud or on-premises environment

Azure Stream Analytics is Microsoft's proprietary technology and is only available on Azure. If you need your solution to be portable across Clouds or on-premises, consider open-source technologies such as Spark Structured Streaming or Storm.

Next steps

- [Create a Stream Analytics job by using the Azure portal](#)
- [Create a Stream Analytics job by using Azure PowerShell](#)
- [Create a Stream Analytics job by using Visual Studio](#)
- [Create a Stream Analytics job by using Visual Studio Code](#)

Choose an Azure Cognitive Services technology

Article • 05/31/2023

Azure Cognitive Services is a set of cloud-based APIs that you can use in AI applications and data flows. It provides pretrained models that are ready to use in your applications, requiring no data and no model training on your part. The services are developed by the Microsoft AI and Research team and expose the latest deep learning algorithms. They're consumed over HTTP REST interfaces. In addition, SDKs are available for many common application development frameworks.

Key benefits:

- Minimal development effort for state-of-the-art AI services. Use predefined algorithms or create custom algorithms on top of pre-built libraries.
- Easy integration into apps via HTTP REST interfaces.
- Developers and data scientists of all skill levels can easily add AI capabilities to apps.

Considerations:

- These services are 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 might not suit all predictive analytics requirements.

Categories of Azure cognitive services

Dozens of cognitive services are available in Azure. Here's a list, categorized by the functional area they support:

Service	Link to decision guide	Description
Language	Choose a language service	Language cognitive services are services that provide Natural Language Processing (NLP) features for understanding and analyzing text.

Service	Link to decision guide	Description
Speech 	Choose a speech service	Speech cognitive services are services that provide speech capabilities like speech-to-text, text-to-speech, speech translation, and speaker recognition.
Vision 	Choose a vision service	Vision cognitive services are services that provide image and video recognition capabilities.
Decision services <ul style="list-style-type: none"> • Anomaly Detector  • Content Moderator  • Personalizer  	Choose a decision API or applied AI service	Decision cognitive services are services that provide NLP features to produce recommendations for informed and efficient decision-making.
Applied AI Services <ul style="list-style-type: none"> • Azure Cognitive Search  		
Azure OpenAI Service 	N/A	Azure OpenAI Service provides REST API access to powerful OpenAI language models.

Common use cases

The following are some common use cases for Azure Cognitive Services.

Use case	Category
Transcribe audible speech into readable, searchable text.	Speech
Convert text to lifelike speech for more natural interfaces.	Speech
Integrate real-time speech translation into your apps.	Speech
Identify and verify the person speaking by using voice characteristics.	Speech
Identify commonly used and domain-specific terms.	Language
Automatically detect sentiments and opinions in text.	Language

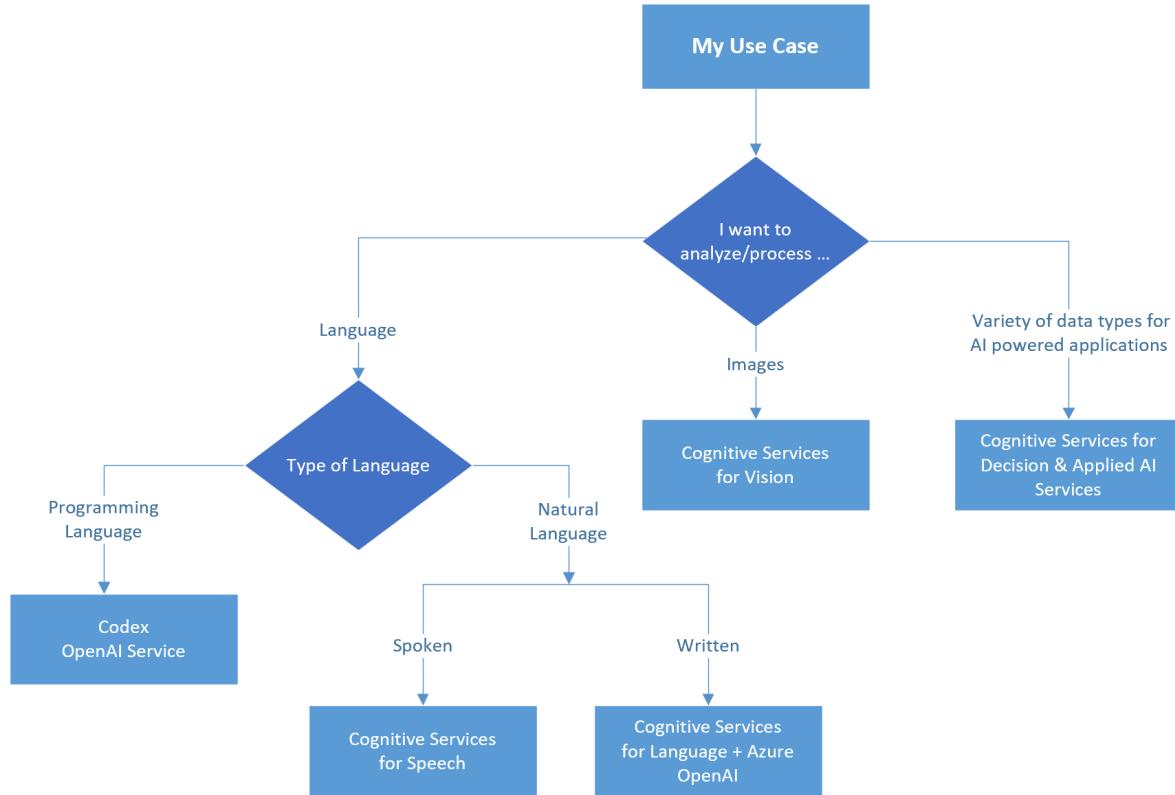
Use case	Category
Distill information into easy-to-navigate questions and answers.	Language
Enable your apps to interact with users through natural language.	Language
Translate more than 100 languages and dialects.	Language
Identify and analyze content in images and video.	Vision
Customize image recognition to fit your business needs.	Vision
Identify potential problems early.	Decision services / Anomaly Detector
Detect potentially offensive or unwanted content.	Decision services / Content Moderator
Create rich, personalized experiences for every user.	Decision services / Personalizer
Apply advanced coding and language models to various use cases.	Azure OpenAI

Key selection criteria

To narrow down the choices, start by answering these questions:

- Are you processing something related to spoken language, or are you processing text, images, or documents?
- Do you have the data to train a model? If yes, consider using the custom services that enable you to train their underlying models with data that you provide. Doing so can improve accuracy and performance.

This flow chart can help you choose the best API service for your use case.



- If your use case requires speech-to-text, text-to-speech, or speech-to-speech, use a [speech API](#).
- If your use case requires language analysis, text assessment, or text-to-text, use a [language API](#).
- If you need to analyze images, video, or text, use a [vision API](#).
- If you need to make a decision, use a [decision API](#) or [Applied AI Services](#).

Deploying services

When you [deploy Cognitive Services](#), you can either deploy services independently or use the Cognitive Services multi-service resource. The multi-service resource deploys decision, language, speech, vision, and applied AI services.

- Deploy an individual service if you don't need other services or if you want to manage access and billing on a per-service basis.
- Deploy the multi-service resource if you're using multiple services and want to manage access and billing for all services together.

(!) Note

The resource categories in these API services change frequently. Be sure to check the latest documentation for new categories.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Ashish Chahuan](#) | Senior Cloud Solution Architect
- [Kruti Mehta](#) | Azure Senior Fast-Track Engineer
- [Zoiner Tejada](#) | CEO and Architect

Other contributors:

- [Mick Alberts](#) | Technical Writer
- [Brandon Cowen](#) | Senior Cloud Solution Architect
- [Oscar Shimabukuro](#) | Senior Cloud Solution Architect
- [Manjit Singh](#) | Software Engineer
- [Christina Skarpathiotaki](#) | Senior Cloud Solution Architect
- [Nathan Widdup](#) | Azure Senior Fast-Track Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- Learning path: Provision and manage Azure Cognitive Services
- [Azure Cognitive Services documentation](#)
- [What are Azure Cognitive Services?](#)
- [Blog post: Which AI Am I?](#)

Related resources

- [Automate document processing by using Azure Form Recognizer](#)
- [Build a chatbot for hotel booking](#)
- [End-to-end computer vision at the edge for manufacturing](#)
- [Image classification on Azure](#)
- [Use a speech-to-text transcription pipeline to analyze recorded conversations](#)

Types of language API services

Article • 06/01/2023

Azure Cognitive Service for Language is a cloud-based service that provides Natural Language Processing (NLP) features for understanding and analyzing text. This service can help you build intelligent applications. It provides tools like the web-based Language Studio, REST APIs, and client libraries.

Services

Here are some details about language API services:

- [Azure Cognitive Service for Language](#) provides several NLP features for understanding and analyzing text. This service brings together Text Analytics, QnA Maker, and LUIS. These features can be:
 - Preconfigured, which means that the AI models the feature uses aren't customizable. You just send your data and use the feature's output in your applications.
 - Customizable, which means that you use Azure Cognitive Services tools to train an AI model to fit your data.
- [Azure OpenAI Service](#) provides REST API access to powerful OpenAI language models, including GPT-3, Codex, and embeddings. You can easily adopt these models to your specific task. Tasks include content generation, summarization, semantic search, and natural-language-to-code translation. You can access the service via REST APIs, a Python SDK, or the web-based interface in Azure OpenAI Studio.
- [Cognitive Services Translator](#) is a translation service that provides text-to-text APIs.
- [QnA Maker](#) is a cloud-based NLP service that you can use to create a natural conversational layer over your data.

Use cases

The following table provides recommended services for specific use cases.

Use case	Service to use	Service category
Translation		
Translate industry-specific text	Cognitive Services Custom Translator	Translator

Use case	Service to use	Service category
Translate generic text that isn't specific to an industry	Cognitive Services Translator	Translator
Translate natural language into SQL queries	Azure OpenAI natural language to SQL	Azure OpenAI
Enable apps to process and analyze natural language	Cognitive Services conversational language understanding	Language
Identification		Language
Identify sensitive information and PII	Cognitive Services Personally Identifiable Information (PII) detection	Language
Identify sensitive information, PII, and PHI	Cognitive Services Text Analytics for health	Language
Identify entities in text and categorize them into predefined types	Cognitive Services named entity recognition	Language
Extract domain-specific entities or information	Cognitive Services custom named entity recognition	Language
Extract the main key phrases from text	Cognitive Services key phrase extraction	Language
Summarize a document	Azure OpenAI GPT-3 text summarization	Azure OpenAI
Classification		
Classify text by using sentiment analysis	Cognitive Services sentiment analysis	Language
Classify text by using custom classes	Cognitive Services custom text classification	Language
Classify items into categories provided at inference time	Azure OpenAI text classification	Azure OpenAI
Classify text by detecting the language	Cognitive Services language detection	Language
Understanding context		
Link an entity with knowledge base articles	Cognitive Services Entity Linking	Language
Understand questions and answers (generic)	Cognitive Services question answering	QnA Maker

Use case	Service to use	Service category
Understand questions and answers (custom)	Cognitive Services custom question answering	QnA Maker
Combine multiple cognitive services	Cognitive Services orchestration workflow	Language

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Ashish Chahuan](#) | Senior Cloud Solution Architect
- [Kruti Mehta](#) | Azure Senior Fast-Track Engineer

Other contributors:

- [Mick Alberts](#) | Technical Writer
- [Brandon Cowen](#) | Senior Cloud Solution Architect
- [Oscar Shimabukuro](#) | Senior Cloud Solution Architect
- [Manjit Singh](#) | Software Engineer
- [Christina Skarpathiotaki](#) | Senior Cloud Solution Architect
- [Nathan Widdup](#) | Azure Senior Fast-Track Engineer

To see nonpublic LinkedIn profiles, sign in to LinkedIn.

Next steps

- [What is Azure Cognitive Service for Language?](#)
- [Language APIs blog post](#)
- [Learning path: Create a language understanding solution with Azure Cognitive Services](#)
- [Learning path: Provision and manage Azure Cognitive Services](#)
- [Learning path: Identify principles and practices for responsible AI](#)
- [Learning path: Introduction to responsible bots](#)

Related resources

- Types of decision APIs and Applied AI Services
- Types of speech API services
- Types of vision API services

Types of speech API services

Article • 06/01/2023

You can use the Azure Cognitive Services Speech service to perform spoken language transformations, including speech-to-text, text-to-speech, speech translation, and speaker recognition.

ⓘ Note

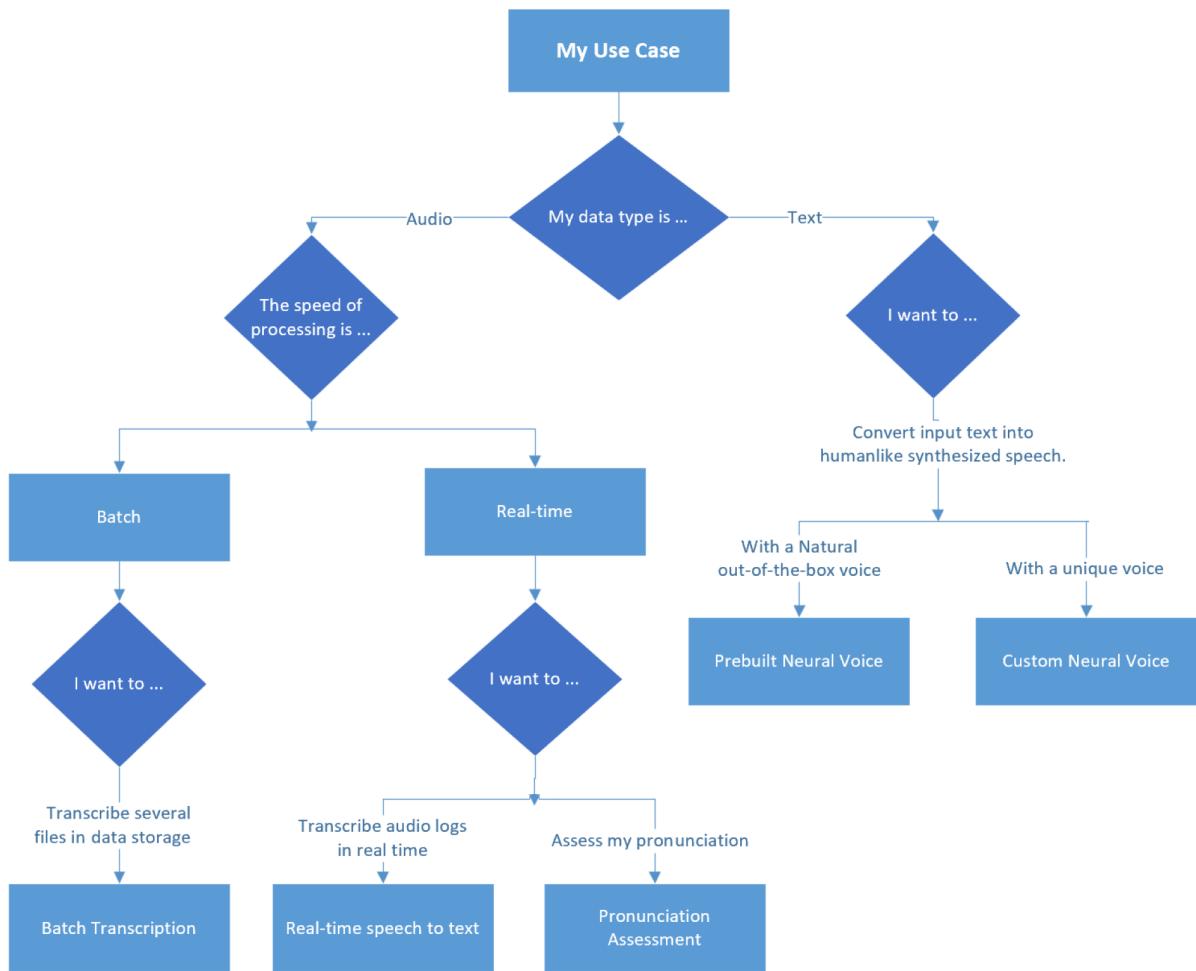
Use [Azure Cognitive Service for Language](#) if you want to gather insights on terms or phrases or get detailed contextual analysis of spoken or written language.

Services

- [Speech-to-text](#) can convert audio streams to text in real time or in batch.
- [Text-to-speech](#) enables applications to convert text to human-like speech.
- [Speech translation](#) provides multi-language speech-to-speech and speech-to-text translation of audio streams.

How to choose a speech service

This flow chart can help you choose the speech service that suits your needs:



The left side of the diagram illustrates audio-to-audio or audio-to-text processes.

- Speech-to-text is used to convert speech from an audio source to a text format.
- Speech-to-speech is used to translate speech in one language to speech in another language.

The right side of the diagram illustrates text-to-audio processes.

- Text-to-speech is used to generate spoken audio from a text source.

Common use cases

The following table recommends services for some common use cases.

Use case	Service to use
Provide closed captions for recorded or live videos	Speech-to-text
Create a transcript of a phone call or meeting	Speech-to-text

Use case	Service to use
Implement automated note dictation	Speech-to-text
Determine intended user input for further processing	Speech-to-text
Generate spoken responses to user input	Text-to-speech
Create voice menus for telephone systems	Text-to-speech
Read email or text messages aloud in hands-free scenarios	Text-to-speech
Broadcast announcements in public locations, like railway stations or airports	Text-to-speech
Produce real-time closed captioning for a speech or simultaneous two-way translation of a spoken conversation	Speech-to-text

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Kruti Mehta](#) | Azure Senior Fast-Track Engineer
- [Oscar Shimabukuro](#) | Senior Cloud Solution Architect

Other contributors:

- [Mick Alberts](#) | Technical Writer
- [Ashish Chahuan](#) | Senior Cloud Solution Architect
- [Brandon Cowen](#) | Senior Cloud Solution Architect
- [Manjit Singh](#) | Software Engineer
- [Christina Skarpathiotaki](#) | Senior Cloud Solution Architect
- [Nathan Widdup](#) | Azure Senior Fast-Track Engineer

To see nonpublic LinkedIn profiles, sign in to LinkedIn.

Next steps

- What is the Speech service?
- Speech APIs blog post ↗
- Learning path: Provision and manage Azure Cognitive Services
- Learning path: Process and translate speech with Azure Cognitive Speech Services

Related resources

- Types of decision APIs and Applied AI Services
- Types of language API services
- Types of vision API services

Types of vision API services

Article • 06/01/2023

Azure Cognitive Service for Vision is one of the broadest categories in Cognitive Services. You can use the APIs to incorporate vision features like image analysis, face detection, spatial analysis, and optical character recognition (OCR) in your applications, even if you have limited knowledge of machine learning.

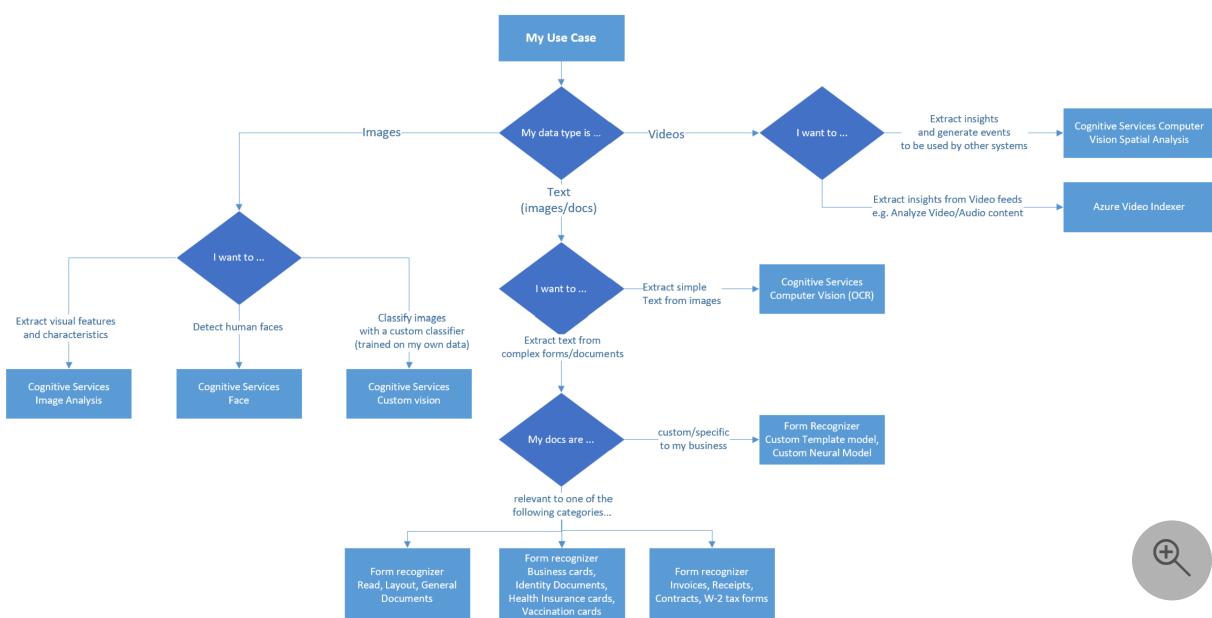
Services

Here are some broad categories of vision APIs:

- [Computer Vision](#) provides advanced algorithms that process images and return information based on the visual features you're interested in. It provides four services: OCR, Face service, Image Analysis, and Spatial Analysis. Form Recognizer is an advanced version of OCR.
- [Custom Vision](#) is an image recognition service that you can use to build, deploy, and improve your own image identifier models.
- [Face service](#) provides AI algorithms that detect, recognize, and analyze human faces in images.

How to choose a service

The following flow chart can help you choose a vision service for your specific use case:



Common use cases

- Computer Vision
 - **Describe an image.** Analyze an image, evaluate the objects that are detected, and generate a human-readable phrase or sentence that describes the image.
 - **Tag visual features.** Apply tags that are based on a set of thousands of recognizable objects.
 - **Categorize an image.** Categorize images based on their content.
 - **Implement OCR.** Detect printed and handwritten text in images.
 - **Detect image types.** For example, identify clip art images or line drawings.
 - **Detect color schemes.** Identify the dominant foreground, background, and dominant and accent colors in an image.
 - **Generate thumbnails.** Create small versions of images.
 - **Moderate content.** Detect images that contain adult content or depict gory scenes.
 - **Detect domain-specific content.** Use two specialized domain models:
 - **Celebrities.** Identify thousands of well-known celebrities from sports, entertainment, and business domains.
 - **Landmarks.** Identify famous landmarks, like the Taj Mahal and the Statue of Liberty.
 - **Detect objects.** Identify common objects and return the coordinates of a bounding box.
 - **Detect brands.** Identify logos from an existing database of thousands of globally recognized product logos.
 - **Detect faces.** Detect and analyze human faces in an image. You can determine the age of the subject and return a bounding box that specifies the locations of faces. The facial analysis capabilities of the Computer Vision service are a subset of the ones provided by the dedicated Face service.
- Custom Vision
 - **Classify images.** Predict a category, or *class*, based on a set of inputs, which are called *features*. Calculate a probability score for each possible class and return a label that indicates the class that the object most likely belongs to. To use this model, you need data that consists of features and their labels.
 - **Detect objects.** Get the coordinates of an object in an image. To use this model, you need data that consists of features and their labels.
- Face services
 - **Detect faces.** Identify the regions of an image that contain a human face, typically by returning bounding-box coordinates that form a rectangle around the face.

- **Analyze faces.** Return information, such as facial landmarks (nose, eyes, eyebrows, lips, and more). You can use these facial landmarks as features to train a machine learning model that can infer information about people, like their perceived age or emotional state.
- **Recognize faces.** Train a machine learning model to identify known individuals from their facial features.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Ashish Chahuan](#) | Senior Cloud Solution Architect
- [Kruti Mehta](#) | Azure Senior Fast-Track Engineer

Other contributors:

- [Mick Alberts](#) | Technical Writer
- [Brandon Cowen](#) | Senior Cloud Solution Architect
- [Oscar Shimabukuro](#) | Senior Cloud Solution Architect
- [Manjit Singh](#) | Software Engineer
- [Christina Skarpathiotaki](#) | Senior Cloud Solution Architect
- [Nathan Widdup](#) | Azure Senior Fast-Track Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [What is Computer Vision?](#)
- [Vision APIs blog post](#)
- [Learning path: Create a Language Understanding solution with Azure Cognitive Services](#)
- [Learning path: Provision and manage Azure Cognitive Services](#)
- [Learning path: Explore computer vision](#)
- [Learning path: Create computer vision solutions with Azure Cognitive Services](#)
- [Learning path: Create an image recognition solution with Azure IoT Edge and Azure Cognitive Services](#)

Related resources

- Types of decision APIs and Applied AI Services
- Types of language API services
- Types of speech API services

Types of decision APIs and Applied AI Services

Article • 06/01/2023

Azure Cognitive Services decision APIs are cloud-based APIs that provide natural language processing (NLP) features to produce recommendations for informed and efficient decision-making. They can help you make smart decisions faster.

Azure Applied AI Services combines Cognitive Services, specialized AI, and built-in business logic to provide ready-to-use AI solutions for frequently encountered business scenarios. Azure Cognitive Search is a cloud search service that has built-in AI capabilities.

Services

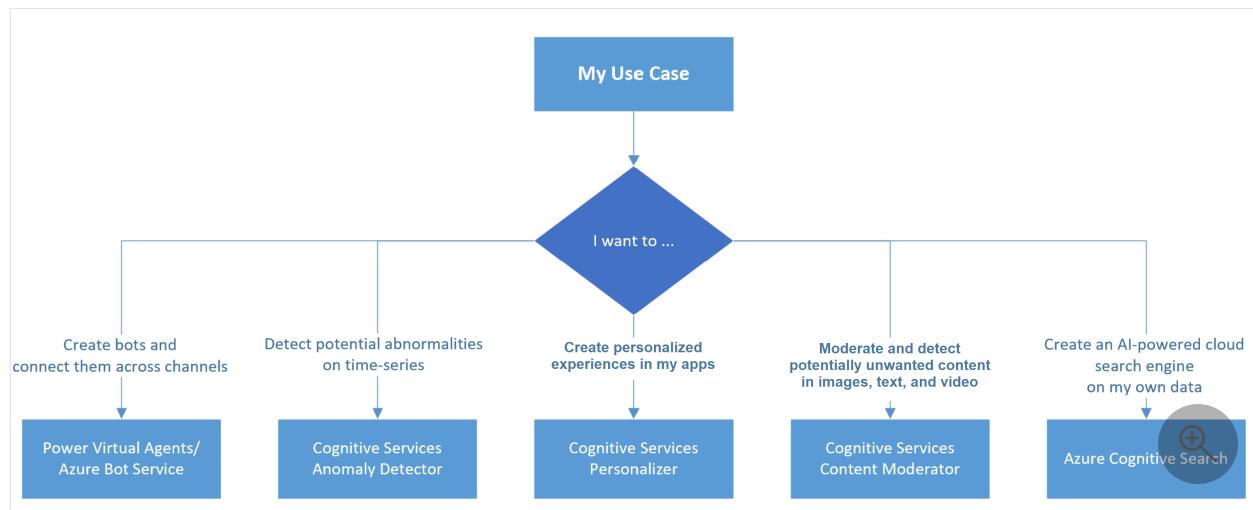
Here are a few of the decision and applied AI services:

- [Azure Bot Service](#) provides an integrated development environment for creating conversational AI bots without writing code. It's integrated with [Power Virtual Agents](#), which is available as both a standalone web app and a discrete app in Microsoft Teams.
- [Anomaly Detector](#) ingests time-series data of all types and selects the best anomaly detection algorithm. The Anomaly Detector API enables you to monitor and detect abnormalities in your time-series data even if you have limited knowledge of machine learning. It uses univariate and multivariate APIs to monitor data over time. You can use it for either batch validation or real-time inference.
- [Personalizer](#) is a cloud-based service that helps your applications choose content items to show your users. Personalizer uses reinforcement learning to select the best item, or *action*, based on collective behavior and reward scores across all users. Actions are content items, like news articles, movies, or products.
- [Content Moderator](#) is a service that checks text, image, and video content for material that's potentially offensive, risky, or otherwise undesirable.
 - **Text moderation** scans text for offensive content, sexually explicit or suggestive content, profanity, and personal data. You can use prebuilt or custom models.
 - **Image moderation** scans images for adult or racy content, detects text in images by using optical character recognition, and detects faces. You can use prebuilt or custom models.
 - **Video moderation** scans videos for adult or racy content and returns time markers for the content. This API currently supports only prebuilt models.

- [Applied AI Services](#) enables you to apply AI to key business data scenarios. These services are built on the AI APIs of Cognitive Services. [Azure Cognitive Search](#) is a key part of Applied AI Services.

How to choose a service

This flow chart can help you choose the decision API or Applied AI Services option that suits your needs:



Common use cases

- **Bot Service**
 - Provide help for sales and support.
 - Provide information about store business hours and more.
 - Provide information about employee health and vacation benefits.
 - Answer common employee questions.
- **Anomaly Detector**
 - Detect anomalies in your streaming data by using previously seen data points to determine whether the latest one is anomalous.
 - Detect anomalies in an entire data series at a specific time. This operation generates a model by using all your time-series data. The same model analyzes each data point.
 - Detect any trend change points that exist in your data at a specific time. This operation generates a model by using all your time-series data. The same model analyzes each data point.
- **Personalizer**
 - Get recommendations for e-commerce. Determine the best products to present to customers to maximize purchases.

- Get content recommendations. Recommend articles that optimize click-through rates.
 - Improve content design. Determine placements for advertisements to optimize user engagement on a website.
 - Improve communications. Determine when and how to send notifications to maximize the likelihood of getting a response.
- **Content Moderator**
 - Scan text, images, or video for potentially risky, offensive, or undesirable content.
 - **Applied AI Services**
 - Implement ready-to-deploy AI solutions. Common use cases include [content filtering](#) and [embeddings](#).

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Kruti Mehta](#) | Azure Senior Fast-Track Engineer
- [Christina Skarpathiotaki](#) | Senior Cloud Solution Architect

Other contributors:

- [Mick Alberts](#) | Technical Writer
- [Ashish Chahuan](#) | Senior Cloud Solution Architect
- [Brandon Cowen](#) | Senior Cloud Solution Architect
- [Oscar Shimabukuro](#) | Senior Cloud Solution Architect
- [Manjit Singh](#) | Software Engineer
- [Nathan Widdup](#) | Azure Senior Fast-Track Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Power Virtual Agents overview](#)
- [Anomaly Detector](#)
- [Content Moderator](#)
- [Personalizer](#)

- [Azure OpenAI](#)
- [Decision APIs blog post ↗](#)
- [Learning path: Provision and manage Azure Cognitive Services](#)
- [Learning path: Identify principles and practices for responsible AI](#)
- [Learning path: Introduction to responsible bots](#)
- [Learning path: Get started with AI](#)

Related resources

- [Types of language API services](#)
- [Types of speech API services](#)
- [Types of vision API services](#)

Natural language processing technology

Azure AI services Azure Databricks Azure HDInsight Azure Synapse Analytics

Natural language processing (NLP) has many uses: sentiment analysis, topic detection, language detection, key phrase extraction, and document categorization.

Specifically, you can use NLP to:

- Classify documents. For instance, you can label documents as sensitive or spam.
- Do subsequent processing or searches. You can use NLP output for these purposes.
- Summarize text by identifying the entities that are present in the document.
- Tag documents with keywords. For the keywords, NLP can use identified entities.
- Do content-based search and retrieval. Tagging makes this functionality possible.
- Summarize a document's important topics. NLP can combine identified entities into topics.
- Categorize documents for navigation. For this purpose, NLP uses detected topics.
- Enumerate related documents based on a selected topic. For this purpose, NLP uses detected topics.
- Score text for sentiment. By using this functionality, you can assess the positive or negative tone of a document.

Apache®, Apache Spark®, and the flame logo are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.

Potential use cases

Business scenarios that can benefit from custom NLP include:

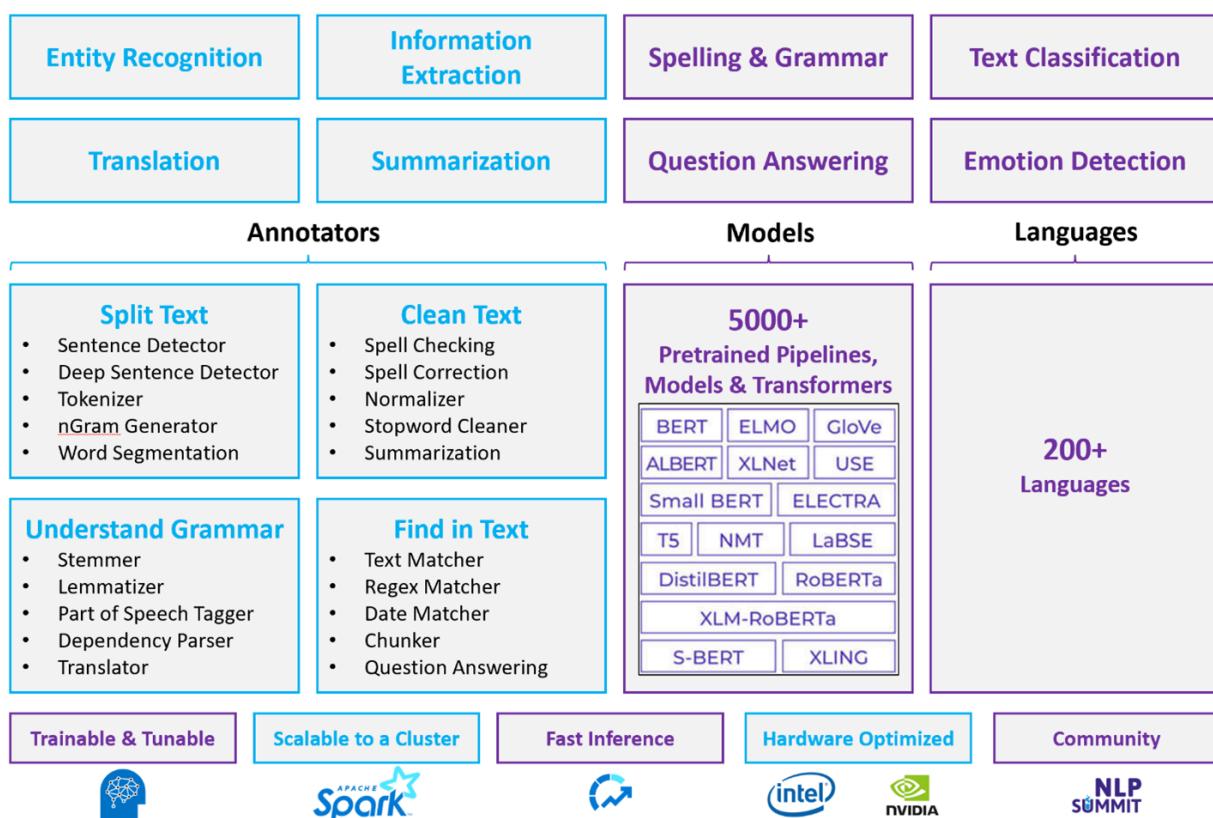
- Document intelligence for handwritten or machine-created documents in finance, healthcare, retail, government, and other sectors.
- Industry-agnostic NLP tasks for text processing, such as name entity recognition (NER), classification, summarization, and relation extraction. These tasks automate the process of retrieving, identifying, and analyzing document information like text and unstructured data. Examples of these tasks include risk stratification models, ontology classification, and retail summarizations.

- Information retrieval and knowledge graph creation for semantic search. This functionality makes it possible to create medical knowledge graphs that support drug discovery and clinical trials.
- Text translation for conversational AI systems in customer-facing applications across retail, finance, travel, and other industries.

Apache Spark as a customized NLP framework

Apache Spark is a parallel processing framework that supports in-memory processing to boost the performance of big-data analytic applications. [Azure Synapse Analytics](#), [Azure HDInsight](#), and [Azure Databricks](#) offer access to Spark and take advantage of its processing power.

For customized NLP workloads, Spark NLP serves as an efficient framework for processing a large amount of text. This open-source NLP library provides Python, Java, and Scala libraries that offer the full functionality of traditional NLP libraries such as spaCy, NLTK, Stanford CoreNLP, and Open NLP. Spark NLP also offers functionality such as spell checking, sentiment analysis, and document classification. Spark NLP improves on previous efforts by providing state-of-the-art accuracy, speed, and scalability.



Recent public benchmarks show Spark NLP as [38 and 80 times faster than spaCy](#), with comparable accuracy for training custom models. Spark NLP is the only open-source library that can use a distributed Spark cluster. Spark NLP is a native extension of Spark

ML that operates directly on data frames. As a result, speedups on a cluster result in another order of magnitude of performance gain. Because every Spark NLP pipeline is a Spark ML pipeline, Spark NLP is well-suited for building unified NLP and machine learning pipelines such as document classification, risk prediction, and recommender pipelines.

Besides excellent performance, Spark NLP also delivers state-of-the-art accuracy for a growing number of NLP tasks. The Spark NLP team regularly reads the latest relevant academic papers and implements state-of-the-art models. In the past two to three years, the best performing models have used deep learning. The library comes with prebuilt deep learning models for named entity recognition, document classification, sentiment and emotion detection, and sentence detection. The library also includes dozens of pre-trained language models that include support for word, chunk, sentence, and document embeddings.

The library has optimized builds for CPUs, GPUS, and the latest Intel Xeon chips. You can scale training and inference processes to take advantage of Spark clusters. These processes can run in production in all popular analytics platforms.

Challenges

- Processing a collection of free-form text documents requires a significant amount of computational resources. The processing is also time intensive. Such processes often involve GPU compute deployment.
- Without a standardized document format, it can be difficult to achieve consistently accurate results when you use 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 date when invoices are from various vendors.

Key selection criteria

In Azure, Spark services like Azure Databricks, Azure Synapse Analytics, and Azure HDInsight provide NLP functionality when you use them with Spark NLP. Azure Cognitive Services is another option for NLP functionality. To decide which service to use, consider these questions:

- Do you want to use prebuilt or pretrained models? If yes, consider using the APIs that Azure Cognitive Services offers. Or download your model of choice through Spark NLP.

- Do you need to train custom models against a large corpus of text data? If yes, consider using Azure Databricks, Azure Synapse Analytics, or Azure HDInsight with 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 Databricks, Azure Synapse Analytics, or Azure HDInsight with Spark NLP. Or use an open-source software library in your processing tool of choice.
- 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 that Cognitive Services offers. Or download your model of choice through Spark NLP.

Capability matrix

The following tables summarize the key differences in the capabilities of NLP services.

General capabilities

[Expand table](#)

Capability	Spark service (Azure Databricks, Azure Synapse Analytics, Azure HDInsight) with Spark NLP	Azure Cognitive Services
Provides pretrained models as a service	Yes	Yes
REST API	Yes	Yes
Programmability	Python, Scala	For supported languages, see Additional Resources
Supports processing of big data sets and large documents	Yes	No

Low-level NLP capabilities

 Expand table

Capability of annotators	Spark service (Azure Databricks, Azure Synapse Analytics, Azure HDInsight) with Spark NLP	Azure Cognitive Services
Sentence detector	Yes	No
Deep sentence detector	Yes	Yes
Tokenizer	Yes	Yes
N-gram generator	Yes	No
Word segmentation	Yes	Yes
Stemmer	Yes	No
Lemmatizer	Yes	No
Part-of-speech tagging	Yes	No
Dependency parser	Yes	No
Translation	Yes	No
Stopword cleaner	Yes	No
Spell correction	Yes	No

Capability of annotators	Spark service (Azure Databricks, Azure Synapse Analytics, Azure HDInsight) with Spark NLP	Azure Cognitive Services
Normalizer	Yes	Yes
Text matcher	Yes	No
TF/IDF	Yes	No
Regular expression matcher	Yes	Embedded in Language Understanding Service (LUIS). Not supported in Conversational Language Understanding (CLU), which is replacing LUIS.
Date matcher	Yes	Possible in LUIS and CLU through DateTime recognizers
Chunker	Yes	No

High-level NLP capabilities

[Expand table](#)

Capability	Spark service (Azure Databricks, Azure Synapse Analytics, Azure HDInsight) with Spark NLP	Azure Cognitive Services
Spell checking	Yes	No
Summarization	Yes	Yes
Question answering	Yes	Yes
Sentiment	Yes	Yes

Capability	Spark service (Azure Databricks, Azure Synapse Analytics, Azure HDInsight) with Spark NLP	Azure Cognitive Services
detection		
Emotion detection	Yes	Supports opinion mining
Token classification	Yes	Yes, through custom models
Text classification	Yes	Yes, through custom models
Text representation	Yes	No
NER	Yes	Yes—text analytics provides a set of NER, and custom models are in entity recognition
Entity recognition	Yes	Yes, through custom models
Language detection	Yes	Yes
Supports languages besides English	Yes, supports over 200 languages	Yes, supports over 97 languages

Set up Spark NLP in Azure

To install Spark NLP, use the following code, but replace <version> with the latest version number. For more information, see the [Spark NLP documentation](#).

Bash

```
# Install Spark NLP from PyPI.
pip install spark-nlp==<version>

# Install Spark NLP from Anacodna or Conda.
conda install -c johnsnowlabs spark-nlp

# Load Spark NLP with Spark Shell.
spark-shell --packages com.johnsnowlabs.nlp:spark-nlp_<version>
```

```

# Load Spark NLP with PySpark.
pyspark --packages com.johnsnowlabs.nlp:spark-nlp_<version>

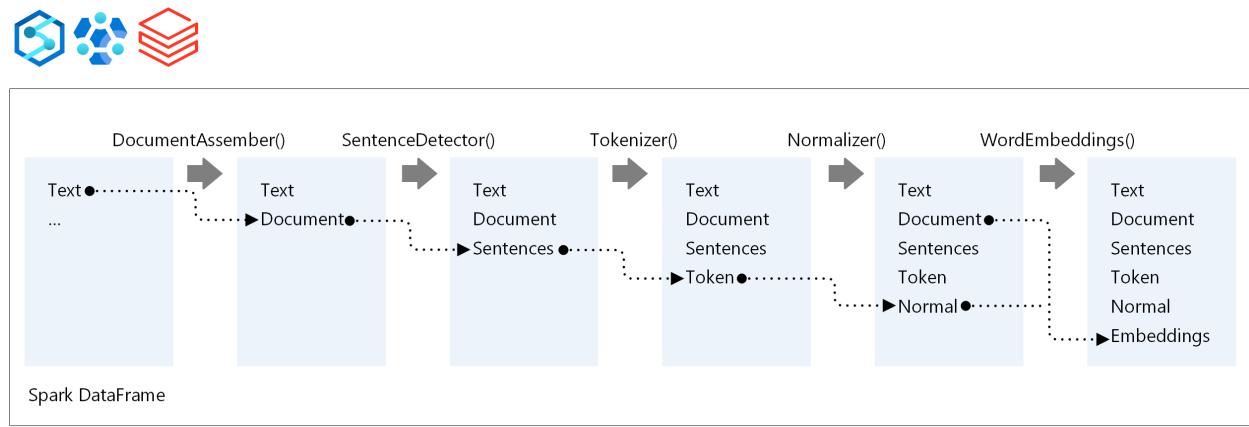
# Load Spark NLP with Spark Submit.
spark-submit --packages com.johnsnowlabs.nlp:spark-nlp_<version>

# Load Spark NLP as an external JAR after compiling and building Spark NLP by using sbt assembly.
spark-shell --jars spark-nlp-assembly-3 <version>.jar

```

Develop NLP pipelines

For the execution order of an NLP pipeline, Spark NLP follows the same development concept as traditional Spark ML machine learning models. But Spark NLP applies NLP techniques.



The core components of a Spark NLP pipeline are:

- **DocumentAssembler**: A transformer that prepares data by changing it into a format that Spark NLP can process. This stage is the entry point for every Spark NLP pipeline. DocumentAssembler can read either a `String` column or an `Array[String]`. You can use `setCleanupMode` to preprocess the text. By default, this mode is turned off.
- **SentenceDetector**: An annotator that detects sentence boundaries by using the approach that it's given. This annotator can return each extracted sentence in an `Array`. It can also return each sentence in a different row, if you set `explodeSentences` to true.
- **Tokenizer**: An annotator that separates raw text into tokens, or units like words, numbers, and symbols, and returns the tokens in a `TokenizedSentence` structure. This class is non-fitted. If you fit a tokenizer, the internal `RuleFactory` uses the

input configuration to set up tokenizing rules. Tokenizer uses open standards to identify tokens. If the default settings don't meet your needs, you can add rules to customize Tokenizer.

- **Normalizer:** An annotator that cleans tokens. Normalizer requires stems. Normalizer uses regular expressions and a dictionary to transform text and remove dirty characters.
- **WordEmbeddings:** Look-up annotators that map tokens to vectors. You can use `setStoragePath` to specify a custom token look-up dictionary for embeddings. Each line of your dictionary needs to contain a token and its vector representation, separated by spaces. If a token isn't found in the dictionary, the result is a zero vector of the same dimension.

Spark NLP uses Spark MLlib pipelines, which MLflow natively supports. [MLflow](#) is an open-source platform for the machine learning lifecycle. Its components include:

- MLflow Tracking: Records experiments and provides a way to query results.
- MLflow Projects: Makes it possible to run data science code on any platform.
- MLflow Models: Deploys models to diverse environments.
- Model Registry: Manages models that you store in a central repository.

MLflow is integrated in Azure Databricks. You can install MLflow in any other Spark-based environment to track and manage your experiments. You can also use MLflow Model Registry to make models available for production purposes.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Moritz Steller](#) | Senior Cloud Solution Architect
- [Zoiner Tejada](#) | CEO and Architect

Next steps

- Spark NLP documentation:
 - [Spark NLP](#)
 - [Spark NLP general documentation](#)
 - [Spark NLP GitHub](#)

- [Spark NLP demo ↗](#)
- [Spark NLP pipelines ↗](#)
- [Spark NLP annotators ↗](#)
- [Spark NLP transformers ↗](#)
- Azure components:
 - [Azure Synapse Analytics](#)
 - [Azure HDInsight](#)
 - [Azure Databricks](#)
 - [Cognitive Services](#)
- Learn resources:
 - [Microsoft Azure AI Fundamentals: Explore natural language processing](#)
 - [Create a Language Understanding solution](#)

Related resources

- [Large-scale custom natural language processing in Azure](#)
- [Choose a Microsoft cognitive services technology](#)
- [Compare the machine learning products and technologies from Microsoft](#)
- [MLflow and Azure Machine Learning](#)
- [AI enrichment with image and natural language processing in Azure Cognitive Search](#)
- [Analyze news feeds with near real-time analytics using image and natural language processing](#)
- [Suggest content tags with NLP using deep learning](#)

Compare Microsoft machine learning products and technologies

Article • 08/31/2023

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

Cloud-based machine learning products

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

Expand table

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

On-premises machine learning products

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

[Expand table](#)

On-premises options	What it is	What you can do with it
SQL Server Machine Learning Services	In-database machine learning for SQL	Train and deploy models inside SQL Server
Machine Learning Services on SQL Server Big Data Clusters	Machine learning in Big Data Clusters	Train and deploy models on SQL Server Big Data Clusters

Development platforms and tools

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

[Expand table](#)

Platforms/tools	What it is	What you can do with it
Azure Data Science Virtual Machine	Virtual machine with pre-installed data science tools	Develop machine learning solutions in a pre-configured environment
ML.NET	Open-source, cross-platform machine learning SDK	Develop machine learning solutions for .NET applications
Windows ML	Windows 10 machine learning platform	Evaluate trained models on a Windows 10 device
SynapseML	Open-source, distributed, machine learning and microservices framework for Apache Spark	Create and deploy scalable machine learning applications for Scala and Python.
Machine Learning extension for Azure Data Studio	Open-source and cross-platform machine learning extension for Azure Data Studio	Manage packages, import machine learning models, make predictions, and create notebooks to run experiments for your SQL databases

Azure Machine Learning

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

deployments in Visual Studio Code. Azure Machine Learning includes features that automate model generation and tuning with ease, efficiency, and accuracy.

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

[Try Azure Machine Learning for free](#).

[+] [Expand table](#)

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

Azure Cognitive Services

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

- **Vision:** Object detection, face recognition, OCR, etc. See [Computer Vision](#), [Face](#), [Form Recognizer](#).

- **Speech:** Speech-to-text, text-to-speech, speaker recognition, etc. See [Speech Service](#).
- **Language:** Translation, Sentiment analysis, key phrase extraction, language understanding, etc. See [Translator](#), [Text Analytics](#), [Language Understanding](#), [QnA Maker](#)
- **Decision:** Anomaly detection, content moderation, reinforcement learning. See [Anomaly Detector](#), [Content Moderator](#), [Personalizer](#).

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

[+] [Expand table](#)

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

SQL machine learning

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

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

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

[+] Expand table

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

Azure Data Science Virtual Machine

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

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

[+] Expand table

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

Azure Databricks

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

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

[+] [Expand table](#)

Item	Description
Type	Apache Spark-based analytics platform
Supported languages	Python, R, Scala, SQL
Machine learning phases	Data preparation Data preprocessing Model training Model tuning Model inference Management Deployment

ML.NET

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

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

[+] [Expand table](#)

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

Windows ML

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

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

[\[\] Expand table](#)

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

SynapseML

[SynapseML](#) (formerly known as MMLSpark) is an open-source library that simplifies the creation of massively scalable machine learning (ML) pipelines. SynapseML provides APIs for a variety of different machine learning tasks such as text analytics, vision, anomaly detection, and many others. SynapseML is built on the [Apache Spark](#) distributed computing framework and shares the same API as the SparkML/MLLib library, allowing you to seamlessly embed SynapseML models into existing Apache Spark workflows.

SynapseML adds many deep learning and data science tools to the Spark ecosystem, including seamless integration of [Spark Machine Learning](#) pipelines with [Light Gradient Boosting Machine \(LightGBM\)](#), [LIME \(Model Interpretability\)](#), and [OpenCV](#). You can use these tools to create powerful predictive models on any Spark cluster, such as [Azure Databricks](#) or [Cosmic Spark](#).

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

[] [Expand table](#)

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

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Zoiner Tejada](#) | CEO and Architect

Next steps

- To learn about all the Artificial Intelligence (AI) development products available from Microsoft, see [Microsoft AI platform](#).

- For training in developing AI and Machine Learning solutions with Microsoft, see [Microsoft Learn training](#).

Related resources

- [Choose a Microsoft cognitive services technology](#)
- [Artificial intelligence \(AI\) architecture design](#)
- [How Azure Machine Learning works: resources and assets](#)

MLflow and Azure Machine Learning

Article • 01/10/2024

APPLIES TO:  Azure CLI ml extension v2 (current)  Python SDK azure-ai-ml v2 (current) ↗

[MLflow](#) ↗ is an open-source framework designed to manage the complete machine learning lifecycle. Its ability to train and serve models on different platforms allows you to use a consistent set of tools regardless of where your experiments are running: whether locally on your computer, on a remote compute target, on a virtual machine, or on an Azure Machine Learning compute instance.

Azure Machine Learning **workspaces** are **MLflow-compatible**, which means that you can use Azure Machine Learning workspaces in the same way that you'd use an MLflow server. This compatibility has the following advantages:

- Azure Machine Learning doesn't host MLflow server instances under the hood; rather, the workspace can speak the MLflow API language.
- You can use Azure Machine Learning workspaces as your tracking server for any MLflow code, whether it runs on Azure Machine Learning or not. You only need to configure MLflow to point to the workspace where the tracking should happen.
- You can run any training routine that uses MLflow in Azure Machine Learning without any change.

💡 Tip

Unlike the Azure Machine Learning SDK v1, there's no logging functionality in the SDK v2. We recommend that you use MLflow for logging, so that your training routines are cloud-agnostic and portable—removing any dependency your code has on Azure Machine Learning.

Tracking with MLflow

Azure Machine Learning uses MLflow tracking to log metrics and store artifacts for your experiments. When you're connected to Azure Machine Learning, all tracking performed using MLflow is materialized in the workspace you're working on. To learn more about how to set up your experiments to use MLflow for tracking experiments and training routines, see [Log metrics, parameters, and files with MLflow](#). You can also use MLflow to [query & compare experiments and runs](#).

MLflow in Azure Machine Learning provides a way to **centralize tracking**. You can connect MLflow to Azure Machine Learning workspaces even when you're working locally or in a different cloud. The workspace provides a centralized, secure, and scalable location to store training metrics and models.

Using MLflow in Azure Machine Learning includes the capabilities to:

- [Track machine learning experiments and models running locally or in the cloud](#).
- [Track Azure Databricks machine learning experiments](#).
- [Track Azure Synapse Analytics machine learning experiments](#).

Example notebooks

- [Training and tracking an XGBoost classifier with MLflow](#) : Demonstrates how to track experiments by using MLflow, log models, and combine multiple flavors into pipelines.
- [Training and tracking an XGBoost classifier with MLflow using service principal authentication](#) : Demonstrates how to track experiments by using MLflow from a compute that's running outside Azure Machine Learning. The example shows how to authenticate against Azure Machine Learning services by using a service principal.
- [Hyper-parameter optimization using HyperOpt and nested runs in MLflow](#) : Demonstrates how to use child runs in MLflow to do hyper-parameter optimization for models by using the popular library `Hyperopt`. The example shows how to transfer metrics, parameters, and artifacts from child runs to parent runs.
- [Logging models with MLflow](#) : Demonstrates how to use the concept of models, instead of artifacts, with MLflow. The example also shows how to construct custom models.
- [Manage runs and experiments with MLflow](#) : Demonstrates how to query experiments, runs, metrics, parameters, and artifacts from Azure Machine Learning by using MLflow.

Tracking with MLflow in R

MLflow support in R has the following limitations:

- MLflow tracking is limited to tracking experiment metrics, parameters, and models on Azure Machine Learning jobs.
- Interactive training on RStudio, Posit (formerly RStudio Workbench), or Jupyter notebooks with R kernels is *not supported*.

- Model management and registration are *not supported* using the MLflow R SDK. Instead, use the Azure Machine Learning CLI or [Azure Machine Learning studio](#) for model registration and management.

To learn about using the MLflow tracking client with Azure Machine Learning, view the examples in [Train R models using the Azure Machine Learning CLI \(v2\)](#).

Tracking with MLflow in Java

MLflow support in Java has the following limitations:

- MLflow tracking is limited to tracking experiment metrics and parameters on Azure Machine Learning jobs.
- Artifacts and models can't be tracked using the MLflow Java SDK. Instead, use the `outputs` folder in jobs along with the `mlflow.save_model` method to save models (or artifacts) that you want to capture.

To learn about using the MLflow tracking client with Azure Machine Learning, view the [Java example that uses the MLflow tracking client with Azure Machine Learning](#).

Model registries with MLflow

Azure Machine Learning supports MLflow for model management. This support represents a convenient way to support the entire model lifecycle for users that are familiar with the MLflow client.

To learn more about how to manage models by using the MLflow API in Azure Machine Learning, view [Manage model registries in Azure Machine Learning with MLflow](#).

Example notebook

- [Manage model registries with MLflow](#): Demonstrates how to manage models in registries by using MLflow.

Model deployment with MLflow

You can [deploy MLflow models to Azure Machine Learning](#) and take advantage of the improved experience when you use MLflow models. Azure Machine Learning supports deployment of MLflow models to both real-time and batch endpoints without having to specify an environment or a scoring script. Deployment is supported using the MLflow

SDK, Azure Machine Learning CLI, Azure Machine Learning SDK for Python, or the [Azure Machine Learning studio](#).

To learn more about deploying MLflow models to Azure Machine Learning for both real-time and batch inferencing, see [Guidelines for deploying MLflow models](#).

Example notebooks

- [Deploy MLflow to online endpoints](#): Demonstrates how to deploy models in MLflow format to online endpoints using the MLflow SDK.
- [Deploy MLflow to online endpoints with safe rollout](#): Demonstrates how to deploy models in MLflow format to online endpoints, using the MLflow SDK with progressive rollout of models. The example also shows deployment of multiple versions of a model to the same endpoint.
- [Deploy MLflow to web services \(V1\)](#): Demonstrates how to deploy models in MLflow format to web services (ACI/AKS v1) using the MLflow SDK.
- [Deploy models trained in Azure Databricks to Azure Machine Learning with MLflow](#): Demonstrates how to train models in Azure Databricks and deploy them in Azure Machine Learning. The example also covers how to handle cases where you also want to track the experiments with the MLflow instance in Azure Databricks.

Training with MLflow projects (preview)

Important

Items marked (preview) in this article are currently in public preview. The preview version is provided without a service level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

You can submit training jobs to Azure Machine Learning by using [MLflow projects](#) (preview). You can submit jobs locally with Azure Machine Learning tracking or migrate your jobs to the cloud via [Azure Machine Learning compute](#).

To learn how to submit training jobs with MLflow Projects that use Azure Machine Learning workspaces for tracking, see [Train machine learning models with MLflow projects and Azure Machine Learning](#).

Example notebooks

- Track an MLflow project in Azure Machine Learning workspaces [↗](#).
- Train and run an MLflow project on Azure Machine Learning jobs [↗](#).

MLflow SDK, Azure Machine Learning v2, and Azure Machine Learning studio capabilities

The following table shows the operations that are possible, using each of the client tools available in the machine learning lifecycle.

[\[+\] Expand table](#)

Feature	MLflow SDK	Azure Machine Learning CLI/SDK	Azure Machine Learning studio
Track and log metrics, parameters, and models	✓		
Retrieve metrics, parameters, and models	✓	1	✓
Submit training jobs	✓ ²	✓	✓
Submit training jobs with Azure Machine Learning data assets		✓	✓
Submit training jobs with machine learning pipelines		✓	✓
Manage experiments and runs	✓	✓	✓
Manage MLflow models	✓ ³	✓	✓
Manage non-MLflow models		✓	✓
Deploy MLflow models to Azure Machine Learning (Online & Batch)	✓ ⁴	✓	✓
Deploy non-MLflow models to Azure Machine Learning		✓	✓

ⓘ Note

- ¹ Only artifacts and models can be downloaded.
- ² Possible by using MLflow projects (preview).

- ³ Some operations may not be supported. View [Manage model registries in Azure Machine Learning with MLflow](#) for details.
- ⁴ Deployment of MLflow models for batch inference by using the MLflow SDK is not possible at the moment. As an alternative, see [Deploy and run MLflow models in Spark jobs](#).

Related content

- [From artifacts to models in MLflow](#).
- [Configure MLflow for Azure Machine Learning](#).
- [Migrate logging from SDK v1 to MLflow](#)
- [Track ML experiments and models with MLflow](#).
- [Log MLflow models](#).
- [Guidelines for deploying MLflow models](#).

Load-balancing options

Azure Load Balancer Azure Front Door Azure Application Gateway Azure Traffic Manager

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

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

This article describes how you can use the [Load balancing](#) page in the Azure portal to determine an appropriate load-balancing solution for your business needs.

Service categorizations

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

Global vs. regional

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

HTTP(S) vs. non-HTTP(S)

- **HTTP(S):** These load-balancing services are [Layer 7](#) load balancers that only accept HTTP(S) traffic. They're intended for web applications or other HTTP(S)

endpoints. They include features such as SSL offload, web application firewall, path-based load balancing, and session affinity.

- **Non-HTTP(S):** These load-balancing services can handle non-HTTP(S) traffic, and we recommend them for nonweb workloads.

The following table summarizes the Azure load-balancing services.

[\[+\] Expand table](#)

Service	Global/Regional	Recommended traffic
Azure Front Door	Global	HTTP(S)
Azure Traffic Manager	Global	Non-HTTP(S)
Azure Application Gateway	Regional	HTTP(S)
Azure Load Balancer	Regional or Global	Non-HTTP(S)

Azure load-balancing services

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

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

Note

At this time, Azure Front Door doesn't support Web Sockets.

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

- [Application Gateway](#) provides application delivery controller as a service, offering various Layer 7 load-balancing capabilities. Use it to optimize web farm productivity by offloading CPU-intensive SSL termination to the gateway.
- [Load Balancer](#) is a high-performance, ultra-low-latency Layer 4 load-balancing service (inbound and outbound) for all UDP and TCP protocols. It's built to handle millions of requests per second while ensuring your solution is highly available. Load Balancer is zone redundant, ensuring high availability across availability zones. It supports both a regional deployment topology and a [cross-region topology](#).

Choose a load-balancing solution by using the Azure portal

You can use the **Load balancing** page in the Azure portal to help guide you to the appropriate load-balancing solution for your business need. Load Balancer includes the decision-making queries described in the workflow in the following section.

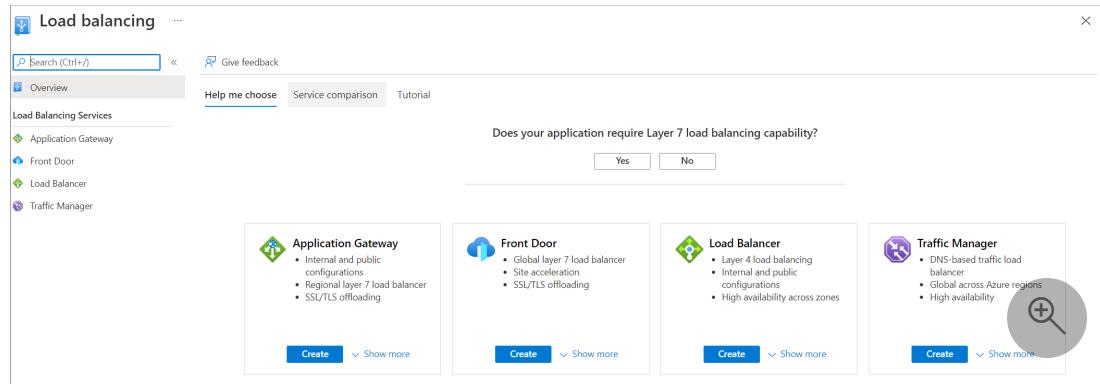
1. Sign in to the [Azure portal](#).
2. In the **Search resources, services, and docs** box at the top of the portal, enter **load balancing**. When **Load balancing** appears in the search results, select it.

Note

To learn about some of the factors considered in selecting a load-balancing solution, see [Decision tree for load balancing in Azure](#) in this article.

3. On the **Load balancing - Help me choose (Preview)** page, use one of the following options:

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



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

Reference architecture examples

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

[Expand table](#)

Services	Article	Description
Load Balancer	Load balance virtual machines (VMs) across availability zones	Load balance VMs across availability zones to help protect your apps and data from an unlikely failure or loss of an entire datacenter. With zone redundancy, one or more availability zones can fail and the data path survives as long as one zone in the region remains healthy.
Azure Front Door	Sharing location in real time by using	Use Azure Front Door to provide higher availability for your applications than deploying to a single region. If a regional outage affects

Services	Article	Description
	low-cost serverless Azure services	the primary region, you can use Azure Front Door to fail over to the secondary region.
Traffic Manager	Multitier web application built for high availability and disaster recovery	Deploy resilient multitier applications built for high availability and disaster recovery. If the primary region becomes unavailable, Traffic Manager fails over to the secondary region.
Azure Front Door + Application Gateway	Multitenant SaaS on Azure	Use a multitenant solution that includes a combination of Azure Front Door and Application Gateway. Azure Front Door helps load balance traffic across regions. Application Gateway routes and load-balances traffic internally in the application to the various services that satisfy client business needs.
Traffic Manager + Load Balancer	Multiregion N-tier application	A multiregion N-tier application that uses Traffic Manager to route incoming requests to a primary region. If that region becomes unavailable, Traffic Manager fails over to the secondary region.
Traffic Manager + Application Gateway	Multiregion load balancing with Traffic Manager and Application Gateway	Learn how to serve web workloads and deploy resilient multitier applications in multiple Azure regions to achieve high availability and a robust disaster recovery infrastructure.

Decision tree for load balancing in Azure

When you select load-balancing options, consider these factors when you select the **Help me choose default** tab on the **Load balancing** page:

- **Traffic type:** Is it a web (HTTP/HTTPS) application? Is it public facing or a private application?
- **Global vs. regional:** Do you need to load balance VMs or containers within a virtual network, or load balance scale unit/deployments across regions, or both?
- **Availability:** What's the [service-level agreement](#)?
- **Cost:** For more information, see [Azure pricing](#). In addition to the cost of the service itself, consider the operations cost for managing a solution built on that

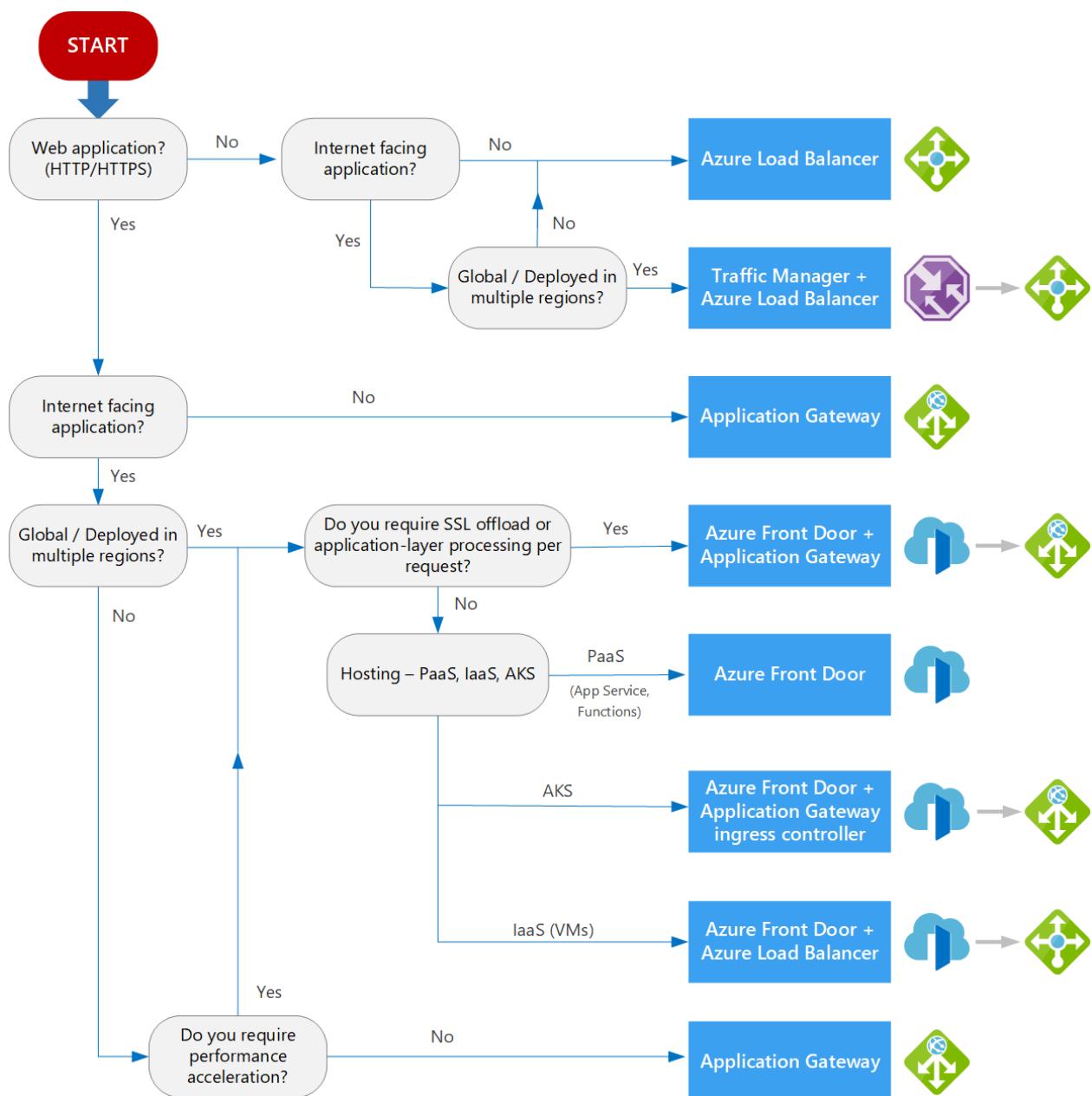
service.

- **Features and limits:** What are the overall limitations of each service? For more information, see [Service limits](#).

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

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

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



Definitions

- **Internet facing:** Applications that are publicly accessible from the internet. As a best practice, application owners apply restrictive access policies or protect the application by setting up offerings like web application firewall and DDoS protection.
- **Global:** End users or clients located beyond a small geographical area. For example, users across multiple continents, across countries/regions within a continent, or even across multiple metropolitan areas within a larger country/region.
- **Platform as a service (PaaS):** Provides a managed hosting environment, where you can deploy your application without needing to manage VMs or networking resources. In this case, PaaS refers to services that provide integrated load balancing within a region. For more information, see [Choose a compute service – Scalability](#).
- **Azure Kubernetes Service (AKS):** Enables you to deploy and manage containerized applications. AKS provides serverless Kubernetes, an integrated continuous integration and continuous delivery experience, and enterprise-grade security and governance. For more information about AKS architectural resources, see [Azure Kubernetes Service architecture design](#).
- **Infrastructure as a service:** A computing option where you provision the VMs that you need, along with associated network and storage components. IaaS applications require internal load balancing within a virtual network by using Load Balancer.
- **Application-layer processing:** Refers to special routing within a virtual network. For example, path-based routing within the virtual network across VMs or virtual machine scale sets. For more information, see [When should we deploy an Application Gateway behind Azure Front Door?](#).
- **Performance acceleration:** Refers to features that accelerate web access. Performance acceleration can be achieved by using content delivery networks (CDNs) or optimized point of presence ingress for accelerated client onboarding into the destination network. Azure Front Door supports both [CDNs](#) and [Anycast traffic acceleration](#). The benefits of both features can be gained with or without Application Gateway in the architecture.

Next steps

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

Choose between virtual network peering and VPN gateways

Microsoft Entra ID

Azure Virtual Network

Azure VPN Gateway

This article compares two ways to connect virtual networks in Azure: virtual network peering and VPN gateways.

A virtual network is a virtual, isolated portion of the Azure public network. By default, traffic cannot be routed between two virtual networks. However, it's possible to connect virtual networks, either within a single region or across two regions, so that traffic can be routed between them.

Virtual network connection types

Virtual network peering. Virtual network peering connects two Azure virtual networks. Once peered, the virtual networks appear as one for connectivity purposes. Traffic between virtual machines in the peered virtual networks is routed through the Microsoft backbone infrastructure, through private IP addresses only. No public internet is involved. You can also peer virtual networks across Azure regions (global peering).

VPN gateways. A VPN gateway is a specific type of virtual network gateway that is used to send traffic between an Azure virtual network and an on-premises location over the public internet. You can also use a VPN gateway to send traffic between Azure virtual networks. Each virtual network can have at most one VPN gateway. You should enable [Azure DDOS Protection](#) on any perimeter virtual network.

Virtual network peering provides a low-latency, high-bandwidth connection. There is no gateway in the path, so there are no extra hops, ensuring low latency connections. It's useful in scenarios such as cross-region data replication and database failover. Because traffic is private and remains on the Microsoft backbone, also consider virtual network peering if you have strict data policies and want to avoid sending any traffic over the internet.

VPN gateways provide a limited bandwidth connection and are useful in scenarios where you need encryption but can tolerate bandwidth restrictions. In these scenarios, customers are also not as latency-sensitive.

Gateway transit

Virtual network peering and VPN Gateways can also coexist via gateway transit

Gateway transit enables you to use a peered virtual network's gateway for connecting to on-premises, instead of creating a new gateway for connectivity. As you increase your workloads in Azure, you need to scale your networks across regions and virtual networks to keep up with the growth. Gateway transit allows you to share an ExpressRoute or VPN gateway with all peered virtual networks and lets you manage the connectivity in one place. Sharing enables cost-savings and reduction in management overhead.

With gateway transit enabled on virtual network peering, you can create a transit virtual network that contains your VPN gateway, Network Virtual Appliance, and other shared services. As your organization grows with new applications or business units and as you spin up new virtual networks, you can connect to your transit virtual network using peering. This prevents adding complexity to your network and reduces management overhead of managing multiple gateways and other appliances.

Configuring connections

Virtual network peering and VPN gateways both support the following connection types:

- Virtual networks in different regions.
- Virtual networks in different Microsoft Entra tenants.
- Virtual networks in different Azure subscriptions.
- Virtual networks that use a mix of Azure deployment models (Resource Manager and classic).

For more information, see the following articles:

- [Create a virtual network peering - Resource Manager, different subscriptions](#)
- [Create a virtual network peering - different deployment models, same subscription](#)
- [Configure a VNet-to-VNet VPN gateway connection by using the Azure portal](#)
- [Connect virtual networks from different deployment models using the portal](#)
- [VPN Gateway FAQ](#)

Comparison of virtual network peering and VPN Gateway

[] [Expand table](#)

Item	Virtual network peering	VPN Gateway
Limits	Up to 500 virtual network peerings per virtual network (see Networking limits).	One VPN gateway per virtual network. The maximum number of tunnels per gateway depends on the gateway SKU .
Pricing model	Ingress/Egress ↗	Hourly + Egress ↗
Encryption	Software-level encryption is recommended.	Custom IPsec/IKE policy can be applied to new or existing connections. See About cryptographic requirements and Azure VPN gateways .
Bandwidth limitations	No bandwidth limitations.	Varies based on SKU. See Gateway SKUs by tunnel, connection, and throughput .
Private?	Yes. Routed through Microsoft backbone and private. No public internet involved.	Public IP involved, but routed through Microsoft backbone if Microsoft global network is enabled.
Transitive relationship	Peering connections are non-transitive. Transitive networking can be achieved using NVAs or gateways in the hub virtual network. See Hub-spoke network topology for an example.	If virtual networks are connected via VPN gateways and BGP is enabled in the virtual network connections, transitivity works.
Initial setup time	Fast	~30 minutes
Typical scenarios	Data replication, database failover, and other scenarios needing frequent backups of large data.	Encryption-specific scenarios that are not latency sensitive and do not need high throughout.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Anavi Nahar | Principal PDM Manager

Next steps

- [Plan virtual networks](#)
- [Choose a solution for connecting an on-premises network to Azure](#)

Asynchronous messaging options

Azure Event Hubs

Azure Event Grid

Azure Service Bus

Azure Stream Analytics

This article describes the different types of messages and the entities that participate in a messaging infrastructure. Based on the requirements of each message type, the article recommends Azure messaging services. The options include Azure Service Bus Messaging, Azure Event Grid, and Azure Event Hubs. For product comparison, see [Compare messaging services](#).

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



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

Commands

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

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

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

Events

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

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

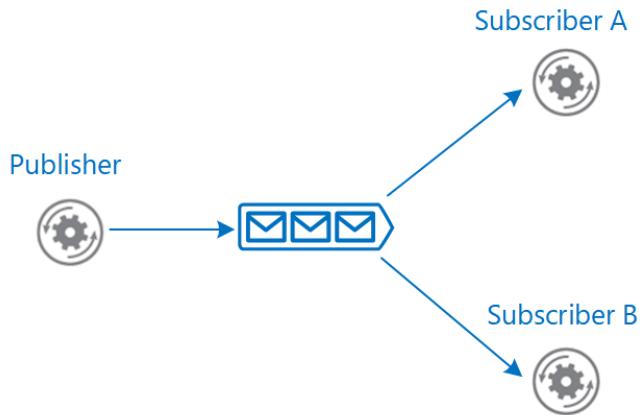
Interested consumer(s) can subscribe, listen for events, and take actions depending on their consumption scenario. Events can have multiple subscribers or no subscribers at all. Two different subscribers can react to an event with different actions and not be aware of one another.

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

There are two categories of events:

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

A common pattern for implementing event messaging is the [Publisher-Subscriber](#) pattern.



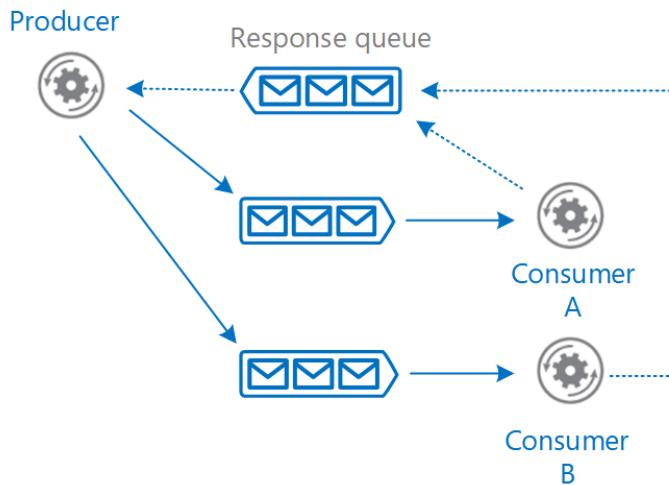
Role and benefits of a message broker

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

Decoupling

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

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



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

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

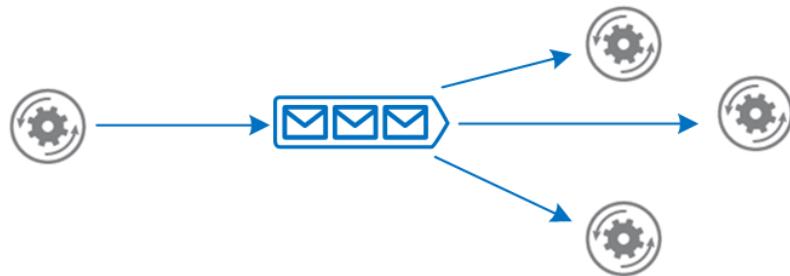
Certain operations can take long to complete. After it issues a command, the producer shouldn't have to wait until the consumer completes it. A message broker helps asynchronous processing of messages.

Load balancing

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

Application instances generating messages

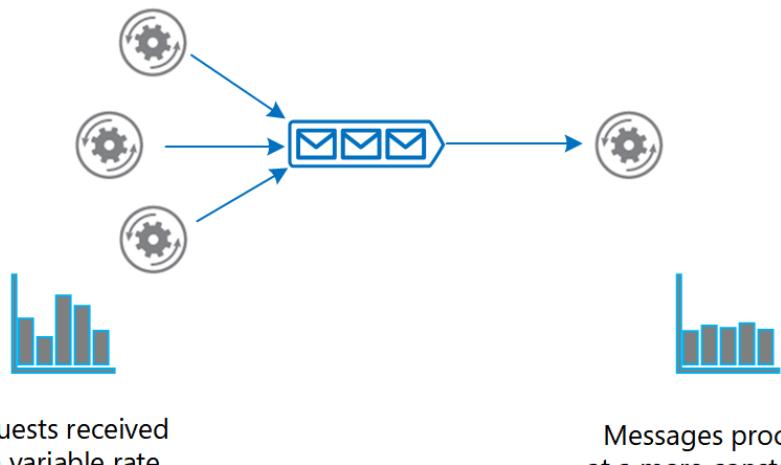
Consumer instance pool processing messages



The [Competing Consumers pattern](#) explains how to process multiple messages concurrently to optimize throughput, improve scalability and availability, and balance the workload.

Load leveling

The volume of messages generated by the producer or a group of producers can be variable. At times there might be a large volume causing spikes in messages. Instead of adding consumers to handle this work, a message broker can act as a buffer, and consumers gradually drain messages at their own pace without stressing the system.



Requests received at a variable rate

Messages processed at a more constant rate

The [Queue-based Load Leveling pattern](#) provides more information.

Reliable messaging

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

Resilient messaging

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

Technology choices for a message broker

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

Azure Service Bus Messaging

[Azure Service Bus Messaging](#) queues are well suited for transferring commands from producers to consumers. Here are some considerations.

Pull model

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

Guaranteed delivery

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

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

A producer might accidentally send the same message twice. For example, a producer instance fails after sending a message. Another producer replaces the original instance and sends the message again. Azure Service Bus queues provide a [built-in de-duping](#)

[capability](#) that detects and removes duplicate messages. There's still a chance that a message is delivered twice. For example, if a consumer fails while processing, the message is returned to the queue and is retrieved by the same or another consumer. The message-processing logic in the consumer should be idempotent so that even if the work is repeated, the state of the system isn't changed.

Message ordering

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

For more information, see [Message sessions](#).

Message persistence

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

Checkpoint long-running transactions

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

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

Dead-letter queue (DLQ)

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

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

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

Examine messages in the DLQ to determine the failure reason.

Hybrid solution

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

The [Messaging Bridge pattern](#) is another way to handle these scenarios.

Topics and subscriptions

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

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

For more information, see [Azure Service Bus topics](#).

Azure Event Grid

We recommend [Azure Event Grid](#) for discrete events. Event Grid follows the Publisher-Subscriber pattern. When event sources trigger events, they're published to [Event Grid topics](#). Consumers of those events create Event Grid subscriptions by specifying event types and event handler that will process the events. If there are no subscribers, the events are discarded. Each event can have multiple subscriptions.

Push Model

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

Integrated with Azure

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

Custom topics

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

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

Filtered events

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

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

For more information about filtering, see [Filter events for Event Grid](#).

High throughput

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

Resilient delivery

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

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

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

Azure Event Hubs

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

Fast ingestion

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

Pull model

Like Event Grid, Event Hubs also offers Publisher-Subscriber capabilities. A key difference between Event Grid and Event Hubs is in the way event data is made available to the subscribers. Event Grid pushes the ingested data to the subscribers whereas Event Hubs makes the data available in a pull model. As events are received, Event Hubs appends

them to the stream. A subscriber manages its cursor and can move forward and back in the stream, select a time offset, and replay a sequence at its pace.

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

If you want to act on each event per partition, you can pull the data by using [Event processor host](#), or by using built-in connector such as [Azure Logic Apps](#) to provide the transformation logic. Another option is to use [Azure Functions](#).

Partitioning

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

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

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

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

For more information about Event Hubs partitioning, see [Partitions](#).

Event Hubs Capture

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

Storage services can also offer additional features for analyzing events. For example, by taking advantage of the access tiers of a blob storage account, you can store

events in a hot tier for data that needs frequent access. You might use that data for visualization. Alternately, you can store data in the archive tier and retrieve it occasionally for auditing purposes.

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

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

Support for Apache Kafka clients

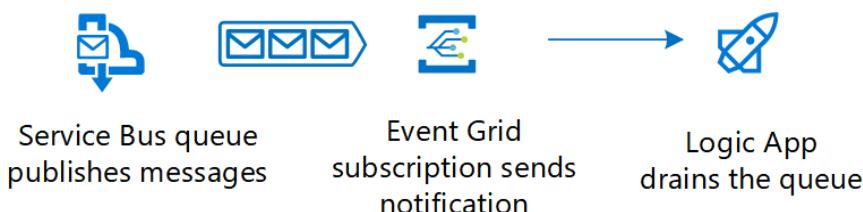
Event Hubs provides an endpoint for [Apache Kafka](#) clients. Existing clients can update their configuration to point to the endpoint and start sending events to Event Hubs. You don't need to make any code changes.

For more information, see [Event Hubs for Apache Kafka](#).

Crossover scenarios

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

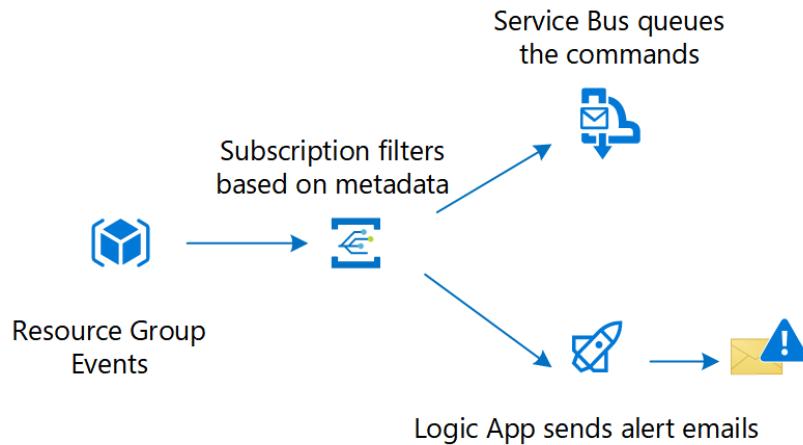
Combining services can increase the efficiency of your messaging system. For instance, in your business transaction, you use Azure Service Bus queues to handle messages. Queues that are mostly idle and receive messages occasionally are inefficient, because the consumer is constantly polling the queue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



For details about connecting Service Bus to Event Grid, see [Azure Service Bus to Event Grid integration overview](#).

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

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



Related patterns

Consider these patterns when implementing asynchronous messaging:

- [Competing Consumers pattern](#). Multiple consumers might need to compete to read messages from a queue. This pattern explains how to process multiple messages concurrently to optimize throughput, to improve scalability and availability, and to balance the workload.
- [Priority Queue pattern](#). For cases where the business logic requires that some messages are processed before others, this pattern describes how messages posted by a producer with a higher priority are received and processed more quickly by a consumer than messages of a lower priority.
- [Queue-based Load Leveling pattern](#). This pattern uses a message broker to act as a buffer between a producer and a consumer to help to minimize the impact on availability and responsiveness of intermittent heavy loads for both those entities.
- [Retry pattern](#). A producer or consumer might be unable connect to a queue, but the reasons for this failure might be temporary and quickly pass. This pattern describes how to handle this situation to add resiliency to an application.
- [Scheduler Agent Supervisor pattern](#). Messaging is often used as part of a workflow implementation. This pattern demonstrates how messaging can coordinate a set of

actions across a distributed set of services and other remote resources, and enable a system to recover and retry actions that fail.

- [Choreography pattern](#). This pattern shows how services can use messaging to control the workflow of a business transaction.
- [Claim-Check pattern](#). This pattern shows how to split a large message into a claim check and a payload.

Community resources

[Jonathon Oliver's blog post: Idempotency ↗](#)

[Martin Fowler's blog post: What do you mean by "Event-Driven"? ↗](#)

Choose an Internet of Things (IoT) solution in Azure

Azure IoT Hub

Azure IoT Central

This article compares using [Azure IoT Central](#) versus individual Azure platform-as-a-service (PaaS) components for building, deploying, and managing internet-of-things (IoT) solutions.

IoT solutions use a combination of technologies to connect devices, events, and actions through cloud applications. The technologies and services you choose depend on your scenario's development, deployment, and management requirements.

The IoT Central application platform-as-a-service (aPaaS) already provides the integrated Azure components and capabilities that an IoT solution needs. Another option is to combine [Azure IoT Hub](#) with other Azure PaaS components to develop your own IoT solutions.

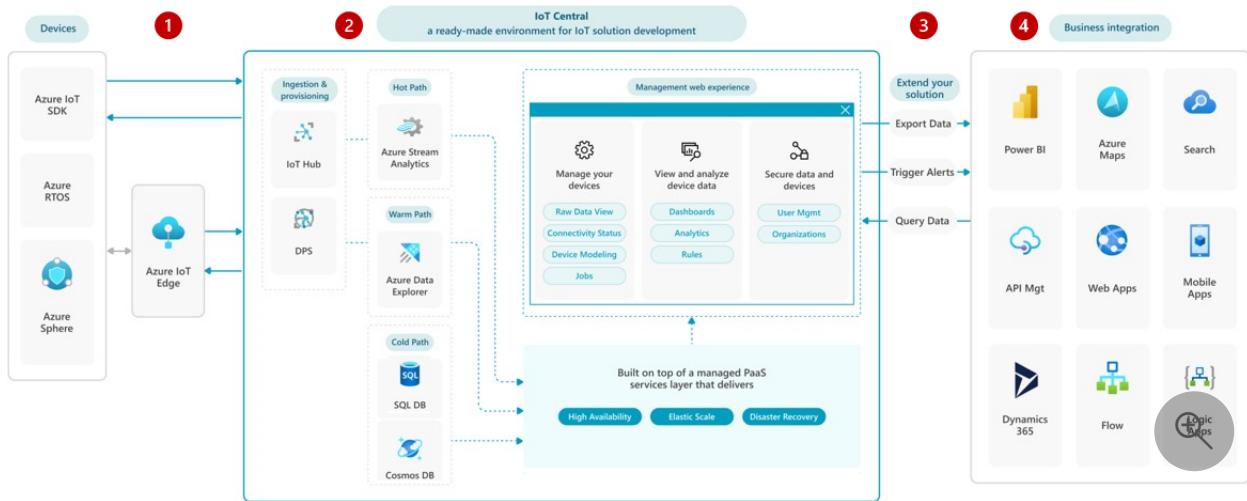
Start with Azure IoT Central

IoT Central is a Microsoft aPaaS that assembles Azure PaaS components into an extensible, fully managed IoT app development and operations platform. IoT Central accelerates solution development, streamlines operations, and simplifies building reliable, scalable, and secure IoT solutions.

IoT Central offers:

- An out-of-the-box web user experience (UX) and API surface area that simplifies device management and rule creation.
- Extension of IoT intelligence into line-of-business applications to help act on insights.
- Built-in disaster recovery, multitenancy, global availability, and a predictable cost structure.

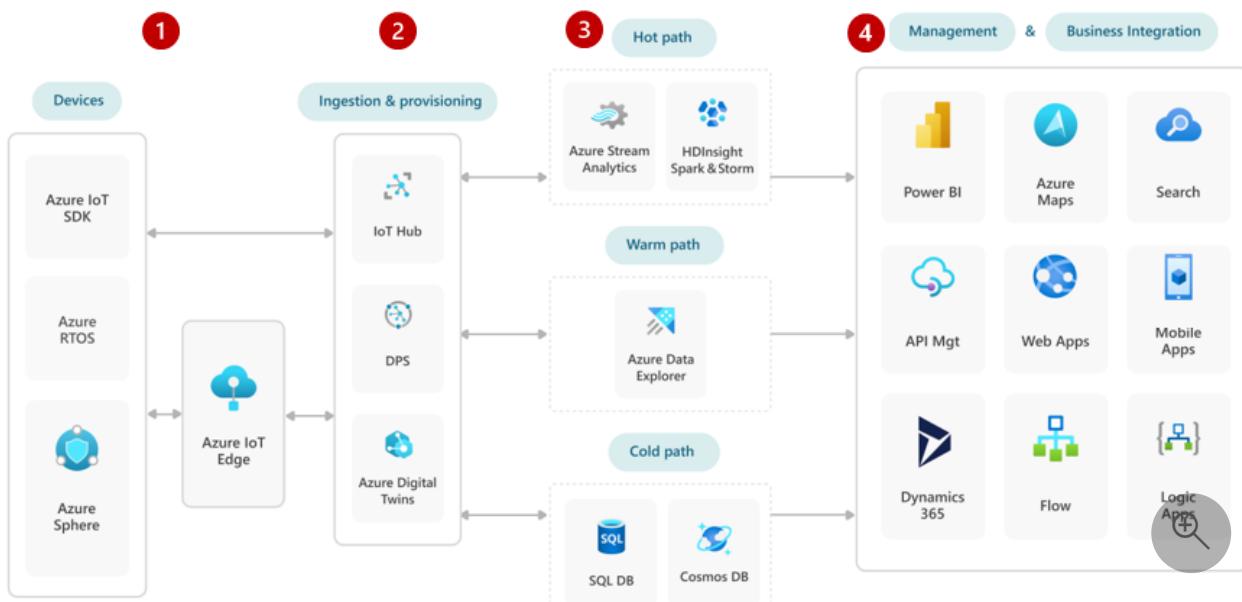
The following diagram shows an IoT Central-based architecture:



1. IoT Central ingests device events and telemetry through the [Azure IoT device SDKs](#), [Azure RTOS](#), [Azure Sphere](#), or [Azure IoT Edge](#).
2. IoT Central is built with multiple Azure PaaS services, so it provides the following capabilities out of the box:
 - Data ingestion and provisioning services.
 - Hot, warm, and cold path data storage and analytics.
 - A managed PaaS layer that delivers High Availability/Disaster Recovery (HADR) and elastic scaling.
 - A management web user experience that lets you:
 - Manage devices with raw data view, connectivity status, device modeling, and jobs.
 - View and analyze device data with dashboards, analytics, and rules.
 - Secure data and devices with user management and organizations.
3. IoT Central extends solutions by triggering alerts, exporting data, and supporting data queries.
4. IoT Central integrates with line-of-business apps like Power BI, Azure Maps, Search, API Management, Web Apps, Mobile Apps, Dynamics 365, Flow, or Logic Apps.

Build with Azure PaaS services

If you need more control and customization, you can use individual Azure PaaS components to build an IoT solution. The following diagram shows Azure services in a PaaS-based IoT architecture:



1. IoT systems can ingest device data through the Azure IoT device SDKs, Azure RTOS, Azure Sphere, or Azure IoT Edge.
2. IoT Hub, [Azure IoT Hub Device Provisioning Service \(DPS\)](#), or [Azure Digital Twins](#) can provide device provisioning, connectivity, and management.
3. For data storage and analytics:
 - The hot path can be through [Azure Stream Analytics](#) or [Azure HDInsight](#).
 - The warm path can be through [Azure Data Explorer](#).
 - The cold path can be through [Azure SQL Database](#) or [Azure Cosmos DB](#).
4. Management and business integration services can include Power BI, Azure Maps, Search, API Management, Web Apps, Mobile Apps, Dynamics 365, Flow, and Logic Apps.

For a detailed PaaS IoT reference architecture and discussion, see [Azure IoT reference architecture](#).

Compare aPaaS and PaaS approaches

IoT Central lets you avoid maintaining and updating a complex and evolving IoT infrastructure. You can focus time and money on transforming your business and designing innovative offerings.

If your solution requires customized features or services that IoT Central doesn't support, you can develop a PaaS solution with IoT Hub as a core element.

The following comparison tables and links can help you decide whether to use an IoT Central managed solution or build a PaaS solution with IoT Hub.

IoT Central vs. IoT Hub-based PaaS solution

The following table describes how IoT Central or an IoT Hub-based PaaS solution achieve various IoT features and capabilities.

[+] Expand table

Feature	IoT Central	IoT Hub-based PaaS
Description	<p>Fully managed aPaaS solution that simplifies device connectivity and management at scale.</p> <p>An aPaaS-based solution is scalable, repeatable, and reliable, with a tradeoff of being less customizable than a PaaS-based solution.</p>	<p>Uses IoT Hub as a central message hub between the IoT application and the devices it manages. Adds more functionality with other Azure PaaS services.</p> <p>This approach is more flexible, but requires greater development and management effort.</p>
Application development	<p>IoT Central is an application platform with support for repeatability of solutions. For more information, see the IoT Central application administration guide</p> <p>Application templates help kick-start IoT solution development. Use a generic application template, or a prebuilt industry-focused template for retail, energy, government, or healthcare.</p>	<p>Design and build your own application solution by using IoT Hub and other PaaS services.</p>
Device template	<p>Device templates help structure device type characteristics and behaviors. Use the templates for supported device management tasks and visualizations.</p>	<p>Define and manage device message templates in a private repository.</p>
Device management	<p>Built-in Azure IoT Device Provisioning Service (DPS) capabilities provide device</p>	<p>Design and build solutions by using IoT Hub primitives, such as device twin and direct</p>

Feature	IoT Central	IoT Hub-based PaaS
	integration and device management.	methods. Enable DPS separately.
OPC UA protocol	Not supported.	Use OPC Publisher to bridge the gap between OPC UA–enabled industrial assets and Azure hosted resources by publishing telemetry data to IoT Hub. OPC Publisher supports IEC62541 OPC UA PubSub standard format and other formats. For more information, see Microsoft OPC Publisher .
SigFox and LoRaWAN protocols	Use Azure IoT Central Device Bridge or Azure IoT Edge LoRaWAN Starter Kit .	Create a custom module for Azure IoT Edge, and integrate it through Azure IoT Hub.
Multi-tenancy	Organizations enable in-app multi-tenancy. You can define a hierarchy to manage which users can see which devices in the IoT Central application.	Achieve multi-tenancy by using separate hubs per customer. You can also build access control into the solution's data layer.
Message retention	IoT Central retains data on a rolling, 30-day basis.	IoT Hub allows data retention in built-in event hubs for a maximum of seven days.
Big data	Manage data from within IoT Central.	Add and manage big data Azure PaaS services.
Data export	Continuously export data by using the export feature . Export data to Azure blob storage, event hubs, service bus, webhook, and Azure Data Explorer. Filter, enrich, and transform messages on egress.	Use the IoT Hub built-in event hub endpoint, and use message routing to export data to other storage locations.
Analytics	An integrated analytics experience explores device data in the context of device management.	Use separate Azure PaaS services to incorporate analytics, insights, and actions, like Stream Analytics, Azure

Feature	IoT Central	IoT Hub-based PaaS
		Data Explorer, and Azure Synapse.
Visualizations	A UX makes it simple to visualize device data, perform analytics queries, and create custom dashboards.	No built-in user interface.
Rules and actions	Use built-in rule and action processing capability with email notification, Azure Monitor group, Power Automate, and webhook actions. For more information, see Azure IoT Central rules and actions .	Send data from IoT Hub to Azure Stream Analytics or Azure Event Grid. Connect to Azure Logic apps or other custom applications to process rules and actions. For more information, see IoT remote monitoring and notifications with Azure Logic Apps .
Scalability	Supports autoscaling. For more information about IoT Central scale limits and autoscaling, see Quotas and limits .	Deploy solutions to enable IoT Hub autoscaling. For more information, see Auto-scale your Azure IoT Hub .
High Availability and Disaster Recovery (HADR)	Manages built-in HADR capabilities automatically. For more information, see Azure IoT Central scalability and high availability .	Design your solution to support multiple HADR scenarios. For more information, see Azure IoT Hub high availability and disaster recovery .
Service Level Agreement (SLA)	Guarantees 99.9% connectivity. For more information, see SLA for Azure IoT Central .	IoT Hub standard and basic tiers guarantee 99.9% uptime. The IoT Hub free tier has no SLA. For more information, see SLA for Azure IoT Hub .
Pricing	The first two active devices are free, if their message volume doesn't exceed 800 (Standard Tier 0 plan), 10,000 (Standard Tier 1 plan), or 60,000 (Standard Tier 2 plan) per month. Added device	For details about IoT Hub pricing, see Azure IoT Hub pricing .

Feature	IoT Central	IoT Hub-based PaaS
	<p>pricing is prorated monthly. IoT Central counts and bills the highest number of active devices each hour. For more information, see Azure IoT Central pricing.</p>	

IoT Central and other Azure PaaS capabilities

The following table shows the level of support for various capabilities in IoT Central and other Azure PaaS services. A filled circle ● means full support, a line — indicates partial support, and an empty circle ○ means no support.

[Expand table](#)

	IoT Central	IoT Hub + DPS	Stream Analytics + Azure Functions	Azure Cosmos DB + Azure Functions	Active Directory
Description	Ready-made IoT solution development environment	IoT data ingestion services	Stream processing services	Data storage services	Universal identity management and security platform
HADR and elastic scale	●	○	○	○	○
Device connectivity management experience	●	—	○	○	○
Data routing, filtering, and rules	—	—	—	○	○

	IoT Central	IoT Hub + DPS	Stream Analytics + Azure Functions	Azure Cosmos DB + Azure Data Explorer	Active Directory
Analytics and visualizations	—	○	—	●	○
Data storage and security	●	○	○	●	●
Export and integration with other services	●	●	●	●	●

Next steps

- [Azure IoT Central overview](#)
- [Azure IoT Hub overview](#)
- [Device management with Azure IoT Hub](#)
- [Azure IoT Hub high availability and disaster recovery](#)
- [Azure IoT Hub SDKs](#)
- [IoT technologies and protocols ↗](#)
- [IoT remote monitoring and notifications with Azure Logic Apps](#)

Related resources

- [IoT conceptual overview](#)
- [Azure IoT reference architecture](#)
- [IoT and data analytics](#)
- [Azure Industrial IoT guidance](#)
- [Vision AI with Azure IoT Edge](#)
- [Retail buy online, pick up in store \(BOPIS\)](#)
- [Environment monitoring and supply chain optimization with IoT](#)
- [Blockchain workflow application ↗](#)
- [IoT using Azure Cosmos DB](#)
- [IoT Edge railroad maintenance and safety solution](#)

- Predictive maintenance for industrial IoT
- Project 15 sustainability open platform
- IoT connected light, power, and internet
- Condition monitoring for industrial IoT

Connecting IoT Devices to Azure: IoT Hub and Event Hubs

Article • 03/16/2023

Azure provides services developed for diverse types of connectivity and communication to help you connect your data to the power of the cloud. Both Azure IoT Hub and Azure Event Hubs are cloud services that can ingest large amounts of data and process or store that data for business insights. The two services are similar in that they both support ingestion of data with low latency and high reliability, but they're designed for different purposes. IoT Hub was developed to address the unique requirements of connecting IoT devices to the Azure cloud while Event Hubs was designed for big data streaming. Microsoft recommends using Azure IoT Hub to connect IoT devices to Azure

Azure IoT Hub is the cloud gateway that connects IoT devices to gather data and drive business insights and automation. In addition, IoT Hub includes features that enrich the relationship between your devices and your backend systems. Bi-directional communication capabilities mean that while you receive data from devices you can also send commands and policies back to devices. For example, use cloud-to-device messaging to update properties or invoke device management actions. Cloud-to-device communication also enables you to send cloud intelligence to your edge devices with Azure IoT Edge. The unique device-level identity provided by IoT Hub helps better secure your IoT solution from potential attacks.

[Azure Event Hubs](#) is the big data streaming service of Azure. It's designed for high throughput data streaming scenarios where customers may send billions of requests per day, and uses a partitioned consumer model to scale out your stream. Event Hubs is integrated into the big data and analytics services of Azure, including Databricks, Stream Analytics, ADLS, and HDInsight. With features like Event Hubs Capture and Auto-Inflate, this service is designed to support your big data apps and solutions. Additionally, IoT Hub uses Event Hubs for its telemetry flow path, so your IoT solution also benefits from the tremendous power of Event Hubs.

To summarize, both solutions are designed for data ingestion at a massive scale. Only IoT Hub provides the rich IoT-specific capabilities that are designed for you to maximize the business value of connecting your IoT devices to the Azure cloud. If your IoT journey is just beginning, starting with IoT Hub to support your data ingestion scenarios assures that you'll have instant access to full-featured IoT capabilities, once your business and technical needs require them.

The following table provides details about how the two tiers of IoT Hub compare to Event Hubs when you're evaluating them for IoT capabilities. For more information about the standard and basic tiers of IoT Hub, see [Choose the right IoT Hub tier for your solution](#).

IoT capability	IoT Hub standard tier	IoT Hub basic tier	Event Hubs
Device-to-cloud messaging	✓	✓	✓
Protocols: HTTPS, AMQP, AMQP over WebSockets	✓	✓	✓
Protocols: MQTT, MQTT over WebSockets	✓	✓	
Per-device identity	✓	✓	
File upload from devices	✓	✓	
Device Provisioning Service	✓	✓	
Cloud-to-device messaging	✓		
Device twin and device management	✓		
Device streams (preview)	✓		
IoT Edge	✓		

Even if the only use case is device-to-cloud data ingestion, we highly recommend using IoT Hub as it provides a service that is designed for IoT device connectivity.

Choose the right Azure command-line tool

Article • 06/19/2023

When it comes to managing Azure, you have many options. This article compares the Azure CLI and Azure PowerShell language and gives a comparison of the shell environments on which they run.

Azure CLI, Azure PowerShell, and Azure Cloud Shell have overlapping functionality. Each operates differently, and the language is sometimes confused with the environment. Use this guide to determine which is the right tool for you.

What's the advantage of using an Azure command-line tool?

Azure runs on automation. Every action you take inside the portal translates somewhere to code being executed to read, create, modify, or delete resources.

Moving your workload to Azure lifts some of the administrative burden but not all. As your rate of adoption with Azure increases so will the overhead. Even though you don't have to worry about the data center, you still have to patch and troubleshoot Azure VMs, failover databases, and configure virtual networks.

By using the existing automation that runs Azure, command-line tools reduce that overhead.

What are Azure command-line tools?

Azure command-line tools automate routine operations, standardize database failovers, and pull data that provide powerful insight. Command-line tools not only give you the ability to scale your tasks in Azure, but they also make much easier to share. Sharing a script is much easier than a lengthy wiki page with time consuming screenshots.

Using an Azure command-line tool isn't always necessary, but it's a useful skill to have.

Azure CLI vs Azure PowerShell

Azure CLI and Azure PowerShell are command-line tools that enable you to create and manage Azure resources. Both are cross-platform, installable on Windows, macOS, and

Linux.

Azure CLI

- Cross-platform command-line interface, installable on Windows, macOS, Linux
- Runs in Windows PowerShell, Cmd, or Bash and other Unix shells.

Azure PowerShell

- Cross-platform PowerShell module, runs on Windows, macOS, Linux
- Requires Windows PowerShell or PowerShell

Different shell environments

Shell Environment	Azure CLI	Azure PowerShell
Cmd	Yes	
Bash	Yes	
Windows PowerShell	Yes	Yes
PowerShell	Yes	Yes

Windows PowerShell, PowerShell, Cmd, and Bash are shell environments. Your shell environment not only determines which tools you can use but also changes your command-line experience.

For example, for the line continuation character, Bash uses the backslash `\` while Windows PowerShell uses the backtick ```. The differences in the shell environment doesn't change how Azure CLI and Azure PowerShell operate. However, they do change your command-line experience.

Azure CLI has an installer that makes its commands executable in all four shell environments.

Azure PowerShell is set of cmdlets packaged as a PowerShell module named `Az`; not an executable. Windows PowerShell or PowerShell must be used to install the `Az` module.

Windows PowerShell is the standard scripting shell that comes preinstalled with most Windows operating systems. PowerShell is a stand-alone installation that uses .NET Core as its run time, allowing it to be installed on macOS, Linux, and Windows.

Key points:

- AzureRM is a PowerShell module that is still referenced for Azure administration with PowerShell. However, it has been replaced by Azure PowerShell and has an official retirement date of February 29 2024.

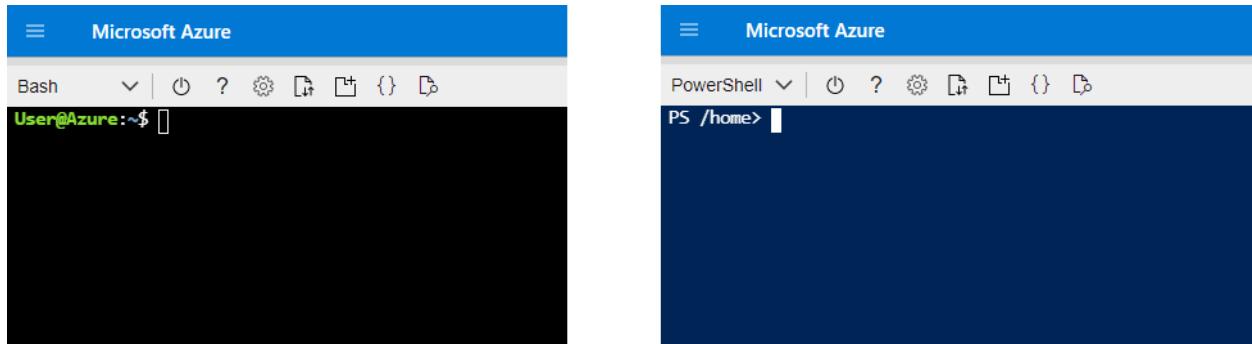
If you're using AzureRM, you can [migrate Azure PowerShell from AzureRM to Az.](#)

What about Azure Cloud Shell?

Azure Cloud Shell is a hosted shell environment that runs on an Ubuntu container.

Cloud Shell provides two shell environments: Bash (with Azure CLI preinstalled) and PowerShell (with Azure PowerShell preinstalled). A dropdown list at the top of the Cloud Shell window allows you to easily switch between the two environments.

Cloud Shell is accessible in a web browser and has integrations for [Windows Terminal](#) and [Visual Studio Code ↗](#).



ⓘ Note

Azure Cloud Shell may not always *immediately reflect* the most recent Azure PowerShell and Azure CLI releases as the publishing schedule for all three tools is different. However, Azure Cloud Shell is generally thought to always contain the most recent versions of both tools.

Which Azure command-line tool is right for you?

When picking the right tool, consider your past experience and current work environment.

Azure CLI syntax is similar to that of Bash scripting. If you work primarily with Linux systems, Azure CLI feels more natural.

Azure PowerShell is a PowerShell module. If you work primarily with Windows systems, Azure PowerShell is a natural fit. Commands follow a verb-noun naming scheme and data is returned as objects.

Choose the tool that uses your experience and shortens your learning curve. Take advantage of [Microsoft learning](#) to become proficient at managing Azure at the command line.

With that said, being open-minded will only improve your abilities. Use a different tool when it makes sense.

Key points:

- Feature parity for Azure services doesn't always exist between Azure CLI and Azure PowerShell.

Azure CLI vs Azure PowerShell: Side-by-side Command Comparison

Sign in, Subscription, and Location Commands:

Command	Azure CLI	Azure PowerShell
Sign in with Web Browser	az login	Connect-AzAccount
Get available subscriptions	az account list	Get-AzSubscription
Set Subscription	az account set --subscription <SubscriptionId>	Set-AzContext -Subscription <SubscriptionID>
List Azure Locations	az account list-locations	Get-AzLocation

Find Versions, Get Help, and View Command Help:

Command	Azure CLI	Azure PowerShell
Find Version	az --version	Get-InstalledModule -Name Az
Get Help	az --help	Get-Help
View Command Help	az vm --help	Get-Help -Name New-AzVM

Create a Resource Group, VM, and Storage Account:

Command	Azure CLI	Azure PowerShell
Create Resource Group	az group create --name <ResourceGroupName> --location eastus	New-AzResourceGroup -Name <ResourceGroupName> -Location eastus
Create Azure Virtual Machine	az vm create --resource-group myResourceGroup --name myVM --image UbuntuLTS --admin-username azureuser --admin-password '<Password>'	New-AzVM -ResourceGroupName <ResourceGroupName> -Name myVM -Image UbuntuLTS -Credential (Get-Credential)
Create Azure Storage Account	az storage account create --name <StorageAccountName> --resource-group <ResourceGroupName> --location eastus --sku Standard_LRS --kind StorageV2	New-AzStorageAccount -Name <StorageAccountName> -ResourceGroupName <ResourceGroupName> -Location eastus -SkuName Standard_LRS -Kind StorageV2

Manage Azure Virtual Machines:

Command	Azure CLI	Azure PowerShell
List VM	az vm list	Get-AzVM
Restart VM	az vm restart --name myVM --resource-group <ResourceGroupName>	Restart-AzVM -Name myVM -ResourceGroupName <ResourceGroupName>
Stop VM	az vm stop --name myVM --resource-group <ResourceGroupName>	Stop-AzVM -Name myVM -ResourceGroupName <ResourceGroupName>
Stop & Deallocate VM	az vm deallocate --name myVM --resource-group <ResourceGroupName>	Stop-AzVM -Name myVM -ResourceGroupName <ResourceGroupName>
Start VM	az vm start --name myVM --resource-group <ResourceGroupName>	Start-AzVM -Name myVM -ResourceGroupName <ResourceGroupName>
Delete VM	az vm delete --name myVM --resource-group <ResourceGroupName>	Remove-AzVM -Name myVM -ResourceGroupName <ResourceGroupName>

Select Properties and Change Output Formats:

Command	Azure CLI	Azure PowerShell
Show all subscription information	az account list --all	Get-AzSubscription Select-Object -Property *
Output as a Table	az account list -o table	Get-AzSubscription Format-Table
Output as JSON	az account show	Get-AzSubscription ConvertTo-Json

Key points:

- Azure CLI defaults to outputting a JSON string. Other format options can be found on the [Output formats for Azure CLI commands](#).
- Azure PowerShell defaults to outputting objects. To learn more about formatting in PowerShell, read the [Using Format Commands to Change Output View](#).

Next steps

Azure CLI:

- [Install the Azure CLI](#)

Azure PowerShell:

- [Install Azure PowerShell](#)

Choose a mobile development framework

Article • 03/04/2022

Developers can use client-side technologies to build mobile applications themselves by using specific frameworks and patterns for a cross-platform approach. Based on their decision factors, developers can build:

- Native single-platform applications by using languages like Objective C and Java
- Cross-platform applications by using Xamarin, .NET, and C#
- Hybrid applications by using Cordova and its variants

Native platforms

Building a native application requires platform-specific programming languages, SDKs, development environments, and other tools provided by OS vendors.

iOS

Created and developed by Apple, iOS is used to build apps on Apple devices, namely the iPhone and the iPad.

- **Programming languages:** Objective-C, Swift
- **IDE:** Xcode
- **SDK:** iOS SDK

Android

Designed by Google and the most popular OS in the world, Android is used to build applications that can run on a range of smartphones and tablets.

- **Programming language:** Java, Kotlin
- **IDE:** Android Studio and Android developer tools
- **SDK:** Android SDK

Windows

- **Programming language:** C#
- **IDE:** Visual Studio, Visual Studio Code

- SDK: Windows SDK

Native platform pros

- Good user experience
- Responsive applications with high performance and the ability to interface with native libraries
- Highly secure applications

Native platform cons

- Application runs on only one platform
- More developer resource intensive and expensive to build an application
- Lower code reuse

Cross-platforms and hybrid applications

Cross-platform applications give you the power to write native mobile applications once, share code, and run them on iOS, Android, and Windows.

Xamarin

Owned by Microsoft, [Xamarin](#) is used to build robust, cross-platform mobile applications in C#. Xamarin has a class library and runtime that works across many platforms, such as iOS, Android, and Windows. It also compiles native (non-interpreted) applications that deliver high performance. Xamarin combines all of the abilities of the native platforms and adds a number of powerful features of its own.

- **Programming language:** C#
- **IDE:** Visual Studio on Windows or Mac

React Native

Released by Facebook in 2015, [React Native](#) is an [open-source](#) JavaScript framework for writing real, natively rendering mobile applications for iOS and Android. It's based on React, Facebook's JavaScript library for building user interfaces. Instead of targeting the browser, it targets mobile platforms. React Native uses native components instead of web components as building blocks.

- **Programming language:** JavaScript
- **IDE:** Visual Studio Code

Unity

Unity is an engine optimized for creating games. You can use it to craft high-quality 2D or 3D applications with C# for platforms such as Windows, iOS, Android, and Xbox.

Cordova

Cordova lets you build hybrid applications by using Visual Studio Tools for Apache Cordova or Visual Studio Code with extensions for Cordova. With the hybrid approach, you can share components with websites and reuse web server-based applications with hosted web application approaches based on Cordova.

Cross-platform pros

- Increased code usability by creating one codebase for multiple platforms
- Cater to a wider audience across many platforms
- Dramatic reduction in development time
- Easy to launch and update

Cross-platform cons

- Lower performance
- Lack of flexibility
- Each platform has a unique set of features and functionality to make the native application more creative
- Increased UI design time
- Tool limitation

Choosing your engine

Article • 06/30/2022

There are several development paths you can take through our documentation. The first step is finding the technology that's right for you. If you already have one in mind, go ahead and jump right to its respective tab below. If you're on the fence or just starting out, take a look through each one and understand what they offer, the available platforms and tools, and start creating!

ⓘ Important

Take a look at our [porting guides overview](#) if you have existing projects that you want to bring over to HoloLens 2 or immersive VR headsets like the Reverb G2. We have guides for projects that are using HTK, MRTK v1, SteamVR or were developed for immersive headsets such as the Oculus Rift or HTC Vive.

Engine overview

- **Unity** is one of the leading real-time development platforms on the market, with underlying runtime code written in C++ and all development scripting is done in C#. Whether you're looking to build games, movies and animation cinematics, or even render architectural or engineering concepts in a virtual world, Unity has the infrastructure to support you.

ⓘ Note

Please make sure to check out [known issues in certain Unity versions](#) before choosing a Unity version.

- **Unreal Engine 4** is a powerful, open source creation engine with full support for mixed reality in both C++ and Blueprints. As of Unreal Engine 4.25, HoloLens support is full-featured and production-ready. With capabilities such as the flexible Blueprints Visual Scripting system, designers can virtually use the full range of concepts and tools generally only available to programmers. Creators across industries can leverage the freedom and control to deliver cutting-edge content, interactive experiences, and immersive virtual worlds.
- **Native** developers with experience writing their own 3D renderers can build a custom engine using OpenXR. OpenXR is an open royalty-free API standard from

Khronos that provides engines native access to a wide range of devices from vendors across the mixed reality spectrum. You can develop using OpenXR on a HoloLens 2 or Windows Mixed Reality immersive headset on the desktop.

- **Web** developers creating compelling cross-browser AR/VR web experiences can use **WebXR**.

Features and devices

Logistics	Unity	Unreal	JavaScript	Custom engine (using OpenXR)
Language	C#	C++	JavaScript	C/C++
Pricing	Unity pricing ↗	Unreal pricing ↗	Free	Free

Device features	Unity	Unreal	JavaScript	Custom engine (using OpenXR)
Device/display tracking	✓	✓	✓	✓
Hand input	✓	✓	✓	✓
Eye input	✓	✓	✗	✓
Voice input	✓	✓	✓	✓
Motion controllers	✓	✓	✓	✓
Plane/mesh hit testing	✓	✓	✓	✓
Scene understanding	✓	✓	✗	✓
Spatial sound	✓	✓	✓	✓
QR code detection	✓	✓	✗	✓

Hardware	Unity	Unreal	JavaScript	Custom engine (using OpenXR)
HoloLens 2	✓	✓	✓	✓
HoloLens (1st gen)	✓	✓	✗	WinRT (Legacy) only

Hardware	Unity	Unreal	JavaScript	Custom engine (using OpenXR)
Windows Mixed Reality headsets	✓	✓	✓	✓
SteamVR headsets	✓	✓	✓	✓
Oculus Quest/Rift	✓	✓	✓	✓
Mobile (ARCore/ARKit)	✓	✓	✓	✗

Tools	Unity	Unreal	JavaScript	Custom engine (using OpenXR)
Mixed Reality Toolkit	✓	✓	✗	✗
World Locking Tools	✓	✗	✗	✗

Cloud services	Unity	Unreal	JavaScript	Custom engine (using OpenXR)
Azure Spatial Anchors	✓	✓	✗	✓
Azure Object Anchors	✓	✗	✗	✓
Azure Remote Rendering	✓ *	✗	✗	✓ *

ⓘ Note

- Azure Remote Rendering is currently supported in apps using the legacy WinRT APIs (Windows XR plugin in Unity). ARR support for OpenXR apps is coming soon.

Next steps

Unity

Next Development Checkpoint

If you're following the Unity for HoloLens development checkpoint journey we've laid out, your next task is to work through our HoloLens 2 tutorial series.

HoloLens 2 tutorial series

Otherwise, continue on to install the right version of Unity and get set up with your first mixed reality Unity project:

Choose the right Unity version for you

You can always go back to the Unity development checkpoints for [HoloLens](#) and [VR](#) at any time.

Best practices in cloud applications

Article • 12/16/2022

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

Catalog of practices

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

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

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

Practice	Summary	Related pillars or patterns
Caching	Improve performance by copying data to fast storage that's close to apps. Cache data that you read often but rarely modify. Manage data expiration and concurrency. See how to populate caches and use the Azure Cache for Redis service.	Data management, Performance efficiency
Content delivery network	Use content delivery networks (CDNs) to efficiently deliver web content to users and reduce load on web apps. Overcome deployment, versioning, security, and resilience challenges.	Data management, Performance efficiency
Data partitioning	Partition data to improve scalability, availability, and performance, and to reduce contention and data storage costs. Use horizontal, vertical, and functional partitioning in efficient ways.	Data management, Performance efficiency, Cost optimization
Data partitioning strategies (by service)	Partition data in Azure SQL Database and Azure Storage services like Azure Table Storage and Azure Blob Storage . Shard your data to distribute loads, reduce latency, and support horizontal scaling.	Data management, Performance efficiency, Cost optimization
Host name preservation	Learn why it's important to preserve the original HTTP host name between a reverse proxy and its back-end web application, and how to implement this recommendation for the most common Azure services.	Design and implementation, Reliability
Message encoding considerations	Use asynchronous messages to exchange information between system components. Choose the payload structure, encoding format, and serialization library that work best with your data.	Messaging, Security
Monitoring and diagnostics	Track system health, usage, and performance with a monitoring and diagnostics pipeline. Turn monitoring data into alerts, reports, and triggers that help in various situations. Examples include detecting and correcting issues, spotting potential problems, meeting performance guarantees, and fulfilling auditing requirements.	Operational excellence
Retry guidance for specific services	Use, adapt, and extend the retry mechanisms that Azure services and client SDKs offer. Develop a systematic and robust approach for managing temporary issues with connections, operations, and resources.	Design and implementation, Reliability

Practice	Summary	Related pillars or patterns
Transient fault handling	Handle transient faults caused by unavailable networks or resources. Overcome challenges when developing appropriate retry strategies. Avoid duplicating layers of retry code and other anti-patterns.	Design and implementation, Reliability

Next steps

- [Web API design](#)
- [Web API implementation](#)

Related resources

- [Cloud design patterns](#)
- [Microsoft Azure Well-Architected Framework](#)

RESTful web API design

Article • 03/28/2023

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

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

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

What is REST?

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

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

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

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

HTTP

<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:

JSON

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

- REST APIs use a uniform interface, which helps to decouple the client and service implementations. For REST APIs built on HTTP, the uniform interface includes using standard HTTP verbs to perform operations on resources. The most common operations are GET, POST, PUT, PATCH, and DELETE.
- REST APIs use a stateless request model. HTTP requests should be independent and may occur in any order, so keeping transient state information between requests is not feasible. The only place where information is stored is in the resources themselves, and each request should be an atomic operation. This constraint enables web services to be highly scalable, because there is no need to retain any affinity between clients and specific servers. Any server can handle any request from any client. That said, other factors can limit scalability. For example, many web services write to a backend data store, which may be hard to scale out. For more information about strategies to scale out a data store, see [Horizontal, vertical, and functional data partitioning](#).
- REST APIs are driven by hypermedia links that are contained in the representation. For example, the following shows a JSON representation of an order. It contains links to get or update the customer associated with the order.

JSON

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

In 2008, Leonard Richardson proposed the following [maturity model](#) for web APIs:

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

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

Organize the API design around resources

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

HTTP

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

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

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

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

HTTP

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

Sending an HTTP GET request to the collection URI retrieves a list of items in the collection. Each item in the collection also has its own unique URI. An HTTP GET request to the item's URI returns the details of that item.

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

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

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

Tip

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

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

information about these performance antipatterns, see [Chatty I/O](#) and [Extraneous Fetching](#).

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

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

Define API operations in terms of HTTP methods

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

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

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

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

The differences between POST, PUT, and PATCH can be confusing.

- A POST request creates a resource. The server assigns a URI for the new resource, and returns that URI to the client. In the REST model, you frequently apply POST requests to collections. The new resource is added to the collection. A POST request can also be used to submit data for processing to an existing resource, without any new resource being created.
- A PUT request creates a resource *or* updates an existing resource. The client specifies the URI for the resource. The request body contains a complete representation of the resource. If a resource with this URI already exists, it is replaced. Otherwise a new resource is created, if the server supports doing so. PUT requests are most frequently applied to resources that are individual items, such as a specific customer, rather than collections. A server might support updates but not creation via PUT. Whether to support creation via PUT depends on whether the client can meaningfully assign a URI to a resource before it exists. If not, then use POST to create resources and PUT or PATCH to update.
- A PATCH request performs a *partial update* to an existing resource. The client specifies the URI for the resource. The request body specifies a set of *changes* to apply to the resource. This can be more efficient than using PUT, because the client only sends the changes, not the entire representation of the resource. Technically PATCH can also create a new resource (by specifying a set of updates to a "null" resource), if the server supports this.

PUT requests must be idempotent. If a client submits the same PUT request multiple times, the results should always be the same (the same resource will be modified with the same values). POST and PATCH requests are not guaranteed to be idempotent.

Conform to HTTP semantics

This section describes some typical considerations for designing an API that conforms to the HTTP specification. However, it doesn't cover every possible detail or scenario. When in doubt, consult the HTTP specifications.

Media types

As mentioned earlier, clients and servers exchange representations of resources. For example, in a POST request, the request body contains a representation of the resource to create. In a GET request, the response body contains a representation of the fetched resource.

In the HTTP protocol, formats are specified through the use of *media types*, also called MIME types. For non-binary data, most web APIs support JSON (media type = application/json) and possibly XML (media type = application/xml).

The Content-Type header in a request or response specifies the format of the representation. Here is an example of a POST request that includes JSON data:

HTTP

```
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:

HTTP

```
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).

If the request was fulfilled but there is no response body included in the HTTP response, then it should return HTTP status code 204 (No Content); for example, a search operation yielding no matches might be implemented with this behavior.

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:

JSON

```
{  
  "name": "gizmo",  
  "category": "widgets",  
  "color": "blue",  
  "price": 10  
}
```

Here is a possible JSON merge patch for this resource:

JSON

```
{  
  "price": 12,  
  "color": null,  
  "size": "small"  
}
```

This tells the server to update `price`, delete `color`, and add `size`, while `name` and `category` are not modified. For the exact details of JSON merge patch, see [RFC 7396](#).

The media type for JSON merge patch is `application/merge-patch+json`.

Merge patch is not suitable if the original resource can contain explicit null values, due to the special meaning of `null` in the patch document. Also, the patch document doesn't specify the order that the server should apply the updates. That may or may not matter, depending on the data and the domain. JSON patch, defined in [RFC 6902](#), is more flexible. It specifies the changes as a sequence of operations to apply. Operations include add, remove, replace, copy, and test (to validate values). The media type for JSON patch is `application/json-patch+json`.

Here are some typical error conditions that might be encountered when processing a PATCH request, along with the appropriate HTTP status code.

Error condition

HTTP status code

Error condition	HTTP status code
The patch document format isn't supported.	415 (Unsupported Media Type)
Malformed patch document.	400 (Bad Request)
The patch document is valid, but the changes can't be applied to the resource in its current state.	409 (Conflict)

DELETE methods

If the delete operation is successful, the web server should respond with HTTP status code 204 (No Content), indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn't exist, the web server can return HTTP 404 (Not Found).

Asynchronous operations

Sometimes a POST, PUT, PATCH, or DELETE operation might require processing that takes a while to complete. If you wait for completion before sending a response to the client, it may cause unacceptable latency. If so, consider making the operation asynchronous. Return HTTP status code 202 (Accepted) to indicate the request was accepted for processing but is not completed.

You should expose an endpoint that returns the status of an asynchronous request, so the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response. For example:

HTTP
<pre>HTTP/1.1 202 Accepted Location: /api/status/12345</pre>

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
<pre>HTTP/1.1 200 OK Content-Type: application/json {</pre>

```
"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

```
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

For more information on how to implement this approach, see [Providing asynchronous support for long-running requests](#) and the [Asynchronous Request-Reply pattern](#).

Empty sets in message bodies

Any time the body of a successful response is empty, the status code should be 204 (No Content). For empty sets, such as a response to a filtered request with no items, the status code should still be 204 (No Content), not 200 (OK).

Filter and paginate data

Exposing a collection of resources through a single URI can lead to applications fetching large amounts of data when only a subset of the information is required. For example, suppose a client application needs to find all orders with a cost over a specific value. It might retrieve all orders from the `/orders` URI and then filter these orders on the client side. Clearly this process is highly inefficient. It wastes network bandwidth and processing power on the server hosting the web API.

Instead, the API can allow passing a filter in the query string of the URI, such as `/orders?minCost=n`. The web API is then responsible for parsing and handling the `minCost` parameter in the query string and returning the filtered results on the server side.

GET requests over collection resources can potentially return a large number of items. You should design a web API to limit the amount of data returned by any single request. Consider supporting query strings that specify the maximum number of items to retrieve and a starting offset into the collection. For example:

HTTP

```
/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:

HTTP

```
HEAD https://adventure-works.com/products/10?fields=productImage HTTP/1.1
```

Here is an example response message:

HTTP

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:

HTTP

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

HTTP/1.1 206 Partial Content

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2500
Content-Range: bytes 0-2499/4580

[...]

A subsequent request from the client application can retrieve the remainder of the resource.

Use HATEOAS to enable navigation to related resources

One of the primary motivations behind REST is that it should be possible to navigate the entire set of resources without requiring prior knowledge of the URI scheme. Each HTTP

GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks included in the response, and it should also be provided with information that describes the operations available on each of these resources. This principle is known as HATEOAS, or Hypertext as the Engine of Application State. The system is effectively a finite state machine, and the response to each request contains the information necessary to move from one state to another; no other information should be necessary.

ⓘ Note

Currently there are no general-purpose standards that define how to model the HATEOAS principle. The examples shown in this section illustrate one possible, proprietary solution.

For example, to handle the relationship between an order and a customer, the representation of an order could include links that identify the available operations for the customer of the order. Here is a possible representation:

JSON

```
{  
    "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": []  
        }  
    ]  
}
```

```

    "types": ["text/xml", "application/json"]
},
{
  "rel": "self",
  "href": "https://adventure-works.com/orders/3",
  "action": "PUT",
  "types": ["application/x-www-form-urlencoded"]
},
{
  "rel": "self",
  "href": "https://adventure-works.com/orders/3",
  "action": "DELETE",
  "types": []
}
]
}

```

In this example, the `links` array has a set of links. Each link represents an operation on a related entity. The data for each link includes the relationship ("customer"), the URI (`https://adventure-works.com/customers/3`), the HTTP method, and the supported MIME types. This is all the information that a client application needs to be able to invoke the operation.

The `links` array also includes self-referencing information about the resource itself that has been retrieved. These have the relationship *self*.

The set of links that are returned may change, depending on the state of the resource. This is what is meant by hypertext being the "engine of application state."

Versioning a RESTful web API

It is highly unlikely that a web API will remain static. As business requirements change new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. While updating a web API to handle new or differing requirements is a relatively straightforward process, you must consider the effects that such changes will have on client applications consuming the web API. The issue is that although the developer designing and implementing a web API has full control over that API, the developer does not have the same degree of control over client applications, which may be built by third-party organizations operating remotely. The primary imperative is to enable existing client applications to continue functioning unchanged while allowing new client applications to take advantage of new features and resources.

Versioning enables a web API to indicate the features and resources that it exposes, and a client application can submit requests that are directed to a specific version of a

feature or resource. The following sections describe several different approaches, each of which has its own benefits and trade-offs.

No versioning

This is the simplest approach, and may be acceptable for some internal APIs. Significant changes could be represented as new resources or new links. Adding content to existing resources might not present a breaking change as client applications that are not expecting to see this content will ignore it.

For example, a request to the URI `https://adventure-works.com/customers/3` should return the details of a single customer containing `id`, `name`, and `address` fields expected by the client application:

```
HTTP
```

```
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
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-
04T12:11:38.0376089Z","address":"1 Microsoft Way Redmond WA 98053"}
```

Existing client applications might continue functioning correctly if they are capable of ignoring unrecognized fields, while new client applications can be designed to handle this new field. However, if more radical changes to the schema of resources occur (such as removing or renaming fields) or the relationships between resources change then these may constitute breaking changes that prevent existing client applications from

functioning correctly. In these situations, you should consider one of the following approaches.

URI versioning

Each time you modify the web API or change the schema of resources, you add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

Extending the previous example, if the `address` field is restructured into subfields containing each constituent part of the address (such as `streetAddress`, `city`, `state`, and `zipCode`), this version of the resource could be exposed through a URI containing a version number, such as `https://adventure-works.com/v2/customers/3`:

```
HTTP  
  
HTTP/1.1 200 OK  
Content-Type: application/json; charset=utf-8  
  
{ "id":3,"name":"Contoso LLC","dateCreated":"2014-09-  
04T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft  
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

This versioning mechanism is very simple but depends on the server routing the request to the appropriate endpoint. However, it can become unwieldy as the web API matures through several iterations and the server has to support a number of different versions. Also, from a purist's point of view, in all cases the client applications are fetching the same data (customer 3), so the URI should not really be different depending on the version. This scheme also complicates implementation of HATEOAS as all links will need to include the version number in their URIs.

Query string versioning

Rather than providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as `https://adventure-works.com/customers/3?version=2`. The `version` parameter should default to a meaningful value such as 1 if it is omitted by older client applications.

This approach has the semantic advantage that the same resource is always retrieved from the same URI, but it depends on the code that handles the request to parse the query string and send back the appropriate HTTP response. This approach also suffers

from the same complications for implementing HATEOAS as the URI versioning mechanism.

ⓘ Note

Some older web browsers and web proxies will not cache responses for requests that include a query string in the URI. This can degrade performance for web applications that use a web API and that run from within such a web browser.

Header versioning

Rather than appending the version number as a query string parameter, you could implement a custom header that indicates the version of the resource. This approach requires that the client application adds the appropriate header to any requests, although the code handling the client request could use a default value (version 1) if the version header is omitted. The following examples use a custom header named *Custom-Header*. The value of this header indicates the version of web API.

Version 1:

HTTP

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=1
```

HTTP

```
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:

HTTP

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=2
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

```
{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft Way","city":"Redmond","state":"WA","zipCode":98053}}
```

As with the previous two approaches, implementing HATEOAS requires including the appropriate custom header in any links.

Media type versioning

When a client application sends an HTTP GET request to a web server it should stipulate the format of the content that it can handle by using an *Accept* header, as described earlier in this guidance. Frequently the purpose of the *Accept* header is to allow the client application to specify whether the body of the response should be XML, JSON, or some other common format that the client can parse. However, it is possible to define custom media types that include information enabling the client application to indicate which version of a resource it is expecting.

The following example shows a request that specifies an *Accept* header with the value *application/vnd.adventure-works.v1+json*. The *vnd.adventure-works.v1* element indicates to the web server that it should return version 1 of the resource, while the *json* element specifies that the format of the response body should be JSON:

HTTP

```
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

```
HTTP/1.1 200 OK
Content-Type: application/vnd.adventure-works.v1+json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"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](#).

Next steps

- [Microsoft Azure REST API Guidelines](#). Detailed recommendations for designing REST APIs on Azure.
- [Web API checklist](#). A useful list of items to consider when designing and implementing a web API.
- [Open API Initiative](#). Documentation and implementation details on Open API.

Web API implementation

Article • 06/12/2023

A carefully designed RESTful web API defines the resources, relationships, and navigation schemes that are accessible to client applications. When you implement and deploy a web API, you should consider the physical requirements of the environment hosting the web API and the way in which the web API is constructed rather than the logical structure of the data. This guidance focuses on best practices for implementing a web API and publishing it to make it available to client applications. For detailed information about web API design, see [Web API design](#).

Processing requests

Consider the following points when you implement the code to handle requests.

GET, PUT, DELETE, HEAD, and PATCH actions should be idempotent

The code that implements these requests should not impose any side-effects. The same request repeated over the same resource should result in the same state. For example, sending multiple DELETE requests to the same URI should have the same effect, although the HTTP status code in the response messages may be different. The first DELETE request might return status code 204 (No Content), while a subsequent DELETE request might return status code 404 (Not Found).

ⓘ Note

The article [Idempotency Patterns](#) on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.

POST actions that create new resources should not have unrelated side-effects

If a POST request is intended to create a new resource, the effects of the request should be limited to the new resource (and possibly any directly related resources if there is some sort of linkage involved) For example, in an e-commerce system, a POST request that creates a new order for a customer might also amend inventory levels and generate

billing information, but it should not modify information not directly related to the order or have any other side-effects on the overall state of the system.

Avoid implementing chatty POST, PUT, and DELETE operations

Support POST, PUT and DELETE requests over resource collections. A POST request can contain the details for multiple new resources and add them all to the same collection, a PUT request can replace the entire set of resources in a collection, and a DELETE request can remove an entire collection.

The OData support included in ASP.NET Web API 2 provides the ability to batch requests. A client application can package up several web API requests and send them to the server in a single HTTP request, and receive a single HTTP response that contains the replies to each request. For more information, [Introducing batch support in Web API and Web API OData ↗](#).

Follow the HTTP specification when sending a response

A web API must return messages that contain the correct HTTP status code to enable the client to determine how to handle the result, the appropriate HTTP headers so that the client understands the nature of the result, and a suitably formatted body to enable the client to parse the result.

For example, a POST operation should return status code 201 (Created) and the response message should include the URI of the newly created resource in the Location header of the response message.

Support content negotiation

The body of a response message may contain data in a variety of formats. For example, an HTTP GET request could return data in JSON, or XML format. When the client submits a request, it can include an Accept header that specifies the data formats that it can handle. These formats are specified as media types. For example, a client that issues a GET request that retrieves an image can specify an Accept header that lists the media types that the client can handle, such as `image/jpeg`, `image/gif`, `image/png`. When the web API returns the result, it should format the data by using one of these media types and specify the format in the Content-Type header of the response.

If the client does not specify an Accept header, then use a sensible default format for the response body. As an example, the ASP.NET Web API framework defaults to JSON

for text-based data.

Provide links to support HATEOAS-style navigation and discovery of resources

The HATEOAS approach enables a client to navigate and discover resources from an initial starting point. This is achieved by using links containing 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 includes HATEOAS links that reference the orders for that customer:

HTTP

```
GET https://adventure-works.com/customers/2 HTTP/1.1
Accept: text/json
...

```

HTTP

```
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"}]
```

```

    "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:

C#

```

public class Customer
{
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public List<Link> Links { get; set; }
    ...
}

public class Link
{
    public string Rel { get; set; }
    public string Href { get; set; }
    public string Action { get; set; }
    public string [] Types { get; set; }
}

```

The HTTP GET operation retrieves the customer data from storage and constructs a `Customer` object, and then populates the `Links` collection. The result is formatted as a JSON response message. Each link comprises the following fields:

- The relationship between the object being returned and the object described by the link. In this case `self` indicates that the link is a reference back to the object itself (similar to a `this` pointer in many object-oriented languages), and `orders` is the name of a collection containing the related order information.
- The hyperlink (`Href`) for the object being described by the link in the form of a URI.
- The type of HTTP request (`Action`) that can be sent to this URI.
- The format of any data (`Types`) that should be provided in the HTTP request or that can be returned in the response, depending on the type of the request.

The HATEOAS links shown in the example HTTP response indicate that a client application can perform the following operations:

- An HTTP GET request to the URI `https://adventure-works.com/customers/2` to fetch the details of the customer (again). The data can be returned as XML or JSON.
- An HTTP PUT request to the URI `https://adventure-works.com/customers/2` to modify the details of the customer. The new data must be provided in the request message in x-www-form-urlencoded format.
- An HTTP DELETE request to the URI `https://adventure-works.com/customers/2` to delete the customer. The request does not expect any additional information or return data in the response message body.
- An HTTP GET request to the URI `https://adventure-works.com/customers/2/orders` to find all the orders for the customer. The data can be returned as XML or JSON.
- An HTTP POST request to the URI `https://adventure-works.com/customers/2/orders` to create a new order for this customer. The data must be provided in the request message in x-www-form-urlencoded format.

Handling exceptions

Consider the following points if an operation throws an uncaught exception.

Capture exceptions and return a meaningful response to clients

The code that implements an HTTP operation should provide comprehensive exception handling rather than letting uncaught exceptions propagate to the framework. If an exception makes it impossible to complete the operation successfully, the exception can be passed back in the response message, but it should include a meaningful description of the error that caused the exception. The exception should also include the appropriate HTTP status code rather than simply returning status code 500 for every situation. For example, if a user request causes a database update that violates a constraint (such as attempting to delete a customer that has outstanding orders), you should return status code 409 (Conflict) and a message body indicating the reason for the conflict. If some other condition renders the request unachievable, you can return status code 400 (Bad Request). You can find a full list of HTTP status codes on the [Status code definitions](#) page on the W3C website.

The code example traps different conditions and returns an appropriate response.

C#

```
[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

Don't 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:

HTTP

```
GET https://adventure-works.com/orders/2 HTTP/1.1
```

HTTP

```
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:

C#

```
public class OrdersController : ApiController
{
    ...
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;

        ...
        // Create a Cache-Control header for the response
        var cacheControlHeader = new CacheControlHeaderValue();
        cacheControlHeader.Private = true;
        cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
        ...

        // Return a response message containing the order and the cache
        control header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>
        (order, this)
        {
            CacheControlHeader = cacheControlHeader
        };
        return response;
    }
}
```

```
    ...  
}
```

This code uses a custom `IActionResult` class named `OkResultWithCaching`. This class enables the controller to set the cache header contents:

C#

```
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) :

C#

```
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

```

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, don't 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:

HTTP

```
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.

① 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:

C#

```
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:

C#

```
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);  
        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:

HTTP

```
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:

C#

```

public class OrdersController : ApiController
{
    [HttpPost]
    [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 don't 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 host 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 build client applications by using the .NET Framework, then all POST and PUT messages will first send messages with Expect: 100-Continue headers by default. As with the server-side, the process is handled transparently by the .NET Framework. However, this process results in each POST and PUT request causing two round-trips to the server, even for small requests. If your application is not sending requests with large amounts of data, you can disable this feature by using the `ServicePointManager` class to create `ServicePoint` objects in the client application. A `ServicePoint` object handles the connections that the client makes to a server based on the scheme and host fragments of URIs that identify resources on the server. You can then set the `Expect100Continue` property of the `ServicePoint` object to false. All subsequent POST and PUT requests made by the client through a URI that matches the scheme and host fragments of the `ServicePoint` object will be sent without Expect: 100-Continue headers. The following code shows how to configure a `ServicePoint` object that configures all requests sent to URLs with a scheme of `http` and a host of `www.contoso.com`.

```
C#
```

```
Uri uri = new Uri("https://www.contoso.com/");  
ServicePoint sp = ServicePointManager.FindServicePoint(uri);  
sp.Expect100Continue = false;
```

You can also set the static `Expect100Continue` property of the `ServicePointManager` class to specify the default value of this property for all subsequently created `ServicePoint`

objects.

Support pagination for requests that may return large numbers of objects

If a collection contains a large number of resources, issuing a GET request to the corresponding URI could result in significant processing on the server hosting the web API affecting performance, and generate a significant amount of network traffic resulting in increased latency.

To handle these cases, the web API should support query strings that enable the client application to refine requests or fetch data in more manageable, discrete blocks (or pages). The code below shows the `GetAllOrders` method in the `Orders` controller. This method retrieves the details of orders. If this method was unconstrained, it could conceivably return a large amount of data. The `limit` and `offset` parameters are intended to reduce the volume of data to a smaller subset, in this case only the first 10 orders by default:

C#

```
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 can't handle URIs that are this long.

Maintaining responsiveness, scalability, and availability

The same web API might be used by many client applications running anywhere in the world. It is important to ensure that the web API is implemented to maintain responsiveness under a heavy load, to be scalable to support a highly varying workload, and to guarantee availability for clients that perform business-critical operations. Consider the following points when determining how to meet these requirements:

Provide asynchronous support for long-running requests

A request that might take a long time to process should be performed without blocking the client that submitted the request. The web API can perform some initial checking to validate the request, initiate a separate task to perform the work, and then return a response message with HTTP code 202 (Accepted). The task could run asynchronously as part of the web API processing, or it could be offloaded to a background task.

The web API should also provide a mechanism to return the results of the processing to the client application. You can achieve this by providing a polling mechanism for client applications to periodically query whether the processing has finished and obtain the result, or enabling the web API to send a notification when the operation has completed.

You can implement a simple polling mechanism by providing a *polling* URI that acts as a virtual resource using the following approach:

1. The client application sends the initial request to the web API.
2. The web API stores information about the request in a table held in [Azure Table Storage](#) or [Microsoft Azure Cache](#) and generates a unique key for this entry, possibly in the form of a GUID. Alternatively, a message containing information about the request and the unique key could be sent via [Azure Service Bus](#) as well.
3. The web API initiates the processing as a [separate task](#) or with a library like [Hangfire](#). The web API records the state of the task in the table as *Running*.
 - If you use Azure Service Bus, the message processing would be done separately from the API, possibly by using [Azure Functions](#) or [AKS](#).
4. The web API returns a response message with HTTP status code 202 (Accepted), and a URI containing the unique key generated - something like `/polling/{guid}`.
5. When the task has completed, the web API stores the results in the table, and it 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*.

- Consider applying [retry techniques](#) to resolve possibly transient failures.
6. While the task is running, the client can continue performing its own processing. It can periodically send a request to the URI it received earlier.
 7. The web API at the 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.
 - If the long-running process has more intermediate states, it's better to use a library that supports the saga pattern, like [NServiceBus](#) or [MassTransit](#).

Options for implementing notifications include:

- Using a notification hub to push asynchronous responses to client applications. For more information, see [Send notifications to specific users by using Azure Notification Hubs](#).
- Using the Comet model to retain a persistent network connection between the client and the server hosting the web API, and using this connection to push messages from the server back to the client. The MSDN magazine article [Building a Simple Comet Application in the Microsoft .NET Framework](#) describes an example solution.
- Using SignalR to push data in real time from the web server to the client over a persistent network connection. SignalR is available for ASP.NET web applications as a NuGet package. You can find more information on the [ASP.NET SignalR](#) website.

Ensure that each request is stateless

Each request should be considered atomic. There should be no dependencies between one request made by a client application and any subsequent requests submitted by the same client. This approach assists in scalability; instances of the web service can be deployed on a number of servers. Client requests can be directed at any of these instances and the results should always be the same. It also improves availability for a similar reason; if a web server fails requests can be routed to another instance (by using Azure Traffic Manager) while the server is restarted with no ill effects on client applications.

Track clients and implement throttling to reduce the chances of DOS attacks

If a specific client makes a large number of requests within a given period of time it might monopolize the service and affect the performance of other clients. To mitigate this issue, a web API can monitor calls from client applications either by tracking the IP address of all incoming requests or by logging each authenticated access. You can use this information to limit resource access. If a client exceeds a defined limit, the web API can return a response message with status 503 (Service Unavailable) and include a Retry-After header that specifies when the client can send the next request without it being declined. This strategy can help to reduce the chances of a Denial Of Service (DOS) attack from a set of clients stalling the system.

Manage persistent HTTP connections carefully

The HTTP protocol supports persistent HTTP connections where they are available. The HTTP 1.0 specification added the Connection:Keep-Alive header that enables a client application to indicate to the server that it can use the same connection to send subsequent requests rather than opening new ones. The connection closes automatically if the client does not reuse the connection within a period defined by the host. This behavior is the default in HTTP 1.1 as used by Azure services, so there is no need to include Keep-Alive headers in messages.

Keeping a connection open can help to improve responsiveness by reducing latency and network congestion, but it can be detrimental to scalability by keeping unnecessary connections open for longer than required, limiting the ability of other concurrent clients to connect. It can also affect battery life if the client application is running on a mobile device; if the application only makes occasional requests to the server, maintaining an open connection can cause the battery to drain more quickly. To ensure that a connection is not made persistent with HTTP 1.1, the client can include a Connection:Close header with messages to override the default behavior. Similarly, if a server is handling a very large number of clients it can include a Connection:Close header in response messages which should close the connection and save server resources.

Note

Persistent HTTP connections are a purely optional feature to reduce the network overhead associated with repeatedly establishing a communications channel. Neither the web API nor the client application should depend on a persistent HTTP connection being available. Don't use persistent HTTP connections to implement Comet-style notification systems; instead you should use sockets (or web sockets if available) at the TCP layer. Finally, note Keep-Alive headers are of limited use if a

client application communicates with a server via a proxy; only the connection with the client and the proxy will be persistent.

Publishing and managing a web API

To make a web API available for client applications, the web API must be deployed to a host environment. This environment is typically a web server, although it may be some other type of host process. You should consider the following points when publishing a web API:

- All requests must be authenticated and authorized, and the appropriate level of access control must be enforced.
- A commercial web API might be subject to various quality guarantees concerning response times. It's 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's useful to be able to decouple these issues from the technical issues concerning the implementation of the web API. For this reason, consider creating a [façade](#), running as a separate process and that routes requests to the web API. The façade can provide the management operations and forward validated requests to the web API. Using a façade can also bring many functional advantages, including:

- Acting as an integration point for multiple web APIs.
- Transforming messages and translating communications protocols for clients built by using varying technologies.
- Caching requests and responses to reduce load on the server hosting the web API.

Testing a web API

A web API should be tested as thoroughly as any other piece of software. You should consider creating unit tests to validate the functionality.

The nature of a web API brings its own additional requirements to verify that it operates correctly. You should pay particular attention to the following aspects:

- Test all routes to verify that they invoke the correct operations. Be especially aware of HTTP status code 405 (Method Not Allowed) being returned unexpectedly as this can indicate a mismatch between a route and the HTTP methods (GET, POST, PUT, DELETE) that can be dispatched to that route.

Send HTTP requests to routes that don't 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 URLs 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.

Using Azure API Management

On Azure, consider using [Azure API Management](#) to publish and manage a web API. Using this facility, you can generate a service that acts as a façade for one or more web APIs. The service is itself a scalable web service that you can create and configure by using the Azure portal. You can use this service to publish and manage a web API as follows:

1. Deploy the web API to a website, Azure cloud service, or Azure virtual machine.
2. Connect the API management service to the web API. Requests sent to the URL of the management API are mapped to URIs in the web API. The same API management service can route requests to more than one web API. This enables you to aggregate multiple web APIs into a single management service. Similarly, the same web API can be referenced from more than one API management service if you need to restrict or partition the functionality available to different applications.

Note

The URIs, in the HATEOAS links that are generated as part of the response for HTTP GET requests, should reference the URL of the API management service

and not the web server that's hosting the web API.

3. For each web API, specify the HTTP operations that the web API exposes together with any optional parameters that an operation can take as input. You can also configure whether the API management service should cache the response received from the web API to optimize repeated requests for the same data. Record the details of the HTTP responses that each operation can generate. This information is used to generate documentation for developers, so it is important that it is accurate and complete.

You can either define operations manually using the wizards provided by the Azure portal, or you can import them from a file containing the definitions in WADL or Swagger format.

4. Configure the security settings for communications between the API management service and the web server hosting the web API. The API management service currently supports Basic authentication and mutual authentication using certificates, and OAuth 2.0 user authorization.
5. Create a product. A product is the unit of publication; you add the web APIs that you previously connected to the management service to the product. When the product is published, the web APIs become available to developers.

 **Note**

Prior to publishing a product, you can also define user-groups that can access the product and add users to these groups. This gives you control over the developers and applications that can use the web API. If a web API is subject to approval, prior to being able to access it a developer must send a request to the product administrator. The administrator can grant or deny access to the developer. Existing developers can also be blocked if circumstances change.

6. Configure policies for each web API. Policies govern aspects such as whether cross-domain calls should be allowed, how to authenticate clients, whether to convert between XML and JSON data formats transparently, whether to restrict calls from a given IP range, usage quotas, and whether to limit the call rate. Policies can be applied globally across the entire product, for a single web API in a product, or for individual operations in a web API.

For more information, see the [API Management documentation](#).

Tip

Azure provides the Azure Traffic Manager which enables you to implement failover and load-balancing, and reduce latency across multiple instances of a web site hosted in different geographic locations. You can use Azure Traffic Manager in conjunction with the API Management Service; the API Management Service can route requests to instances of a web site through Azure Traffic Manager. For more information, see [Traffic Manager routing methods](#).

In this structure, if you use custom DNS names for your web sites, you should configure the appropriate CNAME record for each web site to point to the DNS name of the Azure Traffic Manager web site.

Supporting client-side developers

Developers constructing client applications typically require information on how to access the web API, and documentation concerning the parameters, data types, return types, and return codes that describe the different requests and responses between the web service and the client application.

Document the REST operations for a web API

The Azure API Management Service includes a developer portal that describes the REST operations exposed by a web API. When a product has been published it appears on this portal. Developers can use this portal to sign up for access; the administrator can then approve or deny the request. If the developer is approved, they are assigned a subscription key that is used to authenticate calls from the client applications that they develop. This key must be provided with each web API call otherwise it will be rejected.

This portal also provides:

- Documentation for the product, listing the operations that it exposes, the parameters required, and the different responses that can be returned. Note that this information is generated from the details provided in step 3 in the list in the Publishing a web API by using the Microsoft Azure API Management Service section.
- Code snippets that show how to invoke operations from several languages, including JavaScript, C#, Java, Ruby, Python, and PHP.
- A developers' console that enables a developer to send an HTTP request to test each operation in the product and view the results.
- A page where the developer can report any issues or problems found.

The Azure portal enables you to customize the developer portal to change the styling and layout to match the branding of your organization.

Implement a client SDK

Building a client application that invokes REST requests to access a web API requires writing a significant amount of code to construct each request and format it appropriately, send the request to the server hosting the web service, and parse the response to work out whether the request succeeded or failed and extract any data returned. To insulate the client application from these concerns, you can provide an SDK that wraps the REST interface and abstracts these low-level details inside a more functional set of methods. A client application uses these methods, which transparently convert calls into REST requests and then convert the responses back into method return values. This is a common technique that is implemented by many services, including the Azure SDK.

Creating a client-side SDK is a considerable undertaking as it has to be implemented consistently and tested carefully. However, much of this process can be made mechanical, and many vendors supply tools that can automate many of these tasks.

Monitoring a web API

Depending on how you have published and deployed your web API you can monitor the web API directly, or you can gather usage and health information by analyzing the traffic that passes through the API Management service.

Monitoring a web API directly

If you have implemented your web API by using the ASP.NET Web API template (either as a Web API project or as a Web role in an Azure cloud service) and Visual Studio 2013, you can gather availability, performance, and usage data by using ASP.NET Application Insights. Application Insights is a package that transparently tracks and records information about requests and responses when the web API is deployed to the cloud; once the package is installed and configured, you don't need to amend any code in your web API to use it. When you deploy the web API to an Azure web site, all traffic is examined and the following statistics are gathered:

- Server response time.
- Number of server requests and the details of each request.
- The top slowest requests in terms of average response time.
- The details of any failed requests.

- The number of sessions initiated by different browsers and user agents.
- The most frequently viewed pages (primarily useful for web applications rather than web APIs).
- The different user roles accessing the web API.

You can view this data in real time in the Azure portal. You can also create web tests that monitor the health of the web API. A web test sends a periodic request to a specified URI in the web API and captures the response. You can specify the definition of a successful response (such as HTTP status code 200), and if the request does not return this response you can arrange for an alert to be sent to an administrator. If necessary, the administrator can restart the server hosting the web API if it has failed.

For more information, see [Application Insights - Get started with ASP.NET](#).

Monitoring a web API through the API Management Service

If you have published your web API by using the API Management service, the API Management page on the Azure portal contains a dashboard that enables you to view the overall performance of the service. The Analytics page enables you to drill down into the details of how the product is being used. This page contains the following tabs:

- **Usage.** This tab provides information about the number of API calls made and the bandwidth used to handle these calls over time. You can filter usage details by product, API, and operation.
- **Health.** This tab enables you to view the outcome of API requests (the HTTP status codes returned), the effectiveness of the caching policy, the API response time, and the service response time. Again, you can filter health data by product, API, and operation.
- **Activity.** This tab provides a text summary of the numbers of successful calls, failed calls, blocked calls, average response time, and response times for each product, web API, and operation. This page also lists the number of calls made by each developer.
- **At a glance.** This tab displays a summary of the performance data, including the developers responsible for making the most API calls, and the products, web APIs, and operations that received these calls.

You can use this information to determine whether a particular web API or operation is causing a bottleneck, and if necessary scale the host environment and add more servers. You can also ascertain whether one or more applications are using a disproportionate volume of resources and apply the appropriate policies to set quotas and limit call rates.

Note

You can change the details for a published product, and the changes are applied immediately. For example, you can add or remove an operation from a web API without requiring that you republish the product that contains the web API.

Next steps

- [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

Article • 12/16/2022

Autoscaling is the process of dynamically allocating resources to match performance requirements. As the volume of work grows, an application may need additional resources to maintain the desired performance levels and satisfy service-level agreements (SLAs). As demand slackens and the additional resources are no longer needed, they can be de-allocated to minimize costs.

Autoscaling takes advantage of the elasticity of cloud-hosted environments while easing management overhead. It reduces the need for an operator to continually monitor the performance of a system and make decisions about adding or removing resources.

There are two main ways that an application can scale:

- **Vertical scaling**, also called scaling up and down, means changing the capacity of a resource. For example, you could move an application to a larger VM size. Vertical scaling often requires making the system temporarily unavailable while it is being redeployed. Therefore, it's less common to automate vertical scaling.
- **Horizontal scaling**, also called scaling out and in, means adding or removing instances of a resource. The application continues running without interruption as new resources are provisioned. When the provisioning process is complete, the solution is deployed on these additional resources. If demand drops, the additional resources can be shut down cleanly and deallocated.

Many cloud-based systems, including Microsoft Azure, support automatic horizontal scaling. The rest of this article focuses on horizontal scaling.

ⓘ Note

Autoscaling mostly applies to compute resources. While it's possible to horizontally scale a database or message queue, this usually involves **data partitioning**, which is generally not automated.

Autoscaling components

An autoscaling strategy typically involves the following pieces:

- Instrumentation and monitoring systems at the application, service, and infrastructure levels. These systems capture key metrics, such as response times,

queue lengths, CPU utilization, and memory usage.

- Decision-making logic that evaluates these metrics against predefined thresholds or schedules, and decides whether to scale.
- Components that scale the system.
- Testing, monitoring, and tuning of the autoscaling strategy to ensure that it functions as expected.

Azure provides built-in autoscaling mechanisms that address common scenarios. If a particular service or technology does not have built-in autoscaling functionality, or if you have specific autoscaling requirements beyond its capabilities, you might consider a custom implementation. A custom implementation would collect operational and system metrics, analyze the metrics, and then scale resources accordingly.

Configure autoscaling for an Azure solution

Azure provides built-in autoscaling for most compute options.

- **Azure Virtual Machines** autoscale via [virtual machine scale sets](#), which manage a set of Azure virtual machines as a group. See [How to use automatic scaling and virtual machine scale sets](#).
- **Service Fabric** also supports autoscaling through virtual machine scale sets. Every node type in a Service Fabric cluster is set up as a separate virtual machine scale set. That way, each node type can be scaled in or out independently. See [Scale a Service Fabric cluster in or out using autoscale rules](#).
- **Azure App Service** has built-in autoscaling. Autoscale settings apply to all of the apps within an App Service. See [Scale instance count manually or automatically](#) and [Scale up an app in Azure App Service](#).
- **Azure Cloud Services** has built-in autoscaling at the role level. See [How to configure auto scaling for a Cloud Service in the portal](#).

These compute options all use [Azure Monitor autoscale](#) to provide a common set of autoscaling functionality.

- **Azure Functions** differs from the previous compute options, because you don't need to configure any autoscale rules. Instead, Azure Functions automatically allocates compute power when your code is running, scaling out as necessary to handle load. For more information, see [Choose the correct hosting plan for Azure Functions](#).

Finally, a custom autoscaling solution can sometimes be useful. For example, you could use Azure diagnostics and application-based metrics, along with custom code to monitor and export the application metrics. Then you could define custom rules based on these metrics, and use Resource Manager REST APIs to trigger autoscaling. However, a custom solution is not simple to implement, and should be considered only if none of the previous approaches can fulfill your requirements.

Use the built-in autoscaling features of the platform, if they meet your requirements. If not, carefully consider whether you really need more complex scaling features. Examples of additional requirements may include more granularity of control, different ways to detect trigger events for scaling, scaling across subscriptions, and scaling other types of resources.

Use Azure Monitor autoscale

[Azure Monitor autoscale](#) provide a common set of autoscaling functionality for virtual machine scale sets, Azure App Service, and Azure Cloud Service. Scaling can be performed on a schedule, or based on a runtime metric, such as CPU or memory usage.

Examples:

- Scale out to 10 instances on weekdays, and scale in to 4 instances on Saturday and Sunday.
- Scale out by one instance if average CPU usage is above 70%, and scale in by one instance if CPU usage falls below 50%.
- Scale out by one instance if the number of messages in a queue exceeds a certain threshold.

Scale up the resource when load increases to ensure availability. Similarly, at times of low usage, scale down, so you can optimize cost. Always use a scale-out and scale-in rule combination. Otherwise, the autoscaling takes place only in one direction until it reaches the threshold (maximum or minimum instance counts) set in the profile.

Select a default instance count that's safe for your workload. It's scaled based on that value if maximum or minimum instance counts are not set.

For a list of built-in metrics, see [Azure Monitor autoscaling common metrics](#). You can also implement custom metrics by using Application Insights.

You can configure autoscaling by using PowerShell, the Azure CLI, an Azure Resource Manager template, or the Azure portal. For more detailed control, use the [Azure Resource Manager REST API](#). The [Azure Monitoring Service Management Library](#) and the [Microsoft Insights Library](#) (in preview) are SDKs that allow collecting metrics from

different resources, and perform autoscaling by making use of the REST APIs. For resources where Azure Resource Manager support isn't available, or if you are using Azure Cloud Services, the Service Management REST API can be used for autoscaling. In all other cases, use Azure Resource Manager.

Consider the following points when using Azure autoscale:

- Consider whether you can predict the load on the application accurately enough to use scheduled autoscaling, adding and removing instances to meet anticipated peaks in demand. If this isn't possible, use reactive autoscaling based on runtime metrics, in order to handle unpredictable changes in demand. Typically, you can combine these approaches. For example, create a strategy that adds resources based on a schedule of the times when you know the application is busiest. This helps to ensure that capacity is available when required, without any delay from starting new instances. For each scheduled rule, define metrics that allow reactive autoscaling during that period to ensure that the application can handle sustained but unpredictable peaks in demand.
- It's often difficult to understand the relationship between metrics and capacity requirements, especially when an application is initially deployed. Provision a little extra capacity at the beginning, and then monitor and tune the autoscaling rules to bring the capacity closer to the actual load.
- Configure the autoscaling rules, and then monitor the performance of your application over time. Use the results of this monitoring to adjust the way in which the system scales if necessary. However, keep in mind that autoscaling is not an instantaneous process. It takes time to react to a metric such as average CPU utilization exceeding (or falling below) a specified threshold.
- Autoscaling rules that use a detection mechanism based on a measured trigger attribute (such as CPU usage or queue length) use an aggregated value over time, rather than instantaneous values, to trigger an autoscaling action. By default, the aggregate is an average of the values. This prevents the system from reacting too quickly, or causing rapid oscillation. It also allows time for new instances that are automatically started to settle into running mode, preventing additional autoscaling actions from occurring while the new instances are starting up. For Azure Cloud Services and Azure Virtual Machines, the default period for the aggregation is 45 minutes, so it can take up to this period of time for the metric to trigger autoscaling in response to spikes in demand. You can change the aggregation period by using the SDK, but periods of less than 25 minutes may cause unpredictable results. For Web Apps, the averaging period is much shorter,

allowing new instances to be available in about five minutes after a change to the average trigger measure.

- Avoid *flapping* where scale-in and scale-out actions continually go back and forth. Suppose there are two instances, and upper limit is 80% CPU, lower limit is 60%. When the load is at 85%, another instance is added. After some time, the load decreases to 60%. Before scaling in, the autoscale service calculates the distribution of total load (of three instances) when an instance is removed, taking it to 90%. This means it would have to scale out again immediately. So, it skips scaling-in and you might never see the expected scaling results.

The flapping situation can be controlled by choosing an adequate margin between the scale-out and scale-in thresholds.

- Manual scaling is reset by maximum and minimum number of instances used for autoscaling. If you manually update the instance count to a value higher or lower than the maximum value, the autoscale engine automatically scales back to the minimum (if lower) or the maximum (if higher). For example, you set the range between 3 and 6. If you have one running instance, the autoscale engine scales to three instances on its next run. Likewise, if you manually set the scale to eight instances, on the next run autoscale will scale it back to six instances on its next run. Manual scaling is temporary unless you reset the autoscale rules as well.
- The autoscale engine processes only one profile at a time. If a condition is not met, then it checks for the next profile. Keep key metrics out of the default profile because that profile is checked last. Within a profile, you can have multiple rules. On scale-out, autoscale runs if any rule is met. On scale-in, autoscale require all rules to be met.

For details about how Azure Monitor scales, see [Best practices for Autoscale](#).

- If you configure autoscaling using the SDK rather than the portal, you can specify a more detailed schedule during which the rules are active. You can also create your own metrics and use them with or without any of the existing ones in your autoscaling rules. For example, you may wish to use alternative counters, such as the number of requests per second or the average memory availability, or use custom counters to measure specific business processes.
- When autoscaling Service Fabric, the node types in your cluster are made of virtual machine scale sets at the back end, so you need to set up autoscale rules for each node type. Take into account the number of nodes that you must have before you set up autoscaling. The minimum number of nodes that you must have for the

primary node type is driven by the reliability level you have chosen. For more information, see [scale a Service Fabric cluster in or out using autoscale rules](#).

- You can use the portal to link resources such as SQL Database instances and queues to a Cloud Service instance. This allows you to more easily access the separate manual and automatic scaling configuration options for each of the linked resources. For more information, see [How to: Link a resource to a cloud service](#).
- When you configure multiple policies and rules, they could conflict with each other. Autoscale uses the following conflict resolution rules to ensure that there is always a sufficient number of instances running:
 - Scale-out operations always take precedence over scale-in operations.
 - When scale-out operations conflict, the rule that initiates the largest increase in the number of instances takes precedence.
 - When scale in operations conflict, the rule that initiates the smallest decrease in the number of instances takes precedence.
- In an App Service Environment, any worker pool or front-end metrics can be used to define autoscale rules. For more information, see [Autoscaling and App Service Environment](#).

Application design considerations

Autoscaling isn't an instant solution. Simply adding resources to a system or running more instances of a process doesn't guarantee that the performance of the system will improve. Consider the following points when designing an autoscaling strategy:

- The system must be designed to be horizontally scalable. Avoid making assumptions about instance affinity; do not design solutions that require that the code is always running in a specific instance of a process. When scaling a cloud service or web site horizontally, don't assume that a series of requests from the same source will always be routed to the same instance. For the same reason, design services to be stateless to avoid requiring a series of requests from an application to always be routed to the same instance of a service. When designing a service that reads messages from a queue and processes them, don't make any assumptions about which instance of the service handles a specific message. Autoscaling could start additional instances of a service as the queue length grows. The [Competing Consumers pattern](#) describes how to handle this scenario.
- If the solution implements a long-running task, design this task to support both scaling out and scaling in. Without due care, such a task could prevent an instance

of a process from being shut down cleanly when the system scales in, or it could lose data if the process is forcibly terminated. Ideally, refactor a long-running task and break up the processing that it performs into smaller, discrete chunks. The [Pipes and Filters pattern](#) provides an example of how you can achieve this.

- Alternatively, you can implement a checkpoint mechanism that records state information about the task at regular intervals, and save this state in durable storage that can be accessed by any instance of the process running the task. In this way, if the process is shut down, the work that it was performing can be resumed from the last checkpoint by using another instance. There are libraries that provide this functionality, such as [NServiceBus](#) and [MassTransit](#). They transparently persist state, where the intervals are aligned with the processing of messages from queues in Azure Service Bus.
- 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 the length of the queue over which UI and background compute instances communicate. Use it as a criterion for your autoscaling strategy. This is one possible indicator of an imbalance or difference between the current load and the processing capacity of the background task. There is a slightly more complex, but better attribute to base scaling decisions on. Use the time between when a message was sent and when its processing was complete, known as the *critical time*. If this critical time value is below a meaningful business threshold, then it is unnecessary to scale, even if the queue length is long.
 - For example, there could be 50,000 messages in a queue, but the critical time of the oldest message is 500 ms, and that endpoint is dealing with integration with a 3rd-party web service for sending out emails. It is likely that business stakeholders would consider that to be a period of time that wouldn't justify spending extra money for scaling.
 - On the other hand, there could be 500 messages in a queue, with the same 500 ms critical time, but the endpoint is part of the critical path in some real-time online game, where business stakeholders defined a 100 ms or less response time. In that case, scaling out would make sense.
 - In order to make use of critical time in auto-scaling decisions, it's helpful to have a library automatically add the relevant information to the headers of

messages, while they are sent and processed. One such library that provides this functionality is [NServiceBus](#).

- 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 resources

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

Article • 03/21/2023

Many types of applications require background tasks that run independently of the user interface (UI). Examples include batch jobs, intensive processing tasks, and long-running processes such as workflows. Background jobs can be executed without requiring user interaction--the application can start the job and then continue to process interactive requests from users. This can help to minimize the load on the application UI, which can improve availability and reduce interactive response times.

For example, if an application is required to generate thumbnails of images that are uploaded by users, it can do this as a background job and save the thumbnail to storage when it is complete--without the user needing to wait for the process to be completed. In the same way, a user placing an order can initiate a background workflow that processes the order, while the UI allows the user to continue browsing the web app. When the background job is complete, it can update the stored orders data and send an email to the user that confirms the order.

When you consider whether to implement a task as a background job, the main criteria is whether the task can run without user interaction and without the UI needing to wait for the job to be completed. Tasks that require the user or the UI to wait while they are completed might not be appropriate as background jobs.

Types of background jobs

Background jobs typically include one or more of the following types of jobs:

- CPU-intensive jobs, such as mathematical calculations or structural model analysis.
- I/O-intensive jobs, such as executing a series of storage transactions or indexing files.
- Batch jobs, such as nightly data updates or scheduled processing.
- Long-running workflows, such as order fulfillment, or provisioning services and systems.
- Sensitive-data processing where the task is handed off to a more secure location for processing. For example, you might not want to process sensitive data within a web app. Instead, you might use a pattern such as the [Gatekeeper pattern](#) to transfer the data to an isolated background process that has access to protected storage.

Triggers

Background jobs can be initiated in several different ways. They fall into one of the following categories:

- **Event-driven triggers.** The task is started in response to an event, typically an action taken by a user or a step in a workflow.
- **Schedule-driven triggers.** The task is invoked on a schedule based on a timer. This might be a recurring schedule or a one-off invocation that is specified for a later time.

Event-driven triggers

Event-driven invocation uses a trigger to start the background task. Examples of using event-driven triggers include:

- The UI or another job places a message in a queue. The message contains data about an action that has taken place, such as the user placing an order. The background task listens on this queue and detects the arrival of a new message. It reads the message and uses the data in it as the input to the background job. This pattern is known as [asynchronous message-based communication](#).
- The UI or another job saves or updates a value in storage. The background task monitors the storage and detects changes. It reads the data and uses it as the input to the background job.
- The UI or another job makes a request to an endpoint, such as an HTTPS URI, or an API that is exposed as a web service. It passes the data that is required to complete the background task as part of the request. The endpoint or web service invokes the background task, which uses the data as its input.

Typical examples of tasks that are suited to event-driven invocation include image processing, workflows, sending information to remote services, sending email messages, and provisioning new users in multitenant applications.

Schedule-driven triggers

Schedule-driven invocation uses a timer to start the background task. Examples of using schedule-driven triggers include:

- A timer that is running locally within the application or as part of the application's operating system invokes a background task on a regular basis.
- A timer that is running in a different application, such as Azure Logic Apps, sends a request to an API or web service on a regular basis. The API or web service invokes the background task.

- A separate process or application starts a timer that causes the background task to be invoked once after a specified time delay, or at a specific time.

Typical examples of tasks that are suited to schedule-driven invocation include batch-processing routines (such as updating related-products lists for users based on their recent behavior), routine data processing tasks (such as updating indexes or generating accumulated results), data analysis for daily reports, data retention cleanup, and data consistency checks.

If you use a schedule-driven task that must run as a single instance, be aware of the following:

- If the compute instance that is running the scheduler (such as a virtual machine using Windows scheduled tasks) is scaled, you will have multiple instances of the scheduler running. These could start multiple instances of the task. For more information about this, read this [blog post about idempotence](#).
- If tasks run for longer than the period between scheduler events, the scheduler may start another instance of the task while the previous one is still running.

Returning results

Background jobs execute asynchronously in a separate process, or even in a separate location, from the UI or the process that invoked the background task. Ideally, background tasks are "fire and forget" operations, and their execution progress has no impact on the UI or the calling process. This means that the calling process does not wait for completion of the tasks. Therefore, it cannot automatically detect when the task ends.

If you require a background task to communicate with the calling task to indicate progress or completion, you must implement a mechanism for this. Some examples are:

- Write a status indicator value to storage that is accessible to the UI or caller task, which can monitor or check this value when required. Other data that the background task must return to the caller can be placed into the same storage.
- Establish a reply queue that the UI or caller listens on. The background task can send messages to the queue that indicate status and completion. Data that the background task must return to the caller can be placed into the messages. If you are using Azure Service Bus, you can use the **ReplyTo** and **CorrelationId** properties to implement this capability.
- Expose an API or endpoint from the background task that the UI or caller can access to obtain status information. Data that the background task must return to the caller can be included in the response.

- Have the background task call back to the UI or caller through an API to indicate status at predefined points or on completion. This might be through events raised locally or through a publish-and-subscribe mechanism. Data that the background task must return to the caller can be included in the request or event payload.

Hosting environment

You can host background tasks by using a range of different Azure platform services:

- [Azure Web Apps and WebJobs](#). You can use WebJobs to execute custom jobs based on a range of different types of scripts or executable programs within the context of a web app.
- [Azure Functions](#). You can use functions for background jobs that don't run for a long time. Another use case is if your workload is already hosted on App Service plan and is underutilized.
- [Azure Virtual Machines](#). If you have a Windows service or want to use the Windows Task Scheduler, it is common to host your background tasks within a dedicated virtual machine.
- [Azure Batch](#). Batch is a platform service that schedules compute-intensive work to run on a managed collection of virtual machines. It can automatically scale compute resources.
- [Azure Kubernetes Service \(AKS\)](#). Azure Kubernetes Service provides a managed hosting environment for Kubernetes on Azure.
- [Azure Container Apps](#). Azure Container Apps enables you to build serverless microservices based on containers.

The following sections describe each of these options in more detail, and include considerations to help you choose the appropriate option.

Azure Web Apps and WebJobs

You can use Azure WebJobs to execute custom jobs as background tasks within an Azure Web App. WebJobs run within the context of your web app as a continuous process. WebJobs also run in response to a trigger event from Azure Logic Apps or external factors, such as changes to storage blobs and message queues. Jobs can be started and stopped on demand, and shut down gracefully. If a continuously running WebJob fails, it is automatically restarted. Retry and error actions are configurable.

When you configure a WebJob:

- If you want the job to respond to an event-driven trigger, you should configure it as **Run continuously**. The script or program is stored in the folder named

site/wwwroot/app_data/jobs/continuous.

- If you want the job to respond to a schedule-driven trigger, you should configure it as **Run on a schedule**. The script or program is stored in the folder named site/wwwroot/app_data/jobs/triggered.
- If you choose the **Run on demand** option when you configure a job, it will execute the same code as the **Run on a schedule** option when you start it.

Azure WebJobs run within the sandbox of the web app. This means that they can access environment variables and share information, such as connection strings, with the web app. The job has access to the unique identifier of the machine that is running the job. The connection string named **AzureWebJobsStorage** provides access to Azure storage queues, blobs, and tables for application data, and access to Service Bus for messaging and communication. The connection string named **AzureWebJobsDashboard** provides access to the job action log files.

Azure WebJobs have the following characteristics:

- **Security:** WebJobs are protected by the deployment credentials of the web app.
- **Supported file types:** You can define WebJobs by using command scripts (.cmd), batch files (.bat), PowerShell scripts (.ps1), bash shell scripts (.sh), PHP scripts (.php), Python scripts (.py), JavaScript code (.js), and executable programs (.exe, .jar, and more).
- **Deployment:** You can deploy scripts and executables by using the [Azure portal](#), by using [Visual Studio](#), by using the [Azure WebJobs SDK](#), or by copying them directly to the following locations:
 - For triggered execution: site/wwwroot/app_data/jobs/triggered/{job name}
 - For continuous execution: site/wwwroot/app_data/jobs/continuous/{job name}
- **Logging:** Console.Out is treated (marked) as INFO. Console.Error is treated as ERROR. You can access monitoring and diagnostics information by using the Azure portal. You can download log files directly from the site. They are saved in the following locations:
 - For triggered execution: Vfs/data/jobs/triggered/jobName
 - For continuous execution: Vfs/data/jobs/continuous/jobName
- **Configuration:** You can configure WebJobs by using the portal, the REST API, and PowerShell. You can use a configuration file named settings.job in the same root directory as the job script to provide configuration information for a job. For example:
 - { "stopping_wait_time": 60 }
 - { "is_singleton": true }

Considerations

- By default, WebJobs scale with the web app. However, you can configure jobs to run on single instance by setting the `is_singleton` configuration property to `true`. Single instance WebJobs are useful for tasks that you do not want to scale or run as simultaneous multiple instances, such as reindexing, data analysis, and similar tasks.
- To minimize the impact of jobs on the performance of the web app, consider creating an empty Azure Web App instance in a new App Service plan to host long-running or resource-intensive WebJobs.

Azure Functions

An option that is similar to WebJobs is Azure Functions. This service is serverless that is most suitable for event-driven triggers that run for a short period. A function can also be used to run scheduled jobs through timer triggers, when configured to run at set times.

Azure Functions is not a recommended option for large, long-running tasks because they can cause unexpected timeout issues. However, depending on the hosting plan, they can be considered for schedule-driven triggers.

Considerations

If the background task is expected to run for a short duration in response to an event, consider running the task in a Consumption plan. The execution time is configurable up to a maximum time. A function that runs for longer costs more. Also CPU-intensive jobs that consume more memory can be more expensive. If you use additional triggers for services as part of your task, those are billed separately.

The Premium plan is more suitable if you have a high number of tasks that are short but expected to run continuously. This plan is more expensive because it needs more memory and CPU. The benefit is that you can use features such as virtual network integration.

The Dedicated plan is most suitable for background jobs if your workload already runs on it. If you have underutilized VMs, you can run it on the same VM and share compute costs.

For more information, see these articles:

- [Azure Functions hosting options](#)
- [Timer trigger for Azure Functions](#)

Azure Virtual Machines

Background tasks might be implemented in a way that prevents them from being deployed to Azure Web Apps, or these options might not be convenient. Typical examples are Windows services, and third-party utilities and executable programs. Another example might be programs written for an execution environment that is different than that hosting the application. For example, it might be a Unix or Linux program that you want to execute from a Windows or .NET application. You can choose from a range of operating systems for an Azure virtual machine, and run your service or executable on that virtual machine.

To help you choose when to use Virtual Machines, see [Azure App Services, Cloud Services and Virtual Machines comparison](#). For information about the options for Virtual Machines, see [Sizes for Windows virtual machines in Azure](#). For more information about the operating systems and prebuilt images that are available for Virtual Machines, see [Azure Virtual Machines Marketplace](#).

To initiate the background task in a separate virtual machine, you have a range of options:

- You can execute the task on demand directly from your application by sending a request to an endpoint that the task exposes. This passes in any data that the task requires. This endpoint invokes the task.
- You can configure the task to run on a schedule by using a scheduler or timer that is available in your chosen operating system. For example, on Windows you can use Windows Task Scheduler to execute scripts and tasks. Or, if you have SQL Server installed on the virtual machine, you can use the SQL Server Agent to execute scripts and tasks.
- You can use Azure Logic Apps to initiate the task by adding a message to a queue that the task listens on, or by sending a request to an API that the task exposes.

See the earlier section [Triggers](#) for more information about how you can initiate background tasks.

Considerations

Consider the following points when you are deciding whether to deploy background tasks in an Azure virtual machine:

- Hosting background tasks in a separate Azure virtual machine provides flexibility and allows precise control over initiation, execution, scheduling, and resource

allocation. However, it will increase runtime cost if a virtual machine must be deployed just to run background tasks.

- There is no facility to monitor the tasks in the Azure portal and no automated restart capability for failed tasks--although you can monitor the basic status of the virtual machine and manage it by using the [Azure Resource Manager Cmdlets](#). However, there are no facilities to control processes and threads in compute nodes. Typically, using a virtual machine will require additional effort to implement a mechanism that collects data from instrumentation in the task, and from the operating system in the virtual machine. One solution that might be appropriate is to use the [System Center Management Pack for Azure](#).
- You might consider creating monitoring probes that are exposed through HTTP endpoints. The code for these probes could perform health checks, collect operational information and statistics--or collate error information and return it to a management application. For more information, see the [Health Endpoint Monitoring pattern](#).

For more information, see:

- [Virtual Machines](#)
- [Azure Virtual Machines FAQ](#)

Azure Batch

Consider [Azure Batch](#) if you need to run large, parallel high-performance computing (HPC) workloads across tens, hundreds, or thousands of VMs.

The Batch service provisions the VMs, assign tasks to the VMs, runs the tasks, and monitors the progress. Batch can automatically scale out the VMs in response to the workload. Batch also provides job scheduling. Azure Batch supports both Linux and Windows VMs.

Considerations

Batch works well with intrinsically parallel workloads. It can also perform parallel calculations with a reduce step at the end, or run [Message Passing Interface \(MPI\) applications](#) for parallel tasks that require message passing between nodes.

An Azure Batch job runs on a pool of nodes (VMs). One approach is to allocate a pool only when needed and then delete it after the job completes. This maximizes utilization, because nodes are not idle, but the job must wait for nodes to be allocated.

Alternatively, you can create a pool ahead of time. That approach minimizes the time

that it takes for a job to start, but can result in having nodes that sit idle. For more information, see [Pool and compute node lifetime](#).

For more information, see:

- [What is Azure Batch?](#)
- [Develop large-scale parallel compute solutions with Batch](#)
- [Batch and HPC solutions for large-scale computing workloads](#)

Azure Kubernetes Service

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, which makes it easy to deploy and manage containerized applications.

Containers can be useful for running background jobs. Some of the benefits include:

- Containers support high-density hosting. You can isolate a background task in a container, while placing multiple containers in each VM.
- The container orchestrator handles internal load balancing, configuring the internal network, and other configuration tasks.
- Containers can be started and stopped as needed.
- Azure Container Registry allows you to register your containers inside Azure boundaries. This comes with security, privacy, and proximity benefits.

Considerations

- Requires an understanding of how to use a container orchestrator. Depending on the skill set of your DevOps team, this may or may not be an issue.

For more information, see:

- [Overview of containers in Azure ↗](#)
- [Introduction to private Docker container registries](#)

Azure Container Apps

Azure Container Apps enables you to build serverless microservices based on containers. Distinctive features of Container Apps include:

- Optimized for running general purpose containers, especially for applications that span many microservices deployed in containers.

- Powered by Kubernetes and open-source technologies like [Dapr](#), [KEDA](#), and [envoy](#).
- Supports Kubernetes-style apps and microservices with features like [service discovery](#) and [traffic splitting](#).
- Enables event-driven application architectures by supporting scale based on traffic and pulling from [event sources like queues](#), including [scale to zero](#).
- Support of long running processes and can run [background tasks](#).

Considerations

Azure Container Apps doesn't provide direct access to the underlying Kubernetes APIs. If you require access to the Kubernetes APIs and control plane, you should use [Azure Kubernetes Service](#). However, if you would like to build Kubernetes-style applications and don't require direct access to all the native Kubernetes APIs and cluster management, Container Apps provides a fully managed experience based on best-practices. For these reasons, many teams may prefer to start building container microservices with Azure Container Apps.

For more information, see:

- [Overview of Azure Containers App](#)

You can get started building your first container app [using the quickstarts](#).

Partitioning

If you decide to include background tasks within an existing compute instance, you must consider how this will affect the quality attributes of the compute instance and the background task itself. These factors will help you to decide whether to colocate the tasks with the existing compute instance or separate them out into a separate compute instance:

- **Availability:** Background tasks might not need to have the same level of availability as other parts of the application, in particular the UI and other parts that are directly involved in user interaction. Background tasks might be more tolerant of latency, retried connection failures, and other factors that affect availability because the operations can be queued. However, there must be sufficient capacity to prevent the backup of requests that could block queues and affect the application as a whole.
- **Scalability:** Background tasks are likely to have a different scalability requirement than the UI and the interactive parts of the application. Scaling the UI might be

necessary to meet peaks in demand, while outstanding background tasks might be completed during less busy times by fewer compute instances.

- **Resiliency:** Failure of a compute instance that just hosts background tasks might not fatally affect the application as a whole if the requests for these tasks can be queued or postponed until the task is available again. If the compute instance and/or tasks can be restarted within an appropriate interval, users of the application might not be affected.
- **Security:** Background tasks might have different security requirements or restrictions than the UI or other parts of the application. By using a separate compute instance, you can specify a different security environment for the tasks. You can also use patterns such as Gatekeeper to isolate the background compute instances from the UI in order to maximize security and separation.
- **Performance:** You can choose the type of compute instance for background tasks to specifically match the performance requirements of the tasks. This might mean using a less expensive compute option if the tasks do not require the same processing capabilities as the UI, or a larger instance if they require additional capacity and resources.
- **Manageability:** Background tasks might have a different development and deployment rhythm from the main application code or the UI. Deploying them to a separate compute instance can simplify updates and versioning.
- **Cost:** Adding compute instances to execute background tasks increases hosting costs. You should carefully consider the trade-off between additional capacity and these extra costs.

For more information, see the [Leader Election pattern](#) and the [Competing Consumers pattern](#).

Conflicts

If you have multiple instances of a background job, it is possible that they will compete for access to resources and services, such as databases and storage. This concurrent access can result in resource contention, which might cause conflicts in availability of the services and in the integrity of data in storage. You can resolve resource contention by using a pessimistic locking approach. This prevents competing instances of a task from concurrently accessing a service or corrupting data.

Another approach to resolve conflicts is to define background tasks as a singleton, so that there is only ever one instance running. However, this eliminates the reliability and

performance benefits that a multiple-instance configuration can provide. This is especially true if the UI can supply sufficient work to keep more than one background task busy.

It is vital to ensure that the background task can automatically restart and that it has sufficient capacity to cope with peaks in demand. You can achieve this by allocating a compute instance with sufficient resources, by implementing a queueing mechanism that can store requests for later execution when demand decreases, or by using a combination of these techniques.

Coordination

The background tasks might be complex and might require multiple individual tasks to execute to produce a result or to fulfill all the requirements. It is common in these scenarios to divide the task into smaller discreet steps or subtasks that can be executed by multiple consumers. Multistep jobs can be more efficient and more flexible because individual steps might be reusable in multiple jobs. It is also easy to add, remove, or modify the order of the steps.

Coordinating multiple tasks and steps can be challenging, but there are three common patterns that you can use to guide your implementation of a solution:

- **Decomposing a task into multiple reusable steps.** An application might be required to perform a variety of tasks of varying complexity on the information that it processes. A straightforward but inflexible approach to implementing this application might be to perform this processing as a monolithic module. However, this approach is likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing are required elsewhere within the application. For more information, see the [Pipes and Filters pattern](#).
- **Managing execution of the steps for a task.** An application might perform tasks that comprise a number of steps (some of which might invoke remote services or access remote resources). The individual steps might be independent of each other, but they are orchestrated by the application logic that implements the task. For more information, see [Scheduler Agent Supervisor pattern](#).
- **Managing recovery for task steps that fail.** An application might need to undo the work that is performed by a series of steps (which together define an eventually consistent operation) if one or more of the steps fail. For more information, see the [Compensating Transaction pattern](#).

Resiliency considerations

Background tasks must be resilient in order to provide reliable services to the application. When you are planning and designing background tasks, consider the following points:

- Background tasks must be able to gracefully handle restarts without corrupting data or introducing inconsistency into the application. For long-running or multistep tasks, consider using *check pointing* by saving the state of jobs in persistent storage, or as messages in a queue if this is appropriate. For example, you can persist state information in a message in a queue and incrementally update this state information with the task progress so that the task can be processed from the last known good checkpoint--instead of restarting from the beginning. When using Azure Service Bus queues, you can use message sessions to enable the same scenario. Sessions allow you to save and retrieve the application processing state by using the [SetState](#) and [GetState](#) methods. For more information about designing reliable multistep processes and workflows, see the [Scheduler Agent Supervisor pattern](#).
- When you use queues to communicate with background tasks, the queues can act as a buffer to store requests that are sent to the tasks while the application is under higher than usual load. This allows the tasks to catch up with the UI during less busy periods. It also means that restarts will not block the UI. For more information, see the [Queue-Based Load Leveling pattern](#). If some tasks are more important than others, consider implementing the [Priority Queue pattern](#) to ensure that these tasks run before less important ones.
- Background tasks that are initiated by messages or process messages must be designed to handle inconsistencies, such as messages arriving out of order, messages that repeatedly cause an error (often referred to as *poison messages*), and messages that are delivered more than once. Consider the following:
 - Messages that must be processed in a specific order, such as those that change data based on the existing data value (for example, adding a value to an existing value), might not arrive in the original order in which they were sent. Alternatively, they might be handled by different instances of a background task in a different order due to varying loads on each instance. Messages that must be processed in a specific order should include a sequence number, key, or some other indicator that background tasks can use to ensure that they are processed in the correct order. If you are using Azure Service Bus, you can use message sessions to guarantee the order of delivery. However, it is usually more

efficient, where possible, to design the process so that the message order is not important.

- Typically, a background task will peek at messages in the queue, which temporarily hides them from other message consumers. Then it deletes the messages after they have been successfully processed. If a background task fails when processing a message, that message will reappear on the queue after the peek time-out expires. It will be processed by another instance of the task or during the next processing cycle of this instance. If the message consistently causes an error in the consumer, it will block the task, the queue, and eventually the application itself when the queue becomes full. Therefore, it is vital to detect and remove poison messages from the queue. If you are using Azure Service Bus, messages that cause an error can be moved automatically or manually to an associated [dead letter queue](#).
- Queues are guaranteed at *least once* delivery mechanisms, but they might deliver the same message more than once. In addition, if a background task fails after processing a message but before deleting it from the queue, the message will become available for processing again. Background tasks should be idempotent, which means that processing the same message more than once does not cause an error or inconsistency in the application's data. Some operations are naturally idempotent, such as setting a stored value to a specific new value. However, operations such as adding a value to an existing stored value without checking that the stored value is still the same as when the message was originally sent will cause inconsistencies. Azure Service Bus queues can be configured to automatically remove duplicated messages. For more information on the challenges with at-least-once message delivery, see [the guidance on idempotent message processing](#).
- Some messaging systems, such as Azure storage queues and Azure Service Bus queues, support a de-queue count property that indicates the number of times a message has been read from the queue. This can be useful in handling repeated and poison messages. For more information, see [Asynchronous Messaging Primer](#) and [Idempotency Patterns](#) ↗.

Scaling and performance considerations

Background tasks must offer sufficient performance to ensure they do not block the application, or cause inconsistencies due to delayed operation when the system is under load. Typically, performance is improved by scaling the compute instances that host the

background tasks. When you are planning and designing background tasks, consider the following points around scalability and performance:

- Azure supports autoscaling (both scaling out and scaling back in) based on current demand and load or on a predefined schedule, for Web Apps and Virtual Machines hosted deployments. Use this feature to ensure that the application as a whole has sufficient performance capabilities while minimizing runtime costs.
- Where background tasks have a different performance capability from the other parts of an application (for example, the UI or components such as the data access layer), hosting the background tasks together in a separate compute service allows the UI and background tasks to scale independently to manage the load. If multiple background tasks have significantly different performance capabilities from each other, consider dividing them and scaling each type independently. However, note that this might increase runtime costs.
- Simply scaling the compute resources might not be sufficient to prevent loss of performance under load. You might also need to scale storage queues and other resources to prevent a single point of the overall processing chain from becoming a bottleneck. Also, consider other limitations, such as the maximum throughput of storage and other services that the application and the background tasks rely on.
- Background tasks must be designed for scaling. For example, they must be able to dynamically detect the number of storage queues in use in order to listen on or send messages to the appropriate queue.
- By default, WebJobs scale with their associated Azure Web Apps instance. However, if you want a WebJob to run as only a single instance, you can create a Settings.job file that contains the JSON data { "is_singleton": true }. This forces Azure to only run one instance of the WebJob, even if there are multiple instances of the associated web app. This can be a useful technique for scheduled jobs that must run as only a single instance.

Next steps

- [Compute Partitioning Guidance](#)
- [Asynchronous Messaging Primer](#)
- [Idempotency Patterns ↗](#)

Related resources

- [Queue-Based Load Leveling pattern](#)

- Priority Queue pattern
- Pipes and Filters pattern
- Scheduler Agent Supervisor pattern
- Compensating Transaction pattern
- Leader Election pattern
- Competing Consumers pattern

Caching guidance

Azure Cache for Redis

Caching is a common technique that aims to improve the performance and scalability of a system. It caches data 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:

- They use a private cache, where data is held locally on the computer that's running an instance of an application or service.
- They use a shared cache, serving as a common source that can be accessed by multiple processes and machines.

In both cases, caching can be performed client-side and server-side. Client-side caching is done by the process that provides the user interface for a system, such as a web browser or desktop application. Server-side caching is done by the process that provides the business services that are running remotely.

Private caching

The most basic type of cache is an in-memory store. It's held in the address space of a single process and accessed directly by the code that runs in that process. This type of cache is quick to access. It can also provide an effective means for storing modest amounts of static data. The size of a cache is typically constrained by the amount of memory available on the machine that hosts 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 process will be slower to access than data that's held in memory, but it 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 isn't static, it's 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.

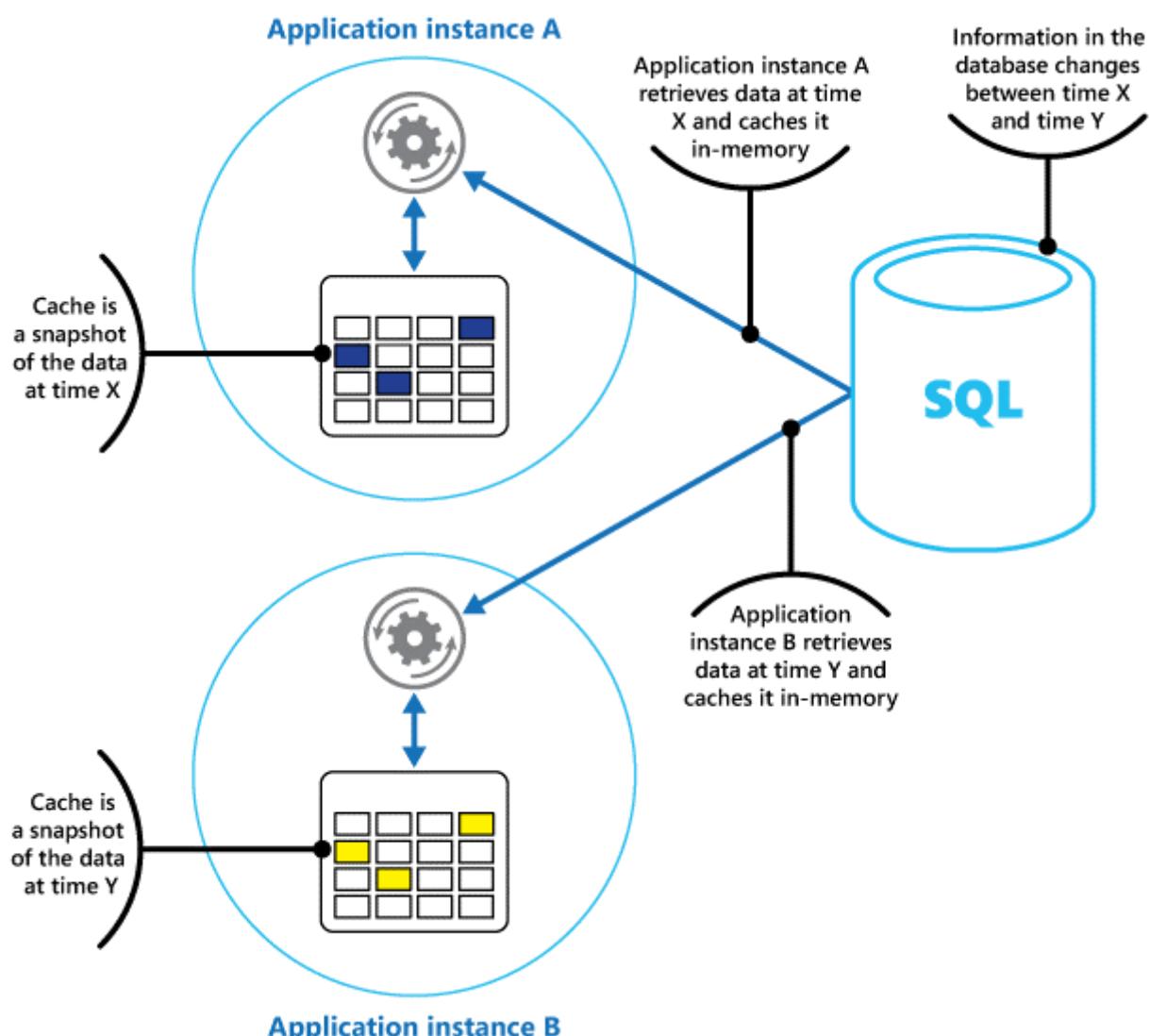


Figure 1: Using an in-memory cache in different instances of an application.

Shared caching

If you use a shared cache, it 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 locates the cache in a separate location, which is typically hosted as part of a separate service, as shown in Figure 2.

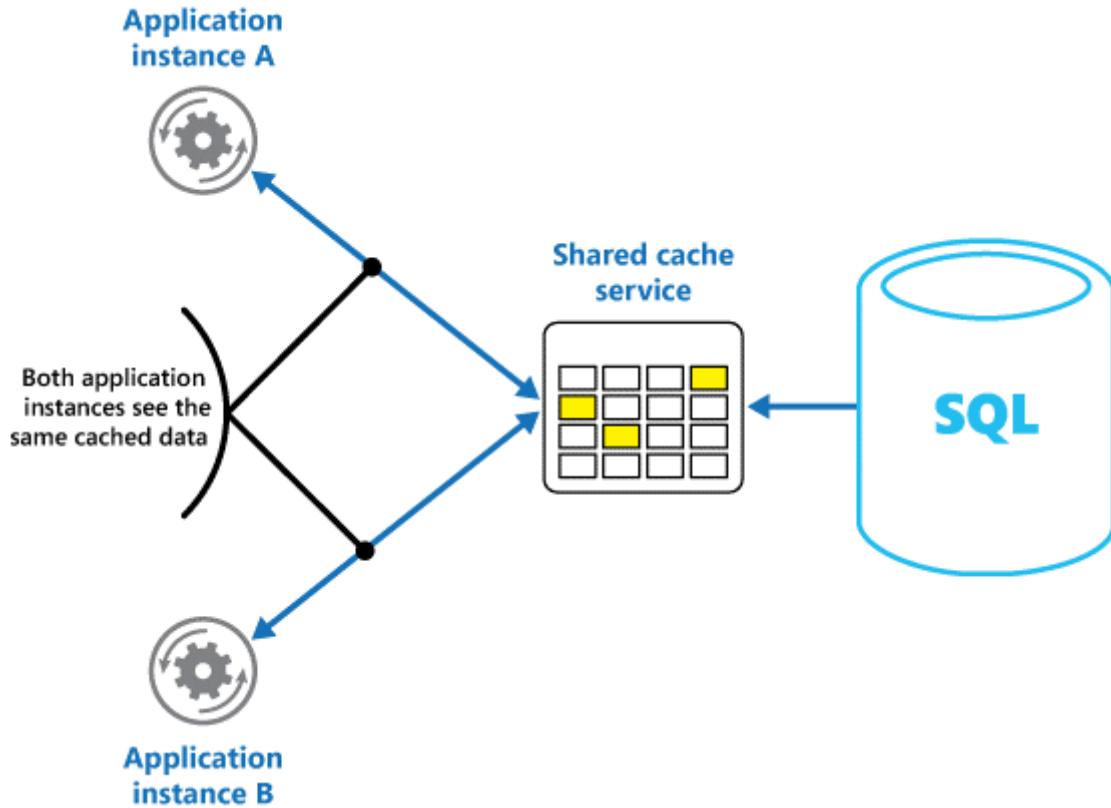


Figure 2: Using a shared cache.

An important benefit of the shared caching approach is the scalability it provides. Many shared cache services are implemented by using a cluster of servers and use software to distribute the data across the cluster transparently. An application instance simply sends a request to the cache service. The underlying infrastructure determines the location of the cached data in the cluster. You can easily scale the cache by adding more servers.

There are two main disadvantages of the shared caching approach:

- The cache is slower to access because it's 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. 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 can't afford to lose are always saved to a persistent data store. 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's retrieved by an application. 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, you can 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. You might also want to have a background process that periodically updates the reference data in the cache to ensure it's up-to-date. Or, the background process can refresh the cache when the 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 quickly or the overhead of synchronizing the cache with the original data store reduces the effectiveness of caching.

A cache doesn't 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. Other elements, 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's required.

We recommend that you carry out performance testing and usage analysis to determine whether prepopulating or on-demand loading of the cache, or a combination of both, is appropriate. The decision should be based on the volatility and usage pattern of the data. Cache utilization and performance analysis are important in applications that encounter heavy loads and must be highly scalable. For example, in highly scalable scenarios you can 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. Don't store valuable data in the cache only; make sure that you maintain the information in the original data store as well. This means that if the cache becomes unavailable, you minimize the chance of losing data.

Cache highly dynamic data

When you store rapidly changing information in a persistent data store, it can impose an overhead on the system. For example, consider a device that continually reports status or some other measurement. If an application chooses not to cache this data on the basis that the cached information will nearly always be outdated, then the same consideration could be true when storing and retrieving this information from the data store. In the time it takes to save and fetch this data, it might have changed.

In a situation such as this, consider the benefits of storing the dynamic information directly in the cache instead of in the persistent data store. If the data is noncritical and doesn't 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 isn't 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 won't 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 can't 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 build 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 don't 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 can't 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 shouldn't become unresponsive or fail while waiting for the cache service to resume.

Therefore, the application must be prepared to detect the availability of the cache service and fall back to the original data store if the cache is inaccessible. The [Circuit-Breaker pattern](#) is useful for handling this scenario. The service that provides the cache can be recovered, and once it becomes available, the cache can be repopulated as data is read from the original data store, following a strategy such as the [Cache-aside pattern](#).

However, system scalability may be affected if the application falls back to the original data store when the cache is temporarily unavailable. While the data store is being recovered, the original data store could be swamped with requests for data, resulting in timeouts and failed connections.

Consider implementing a local, private cache in each instance of an application, together with the shared cache that all application instances access. When the application retrieves an item, it can check first in its local cache, then in the shared cache, and finally in the original data store. The local cache can be populated using the data in either the shared cache, or in the database if the shared cache is unavailable.

This approach requires careful configuration to prevent the local cache from becoming too stale with respect to the shared cache. However, the local cache acts as a buffer if the shared cache is unreachable. Figure 3 shows this structure.

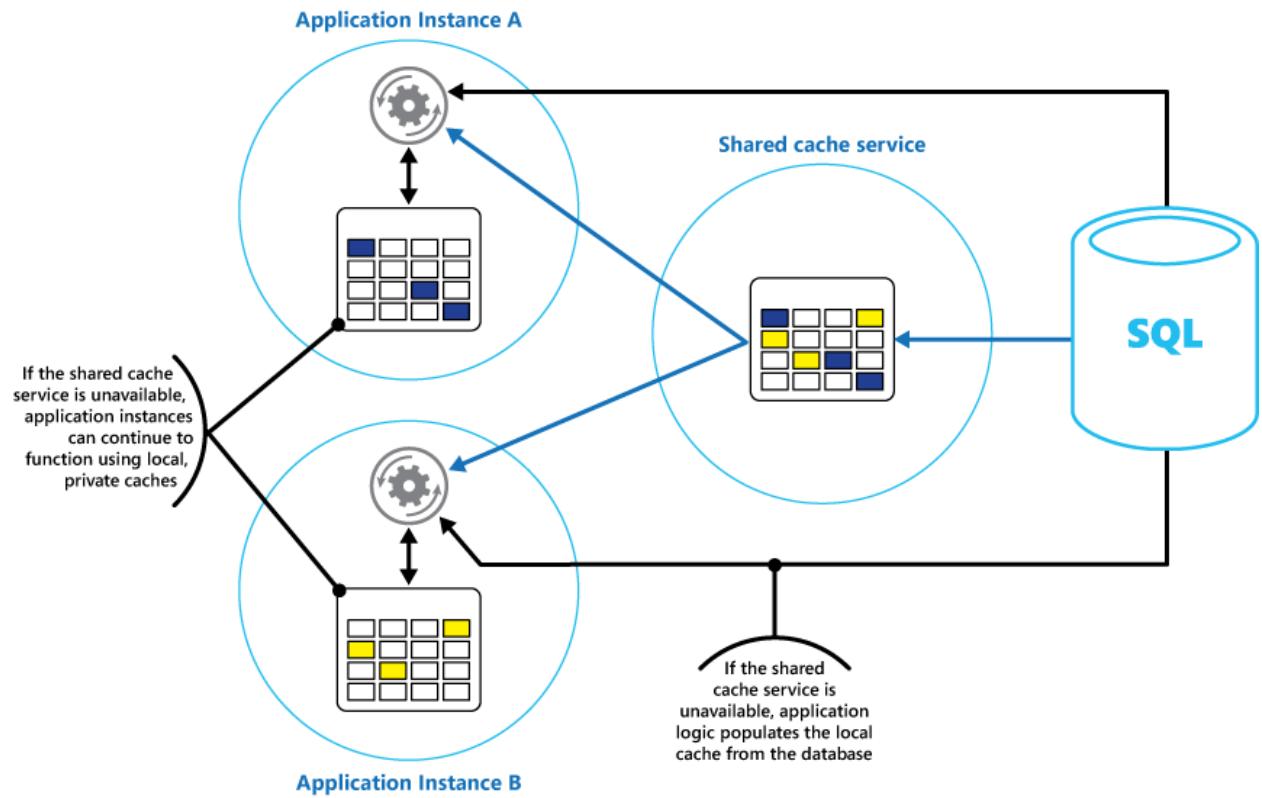


Figure 3: Using a local private cache with a shared cache.

To support large caches that hold relatively long-lived data, some cache services provide a high-availability option that implements automatic failover if the cache becomes unavailable. This approach typically involves replicating the cached data that's stored on a primary cache server to a secondary cache server, and switching to the secondary server if the primary server fails or connectivity is lost.

To reduce the latency that's associated with writing to multiple destinations, the replication to the secondary server might occur asynchronously when data is written to the cache on the primary server. This approach leads to the possibility that some cached information might be lost if there's a failure, but the proportion of this data should be small, compared to the overall size of the cache.

If a shared cache is large, it might be beneficial to partition the cached data across nodes to reduce the chances of contention and improve scalability. Many shared caches support the ability to dynamically add (and remove) nodes and rebalance the data across partitions. This approach might involve clustering, in which the collection of nodes is presented to client applications as a seamless, single cache. Internally, however, the data is dispersed between nodes following a predefined distribution strategy that

balances the load evenly. For more information about possible partitioning strategies, see [Data partitioning guidance](#).

Clustering can also increase the availability of the cache. If a node fails, the remainder of the cache is still accessible. Clustering is frequently used in conjunction with replication and failover. Each node can be replicated, and the replica can be quickly brought online if the node fails.

Many read and write operations are likely to involve single data values or objects. However, at times it might be necessary to store or retrieve large volumes of data quickly. For example, seeding a cache could involve writing hundreds or thousands of items to the cache. An application might also need to retrieve a large number of related items from the cache as part of the same request.

Many large-scale caches provide batch operations for these purposes. This enables a client application to package up a large volume of items into a single request and reduces the overhead that's associated with performing a large number of small requests.

Caching and eventual consistency

For the cache-aside pattern to work, the instance of the application that populates the cache must have access to the most recent and consistent version of the data. In a system that implements eventual consistency (such as a replicated data store) this might not be the case.

One instance of an application could modify a data item and invalidate the cached version of that item. Another instance of the application might attempt to read this item from a cache, which causes a cache-miss, so it reads the data from the data store and adds it to the cache. However, if the data store hasn't been fully synchronized with the other replicas, the application instance could read and populate the cache with the old value.

For more information about handling data consistency, see the [Data consistency primer](#).

Protect cached data

Irrespective of the cache service you use, consider how to protect the data that's held in the cache from unauthorized access. There are two main concerns:

- The privacy of the data in the cache.

- The privacy of data as it flows between the cache and the application that's using the cache.

To protect data in the cache, the cache service might implement an authentication mechanism that requires that applications specify the following:

- Which identities can access data in the cache.
- Which operations (read and write) that these identities are allowed to perform.

To reduce overhead that's associated with reading and writing data, after an identity has been granted write and/or read access to the cache, that identity can use any data in the cache.

If you need to restrict access to subsets of the cached data, you can do one of the following:

- Split the cache into partitions (by using different cache servers) and only grant access to identities for the partitions that they should be allowed to use.
- Encrypt the data in each subset by using different keys, and provide the encryption keys only to identities that should have access to each subset. A client application might still be able to retrieve all of the data in the cache, but it will only be able to decrypt the data for which it has the keys.

You must also protect the data as it flows in and out of the cache. To do this, you depend on the security features provided by the network infrastructure that client applications use to connect to the cache. If the cache is implemented using an on-site server within the same organization that hosts the client applications, then the isolation of the network itself might not require you to take additional steps. If the cache is located remotely and requires a TCP or HTTP connection over a public network (such as the Internet), consider implementing SSL.

Considerations for implementing caching in Azure

[Azure Cache for Redis](#) is an implementation of the open source Redis cache that runs as a service in an Azure datacenter. It provides a caching service that can be accessed from any Azure application, whether the application is implemented as a cloud service, a website, or inside an Azure virtual machine. Caches can be shared by client applications that have the appropriate access key.

Azure Cache for Redis is a high-performance caching solution that provides availability, scalability and security. It typically runs as a service spread across one or more dedicated

machines. It attempts to store as much information as it can in memory to ensure fast access. This architecture is intended to provide low latency and high throughput by reducing the need to perform slow I/O operations.

Azure Cache for Redis is compatible with many of the various APIs that are used by client applications. If you have existing applications that already use Azure Cache for Redis running on-premises, the Azure Cache for Redis provides a quick migration path to caching in the cloud.

Features of Redis

Redis is more than a simple cache server. It provides a distributed in-memory database with an extensive command set that supports many common scenarios. These are described later in this document, in the section [Using Redis caching](#). This section summarizes some of the key features that Redis provides.

Redis as an in-memory database

Redis supports both read and write operations. In Redis, writes can be protected from system failure either by being stored periodically in a local snapshot file or in an append-only log file. This situation isn't the case in many caches, which should be considered transitory data stores.

All writes are asynchronous and don't 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 doesn't guarantee that all writes will be saved if there's a catastrophic failure, but at worst you might lose only a few seconds worth of data. Remember that a cache isn't intended to act as an authoritative data source, and it's the responsibility of the applications using the cache to ensure that critical data is saved successfully to an appropriate data store. For more information, see the [Cache-aside pattern](#).

Redis data types

Redis is a key-value store, where values can contain simple types or complex data structures such as hashes, lists, and sets. It supports a set of atomic operations on these data types. Keys can be permanent or tagged with a limited time-to-live, at which point the key and its corresponding value are automatically removed from the cache. For more information about Redis keys and values, visit the page [An introduction to Redis data types and abstractions](#) on the Redis website.

Redis replication and clustering

Redis supports primary/subordinate replication to help ensure availability and maintain throughput. Write operations to a Redis primary node are replicated to one or more subordinate nodes. Read operations can be served by the primary or any of the subordinates.

If you have a network partition, subordinates can continue to serve data and then transparently resynchronize with the primary when the connection is reestablished. For further details, visit the [Replication](#) page on the Redis website.

Redis also provides clustering, which enables you to transparently partition data into shards across servers and spread the load. This feature improves scalability, because new Redis servers can be added and the data repartitioned as the size of the cache increases.

Furthermore, each server in the cluster can be replicated by using primary/subordinate replication. This ensures availability across each node in the cluster. For more information about clustering and sharding, visit the [Redis cluster tutorial page](#) on the Redis website.

Redis memory use

A Redis cache has a finite size that depends on the resources available on the host computer. When you configure a Redis server, you can specify the maximum amount of memory it can use. You can also configure a key in a Redis cache to have an expiration time, after which it's 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's 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 aren't 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 doesn't roll back the effects of any commands that have already been run. This simplified form of transaction helps to maintain performance and avoid performance problems that are caused by contention.

Redis does implement a form of optimistic locking to assist in maintaining consistency. For detailed information about transactions and locking with Redis, visit the [Transactions page](#) on the Redis website.

Redis also supports nontransactional batching of requests. The Redis protocol that clients use to send commands to a Redis server enables a client to send a series of operations as part of the same request. This can help to reduce packet fragmentation on the network. When the batch is processed, each command is performed. If any of these commands are malformed, they'll be rejected (which doesn't happen with a transaction), but the remaining commands will be performed. There's 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's 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 shouldn't 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 doesn't directly support any form of data encryption, so all encoding must be performed by client applications. Additionally, Redis doesn't provide any form of transport security. If you need to protect data as it flows across the network, we recommend implementing an SSL proxy.

For more information, visit the [Redis security](#) page on the Redis website.

 **Note**

Azure Cache for Redis provides its own security layer through which clients connect. The underlying Redis servers aren't exposed to the public network.

Azure Redis cache

Azure Cache for Redis provides access to Redis servers that are hosted at an Azure datacenter. It acts as a façade that provides access control and security. You can provision a cache by using the Azure portal.

The portal provides a number of predefined configurations. These range from a 53 GB cache running as a dedicated service that supports SSL communications (for privacy) and master/subordinate replication with an SLA of 99.9% availability, down to a 250 MB cache without replication (no availability guarantees) running on shared hardware.

Using the Azure portal, you can also configure the eviction policy of the cache, and control access to the cache by adding users to the roles provided. These roles, which define the operations that members can perform, include Owner, Contributor, and Reader. For example, members of the Owner role have complete control over the cache (including security) and its contents, members of the Contributor role can read and write information in the cache, and members of the Reader role can only retrieve data from the cache.

Most administrative tasks are performed through the Azure portal. For this reason, many of the administrative commands that are available in the standard version of Redis aren't available, including the ability to modify the configuration programmatically, shut down the Redis server, configure additional subordinates, or forcibly save data to disk.

The Azure portal includes a convenient graphical display that enables you to monitor the performance of the cache. For example, you can view the number of connections being made, the number of requests being performed, the volume of reads and writes, and the number of cache hits versus cache misses. Using this information, you can determine the effectiveness of the cache and if necessary, switch to a different configuration or change the eviction policy.

Additionally, you can create alerts that send email messages to an administrator if one or more critical metrics fall outside of an expected range. For example, you might want to alert an administrator if the number of cache misses exceeds a specified value in the last hour, because it means the cache might be too small or data might be being evicted too quickly.

You can also monitor the CPU, memory, and network usage for the cache.

For further information and examples showing how to create and configure an Azure Cache for Redis, visit the page [Lap around Azure Cache for Redis](#) on the Azure blog.

Caching session state and HTML output

If you build ASP.NET web applications that run by using Azure web roles, you can save session state information and HTML output in an Azure Cache for Redis. The session state provider for Azure Cache for Redis enables you to share session information between different instances of an ASP.NET web application, and is very useful in web farm situations where client-server affinity isn't available and caching session data in-memory wouldn't be appropriate.

Using the session state provider with Azure Cache for Redis delivers several benefits, including:

- Sharing session state with a large number of instances of ASP.NET web applications.
- Providing improved scalability.
- Supporting controlled, concurrent access to the same session state data for multiple readers and a single writer.
- Using compression to save memory and improve network performance.

For more information, see [ASP.NET session state provider for Azure Cache for Redis](#).

Note

Don't use the session state provider for Azure Cache for Redis with ASP.NET applications that run outside of the Azure environment. The latency of accessing the cache from outside of Azure can eliminate the performance benefits of caching data.

Similarly, the output cache provider for Azure Cache for Redis enables you to save the HTTP responses generated by an ASP.NET web application. Using the output cache provider with Azure Cache for Redis can improve the response times of applications that render complex HTML output. Application instances that generate similar responses can use the shared output fragments in the cache rather than generating this HTML output afresh. For more information, see [ASP.NET output cache provider for Azure Cache for Redis](#).

Building a custom Redis cache

Azure Cache for Redis acts as a façade to the underlying Redis servers. If you require an advanced configuration that isn't covered by the Azure Redis cache (such as a cache bigger than 53 GB) you can build and host your own Redis servers by using Azure virtual machines.

This is a potentially complex process because you might need to create several VMs to act as primary and subordinate nodes if you want to implement replication.

Furthermore, if you wish to create a cluster, then you need multiple primaries and subordinate servers. A minimal clustered replication topology that provides a high degree of availability and scalability comprises at least six VMs organized as three pairs of primary/subordinate servers (a cluster must contain at least three primary nodes).

Each primary/subordinate pair should be located close together to minimize latency. However, each set of pairs can be running in different Azure datacenters located in different regions, if you wish to locate cached data close to the applications that are most likely to use it. For an example of building and configuring a Redis node running as an Azure VM, see [Running Redis on a CentOS Linux VM in Azure](#).

Note

If you implement your own Redis cache in this way, you're 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 isn't crucial because the cached data is only a transient copy of the data that's held in a database. Cached data on a server that becomes inaccessible can be cached on a different server instead.
- Spreading the load across servers, thereby improving performance and scalability.
- Geolocating data close to the users that access it, thus reducing latency.

For a cache, the most common form of partitioning is sharding. In this strategy, each partition (or shard) is a Redis cache in its own right. Data is directed to a specific partition by using sharding logic, which can use a variety of approaches to distribute the data. The [Sharding pattern](#) provides more information about implementing sharding.

To implement partitioning in a Redis cache, you can take one of the following approaches:

- *Server-side query routing*. In this technique, a client application sends a request to any of the Redis servers that comprise the cache (probably the closest server). Each Redis server stores metadata that describes the partition that it holds, and also contains information about which partitions are located on other servers. The Redis server examines the client request. If it can be resolved locally, it will perform the requested operation. Otherwise it will forward the request on to the appropriate server. This model is implemented by Redis clustering, and is described in more detail on the [Redis cluster tutorial](#) ↗ page on the Redis website. Redis clustering is transparent to client applications, and additional Redis servers can be added to the cluster (and the data re-partitioned) without requiring that you reconfigure the clients.
- *Client-side partitioning*. In this model, the client application contains logic (possibly in the form of a library) that routes requests to the appropriate Redis server. This approach can be used with Azure Cache for Redis. Create multiple Azure Cache for Redis (one for each data partition) and implement the client-side logic that routes the requests to the correct cache. If the partitioning scheme changes (if additional Azure Cache for Redis are created, for example), client applications might need to be reconfigured.
- *Proxy-assisted partitioning*. In this scheme, client applications send requests to an intermediary proxy service which understands how the data is partitioned and then routes the request to the appropriate Redis server. This approach can also be used

with Azure Cache for Redis; the proxy service can be implemented as an Azure cloud service. This approach requires an additional level of complexity to implement the service, and requests might take longer to perform than using client-side partitioning.

The page [Partitioning: how to split data among multiple Redis instances](#) on the Redis website provides further information about implementing partitioning with Redis.

Implement Redis cache client applications

Redis supports client applications written in numerous programming languages. If you build new applications by using the .NET Framework, we recommended you 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's available in Visual Studio as a NuGet package. You can use this same library to connect to an Azure Cache for Redis, or a custom Redis cache hosted on a VM.

To connect to a Redis server you use the static `Connect` method of the `ConnectionMultiplexer` class. The connection that this method creates is designed to be used throughout the lifetime of the client application, and the same connection can be used by multiple concurrent threads. Don't 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 use Azure Cache for Redis, the password is either the primary or secondary key that is generated for Azure Cache for Redis by using the Azure portal.

After you have connected to the Redis server, you can obtain a handle on the Redis database that acts as the cache. The Redis connection provides the `GetDatabase` method to do this. You can then retrieve items from the cache and store data in the cache by using the `StringGet` and `StringSet` methods. These methods expect a key as a parameter, and return the item either in the cache that has a matching value (`StringGet`) or add the item to the cache with this key (`StringSet`).

Depending on the location of the Redis server, many operations might incur some latency while a request is transmitted to the server and a response is returned to the client. The StackExchange library provides asynchronous versions of many of the methods that it exposes to help client applications remain responsive. These methods support the [Task-based Asynchronous pattern](#) in the .NET Framework.

The following code snippet shows a method named `RetrieveItem`. It illustrates an implementation of the cache-aside pattern based on Redis and the StackExchange library. The method takes a string key value and attempts to retrieve the corresponding item from the Redis cache by calling the `StringGetAsync` method (the asynchronous version of `StringGet`).

If the item isn't found, it's 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.

C#

```
// 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 aren't 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:

C#

```
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)
        {
            var jsonString = JsonSerializer.Serialize(o);
            objectDataAsStream = Encoding.ASCII.GetBytes(jsonString);
        }

        return objectDataAsStream;
    }

    static T Deserialize<T>(byte[] stream)
    {
        T result = default(T);

        if (stream != null)
        {
            var jsonString = Encoding.ASCII.GetString(stream);
            result = JsonSerializer.Deserialize<T>(jsonString);
        }

        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:

C#

```
// The BlogPost type
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:

C#

```
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 use the Azure Cache for Redis, see the [Azure Cache for Redis documentation](#). More information is also available at [StackExchange.Redis](#).

The page [Pipelines and multiplexers](#) on the same website provides more information about asynchronous operations and pipelining with Redis and the StackExchange library.

Using Redis caching

The simplest use of Redis for caching concerns is key-value pairs where the value is an uninterpreted string of arbitrary length that can contain any binary data. (It's 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:

C#

```
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:

C#

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string oldValue = await cache.StringGetSetAsync("data:counter", 0);
```

- `MGET` and `MSET`, which can return or change a set of string values as a single operation. The `IDatabase.StringGetAsync` and `IDatabase.StringSetAsync` methods are overloaded to support this functionality, as shown in the following example:

C#

```
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're 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:

C#

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
ITransaction transaction = cache.CreateTransaction();
var tx1 = transaction.StringIncrementAsync("data:counter1");
var tx2 = transaction.StringDecrementAsync("data:counter2");
```

```
bool result = transaction.Execute();
Console.WriteLine("Transaction {0}", result ? "succeeded" : "failed");
Console.WriteLine("Result of increment: {0}", tx1.Result);
Console.WriteLine("Result of decrement: {0}", tx2.Result);
```

Remember that Redis transactions are unlike transactions in relational databases. The `Execute` method simply queues all the commands that comprise the transaction to be run, and if any of them is malformed then the transaction is stopped. If all the commands have been queued successfully, each command runs asynchronously.

If any command fails, the others still continue processing. If you need to verify that a command has completed successfully, you must fetch the results of the command by using the `Result` property of the corresponding task, as shown in the example above. Reading the `Result` property will block the calling thread until the task has completed.

For more information, see [Transactions in Redis ↗](#).

When performing batch operations, you can use the `IBatch` interface of the StackExchange library. This interface provides access to a set of methods similar to those accessed by the `IDatabase` interface, except that all the methods are asynchronous.

You create an `IBatch` object by using the `IDatabase.CreateBatch` method, and then run the batch by using the `IBatch.Execute` method, as shown in the following example. This code simply sets a string value, increments and decrements the same counters used in the previous example, and displays the results:

C#

```
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's important to understand that unlike a transaction, if a command in a batch fails because it's malformed, the other commands might still run. The `IBatch.Execute` method doesn't 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 doesn't wait for the command to be completed. The example below shows how to perform the INCR command as a fire and forget operation:

```
C#  
  
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:

```
C#  
  
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:

```
C#  
  
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:

C#

```
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 = 0;
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:

C#

```

ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Tags are easily represented as Redis Sets
foreach (BlogPost post in posts)
{
    string redisKey = string.Format(CultureInfo.InvariantCulture,
        "blog:posts:{0}:tags", post.Id);
    // Add tags to the blog post in Redis
    await cache.SetAddAsync(
        redisKey, post.Tags.Select(s => (RedisValue)s).ToArray());

    // Now do the inverse so we can figure out which blog posts have a given
    tag
    foreach (var tag in post.Tags)
    {
        await cache.SetAddAsync(string.Format(CultureInfo.InvariantCulture,
            "tag:{0}:blog:posts", tag), post.Id);
    }
}

```

These structures enable you to perform many common queries very efficiently. For example, you can find and display all of the tags for blog post 1 like this:

```
C#
```

```
// 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:

```
C#
```

```
// 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:

```
C#
```

```
// 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 `IDatabase` 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.

```
C#
```

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string redisKey = "blog:recent_posts";
BlogPost blogPost = ...; // Reference to the blog post that has just been
read
await cache.ListLeftPushAsync(
    redisKey, blogPost.Title); // Push the blog post onto the list
```

As more blog posts are read, their titles are pushed onto the same list. The list is ordered by the sequence in which the titles have been added. The most recently read blog posts are toward the left end of the list. (If the same blog post is read more than once, it will have multiple entries in the list.)

You can display the titles of the most recently read posts by using the `IDatabase.ListRange` method. This method takes the key that contains the list, a starting point, and an ending point. The following code retrieves the titles of the 10 blog posts (items from 0 to 9) at the left-most end of the list:

```
C#
```

```
// Show latest ten posts
foreach (string postTitle in await cache.ListRangeAsync(redisKey, 0, 9))
{
    Console.WriteLine(postTitle);
}
```

Note that the `ListRangeAsync` method doesn't 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:

C#

```
await cache.ListTrimAsync(redisKey, 0, 5);
```

Implement a leader board

By default, the items in a set aren't 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.

C#

```
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:

C#

```
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 it doesn't 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:

C#

```
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:

C#

```
// 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":

C#

```
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, but we recommend you 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 `IStackExchangeServer.PublishAsync` method to perform this operation. The next code snippet shows how to publish a message to the "messages:blogPosts" channel:

C#

```
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'll all receive the messages that are published to that channel.
- Subscribers only receive messages that have been published after they've subscribed. Channels aren't 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're 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:

C#

```
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 evaluate the 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's 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's 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 doesn't use message schemas. Being a text-based format, it isn't 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 aren't 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.

Next steps

- [Azure Cache for Redis documentation](#)
- [Azure Cache for Redis FAQ](#)
- [Task-based Asynchronous pattern](#)
- [Redis documentation](#)
- [StackExchange.Redis](#)
- [Data partitioning guide](#)

Related resources

The following patterns might also be relevant to your scenario when you implement caching in your applications:

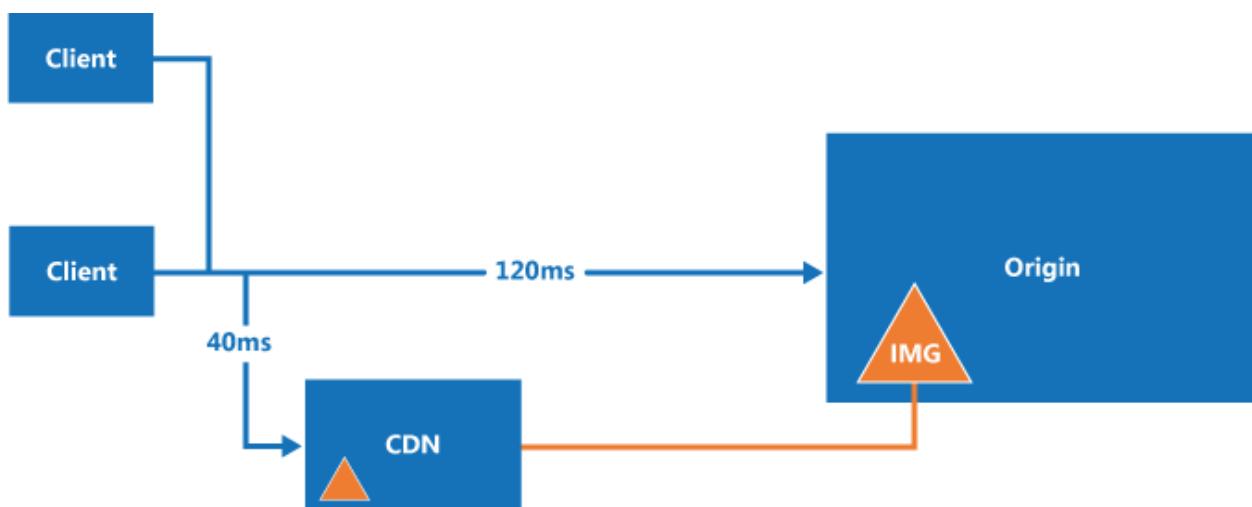
- [Cache-aside pattern](#): This pattern describes how to load data on demand into a cache from a data store. This pattern also helps to maintain consistency between data that's held in the cache and the data in the original data store.
- The [Sharding pattern](#) provides information about implementing horizontal partitioning to help improve scalability when storing and accessing large volumes of data.

CDN guidance

Azure Storage Azure Blob Storage

A content delivery network (CDN) is a distributed network of servers that can efficiently deliver web content to users. CDNs store cached content on edge servers that are close to end users to minimize latency.

CDNs are typically used to deliver static content such as images, style sheets, documents, client-side scripts, and HTML pages. The major advantages of using a CDN are lower latency and faster delivery of content to users, regardless of their geographical location in relation to the datacenter where the application is hosted. CDNs can also help to reduce load on a web application, because the application does not have to service requests for the content that is hosted in the CDN.



In Azure, the [Azure Content Delivery Network](#) is a global CDN solution for delivering high-bandwidth content that is hosted in Azure or any other location. Using Azure CDN, you can cache publicly available objects loaded from Azure blob storage, a web application, virtual machine, any publicly accessible web server.

This topic describes some general best practices and considerations when using a CDN. For more information, see [Azure CDN](#).

How and why a CDN is used

Typical uses for a CDN include:

- Delivering static resources for client applications, often from a website. These resources can be images, style sheets, documents, files, client-side scripts, HTML pages, HTML fragments, or any other content that the server does not need to

modify for each request. The application can create items at runtime and make them available to the CDN (for example, by creating a list of current news headlines), but it does not do so for each request.

- Delivering public static and shared content to devices such as mobile phones and tablet computers. The application itself is a web service that offers an API to clients running on the various devices. The CDN can also deliver static datasets (via the web service) for the clients to use, perhaps to generate the client UI. For example, the CDN could be used to distribute JSON or XML documents.
- Serving entire websites that consist of only public static content to clients, without requiring any dedicated compute resources.
- Streaming video files to the client on demand. Video benefits from the low latency and reliable connectivity available from the globally located datacenters that offer CDN connections. Microsoft Azure Media Services (AMS) integrates with Azure CDN to deliver content directly to the CDN for further distribution. For more information, see [Streaming endpoints overview](#).
- Generally improving the experience for users, especially those located far from the datacenter hosting the application. These users might otherwise suffer higher latency. A large proportion of the total size of the content in a web application is often static, and using the CDN can help to maintain performance and overall user experience while eliminating the requirement to deploy the application to multiple datacenters. For a list of Azure CDN node locations, see [Azure CDN POP Locations](#).
- Supporting IoT (Internet of Things) solutions. The huge numbers of devices and appliances involved in an IoT solution could easily overwhelm an application if it had to distribute firmware updates directly to each device.
- Coping with peaks and surges in demand without requiring the application to scale, avoiding the consequent increase in running costs. For example, when an update to an operating system is released for a hardware device such as a specific model of router, or for a consumer device such as a smart TV, there will be a huge peak in demand as it is downloaded by millions of users and devices over a short period.

Challenges

There are several challenges to take into account when planning to use a CDN.

- **Deployment.** Decide the origin from which the CDN fetches the content, and whether you need to deploy the content in more than one storage system. Take

into account the process for deploying static content and resources. For example, you may need to implement a separate step to load content into Azure blob storage.

- **Versioning and cache-control.** Consider how you will update static content and deploy new versions. Understand how the CDN performs caching and time-to-live (TTL). For Azure CDN, see [How caching works](#).
- **Testing.** It can be difficult to perform local testing of your CDN settings when developing and testing an application locally or in a staging environment.
- **Search engine optimization (SEO).** Content such as images and documents are served from a different domain when you use the CDN. This can have an effect on SEO for this content.
- **Content security.** Not all CDNs offer any form of access control for the content. Some CDN services, including Azure CDN, support token-based authentication to protect CDN content. For more information, see [Securing Azure Content Delivery Network assets with token authentication](#).
- **Client security.** Clients might connect from an environment that does not allow access to resources on the CDN. This could be a security-constrained environment that limits access to only a set of known sources, or one that prevents loading of resources from anything other than the page origin. A fallback implementation is required to handle these cases.
- **Resilience.** The CDN is a potential single point of failure for an application.

Scenarios where a CDN may be less useful include:

- If the content has a low hit rate, it might be accessed only few times while it is valid (determined by its time-to-live setting).
- If the data is private, such as for large enterprises or supply chain ecosystems.

General guidelines and good practices

Using a CDN is a good way to minimize the load on your application, and maximize availability and performance. Consider adopting this strategy for all of the appropriate content and resources your application uses. Consider the points in the following sections when designing your strategy to use a CDN.

Deployment

Static content may need to be provisioned and deployed independently from the application if you do not include it in the application deployment package or process. Consider how this will affect the versioning approach you use to manage both the application components and the static resource content.

Consider using bundling and minification techniques to reduce load times for clients. Bundling combines multiple files into a single file. Minification removes unnecessary characters from scripts and CSS files without altering functionality.

If you need to deploy the content to an additional location, this will be an extra step in the deployment process. If the application updates the content for the CDN, perhaps at regular intervals or in response to an event, it must store the updated content in any additional locations as well as the endpoint for the CDN.

Consider how you will handle local development and testing when some static content is expected to be served from a CDN. For example, you could predeploy the content to the CDN as part of your build script. Alternatively, use compile directives or flags to control how the application loads the resources. For example, in debug mode, the application could load static resources from a local folder. In release mode, the application would use the CDN.

Consider the options for file compression, such as gzip (GNU zip). Compression may be performed on the origin server by the web application hosting or directly on the edge servers by the CDN. For more information, see [Improve performance by compressing files in Azure CDN](#).

Routing and versioning

You may need to use different CDN instances at various times. For example, when you deploy a new version of the application you may want to use a new CDN and retain the old CDN (holding content in an older format) for previous versions. If you use Azure blob storage as the content origin, you can create a separate storage account or a separate container and point the CDN endpoint to it.

Do not use the query string to denote different versions of the application in links to resources on the CDN because, when retrieving content from Azure blob storage, the query string is part of the resource name (the blob name). This approach can also affect how the client caches resources.

Deploying new versions of static content when you update an application can be a challenge if the previous resources are cached on the CDN. For more information, see the section on cache control, below.

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

Cache control

Consider how to manage caching within the system. For example, in Azure CDN, you can set global caching rules, and then set custom caching for particular origin endpoints. You can also control how caching is performed in a CDN by sending cache-directive headers at the origin.

For more information, see [How caching works](#).

To prevent objects from being available on the CDN, you can delete them from the origin, remove or delete the CDN endpoint, or in the case of blob storage, make the container or blob private. However, items are not removed from the CDN until the time-to-live expires. You can also manually purge a CDN endpoint.

Security

The CDN can deliver content over HTTPS (SSL), by using the certificate provided by the CDN, as well as over standard HTTP. To avoid browser warnings about mixed content, you might need to use HTTPS to request static content that is displayed in pages loaded through HTTPS.

If you deliver static assets such as font files by using the CDN, you might encounter same-origin policy issues if you use an *XMLHttpRequest* call to request these resources from a different domain. Many web browsers prevent cross-origin resource sharing (CORS) unless the web server is configured to set the appropriate response headers. You can configure the CDN to support CORS by using one of the following methods:

- Configure the CDN to add CORS headers to the responses. For more information, see [Using Azure CDN with CORS](#).
- If the origin is Azure blob storage, add CORS rules to the storage endpoint. For more information, see [Cross-Origin Resource Sharing \(CORS\) Support for the Azure Storage Services](#).
- Configure the application to set the CORS headers. For example, see [Enabling Cross-Origin Requests \(CORS\)](#) in the ASP.NET Core documentation.

CDN fallback

Consider how your application will cope with a failure or temporary unavailability of the CDN. Client applications may be able to use copies of the resources that were cached locally (on the client) during previous requests, or you can include code that detects failure and instead requests resources from the origin (the application folder or Azure blob container that holds the resources) if the CDN is unavailable.

Data partitioning guidance

Azure Blob Storage

In many large-scale solutions, data is divided into *partitions* that can be managed and accessed separately. Partitioning can improve scalability, reduce contention, and optimize performance. It can also provide a mechanism for dividing data by usage pattern. For example, you can archive older data in cheaper data storage.

However, the partitioning strategy must be chosen carefully to maximize the benefits while minimizing adverse effects.

ⓘ Note

In this article, the term *partitioning* means the process of physically dividing data into separate data stores. It is not the same as SQL Server table partitioning.

Why partition data?

- **Improve scalability.** When you scale up a single database system, it will eventually reach a physical hardware limit. If you divide data across multiple partitions, each hosted on a separate server, you can scale out the system almost indefinitely.
- **Improve performance.** Data access operations on each partition take place over a smaller volume of data. Correctly done, partitioning can make your system more efficient. Operations that affect more than one partition can run in parallel.
- **Improve security.** In some cases, you can separate sensitive and nonsensitive data into different partitions and apply different security controls to the sensitive data.
- **Provide operational flexibility.** Partitioning offers many opportunities for fine-tuning operations, maximizing administrative efficiency, and minimizing cost. For example, you can define different strategies for management, monitoring, backup and restore, and other administrative tasks based on the importance of the data in each partition.
- **Match the data store to the pattern of use.** Partitioning allows each partition to be deployed on a different type of data store, based on cost and the built-in features that data store offers. For example, large binary data can be stored in blob storage,

while more structured data can be held in a document database. See [Choose the right data store](#).

- **Improve availability.** Separating data across multiple servers avoids a single point of failure. If one instance fails, only the data in that partition is unavailable. Operations on other partitions can continue. For managed PaaS data stores, this consideration is less relevant, because these services are designed with built-in redundancy.

Designing partitions

There are three typical strategies for partitioning data:

- **Horizontal partitioning** (often called *sharding*). In this strategy, each partition is a separate data store, but all partitions have the same schema. Each partition is known as a *shard* and holds a specific subset of the data, such as all the orders for a specific set of customers.
- **Vertical partitioning.** In this strategy, each partition holds a subset of the fields for items in the data store. The fields are divided according to their pattern of use. For example, frequently accessed fields might be placed in one vertical partition and less frequently accessed fields in another.
- **Functional partitioning.** In this strategy, data is aggregated according to how it is used by each bounded context in the system. For example, an e-commerce system might store invoice data in one partition and product inventory data in another.

These strategies can be combined, and we recommend that you consider them all when you design a partitioning scheme. For example, you might divide data into shards and then use vertical partitioning to further subdivide the data in each shard.

Horizontal partitioning (sharding)

Figure 1 shows horizontal partitioning or sharding. In this example, product inventory data is divided into shards based on the product key. Each shard holds the data for a contiguous range of shard keys (A-G and H-Z), organized alphabetically. Sharding spreads the load over more computers, which reduces contention and improves performance.

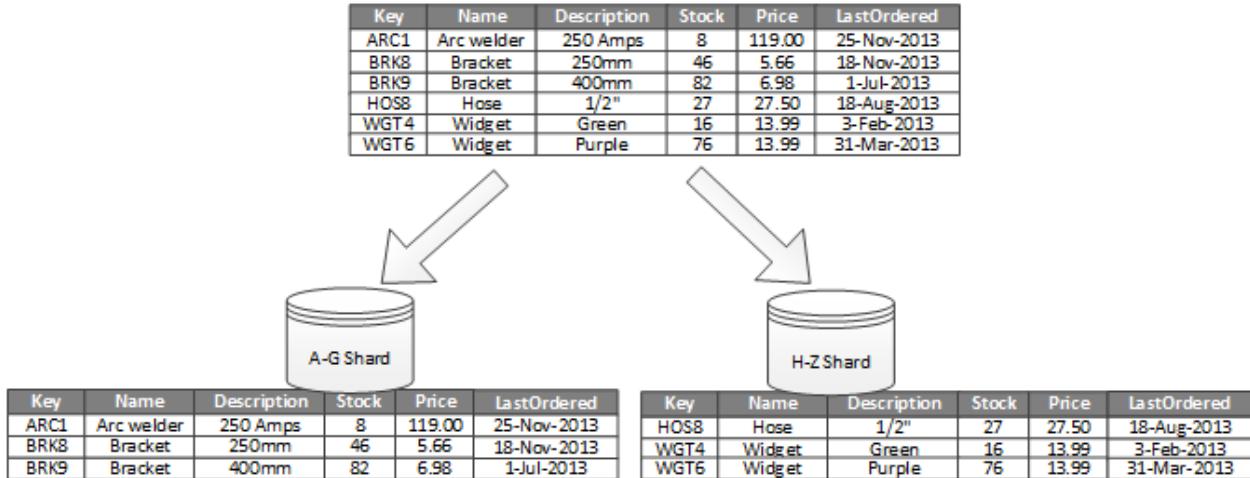


Figure 1 - Horizontally partitioning (sharding) data based on a partition key.

The most important factor is the choice of a sharding key. It can be difficult to change the key after the system is in operation. The key must ensure that data is partitioned to spread the workload as evenly as possible across the shards.

The shards don't have to be the same size. It's more important to balance the number of requests. Some shards might be very large, but each item has a low number of access operations. Other shards might be smaller, but each item is accessed much more frequently. It's also important to ensure that a single shard does not exceed the scale limits (in terms of capacity and processing resources) of the data store.

Avoid creating "hot" partitions that can affect performance and availability. For example, using the first letter of a customer's name causes an unbalanced distribution, because some letters are more common. Instead, use a hash of a customer identifier to distribute data more evenly across partitions.

Choose a sharding key that minimizes any future requirements to split large shards, coalesce small shards into larger partitions, or change the schema. These operations can be very time consuming, and might require taking one or more shards offline while they are performed.

If shards are replicated, it might be possible to keep some of the replicas online while others are split, merged, or reconfigured. However, the system might need to limit the operations that can be performed during the reconfiguration. For example, the data in the replicas might be marked as read-only to prevent data inconsistencies.

For more information about horizontal partitioning, see [sharding pattern](#).

Vertical partitioning

The most common use for vertical partitioning is to reduce the I/O and performance costs associated with fetching items that are frequently accessed. Figure 2 shows an example of vertical partitioning. In this example, different properties of an item are stored in different partitions. One partition holds data that is accessed more frequently, including product name, description, and price. Another partition holds inventory data: the stock count and last-ordered date.

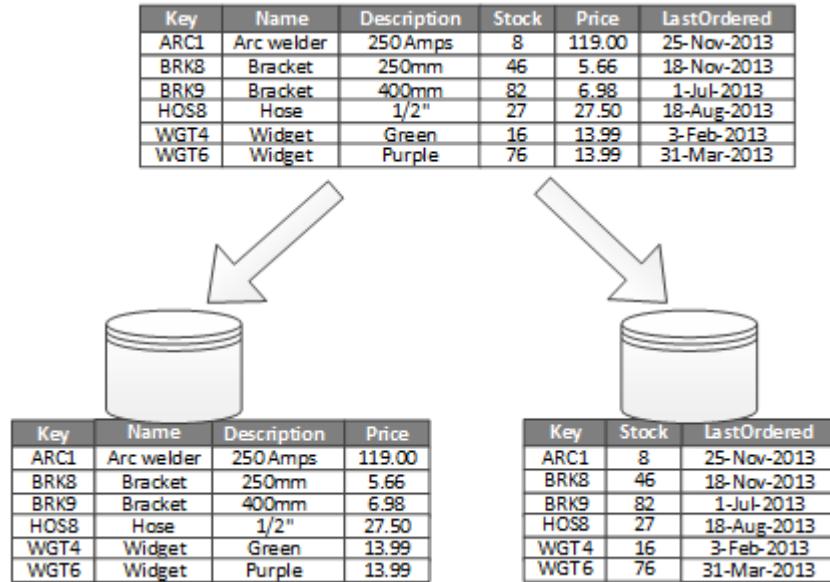


Figure 2 - Vertically partitioning data by its pattern of use.

In this example, the application regularly queries the product name, description, and price when displaying the product details to customers. Stock count and last- ordered date are held in a separate partition because these two items are commonly used together.

Other advantages of vertical partitioning:

- Relatively slow-moving data (product name, description, and price) can be separated from the more dynamic data (stock level and last ordered date). Slow moving data is a good candidate for an application to cache in memory.
- Sensitive data can be stored in a separate partition with additional security controls.
- Vertical partitioning can reduce the amount of concurrent access that's needed.

Vertical partitioning operates at the entity level within a data store, partially normalizing an entity to break it down from a *wide* item to a set of *narrow* items. It is ideally suited for column-oriented data stores such as HBase and Cassandra. If the data in a collection of columns is unlikely to change, you can also consider using column stores in SQL Server.

Functional partitioning

When it's possible to identify a bounded context for each distinct business area in an application, functional partitioning is a way to improve isolation and data access performance. Another common use for functional partitioning is to separate read-write data from read-only data. Figure 3 shows an overview of functional partitioning where inventory data is separated from customer data.

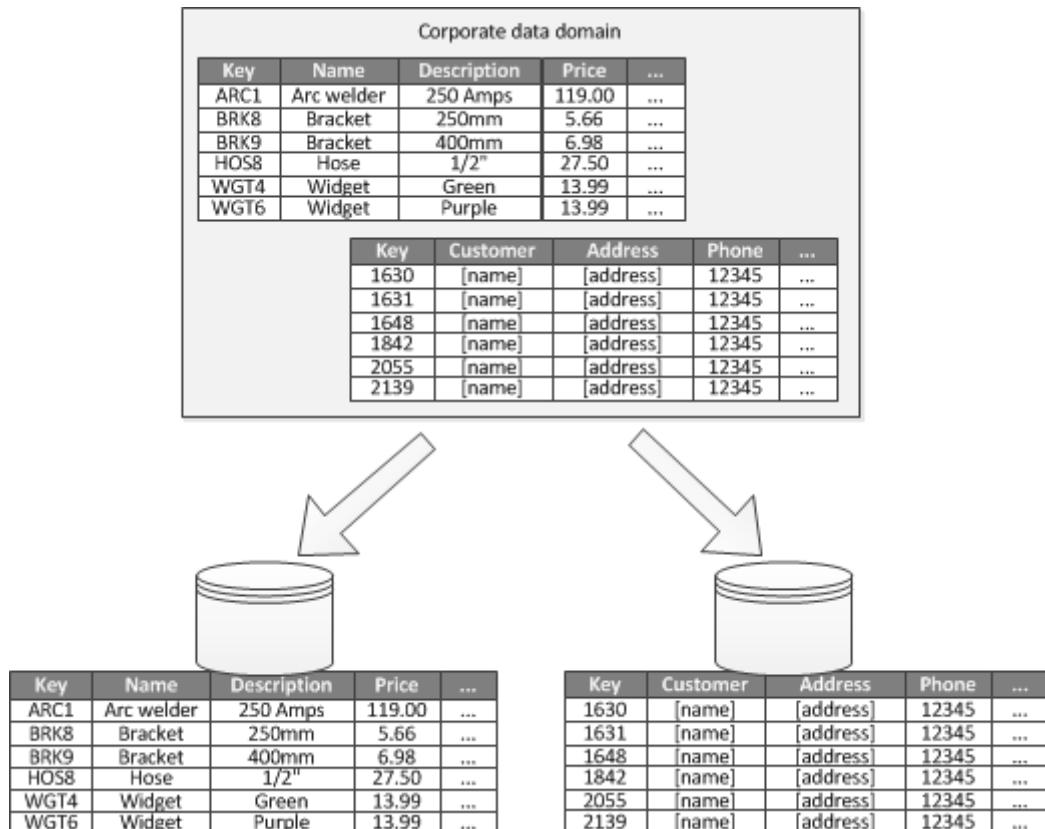


Figure 3 - Functionally partitioning data by bounded context or subdomain.

This partitioning strategy can help reduce data access contention across different parts of a system.

Designing partitions for scalability

It's vital to consider size and workload for each partition and balance them so that data is distributed to achieve maximum scalability. However, you must also partition the data so that it does not exceed the scaling limits of a single partition store.

Follow these steps when designing partitions for scalability:

1. Analyze the application to understand the data access patterns, such as the size of the result set returned by each query, the frequency of access, the inherent latency, and the server-side compute processing requirements. In many cases, a few major entities will demand most of the processing resources.

2. Use this analysis to determine the current and future scalability targets, such as data size and workload. Then distribute the data across the partitions to meet the scalability target. For horizontal partitioning, choosing the right shard key is important to make sure distribution is even. For more information, see the [sharding pattern](#).
3. Make sure each partition has enough resources to handle the scalability requirements, in terms of data size and throughput. Depending on the data store, there might be a limit on the amount of storage space, processing power, or network bandwidth per partition. If the requirements are likely to exceed these limits, you might need to refine your partitioning strategy or split data out further, possibly combining two or more strategies.
4. Monitor the system to verify that data is distributed as expected and that the partitions can handle the load. Actual usage does not always match what an analysis predicts. If so, it might be possible to rebalance the partitions, or else redesign some parts of the system to gain the required balance.

Some cloud environments allocate resources in terms of infrastructure boundaries. Ensure that the limits of your selected boundary provide enough room for any anticipated growth in the volume of data, in terms of data storage, processing power, and bandwidth.

For example, if you use Azure table storage, there is a limit to the volume of requests that can be handled by a single partition in a particular period of time. (For more information, see [Azure storage scalability and performance targets](#).) A busy shard might require more resources than a single partition can handle. If so, the shard might need to be repartitioned to spread the load. If the total size or throughput of these tables exceeds the capacity of a storage account, you might need to create additional storage accounts and spread the tables across these accounts.

Designing partitions for query performance

Query performance can often be boosted by using smaller data sets and by running parallel queries. Each partition should contain a small proportion of the entire data set. This reduction in volume can improve the performance of queries. However, partitioning is not an alternative for designing and configuring a database appropriately. For example, make sure that you have the necessary indexes in place.

Follow these steps when designing partitions for query performance:

1. Examine the application requirements and performance:

- Use business requirements to determine the critical queries that must always perform quickly.
- Monitor the system to identify any queries that perform slowly.
- Find which queries are performed most frequently. Even if a single query has a minimal cost, the cumulative resource consumption could be significant.

2. Partition the data that is causing slow performance:

- Limit the size of each partition so that the query response time is within target.
- If you use horizontal partitioning, design the shard key so that the application can easily select the right partition. This prevents the query from having to scan through every partition.
- Consider the location of a partition. If possible, try to keep data in partitions that are geographically close to the applications and users that access it.

3. If an entity has throughput and query performance requirements, use functional partitioning based on that entity. If this still doesn't satisfy the requirements, apply horizontal partitioning as well. In most cases, a single partitioning strategy will suffice, but in some cases it is more efficient to combine both strategies.

4. Consider running queries in parallel across partitions to improve performance.

Designing partitions for availability

Partitioning data can improve the availability of applications by ensuring that the entire dataset does not constitute a single point of failure and that individual subsets of the dataset can be managed independently.

Consider the following factors that affect availability:

How critical the data is to business operations. Identify which data is critical business information, such as transactions, and which data is less critical operational data, such as log files.

- Consider storing critical data in highly available partitions with an appropriate backup plan.
- Establish separate management and monitoring procedures for the different datasets.
- Place data that has the same level of criticality in the same partition so that it can be backed up together at an appropriate frequency. For example, partitions that

hold transaction data might need to be backed up more frequently than partitions that hold logging or trace information.

How individual partitions can be managed. Designing partitions to support independent management and maintenance provides several advantages. For example:

- If a partition fails, it can be recovered independently without applications that access data in other partitions.
- Partitioning data by geographical area allows scheduled maintenance tasks to occur at off-peak hours for each location. Ensure that partitions are not too large to prevent any planned maintenance from being completed during this period.

Whether to replicate critical data across partitions. This strategy can improve availability and performance, but can also introduce consistency issues. It takes time to synchronize changes with every replica. During this period, different partitions will contain different data values.

Application design considerations

Partitioning adds complexity to the design and development of your system. Consider partitioning as a fundamental part of system design even if the system initially only contains a single partition. If you address partitioning as an afterthought, it will be more challenging because you already have a live system to maintain:

- Data access logic will need to be modified.
- Large quantities of existing data might need to be migrated, to distribute it across partitions.
- Users expect to be able to continue using the system during the migration.

In some cases, partitioning is not considered important because the initial dataset is small and can be easily handled by a single server. This might be true for some workloads, but many commercial systems need to expand as the number of users increases.

Moreover, it's not only large data stores that benefit from partitioning. For example, a small data store might be heavily accessed by hundreds of concurrent clients.

Partitioning the data in this situation can help to reduce contention and improve throughput.

Consider the following points when you design a data partitioning scheme:

Minimize cross-partition data access operations. Where possible, keep data for the most common database operations together in each partition to minimize cross-

partition data access operations. Querying across partitions can be more time-consuming than querying within a single partition, but optimizing partitions for one set of queries might adversely affect other sets of queries. If you must query across partitions, minimize query time by running parallel queries and aggregating the results within the application. (This approach might not be possible in some cases, such as when the result from one query is used in the next query.)

Consider replicating static reference data. If queries use relatively static reference data, such as postal code tables or product lists, consider replicating this data in all of the partitions to reduce separate lookup operations in different partitions. This approach can also reduce the likelihood of the reference data becoming a "hot" dataset, with heavy traffic from across the entire system. However, there is an additional cost associated with synchronizing any changes to the reference data.

Minimize cross-partition joins. Where possible, minimize requirements for referential integrity across vertical and functional partitions. In these schemes, the application is responsible for maintaining referential integrity across partitions. Queries that join data across multiple partitions are inefficient because the application typically needs to perform consecutive queries based on a key and then a foreign key. Instead, consider replicating or de-normalizing the relevant data. If cross-partition joins are necessary, run parallel queries over the partitions and join the data within the application.

Embrace eventual consistency. Evaluate whether strong consistency is actually a requirement. A common approach in distributed systems is to implement eventual consistency. The data in each partition is updated separately, and the application logic ensures that the updates are all completed successfully. It also handles the inconsistencies that can arise from querying data while an eventually consistent operation is running.

Consider how queries locate the correct partition. If a query must scan all partitions to locate the required data, there is a significant impact on performance, even when multiple parallel queries are running. With vertical and functional partitioning, queries can naturally specify the partition. Horizontal partitioning, on the other hand, can make locating an item difficult, because every shard has the same schema. A typical solution to maintain a map that is used to look up the shard location for specific items. This map can be implemented in the sharding logic of the application, or maintained by the data store if it supports transparent sharding.

Consider periodically rebalancing shards. With horizontal partitioning, rebalancing shards can help distribute the data evenly by size and by workload to minimize hotspots, maximize query performance, and work around physical storage limitations. However, this is a complex task that often requires the use of a custom tool or process.

Replicate partitions. If you replicate each partition, it provides additional protection against failure. If a single replica fails, queries can be directed toward a working copy.

If you reach the physical limits of a partitioning strategy, you might need to extend the scalability to a different level. For example, if partitioning is at the database level, you might need to locate or replicate partitions in multiple databases. If partitioning is already at the database level, and physical limitations are an issue, it might mean that you need to locate or replicate partitions in multiple hosting accounts.

Avoid transactions that access data in multiple partitions. Some data stores implement transactional consistency and integrity for operations that modify data, but only when the data is located in a single partition. If you need transactional support across multiple partitions, you will probably need to implement this as part of your application logic because most partitioning systems do not provide native support.

All data stores require some operational management and monitoring activity. The tasks can range from loading data, backing up and restoring data, reorganizing data, and ensuring that the system is performing correctly and efficiently.

Consider the following factors that affect operational management:

- **How to implement appropriate management and operational tasks when the data is partitioned.** These tasks might include backup and restore, archiving data, monitoring the system, and other administrative tasks. For example, maintaining logical consistency during backup and restore operations can be a challenge.
- **How to load the data into multiple partitions and add new data that's arriving from other sources.** Some tools and utilities might not support sharded data operations such as loading data into the correct partition.
- **How to archive and delete the data on a regular basis.** To prevent the excessive growth of partitions, you need to archive and delete data on a regular basis (such as monthly). It might be necessary to transform the data to match a different archive schema.
- **How to locate data integrity issues.** Consider running a periodic process to locate any data integrity issues, such as data in one partition that references missing information in another. The process can either attempt to fix these issues automatically or generate a report for manual review.

Rebalancing partitions

As a system matures, you might have to adjust the partitioning scheme. For example, individual partitions might start getting a disproportionate volume of traffic and become hot, leading to excessive contention. Or you might have underestimated the volume of data in some partitions, causing some partitions to approach capacity limits.

Some data stores, such as Azure Cosmos DB, can automatically rebalance partitions. In other cases, rebalancing is an administrative task that consists of two stages:

1. Determine a new partitioning strategy.
 - Which partitions need to be split (or possibly combined)?
 - What is the new partition key?
2. Migrate data from the old partitioning scheme to the new set of partitions.

Depending on the data store, you might be able to migrate data between partitions while they are in use. This is called *online migration*. If that's not possible, you might need to make partitions unavailable while the data is relocated (*offline migration*).

Offline migration

Offline migration is typically simpler because it reduces the chances of contention occurring. Conceptually, offline migration works as follows:

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

Optionally, you can mark a partition as read-only in step 1, so that applications can still read the data while it is being moved.

Online migration

Online migration is more complex to perform but less disruptive. The process is similar to offline migration, except the original partition is not marked offline. Depending on the granularity of the migration process (for example, item by item versus shard by shard), the data access code in the client applications might have to handle reading and writing data that's held in two locations, the original partition and the new partition.

Next steps

- Learn about partitioning strategies for specific Azure services. See [Data partitioning strategies](#).
- [Azure storage scalability and performance targets](#)

Related resources

- [Choose the right data store](#)

The following design patterns might be relevant to your scenario:

- The [sharding pattern](#) describes some common strategies for sharding data.
- The [index table pattern](#) shows how to create secondary indexes over data. An application can quickly retrieve data with this approach, by using queries that do not reference the primary key of a collection.
- The [materialized view pattern](#) describes how to generate prepopulated views that summarize data to support fast query operations. This approach can be useful in a partitioned data store if the partitions that contain the data being summarized are distributed across multiple sites.

Data partitioning strategies

Azure Table Storage

This article describes some strategies for partitioning data in various Azure data stores. For general guidance about when to partition data and best practices, see [Data partitioning](#).

Partitioning Azure SQL Database

A single SQL database has a limit to the volume of data that it can contain. Throughput is constrained by architectural factors and the number of concurrent connections that it supports.

[Elastic pools](#) support horizontal scaling for a SQL database. Using elastic pools, you can partition your data into shards that are spread across multiple SQL databases. You can also add or remove shards as the volume of data that you need to handle grows and shrinks. Elastic pools can also help reduce contention by distributing the load across databases.

Each shard is implemented as a SQL database. A shard can hold more than one dataset (called a *shardlet*). Each database maintains metadata that describes the shardlets that it contains. A shardlet can be a single data item, or a group of items that share the same shardlet key. For example, in a multitenant application, the shardlet key can be the tenant ID, and all data for a tenant can be held in the same shardlet.

Client applications are responsible for associating a dataset with a shardlet key. A separate SQL database acts as a global shard map manager. This database has a list of all the shards and shardlets in the system. The application connects to the shard map manager database to obtain a copy of the shard map. It caches the shard map locally, and uses the map to route data requests to the appropriate shard. This functionality is hidden behind a series of APIs that are contained in the [Elastic Database client library](#), which is available for Java and .NET.

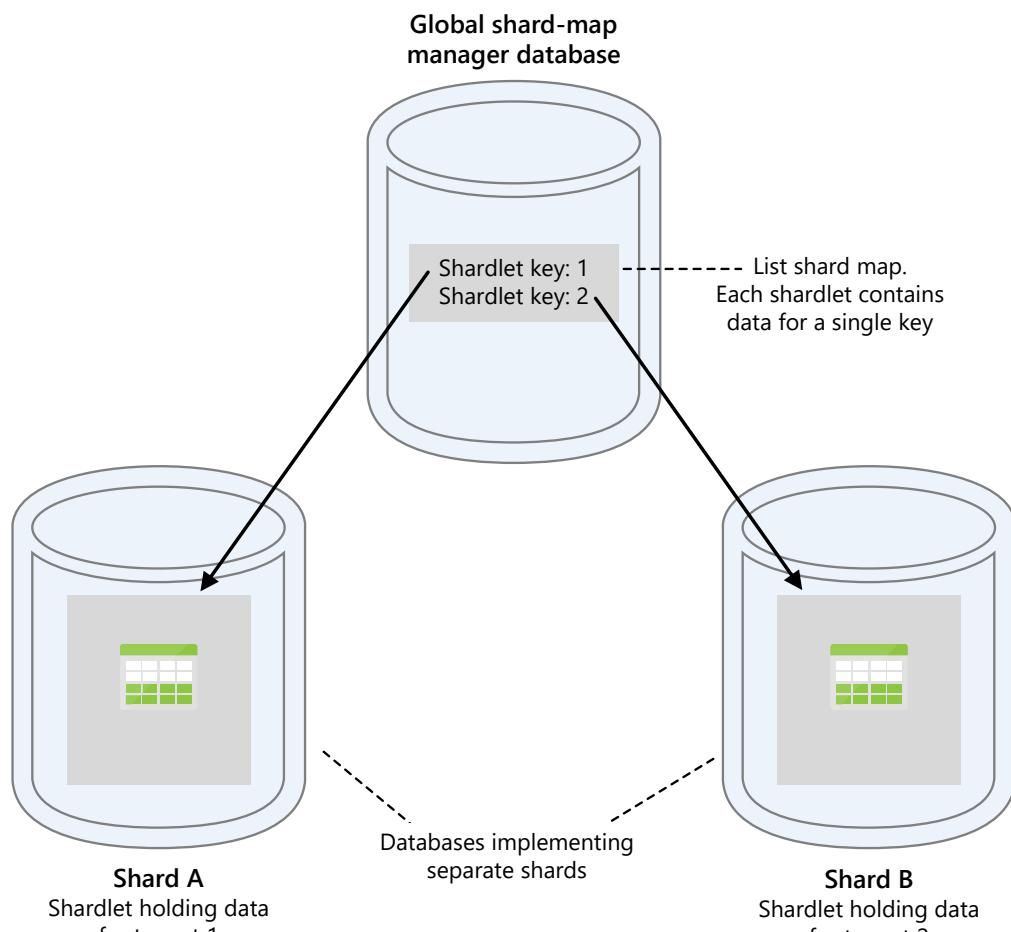
For more information about elastic pools, see [Scaling out with Azure SQL Database](#).

To reduce latency and improve availability, you can replicate the global shard map manager database. With the Premium pricing tiers, you can configure active geo-replication to continuously copy data to databases in different regions.

Alternatively, use [Azure SQL Data Sync](#) or [Azure Data Factory](#) to replicate the shard map manager database across regions. This form of replication runs periodically and is more suitable if the shard map changes infrequently, and does not require Premium tier.

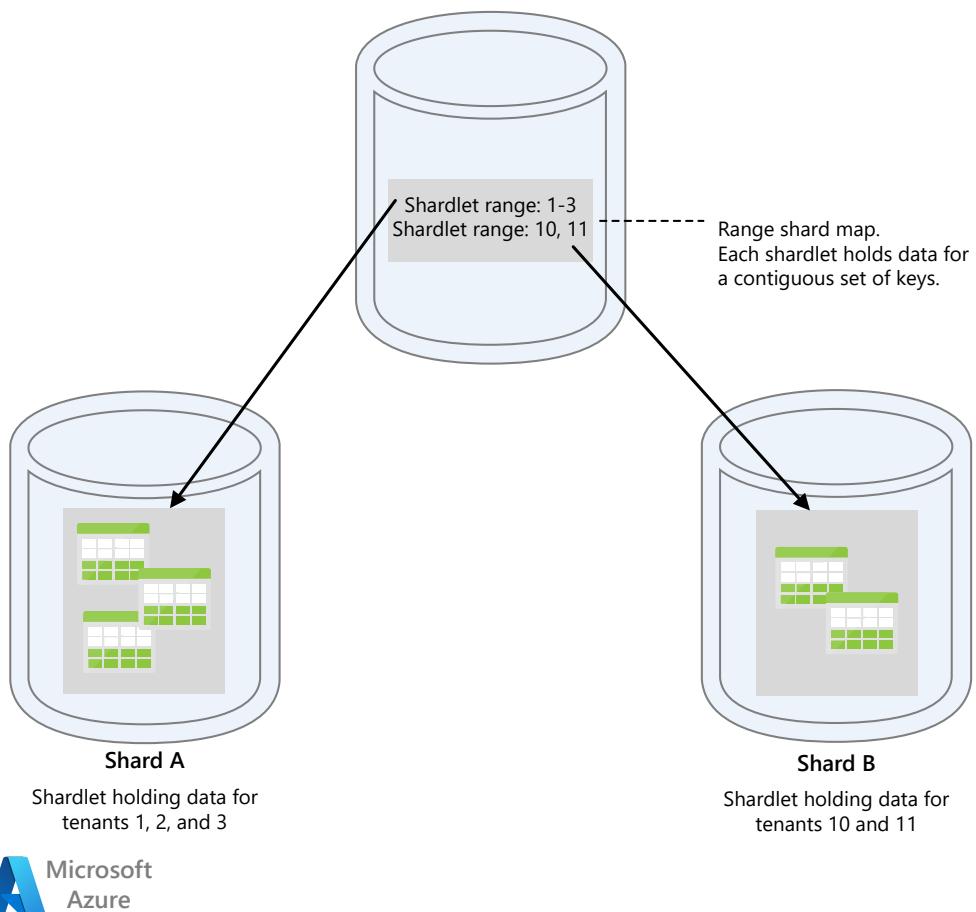
Elastic Database provides two schemes for mapping data to shardlets and storing them in shards:

- A **list shard map** associates a single key to a shardlet. For example, in a multitenant system, the data for each tenant can be associated with a unique key and stored in its own shardlet. To guarantee isolation, each shardlet can be held within its own shard.



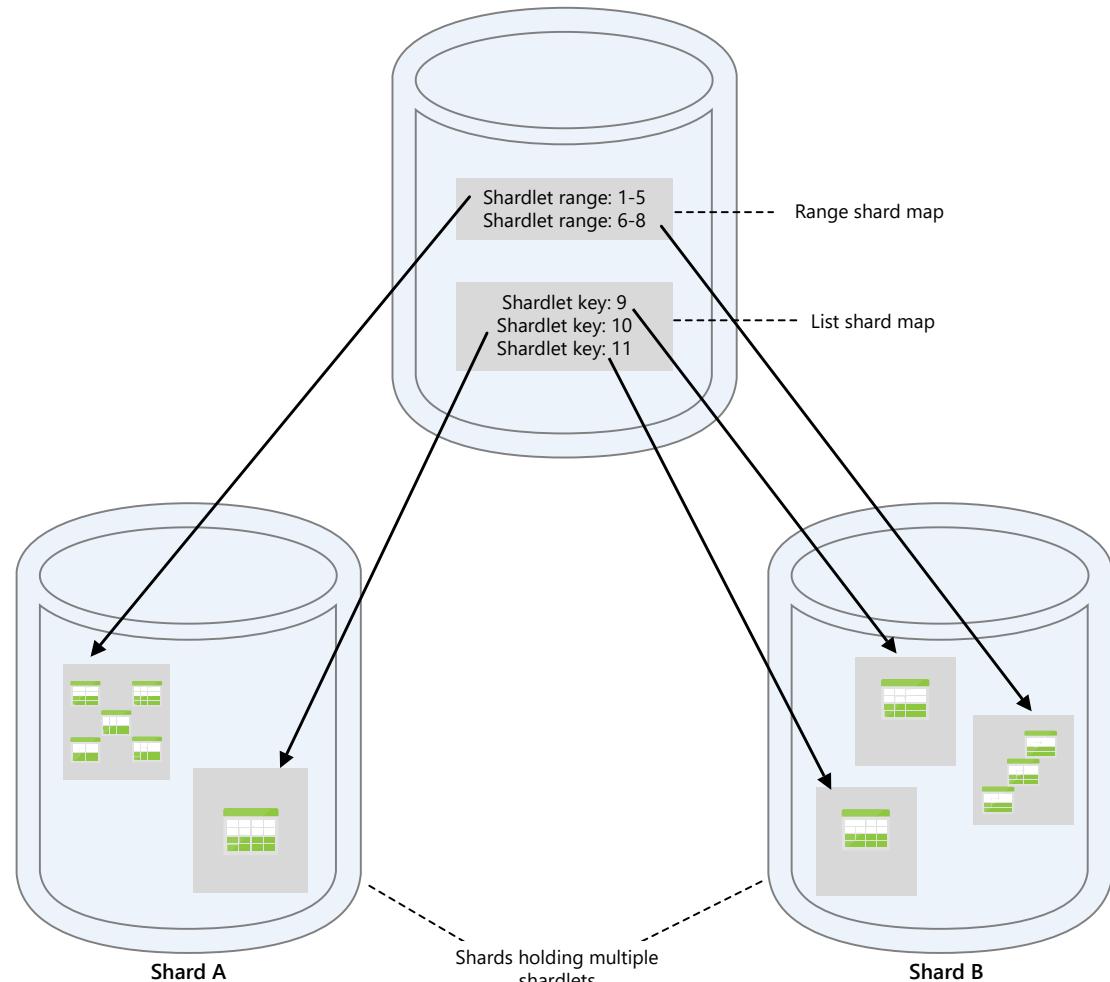
Download a [Visio file](#) of this diagram.

- A **range shard map** associates a set of contiguous key values to a shardlet. For example, you can group the data for a set of tenants (each with their own key) within the same shardlet. This scheme is less expensive than the first, because tenants share data storage, but has less isolation.



Download a [Visio file](#) of this diagram

A single shard can contain the data for several shardlets. For example, you can use list shardlets to store data for different non-contiguous tenants in the same shard. You can also mix range shardlets and list shardlets in the same shard, although they will be addressed through different maps. The following diagram shows this approach:



[Download a Visio file](#) of this diagram.

Elastic pools make it possible to add and remove shards as the volume of data shrinks and grows. Client applications can create and delete shards dynamically, and transparently update the shard map manager. However, removing a shard is a destructive operation that also requires deleting all the data in that shard.

If an application needs to split a shard into two separate shards or combine shards, use the [split-merge tool](#). This tool runs as an Azure web service, and migrates data safely between shards.

The partitioning scheme can significantly affect the performance of your system. It can also affect the rate at which shards have to be added or removed, or that data must be repartitioned across shards. Consider the following points:

- Group data that is used together in the same shard, and avoid operations that access data from multiple shards. A shard is a SQL database in its own right, and cross-database joins must be performed on the client side.

Although SQL Database does not support cross-database joins, you can use the Elastic Database tools to perform [multi-shard queries](#). A multi-shard query sends individual queries to each database and merges the results.

- Don't design a system that has dependencies between shards. Referential integrity constraints, triggers, and stored procedures in one database cannot reference objects in another.
- If you have reference data that is frequently used by queries, consider replicating this data across shards. This approach can remove the need to join data across databases. Ideally, such data should be static or slow-moving, to minimize the replication effort and reduce the chances of it becoming stale.
- Shardlets that belong to the same shard map should have the same schema. This rule is not enforced by SQL Database, but data management and querying becomes very complex if each shardlet has a different schema. Instead, create separate shard maps for each schema. Remember that data belonging to different shardlets can be stored in the same shard.
- Transactional operations are only supported for data within a shard, and not across shards. Transactions can span shardlets as long as they are part of the same shard. Therefore, if your business logic needs to perform transactions, either store the data in the same shard or implement eventual consistency.
- Place shards close to the users that access the data in those shards. This strategy helps reduce latency.
- Avoid having a mixture of highly active and relatively inactive shards. Try to spread the load evenly across shards. This might require hashing the sharding keys. If you are geo-locating shards, make sure that the hashed keys map to shardlets held in shards stored close to the users that access that data.

Partitioning Azure Table storage

Azure Table storage is a key-value store that's designed around partitioning. All entities are stored in a partition, and partitions are managed internally by Azure Table storage. Each entity stored in a table must provide a two-part key that includes:

- **The partition key.** This is a string value that determines the partition where Azure Table storage will place the entity. All entities with the same partition key are stored in the same partition.

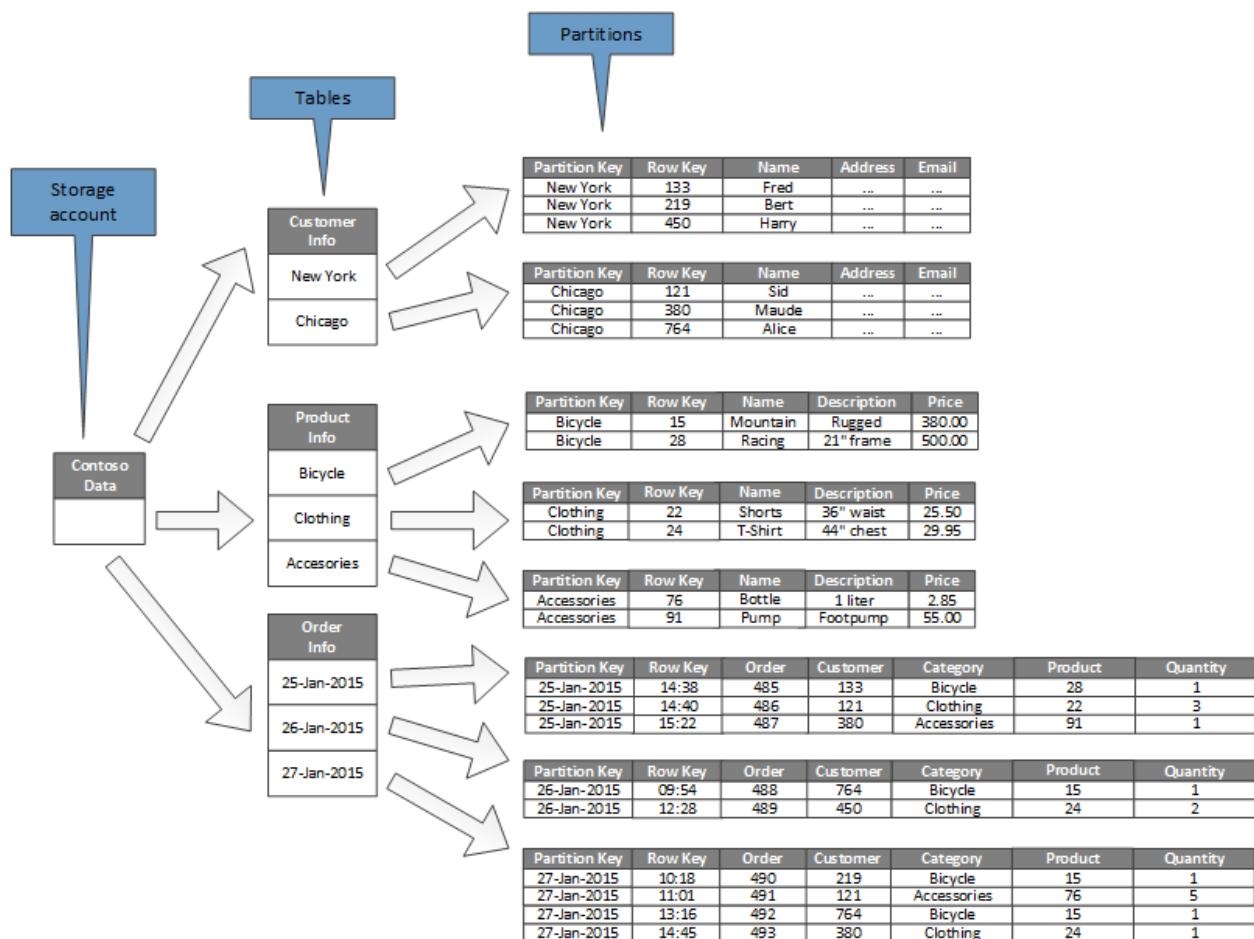
- **The row key.** This is a string value that identifies the entity within the partition. All entities within a partition are sorted lexically, in ascending order, by this key. The partition key/row key combination must be unique for each entity and cannot exceed 1 KB in length.

If an entity is added to a table with a previously unused partition key, Azure Table storage creates a new partition for this entity. Other entities with the same partition key will be stored in the same partition.

This mechanism effectively implements an automatic scale-out strategy. Each partition is stored on the same server in an Azure datacenter to help ensure that queries that retrieve data from a single partition run quickly.

Microsoft has published [scalability targets](#) for Azure Storage. If your system is likely to exceed these limits, consider splitting entities into multiple tables. Use vertical partitioning to divide the fields into the groups that are most likely to be accessed together.

The following diagram shows the logical structure of an example storage account. The storage account contains three tables: Customer Info, Product Info, and Order Info.



Each table has multiple partitions.

- In the Customer Info table, the data is partitioned according to the city where the customer is located. The row key contains the customer ID.
- In the Product Info table, products are partitioned by product category, and the row key contains the product number.
- In the Order Info table, the orders are partitioned by order date, and the row key specifies the time the order was received. All data is ordered by the row key in each partition.

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

- Select a partition key and row key by how the data is accessed. Choose a partition key/row key combination that supports the majority of your queries. The most efficient queries retrieve data by specifying the partition key and the row key. Queries that specify a partition key and a range of row keys can be completed by scanning a single partition. This is relatively fast because the data is held in row key order. If queries don't specify which partition to scan, every partition must be scanned.
- If an entity has one natural key, then use it as the partition key and specify an empty string as the row key. If an entity has a composite key consisting of two properties, select the slowest changing property as the partition key and the other as the row key. If an entity has more than two key properties, use a concatenation of properties to provide the partition and row keys.
- If you regularly perform queries that look up data by using fields other than the partition and row keys, consider implementing the [Index Table pattern](#), or consider using a different data store that supports indexing, such as Azure Cosmos DB.
- If you generate partition keys by using a monotonic sequence (such as "0001", "0002", "0003") and each partition only contains a limited amount of data, Azure Table storage can physically group these partitions together on the same server. Azure Storage assumes that the application is most likely to perform queries across a contiguous range of partitions (range queries) and is optimized for this case. However, this approach can lead to hotspots, because all insertions of new entities are likely to be concentrated at one end the contiguous range. It can also reduce scalability. To spread the load more evenly, consider hashing the partition key.
- Azure Table storage supports transactional operations for entities that belong to the same partition. An application can perform multiple insert, update, delete, replace, or merge operations as an atomic unit, as long as the transaction doesn't include more than 100 entities and the payload of the request doesn't exceed 4 MB. Operations that span multiple partitions are not transactional, and might

require you to implement eventual consistency. For more information about table storage and transactions, see [Performing entity group transactions](#).

- Consider the granularity of the partition key:
 - Using the same partition key for every entity results in a single partition that's held on one server. This prevents the partition from scaling out and focuses the load on a single server. As a result, this approach is only suitable for storing a small number of entities. However, it does ensure that all entities can participate in entity group transactions.
 - Using a unique partition key for every entity causes the table storage service to create a separate partition for each entity, possibly resulting in a large number of small partitions. This approach is more scalable than using a single partition key, but entity group transactions are not possible. Also, queries that fetch more than one entity might involve reading from more than one server. However, if the application performs range queries, then using a monotonic sequence for the partition keys might help to optimize these queries.
 - Sharing the partition key across a subset of entities makes it possible to group related entities in the same partition. Operations that involve related entities can be performed by using entity group transactions, and queries that fetch a set of related entities can be satisfied by accessing a single server.

For more information, see [Azure storage table design guide](#) and [Scalable partitioning strategy](#).

Partitioning Azure Blob Storage

Azure Blob Storage makes it possible to hold large binary objects. Use block blobs in scenarios when you need to upload or download large volumes of data quickly. Use page blobs for applications that require random rather than serial access to parts of the data.

Each blob (either block or page) is held in a container in an Azure storage account. You can use containers to group related blobs that have the same security requirements. This grouping is logical rather than physical. Inside a container, each blob has a unique name.

The partition key for a blob is account name + container name + blob name. The partition key is used to partition data into ranges and these ranges are load-balanced across the system. Blobs can be distributed across many servers in order to scale out access to them, but a single blob can only be served by a single server.

If your naming scheme uses timestamps or numerical identifiers, it can lead to excessive traffic going to one partition, limiting the system from effectively load balancing. For instance, if you have daily operations that use a blob object with a timestamp such as *yyyy-mm-dd*, all the traffic for that operation would go to a single partition server. Instead, consider prefixing the name with a three-digit hash. For more information, see [Partition Naming Convention](#).

The actions of writing a single block or page are atomic, but operations that span blocks, pages, or blobs are not. If you need to ensure consistency when performing write operations across blocks, pages, and blobs, take out a write lock by using a blob lease.

Partitioning Azure storage queues

Azure storage queues enable you to implement asynchronous messaging between processes. An Azure storage account can contain any number of queues, and each queue can contain any number of messages. The only limitation is the space that's available in the storage account. The maximum size of an individual message is 64 KB. If you require messages bigger than this, then consider using Azure Service Bus queues instead.

Each storage queue has a unique name within the storage account that contains it. Azure partitions queues based on the name. All messages for the same queue are stored in the same partition, which is controlled by a single server. Different queues can be managed by different servers to help balance the load. The allocation of queues to servers is transparent to applications and users.

In a large-scale application, don't use the same storage queue for all instances of the application because this approach might cause the server that's hosting the queue to become a hot spot. Instead, use different queues for different functional areas of the application. Azure storage queues do not support transactions, so directing messages to different queues should have little effect on messaging consistency.

An Azure storage queue can handle up to 2,000 messages per second. If you need to process messages at a greater rate than this, consider creating multiple queues. For example, in a global application, create separate storage queues in separate storage accounts to handle application instances that are running in each region.

Partitioning Azure Service Bus

Azure Service Bus uses a message broker to handle messages that are sent to a Service Bus queue or topic. By default, all messages that are sent to a queue or topic are handled by the same message broker process. This architecture can place a limitation on the overall throughput of the message queue. However, you can also partition a queue or topic when it is created. You do this by setting the *EnablePartitioning* property of the queue or topic description to *true*.

A partitioned queue or topic is divided into multiple fragments, each of which is backed by a separate message store and message broker. Service Bus takes responsibility for creating and managing these fragments. When an application posts a message to a partitioned queue or topic, Service Bus assigns the message to a fragment for that queue or topic. When an application receives a message from a queue or subscription, Service Bus checks each fragment for the next available message and then passes it to the application for processing.

This structure helps distribute the load across message brokers and message stores, increasing scalability and improving availability. If the message broker or message store for one fragment is temporarily unavailable, Service Bus can retrieve messages from one of the remaining available fragments.

Service Bus assigns a message to a fragment as follows:

- If the message belongs to a session, all messages with the same value for the *SessionId* property are sent to the same fragment.
- If the message does not belong to a session, but the sender has specified a value for the *PartitionKey* property, then all messages with the same *PartitionKey* value are sent to the same fragment.

 **Note**

If the *SessionId* and *PartitionKey* properties are both specified, then they must be set to the same value or the message will be rejected.

- If the *SessionId* and *PartitionKey* properties for a message are not specified, but duplicate detection is enabled, the *MessageId* property will be used. All messages with the same *MessageId* will be directed to the same fragment.
- If messages do not include a *SessionId*, *PartitionKey*, or *MessageId* property, then Service Bus assigns messages to fragments sequentially. If a fragment is unavailable, Service Bus will move on to the next. This means that a temporary fault in the messaging infrastructure does not cause the message-send operation to fail.

Consider the following points when deciding if or how to partition a Service Bus message queue or topic:

- Service Bus queues and topics are created within the scope of a Service Bus namespace. Service Bus currently allows up to 100 partitioned queues or topics per namespace.
- Each Service Bus namespace imposes quotas on the available resources, such as the number of subscriptions per topic, the number of concurrent send and receive requests per second, and the maximum number of concurrent connections that can be established. These quotas are documented in [Service Bus quotas](#). If you expect to exceed these values, then create additional namespaces with their own queues and topics, and spread the work across these namespaces. For example, in a global application, create separate namespaces in each region and configure application instances to use the queues and topics in the nearest namespace.
- Messages that are sent as part of a transaction must specify a partition key. This can be a *SessionId*, *PartitionKey*, or *MessageId* property. All messages that are sent as part of the same transaction must specify the same partition key because they must be handled by the same message broker process. You cannot send messages to different queues or topics within the same transaction.
- Partitioned queues and topics can't be configured to be automatically deleted when they become idle.
- Partitioned queues and topics can't currently be used with the Advanced Message Queuing Protocol (AMQP) if you are building cross-platform or hybrid solutions.

Partitioning Azure Cosmos DB

[Azure Cosmos DB for NoSQL](#) is a NoSQL database for storing JSON documents. A document in an Azure 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.

Azure 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. Azure Cosmos DB distributes values according to hash of the partition key. A logical partition has a maximum size of 20 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 Azure 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 Azure 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 20 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 an Azure 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 Azure 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 Azure Cosmos DB for NoSQL:

- The resources available to an Azure 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.
- Azure Cosmos DB supports programmable items that can all be stored in a collection alongside documents. These include stored procedures, user-defined functions, and triggers (written in JavaScript). These items can access any document within the same collection. Furthermore, these items run either inside the scope of the ambient transaction (in the case of a trigger that fires as the result of a create, delete, or replace operation performed against a document), or by starting a new transaction (in the case of a stored procedure that is run as the result of an explicit client request). If the code in a programmable item throws an exception, the transaction is rolled back. You can use stored procedures and triggers to maintain integrity and consistency between documents, but these documents must all be part of the same collection.
- The collections that you intend to hold in the databases should be unlikely to exceed the throughput limits defined by the performance levels of the collections. For more information, see [Request Units in Azure Cosmos DB](#). If you anticipate reaching these limits, consider splitting collections across databases in different accounts to reduce the load per collection.

Partitioning Azure Search

The ability to search for data is often the primary method of navigation and exploration that's provided by many web applications. It helps users find resources quickly (for example, products in an e-commerce application) based on combinations of search criteria. The Azure Search service provides full-text search capabilities over web content, and includes features such as type-ahead, suggested queries based on near matches, and faceted navigation. For more information, see [What is Azure Search?](#).

Azure Search stores searchable content as JSON documents in a database. You define indexes that specify the searchable fields in these documents and provide these definitions to Azure Search. When a user submits a search request, Azure Search uses the appropriate indexes to find matching items.

To reduce contention, the storage that's used by Azure Search can be divided into 1, 2, 3, 4, 6, or 12 partitions, and each partition can be replicated up to 6 times. The product of the number of partitions multiplied by the number of replicas is called the *search unit* (SU). A single instance of Azure Search can contain a maximum of 36 SUs (a database with 12 partitions only supports a maximum of 3 replicas).

You are billed for each SU that is allocated to your service. As the volume of searchable content increases or the rate of search requests grows, you can add SUs to an existing instance of Azure Search to handle the extra load. Azure Search itself distributes the documents evenly across the partitions. No manual partitioning strategies are currently supported.

Each partition can contain a maximum of 15 million documents or occupy 300 GB of storage space (whichever is smaller). You can create up to 50 indexes. The performance of the service varies and depends on the complexity of the documents, the available indexes, and the effects of network latency. On average, a single replica (1 SU) should be able to handle 15 queries per second (QPS), although we recommend performing benchmarking with your own data to obtain a more precise measure of throughput. For more information, see [Service limits in Azure Search](#).

Note

You can store a limited set of data types in searchable documents, including strings, Booleans, numeric data, datetime data, and some geographical data. For more information, see the page [Supported data types \(Azure Search\)](#) on the Microsoft website.

You have limited control over how Azure Search partitions data for each instance of the service. However, in a global environment you might be able to improve performance and reduce latency and contention further by partitioning the service itself using either of the following strategies:

- Create an instance of Azure Search in each geographic region, and ensure that client applications are directed toward the nearest available instance. This strategy requires that any updates to searchable content are replicated in a timely manner across all instances of the service.
- Create two tiers of Azure Search:
 - A local service in each region that contains the data that's most frequently accessed by users in that region. Users can direct requests here for fast but limited results.
 - A global service that encompasses all the data. Users can direct requests here for slower but more complete results.

This approach is most suitable when there is a significant regional variation in the data that's being searched.

Partitioning Azure Cache for Redis

Azure Cache for Redis provides a shared caching service in the cloud that's based on the Redis key-value data store. As its name implies, Azure Cache for Redis is intended as a caching solution. Use it only for holding transient data and not as a permanent data store. Applications that use Azure Cache for Redis should be able to continue functioning if the cache is unavailable. Azure Cache for Redis supports primary/secondary replication to provide high availability, but currently limits the maximum cache size to 53 GB. If you need more space than this, you must create additional caches. For more information, see [Azure Cache for Redis](#).

Partitioning a Redis data store involves splitting the data across instances of the Redis service. Each instance constitutes a single partition. Azure Cache for Redis abstracts the Redis services behind a façade and does not expose them directly. The simplest way to implement partitioning is to create multiple Azure Cache for Redis instances and spread the data across them.

You can associate each data item with an identifier (a partition key) that specifies which cache stores the data item. The client application logic can then use this identifier to route requests to the appropriate partition. This scheme is very simple, but if the partitioning scheme changes (for example, if additional Azure Cache for Redis instances are created), client applications might need to be reconfigured.

Native Redis (not Azure Cache for Redis) supports server-side partitioning based on Redis clustering. In this approach, you can divide the data evenly across servers by using a hashing mechanism. Each Redis server stores metadata that describes the range of hash keys that the partition holds, and also contains information about which hash keys are located in the partitions on other servers.

Client applications simply send requests to any of the participating Redis servers (probably the closest one). The Redis server examines the client request. If it can be resolved locally, it performs the requested operation. Otherwise it forwards the request on to the appropriate server.

This model is implemented by using Redis clustering, and is described in more detail on the [Redis cluster tutorial](#) page on the Redis website. Redis clustering is transparent to client applications. Additional Redis servers can be added to the cluster (and the data can be repartitioned) without requiring that you reconfigure the clients.

Important

Azure Cache for Redis currently supports Redis clustering in premium tier only.

The page [Partitioning: how to split data among multiple Redis instances](#) on the Redis website provides more information about implementing partitioning with Redis. The remainder of this section assumes that you are implementing client-side or proxy-assisted partitioning.

Consider the following points when deciding how to partition data with Azure Cache for Redis:

- Azure Cache for Redis is not intended to act as a permanent data store, so whatever partitioning scheme you implement, your application code must be able to retrieve data from a location that's not the cache.
- Data that is frequently accessed together should be kept in the same partition. Redis is a powerful key-value store that provides several highly optimized mechanisms for structuring data. These mechanisms can be one of the following:
 - Simple strings (binary data up to 512 MB in length)
 - Aggregate types such as lists (which can act as queues and stacks)
 - Sets (ordered and unordered)
 - Hashes (which can group related fields together, such as the items that represent the fields in an object)
- The aggregate types enable you to associate many related values with the same key. A Redis key identifies a list, set, or hash rather than the data items that it

contains. These types are all available with Azure Cache for Redis and are described by the [Data types](#) page on the Redis website. For example, in part of an e-commerce system that tracks the orders that are placed by customers, the details of each customer can be stored in a Redis hash that is keyed by using the customer ID. Each hash can hold a collection of order IDs for the customer. A separate Redis set can hold the orders, again structured as hashes, and keyed by using the order ID. Figure 8 shows this structure. Note that Redis does not implement any form of referential integrity, so it is the developer's responsibility to maintain the relationships between customers and orders.

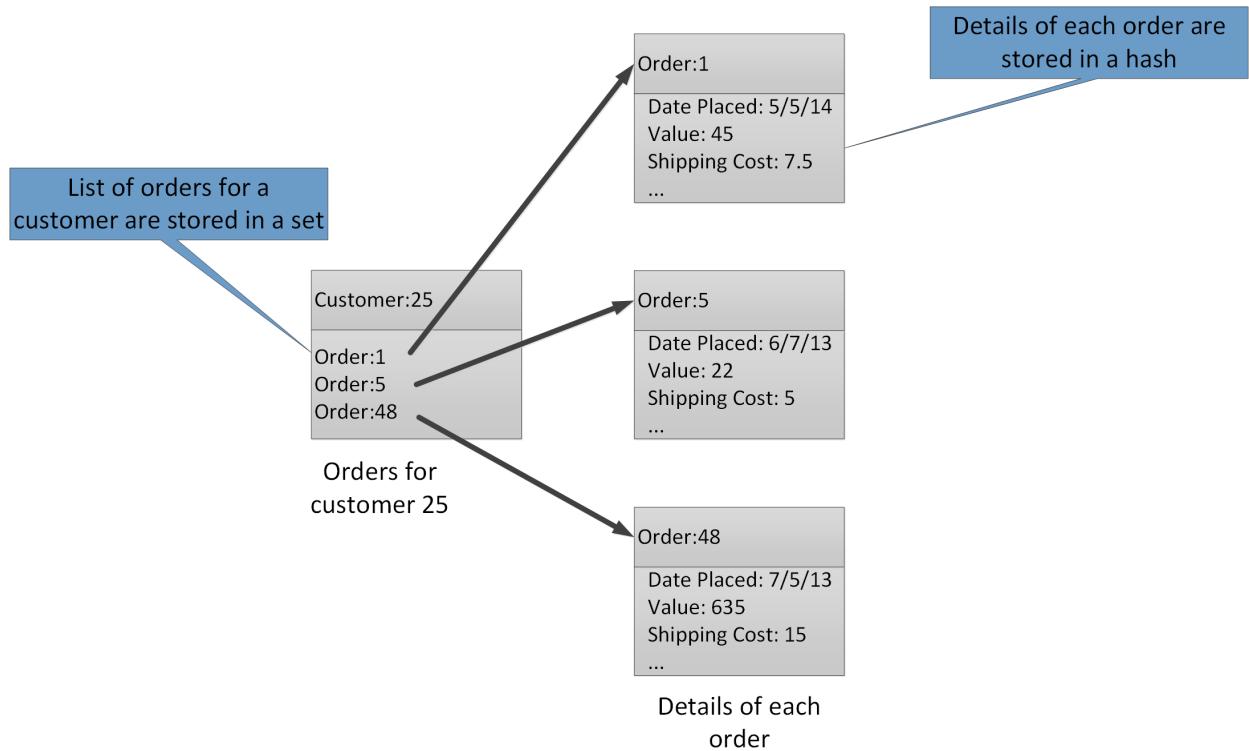


Figure 8. Suggested structure in Redis storage for recording customer orders and their details.

(!) Note

In Redis, all keys are binary data values (like Redis strings) and can contain up to 512 MB of data. In theory, a key can contain almost any information. However, we recommend adopting a consistent naming convention for keys that is descriptive of the type of data and that identifies the entity, but is not excessively long. A common approach is to use keys of the form "entity_type:ID". For example, you can use "customer:99" to indicate the key for a customer with the ID 99.

- You can implement vertical partitioning by storing related information in different aggregations in the same database. For example, in an e-commerce application, you can store commonly accessed information about products in one Redis hash

and less frequently used detailed information in another. Both hashes can use the same product ID as part of the key. For example, you can use "product: nn" (where nn is the product ID) for the product information and "product_details: nn" for the detailed data. This strategy can help reduce the volume of data that most queries are likely to retrieve.

- You can repartition a Redis data store, but keep in mind that it's a complex and time-consuming task. Redis clustering can repartition data automatically, but this capability is not available with Azure Cache for Redis. Therefore, when you design your partitioning scheme, try to leave sufficient free space in each partition to allow for expected data growth over time. However, remember that Azure Cache for Redis is intended to cache data temporarily, and that data held in the cache can have a limited lifetime specified as a time-to-live (TTL) value. For relatively volatile data, the TTL can be short, but for static data the TTL can be a lot longer. Avoid storing large amounts of long-lived data in the cache if the volume of this data is likely to fill the cache. You can specify an eviction policy that causes Azure Cache for Redis to remove data if space is at a premium.

 **Note**

When you use Azure Cache for Redis, you specify the maximum size of the cache (from 250 MB to 53 GB) by selecting the appropriate pricing tier. However, after an Azure Cache for Redis has been created, you cannot increase (or decrease) its size.

- Redis batches and transactions cannot span multiple connections, so all data that is affected by a batch or transaction should be held in the same database (shard).

 **Note**

A sequence of operations in a Redis transaction is not necessarily atomic. The commands that compose a transaction are verified and queued before they run. If an error occurs during this phase, the entire queue is discarded. However, after the transaction has been successfully submitted, the queued commands run in sequence. If any command fails, only that command stops running. All previous and subsequent commands in the queue are performed. For more information, go to the [Transactions](#) page on the Redis website.

- Redis supports a limited number of atomic operations. The only operations of this type that support multiple keys and values are MGET and MSET operations. MGET

operations return a collection of values for a specified list of keys, and MSET operations store a collection of values for a specified list of keys. If you need to use these operations, the key-value pairs that are referenced by the MSET and MGET commands must be stored within the same database.

Partitioning Azure Service Fabric

Azure Service Fabric is a microservices platform that provides a runtime for distributed applications in the cloud. Service Fabric supports .NET guest executables, stateful and stateless services, and containers. Stateful services provide a [reliable collection](#) to persistently store data in a key-value collection within the Service Fabric cluster. For more information about strategies for partitioning keys in a reliable collection, see [guidelines and recommendations for reliable collections in Azure Service Fabric](#).

Next steps

- [Overview of Azure Service Fabric](#) is an introduction to Azure Service Fabric.
- [Partition Service Fabric reliable services](#) provides more information about reliable services in Azure Service Fabric.

Partitioning Azure Event Hubs

[Azure Event Hubs](#) is designed for data streaming at massive scale, and partitioning is built into the service to enable horizontal scaling. Each consumer only reads a specific partition of the message stream.

The event publisher is only aware of its partition key, not the partition to which the events are published. This decoupling of key and partition insulates the sender from needing to know too much about the downstream processing. (It's also possible send events directly to a given partition, but generally that's not recommended.)

Consider long-term scale when you select the partition count. After an event hub is created, you can't change the number of partitions.

Next steps

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](#).

Preserve the original HTTP host name between a reverse proxy and its back-end web application

Azure API Management

Azure App Service

Azure Application Gateway

Azure Front Door

Azure Spring Apps

We recommend that you preserve the original HTTP host name when you use a reverse proxy in front of a web application. Having a different host name at the reverse proxy than the one that's provided to the back-end application server can lead to cookies or redirect URLs that don't work properly. For example, session state can get lost, authentication can fail, or back-end URLs can inadvertently be exposed to end users. You can avoid these problems by preserving the host name of the initial request so that the application server sees the same domain as the web browser.

This guidance applies especially to applications that are hosted in platform as a service (PaaS) offerings like [Azure App Service](#) and [Azure Spring Apps](#). This article provides specific [implementation guidance](#) for [Azure Application Gateway](#), [Azure Front Door](#), and [Azure API Management](#), which are commonly used reverse proxy services.

ⓘ Note

Web APIs are generally less sensitive to the problems caused by host name mismatches. They don't usually depend on cookies, unless you [use cookies to secure communications between a single-page app and its back-end API](#), for example, in a pattern known as [Backends for Frontends](#). Web APIs often don't return absolute URLs back to themselves, except in certain API styles, like [OData](#) and [HATEOAS](#). If your API implementation depends on cookies or generates absolute URLs, the guidance provided in this article does apply.

If you require end-to-end TLS/SSL (the connection between the reverse proxy and the back-end service uses HTTPS), the back-end service also needs a matching TLS certificate for the original host name. This requirement adds operational complexity when you deploy and renew certificates, but many PaaS services offer free TLS certificates that are fully managed.

Context

The host of an HTTP request

In many cases, the application server or some component in the request pipeline needs the internet domain name that was used by the browser to access it. This is the *host* of the request. It can be an IP address, but usually it's a name like `contoso.com` (which the browser then resolves to an IP address by using DNS). The host value is typically determined from the [host component of the request URI](#), which the browser sends to the application as the [Host HTTP header](#).

Important

Never use the value of the host in a security mechanism. The value is provided by the browser or some other user agent and can easily be manipulated by an end user.

In some scenarios, especially when there's an HTTP reverse proxy in the request chain, the original host header can get changed before it reaches the application server. A reverse proxy closes the client network session and sets up a new connection to the back end. In this new session, it can either carry over the original host name of the client session or set a new one. In the latter case, the proxy often still sends the original host value in other HTTP headers, like [Forwarded](#) or [X-Forwarded-Host](#). This value allows applications to determine the original host name, but only if they're coded to read these headers.

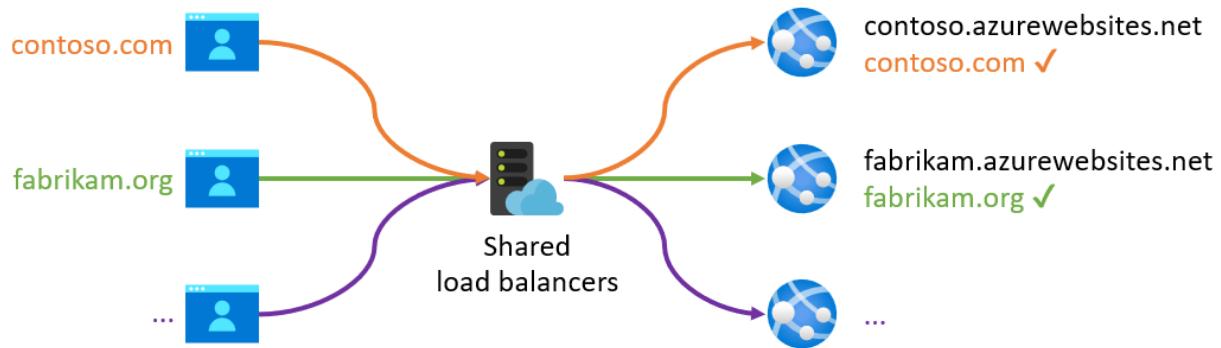
Why web platforms use the host name

Multitenant PaaS services often require a registered and validated host name in order to route an incoming request to the appropriate tenant's back-end server. This is because there's typically a shared pool of load balancers that accept incoming requests for all tenants. The tenants commonly use the incoming host name to look up the correct back end for the customer tenant.

To make it easy to get started, these platforms typically provide a default domain that's preconfigured to route traffic to your deployed instance. For App Service, this default domain is `azurewebsites.net`. Each web app that you create gets its own subdomain, for example, `contoso.azurewebsites.net`. Similarly, the default domain is `azurermicroservices.io` for Spring Apps and `azure-api.net` for API Management.

For production deployments, you don't use these default domains. You instead provide your own domain to align with your organization or application's brand. For example, `contoso.com` could resolve behind the scenes to the `contoso.azurewebsites.net` web

app on App Service, but this domain shouldn't be visible to an end user visiting the website. This custom `contoso.com` host name has to be registered with the PaaS service, however, so the platform can identify the back-end server that should respond to the request.



Why applications use the host name

Two common reasons that an application server needs the host name are to construct absolute URLs and to issue cookies for a specific domain. For example, when the application code needs to:

- Return an absolute rather than a relative URL in its HTTP response (although generally websites tend to render relative links when possible).
- Generate a URL to be used outside of its HTTP response where relative URLs can't be used, like for emailing a link to the website to a user.
- Generate an absolute redirect URL for an external service. For example, to an authentication service like Microsoft Entra ID to indicate where it should return the user after successful authentication.
- Issue HTTP cookies that are restricted to a certain host, as defined in the cookie's [Domain attribute ↗](#).

You can meet all these requirements by adding the expected host name to the application's configuration and using that statically defined value instead of the incoming host name on the request. However, this approach complicates application development and deployment. Also, a single installation of the application can serve multiple hosts. For example, a single web app can be used for multiple application tenants that all have their own unique host names (like `tenant1.contoso.com` and `tenant2.contoso.com`).

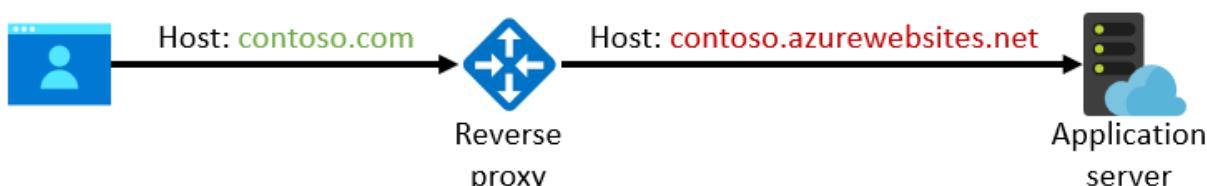
And sometimes the incoming host name is used by components outside of the application code or in middleware on the application server over which you don't have full control. Here are some examples:

- In App Service, you can [enforce HTTPS](#) for your web app. Doing so causes any HTTP requests that aren't secure to redirect to HTTPS. In this case, the incoming host name is used to generate the absolute URL for the HTTP redirect's `Location` header.
- Spring Apps uses a similar feature to [enforce HTTPS](#). It also uses the incoming host to generate the HTTPS URL.
- App Service has an [ARR affinity setting](#) to enable sticky sessions, so that requests from the same browser instance are always served by the same back-end server. This is performed by the App Service front ends, which add a cookie to the HTTP response. The cookie's `Domain` is set to the incoming host.
- App Service provides [authentication and authorization capabilities](#) to easily allow users to sign in and access data in APIs.
 - The incoming host name is used to construct the redirect URL to which the identity provider needs to return the user after successful authentication.
 - [Enabling this feature by default also turns on HTTP-to-HTTPS redirection](#). Again, the incoming host name is used to generate the redirect location.

Why you might be tempted to override the host name

Say you create a web application in App Service that has a default domain of `contoso.azurewebsites.net`. (Or in another service like Spring Apps.) You haven't configured a custom domain on App Service. To put a reverse proxy like Application Gateway (or any similar service) in front of this application, you set the DNS record for `contoso.com` to resolve to the IP address of Application Gateway. It therefore receives the request for `contoso.com` from the browser and is configured to forward that request to the IP address that `contoso.azurewebsites.net` resolves to: this is the final back-end service for the requested host. In this case, however, App Service doesn't recognize the `contoso.com` custom domain and rejects all incoming requests for this host name. It can't determine where to route the request.

It might seem like the easy way to make this configuration work is to override or rewrite the `Host` header of the HTTP request in Application Gateway and set it to the value of `contoso.azurewebsites.net`. If you do, the outgoing request from Application Gateway makes it seem like the original request was really intended for `contoso.azurewebsites.net` instead of `contoso.com`:



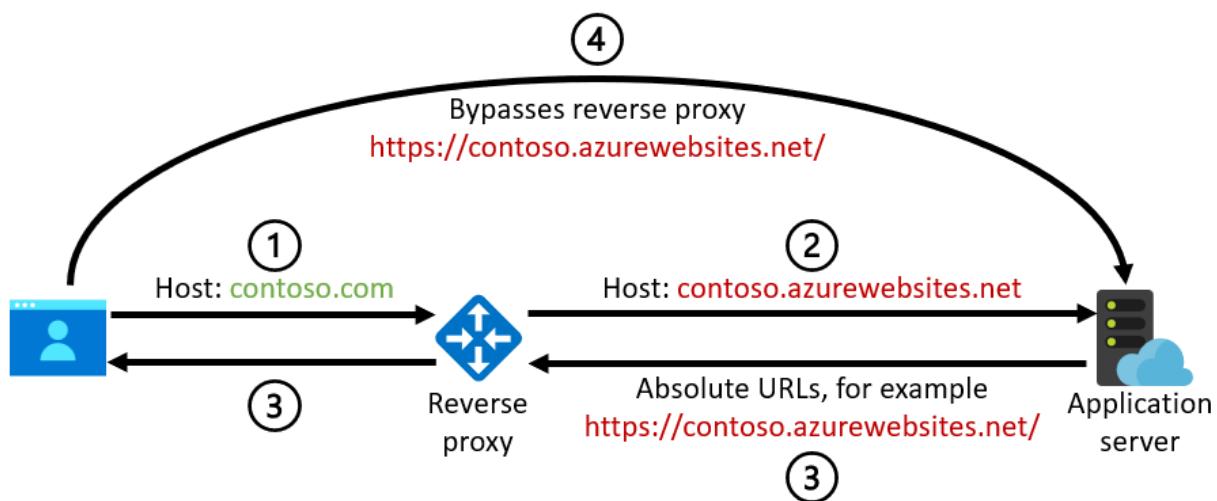
At this point, App Service does recognize the host name, and it accepts the request without requiring that a custom domain name be configured. In fact, [Application Gateway makes it easy to override the host header](#) with the host of the back-end pool. [Azure Front Door even does so by default](#).

The problem with this solution, however, is that it can result in various problems when the app doesn't see the original host name.

Potential problems

Incorrect absolute URLs

If the original host name isn't preserved and the application server uses the incoming host name to generate absolute URLs, the back-end domain might be disclosed to an end user. These absolute URLs could be generated by the application code or, as noted earlier, by platform features like the support for HTTP-to-HTTPS redirection in App Service and Spring Apps. This diagram illustrates the problem:



1. The browser sends a request for `contoso.com` to the reverse proxy.
2. The reverse proxy rewrites the host name to `contoso.azurewebsites.net` in the request to the back-end web application (or to a similar default domain for another service).
3. The application generates an absolute URL that's based on the incoming `contoso.azurewebsites.net` host name, for example,
`https://contoso.azurewebsites.net/`.
4. The browser follows this URL, which goes directly to the back-end service rather than back to the reverse proxy at `contoso.com`.

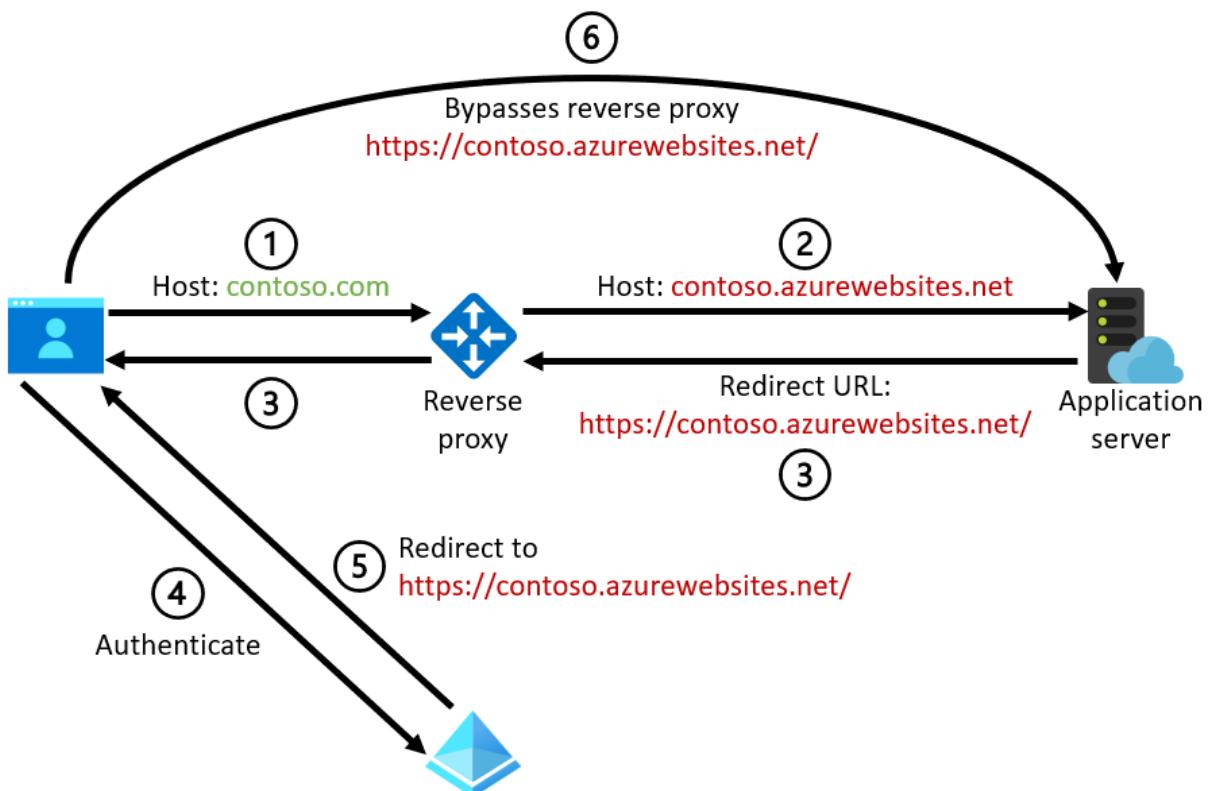
This might even pose a security risk in the common case where the reverse proxy also serves as a web application firewall. The user receives a URL that goes straight to the back-end application and bypasses the reverse proxy.

ⓘ Important

Because of this security risk, you need to ensure that the back-end web application only directly accepts network traffic from the reverse proxy (for example, by using [access restrictions in App Service](#)). If you do, even if an incorrect absolute URL is generated, at least it doesn't work and can't be used by a malicious user to bypass the firewall.

Incorrect redirect URLs

A common and more specific case of the previous scenario occurs when absolute redirect URLs are generated. These URLs are required by identity services like Microsoft Entra ID when you use browser-based identity protocols like OpenID Connect, OAuth 2.0, or SAML 2.0. These redirect URLs could be generated by the application server or middleware itself, or, as noted earlier, by platform features like the App Service [authentication and authorization capabilities](#). This diagram illustrates the problem:

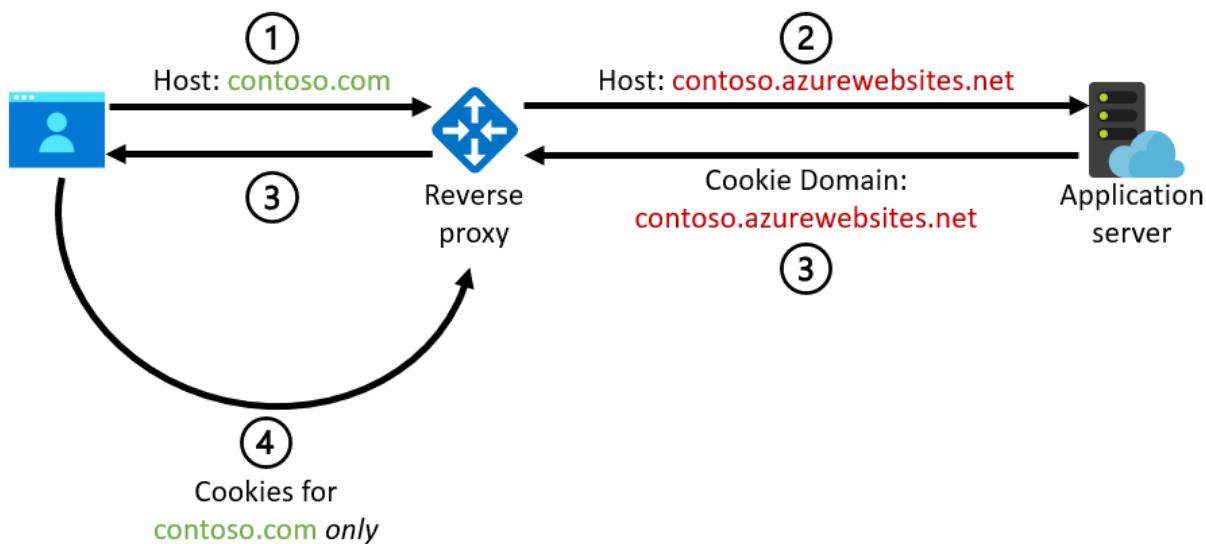


1. The browser sends a request for `contoso.com` to the reverse proxy.

2. The reverse proxy rewrites the host name to `contoso.azurewebsites.net` on the request to the back-end web application (or to a similar default domain for another service).
3. The application generates an absolute redirect URL that's based on the incoming `contoso.azurewebsites.net` host name, for example, `https://contoso.azurewebsites.net/`.
4. The browser goes to the identity provider to authenticate the user. The request includes the generated redirect URL to indicate where to return the user after successful authentication.
5. Identity providers typically require redirect URLs to be registered up front, so at this point the identity provider should reject the request because the provided redirect URL isn't registered. (It wasn't supposed to be used.) If for some reason the redirect URL is registered, however, the identity provider redirects the browser to the redirect URL that's specified in the authentication request. In this case, the URL is `https://contoso.azurewebsites.net/`.
6. The browser follows this URL, which goes directly to the back-end service rather than back to the reverse proxy.

Broken cookies

A host name mismatch can also lead to problems when the application server issues cookies and uses the incoming host name to construct the [Domain attribute of the cookie](#). The Domain attribute ensures that the cookie will be used only for that specific domain. These cookies can be generated by the application code or, as noted earlier, by platform features like the App Service [ARR affinity setting](#). This diagram illustrates the problem:

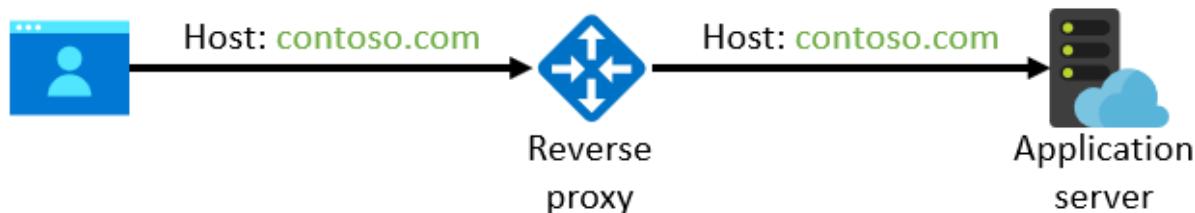


1. The browser sends a request for `contoso.com` to the reverse proxy.

2. The reverse proxy rewrites the host name to be `contoso.azurewebsites.net` in the request to the back-end web application (or to a similar default domain for another service).
3. The application generates a cookie that uses a domain based on the incoming `contoso.azurewebsites.net` host name. The browser stores the cookie for this specific domain rather than the `contoso.com` domain that the user is actually using.
4. The browser doesn't include the cookie on any subsequent request for `contoso.com` because the cookie's `contoso.azurewebsites.net` domain doesn't match the domain of the request. The application doesn't receive the cookie it issued earlier. As a consequence, the user might lose state that's supposed to be in the cookie, or features like ARR affinity don't work. Unfortunately, none of these problems generate an error or are directly visible to the end user. That makes them difficult to troubleshoot.

Implementation guidance for common Azure services

To avoid the potential problems discussed here, we recommend that you preserve the original host name in the call between the reverse proxy and the back-end application server:



Back-end configuration

Many web hosting platforms require that you explicitly configure the allowed incoming host names. The following sections describe how to implement this configuration for the most common Azure services. Other platforms usually provide similar methods for configuring custom domains.

If you host your web application in **App Service**, you can [attach a custom domain name to the web app](#) and avoid using the default `azurewebsites.net` host name towards the back end. You don't need to change your DNS resolution when you attach a custom domain to the web app: you can [verify the domain by using a TXT record](#) without affecting your regular `CNAME` or `A` records. (These records will still resolve to the IP

address of the reverse proxy.) If you require end-to-end TLS/SSL, you can [import an existing certificate from Key Vault](#) or use an [App Service Certificate](#) for your custom domain. (Note that the free [App Service managed certificate](#) can't be used in this case, as it requires the domain's DNS record to resolve directly to App Service, not the reverse proxy.)

Similarly, if you're using [Spring Apps](#), you can [use a custom domain for your app](#) to avoid using the `azuremicroservices.io` host name. You can import an existing or self-signed certificate if you require end-to-end TLS/SSL.

If you have a reverse proxy in front of [API Management](#) (which itself also acts as a reverse proxy), you can [configure a custom domain on your API Management instance](#) to avoid using the `azure-api.net` host name. You can import an existing or free managed certificate if you require end-to-end TLS/SSL. As noted previously, however, APIs are less sensitive to the problems caused by host name mismatches, so this configuration might not be as important.

If you host your applications on [other platforms](#), like on Kubernetes or directly on virtual machines, there's no built-in functionality that depends on the incoming host name. You're responsible for how the host name is used in the application server itself. The recommendation to preserve the host name typically still applies for any components in your application that depend on it, unless you specifically make your application aware of reverse proxies and respect the [forwarded](#) ↗ or [X-Forwarded-Host](#) ↗ headers, for example.

Reverse proxy configuration

When you define the back ends within the reverse proxy, you can still use the default domain of the back-end service, for example, `https://contoso.azurewebsites.net/`. This URL is used by the reverse proxy to resolve the correct IP address for the back-end service. If you use the platform's default domain, the IP address is always guaranteed to be correct. You typically can't use the public-facing domain, like `contoso.com`, because it should resolve to the IP address of the reverse proxy itself. (Unless you use more advanced DNS resolution techniques, like [split-horizon DNS](#)).

ⓘ Important

If you have a next-generation firewall like [Azure Firewall Premium](#) between the reverse proxy and the final back end, you might need to use split-horizon DNS. This type of firewall might explicitly check whether the HTTP `Host` header resolves to the target IP address. In these cases, the original host name that's used by the

browser should resolve to the IP address of the reverse proxy when it's accessed from the public internet. From the point of view of the firewall, however, that host name should resolve to the IP address of the final back-end service. For more information, see [Zero-trust network for web applications with Azure Firewall and Application Gateway](#).

Most reverse proxies allow you to configure which host name is passed to the back-end service. The following information explains how to ensure, for the most common Azure services, that the original host name of the incoming request is used.

Note

In all cases, you can also choose to override the host name with an explicitly defined custom domain rather than taking it from the incoming request. If the application uses only a single domain, that approach might work fine. If the same application deployment accepts requests from multiple domains (for example, in multitenant scenarios), you can't statically define a single domain. You should take the host name from the incoming request (again, unless the application is explicitly coded to take additional HTTP headers into account). Therefore, the general recommendation is that you shouldn't override the host name at all. Pass the incoming host name unmodified to the back end.

Application Gateway

If you use [Application Gateway](#) as the reverse proxy, you can ensure that the original host name is preserved by disabling **Override with new host name** on the back-end HTTP setting. Doing so disables both [Pick host name from back-end address](#) and [Override with specific domain name](#). (Both of these settings override the host name.) In the [Azure Resource Manager properties for Application Gateway](#), this configuration corresponds to setting the `hostName` property to `null` and `pickHostNameFromBackendAddress` to `false`.

Because health probes are sent outside the context of an incoming request, they can't dynamically determine the correct host name. Instead, you have to create a custom health probe, disable **Pick host name from backend HTTP settings**, and [explicitly specify the host name](#). For this host name, you should also use an appropriate custom domain, for consistency. (You could, however, use the default domain of the hosting platform here, because health probes ignore incorrect cookies or redirect URLs in the response.)

Azure Front Door

If you use [Azure Front Door](#), you can avoid overriding the host name by leaving the [back-end host header](#) blank in the back-end pool definition. In the [Resource Manager definition of the back-end pool](#), this configuration corresponds to setting `backendHostHeader` to `null`.

If you use [Azure Front Door Standard or Premium](#), you can preserve the host name by leaving the [origin host header](#) blank in the origin definition. In the [Resource Manager definition of the origin](#), this configuration corresponds to setting `originHostHeader` to `null`.

API Management

By default, [API Management](#) overrides the host name that's sent to the back end with the host component of the API's web service URL (which corresponds to the `serviceUrl` value of the [Resource Manager definition of the API](#)).

You can force API Management to instead use the host name of the incoming request by adding an `inbound` [Set HTTP header](#) policy, as follows:

XML

```
<inbound>
  <base />
  <set-header name="Host" exists-action="override">
    <value>@(context.Request.OriginalUrl.Host)</value>
  </set-header>
</inbound>
```

As noted previously, however, APIs are less sensitive to the problems caused by host name mismatches, so this configuration might not be as important.

Next steps

- [App Service](#)
- [Spring Apps](#)
- [Application Gateway](#)
- [Azure Front Door](#)
- [API Management](#)

Related resources

- Zero-trust network for web applications with Azure Firewall and Application Gateway
- Protect APIs with Application Gateway and API Management
- Enterprise deployment using App Services Environment

Message encoding considerations

Article • 11/08/2022

Many cloud applications use asynchronous messages to exchange information between components of the system. An important aspect of messaging is the format used to encode the payload data. After you [choose a messaging technology](#), the next step is to define how the messages will be encoded. There are many options available, but the right choice depends on your use case.

This article describes some of the considerations.

Message exchange needs

A message exchange between a producer and a consumer needs:

- A shape or structure that defines the payload of the message.
- An encoding format to represent the payload.
- Serialization libraries to read and write the encoded payload.

The producer of the message defines the message shape based on the business logic and the information it wants to send to the consumer(s). To structure the shape, divide the information into discrete or related subjects (fields). Decide the characteristics of the values for those fields. Consider: What is the most efficient datatype? Will the payload always have certain fields? Will the payload have a single record or a repeated set of values?

Then, choose an encoding format depending on your need. Certain factors include the ability to create highly structured data if you need it, time taken to encode and transfer the message, and the ability to parse the payload. Depending on the encoding format, choose a serialization library that is well supported.

A consumer of the message must be aware of those decisions so that it knows how to read incoming messages.

To transfer messages, the producer serializes the message to an encoding format. At the receiving end, the consumer deserializes the payload to use the data. This way both entities share the model and as long as the shape doesn't change, messaging continues without issues. When the contract changes, the encoding format should be capable of handling the change without breaking the consumer.

Some encoding formats such as JSON are self-describing, meaning they can be parsed without referencing a schema. However, such formats tend to yield larger messages.

With other formats, the data may not be parsed as easily but the messages are compact. This article highlights some factors that can help you choose a format.

Encoding format considerations

The encoding format defines how a set of structured data is represented as bytes. The type of message can influence the format choice. Messages related to business transactions most likely will contain highly structured data. Also, you may want to retrieve it later for auditing purposes. For a stream of events, you might want to read a sequence of records as quickly as possible and store it for statistical analysis.

Here are some points to consider when choosing an encoding format.

Human readability

Message encoding can be broadly divided into text-based and binary formats.

With text-based encoding, the message payload is in plain text and therefore can be inspected by a person without using any code libraries. Human readable formats are suitable for archival data. Also, because a human can read the payload, text-based formats are easier to debug and send to logs for troubleshooting errors.

The downside is that the payload tends to be larger. A common text-based format is JSON.

Encryption

If there is sensitive data in the messages, consider whether those messages should be encrypted in their entirety as described in this guidance on [encrypting Azure Service Bus data at rest](#). Alternatively, if only certain fields need to be encrypted and you'd prefer to reduce cloud costs, consider using a library like [NServiceBus](#) for that.

Encoding size

Message size impacts network I/O performance across the wire. Binary formats are more compact than text-based formats. Binary formats require serialization/deserialization libraries. The payload can't be read unless it's decoded.

Use a binary format if you want to reduce wire footprint and transfer messages faster. This category of format is recommended in scenarios where storage or network bandwidth is a concern. Options for binary formats include Apache Avro, Google

Protocol Buffers (protobuf), MessagePack, and Concise Binary Object Representation (CBOR). The pros and cons of those formats are described in [this section](#).

The disadvantage is that the payload isn't human readable. Most binary formats use complex systems that can be costly to maintain. Also, they need specialized libraries to decode, which may not be supported if you want to retrieve archival data.

Understanding the payload

A message payload arrives as a sequence of bytes. To parse this sequence, the consumer must have access to metadata that describes the data fields in the payload. There are two main approaches for storing and distributing metadata:

Tagged metadata. In some encodings, notably JSON, fields are tagged with the data type and identifier, within the body of the message. These formats are *self-describing* because they can be parsed into a dictionary of values without referring to a schema. One way for the consumer to understand the fields is to query for expected values. For example, the producer sends a payload in JSON. The consumer parses the JSON into a dictionary and checks the existence of fields to understand the payload. Another way is for the consumer to apply a data model shared by the producer. For example, if you are using a statically typed language, many JSON serialization libraries can parse a JSON string into a typed class.

Schema. A schema formally defines the structure and data fields of a message. In this model, the producer and consumer have a contract through a well-defined schema. The schema can define the data types, required/optional fields, version information, and the structure of the payload. The producer sends the payload as per the writer schema. The consumer receives the payload by applying a reader schema. The message is serialized/deserialized by using the encoding-specific libraries. There are two ways to distribute schemas:

- Store the schema as a preamble or header in the message but separate from the payload.
- Store the schema externally.

Some encoding formats define the schema and use tools that generate classes from the schema. The producer and consumer use those classes and libraries to serialize and deserialize the payload. The libraries also provide compatibility checks between the writer and reader schema. Both protobuf and Apache Avro follow that approach. The key difference is that protobuf has a language-agnostic schema definition but Avro uses compact JSON. Another difference is in the way both formats provide compatibility checks between reader and writer schemas.

Another way to store the schema externally in a schema registry. The message contains a reference to the schema and the payload. The producer sends the schema identifier in the message and the consumer retrieves the schema by specifying that identifier from an external store. Both parties use format-specific library to read and write messages. Apart from storing the schema a registry can provide compatibility checks to make sure the contract between the producer and consumer isn't broken as the schema evolves.

Before choosing an approach, decide what is more important: the transfer data size or the ability to parse the archived data later.

Storing the schema along with the payload yields a larger encoding size and is preferred for intermittent messages. Choose this approach if transferring smaller chunks of bytes is crucial or you expect a sequence of records. The cost of maintaining an external schema store can be high.

However, if on-demand decoding of the payload is more important than size, including the schema with the payload or the tagged metadata approach guarantees decoding afterwards. There might be a significant increase in message size and may impact the cost of storage.

Schema versioning

As business requirements change, the shape is expected to change, and the schema will evolve. Versioning allows the producer to indicate schema updates that might include new features. There are two aspects to versioning:

- The consumer should be aware of the changes.

One way is for the consumer to check all fields to determine whether the schema has changed. Another way is for the producer to publish a schema version number with the message. When the schema evolves, the producer increments the version.

- Changes must not affect or break the business logic of consumers.

Suppose a field is added to an existing schema. If consumers using the new version get a payload as per the old version, their logic might break if they are not able to overlook the lack of the new field. Considering the reverse case, suppose a field is removed in the new schema. Consumers using the old schema might not be able to read the data.

Encoding formats such as Avro offer the ability to define default values. In the preceding example, if the field is added with a default value, the missing field will

be populated with the default value. Other formats such as protobuf provide similar functionality through required and optional fields.

Payload structure

Consider the way data is arranged in the payload. Is it a sequence of records or a discrete single payload? The payload structure can be categorized into one of these models:

- Array/dictionary/value: Defines entries that hold values in one or multi-dimensional arrays. Entries have unique key-value pairs. It can be extended to represent the complex structures. Some examples include, JSON, Apache Avro, and MessagePack.

This layout is suitable if messages are individual encoded with different schemas. If you have multiple records, the payload can get overly redundant causing the payload to bloat.

- Tabular data: Information is divided into rows and columns. Each column indicates a field, or the subject of the information and each row contains values for those fields. This layout is efficient for a repeating set of information, such as time series data.

CSV is one of the simplest text-based formats. It presents data as a sequence of records with a common header. For binary encoding, Apache Avro has a preamble similar to a CSV header but generate compact encoding size.

Library support

Consider using well-known formats over a proprietary model.

Well-known formats are supported through libraries that are universally supported by the community. With specialized formats, you need specific libraries. Your business logic might have to work around some of the API design choices provided by the libraries.

For schema-based format, choose an encoding library that makes compatibility checks between the reader and writer schema. Certain encoding libraries, such as Apache Avro, expect the consumer to specify both writer and the reader schema before deserializing the message. This check ensures that the consumer is aware of the schema versions.

Interoperability

Your choice of formats might depend on the particular workload or technology ecosystem.

For example:

- Azure Stream Analytics has native support for JSON, CSV, and Avro. When using Stream Analytics, it makes sense to choose one of these formats if possible. If not, you can provide a [custom deserializer](#), but this adds some additional complexity to your solution.
- JSON is a standard interchange format for HTTP REST APIs. If your application receives JSON payloads from clients and then places these onto a message queue for asynchronous processing, it might make sense to use JSON for the messaging, rather than re-encode into a different format.

These are just two examples of interoperability considerations. In general, standardized formats will be more interoperable than custom formats. In text-based options, JSON is one of the most interoperable.

Choices for encoding formats

Here are some popular encoding formats. Factor in the considerations before you choose a format.

JSON

[JSON](#) is an open standard (IETF [RFC8259](#)). It's a text-based format that follows the array/dictionary/value model.

JSON can be used for tagging metadata and you can parse the payload without a schema. JSON supports the option to specify optional fields, which helps with forward and backward compatibility.

The biggest advantage is that its universally available. It's most interoperable and the default encoding format for many messaging services.

Being a text-based format, it isn't efficient over the wire and not an ideal choice in cases where storage is a concern. If you're returning cached items directly to a client via HTTP, storing JSON could save the cost of deserializing from another format and then serializing to JSON.

Use JSON for single-record messages or for a sequence of messages in which each message has a different schema. Avoid using JSON for a sequence of records, such as

for time-series data.

There are other variations of JSON such as [BSON](#), which is a binary encoding aligned to work with MongoDB.

Comma-Separated Values (CSV)

CSV is a text-based tabular format. The header of the table indicates the fields. It's a preferred choice where the message contains a set of records.

The disadvantage is lack of standardization. There are many ways of expressing separators, headers, and empty fields.

Protocol Buffers (protobuf)

[Google Protocol Buffers](#) (or protobuf) is a serialization format that uses strongly typed definition files to define schemas in key/value pairs. These definition files are then compiled to language-specific classes that are used for serializing and deserializing messages.

The message contains a compressed binary small payload, which results in faster transfer. The downside is the payload isn't human readable. Also, because the schema is external, it's not recommended for cases where you have to retrieve archived data.

Apache Avro

[Apache Avro](#) is a binary serialization format that uses definition file similar to protobuf but there isn't a compilation step. Instead, serialized data always includes a schema preamble.

The preamble can hold the header or a schema identifier. Because of the smaller encoding size, Avro is recommended for streaming data. Also, because it has a header that applies to a set of records, it's a good choice for tabular data.

MessagePack

[MessagePack](#) is a binary serialization format that is designed to be compact for transmission over the wire. There are no message schemas or message type checking. This format isn't recommended for bulk storage.

CBOR

[Concise Binary Object Representation](#) (CBOR) (Specification) is a binary format that offers small encoding size. The advantage of CBOR over MessagePack is that its compliant with IETF in RFC7049.

Next steps

- Understand [messaging design patterns](#) for cloud applications.

Monitoring and diagnostics guidance

Azure Monitor

Distributed applications and services running in the cloud are, by their nature, complex pieces of software that comprise many moving parts. In a production environment, it's important to be able to track the way in which users use your system, trace resource utilization, and generally monitor the health and performance of your system. You can use this information as a diagnostic aid to detect and correct issues, and also to help spot potential problems and prevent them from occurring.

Monitoring and diagnostics scenarios

You can use monitoring to gain an insight into how well a system is functioning. Monitoring is a crucial part of maintaining quality-of-service targets. Common scenarios for collecting monitoring data include:

- Ensuring that the system remains healthy.
- Tracking the availability of the system and its component elements.
- Maintaining performance to ensure that the throughput of the system does not degrade unexpectedly as the volume of work increases.
- Guaranteeing that the system meets any service-level agreements (SLAs) established with customers.
- Protecting the privacy and security of the system, users, and their data.
- Tracking the operations that are performed for auditing or regulatory purposes.
- Monitoring the day-to-day usage of the system and spotting trends that might lead to problems if they're not addressed.
- Tracking issues that occur, from initial report through to analysis of possible causes, rectification, consequent software updates, and deployment.
- Tracing operations and debugging software releases.

Note

This list is not intended to be comprehensive. This document focuses on these scenarios as the most common situations for performing monitoring. There might be others that are less common or are specific to your environment.

The following sections describe these scenarios in more detail. The information for each scenario is discussed in the following format:

1. A brief overview of the scenario.
2. The typical requirements of this scenario.
3. The raw instrumentation data that's required to support the scenario, and possible sources of this information.
4. How this raw data can be analyzed and combined to generate meaningful diagnostic information.

Health monitoring

A system is healthy if it is running and capable of processing requests. The purpose of health monitoring is to generate a snapshot of the current health of the system so that you can verify that all components of the system are functioning as expected.

Requirements for health monitoring

An operator should be alerted quickly (within a matter of seconds) if any part of the system is deemed to be unhealthy. The operator should be able to ascertain which parts of the system are functioning normally, and which parts are experiencing problems.

System health can be highlighted through a traffic-light system:

- Red for unhealthy (the system has stopped)
- Yellow for partially healthy (the system is running with reduced functionality)
- Green for completely healthy

A comprehensive health-monitoring system enables an operator to drill down through the system to view the health status of subsystems and components. For example, if the overall system is depicted as partially healthy, the operator should be able to zoom in and determine which functionality is currently unavailable.

Data sources, instrumentation, and data-collection requirements

The raw data that's required to support health monitoring can be generated as a result of:

- Tracing execution of user requests. This information can be used to determine which requests have succeeded, which have failed, and how long each request takes.

- Synthetic user monitoring. This process simulates the steps performed by a user and follows a predefined series of steps. The results of each step should be captured.
- Logging exceptions, faults, and warnings. This information can be captured as a result of trace statements embedded into the application code, as well as retrieving information from the event logs of any services that the system references.
- Monitoring the health of any third-party services that the system uses. This monitoring might require retrieving and parsing health data that these services supply. This information might take a variety of formats.
- Endpoint monitoring. This mechanism is described in more detail in the "Availability monitoring" section.
- Collecting ambient performance information, such as background CPU utilization or I/O (including network) activity.

Analyzing health data

The primary focus of health monitoring is to quickly indicate whether the system is running. Hot analysis of the immediate data can trigger an alert if a critical component is detected as unhealthy. (It fails to respond to a consecutive series of pings, for example.) The operator can then take the appropriate corrective action.

A more advanced system might include a predictive element that performs a cold analysis over recent and current workloads. A cold analysis can spot trends and determine whether the system is likely to remain healthy or whether the system will need additional resources. This predictive element should be based on critical performance metrics, such as:

- The rate of requests directed at each service or subsystem.
- The response times of these requests.
- The volume of data flowing into and out of each service.

If the value of any metric exceeds a defined threshold, the system can raise an alert to enable an operator or autoscaling (if available) to take the preventative actions necessary to maintain system health. These actions might involve adding resources, restarting one or more services that are failing, or applying throttling to lower-priority requests.

Availability monitoring

A truly healthy system requires that the components and subsystems that compose the system are available. Availability monitoring is closely related to health monitoring. But whereas health monitoring provides an immediate view of the current health of the system, availability monitoring is concerned with tracking the availability of the system and its components to generate statistics about the uptime of the system.

In many systems, some components (such as a database) are configured with built-in redundancy to permit rapid failover in the event of a serious fault or loss of connectivity. Ideally, users should not be aware that such a failure has occurred. But from an availability monitoring perspective, it's necessary to gather as much information as possible about such failures to determine the cause and take corrective actions to prevent them from recurring.

The data that's required to track availability might depend on a number of lower-level factors. Many of these factors might be specific to the application, system, and environment. An effective monitoring system captures the availability data that corresponds to these low-level factors and then aggregates them to give an overall picture of the system. For example, in an e-commerce system, the business functionality that enables a customer to place orders might depend on the repository where order details are stored and the payment system that handles the monetary transactions for paying for these orders. The availability of the order-placement part of the system is therefore a function of the availability of the repository and the payment subsystem.

Requirements for availability monitoring

An operator should also be able to view the historical availability of each system and subsystem, and use this information to spot any trends that might cause one or more subsystems to periodically fail. (Do services start to fail at a particular time of day that corresponds to peak processing hours?)

A monitoring solution should provide an immediate and historical view of the availability or unavailability of each subsystem. It should also be capable of quickly alerting an operator when one or more services fail or when users can't connect to services. This is a matter of not only monitoring each service, but also examining the actions that each user performs if these actions fail when they attempt to communicate with a service. To some extent, a degree of connectivity failure is normal and might be due to transient errors. But it might be useful to allow the system to raise an alert for the number of connectivity failures to a specified subsystem that occur during a specific period.

Data sources, instrumentation, and data-collection requirements

As with health monitoring, the raw data that's required to support availability monitoring can be generated as a result of synthetic user monitoring and logging any exceptions, faults, and warnings that might occur. In addition, availability data can be obtained from performing endpoint monitoring. The application can expose one or more health endpoints, each testing access to a functional area within the system. The monitoring system can ping each endpoint by following a defined schedule and collect the results (success or fail).

All timeouts, network connectivity failures, and connection retry attempts must be recorded. All data should be timestamped.

Analyzing availability data

The instrumentation data must be aggregated and correlated to support the following types of analysis:

- The immediate availability of the system and subsystems.
- The availability failure rates of the system and subsystems. Ideally, an operator should be able to correlate failures with specific activities: what was happening when the system failed?
- A historical view of failure rates of the system or any subsystems across any specified period, and the load on the system (number of user requests, for example) when a failure occurred.
- The reasons for unavailability of the system or any subsystems. For example, the reasons might be service not running, connectivity lost, connected but timing out, and connected but returning errors.

You can calculate the percentage availability of a service over a period of time by using the following formula:

Console

```
%Availability = ((Total Time - Total Downtime) / Total Time ) * 100
```

This is useful for SLA purposes. ([SLA monitoring](#) is described in more detail later in this guidance.) The definition of *downtime* depends on the service. For example, Visual Studio Team Services Build Service defines downtime as the period (total accumulated minutes) during which Build Service is unavailable. A minute is considered unavailable if

all continuous HTTP requests to Build Service to perform customer-initiated operations throughout the minute either result in an error code or do not return a response.

Performance monitoring

As the system is placed under more and more stress (by increasing the volume of users), the size of the datasets that these users access grows and the possibility of failure of one or more components becomes more likely. Frequently, component failure is preceded by a decrease in performance. If you're able detect such a decrease, you can take proactive steps to remedy the situation.

System performance depends on a number of factors. Each factor is typically measured through key performance indicators (KPIs), such as the number of database transactions per second or the volume of network requests that are successfully serviced in a specified time frame. Some of these KPIs might be available as specific performance measures, whereas others might be derived from a combination of metrics.

ⓘ Note

Determining poor or good performance requires that you understand the level of performance at which the system should be capable of running. This requires observing the system while it's functioning under a typical load and capturing the data for each KPI over a period of time. This might involve running the system under a simulated load in a test environment and gathering the appropriate data before deploying the system to a production environment.

You should also ensure that monitoring for performance purposes does not become a burden on the system. You might be able to dynamically adjust the level of detail for the data that the performance monitoring process gathers.

Requirements for performance monitoring

To examine system performance, an operator typically needs to see information that includes:

- The response rates for user requests.
- The number of concurrent user requests.
- The volume of network traffic.
- The rates at which business transactions are being completed.
- The average processing time for requests.

It can also be helpful to provide tools that enable an operator to help spot correlations, such as:

- The number of concurrent users versus request latency times (how long it takes to start processing a request after the user has sent it).
- The number of concurrent users versus the average response time (how long it takes to complete a request after it has started processing).
- The volume of requests versus the number of processing errors.

Along with this high-level functional information, an operator should be able to obtain a detailed view of the performance for each component in the system. This data is typically provided through low-level performance counters that track information such as:

- Memory utilization.
- Number of threads.
- CPU processing time.
- Request queue length.
- Disk or network I/O rates and errors.
- Number of bytes written or read.
- Middleware indicators, such as queue length.

All visualizations should allow an operator to specify a time period. The displayed data might be a snapshot of the current situation and/or a historical view of the performance.

An operator should be able to raise an alert based on any performance measure for any specified value during any specified time interval.

Data sources, instrumentation, and data-collection requirements

You can gather high-level performance data (throughput, number of concurrent users, number of business transactions, error rates, and so on) by monitoring the progress of users' requests as they arrive and pass through the system. This involves incorporating tracing statements at key points in the application code, together with timing information. All faults, exceptions, and warnings should be captured with sufficient data for correlating them with the requests that caused them. The Internet Information Services (IIS) log is another useful source.

If possible, you should also capture performance data for any external systems that the application uses. These external systems might provide their own performance counters or other features for requesting performance data. If this is not possible, record

information such as the start time and end time of each request made to an external system, together with the status (success, fail, or warning) of the operation. For example, you can use a stopwatch approach to time requests: start a timer when the request starts and then stop the timer when the request finishes.

Low-level performance data for individual components in a system might be available through features and services such as Windows performance counters and Azure Diagnostics.

Analyzing performance data

Much of the analysis work consists of aggregating performance data by user request type and/or the subsystem or service to which each request is sent. An example of a user request is adding an item to a shopping cart or performing the checkout process in an e-commerce system.

Another common requirement is summarizing performance data in selected percentiles. For example, an operator might determine the response times for 99 percent of requests, 95 percent of requests, and 70 percent of requests. There might be SLA targets or other goals set for each percentile. The ongoing results should be reported in near real time to help detect immediate issues. The results should also be aggregated over the longer time for statistical purposes.

In the case of latency issues affecting performance, an operator should be able to quickly identify the cause of the bottleneck by examining the latency of each step that each request performs. The performance data must therefore provide a means of correlating performance measures for each step to tie them to a specific request.

Depending on the visualization requirements, it might be useful to generate and store a data cube that contains views of the raw data. This data cube can allow complex ad hoc querying and analysis of the performance information.

Security monitoring

All commercial systems that include sensitive data must implement a security structure. The complexity of the security mechanism is usually a function of the sensitivity of the data. In a system that requires users to be authenticated, you should record:

- All sign-in attempts, whether they fail or succeed.
- All operations performed by—and the details of all resources accessed by—an authenticated user.
- When a user ends a session and signs out.

Monitoring might be able to help detect attacks on the system. For example, a large number of failed sign-in attempts might indicate a brute-force attack. An unexpected surge in requests might be the result of a distributed denial-of-service (DDoS) attack. You must be prepared to monitor all requests to all resources regardless of the source of these requests. A system that has a sign-in vulnerability might accidentally expose resources to the outside world without requiring a user to actually sign in.

Requirements for security monitoring

The most critical aspects of security monitoring should enable an operator to quickly:

- Detect attempted intrusions by an unauthenticated entity.
- Identify attempts by entities to perform operations on data for which they have not been granted access.
- Determine whether the system, or some part of the system, is under attack from outside or inside. (For example, a malicious authenticated user might be attempting to bring the system down.)

To support these requirements, an operator should be notified if:

- One account makes repeated failed sign-in attempts within a specified period.
- One authenticated account repeatedly tries to access a prohibited resource during a specified period.
- A large number of unauthenticated or unauthorized requests occur during a specified period.

The information that's provided to an operator should include the host address of the source for each request. If security violations regularly arise from a particular range of addresses, these hosts might be blocked.

A key part in maintaining the security of a system is being able to quickly detect actions that deviate from the usual pattern. Information such as the number of failed and/or successful sign-in requests can be displayed visually to help detect whether there is a spike in activity at an unusual time. (An example of this activity is users signing in at 3:00 AM and performing a large number of operations when their working day starts at 9:00 AM). This information can also be used to help configure time-based autoscaling. For example, if an operator observes that a large number of users regularly sign in at a particular time of day, the operator can arrange to start additional authentication services to handle the volume of work, and then shut down these additional services when the peak has passed.

Data sources, instrumentation, and data-collection requirements

Security is an all-encompassing aspect of most distributed systems. The pertinent data is likely to be generated at multiple points throughout a system. You should consider adopting a Security Information and Event Management (SIEM) approach to gather the security-related information that results from events raised by the application, network equipment, servers, firewalls, antivirus software, and other intrusion-prevention elements.

Security monitoring can incorporate data from tools that are not part of your application. These tools can include utilities that identify port-scanning activities by external agencies, or network filters that detect attempts to gain unauthenticated access to your application and data.

In all cases, the gathered data must enable an administrator to determine the nature of any attack and take the appropriate countermeasures.

Analyzing security data

A feature of security monitoring is the variety of sources from which the data arises. The different formats and level of detail often require complex analysis of the captured data to tie it together into a coherent thread of information. Apart from the simplest of cases (such as detecting a large number of failed sign-ins, or repeated attempts to gain unauthorized access to critical resources), it might not be possible to perform any complex automated processing of security data. Instead, it might be preferable to write this data, timestamped but otherwise in its original form, to a secure repository to allow for expert manual analysis.

SLA monitoring

Many commercial systems that support paying customers make guarantees about the performance of the system in the form of SLAs. Essentially, SLAs state that the system can handle a defined volume of work within an agreed time frame and without losing critical information. SLA monitoring is concerned with ensuring that the system can meet measurable SLAs.

ⓘ Note

SLA monitoring is closely related to performance monitoring. But whereas performance monitoring is concerned with ensuring that the system functions

optimally, SLA monitoring is governed by a contractual obligation that defines what *optimally* actually means.

SLAs are often defined in terms of:

- Overall system availability. For example, an organization might guarantee that the system will be available for 99.9 percent of the time. This equates to no more than 9 hours of downtime per year, or approximately 10 minutes a week.
- Operational throughput. This aspect is often expressed as one or more high-water marks, such as guaranteeing that the system can support up to 100,000 concurrent user requests or handle 10,000 concurrent business transactions.
- Operational response time. The system might also make guarantees for the rate at which requests are processed. An example is that 99 percent of all business transactions will finish within 2 seconds, and no single transaction will take longer than 10 seconds.

 **Note**

Some contracts for commercial systems might also include SLAs for customer support. An example is that all help-desk requests will elicit a response within five minutes, and that 99 percent of all problems will be fully addressed within 1 working day. Effective **issue tracking** (described later in this section) is key to meeting SLAs such as these.

Requirements for SLA monitoring

At the highest level, an operator should be able to determine at a glance whether the system is meeting the agreed SLAs or not. And if not, the operator should be able to drill down and examine the underlying factors to determine the reasons for substandard performance.

Typical high-level indicators that can be depicted visually include:

- The percentage of service uptime.
- The application throughput (measured in terms of successful transactions and/or operations per second).
- The number of successful/failing application requests.
- The number of application and system faults, exceptions, and warnings.

All of these indicators should be capable of being filtered by a specified period of time.

A cloud application will likely comprise a number of subsystems and components. An operator should be able to select a high-level indicator and see how it's composed from the health of the underlying elements. For example, if the uptime of the overall system falls below an acceptable value, an operator should be able to zoom in and determine which elements are contributing to this failure.

ⓘ Note

System uptime needs to be defined carefully. In a system that uses redundancy to ensure maximum availability, individual instances of elements might fail, but the system can remain functional. System uptime as presented by health monitoring should indicate the aggregate uptime of each element and not necessarily whether the system has actually halted. Additionally, failures might be isolated. So even if a specific system is unavailable, the remainder of the system might remain available, although with decreased functionality. (In an e-commerce system, a failure in the system might prevent a customer from placing orders, but the customer might still be able to browse the product catalog.)

For alerting purposes, the system should be able to raise an event if any of the high-level indicators exceed a specified threshold. The lower-level details of the various factors that compose the high-level indicator should be available as contextual data to the alerting system.

Data sources, instrumentation, and data-collection requirements

The raw data that's required to support SLA monitoring is similar to the raw data that's required for performance monitoring, together with some aspects of health and availability monitoring. (See those sections for more details.) You can capture this data by:

- Performing endpoint monitoring.
- Logging exceptions, faults, and warnings.
- Tracing the execution of user requests.
- Monitoring the availability of any third-party services that the system uses.
- Using performance metrics and counters.

All data must be timed and timestamped.

Analyzing SLA data

The instrumentation data must be aggregated to generate a picture of the overall performance of the system. Aggregated data must also support drill-down to enable examination of the performance of the underlying subsystems. For example, you should be able to:

- Calculate the total number of user requests during a specified period and determine the success and failure rate of these requests.
- Combine the response times of user requests to generate an overall view of system response times.
- Analyze the progress of user requests to break down the overall response time of a request into the response times of the individual work items in that request.
- Determine the overall availability of the system as a percentage of uptime for any specific period.
- Analyze the percentage time availability of the individual components and services in the system. This might involve parsing logs that third-party services have generated.

Many commercial systems are required to report real performance figures against agreed SLAs for a specified period, typically a month. This information can be used to calculate credits or other forms of repayments for customers if the SLAs are not met during that period. You can calculate availability for a service by using the technique described in the section [Analyzing availability data](#).

For internal purposes, an organization might also track the number and nature of incidents that caused services to fail. Learning how to resolve these issues quickly, or eliminate them completely, will help to reduce downtime and meet SLAs.

Auditing

Depending on the nature of the application, there might be statutory or other legal regulations that specify requirements for auditing users' operations and recording all data access. Auditing can provide evidence that links customers to specific requests. Nonrepudiation is an important factor in many e-business systems to help maintain trust between a customer and the organization that's responsible for the application or service.

Requirements for auditing

An analyst must be able to trace the sequence of business operations that users are performing so that you can reconstruct users' actions. This might be necessary simply as a matter of record, or as part of a forensic investigation.

Audit information is highly sensitive. It will likely include data that identifies the users of the system, together with the tasks that they're performing. For this reason, audit information will most likely take the form of reports that are available only to trusted analysts rather than as an interactive system that supports drill-down of graphical operations. An analyst should be able to generate a range of reports. For example, reports might list all users' activities occurring during a specified time frame, detail the chronology of activity for a single user, or list the sequence of operations performed against one or more resources.

Data sources, instrumentation, and data-collection requirements

The primary sources of information for auditing can include:

- The security system that manages user authentication.
- Trace logs that record user activity.
- Security logs that track all identifiable and unidentifiable network requests.

The format of the audit data and the way in which it's stored might be driven by regulatory requirements. For example, it might not be possible to clean the data in any way. (It must be recorded in its original format.) Access to the repository where it's held must be protected to prevent tampering.

Analyzing audit data

An analyst must be able to access the raw data in its entirety, in its original form. Aside from the requirement to generate common audit reports, the tools for analyzing this data are likely to be specialized and kept external to the system.

Usage monitoring

Usage monitoring tracks how the features and components of an application are used. An operator can use the gathered data to:

- Determine which features are heavily used and determine any potential hotspots in the system. High-traffic elements might benefit from functional partitioning or even replication to spread the load more evenly. An operator can also use this information to ascertain which features are infrequently used and are possible candidates for retirement or replacement in a future version of the system.

- Obtain information about the operational events of the system under normal use. For example, in an e-commerce site, you can record the statistical information about the number of transactions and the volume of customers that are responsible for them. This information can be used for capacity planning as the number of customers grows.
- Detect (possibly indirectly) user satisfaction with the performance or functionality of the system. For example, if a large number of customers in an e-commerce system regularly abandon their shopping carts, this might be due to a problem with the checkout functionality.
- Generate billing information. A commercial application or multitenant service might charge customers for the resources that they use.
- Enforce quotas. If a user in a multitenant system exceeds their paid quota of processing time or resource usage during a specified period, their access can be limited or processing can be throttled.

Requirements for usage monitoring

To examine system usage, an operator typically needs to see information that includes:

- The number of requests that are processed by each subsystem and directed to each resource.
- The work that each user is performing.
- The volume of data storage that each user occupies.
- The resources that each user is accessing.

An operator should also be able to generate graphs. For example, a graph might display the most resource-hungry users, or the most frequently accessed resources or system features.

Data sources, instrumentation, and data-collection requirements

Usage tracking can be performed at a relatively high level. It can note the start and end times of each request and the nature of the request (read, write, and so on, depending on the resource in question). You can obtain this information by:

- Tracing user activity.
- Capturing performance counters that measure the utilization for each resource.
- Monitoring the resource consumption by each user.

For metering purposes, you also need to be able to identify which users are responsible for performing which operations, and the resources that these operations use. The gathered information should be detailed enough to enable accurate billing.

Issue tracking

Customers and other users might report issues if unexpected events or behavior occurs in the system. Issue tracking is concerned with managing these issues, associating them with efforts to resolve any underlying problems in the system, and informing customers of possible resolutions.

Requirements for issue tracking

Operators often perform issue tracking by using a separate system that enables them to record and report the details of problems that users report. These details can include the tasks that the user was trying to perform, symptoms of the problem, the sequence of events, and any error or warning messages that were issued.

Data sources, instrumentation, and data-collection requirements

The initial data source for issue-tracking data is the user who reported the issue in the first place. The user might be able to provide additional data such as:

- A crash dump (if the application includes a component that runs on the user's desktop).
- A screen snapshot.
- The date and time when the error occurred, together with any other environmental information such as the user's location.

This information can be used to help the debugging effort and help construct a backlog for future releases of the software.

Analyzing issue-tracking data

Different users might report the same problem. The issue-tracking system should associate common reports.

The progress of the debugging effort should be recorded against each issue report. When the problem is resolved, the customer can be informed of the solution.

If a user reports an issue that has a known solution in the issue-tracking system, the operator should be able to inform the user of the solution immediately.

Tracing operations and debugging software releases

When a user reports an issue, the user is often only aware of the immediate effect that it has on their operations. The user can only report the results of their own experience back to an operator who is responsible for maintaining the system. These experiences are usually just a visible symptom of one or more fundamental problems. In many cases, an analyst will need to dig through the chronology of the underlying operations to establish the root cause of the problem. This process is called *root cause analysis*.

Note

Root cause analysis might uncover inefficiencies in the design of an application. In these situations, it might be possible to rework the affected elements and deploy them as part of a subsequent release. This process requires careful control, and the updated components should be monitored closely.

Requirements for tracing and debugging

For tracing unexpected events and other problems, it's vital that the monitoring data provides enough information to enable an analyst to trace back to the origins of these issues and reconstruct the sequence of events that occurred. This information must be sufficient to enable an analyst to diagnose the root cause of any problems. A developer can then make the necessary modifications to prevent them from recurring.

Data sources, instrumentation, and data-collection requirements

Troubleshooting can involve tracing all the methods (and their parameters) invoked as part of an operation to build up a tree that depicts the logical flow through the system when a customer makes a specific request. Exceptions and warnings that the system generates as a result of this flow need to be captured and logged.

To support debugging, the system can provide hooks that enable an operator to capture state information at crucial points in the system. Or, the system can deliver detailed step-by-step information as selected operations progress. Capturing data at this level of

detail can impose an additional load on the system and should be a temporary process. An operator uses this process mainly when a highly unusual series of events occurs and is difficult to replicate, or when a new release of one or more elements into a system requires careful monitoring to ensure that the elements function as expected.

The monitoring and diagnostics pipeline

Monitoring a large-scale distributed system poses a significant challenge. Each of the scenarios described in the previous section should not necessarily be considered in isolation. There is likely to be a significant overlap in the monitoring and diagnostic data that's required for each situation, although this data might need to be processed and presented in different ways. For these reasons, you should take a holistic view of monitoring and diagnostics.

You can envisage the entire monitoring and diagnostics process as a pipeline that comprises the stages shown in Figure 1.

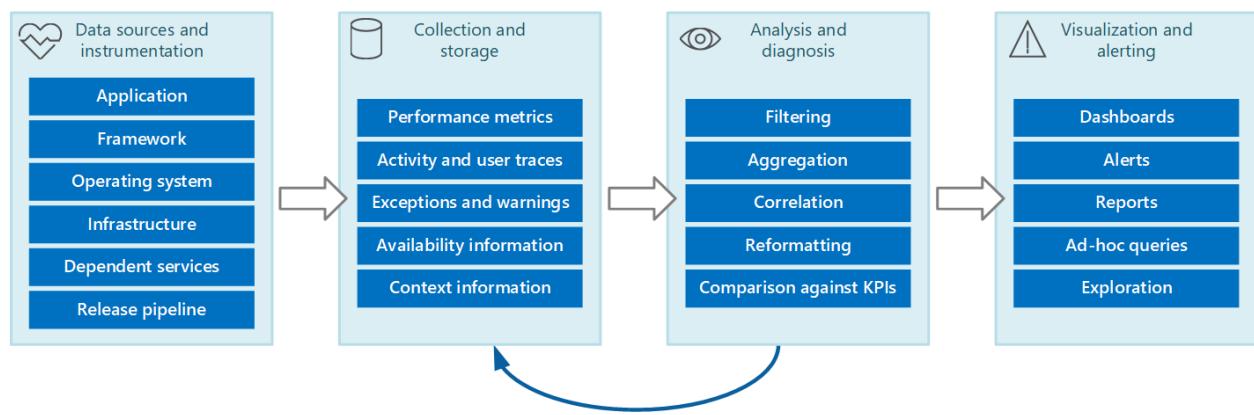


Figure 1 - The stages in the monitoring and diagnostics pipeline.

Figure 1 highlights how the data for monitoring and diagnostics can come from a variety of data sources. The instrumentation and collection stages are concerned with identifying the sources from where the data needs to be captured, determining which data to capture, how to capture it, and how to format this data so that it can be easily examined. The analysis/diagnosis stage takes the raw data and uses it to generate meaningful information that an operator can use to determine the state of the system. The operator can use this information to make decisions about possible actions to take, and then feed the results back into the instrumentation and collection stages. The visualization/alerting stage presents a consumable view of the system state. It can display information in near real time by using a series of dashboards. And it can generate reports, graphs, and charts to provide a historical view of the data that can help identify long-term trends. If information indicates that a KPI is likely to exceed acceptable bounds, this stage can also trigger an alert to an operator. In some cases, an

alert can also be used to trigger an automated process that attempts to take corrective actions, such as autoscaling.

Note that these steps constitute a continuous-flow process where the stages are happening in parallel. Ideally, all the phases should be dynamically configurable. At some points, especially when a system has been newly deployed or is experiencing problems, it might be necessary to gather extended data on a more frequent basis. At other times, it should be possible to revert to capturing a base level of essential information to verify that the system is functioning properly.

Additionally, the entire monitoring process should be considered a live, ongoing solution that's subject to fine-tuning and improvements as a result of feedback. For example, you might start with measuring many factors to determine system health. Analysis over time might lead to a refinement as you discard measures that aren't relevant, enabling you to more precisely focus on the data that you need while minimizing background noise.

Sources of monitoring and diagnostic data

The information that the monitoring process uses can come from several sources, as illustrated in Figure 1. At the application level, information comes from trace logs incorporated into the code of the system. Developers should follow a standard approach for tracking the flow of control through their code. For example, an entry to a method can emit a trace message that specifies the name of the method, the current time, the value of each parameter, and any other pertinent information. Recording the entry and exit times can also prove useful.

You should log all exceptions and warnings, and ensure that you retain a full trace of any nested exceptions and warnings. Ideally, you should also capture information that identifies the user who is running the code, together with activity correlation information (to track requests as they pass through the system). And you should log attempts to access all resources such as message queues, databases, files, and other dependent services. This information can be used for metering and auditing purposes.

Many applications use libraries and frameworks to perform common tasks such as accessing a data store or communicating over a network. These frameworks might be configurable to provide their own trace messages and raw diagnostic information, such as transaction rates and data transmission successes and failures.

① Note

Many modern frameworks automatically publish performance and trace events. Capturing this information is simply a matter of providing a means to retrieve and store it where it can be processed and analyzed.

The operating system where the application is running can be a source of low-level system-wide information, such as performance counters that indicate I/O rates, memory utilization, and CPU usage. Operating system errors (such as the failure to open a file correctly) might also be reported.

You should also consider the underlying infrastructure and components on which your system runs. Virtual machines, virtual networks, and storage services can all be sources of important infrastructure-level performance counters and other diagnostic data.

If your application uses other external services, such as a web server or database management system, these services might publish their own trace information, logs, and performance counters. Examples include SQL Server Dynamic Management Views for tracking operations performed against a SQL Server database, and IIS trace logs for recording requests made to a web server.

As the components of a system are modified and new versions are deployed, it's important to be able to attribute issues, events, and metrics to each version. This information should be tied back to the release pipeline so that problems with a specific version of a component can be tracked quickly and rectified.

Security issues might occur at any point in the system. For example, a user might attempt to sign in with an invalid user ID or password. An authenticated user might try to obtain unauthorized access to a resource. Or a user might provide an invalid or outdated key to access encrypted information. Security-related information for successful and failing requests should always be logged.

The section [Instrumenting an application](#) contains more guidance on the information that you should capture. But you can use a variety of strategies to gather this information:

- **Application/system monitoring.** This strategy uses internal sources within the application, application frameworks, operating system, and infrastructure. The application code can generate its own monitoring data at notable points during the lifecycle of a client request. The application can include tracing statements that might be selectively enabled or disabled as circumstances dictate. It might also be possible to inject diagnostics dynamically by using a diagnostics framework. These frameworks typically provide plug-ins that can attach to various instrumentation points in your code and capture trace data at these points.

Additionally, your code and/or the underlying infrastructure might raise events at critical points. Monitoring agents that are configured to listen for these events can record the event information.

- **Real user monitoring.** This approach records the interactions between a user and the application and observes the flow of each request and response. This information can have a two-fold purpose: it can be used for metering usage by each user, and it can be used to determine whether users are receiving a suitable quality of service (for example, fast response times, low latency, and minimal errors). You can use the captured data to identify areas of concern where failures occur most often. You can also use the data to identify elements where the system slows down, possibly due to hotspots in the application or some other form of bottleneck. If you implement this approach carefully, it might be possible to reconstruct users' flows through the application for debugging and testing purposes.

 **Important**

You should consider the data that's captured by monitoring real users to be highly sensitive because it might include confidential material. If you save captured data, store it securely. If you want to use the data for performance monitoring or debugging purposes, strip out all personally identifiable information first.

- **Synthetic user monitoring.** In this approach, you write your own test client that simulates a user and performs a configurable but typical series of operations. You can track the performance of the test client to help determine the state of the system. You can also use multiple instances of the test client as part of a load-testing operation to establish how the system responds under stress, and what sort of monitoring output is generated under these conditions.

 **Note**

You can implement real and synthetic user monitoring by including code that traces and times the execution of method calls and other critical parts of an application.

- **Profiling.** This approach is primarily targeted at monitoring and improving application performance. Rather than operating at the functional level of real and synthetic user monitoring, it captures lower-level information as the application

runs. You can implement profiling by using periodic sampling of the execution state of an application (determining which piece of code that the application is running at a given point in time). You can also use instrumentation that inserts probes into the code at important junctures (such as the start and end of a method call) and records which methods were invoked, at what time, and how long each call took. You can then analyze this data to determine which parts of the application might cause performance problems.

- **Endpoint monitoring.** This technique uses one or more diagnostic endpoints that the application exposes specifically to enable monitoring. An endpoint provides a pathway into the application code and can return information about the health of the system. Different endpoints can focus on various aspects of the functionality. You can write your own diagnostics client that sends periodic requests to these endpoints and assimilate the responses. For more information, see the [Health Endpoint Monitoring pattern](#).

For maximum coverage, you should use a combination of these techniques.

Instrumenting an application

Instrumentation is a critical part of the monitoring process. You can make meaningful decisions about the performance and health of a system only if you first capture the data that enables you to make these decisions. The information that you gather by using instrumentation should be sufficient to enable you to assess performance, diagnose problems, and make decisions without requiring you to sign in to a remote production server to perform tracing (and debugging) manually. Instrumentation data typically comprises metrics and information that's written to trace logs.

The contents of a trace log can be the result of textual data that's written by the application or binary data that's created as the result of a trace event (if the application is using Event Tracing for Windows--ETW). They can also be generated from system logs that record events arising from parts of the infrastructure, such as a web server. Textual log messages are often designed to be human-readable, but they should also be written in a format that enables an automated system to parse them easily.

You should also categorize logs. Don't write all trace data to a single log, but use separate logs to record the trace output from different operational aspects of the system. You can then quickly filter log messages by reading from the appropriate log rather than having to process a single lengthy file. Never write information that has different security requirements (such as audit information and debugging data) to the same log.

Note

A log might be implemented as a file on the file system, or it might be held in some other format, such as a blob in blob storage. Log information might also be held in more structured storage, such as rows in a table.

Metrics will generally be a measure or count of some aspect or resource in the system at a specific time, with one or more associated tags or dimensions (sometimes called a *sample*). A single instance of a metric is usually not useful in isolation. Instead, metrics have to be captured over time. The key issue to consider is which metrics you should record and how frequently. Generating data for metrics too often can impose a significant additional load on the system, whereas capturing metrics infrequently might cause you to miss the circumstances that lead to a significant event. The considerations will vary from metric to metric. For example, CPU utilization on a server might vary significantly from second to second, but high utilization becomes an issue only if it's long-lived over a number of minutes.

Information for correlating data

You can easily monitor individual system-level performance counters, capture metrics for resources, and obtain application trace information from various log files. But some forms of monitoring require the analysis and diagnostics stage in the monitoring pipeline to correlate the data that's retrieved from several sources. This data might take several forms in the raw data, and the analysis process must be provided with sufficient instrumentation data to be able to map these different forms. For example, at the application framework level, a task might be identified by a thread ID. Within an application, the same work might be associated with the user ID for the user who is performing that task.

Also, there's unlikely to be a 1:1 mapping between threads and user requests, because asynchronous operations might reuse the same threads to perform operations on behalf of more than one user. To complicate matters further, a single request might be handled by more than one thread as execution flows through the system. If possible, associate each request with a unique activity ID that's propagated through the system as part of the request context. (The technique for generating and including activity IDs in trace information depends on the technology that's used to capture the trace data.)

All monitoring data should be timestamped in the same way. For consistency, record all dates and times by using Coordinated Universal Time. This will help you more easily trace sequences of events.

Note

Computers operating in different time zones and networks might not be synchronized. Don't depend on using timestamps alone for correlating instrumentation data that spans multiple machines.

Information to include in the instrumentation data

Consider the following points when you're deciding which instrumentation data you need to collect:

- Make sure that information captured by trace events is machine and human readable. Adopt well-defined schemas for this information to facilitate automated processing of log data across systems, and to provide consistency to operations and engineering staff reading the logs. Include environmental information, such as the deployment environment, the machine on which the process is running, the details of the process, and the call stack.
- Enable profiling only when necessary because it can impose a significant overhead on the system. Profiling by using instrumentation records an event (such as a method call) every time it occurs, whereas sampling records only selected events. The selection can be time-based (once every n seconds), or frequency-based (once every n requests). If events occur very frequently, profiling by instrumentation might cause too much of a burden and itself affect overall performance. In this case, the sampling approach might be preferable. However, if the frequency of events is low, sampling might miss them. In this case, instrumentation might be the better approach.
- Provide sufficient context to enable a developer or administrator to determine the source of each request. This might include some form of activity ID that identifies a specific instance of a request. It might also include information that can be used to correlate this activity with the computational work performed and the resources used. Note that this work might cross process and machine boundaries. For metering, the context should also include (either directly or indirectly via other correlated information) a reference to the customer who caused the request to be made. This context provides valuable information about the application state at the time that the monitoring data was captured.
- Record all requests, and the locations or regions from which these requests are made. This information can assist in determining whether there are any location-

specific hotspots. This information can also be useful in determining whether to repartition an application or the data that it uses.

- Record and capture the details of exceptions carefully. Often, critical debug information is lost as a result of poor exception handling. Capture the full details of exceptions that the application throws, including any inner exceptions and other context information. Include the call stack if possible.
- Be consistent in the data that the different elements of your application capture, because this can assist in analyzing events and correlating them with user requests. Consider using a comprehensive and configurable logging package to gather information, rather than depending on developers to adopt the same approach as they implement different parts of the system. Gather data from key performance counters, such as the volume of I/O being performed, network utilization, number of requests, memory use, and CPU utilization. Some infrastructure services might provide their own specific performance counters, such as the number of connections to a database, the rate at which transactions are being performed, and the number of transactions that succeed or fail. Applications might also define their own specific performance counters.
- Log all calls made to external services, such as database systems, web services, or other system-level services that are part of the infrastructure. Record information about the time taken to perform each call, and the success or failure of the call. If possible, capture information about all retry attempts and failures for any transient errors that occur.

Ensuring compatibility with telemetry systems

In many cases, the information that instrumentation produces is generated as a series of events and passed to a separate telemetry system for processing and analysis. A telemetry system is typically independent of any specific application or technology, but it expects information to follow a specific format that's usually defined by a schema. The schema effectively specifies a contract that defines the data fields and types that the telemetry system can ingest. The schema should be generalized to allow for data arriving from a range of platforms and devices.

A common schema should include fields that are common to all instrumentation events, such as the event name, the event time, the IP address of the sender, and the details that are required for correlating with other events (such as a user ID, a device ID, and an application ID). Remember that any number of devices might raise events, so the schema should not depend on the device type. Additionally, various devices might raise

events for the same application; the application might support roaming or some other form of cross-device distribution.

The schema might also include domain fields that are relevant to a particular scenario that's common across different applications. This might be information about exceptions, application start and end events, and success and/or failure of web service API calls. All applications that use the same set of domain fields should emit the same set of events, enabling a set of common reports and analytics to be built.

Finally, a schema might contain custom fields for capturing the details of application-specific events.

Best practices for instrumenting applications

The following list summarizes best practices for instrumenting a distributed application running in the cloud.

- Make logs easy to read and easy to parse. Use structured logging where possible. Be concise and descriptive in log messages.
- In all logs, identify the source and provide context and timing information as each log record is written.
- Use the same time zone and format for all timestamps. This will help to correlate events for operations that span hardware and services running in different geographic regions.
- Categorize logs and write messages to the appropriate log file.
- Do not disclose sensitive information about the system or personal information about users. Scrub this information before it's logged, but ensure that the relevant details are retained. For example, remove the ID and password from any database connection strings, but write the remaining information to the log so that an analyst can determine that the system is accessing the correct database. Log all critical exceptions, but enable the administrator to turn logging on and off for lower levels of exceptions and warnings. Also, capture and log all retry logic information. This data can be useful in monitoring the transient health of the system.
- Trace out of process calls, such as requests to external web services or databases.
- Don't mix log messages with different security requirements in the same log file. For example, don't write debug and audit information to the same log.

- With the exception of auditing events, make sure that all logging calls are fire-and-forget operations that do not block the progress of business operations. Auditing events are exceptional because they are critical to the business and can be classified as a fundamental part of business operations.
- Make sure that logging is extensible and does not have any direct dependencies on a concrete target. For example, rather than writing information by using *System.Diagnostics.Trace*, define an abstract interface (such as *ILogger*) that exposes logging methods and that can be implemented through any appropriate means.
- Make sure that all logging is fail-safe and never triggers any cascading errors. Logging must not throw any exceptions.
- Treat instrumentation as an ongoing iterative process and review logs regularly, not just when there is a problem.

Collecting and storing data

The collection stage of the monitoring process is concerned with retrieving the information that instrumentation generates, formatting this data to make it easier for the analysis/diagnosis stage to consume, and saving the transformed data in reliable storage. The instrumentation data that you gather from different parts of a distributed system can be held in a variety of locations and with varying formats. For example, your application code might generate trace log files and generate application event log data, whereas performance counters that monitor key aspects of the infrastructure that your application uses can be captured through other technologies. Any third-party components and services that your application uses might provide instrumentation information in different formats, by using separate trace files, blob storage, or even a custom data store.

Data collection is often performed through a collection service that can run autonomously from the application that generates the instrumentation data. Figure 2 illustrates an example of this architecture, highlighting the instrumentation data-collection subsystem.

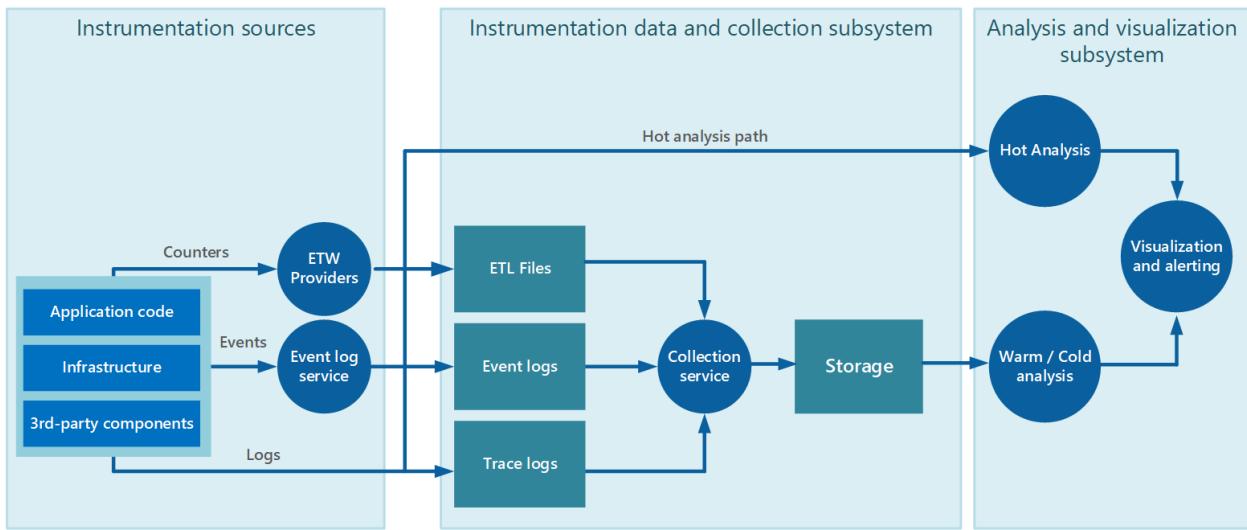


Figure 2 - Collecting instrumentation data.

Note that this is a simplified view. The collection service is not necessarily a single process and might comprise many constituent parts running on different machines, as described in the following sections. Additionally, if the analysis of some telemetry data must be performed quickly (hot analysis, as described in the section [Supporting hot, warm, and cold analysis](#) later in this document), local components that operate outside the collection service might perform the analysis tasks immediately. Figure 2 depicts this situation for selected events. After analytical processing, the results can be sent directly to the visualization and alerting subsystem. Data that's subjected to warm or cold analysis is held in storage while it awaits processing.

For Azure applications and services, Azure Diagnostics provides one possible solution for capturing data. Azure Diagnostics gathers data from the following sources for each compute node, aggregates it, and then uploads it to Azure Storage:

- IIS logs
- IIS Failed Request logs
- Windows event logs
- Performance counters
- Crash dumps
- Azure Diagnostics infrastructure logs
- Custom error logs
- .NET EventSource
- Manifest-based ETW

For more information, see the article [Azure: Telemetry Basics and Troubleshooting](#).

Strategies for collecting instrumentation data

Considering the elastic nature of the cloud, and to avoid the necessity of manually retrieving telemetry data from every node in the system, you should arrange for the data to be transferred to a central location and consolidated. In a system that spans multiple datacenters, it might be useful to first collect, consolidate, and store data on a region-by-region basis, and then aggregate the regional data into a single central system.

To optimize the use of bandwidth, you can elect to transfer less urgent data in chunks, as batches. However, the data must not be delayed indefinitely, especially if it contains time-sensitive information.

Pulling and pushing instrumentation data

The instrumentation data-collection subsystem can actively retrieve instrumentation data from the various logs and other sources for each instance of the application (the *pull model*). Or, it can act as a passive receiver that waits for the data to be sent from the components that constitute each instance of the application (the *push model*).

One approach to implementing the pull model is to use monitoring agents that run locally with each instance of the application. A monitoring agent is a separate process that periodically retrieves (pulls) telemetry data collected at the local node and writes this information directly to centralized storage that all instances of the application share. This is the mechanism that Azure Diagnostics implements. Each instance of an Azure web or worker role can be configured to capture diagnostic and other trace information that's stored locally. The monitoring agent that runs alongside each instance copies the specified data to Azure Storage. The article [Enabling Diagnostics in Azure Cloud Services and Virtual Machines](#) provides more details on this process. Some elements, such as IIS logs, crash dumps, and custom error logs, are written to blob storage. Data from the Windows event log, ETW events, and performance counters is recorded in table storage. Figure 3 illustrates this mechanism.

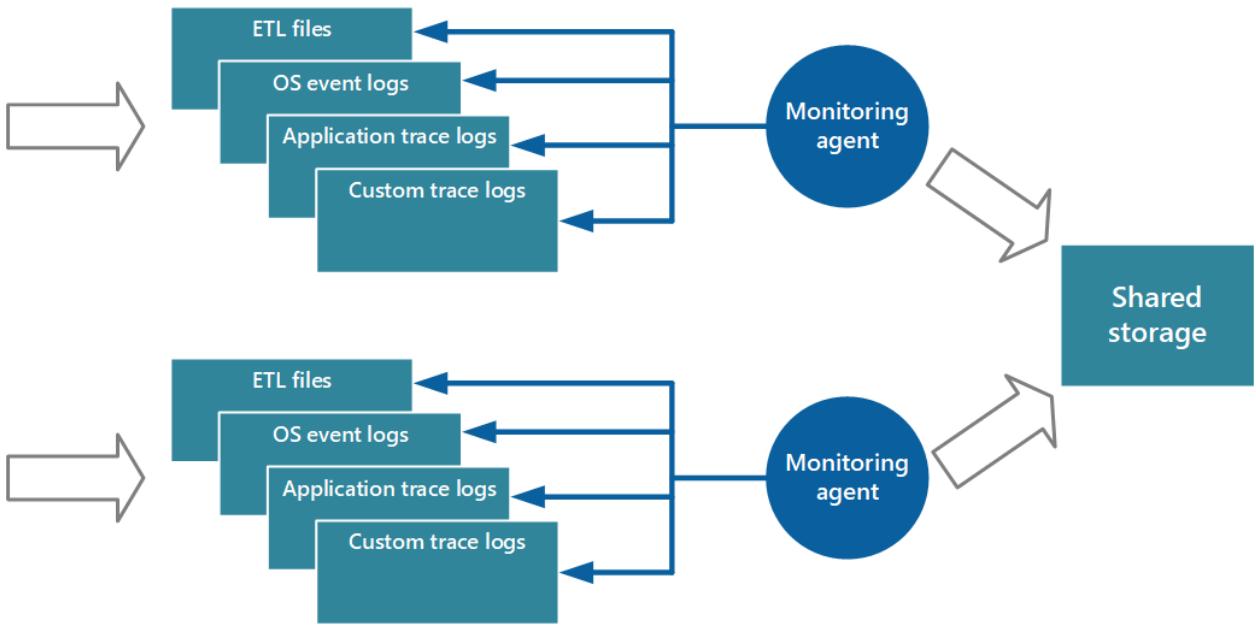


Figure 3 - Using a monitoring agent to pull information and write to shared storage.

① Note

Using a monitoring agent is ideally suited to capturing instrumentation data that's naturally pulled from a data source. An example is information from SQL Server Dynamic Management Views or the length of an Azure Service Bus queue.

It's feasible to use the approach just described to store telemetry data for a small-scale application running on a limited number of nodes in a single location. However, a complex, highly scalable, global cloud application might generate huge volumes of data from hundreds of web and worker roles, database shards, and other services. This flood of data can easily overwhelm the I/O bandwidth available with a single, central location. Therefore, your telemetry solution must be scalable to prevent it from acting as a bottleneck as the system expands. Ideally, your solution should incorporate a degree of redundancy to reduce the risks of losing important monitoring information (such as auditing or billing data) if part of the system fails.

To address these issues, you can implement queuing, as shown in Figure 4. In this architecture, the local monitoring agent (if it can be configured appropriately) or custom data-collection service (if not) posts data to a queue. A separate process running asynchronously (the storage writing service in Figure 4) takes the data in this queue and writes it to shared storage. A message queue is suitable for this scenario because it provides "at least once" semantics that help ensure that queued data will not be lost after it's posted. You can implement the storage writing service by using a separate worker role.

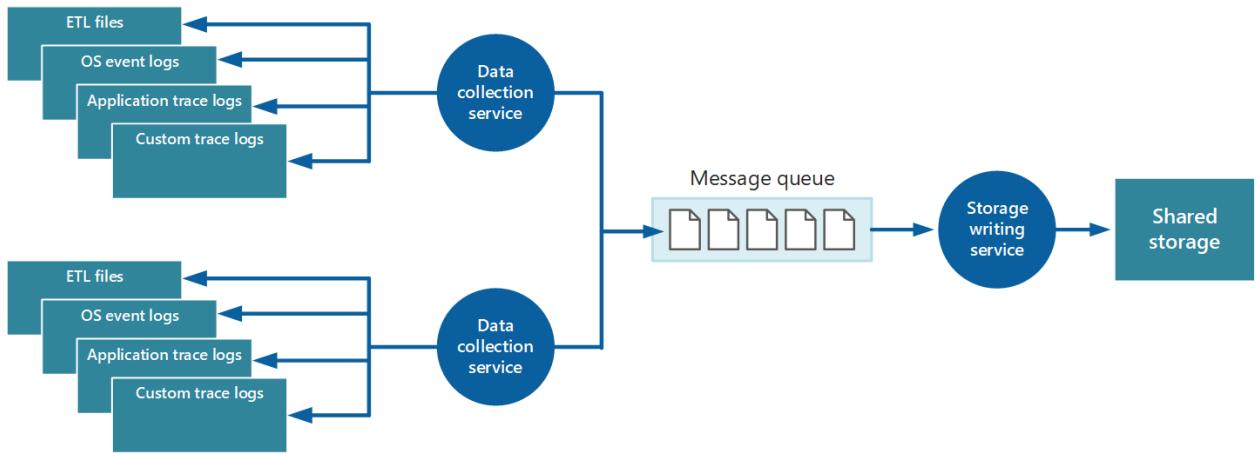


Figure 4 - Using a queue to buffer instrumentation data.

The local data-collection service can add data to a queue immediately after it's received. The queue acts as a buffer, and the storage writing service can retrieve and write the data at its own pace. By default, a queue operates on a first-in, first-out basis. But you can prioritize messages to accelerate them through the queue if they contain data that must be handled more quickly. For more information, see the [Priority Queue pattern](#). Alternatively, you can use different channels (such as Service Bus topics) to direct data to different destinations depending on the form of analytical processing that's required.

For scalability, you can run multiple instances of the storage writing service. If there is a high volume of events, you can use an event hub to dispatch the data to different compute resources for processing and storage.

Consolidating instrumentation data

The instrumentation data that the data-collection service retrieves from a single instance of an application gives a localized view of the health and performance of that instance. To assess the overall health of the system, it's necessary to consolidate some aspects of the data in the local views. You can perform this after the data has been stored, but in some cases, you can also achieve it as the data is collected. Rather than being written directly to shared storage, the instrumentation data can pass through a separate data consolidation service that combines data and acts as a filter and cleanup process. For example, instrumentation data that includes the same correlation information such as an activity ID can be amalgamated. (It's possible that a user starts performing a business operation on one node and then gets transferred to another node in the event of node failure, or depending on how load balancing is configured.) This process can also detect and remove any duplicated data (always a possibility if the telemetry service uses message queues to push instrumentation data out to storage). Figure 5 illustrates an example of this structure.

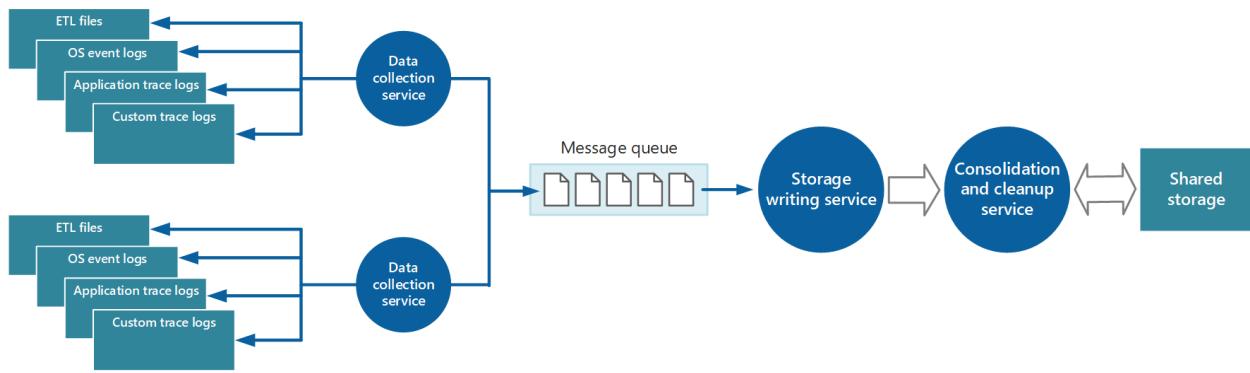


Figure 5 - Using a separate service to consolidate and clean up instrumentation data.

Storing instrumentation data

The previous discussions have depicted a rather simplistic view of the way in which instrumentation data is stored. In reality, it can make sense to store the different types of information by using technologies that are most appropriate to the way in which each type is likely to be used.

For example, Azure blob and table storage have some similarities in the way in which they're accessed. But they have limitations in the operations that you can perform by using them, and the granularity of the data that they hold is quite different. If you need to perform more analytical operations or require full-text search capabilities on the data, it might be more appropriate to use data storage that provides capabilities that are optimized for specific types of queries and data access. For example:

- Performance counter data can be stored in a SQL database to enable ad hoc analysis.
- Trace logs might be better stored in Azure Cosmos DB.
- Security information can be written to HDFS.
- Information that requires full-text search can be stored through Elasticsearch (which can also speed searches by using rich indexing).

You can implement an additional service that periodically retrieves the data from shared storage, partitions and filters the data according to its purpose, and then writes it to an appropriate set of data stores as shown in Figure 6. An alternative approach is to include this functionality in the consolidation and cleanup process and write the data directly to these stores as it's retrieved rather than saving it in an intermediate shared storage area. Each approach has its advantages and disadvantages. Implementing a separate partitioning service lessens the load on the consolidation and cleanup service, and it enables at least some of the partitioned data to be regenerated if necessary (depending on how much data is retained in shared storage). However, it consumes additional resources. Also, there might be a delay between the receipt of instrumentation data

from each application instance and the conversion of this data into actionable information.

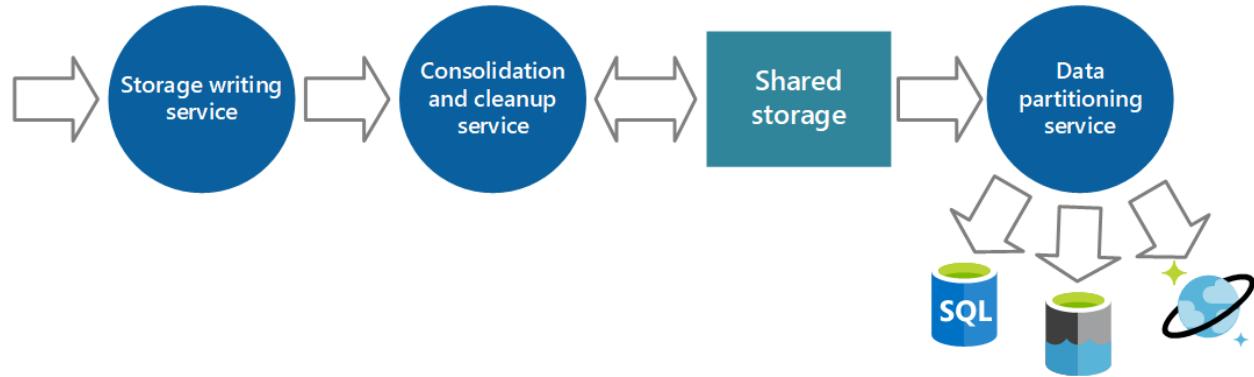


Figure 6 - Partitioning data according to analytical and storage requirements.

The same instrumentation data might be required for more than one purpose. For example, performance counters can be used to provide a historical view of system performance over time. This information might be combined with other usage data to generate customer billing information. In these situations, the same data might be sent to more than one destination, such as a document database that can act as a long-term store for holding billing information, and a multidimensional store for handling complex performance analytics.

You should also consider how urgently the data is required. Data that provides information for alerting must be accessed quickly, so it should be held in fast data storage and indexed or structured to optimize the queries that the alerting system performs. In some cases, it might be necessary for the telemetry service that gathers the data on each node to format and save data locally so that a local instance of the alerting system can quickly notify you of any issues. The same data can be dispatched to the storage writing service shown in the previous diagrams and stored centrally if it's also required for other purposes.

Information that's used for more considered analysis, for reporting, and for spotting historical trends is less urgent and can be stored in a manner that supports data mining and ad hoc queries. For more information, see the section [Supporting hot, warm, and cold analysis](#) later in this document.

Log rotation and data retention

Instrumentation can generate considerable volumes of data. This data can be held in several places, starting with the raw log files, trace files, and other information captured at each node to the consolidated, cleaned, and partitioned view of this data held in shared storage. In some cases, after the data has been processed and transferred, the original raw source data can be removed from each node. In other cases, it might be

necessary or simply useful to save the raw information. For example, data that's generated for debugging purposes might be best left available in its raw form but can then be discarded quickly after any bugs have been rectified.

Performance data often has a longer life so that it can be used for spotting performance trends and for capacity planning. The consolidated view of this data is usually kept online for a finite period to enable fast access. After that, it can be archived or discarded. Data gathered for metering and billing customers might need to be saved indefinitely. Additionally, regulatory requirements might dictate that information collected for auditing and security purposes also needs to be archived and saved. This data is also sensitive and might need to be encrypted or otherwise protected to prevent tampering. You should never record users' passwords or other information that might be used to commit identity fraud. Such details should be scrubbed from the data before it's stored.

Down-sampling

It's useful to store historical data so you can spot long-term trends. Rather than saving old data in its entirety, it might be possible to down-sample the data to reduce its resolution and save storage costs. As an example, rather than saving minute-by-minute performance indicators, you can consolidate data that's more than a month old to form an hour-by-hour view.

Best practices for collecting and storing logging information

The following list summarizes best practices for capturing and storing logging information:

- The monitoring agent or data-collection service should run as an out-of-process service and should be simple to deploy.
- All output from the monitoring agent or data-collection service should be an agnostic format that's independent of the machine, operating system, or network protocol. For example, emit information in a self-describing format such as JSON, MessagePack, or Protobuf rather than ETL/ETW. Using a standard format enables the system to construct processing pipelines; components that read, transform, and send data in the agreed format can be easily integrated.
- The monitoring and data-collection process must be fail-safe and must not trigger any cascading error conditions.

- In the event of a transient failure in sending information to a data sink, the monitoring agent or data-collection service should be prepared to reorder telemetry data so that the newest information is sent first. (The monitoring agent/data-collection service might elect to drop the older data, or save it locally and transmit it later to catch up, at its own discretion.)

Analyzing data and diagnosing issues

An important part of the monitoring and diagnostics process is analyzing the gathered data to obtain a picture of the overall well-being of the system. You should have defined your own KPIs and performance metrics, and it's important to understand how you can structure the data that has been gathered to meet your analysis requirements. It's also important to understand how the data that's captured in different metrics and log files is correlated, because this information can be key to tracking a sequence of events and help diagnose problems that arise.

As described in the section [Consolidating instrumentation data](#), the data for each part of the system is typically captured locally, but it generally needs to be combined with data generated at other sites that participate in the system. This information requires careful correlation to ensure that data is combined accurately. For example, the usage data for an operation might span a node that hosts a website to which a user connects, a node that runs a separate service accessed as part of this operation, and data storage held on another node. This information needs to be tied together to provide an overall view of the resource and processing usage for the operation. Some preprocessing and filtering of data might occur on the node on which the data is captured, whereas aggregation and formatting are more likely to occur on a central node.

Supporting hot, warm, and cold analysis

Analyzing and reformatting data for visualization, reporting, and alerting purposes can be a complex process that consumes its own set of resources. Some forms of monitoring are time-critical and require immediate analysis of data to be effective. This is known as *hot analysis*. Examples include the analyses that are required for alerting and some aspects of security monitoring (such as detecting an attack on the system). Data that's required for these purposes must be quickly available and structured for efficient processing. In some cases, it might be necessary to move the analysis processing to the individual nodes where the data is held.

Other forms of analysis are less time-critical and might require some computation and aggregation after the raw data has been received. This is called *warm analysis*. Performance analysis often falls into this category. In this case, an isolated, single

performance event is unlikely to be statistically significant. (It might be caused by a sudden spike or glitch.) The data from a series of events should provide a more reliable picture of system performance.

Warm analysis can also be used to help diagnose health issues. A health event is typically processed through hot analysis and can raise an alert immediately. An operator should be able to drill into the reasons for the health event by examining the data from the warm path. This data should contain information about the events leading up to the issue that caused the health event.

Some types of monitoring generate more long-term data. This analysis can be performed at a later date, possibly according to a predefined schedule. In some cases, the analysis might need to perform complex filtering of large volumes of data captured over a period of time. This is called *cold analysis*. The key requirement is that the data is stored safely after it has been captured. For example, usage monitoring and auditing require an accurate picture of the state of the system at regular points in time, but this state information does not have to be available for processing immediately after it has been gathered.

An operator can also use cold analysis to provide the data for predictive health analysis. The operator can gather historical information over a specified period and use it in conjunction with the current health data (retrieved from the hot path) to spot trends that might soon cause health issues. In these cases, it might be necessary to raise an alert so that corrective action can be taken.

Correlating data

The data that instrumentation captures can provide a snapshot of the system state, but the purpose of analysis is to make this data actionable. For example:

- What has caused an intense I/O loading at the system level at a specific time?
- Is it the result of a large number of database operations?
- Is this reflected in the database response times, the number of transactions per second, and application response times at the same juncture?

If so, one remedial action that might reduce the load might be to shard the data over more servers. In addition, exceptions can arise as a result of a fault in any level of the system. An exception in one level often triggers another fault in the level above.

For these reasons, you need to be able to correlate the different types of monitoring data at each level to produce an overall view of the state of the system and the applications that are running on it. You can then use this information to make decisions

about whether the system is functioning acceptably or not, and determine what can be done to improve the quality of the system.

As described in the section [Information for correlating data](#), you must ensure that the raw instrumentation data includes sufficient context and activity ID information to support the required aggregations for correlating events. Additionally, this data might be held in different formats, and it might be necessary to parse this information to convert it into a standardized format for analysis.

Troubleshooting and diagnosing issues

Diagnosis requires the ability to determine the cause of faults or unexpected behavior, including performing root cause analysis. The information that's required typically includes:

- Detailed information from event logs and traces, either for the entire system or for a specified subsystem during a specified time window.
- Complete stack traces resulting from exceptions and faults of any specified level that occur within the system or a specified subsystem during a specified period.
- Crash dumps for any failed processes either anywhere in the system or for a specified subsystem during a specified time window.
- Activity logs recording the operations that are performed either by all users or for selected users during a specified period.

Analyzing data for troubleshooting purposes often requires a deep technical understanding of the system architecture and the various components that compose the solution. As a result, a large degree of manual intervention is often required to interpret the data, establish the cause of problems, and recommend an appropriate strategy to correct them. It might be appropriate simply to store a copy of this information in its original format and make it available for cold analysis by an expert.

Visualizing data and raising alerts

An important aspect of any monitoring system is the ability to present the data in such a way that an operator can quickly spot any trends or problems. Also important is the ability to quickly inform an operator if a significant event has occurred that might require attention.

Data presentation can take several forms, including visualization by using dashboards, alerting, and reporting.

Visualization by using dashboards

The most common way to visualize data is to use dashboards that can display information as a series of charts, graphs, or some other illustration. These items can be parameterized, and an analyst should be able to select the important parameters (such as the time period) for any specific situation.

Dashboards can be organized hierarchically. Top-level dashboards can give an overall view of each aspect of the system but enable an operator to drill down to the details. For example, a dashboard that depicts the overall disk I/O for the system should allow an analyst to view the I/O rates for each individual disk to ascertain whether one or more specific devices account for a disproportionate volume of traffic. Ideally, the dashboard should also display related information, such as the source of each request (the user or activity) that's generating this I/O. This information can then be used to determine whether (and how) to spread the load more evenly across devices, and whether the system would perform better if more devices were added.

A dashboard might also use color-coding or some other visual cues to indicate values that appear anomalous or that are outside an expected range. Using the previous example:

- A disk with an I/O rate that's approaching its maximum capacity over an extended period (a hot disk) can be highlighted in red.
- A disk with an I/O rate that periodically runs at its maximum limit over short periods (a warm disk) can be highlighted in yellow.
- A disk that's exhibiting normal usage can be displayed in green.

Note that for a dashboard system to work effectively, it must have the raw data to work with. If you are building your own dashboard system, or using a dashboard developed by another organization, you must understand which instrumentation data you need to collect, at what levels of granularity, and how it should be formatted for the dashboard to consume.

A good dashboard does not only display information, it also enables an analyst to pose ad hoc questions about that information. Some systems provide management tools that an operator can use to perform these tasks and explore the underlying data. Alternatively, depending on the repository that's used to hold this information, it might be possible to query this data directly, or import it into tools such as Microsoft Excel for further analysis and reporting.

① 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.

Next steps

- [Azure Monitor overview](#)
- [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#)
- [Overview of alerts in Microsoft Azure](#)
- [View service health notifications by using the Azure portal](#)

- [What is Application Insights?](#)
- [Performance diagnostics for Azure virtual machines](#)
- [Download and install SQL Server Data Tools \(SSDT\) for Visual Studio](#)

Related resources

- [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.

Retry guidance for Azure services

Article • 06/08/2023

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 most Azure services, and includes information to help you use, adapt, or extend the retry mechanism for that service.

For general guidance on handling transient faults, and retrying connections and operations against services and resources, see [Retry guidance](#).

The following table summarizes the retry features for the Azure services described in this guidance.

Service	Retry capabilities	Policy configuration	Scope	Telemetry features
Azure Active Directory	Native in MSAL library	Embedded into MSAL library	Internal	None
Azure Cosmos DB	Native in service	Non-configurable	Global	TraceSource
Data Lake Store	Native in client	Non-configurable	Individual operations	None
Event Hubs	Native in client	Programmatic	Client	None
IoT Hub	Native in client SDK	Programmatic	Client	None
Azure Cache for Redis	Native in client	Programmatic	Client	TextWriter
Search	Native in client	Programmatic	Client	ETW or Custom
Service Bus	Native in client	Programmatic	Namespace Manager, Messaging Factory, and Client	ETW
Service Fabric	Native in client	Programmatic	Client	None

Service	Retry capabilities	Policy configuration	Scope	Telemetry features
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 to 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's a built-in retry mechanism for Azure Active Directory in the [Microsoft Authentication Library \(MSAL\)](#). To avoid unexpected lockouts, we recommend that third-

party libraries and application code do **not** retry failed connections, but allow MSAL to handle retries.

Retry usage guidance

Consider the following guidelines when using Azure Active Directory:

- When possible, use the MSAL library and the built-in support for retries.
- If you're 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. Don't retry for any other errors.
- For 429 errors, only retry after the time indicated in the **Retry-After** header.
- For 5xx errors, use exponential back-off, with the first retry at least 5 seconds after the response.
- Don't retry on errors other than 429 and 5xx.

Next steps

- [Microsoft Authentication Library \(MSAL\)](#)

Azure Cosmos DB

Azure Cosmos DB is a fully managed multi-model database that supports schemaless JSON data. It offers configurable and reliable performance, native JavaScript transactional processing, and is built for the cloud with elastic scale.

Retry mechanism

The Azure Cosmos DB SDKs automatically retry on certain error conditions, and user applications are encouraged to have their own retry policies. See the [guide to designing resilient applications with Azure Cosmos DB SDKs](#) for a complete list of error conditions and when to retry.

Telemetry

Depending on the language of your application, diagnostics and telemetry are exposed as logs or promoted properties on the operation responses. For more information, see the "Capture the diagnostics" section in [Azure Cosmos DB C# SDK](#) and [Azure Cosmos DB Java SDK](#).

Data Lake Store

[Data Lake Storage Gen2](#) makes Azure Storage the foundation for building enterprise data lakes on Azure. Data Lake Storage Gen2 allows you to easily manage massive amounts of data.

The [Azure Storage Files Data Lake client library](#) includes all the capabilities required to make it easy for developers, data scientists, and analysts to store data of any size, shape, and speed, and do all types of processing and analytics across platforms and languages.

Retry mechanism

The [DataLakeServiceClient](#) allows you to manipulate Azure Data Lake service resources and file systems. The storage account provides the top-level namespace for the Data Lake service. When you create the client you could provide the client configuration options for connecting to Azure Data Lake service ([DataLakeClientOptions](#)). The [DataLakeClientOptions](#) includes a [Retry](#) property (inherited from [Azure.Core.ClientOptions](#)) that can be configured ([RetryOptions class](#)).

Telemetry

[Monitoring](#) the use and performance of Azure Storage is an important part of operationalizing your service. Examples include frequent operations, operations with high latency, or operations that cause service-side throttling.

All of the telemetry for your storage account is available through Azure Storage logs in Azure Monitor. This feature integrates your storage account with Log Analytics and Event Hubs, while also enabling you to archive logs to another storage account. To see the full list of metrics and resources logs and their associated schema, see [Azure Storage monitoring data reference](#).

Event Hubs

Azure Event Hubs is a hyperscale telemetry ingestion service that collects, transforms, and stores millions of events.

Retry mechanism

Retry behavior in the Azure Event Hubs Client Library is controlled by the [RetryPolicy](#) property on the [EventHubClient](#) class. The default policy retries with exponential backoff

when Azure Event Hubs returns a transient `EventHubsException` or an `OperationCanceledException`. Default retry policy for Event Hubs is to retry up to 9 times with an exponential back-off time of up to 30 seconds.

Example

C#

```
EventHubClient client = EventHubClient.CreateFromConnectionString("event_hub_connection_string");
client.RetryPolicy = RetryPolicy.Default;
```

Next steps

[.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 information, see [IoT Hub retry policy configuration](#).

Next steps

- [IoT Hub retry policy](#)
- [Troubleshoot IoT Hub device disconnection](#)

Azure Cache for Redis

Azure Cache for Redis is a fast data access and low latency cache service based on the popular open-source Redis cache. It's 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.

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.

C#

```
var deltaBackOffInMilliseconds = TimeSpan.FromSeconds(5).TotalMilliseconds;  
var maxDeltaBackOffInMilliseconds =
```

```

TimeSpan.FromSeconds(20).TotalMilliseconds;
var options = new ConfigurationOptions
{
    EndPoints = {"localhost"},
    ConnectRetry = 3,
    ReconnectRetryPolicy = new ExponentialRetry(deltaBackOffInMilliseconds,
maxDeltaBackOffInMilliseconds),
    ConnectTimeout = 2000
};
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options,
writer);

```

Alternatively, you can specify the options as a string, and pass this to the `Connect` method. The `ReconnectRetryPolicy` property can't be set this way, only through code.

C#

```

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.

C#

```

var conn =
ConnectionMultiplexer.Connect("redis0:6380,redis1:6380,connectRetry=3");

```

For more information, see [Stack Exchange Redis Configuration](#) in the `StackExchange.Redis` documentation.

The following table shows the default settings for the built-in retry policy.

Context	Setting	Default value (v 1.2.2)	Meaning
ConfigurationOptions	ConnectRetry	3	The number of times to repeat connect attempts during the initial connection operation.
	ConnectTimeout	Maximum 5000 ms plus SyncTimeout	Timeout (ms) for connect operations. Not a delay between retry attempts.
	SyncTimeout	1000	Time (ms) to allow for synchronous operations.
	ReconnectRetryPolicy	LinearRetry 5000 ms	Retry every 5000 ms.

ⓘ Note

For synchronous operations, `SyncTimeout` can add to the end-to-end latency, but setting the value too low can cause excessive timeouts. See [How to troubleshoot Azure Cache for Redis](#). In general, avoid using synchronous operations, and use asynchronous operations instead. For more information, see [Pipelines and Multiplexers](#).

Retry usage guidance

Consider the following guidelines when using Azure Cache for Redis:

- The StackExchange Redis client manages its own retries, but only when establishing a connection to the cache when the application first starts. You can configure the connection timeout, the number of retry attempts, and the time between retries to establish this connection, but the retry policy doesn't 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](#).

```
C#
```

```
var writer = new StringWriter();
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options,
writer);
```

An example of the output this generates is shown below.

```
text
```

```
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.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StackExchange.Redis;

namespace RetryCodeSamples
{
    class CacheRedisCodeSamples
    {
        public async static Task Samples()
        {
            var writer = new StringWriter();
            {
                try
                {
                    var retryTimeInMilliseconds =
TimeSpan.FromSeconds(4).TotalMilliseconds; // delay between retries

                    // Using object-based configuration.
                    var options = new ConfigurationOptions
                    {
                        EndPoints = { "localhost" },
                        ConnectRetry = 3,
                        ReconnectRetryPolicy = new
LinearRetry(retryTimeInMilliseconds)
                    };
                    ConnectionMultiplexer redis =
ConnectionMultiplexer.Connect(options, writer);

                    // Store a reference to the multiplexer for use in the

```

```
application.

    }

    catch
    {
        Console.WriteLine(writer.ToString());
        throw;
    }
}

}

}
```

The next example sets the configuration by specifying the options as a string. The connection timeout is the maximum period of time to wait for a connection to the cache, not the delay between retry attempts. The `ReconnectRetryPolicy` property can only be set by code.

C#

```
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 =

```

```
// 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.

Next steps

- [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

Azure SDK for .NET includes an [Azure.Search.Documents](#) client library from the Azure SDK team that is functionally equivalent to the previous client library, [Microsoft.Azure.Search](#).

Retry behavior in [Microsoft.Azure.Search](#) 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.

Retry behavior in [Azure.Search.Documents](#) is controlled by [SearchClientOptions](#) (It is part of the [SearchClient constructor](#)) in the property `Retry`, which belongs to the class [Azure.Core.RetryOptions](#)(where all configurations are available).

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

The namespace and some of the configuration details depend on which Service Bus client SDK package is used:

Package	Description	Namespace
Azure.Messaging.ServiceBus	Azure Service Bus client library for .NET	<code>Azure.Messaging.ServiceBus</code>
WindowsAzure.ServiceBus	This package is the older Service Bus client library. It requires .NET Framework 4.5.2.	<code>Microsoft.Azure.ServiceBus</code>

Retry usage guidance

The `ServiceBusRetryOptions` property specifies the retry options for the `ServiceBusClient` object:

Setting	Default value	Meaning
CustomRetryPolicy		A custom retry policy to be used in place of the individual option values.
Delay	0.8 seconds	The delay between retry attempts for a fixed approach or the delay on which to base calculations for a backoff-based approach.
MaxDelay	60 seconds	The maximum permissible delay between retry attempts.
MaxRetries	3	The maximum number of retry attempts before considering the associated operation to have failed.
Mode	Exponential	The approach to use for calculating retry delays.
TryTimeout	60 seconds	The maximum duration to wait for completion of a single attempt, whether the initial attempt or a retry.

Set the `Mode` property to configure the `ServiceBusRetryMode` with any of these values:

Property	Value	Description
Exponential	1	Retry attempts will delay based on a backoff strategy, where each attempt will increase the duration that it waits before retrying.

Property	Value	Description
Fixed	0	Retry attempts happen at fixed intervals; each delay is a consistent duration.

Example:

C#

```
using Azure.Messaging.ServiceBus;

string connectionString = "<connection_string>";
string queueName = "<queue_name>";

// Because ServiceBusClient implements IAsyncDisposable, we'll create it
// with "await using" so that it is automatically disposed for us.
var options = new ServiceBusClientOptions();
options.RetryOptions = new ServiceBusRetryOptions
{
    Delay = TimeSpan.FromSeconds(10),
    MaxDelay = TimeSpan.FromSeconds(30),
    Mode = ServiceBusRetryMode.Exponential,
    MaxRetries = 3,
};
await using var client = new ServiceBusClient(connectionString, options);
```

Telemetry

Service Bus collects the same kinds of monitoring data as other Azure resources. You can [Monitor Azure Service Bus](#) using Azure Monitor.

You also have various options for sending telemetry with the Service Bus .NET client libraries.

- [Tracking with Azure Application Insights](#)
- [Tracking with OpenTelemetry](#)

Example

The following code example shows how to use the `Azure.Messaging.ServiceBus` package to:

- Set the retry policy for a `ServiceBusClient` using a new `ServiceBusClientOptions`.
- Create a new message with a new instance of a `ServiceBusMessage`.
- Send a message to the Service Bus using the `ServiceBusSender.SendMessageAsync(message)` method.

- Receive using the `ServiceBusReceiver`, which are represented as `ServiceBusReceivedMessage` objects.

C#

```
// using Azure.Messaging.ServiceBus;

using Azure.Messaging.ServiceBus;

string connectionString = "<connection_string>";
string queueName = "queue1";

// Because ServiceBusClient implements IAsyncDisposable, we'll create it
// with "await using" so that it is automatically disposed for us.
var options = new ServiceBusClientOptions();
options.RetryOptions = new ServiceBusRetryOptions
{
    Delay = TimeSpan.FromSeconds(10),
    MaxDelay = TimeSpan.FromSeconds(30),
    Mode = ServiceBusRetryMode.Exponential,
    MaxRetries = 3,
};
await using var client = new ServiceBusClient(connectionString, options);

// The sender is responsible for publishing messages to the queue.
ServiceBusSender sender = client.CreateSender(queueName);
ServiceBusMessage message = new ServiceBusMessage("Hello world!");

await sender.SendMessageAsync(message);

// The receiver is responsible for reading messages from the queue.
ServiceBusReceiver receiver = client.CreateReceiver(queueName);
ServiceBusReceivedMessage receivedMessage = await
receiver.ReceiveMessageAsync();

string body = receivedMessage.Body.ToString();
Console.WriteLine(body);
```

Next steps

- [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 shouldn't be necessary, and the default settings will be fine.

C#

```
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));
```

Next steps

- [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 isn't 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 isn't recommended for use with Azure SQL Database except for interactive scenarios where there are only a few retries at short intervals. Instead, consider using an exponential back-off strategy for most 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's effectively added to the retry delay specified in the retry policy for every retry attempt.
- Close the connection after some 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's a chance that the same connection will be chosen from the pool, even after closing and reopening a connection. If so, a technique to resolve it's to call the **ClearPool** method of the **SqlConnection** class to mark the connection as not reusable. However, you should do this only after several connection attempts have failed, and only when encountering the specific class of transient failures such as SQL timeouts (error code -2) related to faulty connections.
- If the data access code uses transactions initiated as **TransactionScope** instances, the retry logic should reopen the connection and initiate a new transaction scope.

For this reason, the retryable code block should encompass the entire scope of the transaction.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

Context	Sample target E2E max latency	Retry strategy	Settings	Values	How it works
Interactive, UI, or foreground	2 sec	FixedInterval	Retry count Retry interval First fast retry	3 500 ms true	Attempt 1 - delay 0 sec Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	30 sec	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.

C#

```

public async static Task<SqlDataReader> ExecuteReaderWithRetryAsync(this
    SqlCommand command)
{
    GuardConnectionIsNotNull(command);

    var policy = Policy.Handle<Exception>().WaitAndRetryAsync(
        retryCount: 3, // Retry 3 times
        sleepDurationProvider: attempt => TimeSpan.FromMilliseconds(200 *
    Math.Pow(2, attempt - 1)), // Exponential backoff based on an initial 200 ms
    delay.

        onRetry: (exception, attempt) =>
    {
        // Capture some information for logging/telemetry.
        logger.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.

C#

```

var sqlCommand = sqlConnection.CreateCommand();
sqlCommand.CommandText = "[some query]";

using (var reader = await sqlCommand.ExecuteReaderWithRetryAsync())
{
    // Do something with the values
}

```

Next steps

- 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 aren't 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 doesn't 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 can't 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.

```
C#  
  
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.

```
C#  
  
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.

```
C#  
  
[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.

C#

```
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 isn't 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 doesn't 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 can't be changed.
DefaultRandomFactor	1.1	The multiplier used to add a random delay for each entry. This value can't be changed.
DefaultExponentialBase	2	The multiplier used to calculate the next delay. This value can't 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 isn't 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 can't 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 can't specify the delay between retry attempts (it's fixed and generated as an exponential sequence). You can specify only the maximum values, as shown here; unless you create a custom retry strategy. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

Context	Sample target	Retry policy	Settings	Values	How it works
	E2E max latency				

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**.

C#

```
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](#).

Next steps

- [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.

C#

```
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.

C#

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var transaction = db.Database.BeginTransaction())
        {
            db.Blogs.Add(new Blog { Url = "https://blogs.msdn.com/dotnet"
        }
    }
}
```

```

});  

    db.SaveChanges();  
  

    db.Blogs.Add(new Blog { Url =  

"https://blogs.msdn.com/visualstudio" });  

    db.SaveChanges();  
  

    transaction.Commit();  

}  

});  

}

```

Azure Storage

Azure Storage services include blob storage, files, and storage queues.

Blobs, Queues and Files

The ClientOptions Class is the base type for all client option types and exposes various common client options like Diagnostics, Retry, Transport. To provide the client configuration options for connecting to Azure Queue, Blob, and File Storage you must use the corresponding derived type. In the next example, you use the QueueClientOptions class (derived from ClientOptions) to configure a client to connect to Azure Queue Service. The Retry property is the set of options that can be specified to influence how retry attempts are made, and how a failure is eligible to be retried.

C#

```

using System;  

using System.Threading;  

using Azure.Core;  

using Azure.Identity;  

using Azure.Storage;  

using Azure.Storage.Queues;  

using Azure.Storage.Queues.Models;  
  

namespace RetryCodeSamples  

{  

    class AzureStorageCodeSamples {  
  

        public async static Task Samples() {  
  

            // Provide the client configuration options for connecting to  

            // Azure Queue Storage  

            QueueClientOptions queueClientOptions = new  

            QueueClientOptions()  

            {  

                Retry = {

```

```

        Delay = TimeSpan.FromSeconds(2),           //The delay between
retry attempts for a fixed approach or the delay on which to base
                                                //calculations for
a backoff-based approach
        MaxRetries = 5,                         //The maximum
number of retry attempts before giving up
        Mode = RetryMode.Exponential,          //The approach to
use for calculating retry delays
        MaxDelay = TimeSpan.FromSeconds(10)    //The maximum
permissible delay between retry attempts
    },

    GeoRedundantSecondaryUri = new Uri("https://...")
    // If the GeoRedundantSecondaryUri property is set, the
secondary Uri will be used for GET or HEAD requests during retries.
    // If the status of the response from the secondary Uri
is a 404, then subsequent retries for the request will not use the
    // secondary Uri again, as this indicates that the
resource may not have propagated there yet.
    // Otherwise, subsequent retries will alternate back and
forth between primary and secondary Uri.
};

Uri queueServiceUri = new
Uri("https://storageaccount.queue.core.windows.net/");
string accountName = "Storage account name";
string accountKey = "storage account key";

// Create a client object for the Queue service, including
QueueClientOptions.
QueueServiceClient serviceClient = new
QueueServiceClient(queueServiceUri, new DefaultAzureCredential(),
queueClientOptions);

CancellationTokenSource source = new
CancellationTokenSource();
CancellationToken cancellationToken = source.Token;

// Return an async collection of queues in the storage
account.
var queues = serviceClient.GetQueuesAsync(QueueTraits.None,
null, cancellationToken);

```

Table Support

Note

WindowsAzure.Storage Nuget Package and Microsoft.Azure.Cosmos.Table Nuget Package have been deprecated. For Azure table support, see [Azure.Data.Tables Nuget Package](#)

Retry mechanism

The client library is based on [Azure Core library](#), which is a library that provides cross-cutting services to other client libraries.

There are many reasons why failure can occur when a client application attempts to send a network request to a service. Some examples are timeout, network infrastructure failures, service rejecting the request due to throttle/busy, service instance terminating due to service scale-down, service instance going down to be replaced with another version, service crashing due to an unhandled exception, etc. By offering a built-in retry mechanism (with a default configuration the consumer can override), our SDKs and the consumer's application become resilient to these kinds of failures. Note that some services charge real money for each request and so consumers should be able to disable retries entirely if they prefer to save money over resiliency.

Policy configuration

Retry policies are configured programmatically. The configuration is based on the [RetryOption class](#). There is an attribute on [TableClientOptions](#) inherited from [ClientOptions](#)

C#

```
var tableClientOptions = new TableClientOptions();
tableClientOptions.Retry.Mode = RetryMode.Exponential;
tableClientOptions.Retry.MaxRetries = 5;
var serviceClient = new TableServiceClient(connectionString,
tableClientOptions);
```

The following tables show the possibilities for the built-in retry policies.

RetryOption

Setting	Meaning
Delay	The delay between retry attempts for a fixed approach or the delay on which to base calculations for a backoff-based approach. If the service provides a Retry-After response header, the next retry will be delayed by the duration specified by the header value.
MaxDelay	The maximum permissible delay between retry attempts when the service does not provide a Retry-After response header. If the service provides a Retry-After response header, the next retry will be delayed by the duration specified by the header value.

Setting	Meaning
Mode	The approach to use for calculating retry delays.
NetworkTimeout	The timeout applied to an individual network operations.

RetryMode

Setting	Meaning
Exponential	Retry attempts will delay based on a backoff strategy, where each attempt will increase the duration that it waits before retrying.
MaxDelay	Retry attempts happen at fixed intervals; each delay is a consistent duration.

Telemetry

The simplest way to see the logs is to enable console logging. To create an Azure SDK log listener that outputs messages to console use `AzureEventSourceListener.CreateConsoleLogger` method.

C#

```
// Setup a listener to monitor logged events.
using AzureEventSourceListener listener =
AzureEventSourceListener.CreateConsoleLogger();
```

Examples

Executing the following code example with the storage emulator shut down will allow us to see information about retries in the console.

C#

```
using Azure.Core;
using Azure.Core.Diagnostics;
using Azure.Data.Tables;
using Azure.Data.Tables.Models;

namespace RetryCodeSamples
{
    class AzureStorageCodeSamples
    {
        private const string connectionString =
"UseDevelopmentStorage=true";
        private const string tableName = "RetryTestTable";
```

```

public async static Task SamplesAsync()
{
    // Setup a listener to monitor logged events.
    using AzureEventSourceListener listener =
AzureEventSourceListener.CreateConsoleLogger();

    var tableClientOptions = new TableClientOptions();
    tableClientOptions.Retry.Mode = RetryMode.Exponential;
    tableClientOptions.Retry.MaxRetries = 5;

    var serviceClient = new TableServiceClient(connectionString,
tableClientOptions);

    TableItem table = await
serviceClient.CreateTableIfNotExistsAsync(tableName);
    Console.WriteLine($"The created table's name is {table.Name}.");
}
}
}

```

General REST and retry guidelines

Consider the following when accessing Azure or third-party services:

- Use a systematic approach to managing retries, perhaps as reusable code, so that you can apply a consistent methodology across all clients and all solutions.
- Consider using a retry framework such as [Polly](#) to manage retries if the target service or client has no built-in retry mechanism. This will help you implement a consistent retry behavior, and it may provide a suitable default retry strategy for the target service. However, you may need to create custom retry code for services that have nonstandard behavior that do not rely on exceptions to indicate transient failures or if you want to use a **Retry-Response** reply to manage retry behavior.
- The transient detection logic will depend on the actual client API you use to invoke the REST calls. Some clients, such as the newer **HttpClient** class, won't 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.
- Don't retry for status codes representing client errors (errors in the 4xx range) except for a 408 Request Timeout and 429 Too Many Requests.
- Thoroughly test your retry strategies and mechanisms under a range of conditions, such as different network states and varying system loadings.

Retry strategies

The following are the typical types of retry strategy intervals:

- **Exponential**. A retry policy that performs a specified number of retries, using a randomized exponential back off approach to determine the interval between retries. For example:

C#

```
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:

```
C#
```

```
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:

```
C#
```

```
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 doesn't 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.

Next steps

- [connection resiliency](#)
- [Data Points - EF Core 1.1](#)

Transient fault handling

Article • 02/24/2023

All applications that communicate with remote services and resources must be sensitive to transient faults. This is especially true for applications that run in the cloud, where, because of the nature of the environment and connectivity over the internet, this type of fault is 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, and timeouts that occur when a service is busy. These faults are often self-correcting, so, if the action is repeated after a suitable delay, it's likely to succeed.

This article provides general guidance for transient fault handling. For information about handling transient faults when you're using Azure services, see [Retry guidance for Azure services](#).

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. For solutions that run on local on-premises infrastructure, the performance and availability of the application and its components are typically maintained via expensive and often underused hardware redundancy, and components and resources are located close to each other. This approach makes failure less likely, but transient faults can still occur, as can outages caused by unforeseen events like external power supply or network issues, or other disaster scenarios.

Cloud hosting, including private cloud systems, can offer higher overall availability by using shared resources, redundancy, automatic failover, and dynamic resource allocation across many commodity compute nodes. However, because of the nature of cloud environments, 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 resources. Some services refuse connections when the load rises to a specific level, or when a maximum throughput rate is reached, 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 that use the shared resource.
- Cloud environments use large numbers of commodity hardware units. They deliver performance by dynamically distributing load across multiple computing units and infrastructure components. They deliver reliability by automatically recycling or

replacing failed units. Because of this dynamic nature, transient faults and temporary connection failures might occasionally occur.

- There are often more hardware components, including network infrastructure like routers and load balancers, between the application and the resources and services that it uses. This additional infrastructure can occasionally introduce additional connection latency and transient connection faults.
- Network conditions between the client and the server might be variable, especially when communication crosses the internet. Even in on-premises locations, heavy traffic loads can slow communication and cause intermittent connection failures.

Challenges

Transient faults can have a big effect on the perceived availability of an application, even if it's been thoroughly tested under all foreseeable circumstances. To ensure that cloud-hosted applications operate reliably, you need to ensure that they can respond to the following challenges:

- The application must be able to detect faults when they occur and determine if the faults are likely to be transient, are long-lasting, or are terminal failures. Different resources are likely to return different responses when a fault occurs, and these responses can also vary depending on the context of the operation. For example, the response for an error when the application is reading from storage might differ from the response for an error when it's writing to storage. Many resources and services have well-documented transient-failure contracts. However, when such information isn't available, it can be difficult to discover the nature of the fault and whether it's likely to be transient.
- The application must be able to retry the operation if it determines that the fault is likely to be transient. It also needs to keep track of the number of times the operation is retried.
- The application must use an appropriate strategy for retries. The strategy specifies the number of times the application 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. The strategy will vary depending on the type of resource and on the current operating conditions of the resource and the application.

General guidelines

The following guidelines can help you design suitable transient fault handling mechanisms for your applications.

Determine if there's 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 might return information that can help you determine whether a retry is appropriate and how long to wait before the next retry attempt.
- You should use the built-in retry mechanism when one is available, unless you have specific and well-understood requirements that make a different retry behavior more appropriate.

Determine if the operation is suitable for retrying

- Perform retry operations only when the faults are transient (typically indicated by the nature of the error) and when there's at least some likelihood that the operation will succeed when retried. There's no point in retrying operations that attempt an invalid operation, like a database update to an item that doesn't exist or a request to a service or resource that suffered a fatal error.
- In general, implement retries only when you can determine the full effect of doing so and when the conditions are well understood and can be validated. Otherwise, let the calling code implement retries. Remember that the errors returned from resources and services outside your control might evolve over time, and you might need to revisit your transient fault detection logic.
- When you create services or components, consider implementing error codes and messages that help clients determine whether they should retry failed operations. In particular, indicate whether 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 that are defined within your service contracts. Even though generic clients might not be able to read these errors, they're useful in the creation of custom clients.

Determine an appropriate retry count and interval

- Optimize the retry count and the interval to the type of use case. If you don't retry enough times, the application can't complete the operation and will probably fail.

If you retry too many times, or with too short an interval between tries, the application might hold resources like threads, connections, and memory for long periods, which adversely affects the health of the application.

- Adapt values for the time interval and the number of retry attempts to the type of operation. For example, if the operation is part of a user interaction, the interval should be short and only a few retries should be attempted. By using this approach, you can avoid making users wait for a response, which holds open connections and can reduce availability for other users. If the operation is part of a long running or critical workflow, where canceling and restarting the process is expensive or time-consuming, it's appropriate to wait longer between attempts and retry more times.
- Keep in mind that 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 increases the time between each subsequent retry. For example, it might 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 increases the time between each subsequent retry. For example, it might 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 might retry the operation every 3 seconds.
 - **Immediate retry.** Sometimes a transient fault is brief, possibly caused by an event like a network packet collision or a spike in a hardware component. In this case, retrying the operation immediately is appropriate because it might succeed if the fault is cleared in the time that it takes the application to assemble and send the next request. However, there should never be more than one immediate retry attempt. You should switch to alternative strategies, like exponential back-off or fallback actions, if the immediate retry fails.
 - **Randomization.** Any of the retry strategies listed previously can include a randomization to prevent multiple instances of the client sending subsequent retry attempts at the same time. For example, one instance might retry the operation after 3 seconds, 11 seconds, 28 seconds, and so on, while another instance might retry the operation after 4 seconds, 12 seconds, 26 seconds, and

so on. Randomization is a useful technique that can be combined with other strategies.

- As a general guideline, use an exponential back-off strategy for background operations, and use 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 factors that contribute to the overall maximum timeout for a retried operation. These factors include the time it takes for a failed connection to produce a response (typically set by a timeout value in the client), the delay between retry attempts, and the maximum number of retries. The total of all these times can result in long overall operation times, especially when you use an exponential delay strategy where the interval between retries grows rapidly after each failure. If a process must meet a specific service level agreement (SLA), the overall operation time, including all timeouts and delays, must be within the limits defined in the SLA.
- Don't implement overly aggressive retry strategies. These are strategies that have intervals that are too short or retries that are too frequent. They can have an adverse effect on the target resource or service. These strategies might prevent the resource or service from recovering from its overloaded state, and it will continue to block or refuse requests. This scenario results in a vicious circle, where more and more requests are sent to the resource or service. Consequently, its ability to recover is further reduced.
- Take into account the timeout of the operations when you choose retry intervals in order to avoid launching a subsequent attempt immediately (for example, if the timeout period is similar to the retry interval). Also, consider whether you need to keep the total possible period (the timeout plus the retry intervals) below a specific total time. If an operation has an unusually short or long timeout, the timeout might 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 number of retries and the interval between them. For example, some exceptions or error codes (like the HTTP code 503, Service Unavailable, with a Retry-After header in the response) might indicate how long the error might last, or that the service failed and won't respond to any subsequent attempt.

Avoid anti-patterns

- In most cases, 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 require doing so. In these exceptional circumstances, use policies that prevent excessive numbers of retries and delay periods, and make sure you understand the consequences. For example, say one component makes a request to another, which then accesses the target service. If you implement retry with a count of three on both calls, there are nine retry attempts in total against the service. Many services and resources implement a built-in retry mechanism. You should investigate how you can disable or modify these mechanisms if you need to implement retries at a higher level.
- Never implement an endless retry mechanism. Doing so is likely to prevent the resource or service from recovering from overload situations and to cause throttling and refused connections to continue for a longer time. Use a finite number of retries, or implement a pattern like [Circuit Breaker](#) to allow the service to recover.
- Never perform an immediate retry more than once.
- Avoid using a regular retry interval when you access services and resources on Azure, especially when you have a high number of retry attempts. The best 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 simultaneously. If this scenario is likely to occur, introduce randomization into the retry intervals.

Test your retry strategy and implementation

- Fully test your retry strategy under as wide a set of circumstances as possible, especially when both the application and the target resources or services that it uses are under extreme load. To check behavior during testing, you can:
 - Inject transient and nontransient faults into the service. For example, send invalid requests or add code that detects test requests and responds with different types of errors. For examples that use TestApi, see [Fault Injection Testing with TestApi](#) and [Introduction to TestApi – Part 5: Managed Code Fault Injection APIs](#).
 - Create a mockup of the resource or service that returns a range of errors that the real service might return. Cover all the types of errors that your retry

strategy is designed to detect.

- For custom services that you create and deploy, force transient errors to occur by temporarily disabling or overloading the service. (Don't attempt to overload any shared resources or shared services in 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 (like the HTTP status code, headers, body, or other factors). Doing so enables deterministic testing of a subset of the failure conditions, for transient faults and other types of failures. 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. These tests will also help ensure that the retry doesn't 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 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 (like regular, exponential back-off, and randomization), the actual interval values, and the number of times to retry.
- Implement retries in many places, even in 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 to store all policies. For example, store values like the interval and retry count in application configuration files, read them at runtime, and programmatically build the retry policies. Doing so 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 use suitable defaults if the values can't 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 you can change them without needing to restart the application.
- Take advantage of built-in or default retry strategies that are available in the client APIs that you use, but only when they're appropriate for your scenario. These strategies are typically generic. In some scenarios, they might be all you need, but

in other scenarios they don't offer the full range of options to suit your specific requirements. To determine the most appropriate values, you need to perform testing to understand how the settings affect your application.

Log and track transient and nontransient faults

- As part of your retry strategy, include exception handling and other instrumentation that logs retry attempts. An occasional transient failure and retry are expected and don't indicate a problem. Regular and increasing numbers of retries, however, are often an indicator of a problem that might cause a failure or that degrades application performance and availability.
- Log transient faults as warning entries rather than as error entries so that monitoring systems don't detect them as application errors that might trigger false alerts.
- Consider storing a value in your log entries that indicates whether retries are caused by throttling in the service or by other types of faults, like 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 elapsed times for operations that include a retry mechanism. This metric is a good indicator of the overall effect of transient faults on user response times, process latency, and the efficiency of application use cases. Also log the number of retries that occur so you can understand the factors that contribute 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 elapsed before operations succeed is increasing.

Manage operations that continually fail

- Consider how you'll handle operations that continue to fail at every attempt. Situations like this are inevitable.
 - Although a retry strategy defines the maximum number of times that an operation should be retried, it doesn't prevent the application from 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 might detect a connection timeout and consider it to be a transient fault. The code retries the operation a specified number of times and then gives up. However, when another customer places an order, the operation is attempted again, even though it will fail every time.

- To prevent continual retries for operations that continually fail, you should consider implementing the [Circuit Breaker pattern](#). When you use this pattern, if the number of failures within a specified time window exceeds a threshold, requests return to the caller immediately as errors, and there's no attempt to access the failed resource or service.
- The application can periodically test the service, on an intermittent basis and with long intervals between requests, to detect when it becomes available. An appropriate interval depends on factors like the criticality of the operation and the nature of the service. It might be anything between a few minutes and several hours. When the test succeeds, the application can resume normal operations and pass requests to the newly recovered service.
- In the meantime, you might be able to fall back to another instance of the service (maybe in a different datacenter or application), use a similar service that offers compatible (maybe simpler) functionality, or perform some alternative operations based on the hope that the service will be available soon. For example, it might be appropriate to store requests for the service in a queue or data store and retry them later. Or 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 to indicate that the application isn't currently available.

Other considerations

- When you're deciding on the values for the number of retries and the retry intervals for a policy, consider whether the operation on the service or resource is part of a long-running or multistep operation. It might 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 might be acceptable as long as that strategy doesn't block other operations by holding or locking scarce resources.
- Consider whether retrying the same operation could cause inconsistencies in data. If some parts of a multistep process are repeated and the operations aren't idempotent, inconsistencies might occur. For example, if an operation that increments a value is repeated, it produces an invalid result. Repeating an

operation that sends a message to a queue might cause an inconsistency in the message consumer if the consumer can't detect duplicate messages. To prevent these scenarios, design each step as an idempotent operation. For more information, see [Idempotency patterns](#).

- Consider the scope of operations that are retried. For example, it might be easier to implement retry code at a level that encompasses several operations and retry them all if one fails. However, doing so might 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 you determine retry intervals, when you monitor the elapsed times of the operation, and before you raise alerts for failures.
- Consider how your retry strategy might affect neighbors and other tenants in a shared application and when you use 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 might be affected by the retry policies implemented by other users of the resources and services. For business-critical applications, you might want to use premium services that aren't shared. Doing so provides you with more control over the load and consequent throttling of these resources and services, which can help to justify the extra cost.

Related resources

- [Retry guidance for Azure services](#)
- [Circuit Breaker pattern](#)
- [Compensating Transaction pattern](#)
- [Idempotency patterns](#)

Performance tuning a distributed application

Article • 10/04/2023

In this series, we walk through several cloud application scenarios, showing how a development team used load tests and metrics to diagnose performance issues. These articles are based on actual load testing that we performed when developing example applications. The code for each scenario is available on GitHub.

Scenarios:

- [Distributed business transaction](#)
- [Calling multiple backend services](#)
- [Event stream processing](#)

What is performance?

Performance is frequently measured in terms of throughput, response time, and availability. Performance targets should be based on business operations. Customer-facing tasks may have more stringent requirements than operational tasks such as generating reports.

Define a service level objective (SLO) that defines performance targets for each workload. You typically achieve this objective by breaking a performance target into a set of Key Performance Indicators (KPIs), such as:

- Latency or response time of specific requests
- The number of requests performed per second
- The rate at which the system generates exceptions.

Performance targets should explicitly include a target load. Also, not all users receive exactly the same level of performance, even when accessing the system simultaneously and performing the same work. So an SLO should be framed in terms of percentiles.

An example SLO for might be: "Client requests have a response within 500 ms @ P90, at loads up to 25 K requests/second."

Challenges of performance tuning a distributed system

It can be especially challenging to diagnose performance issues in a distributed application. Some of the challenges are:

- A single business transaction or operation typically involves multiple components of the system. It can be hard to get a holistic end-to-end view of a single operation.
- Resource consumption is distributed across multiple nodes. To get a consistent view, you need to aggregate logs and metrics in one place.
- The cloud offers elastic scale. Autoscaling is an important technique for handling spikes in load, but it can also mask underlying issues. Also, it can be hard to know which components need to scale and when.
- Workloads often don't scale across cores or threads. It's important to understand the requirements of your workloads and look into better optimized sizes. Some sizes offer constrained cores and disabled hyperthreading to improve single core oriented and per core licensed workloads.
- Cascading failures can cause failures upstream of the root problem. As a result, the first signal of the problem may appear in a different component than the root cause.

General best practices

Performance tuning is both an art and a science, but it can be made closer to science by taking a systematic approach. Here are some best practices:

- Enable telemetry to collect metrics. Instrument your code. Follow [best practices for monitoring](#). Use correlated tracing so that you can view all the steps in a transaction.
- Monitor the 90/95/99 percentiles, not just average. The average can mask outliers. The sampling rate for metrics also matters. If the sampling rate is too low, it can hide spikes or outliers that might indicate problems.
- Attack one bottleneck at a time. Form a hypothesis and test it by changing one variable at a time. Removing one bottleneck will often uncover another bottleneck further upstream or downstream.
- Errors and retries can have a large impact on performance. If you see that backend services are throttling your system, scale out or try to optimize usage (for example by tuning database queries).

- Look for common [performance anti-patterns](#).
- Look for opportunities to parallelize. Two common sources of bottlenecks are message queues and databases. In both cases, sharding can help. For more information, see [Horizontal, vertical, and functional data partitioning](#). Look for hot partitions that might indicate imbalanced read or write loads.

Next steps

Read the performance tuning scenarios

- [Distributed business transaction](#)
- [Calling multiple backend services](#)
- [Event stream processing](#)

Distributed business transaction performance tuning

Azure Kubernetes Service (AKS)

Azure Cache for Redis

This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that we did for a sample application. The application is from the [Azure Kubernetes Service \(AKS\) Baseline for microservices](#).

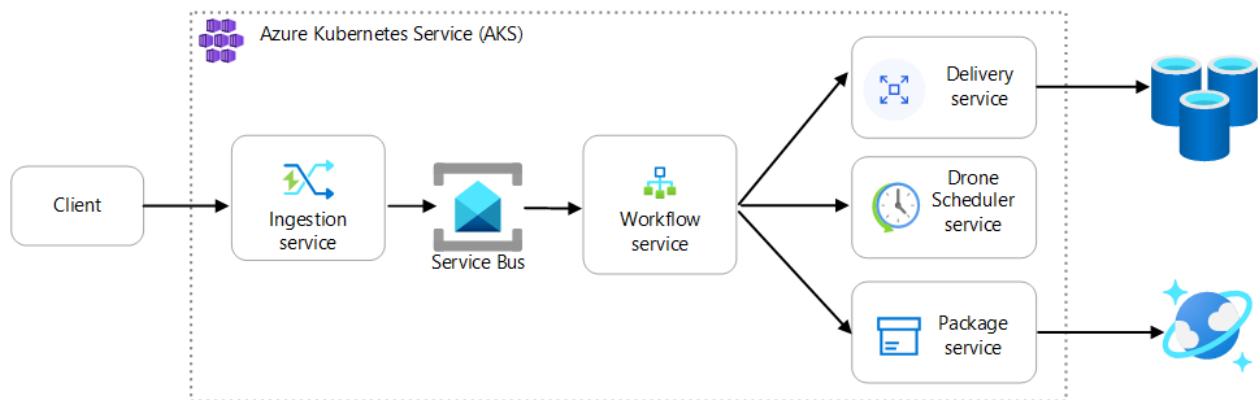
This article is part of a series. Read the first part [here](#).

Scenario: A client application initiates a business transaction that involves multiple steps.

This scenario involves a drone delivery application that runs on AKS. Customers use a web app to schedule deliveries by drone. Each transaction requires multiple steps that are performed by separate microservices on the back end:

- The Delivery service manages deliveries.
- The Drone Scheduler service schedules drones for pickup.
- The Package service manages packages.

There are two other services: An Ingestion service that accepts client requests and puts them on a queue for processing, and a Workflow service that coordinates the steps in the workflow.



For more information about this scenario, see [Designing a microservices architecture](#).

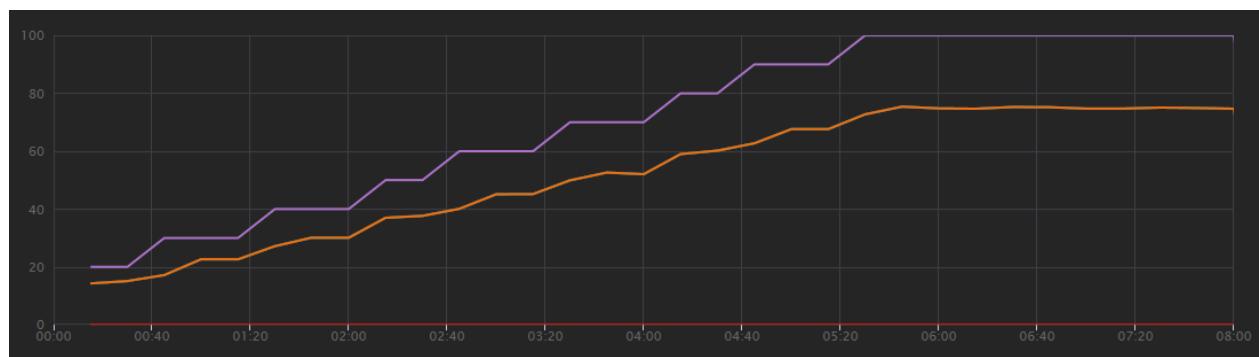
Test 1: Baseline

For the first load test, the team created a six-node AKS cluster and deployed three replicas of each microservice. The load test was a step-load test, starting at two simulated users and ramping up to 40 simulated users.

[Expand table](#)

Setting	Value
Cluster nodes	6
Pods	3 per service

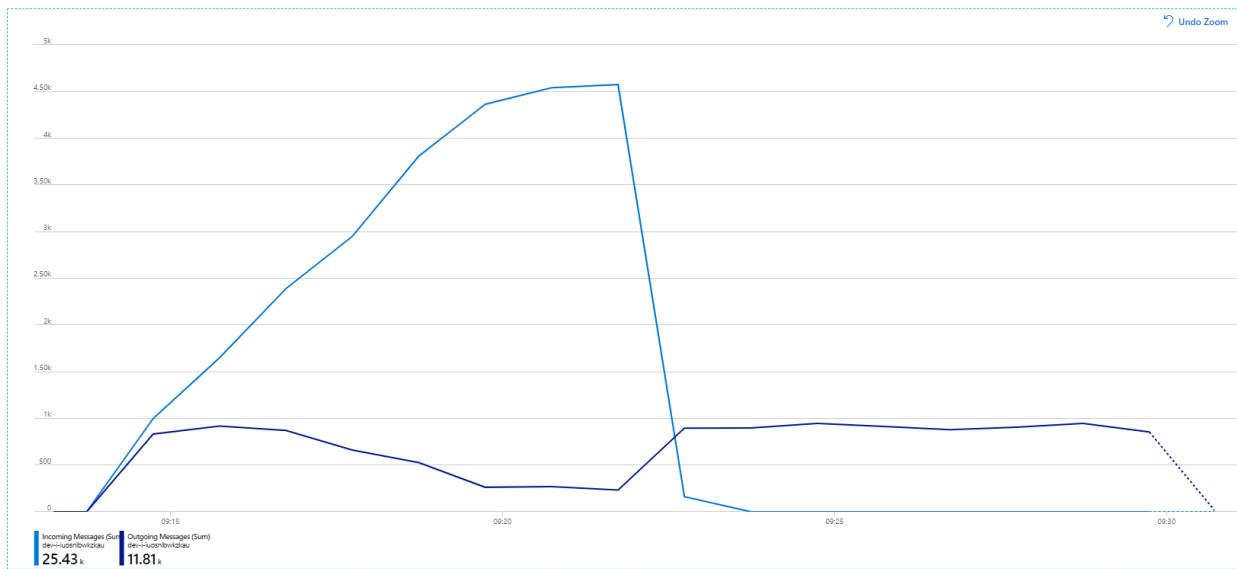
The following graph shows the results of the load test, as shown in Visual Studio. The purple line plots user load, and the orange line plots total requests.



The first thing to realize about this scenario is that client requests per second is not a useful metric of performance. That's because the application processes requests asynchronously, so the client gets a response right away. The response code is always HTTP 202 (Accepted), meaning the request was accepted but processing is not complete.

What we really want to know is whether the backend is keeping up with the request rate. The Service Bus queue can absorb spikes, but if the backend cannot handle a sustained load, processing will fall further and further behind.

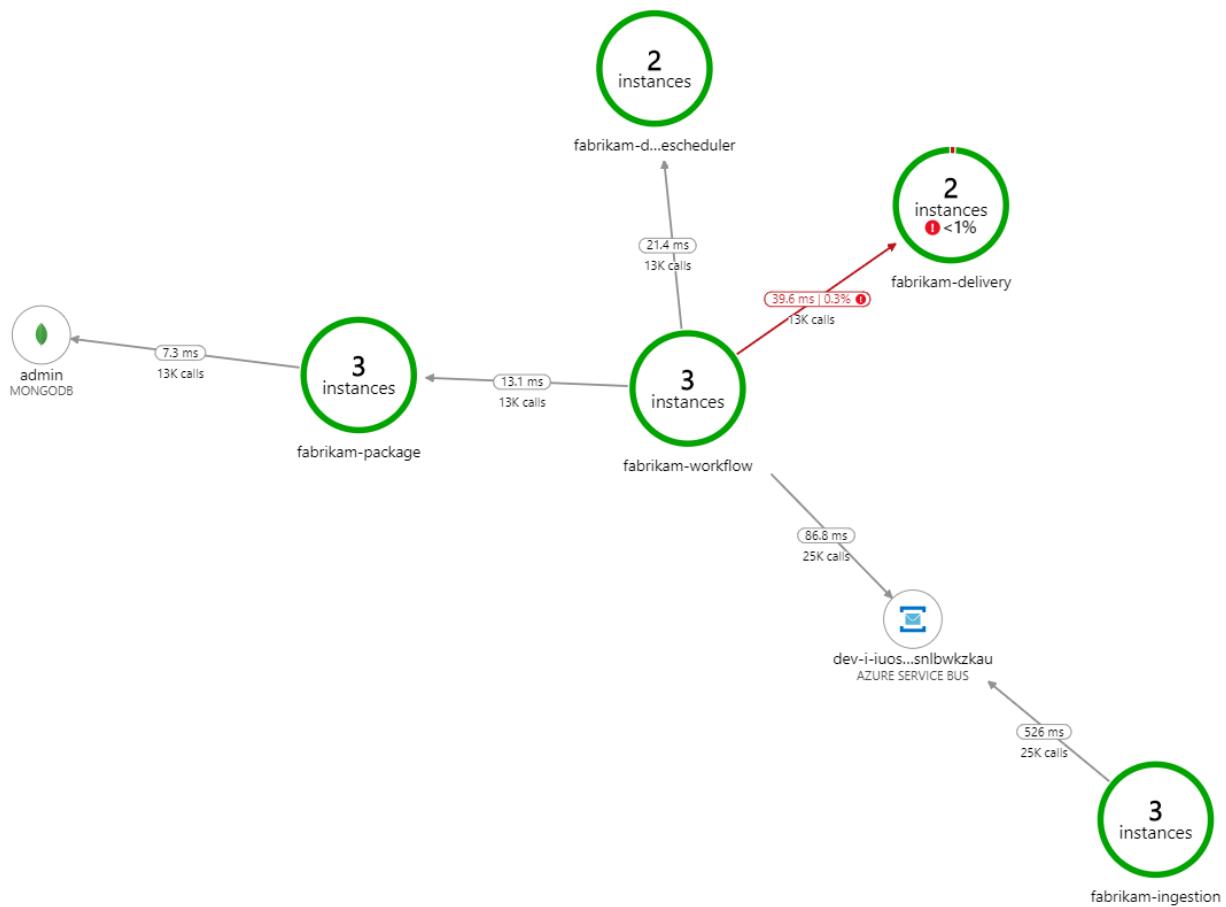
Here's a more informative graph. It plots the number incoming and outgoing messages on the Service Bus queue. Incoming messages are shown in light blue, and outgoing messages are shown in dark blue:



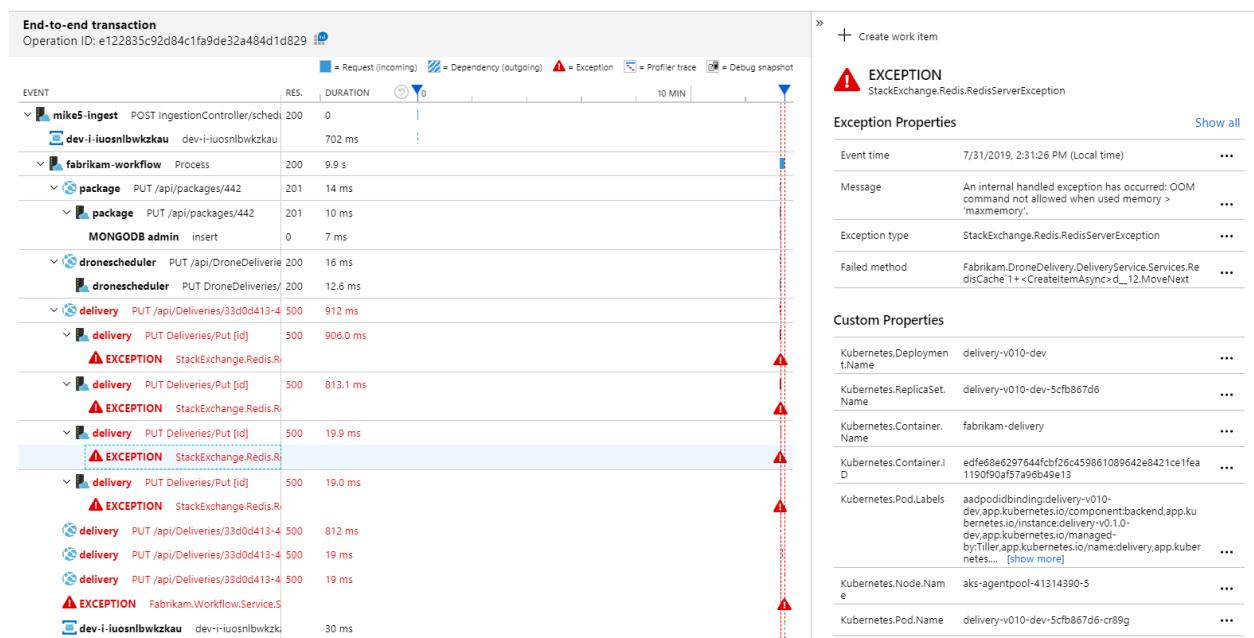
This chart is showing that the rate of incoming messages increases, reaching a peak and then dropping back to zero at the end of the load test. But the number of outgoing messages peaks early in the test and then actually drops. That means the Workflow service, which handles the requests, isn't keeping up. Even after the load test ends (around 9:22 on the graph), messages are still being processed as the Workflow service continues to drain the queue.

What's slowing down the processing? The first thing to look for is errors or exceptions that might indicate a systematic issue. The [Application Map](#) in Azure Monitor shows the graph of calls between components, and is a quick way to spot issues and then click through to get more details.

Sure enough, the Application Map shows that the Workflow service is getting errors from the Delivery service:



To see more details, you can select a node in the graph and click into an end-to-end transaction view. In this case, it shows that the Delivery service is returning HTTP 500 errors. The error messages indicate that an exception is being thrown due to memory limits in Azure Cache for Redis.

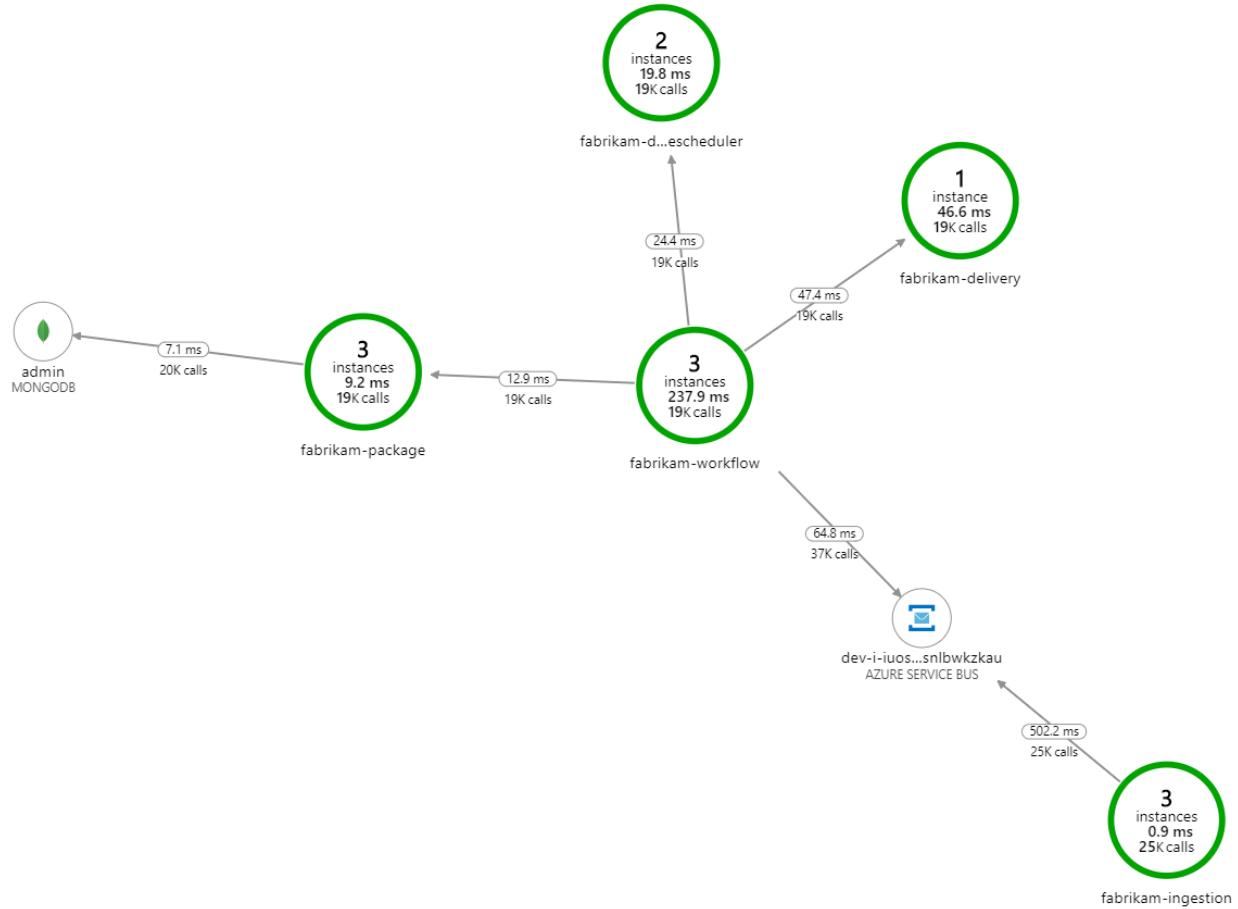


You may notice that these calls to Redis don't appear in the Application Map. That's because the .NET library for Application Insights doesn't have built-in support for tracking Redis as a dependency. (For a list of what's supported out of the box, see [Dependency auto-collection](#).) As a fallback, you can use the [TrackDependency](#) API to

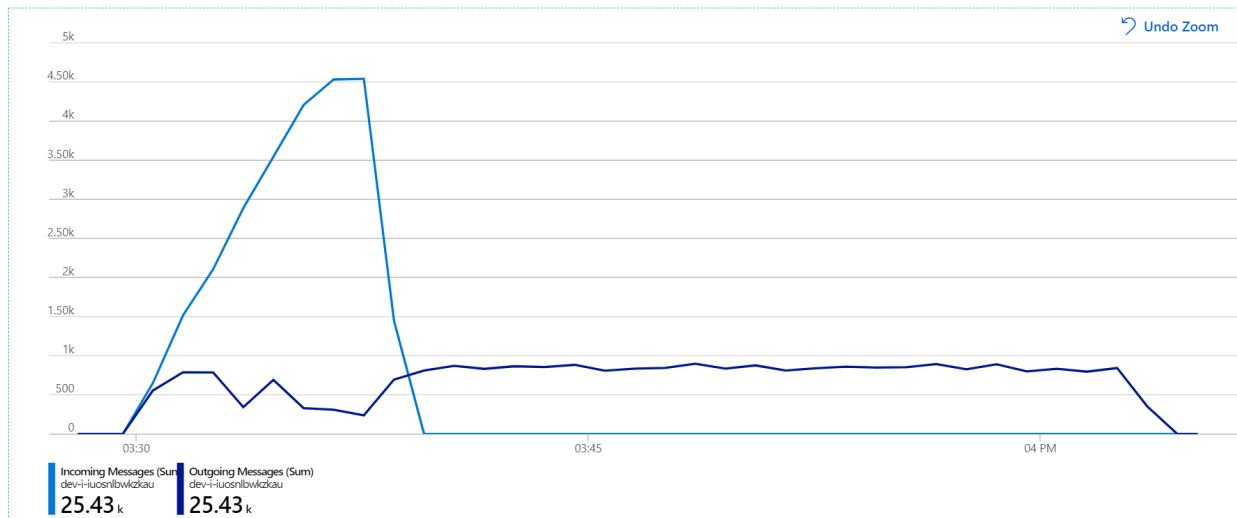
track any dependency. Load testing often reveals these kinds of gaps in the telemetry, which can be remediated.

Test 2: Increased cache size

For the second load test, the development team increased the cache size in Azure Cache for Redis. (See [How to Scale Azure Cache for Redis](#).) This change resolved the out-of-memory exceptions, and now the Application Map shows zero errors:



However, there is still a dramatic lag in processing messages. At the peak of the load test, the incoming message rate is more than 5× the outgoing rate:



This graph was generated by running a query in the Log Analytics workspace, using the [Kusto query language](#):

```
Kusto

let start=datetime("2020-07-31T22:30:00.000Z");
let end=datetime("2020-07-31T22:45:00.000Z");
dependencies
| where cloud_RoleName == 'fabrikam-workflow'
| where timestamp > start and timestamp < end
| where type == 'Azure Service Bus'
| where target has 'https://dev-i-iuosnlbwkzkau.servicebus.windows.net'
| where client_Type == "PC"
| where name == "Complete"
| summarize succeeded=sumif(itemCount, success == true),
```

```
failed=sumif(itemCount, success == false) by bin(timestamp, 5s)
| render timechart
```

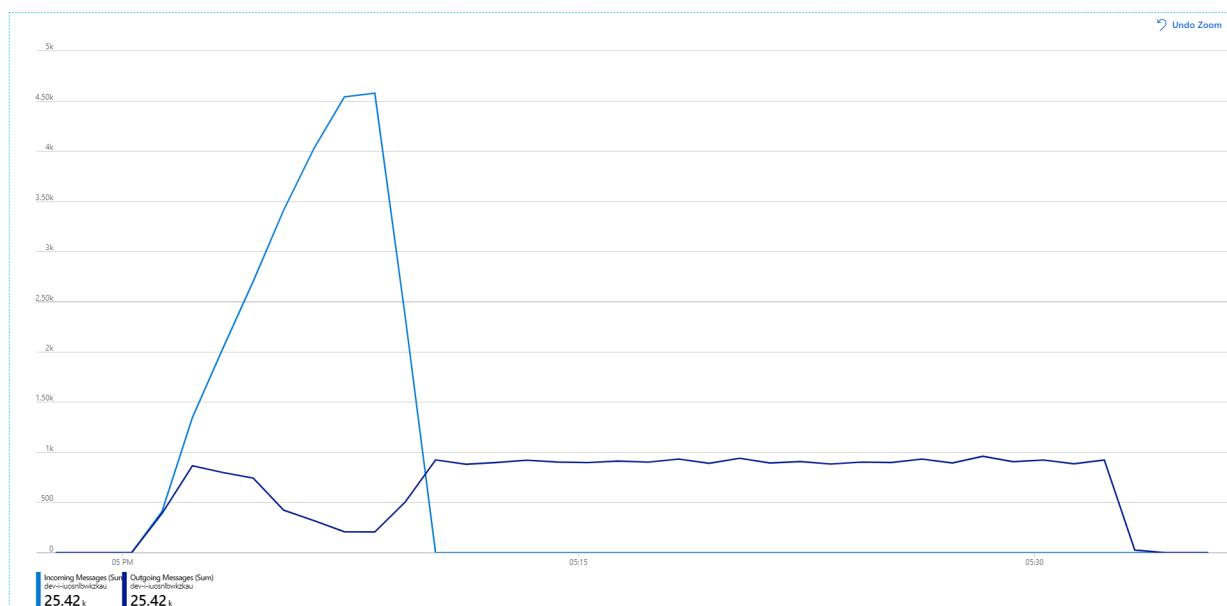
Test 3: Scale out the backend services

It appears the back end is the bottleneck. An easy next step is to scale out the business services (Package, Delivery, and Drone Scheduler), and see if throughput improves. For the next load test, the team scaled these services out from three replicas to six replicas.

[+] Expand table

Setting	Value
Cluster nodes	6
Ingestion service	3 replicas
Workflow service	3 replicas
Package, Delivery, Drone Scheduler services	6 replicas each

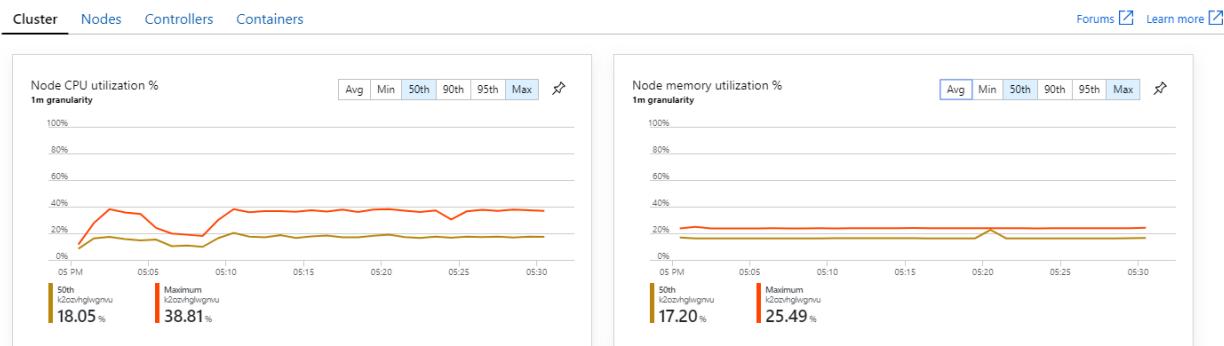
Unfortunately this load test shows only modest improvement. Outgoing messages are still not keeping up with incoming messages:



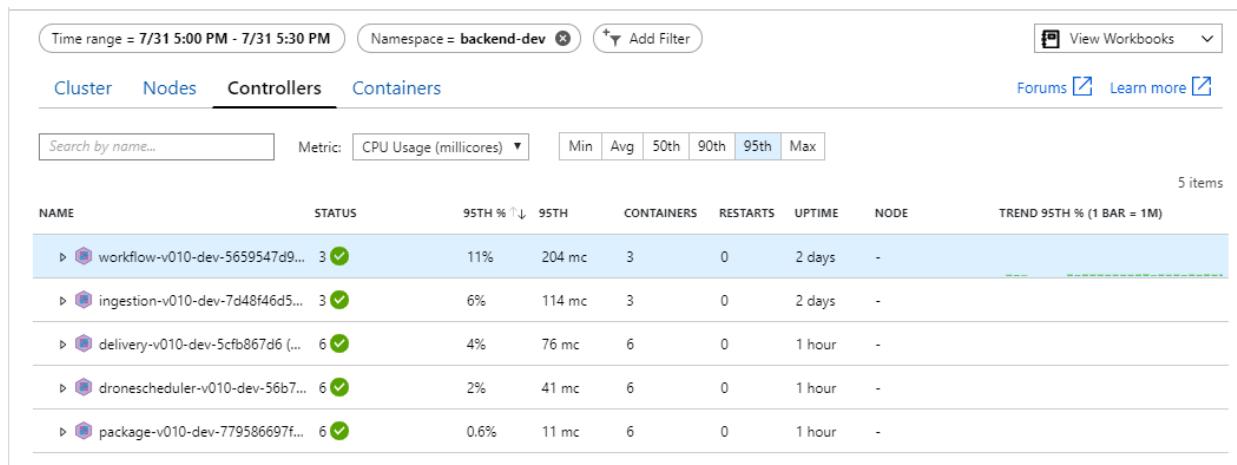
Throughput is more consistent, but the maximum achieved is about the same as the previous test:



Moreover, looking at [Azure Monitor container insights](#), it appears the problem is not caused by resource exhaustion within the cluster. First, the node-level metrics show that CPU utilization remains under 40% even at the 95th percentile, and memory utilization is about 20%.



In a Kubernetes environment, it's possible for individual pods to be resource-constrained even when the nodes aren't. But the pod-level view shows that all pods are healthy.



From this test, it seems that just adding more pods to the back end won't help. The next step is to look more closely at the Workflow service to understand what's happening when it processes messages. Application Insights shows that the average duration of the Workflow service's `Process` operation is 246 ms.

NAME	DURATION (AVG)
Process	221.4 ms

Investigate performance

We can also run a query to get metrics on the individual operations within each transaction:

[Expand table](#)

target	percentile_duration_50	percentile_duration_95
<code>https://dev-i-iuosnlbwkzkau.servicebus.windows.net/</code> dev-i-iuosnlbwkzkau	86.66950203	283.4255578
delivery	37	57
package	12	17
dronescheduler	21	41

The first row in this table represents the Service Bus queue. The other rows are the calls to the backend services. For reference, here's the Log Analytics query for this table:

```
Kusto

let start=datetime("2020-07-31T22:30:00.000Z");
let end=datetime("2020-07-31T22:45:00.000Z");
let dataset=dependencies
| where timestamp > start and timestamp < end
```

```

| where (cloud_RoleName == 'fabrikam-workflow')
| where name == 'Complete' or target in ('package', 'delivery',
'dronescheduler');
dataset
| summarize percentiles(duration, 50, 95) by target

```

```

let start=datetime("2019-08-01T00:00:00.000Z");
let end=datetime("2019-08-01T00:30:00.000Z");
let dataset=dependencies
| where timestamp > start and timestamp < end
| where (cloud_RoleName == 'fabrikam-workflow')
| where name == 'Complete' or target in ('package', 'delivery', 'dronescheduler');
dataset
| summarize percentiles(duration, 50, 95) by target

```

Completed 00:00:03.200 4 records

TABLE CHART Columns ▾

Drag a column header and drop it here to group by that column

target	percentile_duration_50	percentile_duration_95
> delivery	37	57
> https://dev-i-iuosnlbwkzkau.servicebus.windows.net/ dev-i-iuosnlbwkzkau	86.6695020315	283.4255577965
> package	12	17
> dronescheduler	21	41

These latencies look reasonable. But here is the key insight: If the total operation time is ~250 ms, that puts a strict upper bound on how fast messages can be processed in serial. The key to improving throughput, therefore, is greater parallelism.

That should be possible in this scenario, for two reasons:

- These are network calls, so most of the time is spent waiting for I/O completion
- The messages are independent, and don't need to be processed in order.

Test 4: Increase parallelism

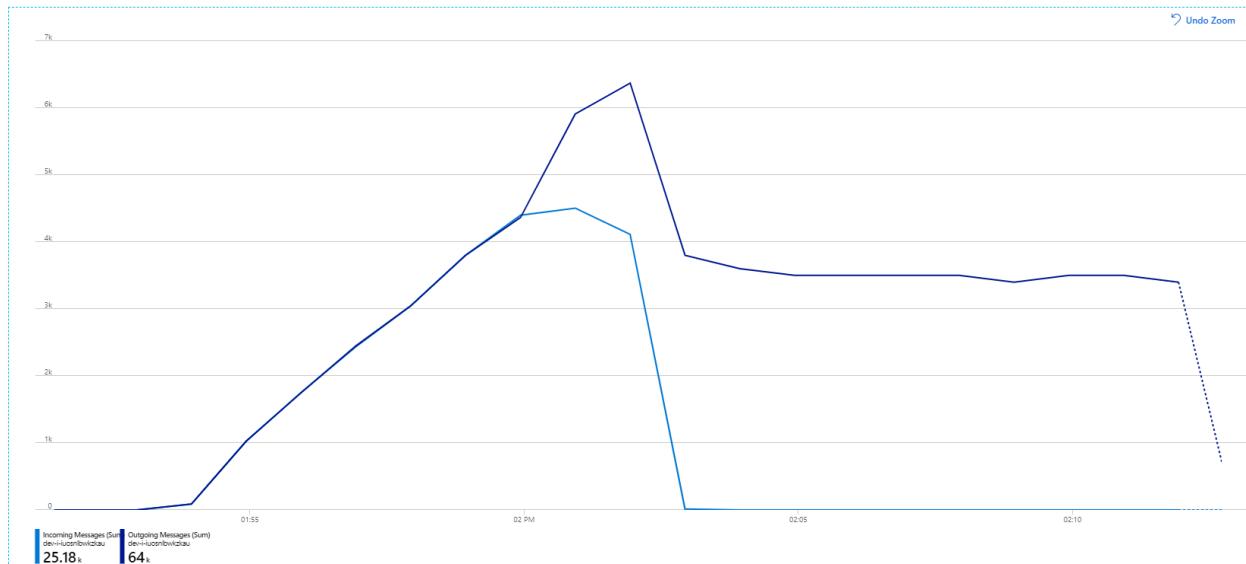
For this test, the team focused on increasing parallelism. To do so, they adjusted two settings on the Service Bus client used by the Workflow service:

Expand table

Setting	Description	Default	New value
MaxConcurrentCalls	The maximum number of messages to process concurrently.	1	20
PrefetchCount	How many messages the client will	0	3000

Setting	Description	Default	New value
	fetch ahead of time into its local cache.		

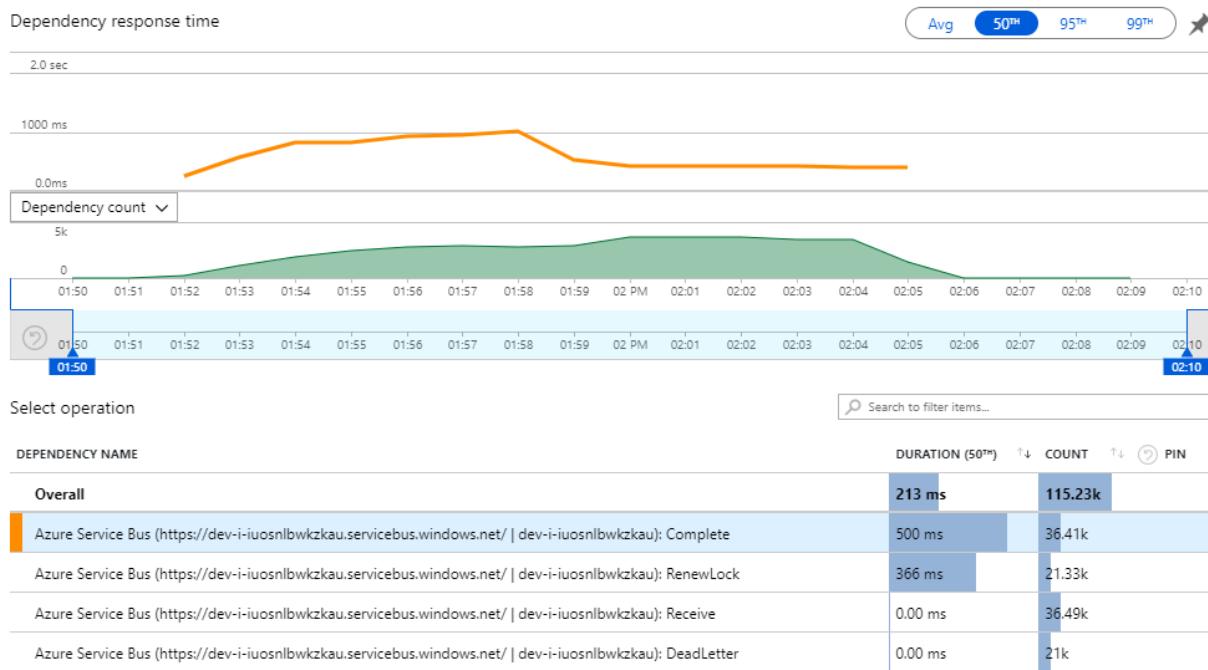
For more information about these settings, see [Best Practices for performance improvements using Service Bus Messaging](#). Running the test with these settings produced the following graph:



Recall that incoming messages are shown in light blue, and outgoing messages are shown in dark blue.

At first glance, this is a very weird graph. For a while, the outgoing message rate exactly tracks the incoming rate. But then, at about the 2:03 mark, the rate of incoming messages levels off, while the number of outgoing messages continues to rise, actually exceeding the total number of incoming messages. That seems impossible.

The clue to this mystery can be found in the **Dependencies** view in Application Insights. This chart summarizes all of the calls that the Workflow service made to Service Bus:



Notice that entry for `DeadLetter`. That calls indicates messages are going into the Service Bus [dead-letter queue](#).

To understand what's happening, you need to understand [Peek-Lock](#) semantics in Service Bus. When a client uses Peek-Lock, Service Bus atomically retrieves and locks a message. While the lock is held, the message is guaranteed not to be delivered to other receivers. If the lock expires, the message becomes available to other receivers. After a maximum number of delivery attempts (which is configurable), Service Bus will put the messages onto a [dead-letter queue](#), where it can be examined later.

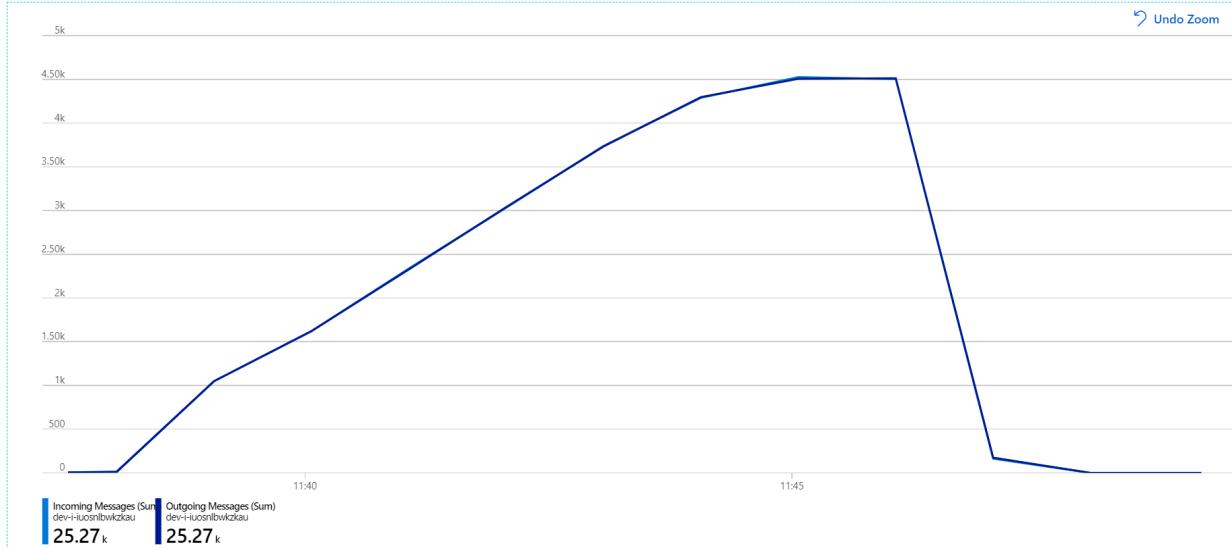
Remember that the Workflow service is prefetching large batches of messages — 3000 messages at a time). That means the total time to process each message is longer, which results in messages timing out, going back onto the queue, and eventually going into the dead-letter queue.

You can also see this behavior in the exceptions, where numerous `MessageLostLockException` exceptions are recorded:

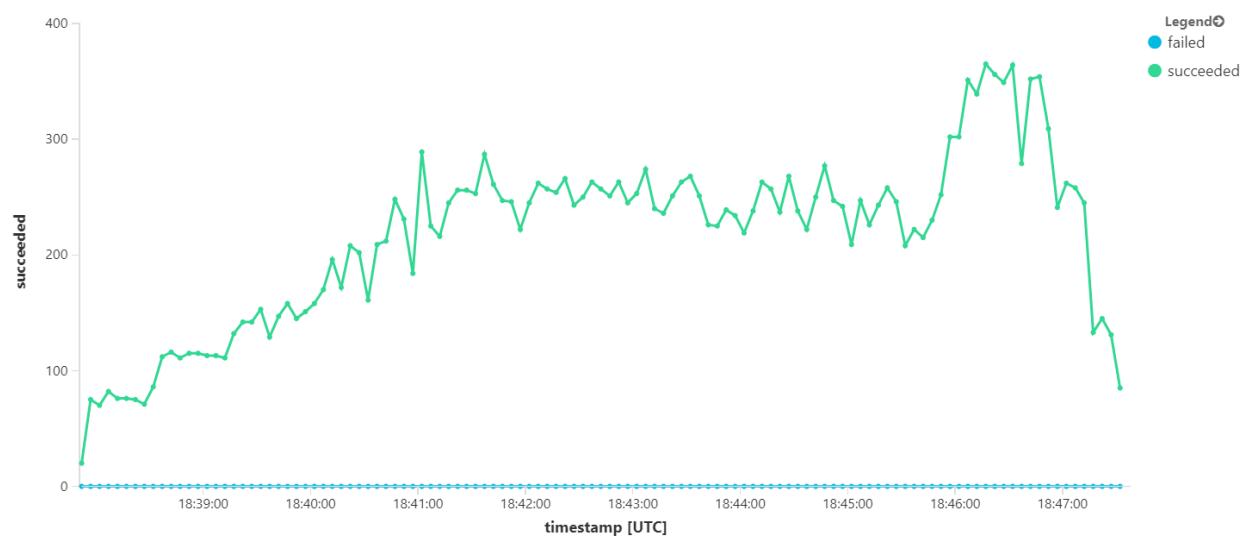
Top 10 exception types		COUNT	FILTERING
MessageLockLostE...		161.94k	+ -
IOException	MessageLockLostException	3	
SocketException		1	

Test 5: Increase lock duration

For this load test, the message lock duration was set to 5 minutes, to prevent lock timeouts. The graph of incoming and outgoing messages now shows that the system is keeping up with the rate of incoming messages:



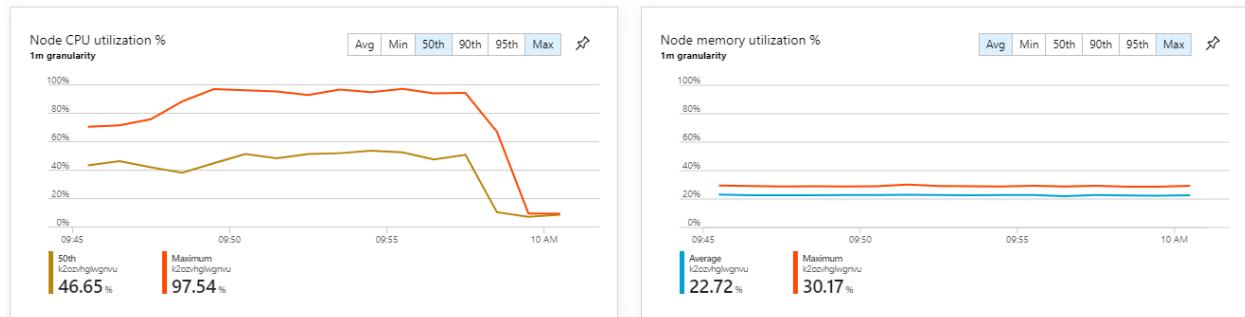
Over the total duration of the 8-minute load test, the application completed 25 K operations, with a peak throughput of 72 operations/sec, representing a 400% increase in maximum throughput.



However, running the same test with a longer duration showed that the application could not sustain this rate:



The container metrics show that maximum CPU utilization was close to 100%. At this point, the application appears to be CPU-bound. Scaling the cluster might improve performance now, unlike the previous attempt at scaling out.



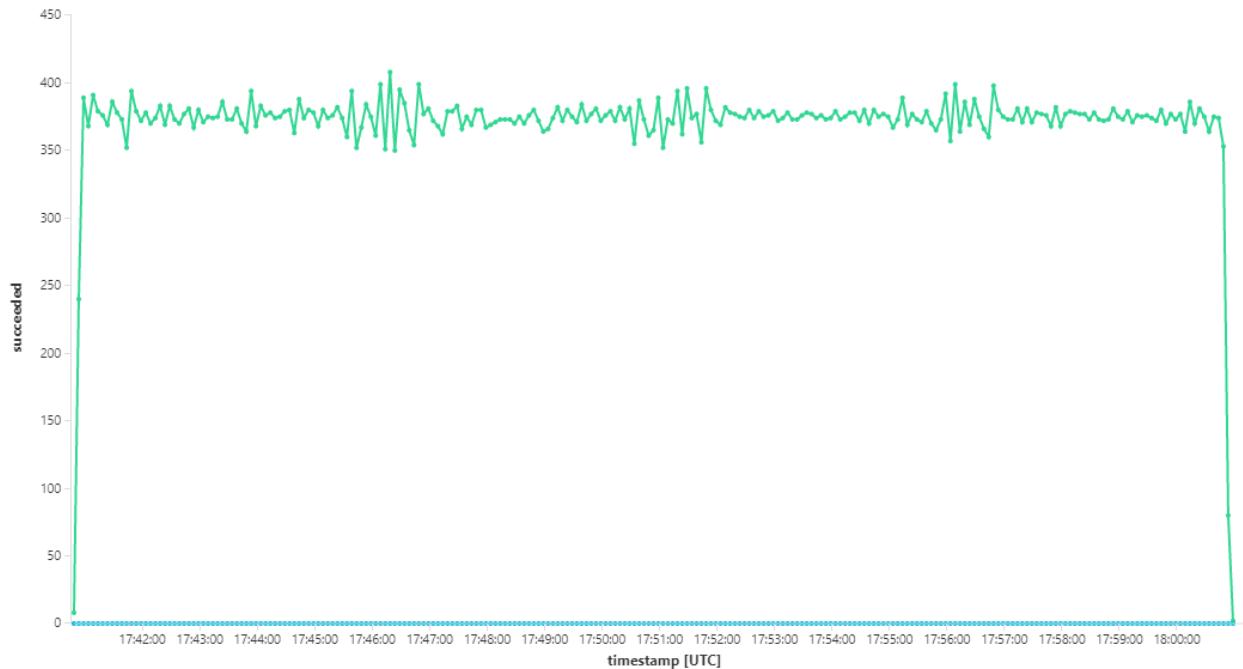
Test 6: Scale out the backend services (again)

For the final load test in the series, the team scaled out the Kubernetes cluster and pods as follows:

[\[\] Expand table](#)

Setting	Value
Cluster nodes	12
Ingestion service	3 replicas
Workflow service	6 replicas
Package, Delivery, Drone Scheduler services	9 replicas each

This test resulted in a higher sustained throughput, with no significant lags in processing messages. Moreover, node CPU utilization stayed below 80%.



Summary

For this scenario, the following bottlenecks were identified:

- Out-of-memory exceptions in Azure Cache for Redis.
- Lack of parallelism in message processing.
- Insufficient message lock duration, leading to lock timeouts and messages being placed in the dead letter queue.
- CPU exhaustion.

To diagnose these issues, the development team relied on the following metrics:

- The rate of incoming and outgoing Service Bus messages.
- Application Map in Application Insights.
- Errors and exceptions.
- Custom Log Analytics queries.
- CPU and memory utilization in Azure Monitor container insights.

Next steps

For more information about the design of this scenario, see [Designing a microservices architecture](#).

Performance tuning - Multiple backend services

Azure Kubernetes Service (AKS)

Azure Cosmos DB

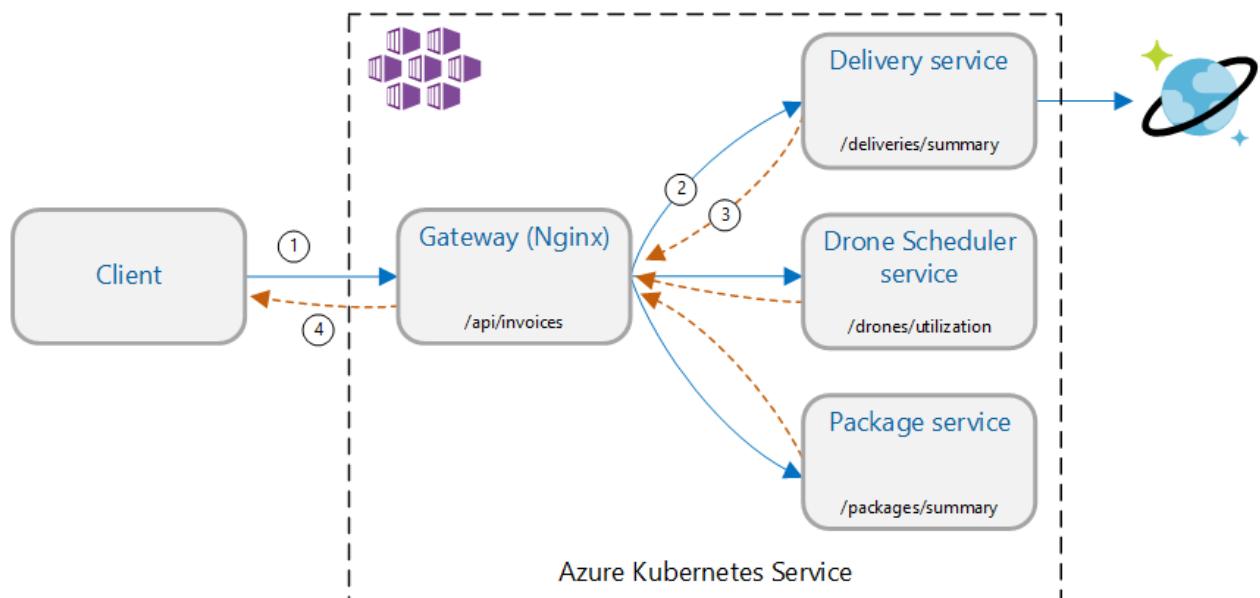
This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that was done for a sample application. The application is from the [Azure Kubernetes Service \(AKS\) Baseline for microservices](#), along with a [Visual Studio load test project](#) used to generate the results.

This article is part of a series. Read the first part [here](#).

Scenario: Call multiple backend services to retrieve information and then aggregate the results.

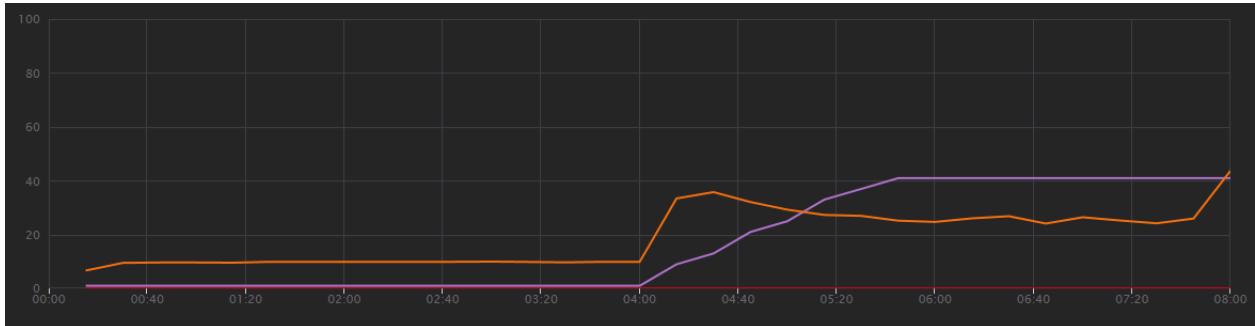
This scenario involves a drone delivery application. Clients can query a REST API to get their latest invoice information. The invoice includes a summary of the customer's deliveries, packages, and total drone utilization. This application uses a microservices architecture running on AKS, and the information needed for the invoice is spread across several microservices.

Rather than the client calling each service directly, the application implements the [Gateway Aggregation](#) pattern. Using this pattern, the client makes a single request to a gateway service. The gateway in turn calls the backend services in parallel, and then aggregates the results into a single response payload.



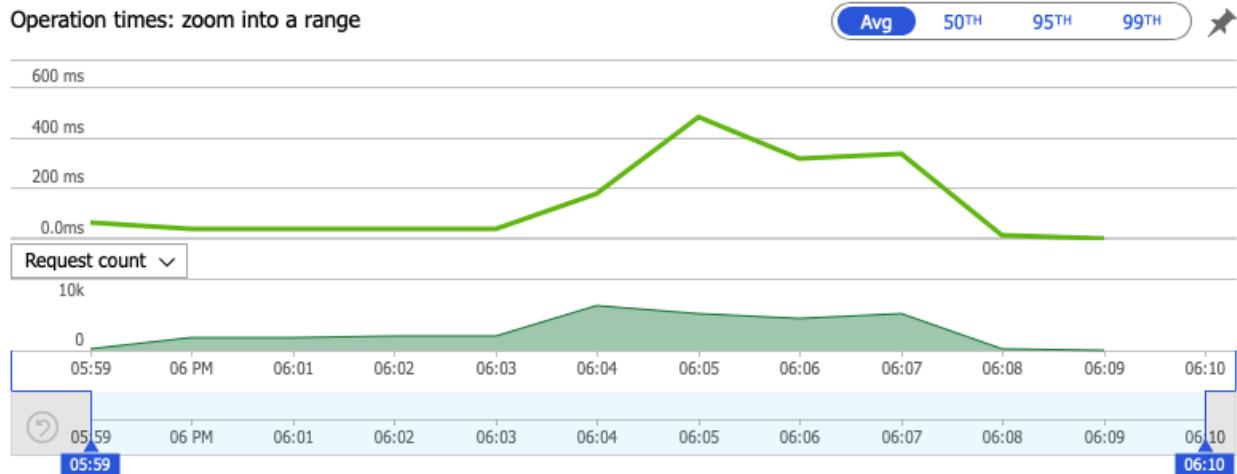
Test 1: Baseline performance

To establish a baseline, the development team started with a step-load test, ramping the load from one simulated user up to 40 users, over a total duration of 8 minutes. The following chart, taken from Visual Studio, shows the results. The purple line shows the user load, and the orange line shows throughput (average requests per second).



The red line along the bottom of the chart shows that no errors were returned to the client, which is encouraging. However, the average throughput peaks about half way through the test, and then drops off for the remainder, even while the load continues to increase. That indicates the back end is not able to keep up. The pattern seen here is common when a system starts to reach resource limits — after reaching a maximum, throughput actually falls significantly. Resource contention, transient errors, or an increase in the rate of exceptions can all contribute to this pattern.

Let's dig into the monitoring data to learn what's happening inside the system. The next chart is taken from Application Insights. It shows the average durations of the HTTP calls from the gateway to the backend services.



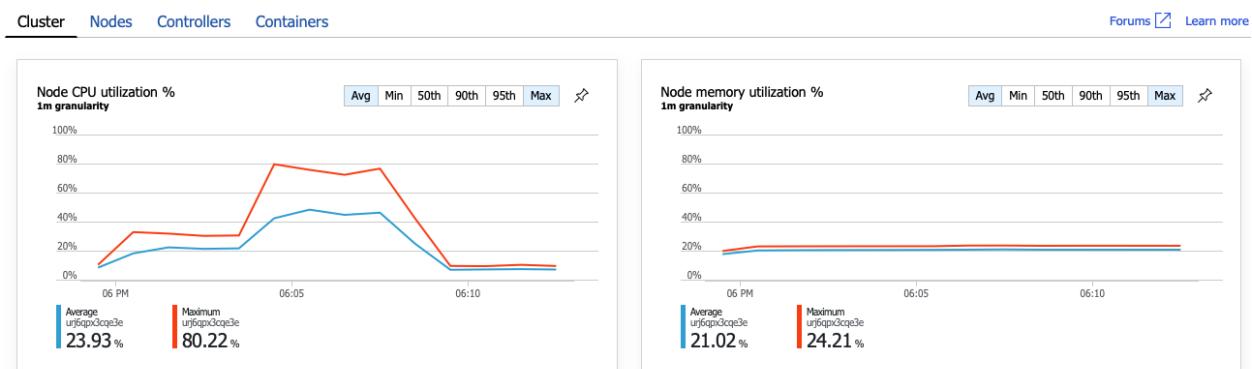
Select operation

Search to filter items...

OPERATION NAME	DURATION (AVG)	COUNT	PIN
Overall	244 ms	28.79k	?
GET DroneDeliveries/GetDroneUtilization	669 ms	10.19k	
GET Deliveries/GetSummary	21.3 ms	9.31k	
GET /api/packages/summary	1.97 ms	9.21k	

This chart shows that one operation in particular, `GetDroneUtilization`, takes much longer on average — by an order of magnitude. The gateway makes these calls in parallel, so the slowest operation determines how long it takes for the entire request to complete.

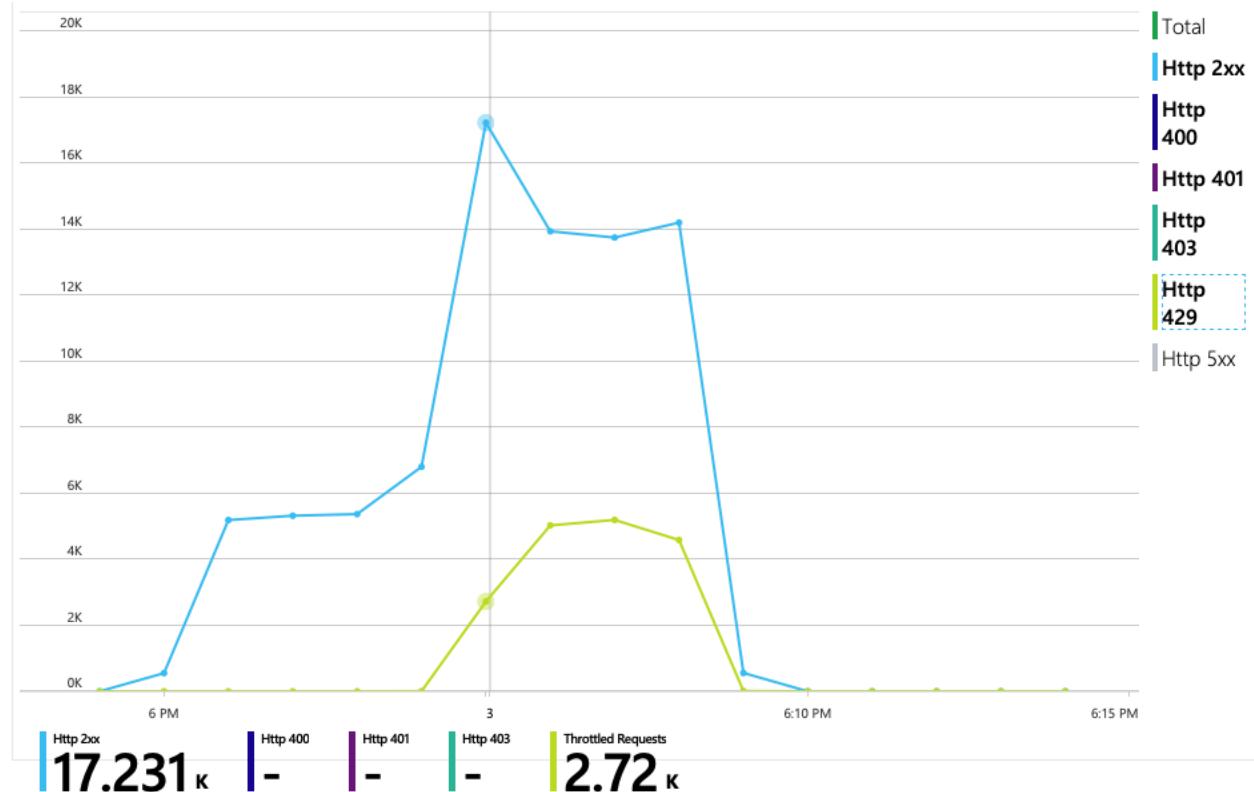
Clearly the next step is dig into the `GetDroneUtilization` operation and look for any bottlenecks. One possibility is resource exhaustion. Perhaps this particular backend service is running out of CPU or memory. For an AKS cluster, this information is available in the Azure portal through the [Azure Monitor container insights](#) feature. The following graphs show resource utilization at the cluster level:



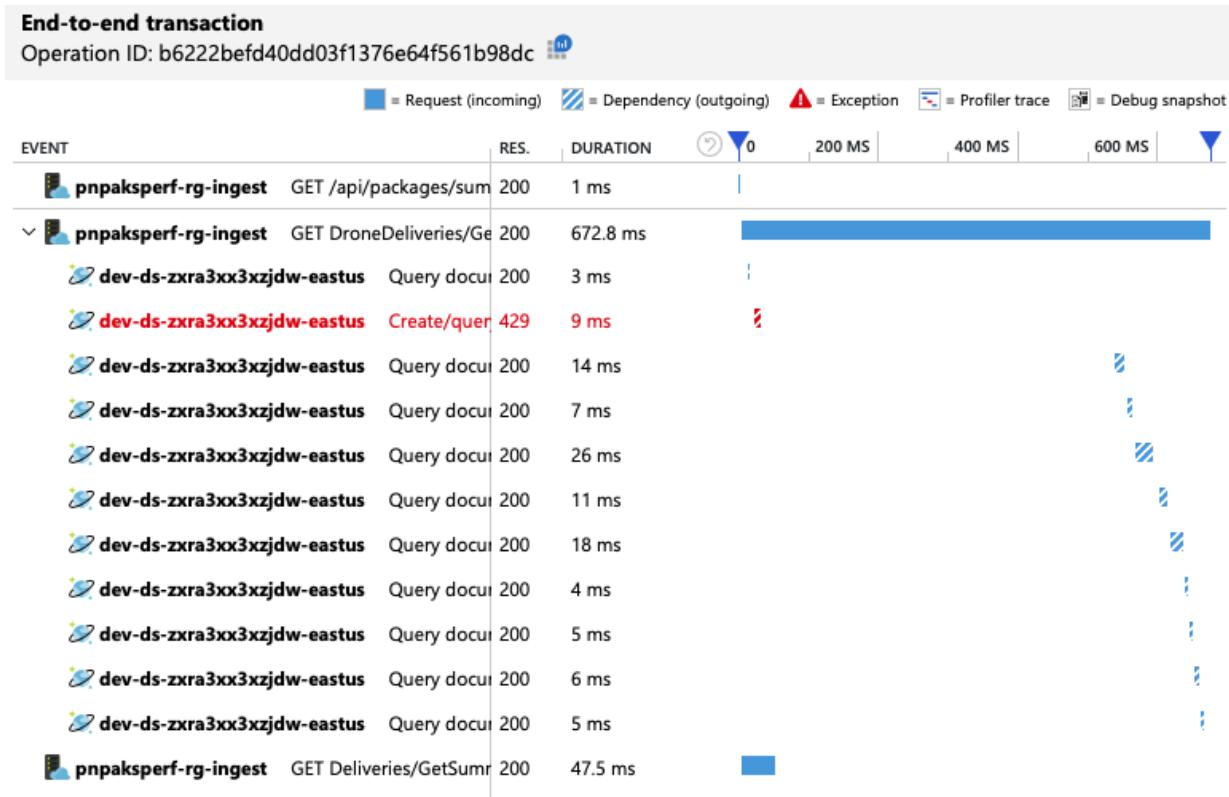
In this screenshot, both the average and maximum values are shown. It's important to look at more than just the average, because the average can hide spikes in the data. Here, the average CPU utilization stays below 50%, but there are a couple of spikes to

80%. That's close to capacity but still within tolerances. Something else is causing the bottleneck.

The next chart reveals the true culprit. This chart shows HTTP response codes from the Delivery service's backend database, which in this case is Azure Cosmos DB. The blue line represents success codes (HTTP 2xx), while the green line represents HTTP 429 errors. An HTTP 429 return code means that Azure Cosmos DB is temporarily throttling requests, because the caller is consuming more resource units (RU) than provisioned.



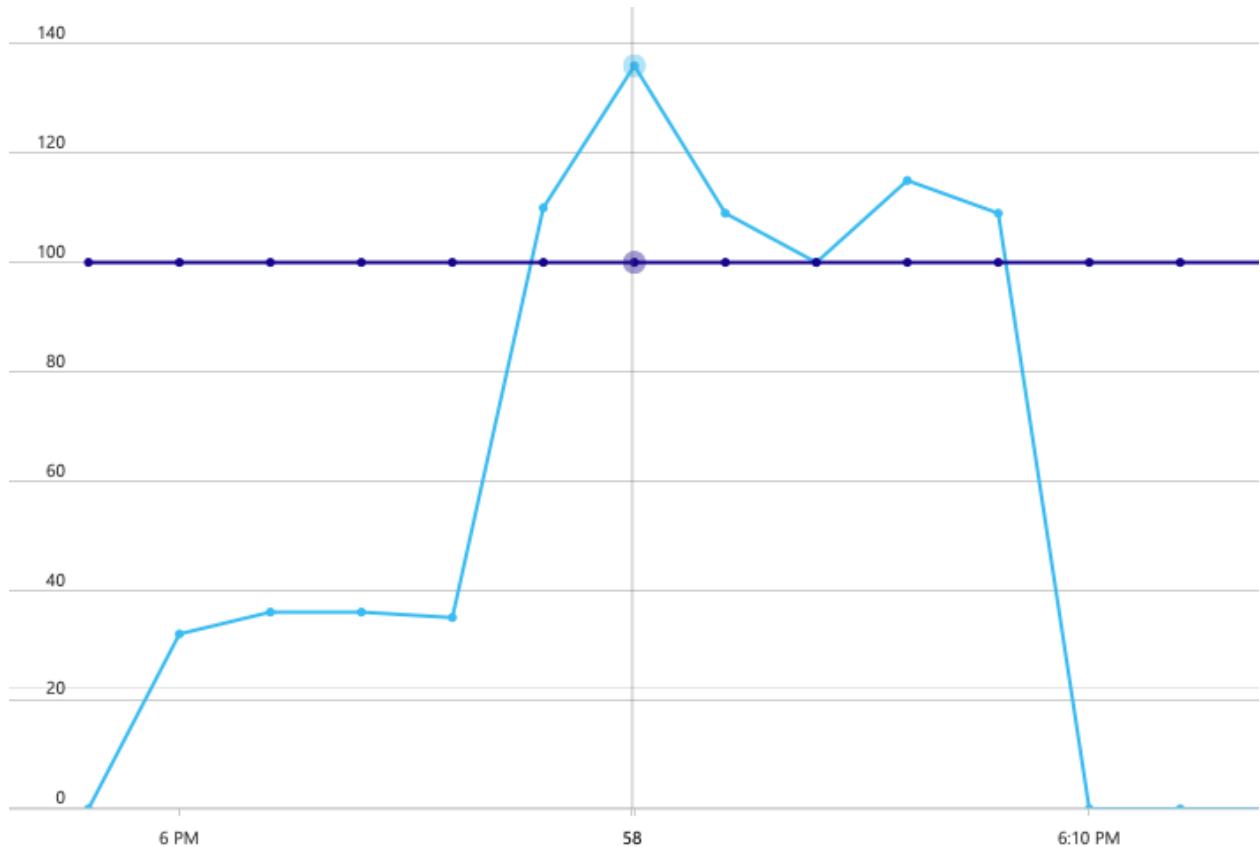
To get further insight, the development team used Application Insights to view the end-to-end telemetry for a representative sample of requests. Here is one instance:



This view shows the calls related to a single client request, along with timing information and response codes. The top-level calls are from the gateway to the backend services. The call to `GetDroneUtilization` is expanded to show calls to external dependencies — in this case, to Azure Cosmos DB. The call in red returned an HTTP 429 error.

Note the large gap between the HTTP 429 error and the next call. When the Azure Cosmos DB client library receives an HTTP 429 error, it automatically backs off and waits to retry the operation. What this view shows is that during the 672 ms this operation took, most of that time was spent waiting to retry Azure Cosmos DB.

Here's another interesting graph for this analysis. It shows RU consumption per physical partition versus provisioned RUs per physical partition:



To make sense of this graph, you need to understand how Azure Cosmos DB manages partitions. Collections in Azure Cosmos DB can have a *partition key*. Each possible key value defines a logical partition of the data within the collection. Azure Cosmos DB distributes these logical partitions across one or more *physical* partitions. The management of physical partitions is handled automatically by Azure Cosmos DB. As you store more data, Azure Cosmos DB might move logical partitions into new physical partitions, in order to spread load across the physical partitions.

For this load test, the Azure Cosmos DB collection was provisioned with 900 RUs. The chart shows 100 RU per physical partition, which implies a total of nine physical partitions. Although Azure Cosmos DB automatically handles the sharding of physical partitions, knowing the partition count can give insight into performance. The development team will use this information later, as they continue to optimize. Where the blue line crosses the purple horizontal line, RU consumption has exceeded the provisioned RUs. That's the point where Azure Cosmos DB will begin to throttle calls.

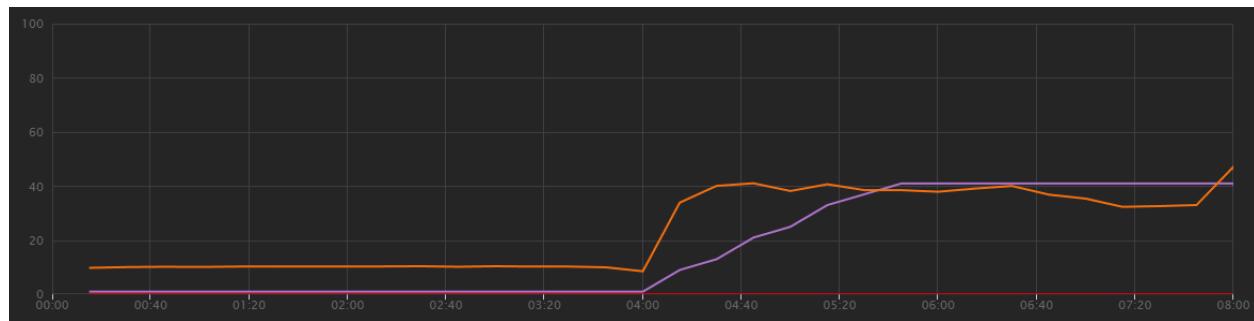
Test 2: Increase resource units

For the second load test, the team scaled out the Azure Cosmos DB collection from 900 RU to 2500 RU. Throughput increased from 19 requests/second to 23 requests/second, and average latency dropped from 669 ms to 569 ms.

 Expand table

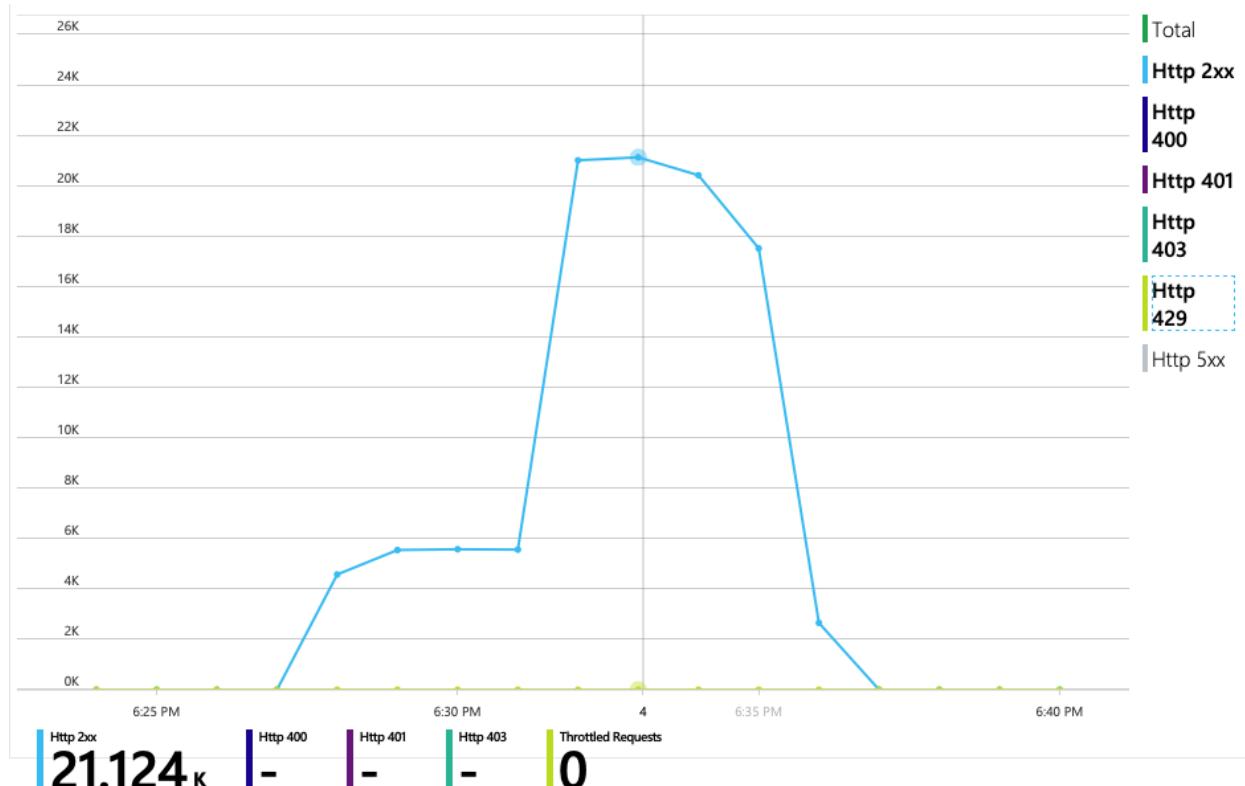
Metric	Test 1	Test 2
Throughput (req/sec)	19	23
Average latency (ms)	669	569
Successful requests	9.8 K	11 K

These aren't huge gains, but looking at the graph over time shows a more complete picture:

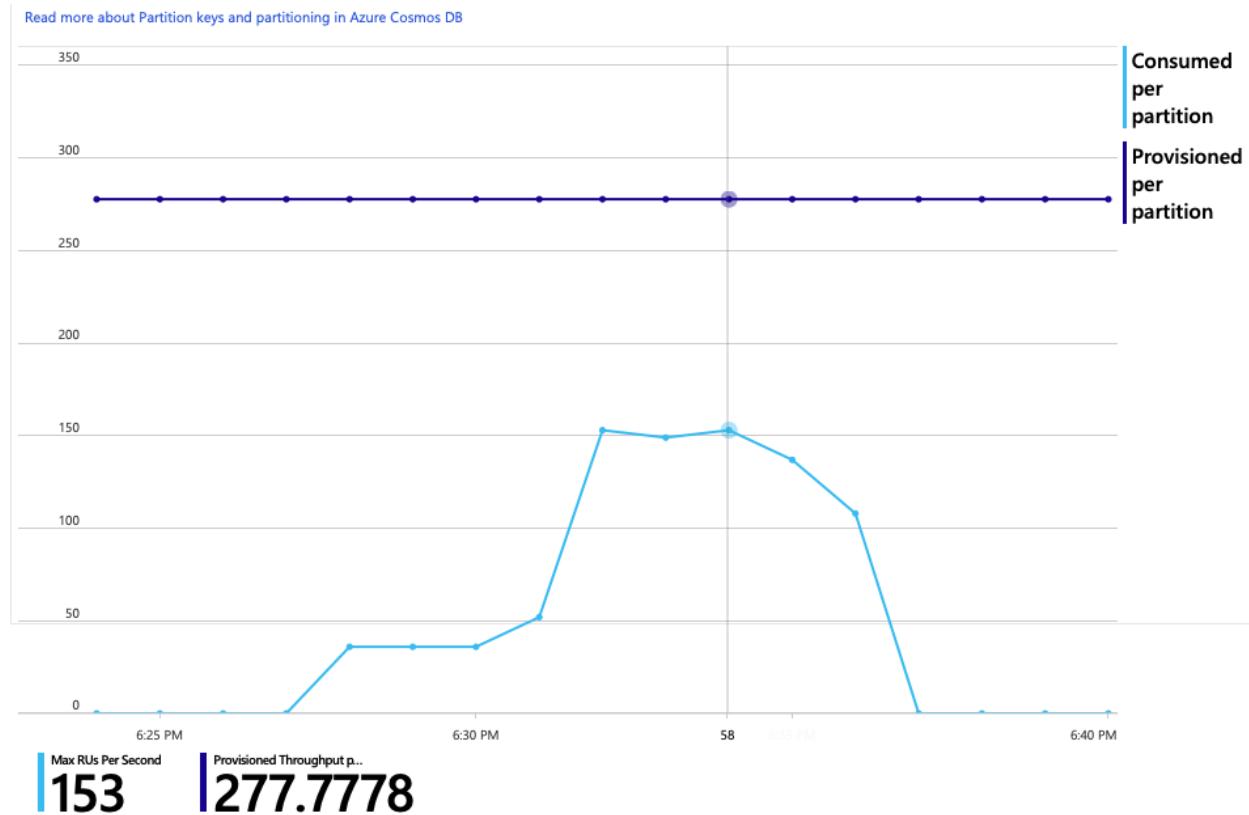


Whereas the previous test showed an initial spike followed by a sharp drop, this test shows more consistent throughput. However, the maximum throughput is not significantly higher.

All requests to Azure Cosmos DB returned a 2xx status, and the HTTP 429 errors went away:



The graph of RU consumption versus provisioned RUs shows there is plenty of headroom. There are about 275 RUs per physical partition, and the load test peaked at about 100 RUs consumed per second.



Another interesting metric is the number of calls to Azure Cosmos DB per successful operation:

[+] Expand table

Metric	Test 1	Test 2
Calls per operation	11	9

Assuming no errors, the number of calls should match the actual query plan. In this case, the operation involves a cross-partition query that hits all nine physical partitions. The higher value in the first load test reflects the number of calls that returned a 429 error.

This metric was calculated by running a custom Log Analytics query:

Kusto

```
let start=datetime("2020-06-18T20:59:00.000Z");
let end=datetime("2020-07-24T21:10:00.000Z");
let operationNameToEval="GET DroneDeliveries/GetDroneUtilization";
let dependencyType="Azure DocumentDB";
```

```

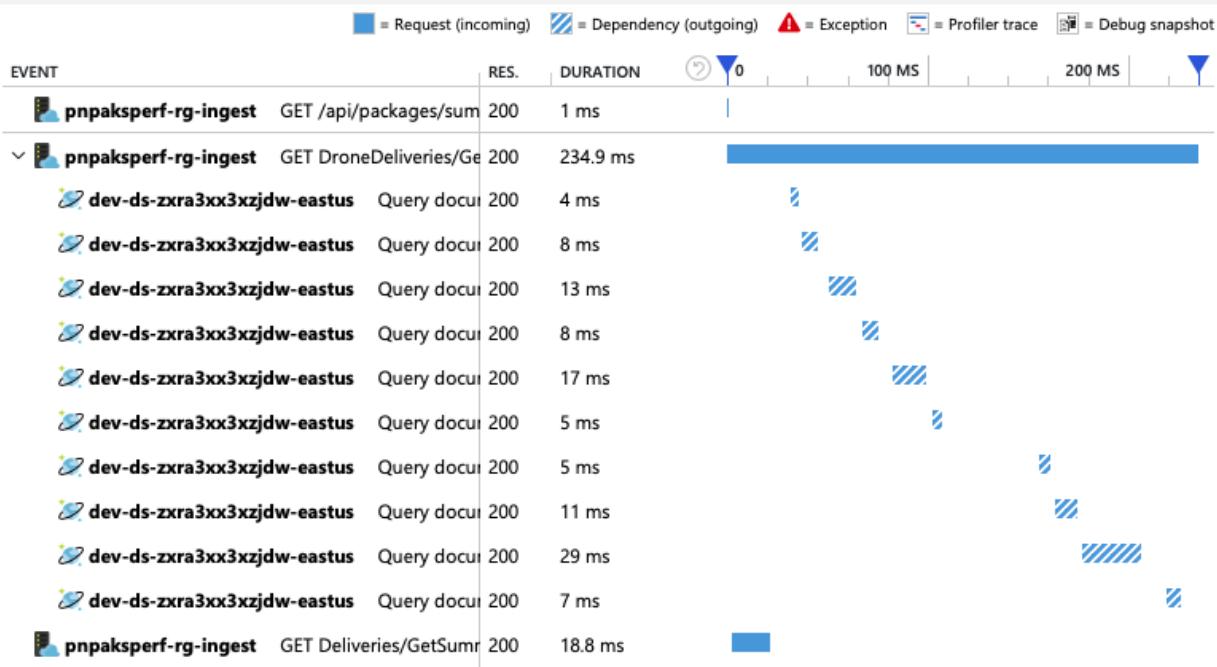
let dataset=requests
| where timestamp > start and timestamp < end
| where success == true
| where name == operationNameToEval;
dataset
| project reqOk=itemCount
| summarize
    SuccessRequests=sum(reqOk),
    TotalNumberOfDepCalls=(toscalar(dependencies
        | where timestamp > start and timestamp < end
        | where type == dependencyType
        | summarize sum(itemCount)))
| project
    OperationName=operationNameToEval,
    DependencyName=dependencyType,
    SuccessRequests,
    AverageNumberOfDepCallsPerOperation=
    (TotalNumberOfDepCalls/SuccessRequests)

```

To summarize, the second load test shows improvement. However, the `GetDroneUtilization` operation still takes about an order of magnitude longer than the next-slowest operation. Looking at the end-to-end transactions helps to explain why:

End-to-end transaction

Operation ID: a20dc1b40236cca6dcf3915c25c85607 



As mentioned earlier, the `GetDroneUtilization` operation involves a cross-partition query to Azure Cosmos DB. This means the Azure Cosmos DB client has to fan out the query to each physical partition and collect the results. As the end-to-end transaction view shows, these queries are being performed in serial. The operation takes as long as the sum of all the queries — and this problem will only get worse as the size of the data grows and more physical partitions are added.

Test 3: Parallel queries

Based on the previous results, an obvious way to reduce latency is to issue the queries in parallel. The Azure Cosmos DB client SDK has a setting that controls the maximum degree of parallelism.

[Expand table](#)

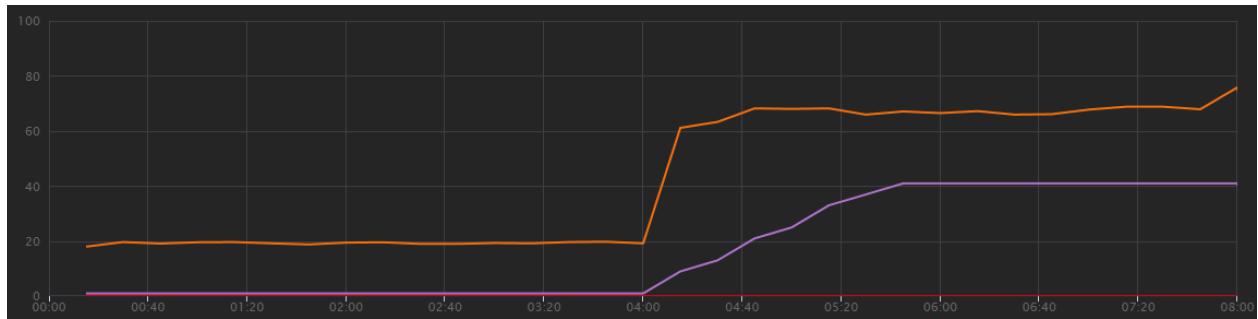
Value	Description
0	No parallelism (default)
> 0	Maximum number of parallel calls
-1	The client SDK selects an optimal degree of parallelism

For the third load test, this setting was changed from 0 to -1. The following table summarizes the results:

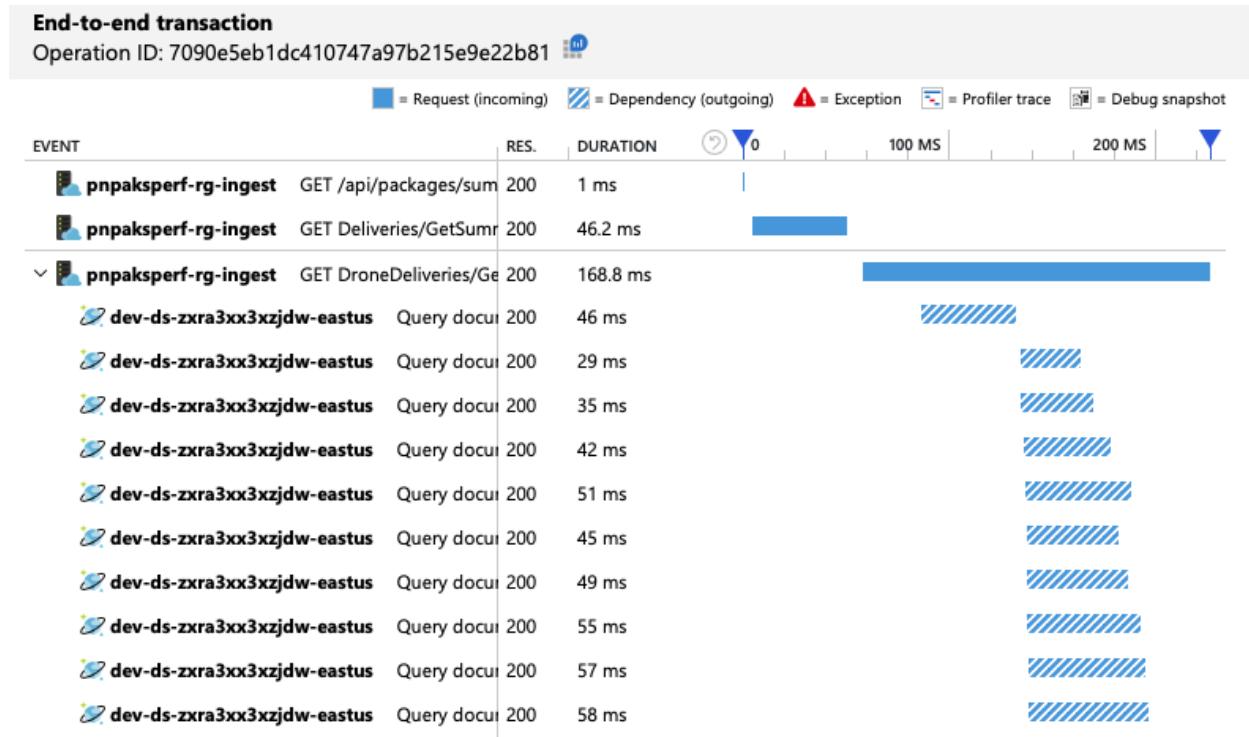
[Expand table](#)

Metric	Test 1	Test 2	Test 3
Throughput (req/sec)	19	23	42
Average latency (ms)	669	569	215
Successful requests	9.8 K	11 K	20 K
Throttled requests	2.72 K	0	0

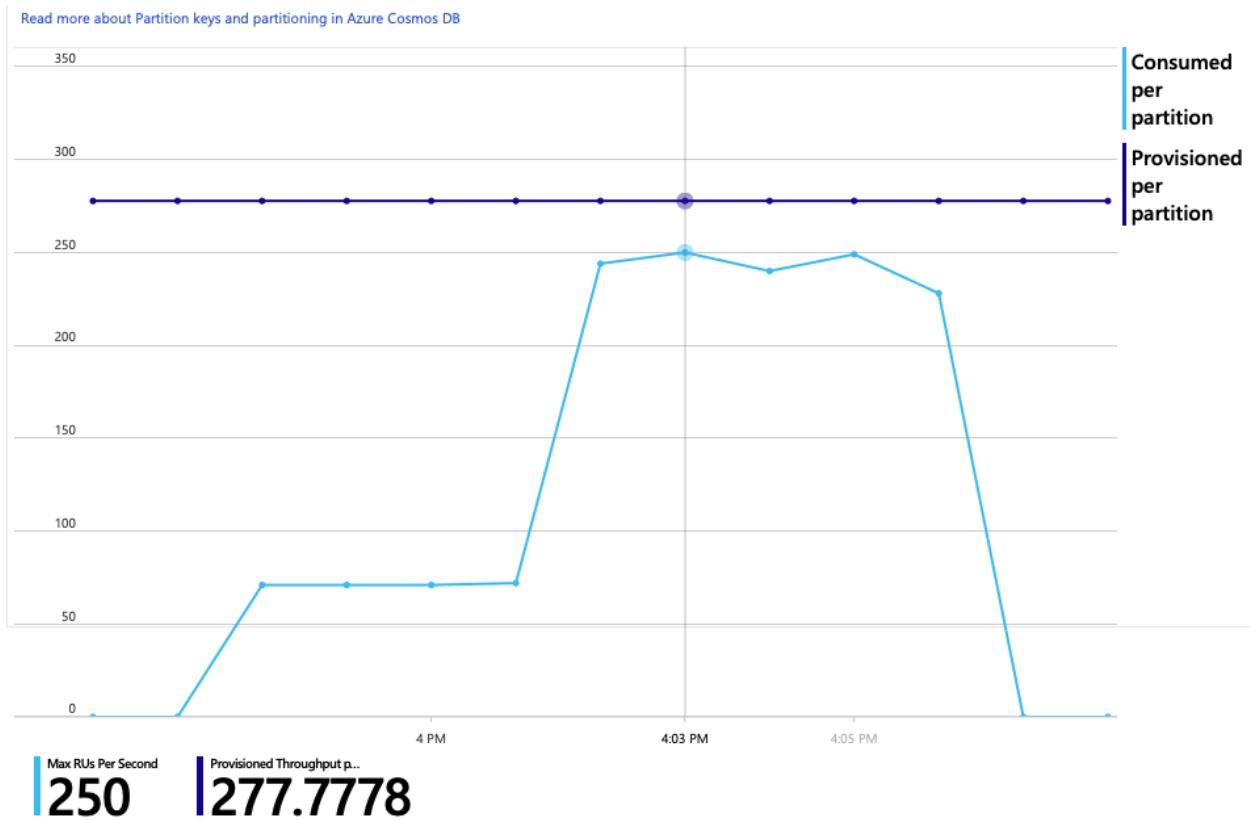
From the load test graph, not only is the overall throughput much higher (the orange line), but throughput also keeps pace with the load (the purple line).



We can verify that the Azure Cosmos DB client is making queries in parallel by looking at the end-to-end transaction view:



Interestingly, a side effect of increasing the throughput is that the number of RUs consumed per second also increases. Although Azure Cosmos DB did not throttle any requests during this test, the consumption was close to the provisioned RU limit:



This graph might be a signal to further scale out the database. However, it turns out that we can optimize the query instead.

Step 4: Optimize the query

The previous load test showed better performance in terms of latency and throughput. Average request latency was reduced by 68% and throughput increased 220%. However, the cross-partition query is a concern.

The problem with cross-partition queries is that you pay for RU across every partition. If the query is only run occasionally — say, once an hour — it might not matter. But whenever you see a read-heavy workload that involves a cross-partition query, you should see whether the query can be optimized by including a partition key. (You might need to redesign the collection to use a different partition key.)

Here's the query for this particular scenario:

SQL

```
SELECT * FROM c
WHERE c.ownerId = <ownerIdValue> and
    c.year = <yearValue> and
    c.month = <monthValue>
```

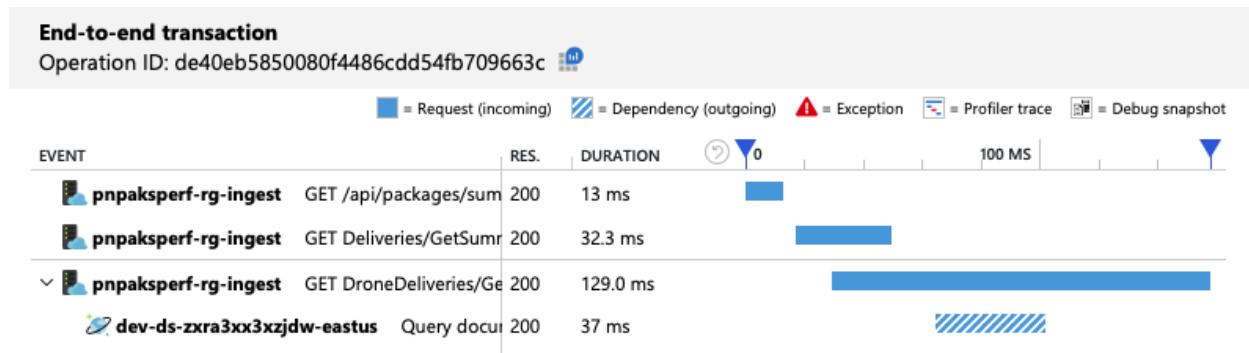
This query selects records that match a particular owner ID and month/year. In the original design, none of these properties is the partition key. That requires the client to fan out the query to each physical partition and gather the results. To improve query performance, the development team changed the design so that owner ID is the partition key for the collection. That way, the query can target a specific physical partition. (Azure Cosmos DB handles this automatically; you don't have to manage the mapping between partition key values and physical partitions.)

After switching the collection to the new partition key, there was a dramatic improvement in RU consumption, which translates directly into lower costs.

[\[+\] Expand table](#)

Metric	Test 1	Test 2	Test 3	Test 4
RUs per operation	29	29	29	3.4
Calls per operation	11	9	10	1

The end-to-end transaction view shows that as predicted, the query reads only one physical partition:



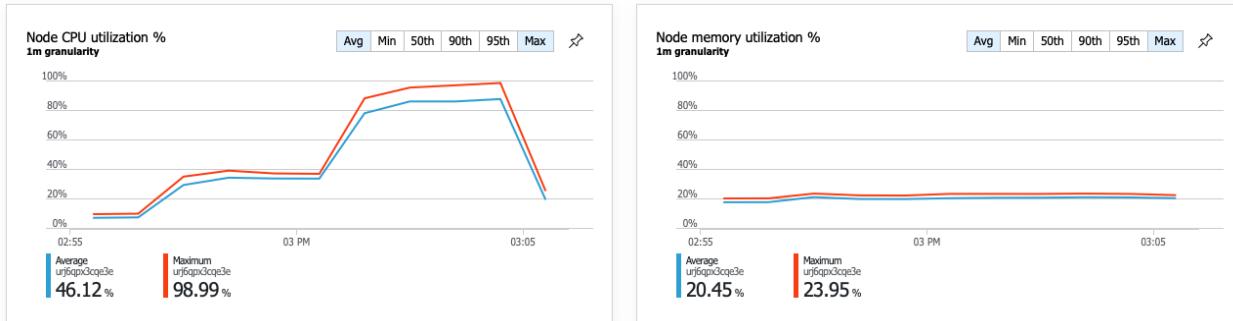
The load test shows improved throughput and latency:

[\[+\] Expand table](#)

Metric	Test 1	Test 2	Test 3	Test 4
Throughput (req/sec)	19	23	42	59
Average latency (ms)	669	569	215	176

Metric	Test 1	Test 2	Test 3	Test 4
Successful requests	9.8 K	11 K	20 K	29 K
Throttled requests	2.72 K	0	0	0

A consequence of the improved performance is that node CPU utilization becomes very high:



Toward the end of the load test, average CPU reached about 90%, and maximum CPU reached 100%. This metric indicates that CPU is the next bottleneck in the system. If higher throughput is needed, the next step might be scaling out the Delivery service to more instances.

Summary

For this scenario, the following bottlenecks were identified:

- Azure Cosmos DB throttling requests due to insufficient RUs provisioned.
- High latency caused by querying multiple database partitions in serial.
- Inefficient cross-partition query, because the query did not include the partition key.

In addition, CPU utilization was identified as a potential bottleneck at higher scale. To diagnose these issues, the development team looked at:

- Latency and throughput from the load test.
- Azure Cosmos DB errors and RU consumption.
- The end-to-end transaction view in Application Insight.
- CPU and memory utilization in Azure Monitor container insights.

Next steps

[Review performance antipatterns](#)

Performance tuning - Event streaming

Azure Functions

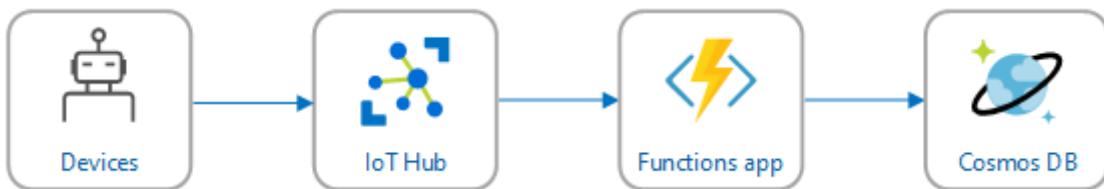
Azure IoT Hub

Azure Cosmos DB

This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that we did for a sample application.

This article is part of a series. Read the first part [here](#).

Scenario: Process a stream of events using Azure Functions.



In this scenario, a fleet of drones sends position data in real time to Azure IoT Hub. A Functions app receives the events, transforms the data into [GeoJSON](#) format, and writes the transformed data to Azure Cosmos DB. Azure Cosmos DB has native support for [geospatial data](#), and Azure Cosmos DB collections can be indexed for efficient spatial queries. For example, a client application could query for all drones within 1 km of a given location, or find all drones within a certain area.

These processing requirements are simple enough that they don't require a full-fledged stream processing engine. In particular, the processing doesn't join streams, aggregate data, or process across time windows. Based on these requirements, Azure Functions is a good fit for processing the messages. Azure Cosmos DB can also scale to support very high write throughput.

Monitoring throughput

This scenario presents an interesting performance challenge. The data rate *per device* is known, but the number of devices might fluctuate. For this business scenario, the latency requirements are not particularly stringent. The reported position of a drone only needs to be accurate within a minute. That said, the function app must keep up with the average ingestion rate over time.

IoT Hub stores messages in a log stream. Incoming messages are appended to the tail of the stream. A reader of the stream — in this case, the function app — controls its own rate of traversing the stream. This decoupling of the read and write paths makes IoT Hub very efficient, but also means that a slow reader can fall behind. To detect this condition, the development team added a custom metric to measure message lateness. This metric records the delta between when a message arrives at IoT Hub, and when the function receives the message for processing.

C#

```
var ticksUTCNow = DateTimeOffset.UtcNow;

// Track whether messages are arriving at the function late.
DateTime? firstMsgEnqueuedTicksUtc = messages[0]?.EnqueuedTimeUtc;
if (firstMsgEnqueuedTicksUtc.HasValue)
{
    CustomTelemetry.TrackMetric(
        context,
        "IoTHubMessagesReceivedFreshnessMsec",
        (ticksUTCNow -
    firstMsgEnqueuedTicksUtc.Value).TotalMilliseconds);
}
```

The `TrackMetric` method writes a custom metric to Application Insights. For information about using `TrackMetric` inside an Azure Function, see [Custom telemetry in C# function](#).

If the function keeps up with the volume of messages, this metric should stay at a low steady state. Some latency is unavoidable, so the value will never be zero. But if the function falls behind, the delta between enqueued time and processing time will start to go up.

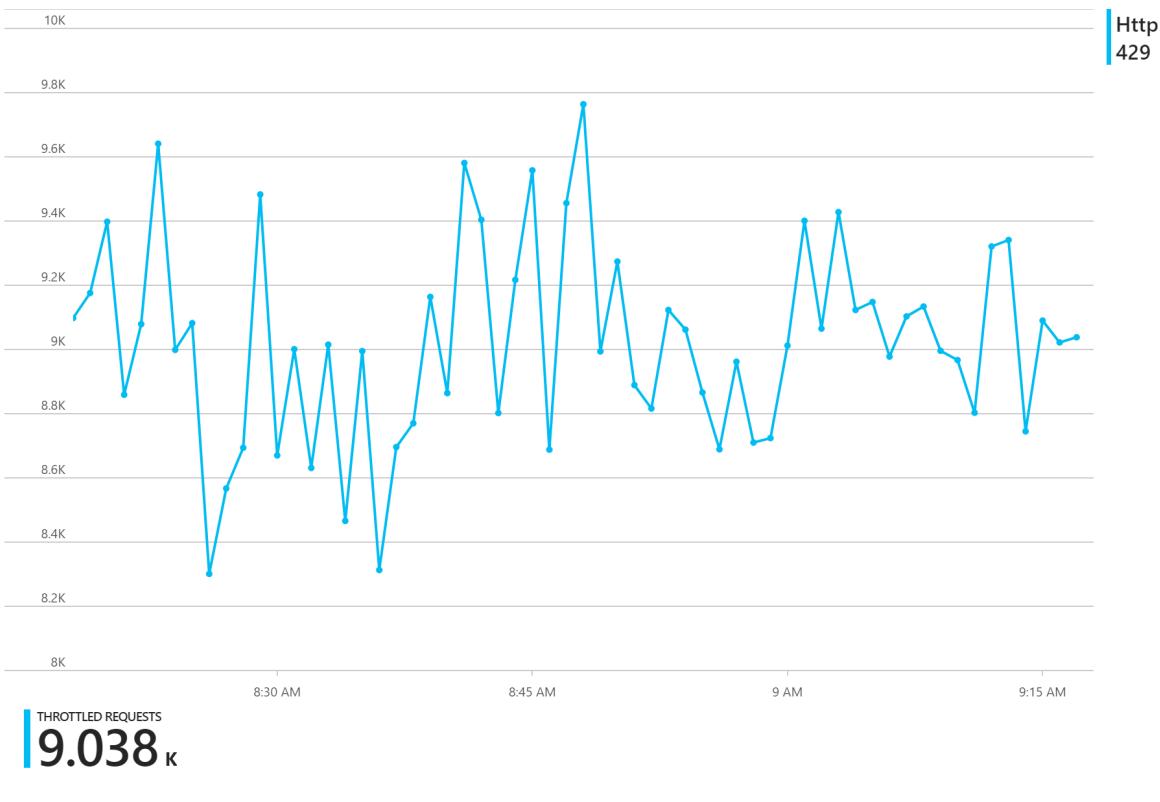
Test 1: Baseline

The first load test showed an immediate problem: The Function app consistently received HTTP 429 errors from Azure Cosmos DB, indicating that Azure Cosmos DB was throttling the write requests.

Number of requests exceeded capacity (aggregated over 1 minute interval)

↗

Number of requests that failed due to exceeding throughput or storage capacity provisioned for the collection.



In response, the team scaled Azure Cosmos DB by increasing the number RUs allocated for the collection, but the errors continued. This seemed strange, because their back-of-envelope calculation showed that Azure Cosmos DB should have no problem keeping up with the volume of write requests.

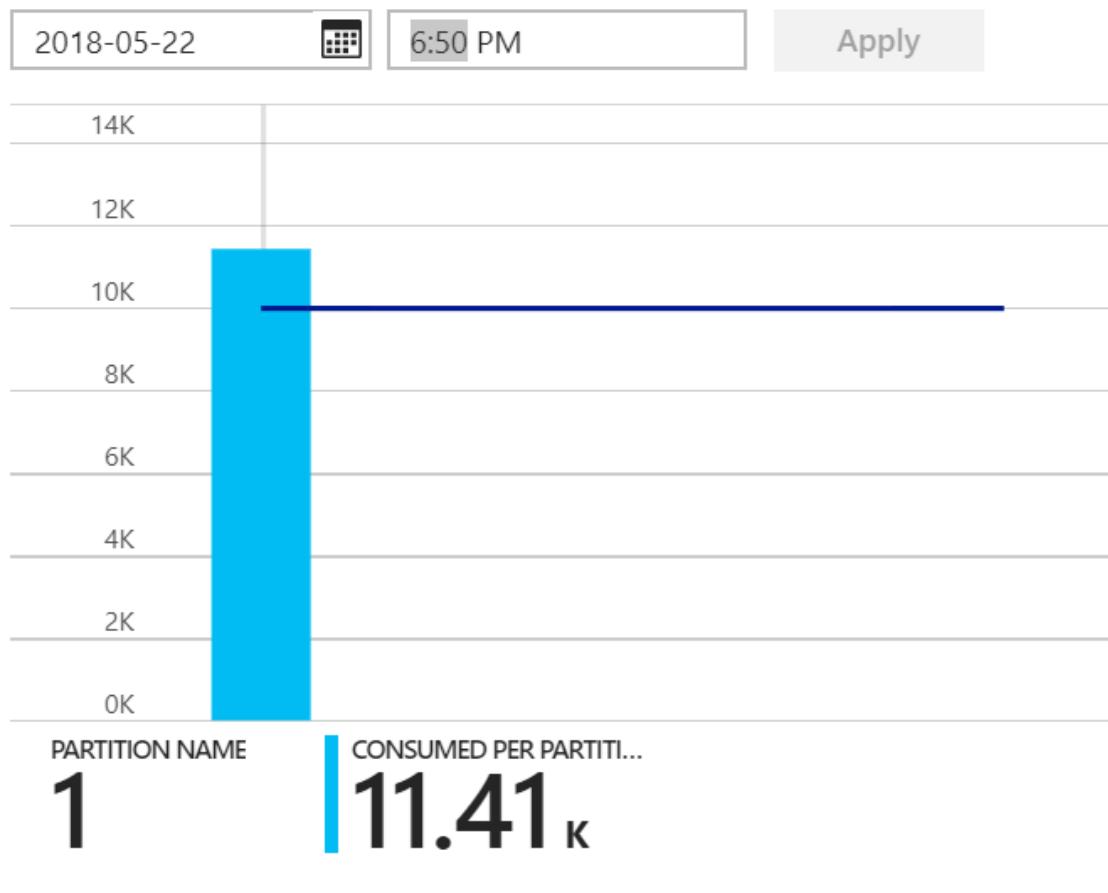
Later that day, one of the developers sent the following email to the team:

I looked at Azure Cosmos DB for the warm path. There's one thing I don't understand. The partition key is deliveryId, however we don't send deliveryId to Azure Cosmos DB. Am I missing something?

That was the clue. Looking at the partition heat map, it turned out that all of the documents were landing on the same partition.

Max consumed RU/s by each partition key range at 5/22/2018, 6:50:38 PM ⓘ

Select partition to view top selected partition keys for respective partition.



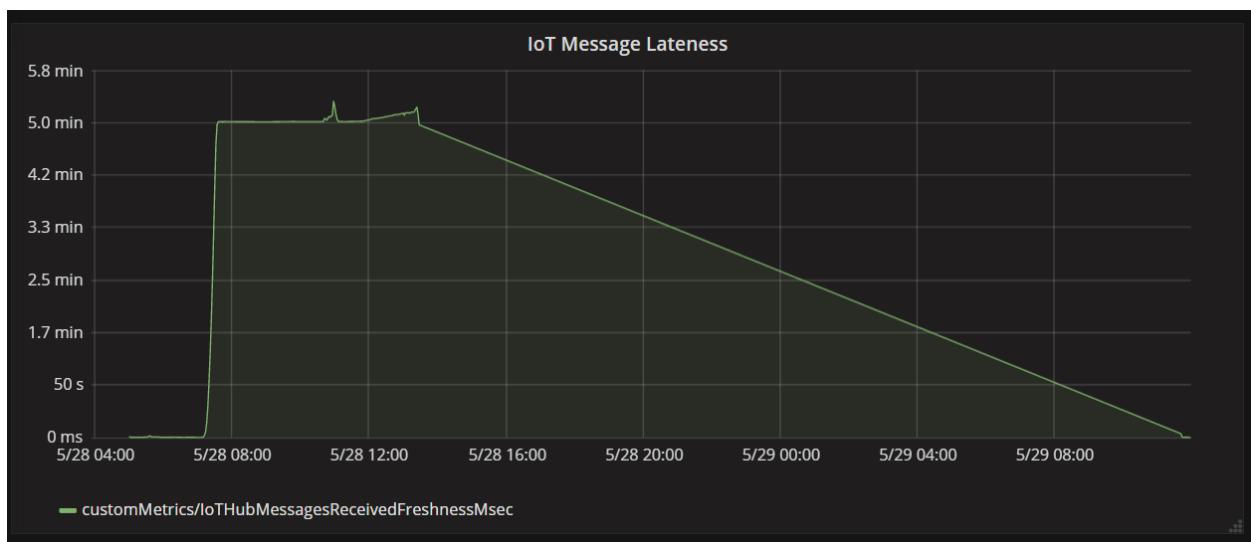
What you want to see in the heat map is an even distribution across all of the partitions. In this case, because every document was getting written to the same partition, adding RUs didn't help. The problem turned out to be a bug in the code. Although the Azure Cosmos DB collection had a partition key, the Azure Function didn't actually include the partition key in the document. For more information about the partition heat map, see [Determine the throughput distribution across partitions](#).

Test 2: Fix partitioning issue

When the team deployed a code fix and re-ran the test, Azure Cosmos DB stopped throttling. For a while, everything looked good. But at a certain load, telemetry showed that the function was writing fewer documents than it should. The following graph shows messages receives from IoT Hub versus documents written to Azure Cosmos DB. The yellow line is number of messages received per batch, and the green is the number of documents written per batch. These should be proportional. Instead, the number of database write operations per batch drops significantly at about 07:30.



The next graph shows the latency between when a message arrives at IoT Hub from a device, and when the function app processes that message. You can see that at the same point in time, the lateness spikes dramatically, levels off, and then declines.



The reason the value peaks at 5 minutes and then drops to zero is because the function app discards messages that are more than 5 minutes late:

C#

```

foreach (var message in messages)
{
    // Drop stale messages,
    if (message.EnqueueTimeUtc < cutoffTime)
    {
        log.Info($"Dropping late message batch. Enqueued time =
{message.EnqueueTimeUtc}, Cutoff = {cutoffTime}");
        droppedMessages++;
        continue;
    }
}

```

You can see this in the graph when the lateness metric drops back to zero. In the meantime, data has been lost, because the function was throwing away messages.

What was happening? For this particular load test, the Azure Cosmos DB collection had RUs to spare, so the bottleneck was not at the database. Rather, the problem was in the message processing loop. Simply put, the function was not writing documents quickly enough to keep up with the incoming volume of messages. Over time, it fell further and further behind.

Test 3: Parallel writes

If the time to process a message is the bottleneck, one solution is to process more messages in parallel. In this scenario:

- Increase the number of IoT Hub partitions. Each IoT Hub partition gets assigned one function instance at a time, so we would expect throughput to scale linearly with the number of partitions.
- Parallelize the document writes within the function.

To explore the second option, the team modified the function to support parallel writes. The original version of the function used the Azure Cosmos DB [output binding](#). The optimized version calls the Azure Cosmos DB client directly and performs the writes in parallel using [Task.WhenAll](#):

C#

```
private async Task<(long documentsUpserted,
                     long droppedMessages,
                     long cosmosDbTotalMilliseconds)>
    ProcessMessagesFromEventHub(
        int taskCount,
        int numberOfDocumentsToUpsertPerTask,
        EventData[] messages,
        TraceWriter log)
{
    DateTimeOffset cutoffTime = DateTimeOffset.UtcNow.AddMinutes(-5);

    var tasks = new List<Task>();

    for (var i = 0; i < taskCount; i++)
    {
        var docsToUpsert = messages
            .Skip(i * numberOfDocumentsToUpsertPerTask)
            .Take(numberOfDocumentsToUpsertPerTask);
        // client will attempt to create connections to the data
        // nodes on Azure Cosmos DB clusters on a range of port numbers
        tasks.Add(UpsertDocuments(i, docsToUpsert, cutoffTime, log));
    }
}
```

```
    }

    await Task.WhenAll(tasks);

    return (this.UpsertedDocuments,
            this.DroppedMessages,
            this.CosmosDbTotalMilliseconds);
}
```

Note that race conditions are possible with approach. Suppose that two messages from the same drone happen to arrive in the same batch of messages. By writing them in parallel, the earlier message could overwrite the later message. For this particular scenario, the application can tolerate losing an occasional message. Drones send new position data every 5 seconds, so the data in Azure Cosmos DB is updated continually. In other scenarios, however, it may be important to process messages strictly in order.

After deploying this code change, the application was able to ingest more than 2500 requests/sec, using an IoT Hub with 32 partitions.

Client-side considerations

Overall client experience might be diminished by aggressive parallelization on server side. Consider using [Azure Cosmos DB bulk executor library](#) (not shown in this implementation) which significantly reduces the client-side compute resources needed to saturate the throughput allocated to an Azure Cosmos DB container. A single threaded application that writes data using the bulk import API achieves nearly ten times greater write throughput when compared to a multi-threaded application that writes data in parallel while saturating the client machine's CPU.

Summary

For this scenario, the following bottlenecks were identified:

- Hot write partition, due to a missing partition key value in the documents being written.
- Writing documents in serial per IoT Hub partition.

To diagnose these issues, the development team relied on the following metrics:

- Throttled requests in Azure Cosmos DB.
- Partition heat map — Maximum consumed RUs per partition.
- Messages received versus documents created.
- Message lateness.

Next steps

Review [performance antipatterns](#)

Performance testing and antipatterns for cloud applications

Article • 12/13/2023

Performance antipatterns, much like design patterns, are common defective processes and implementations within organizations. These are common practices that are likely to cause scalability problems when an application is under pressure. Awareness of these practices can help simplify communication of high-level concepts amongst software practitioners, and knowledge of antipatterns can be helpful when reviewing code or diagnosing performance issues.

Here's a common scenario: An application behaves well during performance testing. It's released to production, and begins to handle real workloads. At that point, it starts to perform poorly—rejecting user requests, stalling, or throwing exceptions. The development team is then faced with two questions:

- Why didn't this behavior show up during testing?
- How do we fix it?

The answer to the first question is straightforward. It's difficult to simulate real users in a test environment, along with their behavior patterns and the volumes of work they might perform. The only completely sure way to understand how a system behaves under load is to observe it in production. To be clear, we aren't suggesting that you should skip performance testing. Performance testing is crucial for getting baseline performance metrics. But you must be prepared to observe and correct performance issues when they arise in the live system.

The answer to the second question, how to fix the problem, is less straightforward. Any number of factors might contribute, and sometimes the problem only manifests under certain circumstances. Instrumentation and logging are key to finding the root cause, but you also have to know what to look for.

Based on our engagements with Microsoft Azure customers, we've identified some of the most common performance issues that customers see in production. For each antipattern, we describe why the antipattern typically occurs, symptoms of the antipattern, and techniques for resolving the problem. We also provide sample code that illustrates both the antipattern and a suggested scalability solution.

Some of these antipatterns might seem obvious when you read the descriptions, but they occur more often than you might think. Sometimes an application inherits a design that worked on-premises, but doesn't scale in the cloud. Or an application might start

with a very clean design, but as new features are added, one or more of these antipatterns creeps in. Regardless, this guide will help you to identify and fix these antipatterns.

Catalog of antipatterns

Here's the list of the antipatterns that we've identified:

[] Expand table

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.
No Caching	Failing to cache data.
Noisy Neighbor	A single tenant uses a disproportionate amount of the resources.
Retry Storm	Retrying failed requests to a server too often.
Synchronous I/O	Blocking the calling thread while I/O completes.

Next steps

For more about performance tuning, see [Performance tuning a distributed application](#)

Busy Database antipattern

Article • 12/16/2022

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 ↗](#).

SQL

```
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.

SQL

```
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.

C#

```
// Create a new SqlCommand to run the Transact-SQL query
using (var command = new SqlCommand(...))
{
    command.Parameters.AddWithValue("@TerritoryId", id);

    // Run the query and create the initial XML document
    using (var reader = await command.ExecuteReaderAsync())
    {
        var lastOrderNumber = string.Empty;
        var doc = new XDocument();
        var orders = new XElement("Orders");
        doc.Add(orders);

        XElement lineItems = null;
        // Fetch each row in turn, format the results as XML, and add them
        to the XML document
        while (await reader.ReadAsync())
        {
            var orderNumber = reader["OrderNumber"].ToString();
            if (orderNumber != lastOrderNumber)
            {
                lastOrderNumber = orderNumber;

                var order = new XElement("Order");
                orders.Add(order);
                var customer = new XElement("Customer");
                lineItems = new XElement("OrderLineItems");
                order.Add(customer, lineItems);

                var orderDate = (DateTime)reader["OrderDate"];
                var totalDue = (Decimal)reader["TotalDue"];
                var reviewRequired = totalDue > 5000 ? 'Y' : 'N';

                order.Add(
                    new XAttribute("OrderNumber", orderNumber),
                    new XAttribute("Status", reader["Status"]),
                    new XAttribute("ShipDate", reader["ShipDate"]),
                    ... // More attributes, not shown.

                    var fullName = string.Join(" ",
                    reader["CustomerTitle"],
                    reader["CustomerFirstName"],
                    reader["CustomerMiddleName"],
                    reader["CustomerLastName"],
                    reader["CustomerSuffix"])
                )
                .Replace(" ", " ") //remove double spaces
                .Trim()
                .ToUpper();

                customer.Add(
                    new XAttribute("AccountNumber",
```

```

        reader["AccountNumber"]),
            new XAttribute("FullName", fullName));
    }

    var productId = (int)reader["ProductID"];
    var quantity = (short)reader["Quantity"];
    var inventoryCheckRequired = (productId >= 710 && productId <=
720 && quantity >= 5) ? 'Y' : 'N';

    lineItems.Add(
        new XElement("LineItem",
            new XAttribute("Quantity", quantity),
            new XAttribute("UnitPrice",
((Decimal)reader["UnitPrice"]).ToString("C")),
            new XAttribute("LineTotal",
RoundAndFormat(reader["LineTotal"])),
            new XAttribute("ProductId", productId),
            new XAttribute("InventoryCheckRequired",
inventoryCheckRequired)
        ));
    }

    // Match the exact formatting of the XML returned from SQL
    var xml = doc
        .ToString(SaveOptions.DisableFormatting)
        .Replace(" />", "/>");
}
}

```

Note

This code is somewhat complex. For a new application, you might prefer to use a serialization library. However, the assumption here is that the development team is refactoring an existing application, so the method needs to return the exact same format as the original code.

Considerations

- Many database systems are highly optimized to perform certain types of data processing, such as calculating aggregate values over large datasets. Don't move those types of processing out of the database.
- Do not relocate processing if doing so causes the database to transfer far more data over the network. See the [Extraneous Fetching antipattern](#).
- If you move processing to an application tier, that tier may need to scale out to handle the additional work.

How to detect the problem

Symptoms of a busy database include a disproportionate decline in throughput and response times in operations that access the database.

You can perform the following steps to help identify this problem:

1. Use performance monitoring to identify how much time the production system spends performing database activity.
2. Examine the work performed by the database during these periods.
3. If you suspect that particular operations might cause too much database activity, perform load testing in a controlled environment. Each test should run a mixture of the suspect operations with a variable user load. Examine the telemetry from the load tests to observe how the database is used.
4. If the database activity reveals significant processing but little data traffic, review the source code to determine whether the processing can better be performed elsewhere.

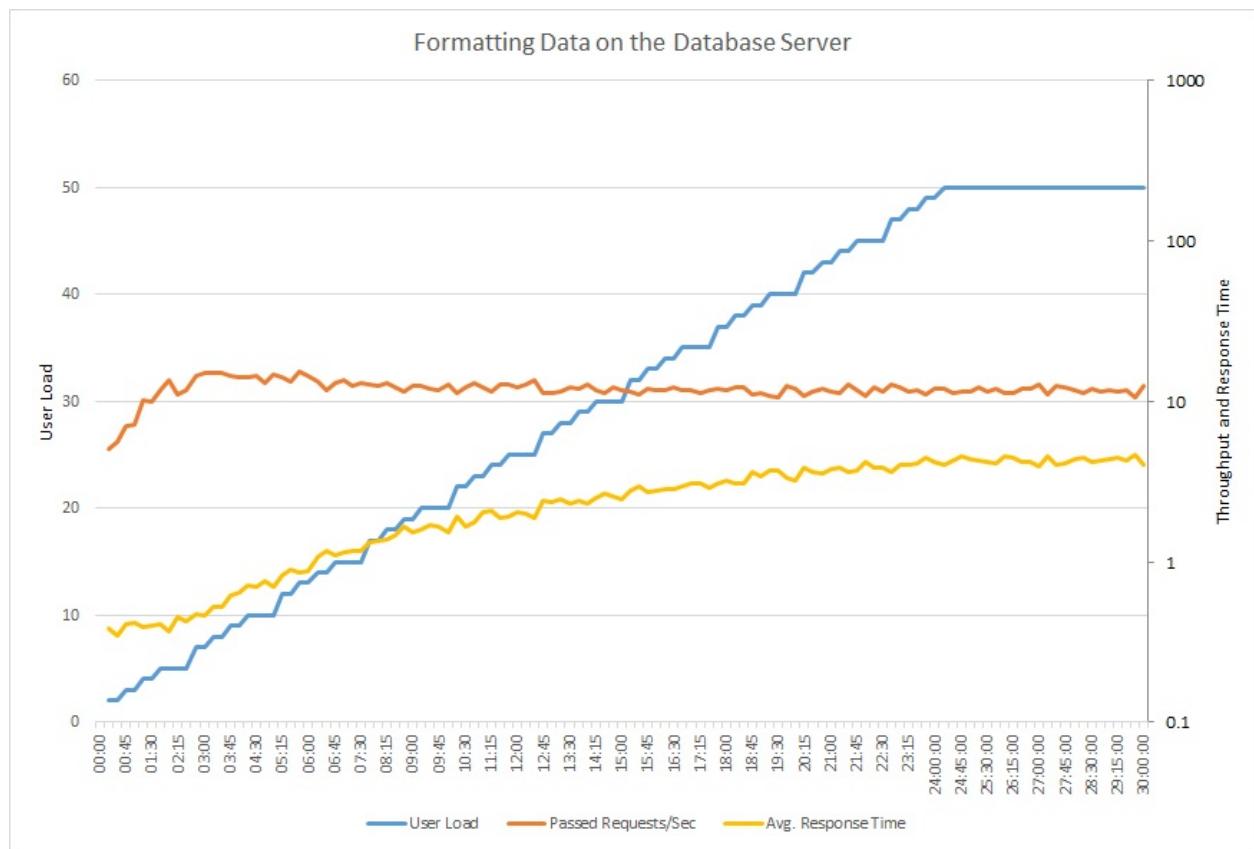
If the volume of database activity is low or response times are relatively fast, then a busy database is unlikely to be a performance problem.

Example diagnosis

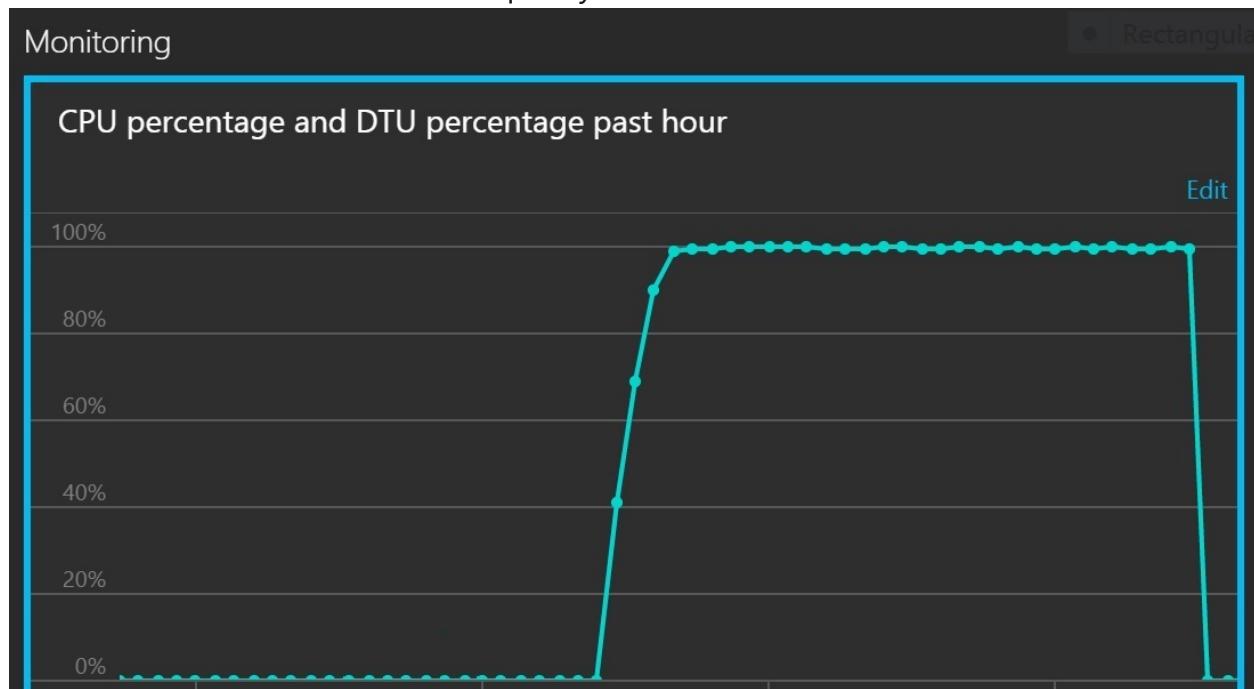
The following sections apply these steps to the sample application described earlier.

Monitor the volume of database activity

The following graph shows the results of running a load test against the sample application, using a step load of up to 50 concurrent users. The volume of requests quickly reaches a limit and stays at that level, while the average response time steadily increases. A logarithmic scale is used for those two metrics.



The next graph shows CPU utilization and DTUs as a percentage of service quota. DTUs provide a measure of how much processing the database performs. The graph shows that CPU and DTU utilization both quickly reached 100%.



Examine the work performed by the database

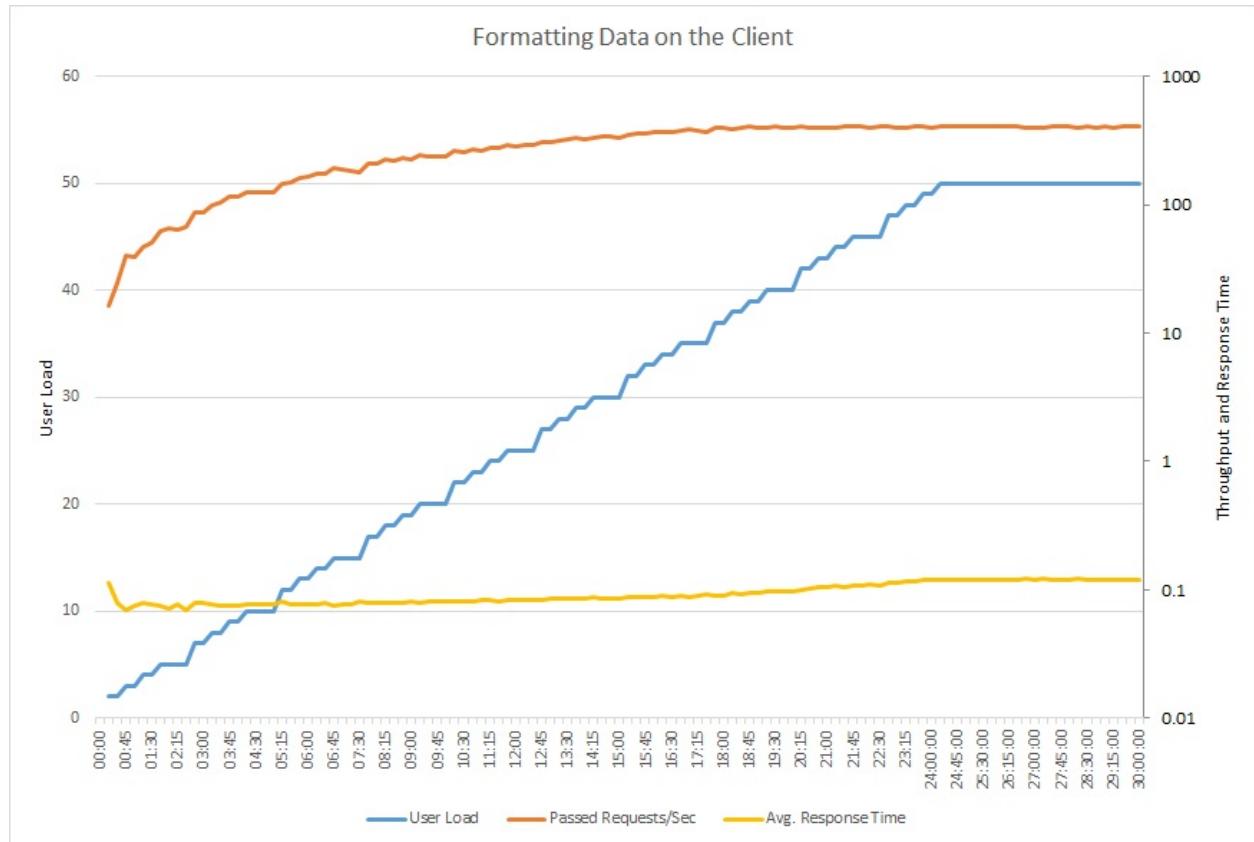
It could be that the tasks performed by the database are genuine data access operations, rather than processing, so it is important to understand the SQL statements

being run while the database is busy. Monitor the system to capture the SQL traffic and correlate the SQL operations with application requests.

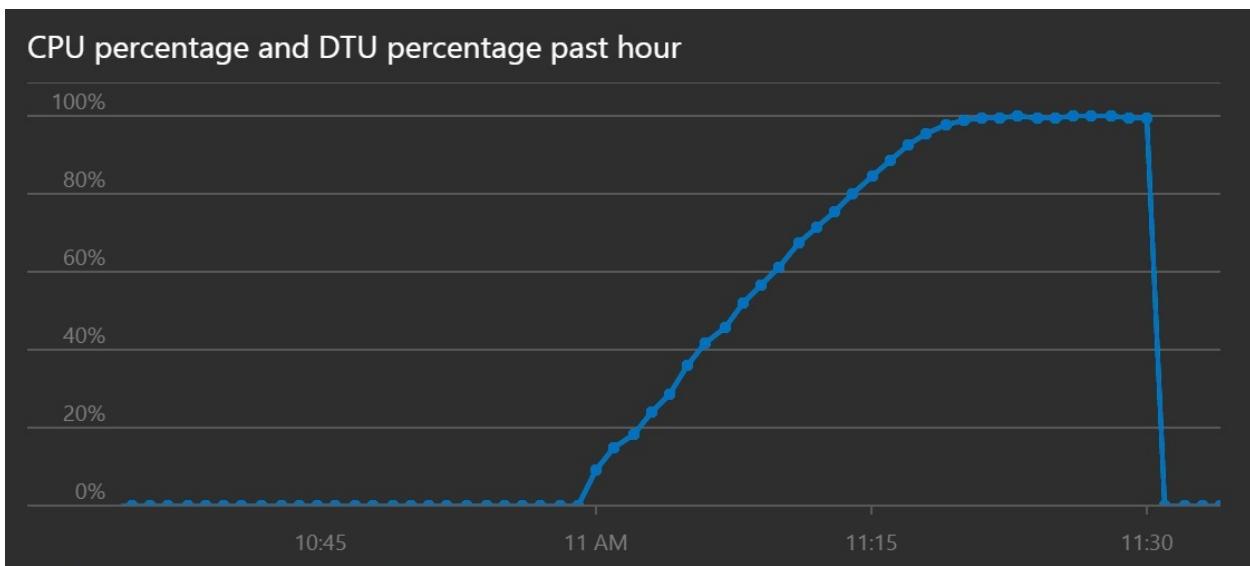
If the database operations are purely data access operations, without a lot of processing, then the problem might be [Extraneous Fetching](#).

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

Article • 12/16/2022

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 ↗](#).

C#

```
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.

C#

```

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.

C#

```

public async Task RunAsync(CancellationToken cancellationToken)
{
    this._queueClient.OnMessageAsync(
        // This lambda is invoked for each message received.
        async (receivedMessage) =>
    {
        try
        {
            // Simulate processing of message
            Thread.SpinWait(Int32.MaxValue / 1000);

            await receivedMessage.CompleteAsync();
        }
        catch
        {
            receivedMessage.Abandon();
        }
    });
}

```

Considerations

- This approach adds some additional complexity to the application. You must handle queuing and dequeuing safely to avoid losing requests in the event of a failure.
- The application takes a dependency on an additional service for the message queue.
- The processing environment must be sufficiently scalable to handle the expected workload and meet the required throughput targets.
- While this approach should improve overall responsiveness, the tasks that are moved to the back end may take longer to complete.

How to detect the problem

Symptoms of a busy front end include high latency when resource-intensive tasks are being performed. End users are likely to report extended response times or failures caused by services timing out. These failures could also return HTTP 500 (Internal Server) errors or HTTP 503 (Service Unavailable) errors. Examine the event logs for the web server, which are likely to contain more detailed information about the causes and circumstances of the errors.

You can perform the following steps to help identify this problem:

1. Perform process monitoring of the production system, to identify points when response times slow down.
2. Examine the telemetry data captured at these points to determine the mix of operations being performed and the resources being used.
3. Find any correlations between poor response times and the volumes and combinations of operations that were happening at those times.
4. Load test each suspected operation to identify which operations are consuming resources and starving other operations.
5. Review the source code for those operations to determine why they might cause excessive resource consumption.

Example diagnosis

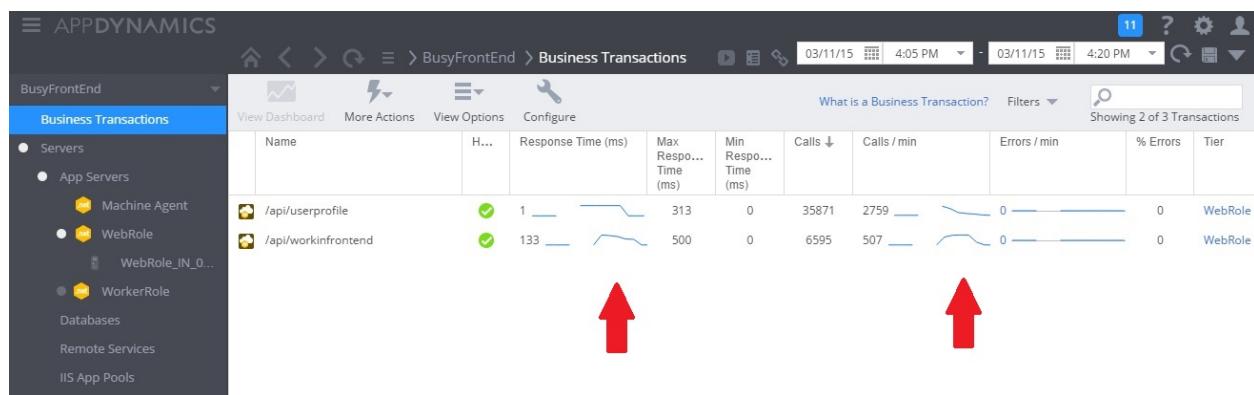
The following sections apply these steps to the sample application described earlier.

Identify points of slowdown

Instrument each method to track the duration and resources consumed by each request. Then monitor the application in production. This can provide an overall view of how

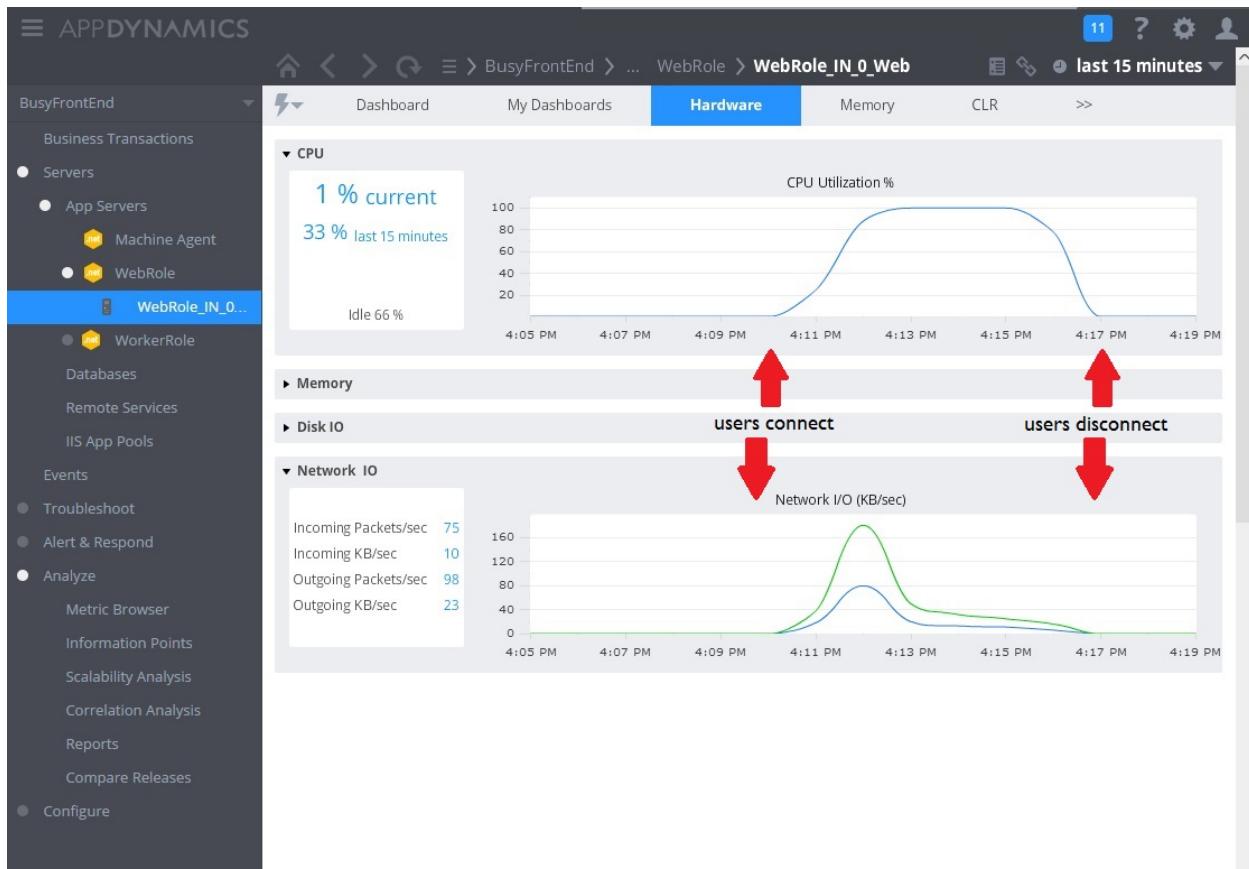
requests compete with each other. During periods of stress, slow-running resource-hungry requests will likely affect other operations, and this behavior can be observed by monitoring the system and noting the drop off in performance.

The following image shows a monitoring dashboard. (We used [AppDynamics](#) for our tests.) Initially, the system has light load. Then users start requesting the `UserProfile` GET method. The performance is reasonably good until other users start issuing requests to the `WorkInFrontEnd` POST method. At that point, response times increase dramatically (first arrow). Response times only improve after the volume of requests to the `WorkInFrontEnd` controller diminishes (second arrow).



Examine telemetry data and find correlations

The next image shows some of the metrics gathered to monitor resource utilization during the same interval. At first, few users are accessing the system. As more users connect, CPU utilization becomes very high (100%). Also notice that the network I/O rate initially goes up as CPU usage rises. But once CPU usage peaks, network I/O actually goes down. That's because the system can only handle a relatively small number of requests once the CPU is at capacity. As users disconnect, the CPU load tails off.

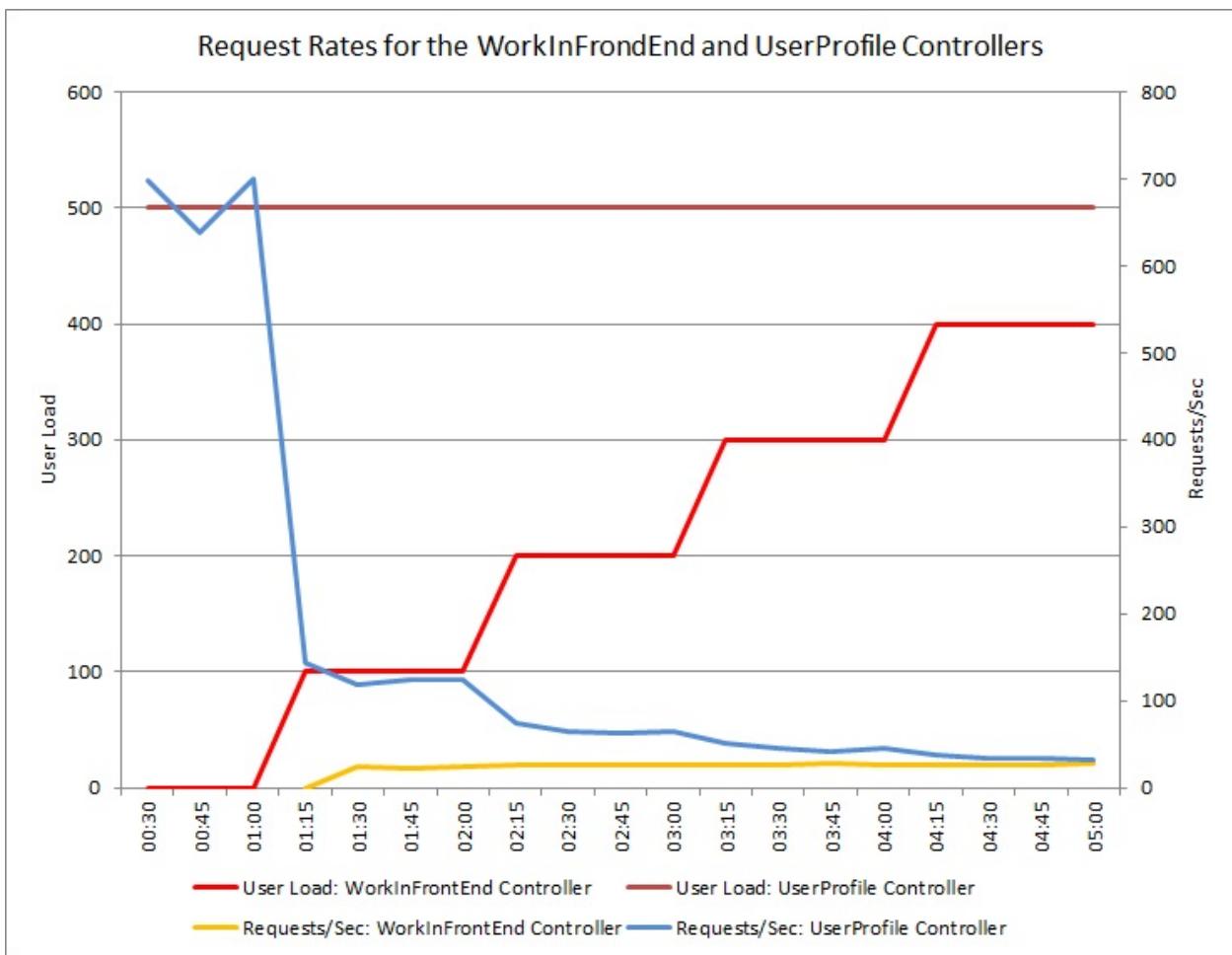


At this point, it appears the `Post` method in the `WorkInFrontEnd` controller is a prime candidate for closer examination. Further work in a controlled environment is needed to confirm the hypothesis.

Perform load testing

The next step is to perform tests in a controlled environment. For example, run a series of load tests that include and then omit each request in turn to see the effects.

The graph below shows the results of a load test performed against an identical deployment of the cloud service used in the previous tests. The test used a constant load of 500 users performing the `Get` operation in the `UserProfile` controller, along with a step load of users performing the `Post` operation in the `WorkInFrontEnd` controller.



Initially, the step load is 0, so the only active users are performing the `UserProfile` requests. The system is able to respond to approximately 500 requests per second. After 60 seconds, a load of 100 additional users starts sending POST requests to the `WorkInFrontEnd` controller. Almost immediately, the workload sent to the `UserProfile` controller drops to about 150 requests per second. This is due to the way the load-test runner functions. It waits for a response before sending the next request, so the longer it takes to receive a response, the lower the request rate.

As more users send POST requests to the `WorkInFrontEnd` controller, the response rate of the `UserProfile` controller continues to drop. But note that the volume of requests handled by the `WorkInFrontEnd` controller remains relatively constant. The saturation of the system becomes apparent as the overall rate of both requests tends toward a steady but low limit.

Review the source code

The final step is to look at the source code. The development team was aware that the `Post` method could take a considerable amount of time, which is why the original implementation used a separate thread. That solved the immediate problem, because the `Post` method did not block waiting for a long-running task to complete.

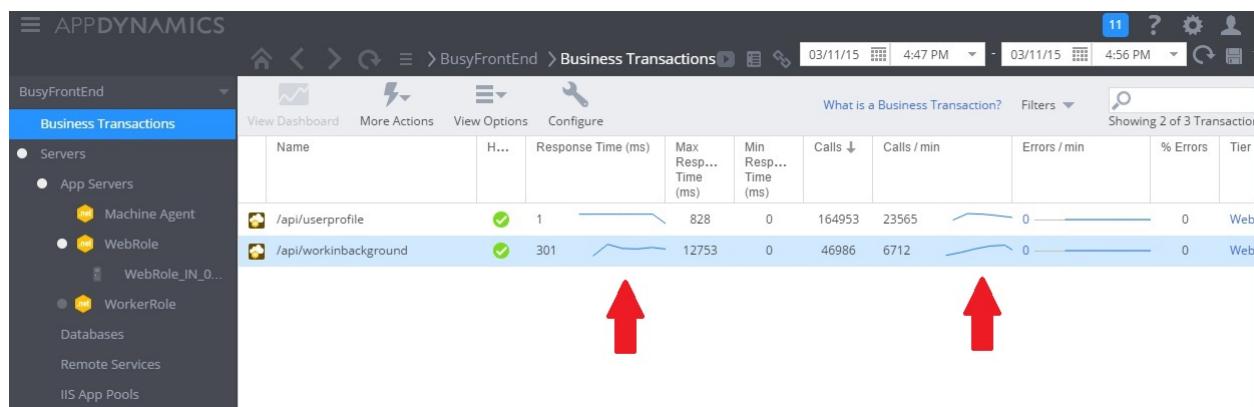
However, the work performed by this method still consumes CPU, memory, and other resources. Enabling this process to run asynchronously might actually damage performance, as users can trigger a large number of these operations simultaneously, in an uncontrolled manner. There is a limit to the number of threads that a server can run. Past this limit, the application is likely to get an exception when it tries to start a new thread.

ⓘ Note

This doesn't mean you should avoid asynchronous operations. Performing an asynchronous await on a network call is a recommended practice. (See the [Synchronous I/O antipattern](#).) The problem here is that CPU-intensive work was spawned on another thread.

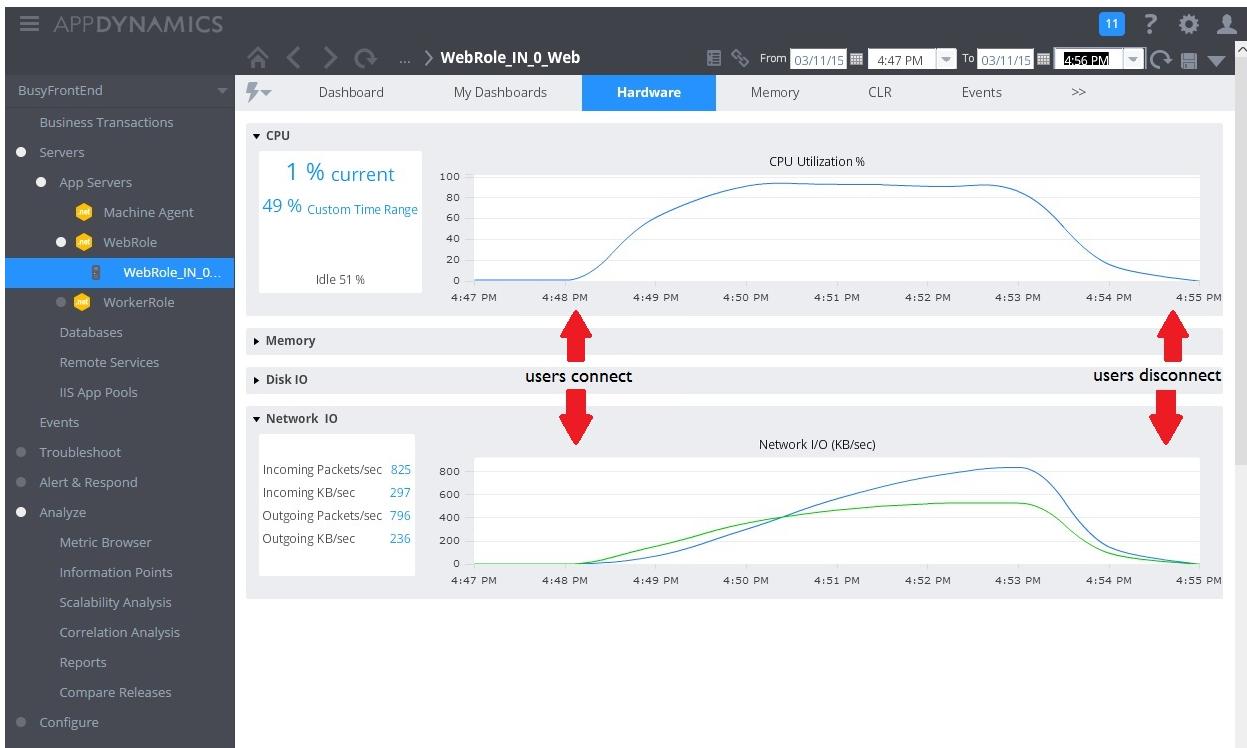
Implement the solution and verify the result

The following image shows performance monitoring after the solution was implemented. The load was similar to that shown earlier, but the response times for the `UserProfile` controller are now much faster. The volume of requests increased over the same duration, from 2,759 to 23,565.

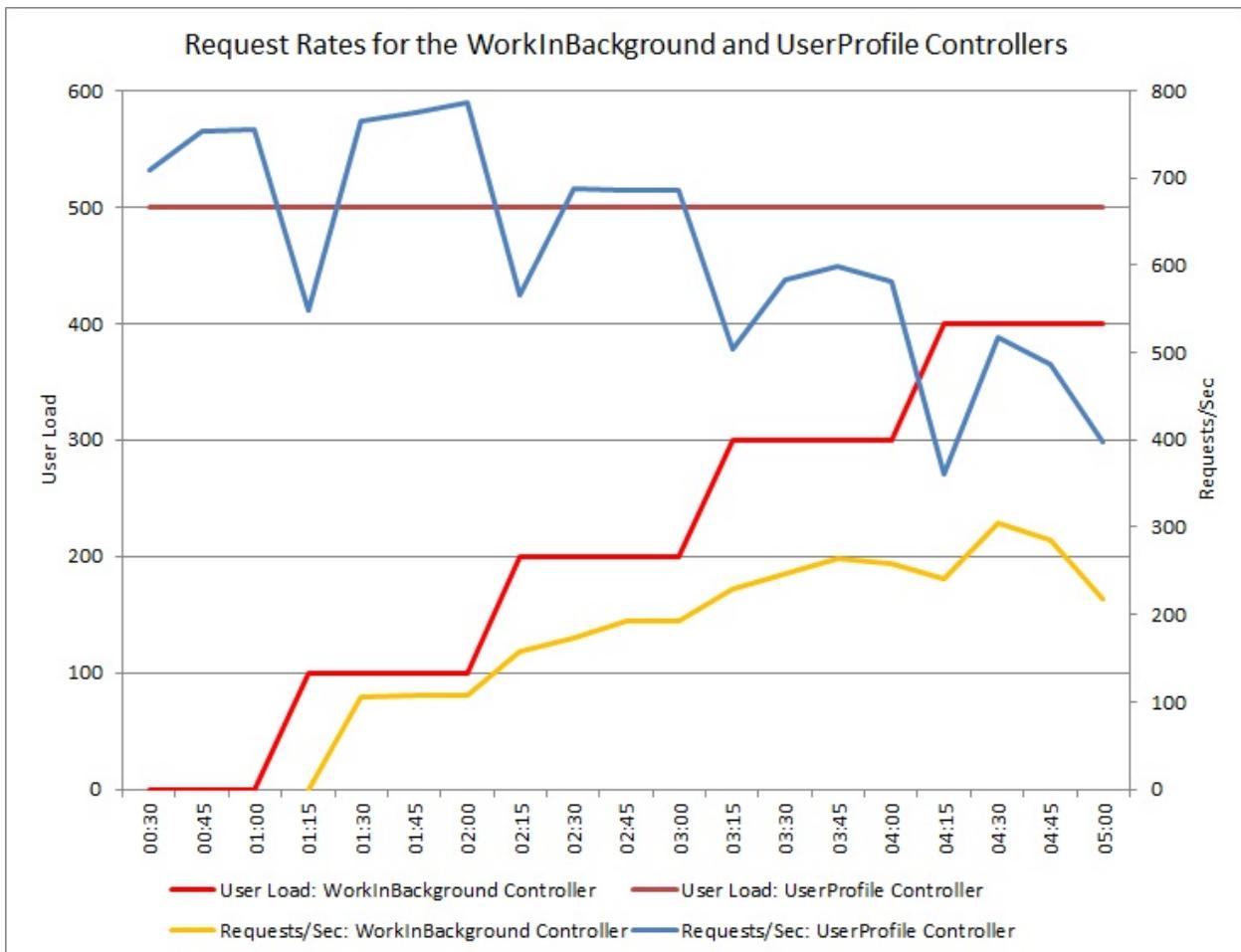


Note that the `WorkInBackground` controller also handled a much larger volume of requests. However, you can't make a direct comparison in this case, because the work being performed in this controller is very different from the original code. The new version simply queues a request, rather than performing a time consuming calculation. The main point is that this method no longer drags down the entire system under load.

CPU and network utilization also show the improved performance. The CPU utilization never reached 100%, and the volume of handled network requests was far greater than earlier, and did not tail off until the workload dropped.



The following graph shows the results of a load test. The overall volume of requests serviced is greatly improved compared to the earlier tests.



Related guidance

- Autoscaling best practices
- Background jobs best practices
- Queue-Based Load Leveling pattern
- Web Queue Worker architecture style

Chatty I/O antipattern

Article • 12/16/2022

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 ↗](#).

C#

```
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.

C#

```

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.

C#

```
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.

C#

```
private async Task SaveCustomerToFileAsync(Customer customer)  
{  
    using (Stream fileStream = new FileStream(CustomersFileName,  
        FileMode.Append))  
    {  
        BinaryFormatter formatter = new BinaryFormatter();  
        byte [] data = null;  
        using (MemoryStream memStream = new MemoryStream())  
        {  
            formatter.Serialize(memStream, customer);  
            data = memStream.ToArray();  
        }  
        fileStream.Write(data, 0, data.Length);  
    }  
}
```

```
        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.

C#

```
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.

C#

```
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.

C#

```
// Save a list of customer objects to a file
private async Task SaveCustomerListToFileAsync(List<Customer> customers)
{
    using (FileStream fileStream = new FileStream(CustomersFileName,
FileMode.Append))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        foreach (var customer in customers)
        {
            byte[] data = null;
            using (MemoryStream memStream = new MemoryStream())
            {
                formatter.Serialize(memStream, customer);
                data = memStream.ToArray();
            }
            await fileStream.WriteAsync(data, 0, data.Length);
        }
    }
}

// In-memory buffer for customers.
List<Customer> customers = new List<Customers>();

// Create a new customer and add it to the buffer
var customer = new Customer(...);
customers.Add(customer);

// Add more customers to the list as they are created
...

// Save the contents of the list, writing all customers in a single
operation
await SaveCustomerListToFileAsync(customers);
```

Considerations

- The first two examples make *fewer* I/O calls, but each one retrieves *more* information. You must consider the tradeoff between these two factors. The right answer will depend on the actual usage patterns. For example, in the web API example, it might turn out that clients often need just the user name. In that case, it might make sense to expose it as a separate API call. For more information, see the [Extraneous Fetching](#) antipattern.
- When reading data, do not make your I/O requests too large. An application should only retrieve the information that it is likely to use.
- Sometimes it helps to partition the information for an object into two chunks, *frequently accessed data* that accounts for most requests, and *less frequently accessed data* that is used rarely. Often the most frequently accessed data is a relatively small portion of the total data for an object, so returning just that portion can save significant I/O overhead.
- When writing data, avoid locking resources for longer than necessary, to reduce the chances of contention during a lengthy operation. If a write operation spans multiple data stores, files, or services, then adopt an eventually consistent approach. See [Data Consistency guidance](#).
- If you buffer data in memory before writing it, the data is vulnerable if the process crashes. If the data rate typically has bursts or is relatively sparse, it may be safer to buffer the data in an external durable queue such as [Event Hubs](#)↗.
- Consider caching data that you retrieve from a service or a database. This can help to reduce the volume of I/O by avoiding repeated requests for the same data. For more information, see [Caching best practices](#).

How to detect the problem

Symptoms of chatty I/O include high latency and low throughput. End users are likely to report extended response times or failures caused by services timing out, due to increased contention for I/O resources.

You can perform the following steps to help identify the causes of any problems:

1. Perform process monitoring of the production system to identify operations with poor response times.
2. Perform load testing of each operation identified in the previous step.
3. During the load tests, gather telemetry data about the data access requests made by each operation.
4. Gather detailed statistics for each request sent to a data store.

5. Profile the application in the test environment to establish where possible I/O bottlenecks might be occurring.

Look for any of these symptoms:

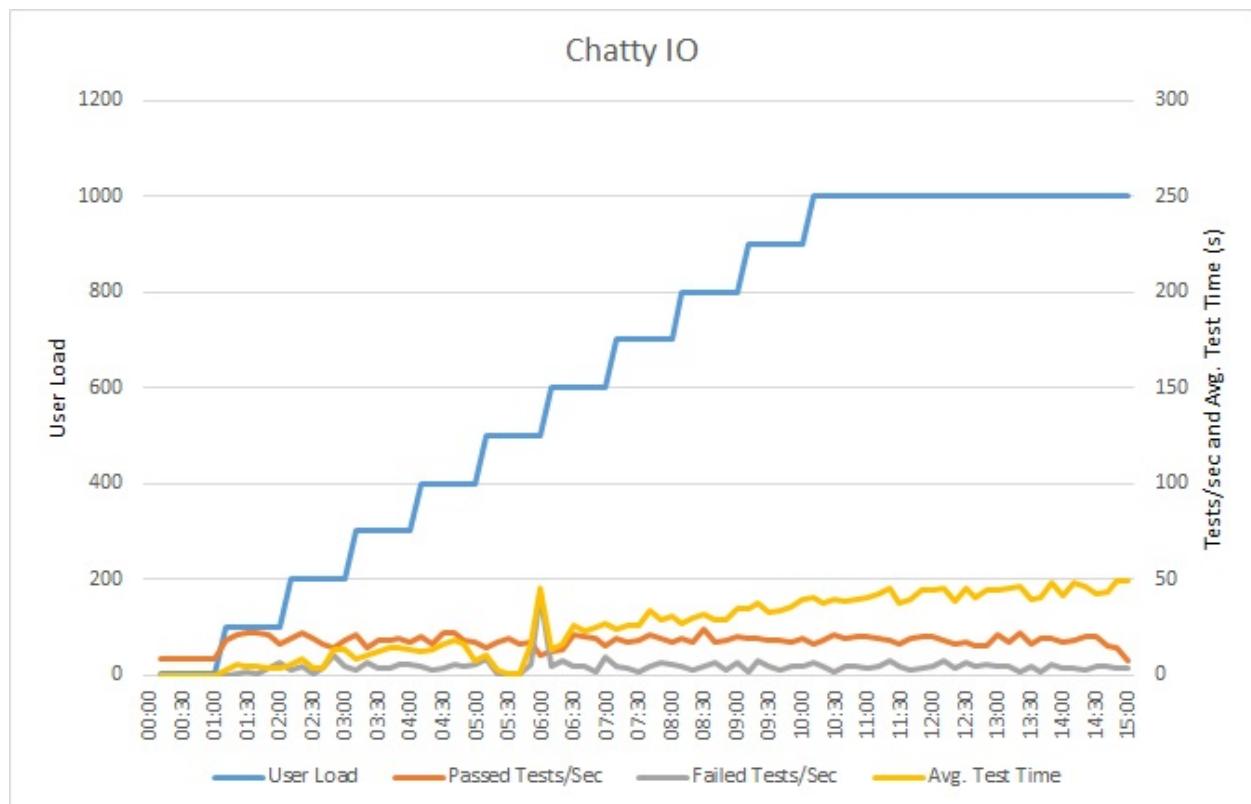
- A large number of small I/O requests made to the same file.
- A large number of small network requests made by an application instance to the same service.
- A large number of small requests made by an application instance to the same data store.
- Applications and services becoming I/O bound.

Example diagnosis

The following sections apply these steps to the example shown earlier that queries a database.

Load test the application

This graph shows the results of load testing. Median response time is measured in tens of seconds per request. The graph shows very high latency. With a load of 1000 users, a user might have to wait for nearly a minute to see the results of a query.



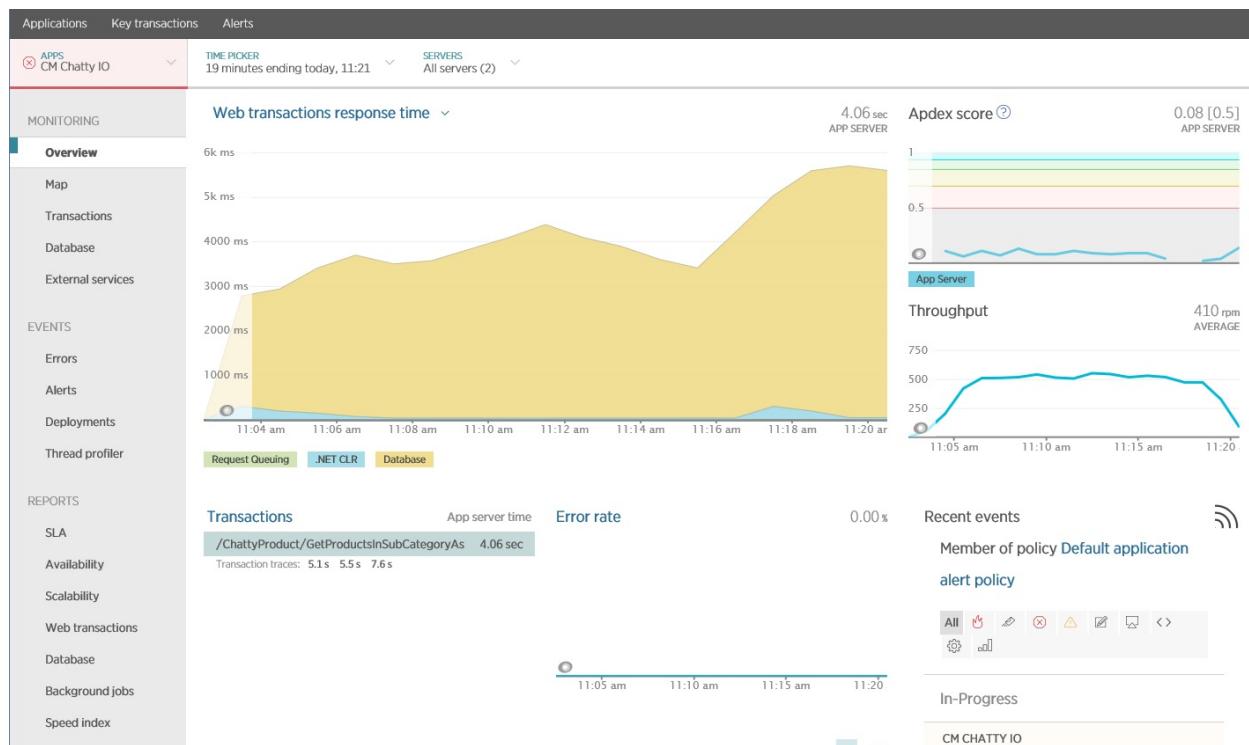
① Note

The application was deployed as an Azure App Service web app, using Azure SQL Database. The load test used a simulated step workload of up to 1000 concurrent users. The database was configured with a connection pool supporting up to 1000 concurrent connections, to reduce the chance that contention for connections would affect the results.

Monitor the application

You can use an application performance monitoring (APM) package to capture and analyze the key metrics that might identify chatty I/O. Which metrics are important will depend on the I/O workload. For this example, the interesting I/O requests were the database queries.

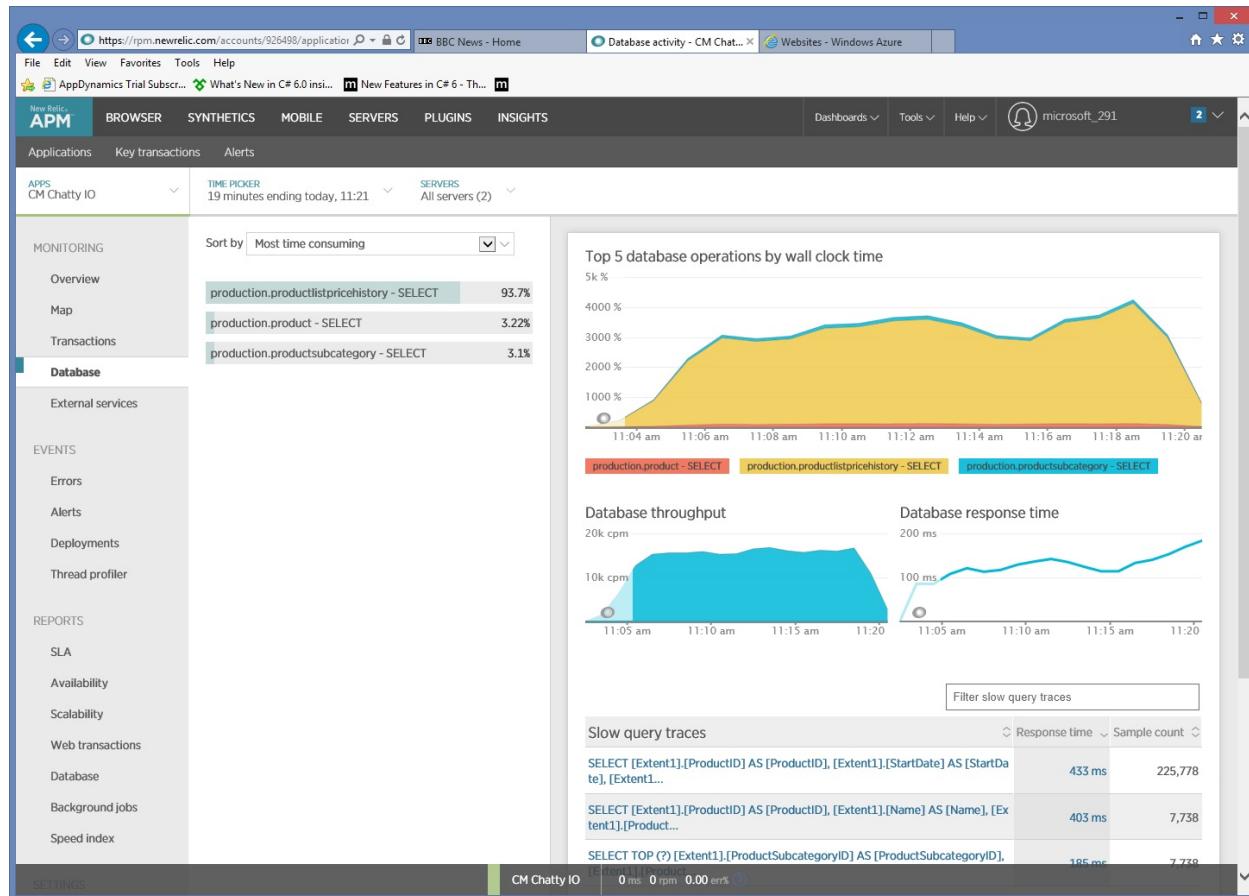
The following image shows results generated using [New Relic APM](#). The average database response time peaked at approximately 5.6 seconds per request during the maximum workload. The system was able to support an average of 410 requests per minute throughout the test.



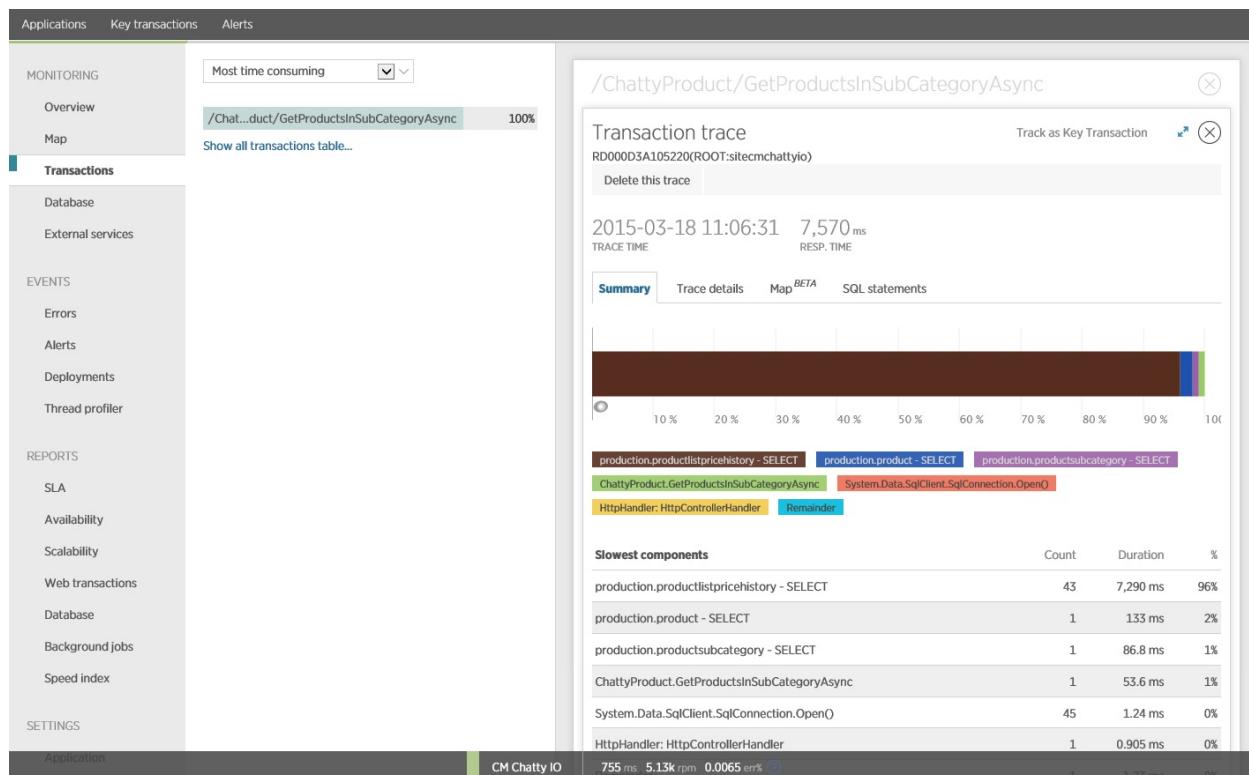
Gather detailed data access information

Digging deeper into the monitoring data shows the application executes three different SQL SELECT statements. These correspond to the requests generated by Entity Framework to fetch data from the `ProductListPriceHistory`, `Product`, and `ProductSubcategory` tables. Furthermore, the query that retrieves data from the

`ProductListPriceHistory` table is by far the most frequently executed SELECT statement, by an order of magnitude.



It turns out that the `GetProductsInSubCategoryAsync` method, shown earlier, performs 45 SELECT queries. Each query causes the application to open a new SQL connection.



(!) Note

This image shows trace information for the slowest instance of the `GetProductsInSubCategoryAsync` operation in the load test. In a production environment, it's useful to examine traces of the slowest instances, to see if there is a pattern that suggests a problem. If you just look at the average values, you might overlook problems that will get dramatically worse under load.

The next image shows the actual SQL statements that were issued. The query that fetches price information is run for each individual product in the product subcategory. Using a join would considerably reduce the number of database calls.

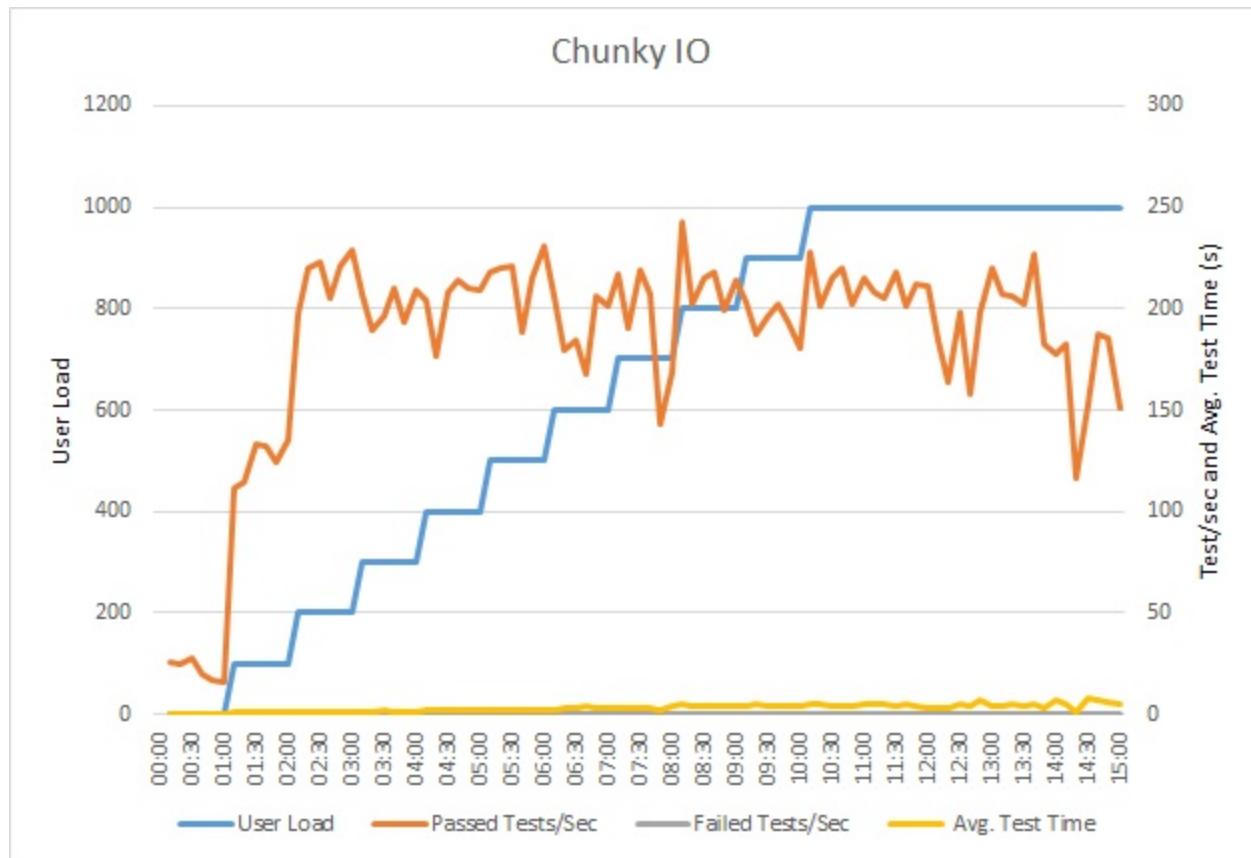
The screenshot shows the Application Insights interface. On the left, the navigation menu includes Applications, Key transactions, and Alerts. Under MONITORING, the 'Transactions' section is selected, showing a dropdown for 'Most time consuming' which is set to '/Chat...duct/GetProductsInSubCategoryAsync'. A progress bar indicates 100% completion. Below this, there are links for Overview, Map, and Show all transactions table... On the right, a detailed transaction trace for the URL /ChattyProduct/GetProductsInSubCategoryAsync is displayed. The trace header shows the date and time (2015-03-18 11:06:31), duration (7,570 ms), and trace ID (RD000D3A105220(ROOT:sitemchattyio)). The trace details tab is selected. The SQL statements section lists three queries:

Total duration	Call count	SQL
7,290 ms	43	SELECT [Extent1].[ProductID] AS [ProductID], [Extent1].[StartDate] AS [StartDate], [Extent1].[EndDate] AS [EndDate], [Extent1].[ListPrice] AS [ListPrice] 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.

Applications Key transactions Alerts

APPS CM Chatty IO TIME PICKER 19 minutes ending today, 11:54 SERVERS All servers (2)

MONITORING Overview Map Transactions Database External services EVENTS Errors Alerts Deployments Thread profiler REPORTS SLA Availability Scalability Web transactions Database Background jobs Speed index

Most time consuming /Chun...uct/GetProductCategoryDetailsAsync 100% Show all transactions table...

/ChunkyProduct/GetProductCategoryDetailsAsync Track as key transaction

App performance App server breakdown 0.74 APDEX 623 ms AVERAGE

1000 ms
750 ms
500 ms
250 ms
1:36 am 11:38 am 11:40 am 11:42 am 11:44 am 11:46 am 11:48 am 11:50 am 11:52 am

INNER_SELECT - SELECT System.Data.SqlClient.SqlConnection.Open() ExecuteRequestHandler ChunkyProduct.GetProductCategoryDetailsAsync() HttpControllerHandler Other

Throughput 3,970 rpm AVERAGE

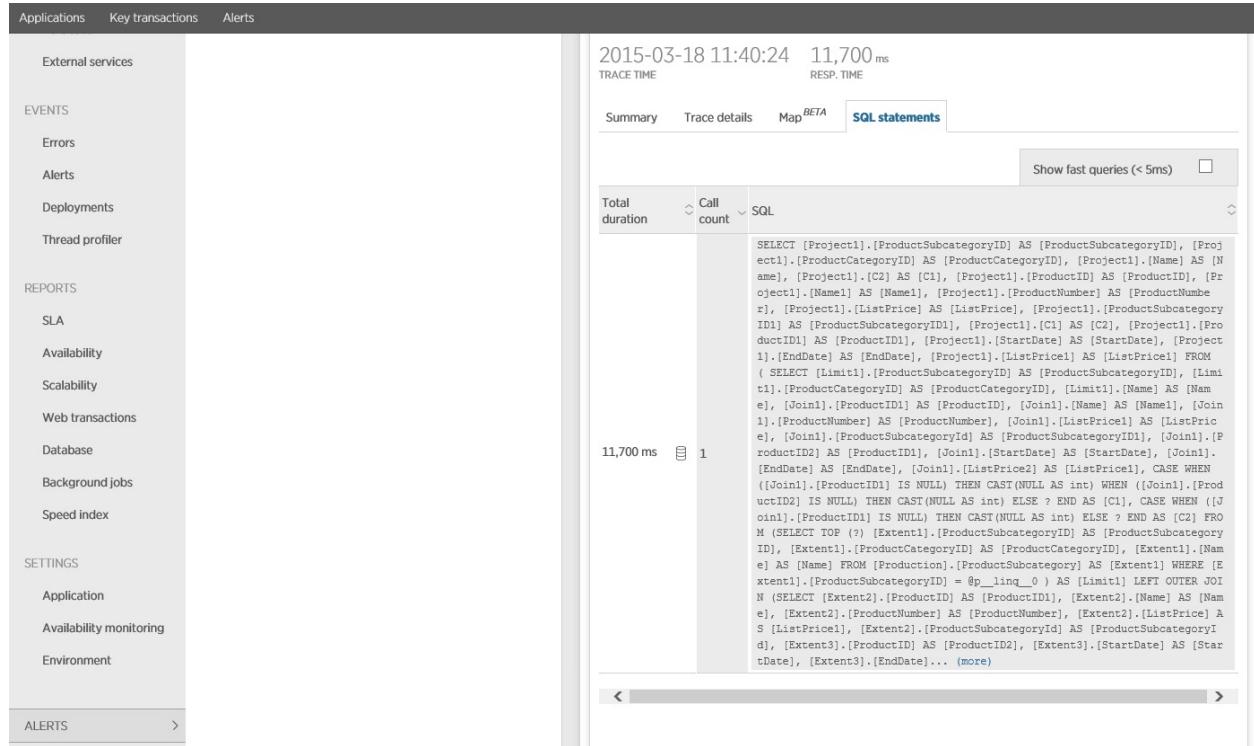
7.5k rpm
5k rpm
2.5k rpm
1:36 am 11:38 am 11:40 am 11:42 am 11:44 am 11:46 am 11:48 am 11:50 am 11:52 am

Breakdown table

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Database	INNER_SELECT - SELECT	91.2	1.0	568

CM Chatty IO CPU 0 ms Memory 0 ms Network 0.00 err% System 0 ms SQLClient.SqlConnection.Open() 2.9 1.0 17.7

Tracing the SQL statement shows that all the data is fetched in a single SELECT statement. Although this query is considerably more complex, it is performed only once per operation. And while complex joins can become expensive, relational database systems are optimized for this type of query.



Related resources

- [API Design best practices](#)
- [Caching best practices](#)
- [Data Consistency Primer](#)
- [Extraneous Fetching antipattern](#)
- [No Caching antipattern](#)

Extraneous Fetching antipattern

Article • 12/16/2022

Anti-patterns are common design flaws that can break your software or applications under stress situations and should not be overlooked. In an *extraneous fetching antipattern*, more than needed data is retrieved for a business operation, often resulting in unnecessary I/O overhead and reduced responsiveness.

Examples of extraneous fetching antipattern

This antipattern can occur if the application tries to minimize I/O requests by retrieving all of the data that it *might* need. This is often a result of overcompensating for the [Chatty I/O](#) antipattern. For example, an application might fetch the details for every product in a database. But the user may need just a subset of the details (some may not be relevant to customers), and probably doesn't need to see *all* of the products at once. Even if the user is browsing the entire catalog, it would make sense to paginate the results—showing 20 at a time, for example.

Another source of this problem is following poor programming or design practices. For example, the following code uses Entity Framework to fetch the complete details for every product. Then it filters the results to return only a subset of the fields, discarding the rest. You can find the complete sample [here ↗](#).

C#

```
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](#).

```
C#
```

```
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.

```
C#
```

```
var query = from p in context.Products.AsEnumerable()
            where p.SellStartDate < DateTime.Now.AddDays(-7) // AddDays
cannot be mapped by LINQ to Entities
            select ...;

List<Product> products = query.ToList();
```

The application is trying to find products with a `SellStartDate` more than a week old. In most cases, LINQ to Entities would translate a `where` clause to a SQL statement that is executed by the database. In this case, however, LINQ to Entities cannot map the `AddDays` method to SQL. Instead, every row from the `Product` table is returned, and the results are filtered in memory.

The call to `AsEnumerable` is a hint that there is a problem. This method converts the results to an `IEnumerable` interface. Although `IEnumerable` supports filtering, the filtering is done on the *client* side, not the database. By default, LINQ to Entities uses `IQueryable`, which passes the responsibility for filtering to the data source.

How to fix extraneous fetching antipattern

Avoid fetching large volumes of data that may quickly become outdated or might be discarded, and only fetch the data needed for the operation being performed.

Instead of getting every column from a table and then filtering them, select the columns that you need from the database.

C#

```
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.

C#

```
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.

C#

```
DateTime dateSince = DateTime.Now.AddDays(-7); // AddDays has been factored
out.
var query = from p in context.Products
            where p.SellStartDate < dateSince // This criterion can be
passed to the database by LINQ to Entities
```

```
select ...;  
  
List<Product> products = query.ToList();
```

Considerations

- In some cases, you can improve performance by partitioning data horizontally. If different operations access different attributes of the data, horizontal partitioning may reduce contention. Often, most operations are run against a small subset of the data, so spreading this load may improve performance. See [Data partitioning](#).
- For operations that have to support unbounded queries, implement pagination and only fetch a limited number of entities at a time. For example, if a customer is browsing a product catalog, you can show one page of results at a time.
- When possible, take advantage of features built into the data store. For example, SQL databases typically provide aggregate functions.
- If you're using a data store that doesn't support a particular function, such as aggregation, you could store the calculated result elsewhere, updating the value as records are added or updated, so the application doesn't have to recalculate the value each time it's needed.
- If you see that requests are retrieving a large number of fields, examine the source code to determine whether all of these fields are necessary. Sometimes these requests are the result of poorly designed `SELECT *` query.
- Similarly, requests that retrieve a large number of entities may be sign that the application is not filtering data correctly. Verify that all of these entities are needed. Use database-side filtering if possible, for example, by using `WHERE` clauses in SQL.
- Offloading processing to the database is not always the best option. Only use this strategy when the database is designed or optimized to do so. Most database systems are highly optimized for certain functions, but are not designed to act as general-purpose application engines. For more information, see the [Busy Database antipattern](#).

How to detect extraneous fetching antipattern

Symptoms of extraneous fetching include high latency and low throughput. If the data is retrieved from a data store, increased contention is also probable. End users are likely to report extended response times or failures caused by services timing out. These failures

could return HTTP 500 (Internal Server) errors or HTTP 503 (Service Unavailable) errors. Examine the event logs for the web server, which likely contain more detailed information about the causes and circumstances of the errors.

The symptoms of this antipattern and some of the telemetry obtained might be very similar to those of the [Monolithic Persistence antipattern](#).

You can perform the following steps to help identify the cause:

1. Identify slow workloads or transactions by performing load-testing, process monitoring, or other methods of capturing instrumentation data.
2. Observe any behavioral patterns exhibited by the system. Are there particular limits in terms of transactions per second or volume of users?
3. Correlate the instances of slow workloads with behavioral patterns.
4. Identify the data stores being used. For each data source, run lower-level telemetry to observe the behavior of operations.
5. Identify any slow-running queries that reference these data sources.
6. Perform a resource-specific analysis of the slow-running queries and ascertain how the data is used and consumed.

Look for any of these symptoms:

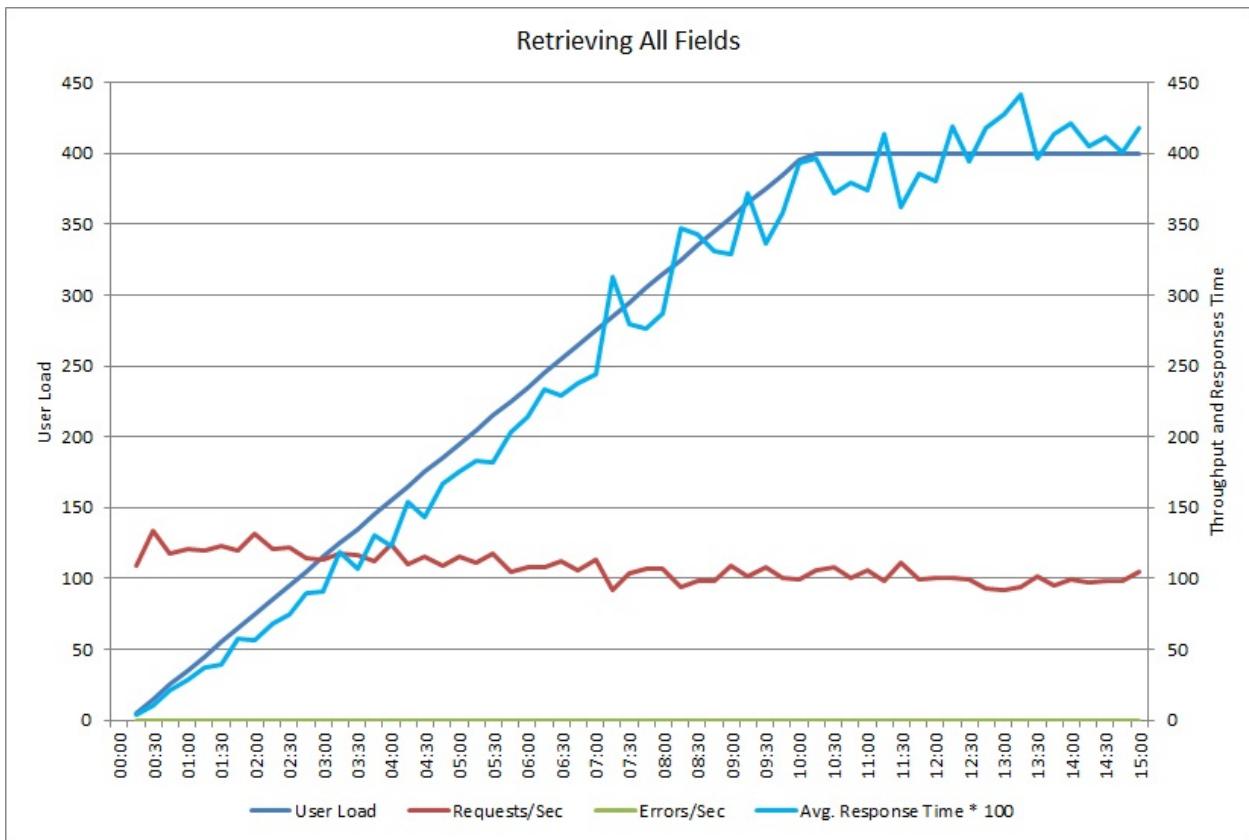
- Frequent, large I/O requests made to the same resource or data store.
- Contention in a shared resource or data store.
- An operation that frequently receives large volumes of data over the network.
- Applications and services spending significant time waiting for I/O to complete.

Example diagnosis

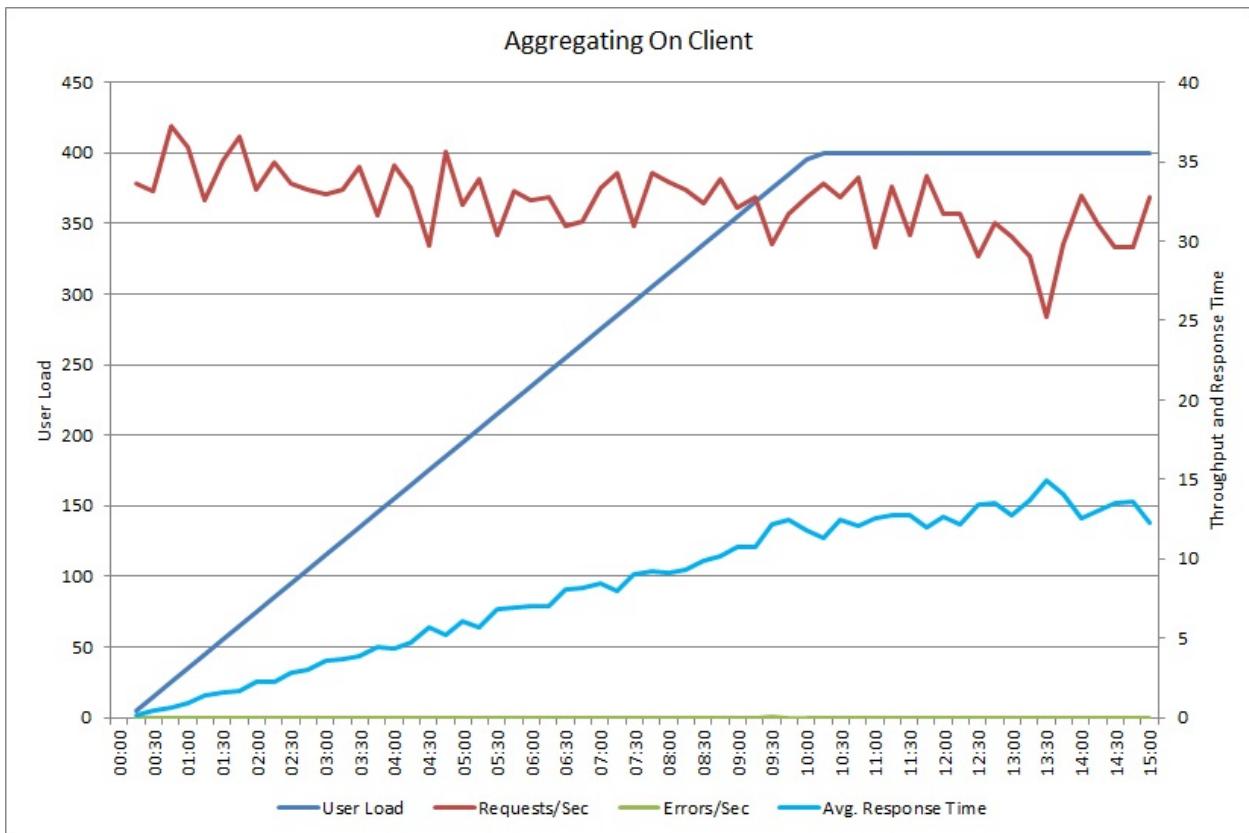
The following sections apply these steps to the previous examples.

Identify slow workloads

This graph shows performance results from a load test that simulated up to 400 concurrent users running the `GetAllFieldsAsync` method shown earlier. Throughput diminishes slowly as the load increases. Average response time goes up as the workload increases.



A load test for the `AggregateOnClientAsync` operation shows a similar pattern. The volume of requests is reasonably stable. The average response time increases with the workload, although more slowly than the previous graph.



Correlate slow workloads with behavioral patterns

Any correlation between regular periods of high usage and slowing performance can indicate areas of concern. Closely examine the performance profile of functionality that is suspected to be slow running, to determine whether it matches the load testing performed earlier.

Load test the same functionality using step-based user loads, to find the point where performance drops significantly or fails completely. If that point falls within the bounds of your expected real-world usage, examine how the functionality is implemented.

A slow operation is not necessarily a problem, if it is not being performed when the system is under stress, is not time critical, and does not negatively affect the performance of other important operations. For example, generating monthly operational statistics might be a long-running operation, but it can probably be performed as a batch process and run as a low-priority job. On the other hand, customers querying the product catalog is a critical business operation. Focus on the telemetry generated by these critical operations to see how the performance varies during periods of high usage.

Identify data sources in slow workloads

If you suspect that a service is performing poorly because of the way it retrieves data, investigate how the application interacts with the repositories it uses. Monitor the live system to see which sources are accessed during periods of poor performance.

For each data source, instrument the system to capture the following:

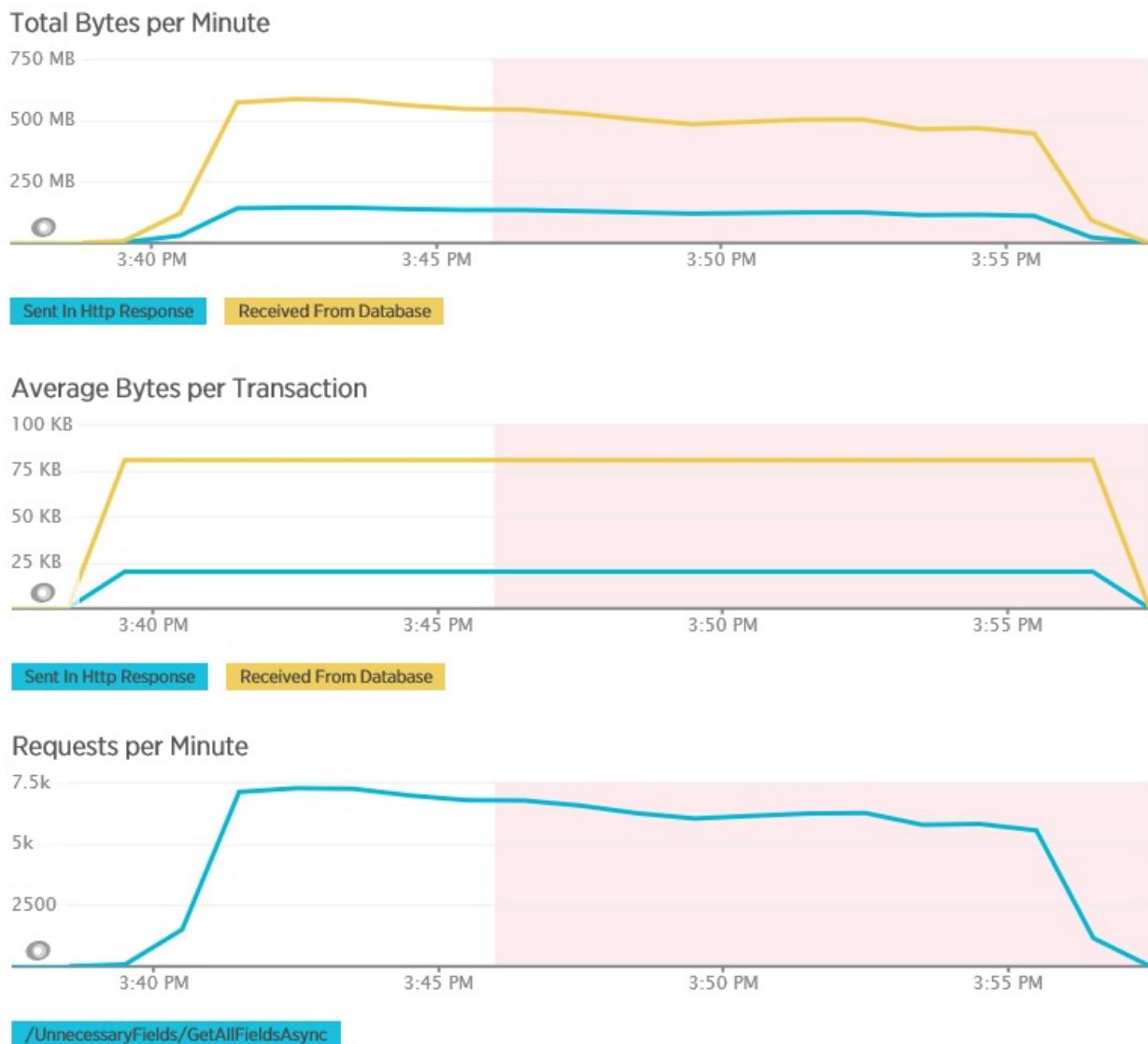
- The frequency that each data store is accessed.
- The volume of data entering and exiting the data store.
- The timing of these operations, especially the latency of requests.
- The nature and rate of any errors that occur while accessing each data store under typical load.

Compare this information against the volume of data being returned by the application to the client. Track the ratio of the volume of data returned by the data store against the volume of data returned to the client. If there is any large disparity, investigate to determine whether the application is fetching data that it doesn't need.

You may be able to capture this data by observing the live system and tracing the lifecycle of each user request, or you can model a series of synthetic workloads and run them against a test system.

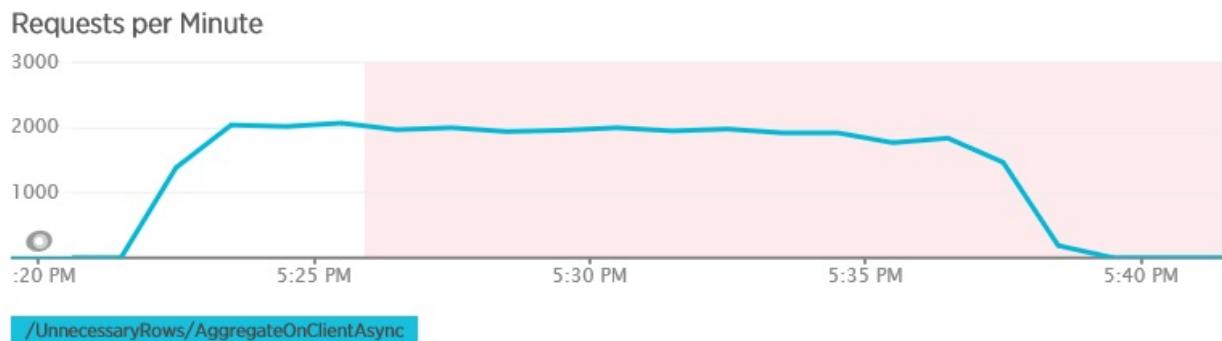
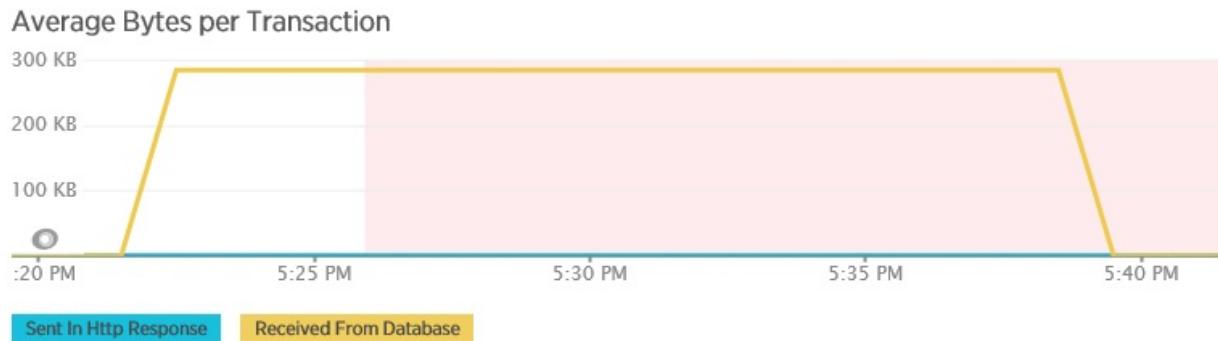
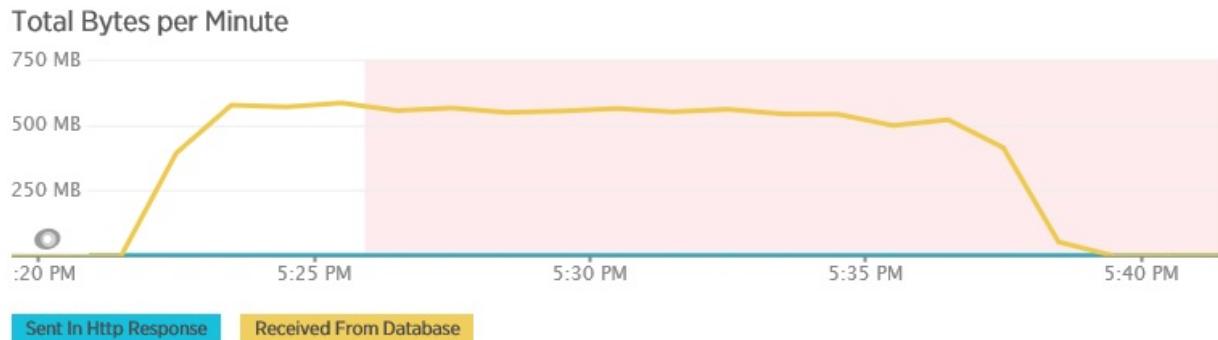
The following graphs show telemetry captured using [New Relic APM](#) during a load test of the `GetAllFieldsAsync` method. Note the difference between the volumes of data

received from the database and the corresponding HTTP responses.



For each request, the database returned 80,503 bytes, but the response to the client only contained 19,855 bytes, about 25% of the size of the database response. The size of the data returned to the client can vary depending on the format. For this load test, the client requested JSON data. Separate testing using XML (not shown) had a response size of 35,655 bytes, or 44% of the size of the database response.

The load test for the `AggregateOnClientAsync` method shows more extreme results. In this case, each test performed a query that retrieved over 280 Kb of data from the database, but the JSON response was a mere 14 bytes. The wide disparity is because the method calculates an aggregated result from a large volume of data.



Identify and analyze slow queries

Look for database queries that consume the most resources and take the most time to execute. You can add instrumentation to find the start and completion times for many database operations. Many data stores also provide in-depth information on how queries are performed and optimized. For example, the Query Performance pane in the Azure SQL Database management portal lets you select a query and view detailed runtime performance information. Here is the query generated by the `GetAllFieldsAsync` operation:

```

SELECT
    [Extent1].[ProductID] AS [ProductID],
    [Extent1].[Name] AS [Name],
    [Extent1].[ProductNumber] AS [ProductNumber],
    [Extent1].[MakeFlag] AS [MakeFlag],
    [Extent1].[FinishedGoodsFlag] AS [FinishedGoodsFlag],
    [Extent1].[Color] AS [Color],
    [Extent1].[SafetyStockLevel] AS [SafetyStockLevel],
    [Extent1].[ReorderPoint] AS [ReorderPoint],
    [Extent1].[StandardCost] AS [StandardCost],
    [Extent1].[ListPrice] AS [ListPrice],
    [Extent1].[Size] AS [Size],
    [Extent1].[SizeUnitMeasureCode] AS [SizeUnitMeasureCode],
    [Extent1].[WeightUnitMeasureCode] AS [WeightUnitMeasureCode],
    [Extent1].[Weight] AS [Weight],

```

Query Plan Details

[Query Plan](#)

Resource Use

Resource	Total / sec	Total	Last Run	Minimum	Maximum
Duration (ms)	117	63450	40	0	3867
CPU (ms)	1	752	0	0	3
Logical Reads	25	13872	16	16	16
Physical Reads	0	0	0	0	0
Logical Writes	0	0	0	0	0

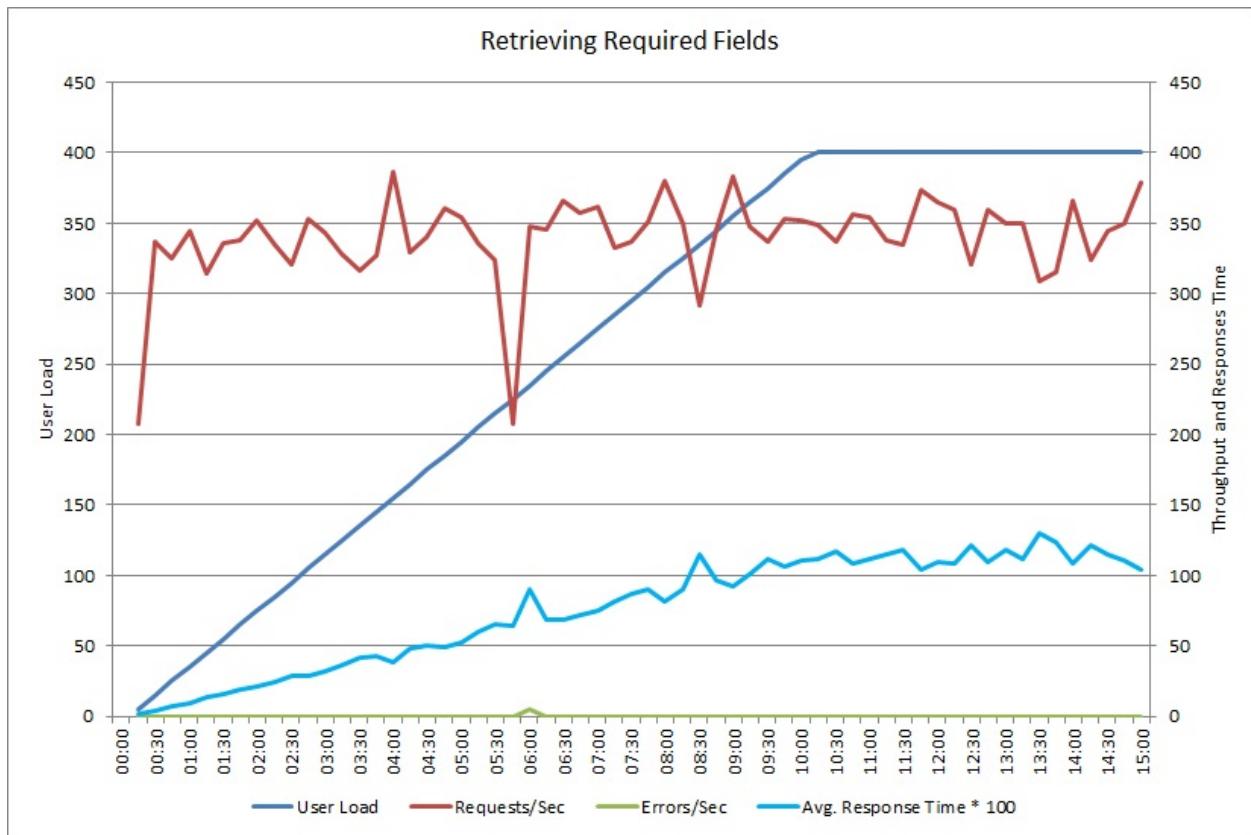
Plan Information

Advanced Information

Run Count	867	Plan Handle	0x0600E0004D00CD02A0EDD87D55000000100000000000000000000000000000000
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	0xDA11AA10A268A7B

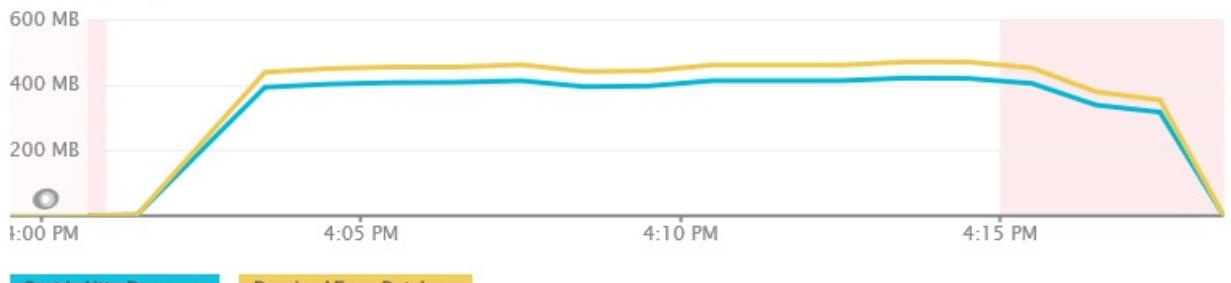
Implement the solution and verify the result

After changing the `GetRequiredFieldsAsync` method to use a SELECT statement on the database side, load testing showed the following results.

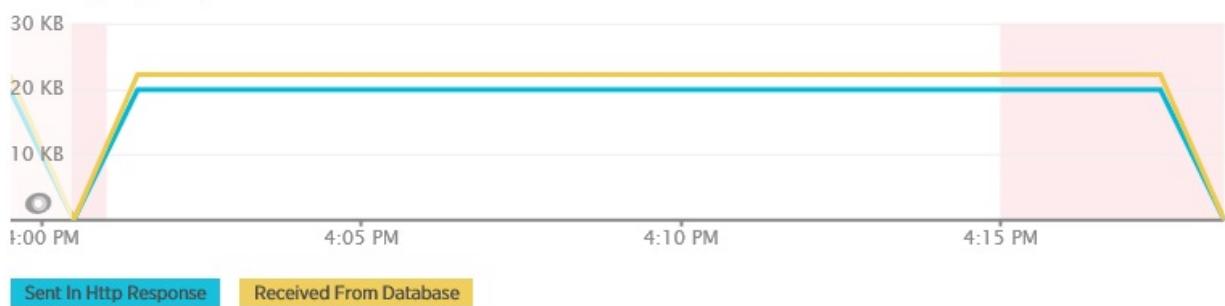


This load test used the same deployment and the same simulated workload of 400 concurrent users as before. The graph shows much lower latency. Response time rises with load to approximately 1.3 seconds, compared to 4 seconds in the previous case. The throughput is also higher at 350 requests per second compared to 100 earlier. The volume of data retrieved from the database now closely matches the size of the HTTP response messages.

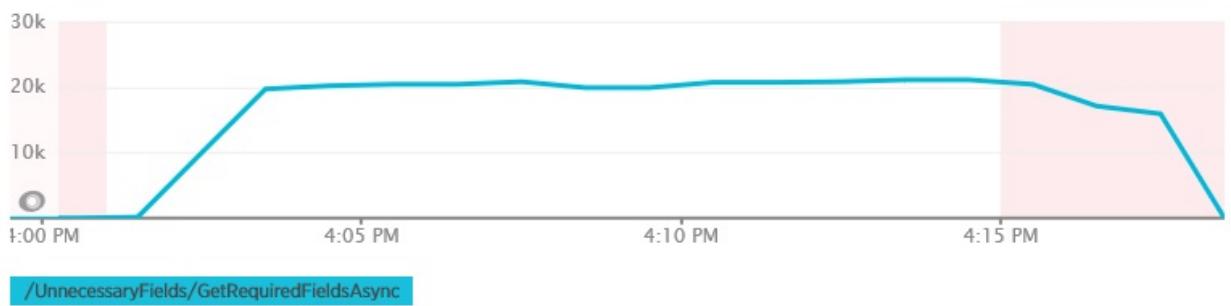
Total Bytes per Minute



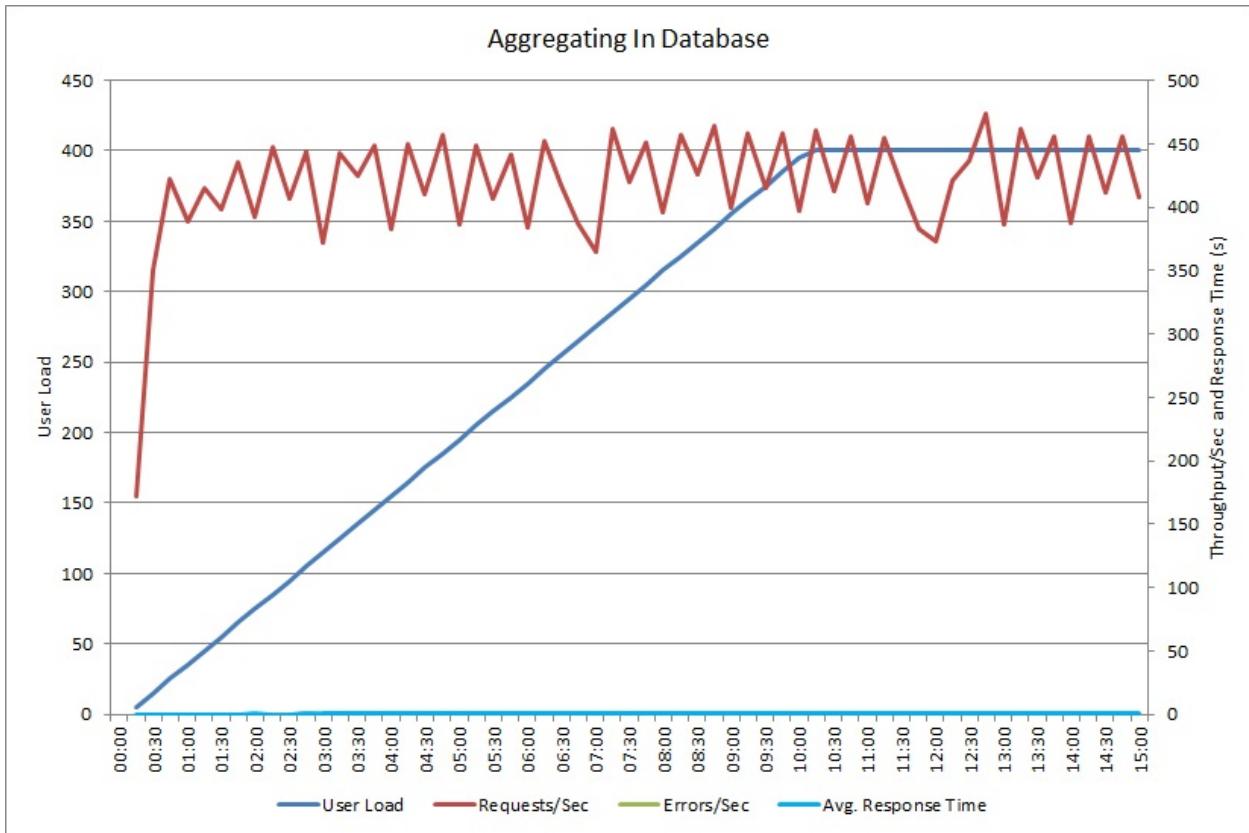
Average Bytes per Transaction



Requests per Minute



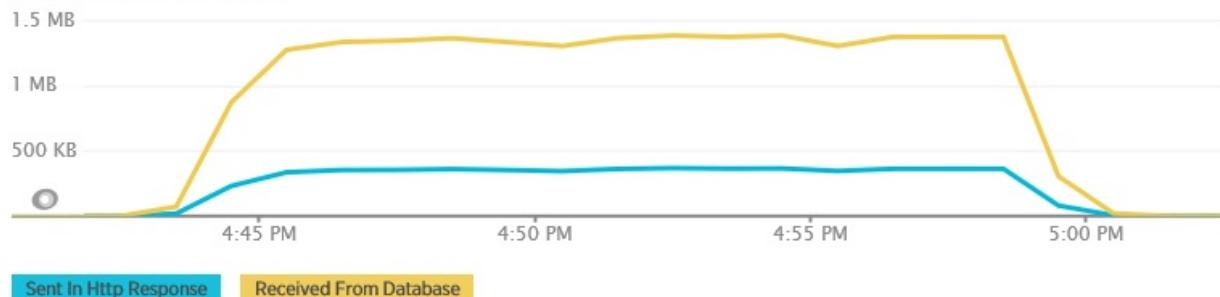
Load testing using the `AggregateOnDatabaseAsync` method generates the following results:



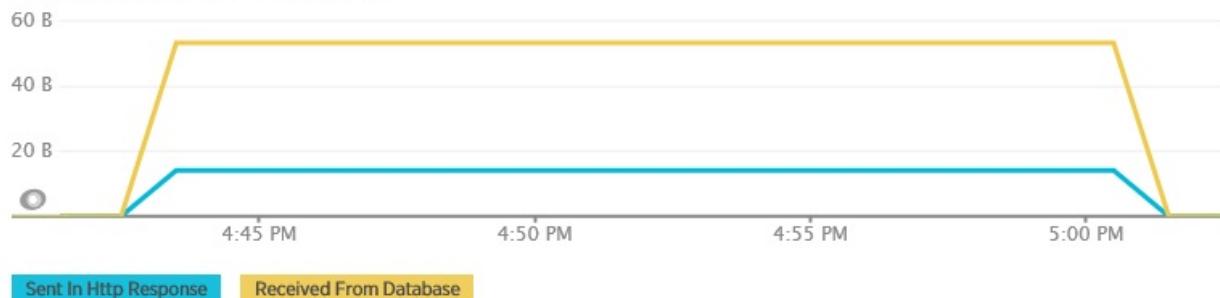
The average response time is now minimal. This is an order of magnitude improvement in performance, caused primarily by the large reduction in I/O from the database.

Here is the corresponding telemetry for the `AggregateOnDatabaseAsync` method. The amount of data retrieved from the database was vastly reduced, from over 280 Kb per transaction to 53 bytes. As a result, the maximum sustained number of requests per minute was raised from around 2,000 to over 25,000.

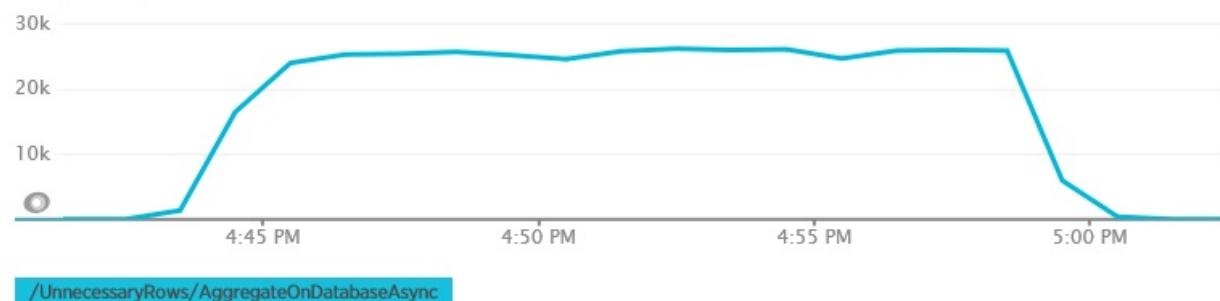
Total Bytes per Minute



Average Bytes per Transaction



Requests per Minute



Related resources

- [Busy Database antipattern](#)
- [Chatty I/O antipattern](#)
- [Data partitioning best practices](#)

Improper Instantiation antipattern

Article • 12/16/2022

Sometimes new instances of a class are continually created, when it is meant to be created once and then shared. This behavior can hurt performance, and is called an *improper instantiation antipattern*. An antipattern is a common response to a recurring problem that is usually ineffective and may even be counter-productive.

Problem description

Many libraries provide abstractions of external resources. Internally, these classes typically manage their own connections to the resource, acting as brokers that clients can use to access the resource. Here are some examples of broker classes that are relevant to Azure applications:

- `System.Net.Http.HttpClient`. Communicates with a web service using HTTP.
- `Microsoft.ServiceBus.Messaging.QueueClient`. Posts and receives messages to a Service Bus queue.
- `Microsoft.Azure.Documents.Client.DocumentClient`. Connects to an Azure Cosmos DB instance.
- `StackExchange.Redis.ConnectionMultiplexer`. Connects to Redis, including Azure Cache for Redis.

These classes are intended to be instantiated once and reused throughout the lifetime of an application. However, it's a common misunderstanding that these classes should be acquired only as necessary and released quickly. (The ones listed here happen to be .NET libraries, but the pattern is not unique to .NET.) The following ASP.NET example creates an instance of `HttpClient` to communicate with a remote service. You can find the complete sample [here ↗](#).

C#

```
public class NewHttpClientInstancePerRequestController : ApiController
{
    // This method creates a new instance of HttpClient and disposes it for
    // every call to GetProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        using (var httpClient = new HttpClient())
        {
            var hostName = HttpContext.Current.Request.Url.Host;
            var result = await
                httpClient.GetStringAsync(string.Format("http://{0}:8080/api/...",
```

```
        hostName));
            return new Product { Name = result };
        }
    }
}
```

In a web application, this technique is not scalable. A new `HttpClient` object is created for each user request. Under heavy load, the web server may exhaust the number of available sockets, resulting in `SocketException` errors.

This problem is not restricted to the `HttpClient` class. Other classes that wrap resources or are expensive to create might cause similar issues. The following example creates an instance of the `ExpensiveToCreateService` class. Here the issue is not necessarily socket exhaustion, but simply how long it takes to create each instance. Continually creating and destroying instances of this class might adversely affect the scalability of the system.

C#

```
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
        Thread.SpinWait(Int32.MaxValue / 100);
    }
    ...
}
```

How to fix improper instantiation antipattern

If the class that wraps the external resource is shareable and thread-safe, create a shared singleton instance or a pool of reusable instances of the class.

The following example uses a static `HttpClient` instance, thus sharing the connection across all requests.

C#

```
public class SingleHttpClientInstanceController : ApiController
{
    private static readonly HttpClient httpClient;

    static SingleHttpClientInstanceController()
    {
        httpClient = new HttpClient();
    }

    // This method uses the shared instance of HttpClient for every call to
    GetProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        var hostName = HttpContext.Current.Request.Url.Host;
        var result = await
    httpClient.GetStringAsync(string.Format("http://{0}:8080/api/...", hostName));
        return new Product { Name = result };
    }
}
```

Considerations

- The key element of this antipattern is repeatedly creating and destroying instances of a *shareable* object. If a class is not shareable (not thread-safe), then this antipattern does not apply.
- The type of shared resource might dictate whether you should use a singleton or create a pool. The `HttpClient` class is designed to be shared rather than pooled. Other objects might support pooling, enabling the system to spread the workload across multiple instances.
- Objects that you share across multiple requests *must* be thread-safe. The `HttpClient` class is designed to be used in this manner, but other classes might not support concurrent requests, so check the available documentation.
- Be careful about setting properties on shared objects, as this can lead to race conditions. For example, setting `DefaultRequestHeaders` on the `HttpClient` class before each request can create a race condition. Set such properties once (for example, during startup), and create separate instances if you need to configure different settings.
- Some resource types are scarce and should not be held onto. Database connections are an example. Holding an open database connection that is not

required may prevent other concurrent users from gaining access to the database.

- In the .NET Framework, many objects that establish connections to external resources are created by using static factory methods of other classes that manage these connections. These objects are intended to be saved and reused, rather than disposed and re-created. For example, in Azure Service Bus, the `QueueClient` object is created through a `MessagingFactory` object. Internally, the `MessagingFactory` manages connections. For more information, see [Best Practices for performance improvements using Service Bus Messaging](#).

How to detect improper instantiation antipattern

Symptoms of this problem include a drop in throughput or an increased error rate, along with one or more of the following:

- An increase in exceptions that indicate exhaustion of resources such as sockets, database connections, file handles, and so on.
- Increased memory use and garbage collection.
- An increase in network, disk, or database activity.

You can perform the following steps to help identify this problem:

1. Performing process monitoring of the production system, to identify points when response times slow down or the system fails due to lack of resources.
2. Examine the telemetry data captured at these points to determine which operations might be creating and destroying resource-consuming objects.
3. Load test each suspected operation, in a controlled test environment rather than the production system.
4. Review the source code and examine the how broker objects are managed.

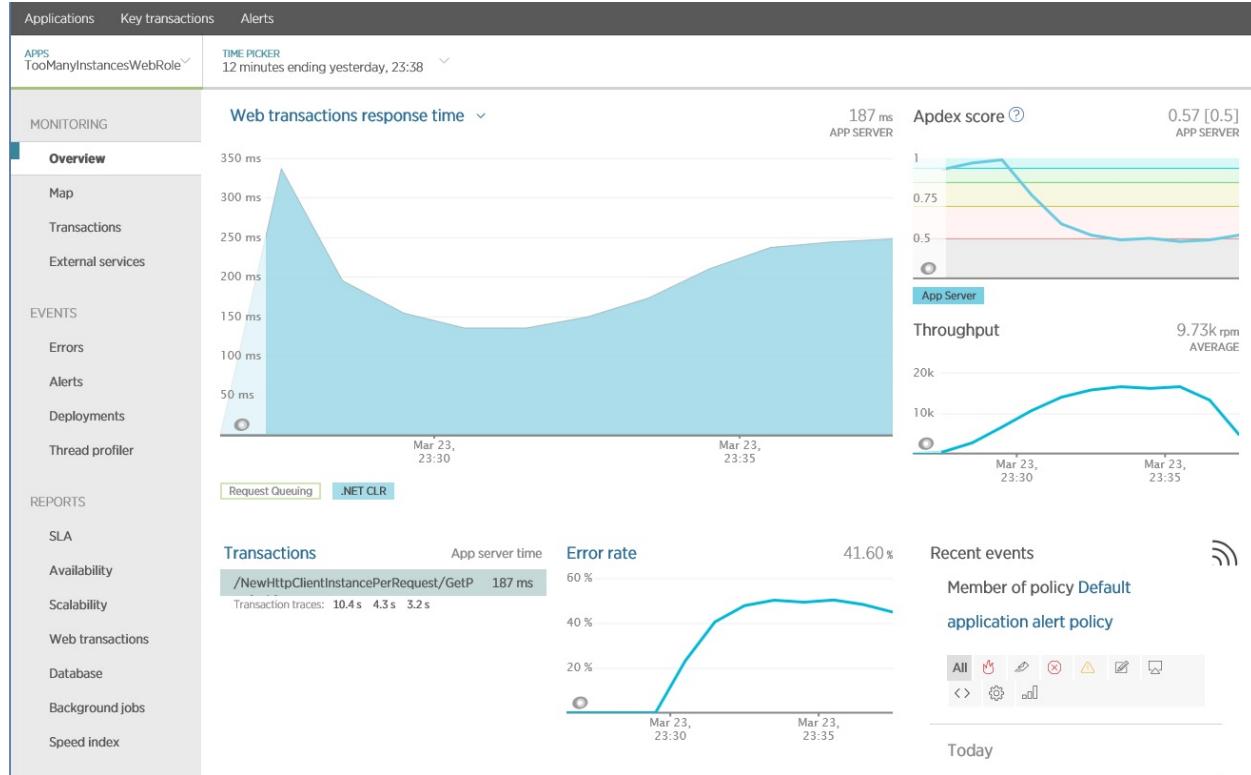
Look at stack traces for operations that are slow-running or that generate exceptions when the system is under load. This information can help to identify how these operations are using resources. Exceptions can help to determine whether errors are caused by shared resources being exhausted.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

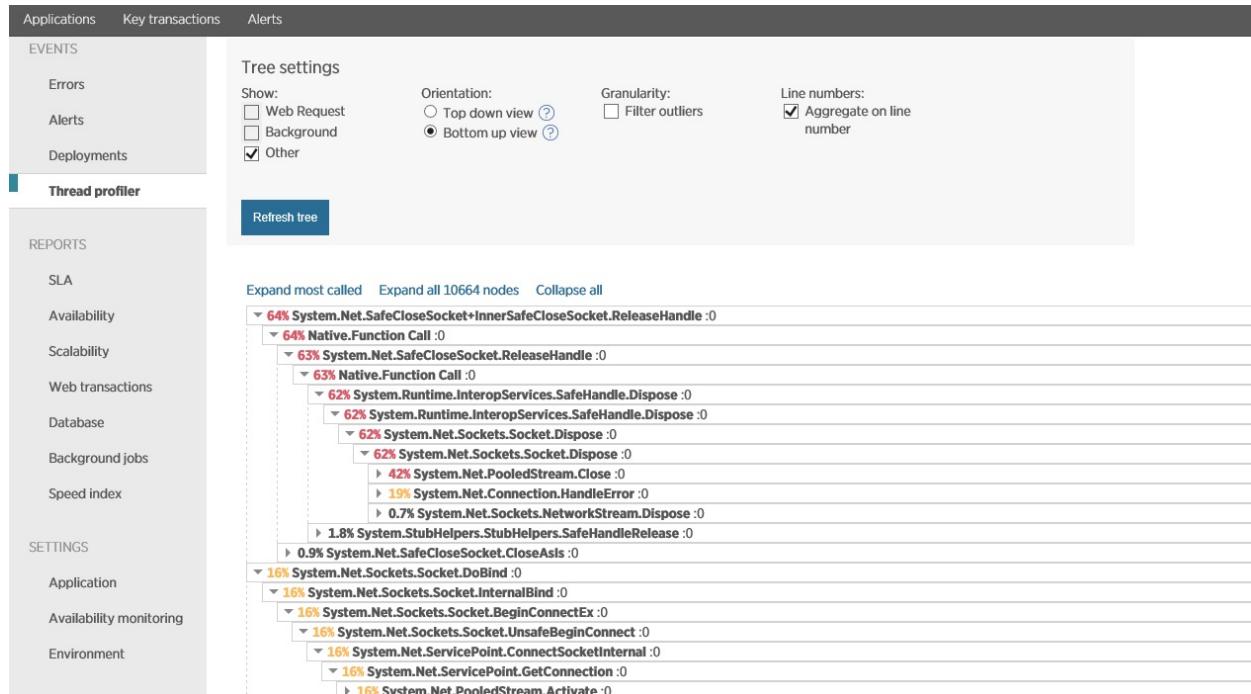
Identify points of slowdown or failure

The following image shows results generated using New Relic APM², showing operations that have a poor response time. In this case, the `GetProductAsync` method in the `NewHttpClientInstancePerRequest` controller is worth investigating further. Notice that the error rate also increases when these operations are running.



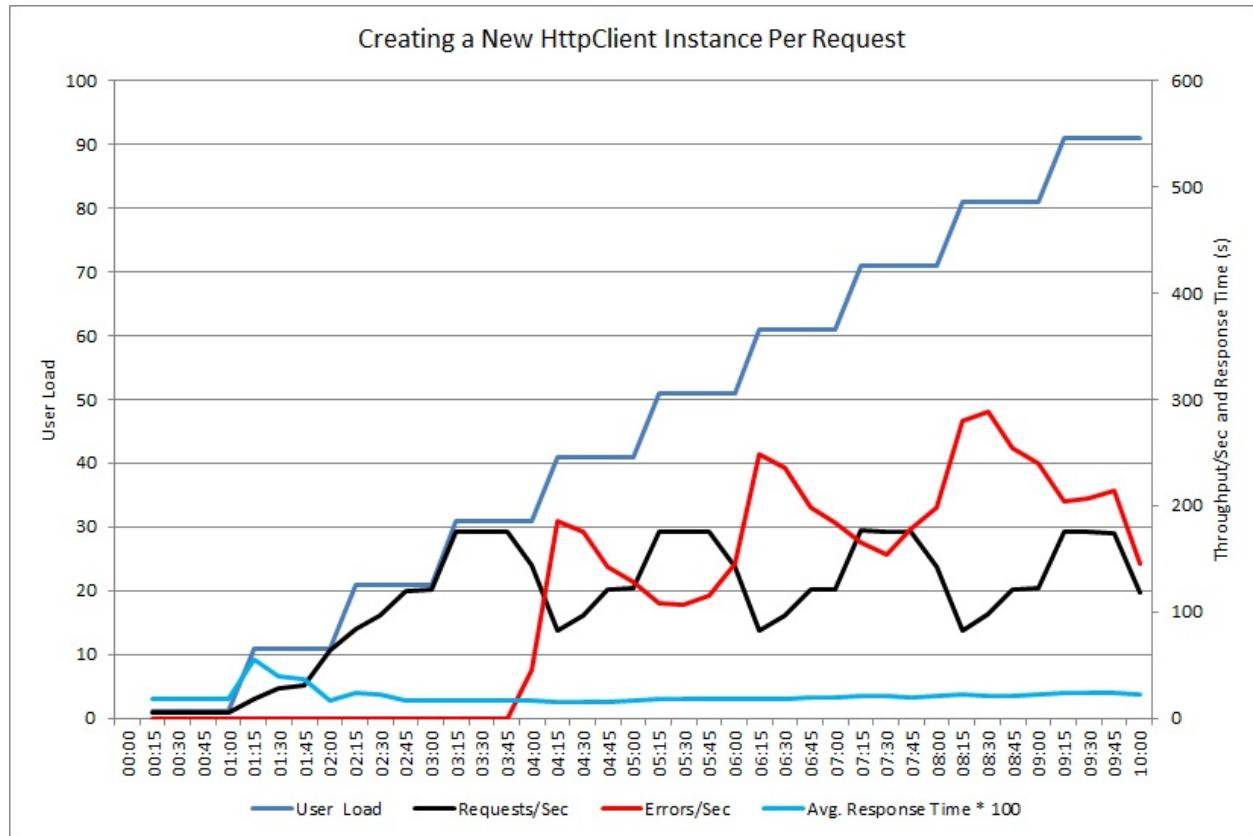
Examine telemetry data and find correlations

The next image shows data captured using thread profiling, over the same period corresponding as the previous image. The system spends a significant time opening socket connections, and even more time closing them and handling socket exceptions.



Performing load testing

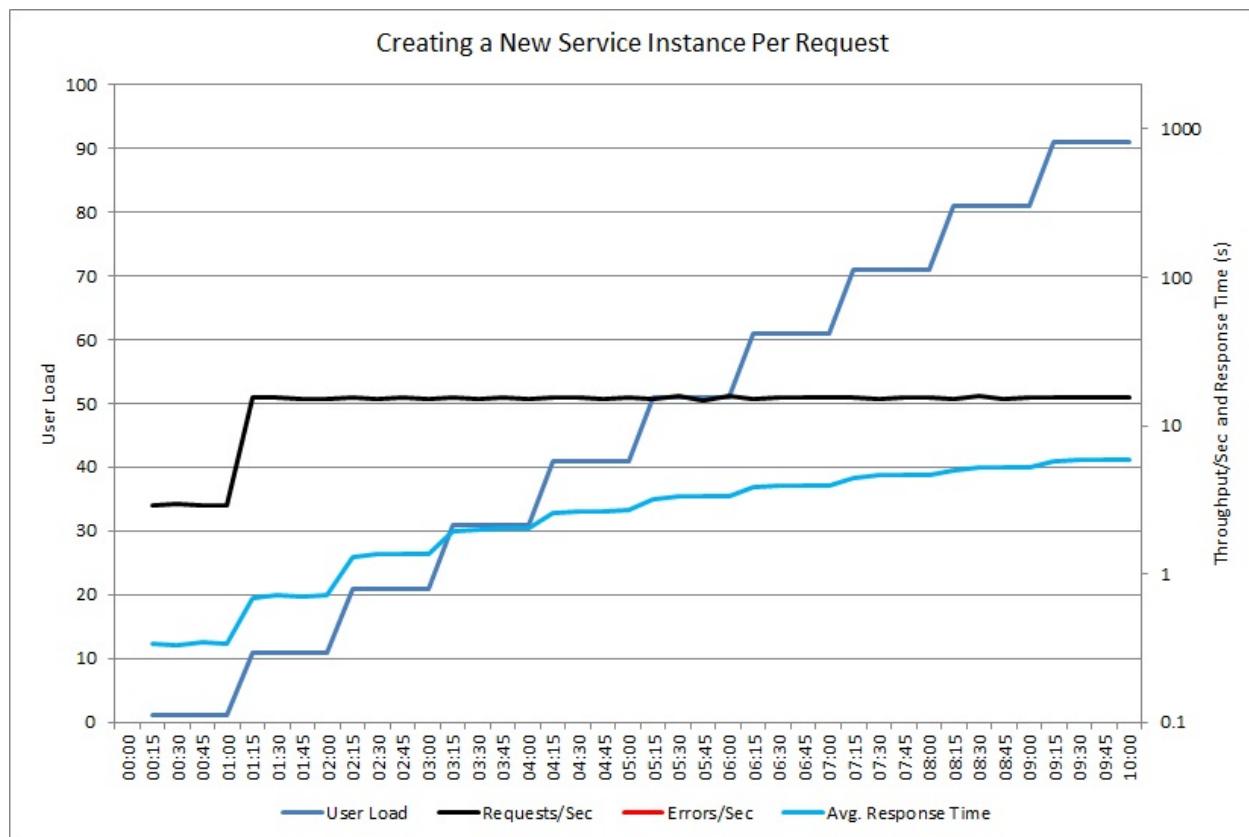
Use load testing to simulate the typical operations that users might perform. This can help to identify which parts of a system suffer from resource exhaustion under varying loads. Perform these tests in a controlled environment rather than the production system. The following graph shows the throughput of requests handled by the `NewHttpClientInstancePerRequest` controller as the user load increases to 100 concurrent users.



At first, the volume of requests handled per second increases as the workload increases. At about 30 users, however, the volume of successful requests reaches a limit, and the system starts to generate exceptions. From then on, the volume of exceptions gradually increases with the user load.

The load test reported these failures as HTTP 500 (Internal Server) errors. Reviewing the telemetry showed that these errors were caused by the system running out of socket resources, as more and more `HttpClient` objects were created.

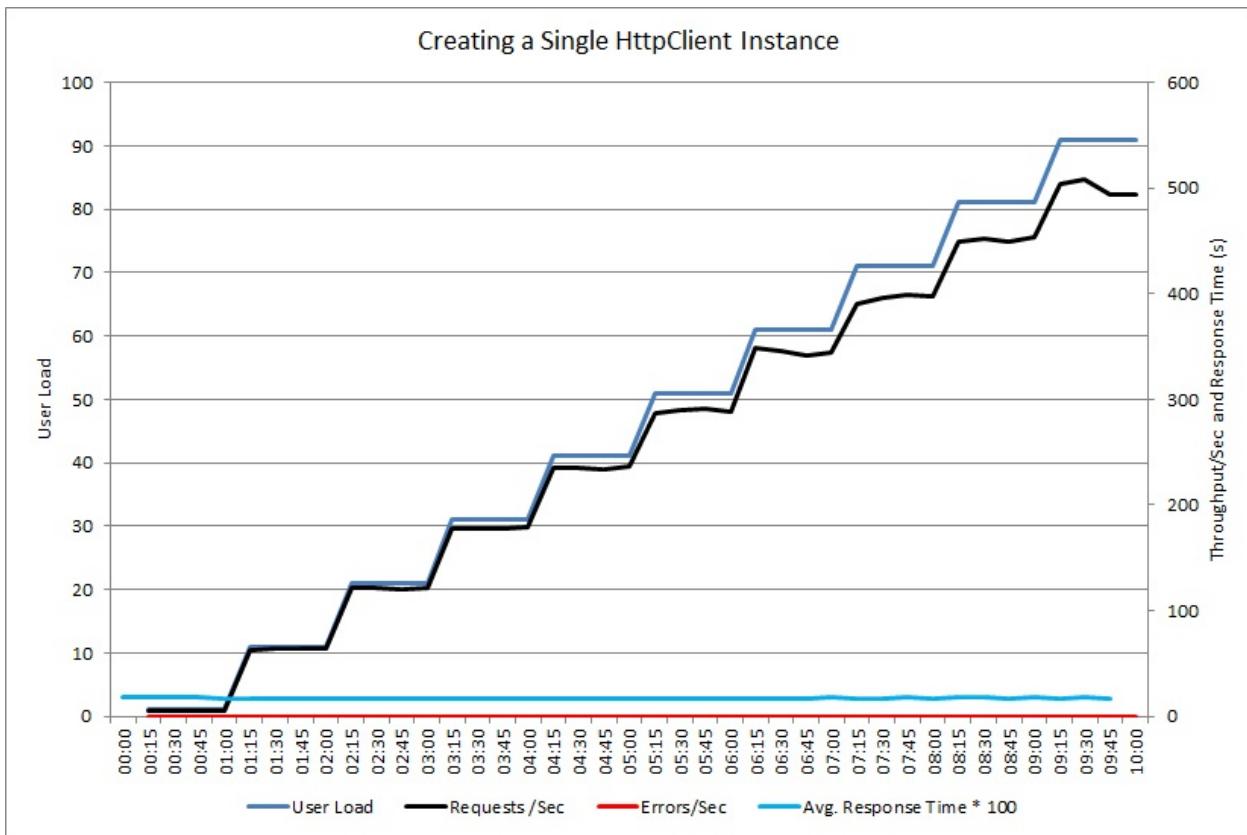
The next graph shows a similar test for a controller that creates the custom `ExpensiveToCreateService` object.



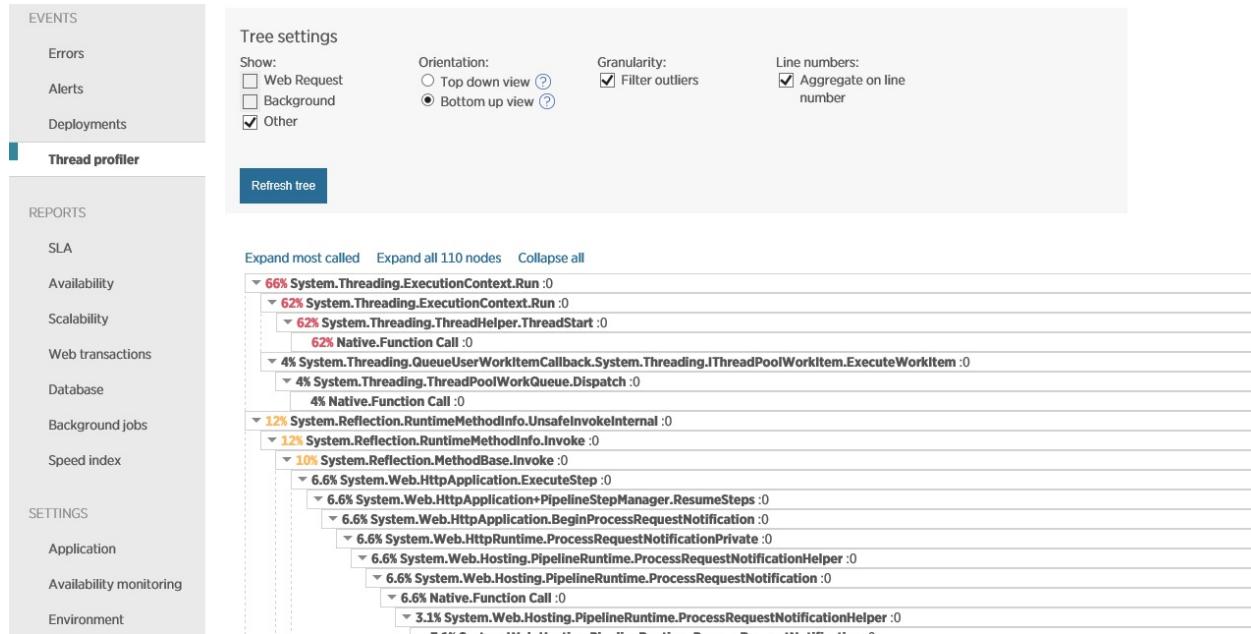
This time, the controller does not generate any exceptions, but throughput still reaches a plateau, while the average response time increases by a factor of 20. (The graph uses a logarithmic scale for response time and throughput.) Telemetry showed that creating new instances of the `ExpensiveToCreateService` was the main cause of the problem.

Implement the solution and verify the result

After switching the `GetProductAsync` method to share a single `HttpClient` instance, a second load test showed improved performance. No errors were reported, and the system was able to handle an increasing load of up to 500 requests per second. The average response time was cut in half, compared with the previous test.

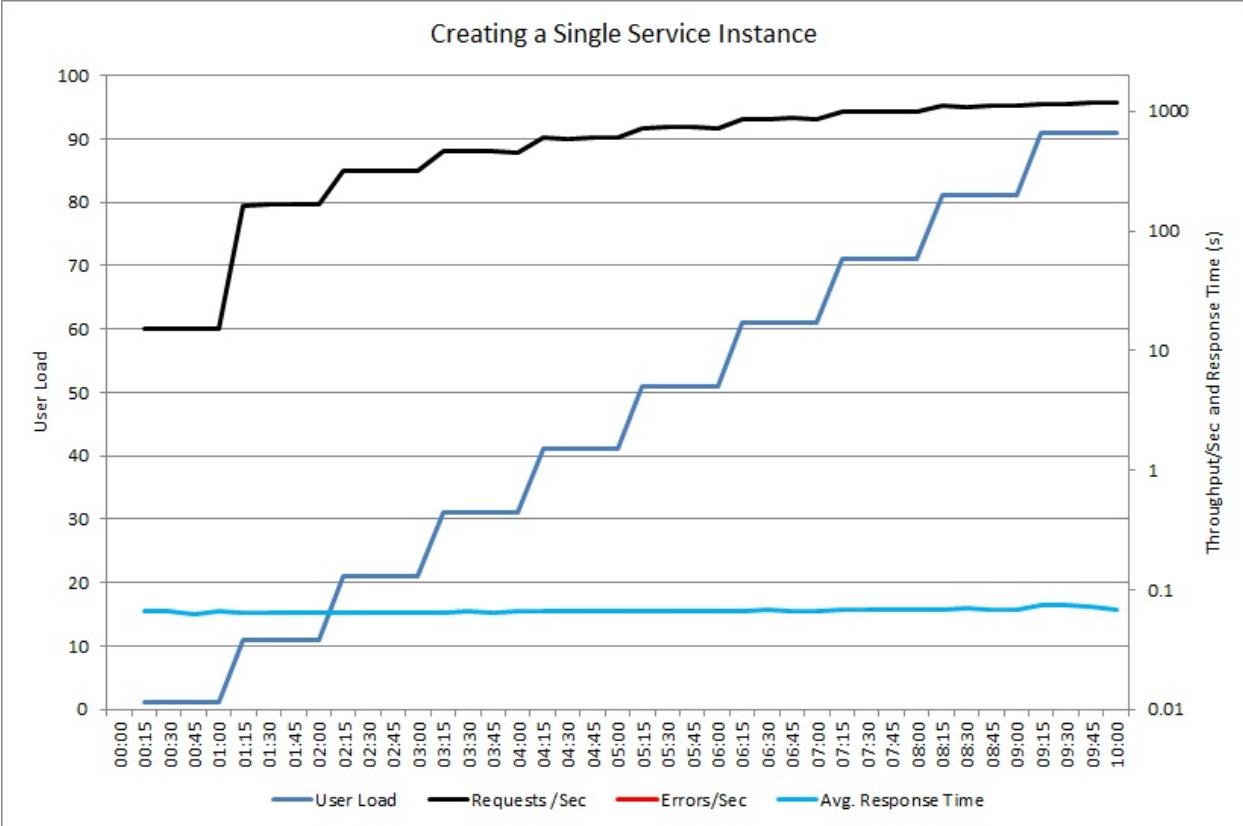


For comparison, the following image shows the stack trace telemetry. This time, the system spends most of its time performing real work, rather than opening and closing sockets.



The next graph shows a similar load test using a shared instance of the `ExpensiveToCreateService` object. Again, the volume of handled requests increases in line with the user load, while the average response time remains low.

Creating a Single Service Instance



Monolithic Persistence antipattern

Article • 12/16/2022

Putting all of an application's data into a single data store can hurt performance, either because it leads to resource contention, or because the data store is not a good fit for some of the data.

Problem description

Historically, applications have often used a single data store, regardless of the different types of data that the application might need to store. Usually this was done to simplify the application design, or else to match the existing skill set of the development team.

Modern cloud-based systems often have additional functional and nonfunctional requirements, and need to store many heterogeneous types of data, such as documents, images, cached data, queued messages, application logs, and telemetry. Following the traditional approach and putting all of this information into the same data store can hurt performance, for two main reasons:

- Storing and retrieving large amounts of unrelated data in the same data store can cause contention, which in turn leads to slow response times and connection failures.
- Whichever data store is chosen, it might not be the best fit for all of the different types of data, or it might not be optimized for the operations that the application performs.

The following example shows an ASP.NET Web API controller that adds a new record to a database and also records the result to a log. The log is held in the same database as the business data. You can find the complete sample [here](#).

C#

```
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:

C#

```
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 [Azure Cosmos DB](#), but use [Azure Cache for Redis](#) to cache temporary data.
- If you follow this guidance but still reach the limits of the database, you may need to scale up the database. Also consider scaling horizontally and partitioning the load across database servers. However, partitioning may require redesigning the application. For more information, see [Data partitioning](#).

How to detect the problem

The system will likely slow down dramatically and eventually fail, as the system runs out of resources such as database connections.

You can perform the following steps to help identify the cause.

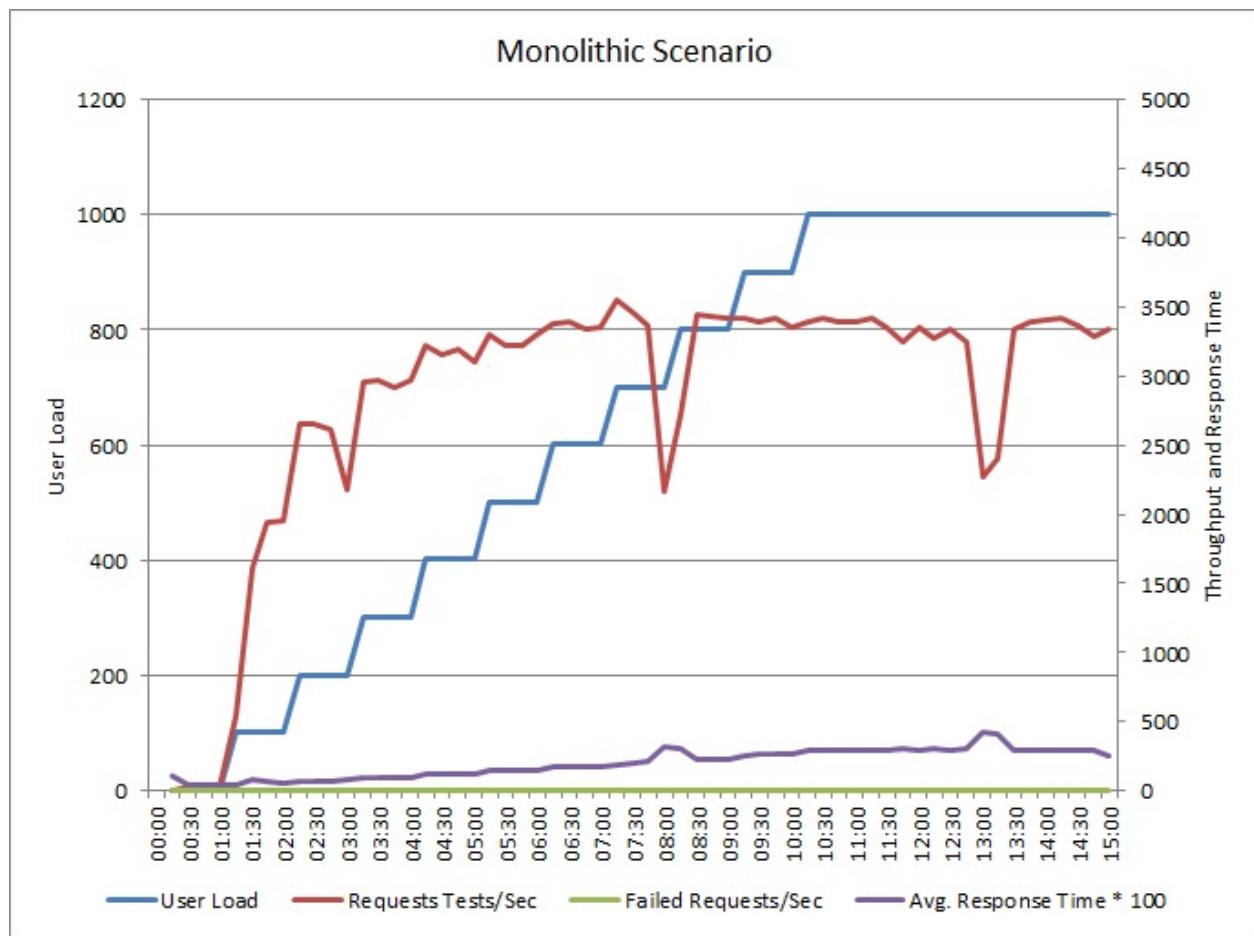
1. Instrument the system to record the key performance statistics. Capture timing information for each operation, as well as the points where the application reads and writes data.
2. If possible, monitor the system running for a few days in a production environment to get a real-world view of how the system is used. If this is not possible, run scripted load tests with a realistic volume of virtual users performing a typical series of operations.
3. Use the telemetry data to identify periods of poor performance.
4. Identify which data stores were accessed during those periods.
5. Identify data storage resources that might be experiencing contention.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Instrument and monitor the system

The following graph shows the results of load testing the sample application described earlier. The test used a step load of up to 1000 concurrent users.



As the load increases to 700 users, so does the throughput. But at that point, throughput levels off, and the system appears to be running at its maximum capacity. The average response gradually increases with user load, showing that the system can't keep up with demand.

Identify periods of poor performance

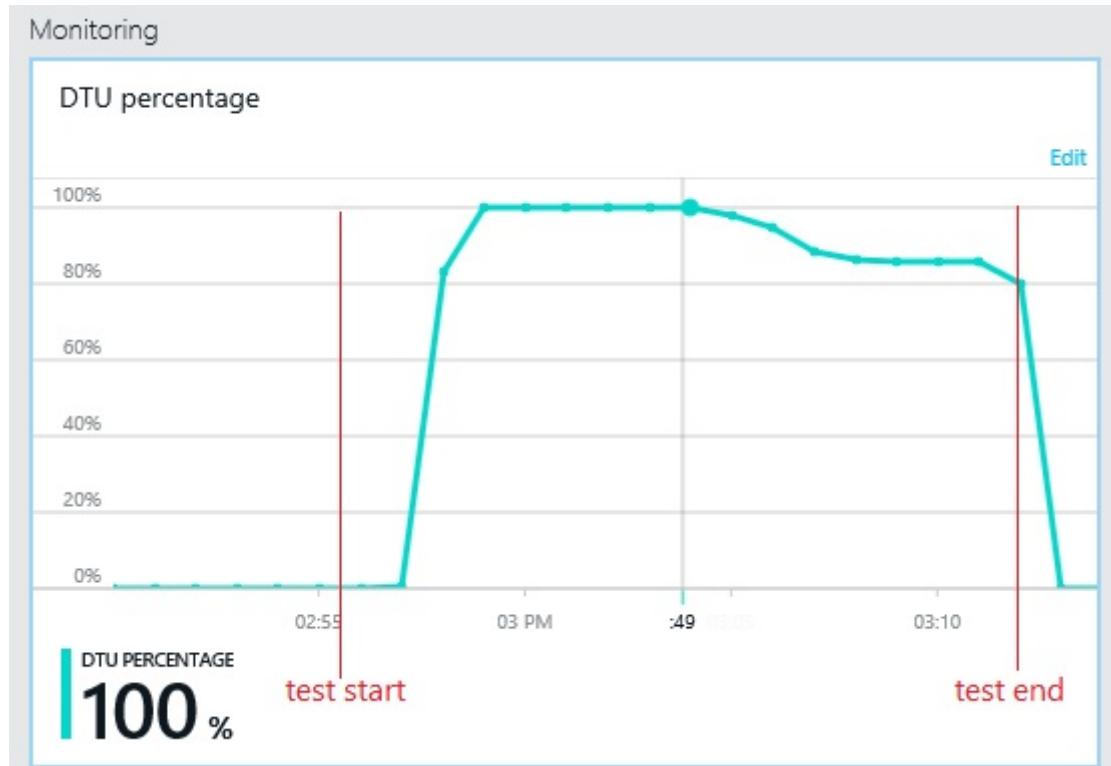
If you are monitoring the production system, you might notice patterns. For example, response times might drop off significantly at the same time each day. This could be caused by a regular workload or scheduled batch job, or just because the system has more users at certain times. You should focus on the telemetry data for these events.

Look for correlations between increased response times and increased database activity or I/O to shared resources. If there are correlations, it means the database might be a bottleneck.

Identify which data stores are accessed during those periods

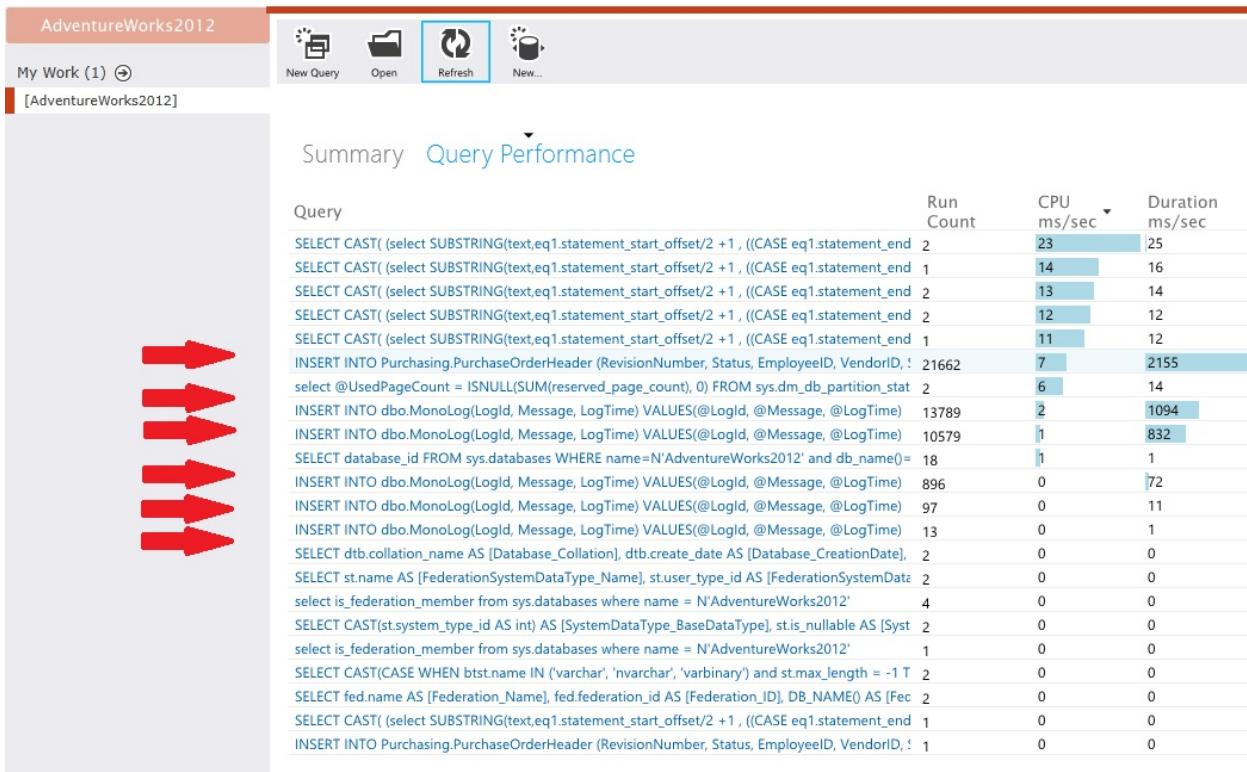
The next graph shows the utilization of database throughput units (DTU) during the load test. (A DTU is a measure of available capacity, and is a combination of CPU utilization,

memory allocation, I/O rate.) Utilization of DTUs quickly reached 100%. This is roughly the point where throughput peaked in the previous graph. Database utilization remained very high until the test finished. There is a slight drop toward the end, which could be caused by throttling, competition for database connections, or other factors.



Examine the telemetry for the data stores

Instrument the data stores to capture the low-level details of the activity. In the sample application, the data access statistics showed a high volume of insert operations performed against both the `PurchaseOrderHeader` table and the `MonoLog` table.



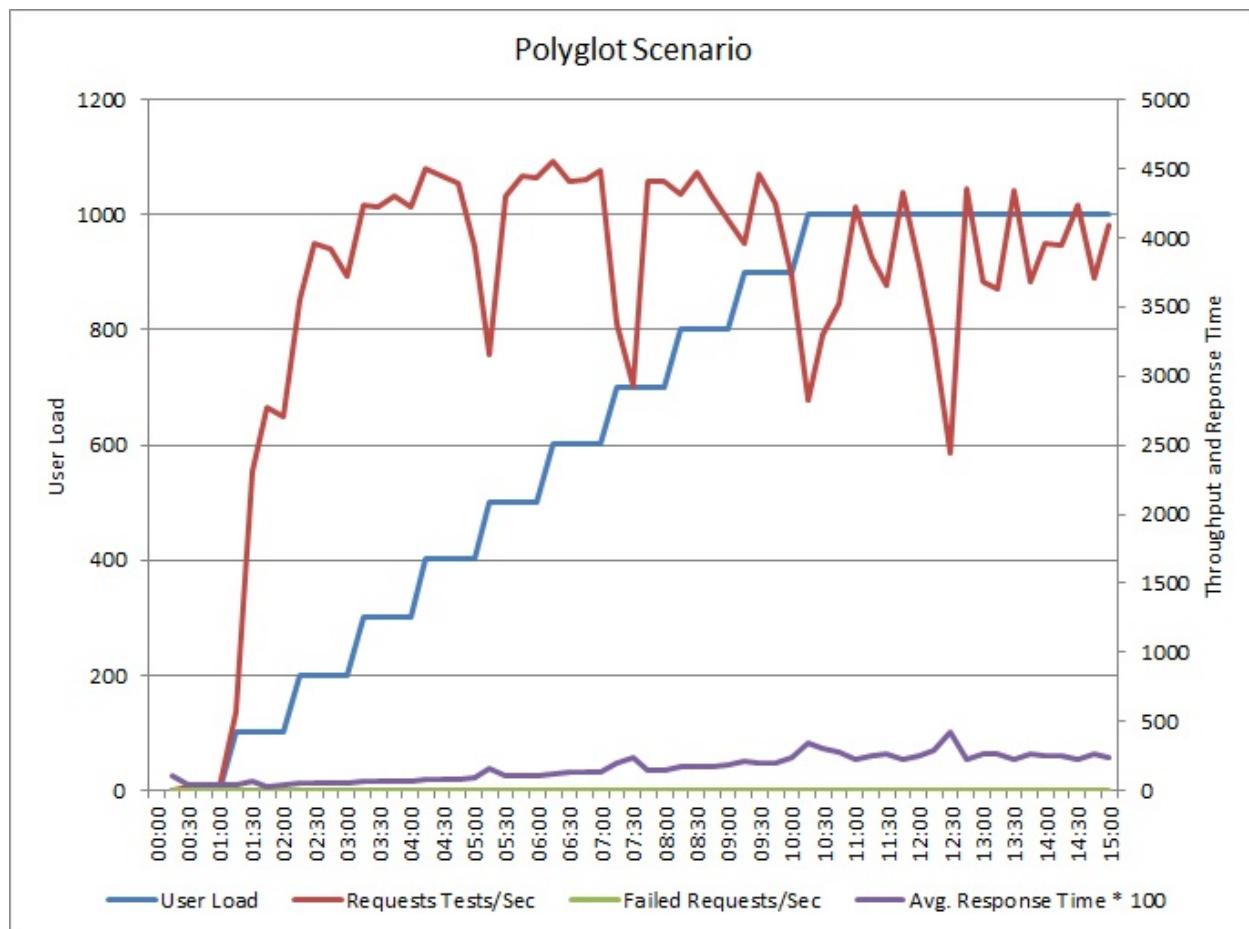
Identify resource contention

At this point, you can review the source code, focusing on the points where contended resources are accessed by the application. Look for situations such as:

- Data that is logically separate being written to the same store. Data such as logs, reports, and queued messages should not be held in the same database as business information.
- A mismatch between the choice of data store and the type of data, such as large blobs or XML documents in a relational database.
- Data with significantly different usage patterns that share the same store, such as high-write/low-read data being stored with low-write/high-read data.

Implement the solution and verify the result

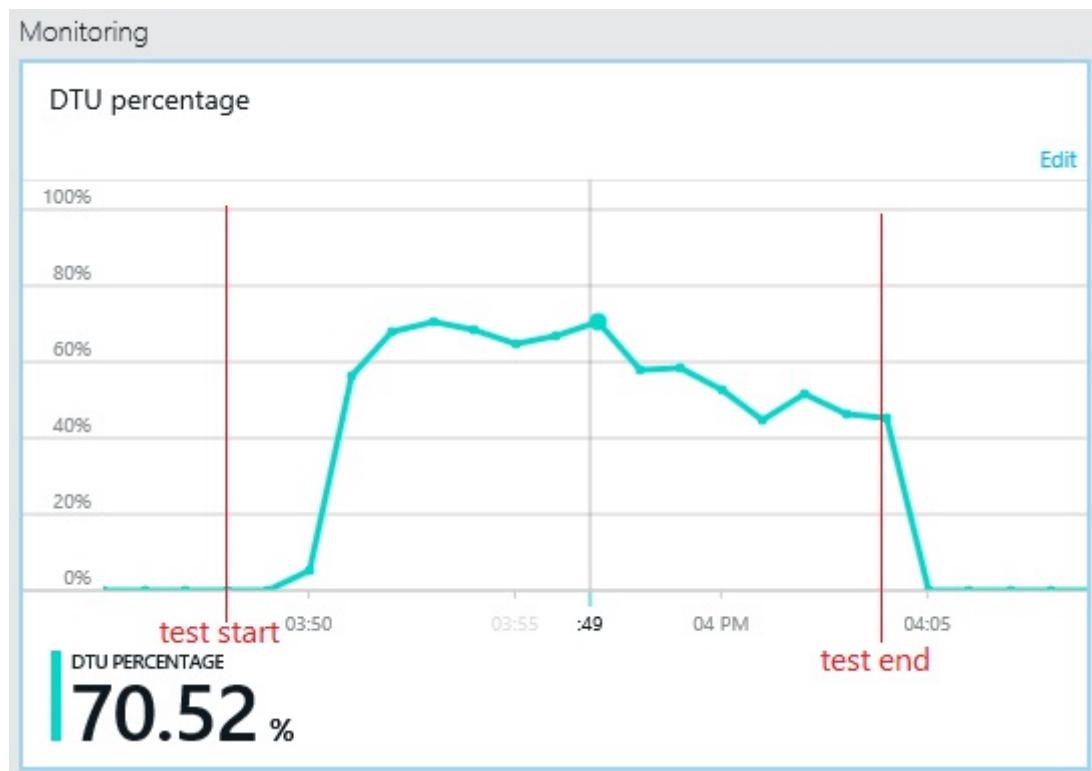
The application was changed to write logs to a separate data store. Here are the load test results:



The pattern of throughput is similar to the earlier graph, but the point at which performance peaks is approximately 500 requests per second higher. The average response time is marginally lower. However, these statistics don't tell the full story. Telemetry for the business database shows that DTU utilization peaks at around 75%, rather than 100%.



Similarly, the maximum DTU utilization of the log database only reaches about 70%. The databases are no longer the limiting factor in the performance of the system.



Related resources

- Choose the right data store
- Criteria for choosing a data store
- Data Access for Highly Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence
- Data partitioning

No Caching antipattern

Article • 12/16/2022

Anti-patterns are common design flaws that can break your software or applications under stress situations and should not be overlooked. A *no caching antipattern* occurs when a cloud application that handles many concurrent requests, repeatedly fetches the same data. This can reduce performance and scalability.

When data is not cached, it can cause a number of undesirable behaviors, including:

- Repeatedly fetching the same information from a resource that is expensive to access, in terms of I/O overhead or latency.
- Repeatedly constructing the same objects or data structures for multiple requests.
- Making excessive calls to a remote service that has a service quota and throttles clients past a certain limit.

In turn, these problems can lead to poor response times, increased contention in the data store, and poor scalability.

Examples of no caching antipattern

The following example uses Entity Framework to connect to a database. Every client request results in a call to the database, even if multiple requests are fetching exactly the same data. The cost of repeated requests, in terms of I/O overhead and data access charges, can accumulate quickly.

C#

```
public class PersonRepository : IPersonRepository
{
    public async Task<Person> GetAsync(int id)
    {
        using (var context = new AdventureWorksContext())
        {
            return await context.People
                .Where(p => p.Id == id)
                .FirstOrDefaultAsync()
                .ConfigureAwait(false);
        }
    }
}
```

You can find the complete sample [here ↗](#).

This antipattern typically occurs because:

- Not using a cache is simpler to implement, and it works fine under low loads. Caching makes the code more complicated.
- The benefits and drawbacks of using a cache are not clearly understood.
- There is concern about the overhead of maintaining the accuracy and freshness of cached data.
- An application was migrated from an on-premises system, where network latency was not an issue, and the system ran on expensive high-performance hardware, so caching wasn't considered in the original design.
- Developers aren't aware that caching is a possibility in a given scenario. For example, developers may not think of using ETags when implementing a web API.

How to fix the no caching antipattern

The most popular caching strategy is the *on-demand* or *cache-aside* strategy.

- On read, the application tries to read the data from the cache. If the data isn't in the cache, the application retrieves it from the data source and adds it to the cache.
- On write, the application writes the change directly to the data source and removes the old value from the cache. It will be retrieved and added to the cache the next time it is required.

This approach is suitable for data that changes frequently. Here is the previous example updated to use the [Cache-Aside](#) pattern.

C#

```
public class CachedPersonRepository : IPersonRepository
{
    private readonly PersonRepository _innerRepository;

    public CachedPersonRepository(PersonRepository innerRepository)
    {
        _innerRepository = innerRepository;
    }

    public async Task<Person> GetAsync(int id)
    {
        return await CacheService.GetAsync<Person>("p:" + id, () =>
_innerRepository.GetAsync(id)).ConfigureAwait(false);
    }
}

public class CacheService
```

```

{
    private static ConnectionMultiplexer _connection;

    public static async Task<T> GetAsync<T>(string key, Func<Task<T>>
loadCache, double expirationTimeInMinutes)
    {
        IDatabase cache = Connection.GetDatabase();
        T value = await GetAsync<T>(cache, key).ConfigureAwait(false);
        if (value == null)
        {
            // Value was not found in the cache. Call the lambda to get the
            value from the database.
            value = await loadCache().ConfigureAwait(false);
            if (value != null)
            {
                // Add the value to the cache.
                await SetAsync(cache, key, value,
                expirationTimeInMinutes).ConfigureAwait(false);
            }
        }
        return value;
    }
}

```

Notice that the `GetAsync` method now calls the `CacheService` class, rather than calling the database directly. The `CacheService` class first tries to get the item from Azure Cache for Redis. If the value isn't found in the cache, the `CacheService` invokes a lambda function that was passed to it by the caller. The lambda function is responsible for fetching the data from the database. This implementation decouples the repository from the particular caching solution, and decouples the `CacheService` from the database.

Considerations for caching strategy

- If the cache is unavailable, perhaps because of a transient failure, don't return an error to the client. Instead, fetch the data from the original data source. However, be aware that while the cache is being recovered, the original data store could be swamped with requests, resulting in timeouts and failed connections. (After all, this is one of the motivations for using a cache in the first place.) Use a technique such as the [Circuit Breaker pattern](#) to avoid overwhelming the data source.
- Applications that cache dynamic data should be designed to support eventual consistency.
- For web APIs, you can support client-side caching by including a Cache-Control header in request and response messages, and using ETags to identify versions of objects. For more information, see [API implementation](#).

- You don't have to cache entire entities. If most of an entity is static but only a small piece changes frequently, cache the static elements and retrieve the dynamic elements from the data source. This approach can help to reduce the volume of I/O being performed against the data source.
- In some cases, if volatile data is short-lived, it can be useful to cache it. For example, consider a device that continually sends status updates. It might make sense to cache this information as it arrives, and not write it to a persistent store at all.
- To prevent data from becoming stale, many caching solutions support configurable expiration periods, so that data is automatically removed from the cache after a specified interval. You may need to tune the expiration time for your scenario. Data that is highly static can stay in the cache for longer periods than volatile data that may become stale quickly.
- If the caching solution doesn't provide built-in expiration, you may need to implement a background process that occasionally sweeps the cache, to prevent it from growing without limits.
- Besides caching data from an external data source, you can use caching to save the results of complex computations. Before you do that, however, instrument the application to determine whether the application is really CPU bound.
- It might be useful to prime the cache when the application starts. Populate the cache with the data that is most likely to be used.
- Always include instrumentation that detects cache hits and cache misses. Use this information to tune caching policies, such what data to cache, and how long to hold data in the cache before it expires.
- If the lack of caching is a bottleneck, then adding caching may increase the volume of requests so much that the web front end becomes overloaded. Clients may start to receive HTTP 503 (Service Unavailable) errors. These are an indication that you should scale out the front end.

How to detect a no caching antipattern

You can perform the following steps to help identify whether lack of caching is causing performance problems:

1. Review the application design. Take an inventory of all the data stores that the application uses. For each, determine whether the application is using a cache. If

possible, determine how frequently the data changes. Good initial candidates for caching include data that changes slowly, and static reference data that is read frequently.

2. Instrument the application and monitor the live system to find out how frequently the application retrieves data or calculates information.
3. Profile the application in a test environment to capture low-level metrics about the overhead associated with data access operations or other frequently performed calculations.
4. Perform load testing in a test environment to identify how the system responds under a normal workload and under heavy load. Load testing should simulate the pattern of data access observed in the production environment using realistic workloads.
5. Examine the data access statistics for the underlying data stores and review how often the same data requests are repeated.

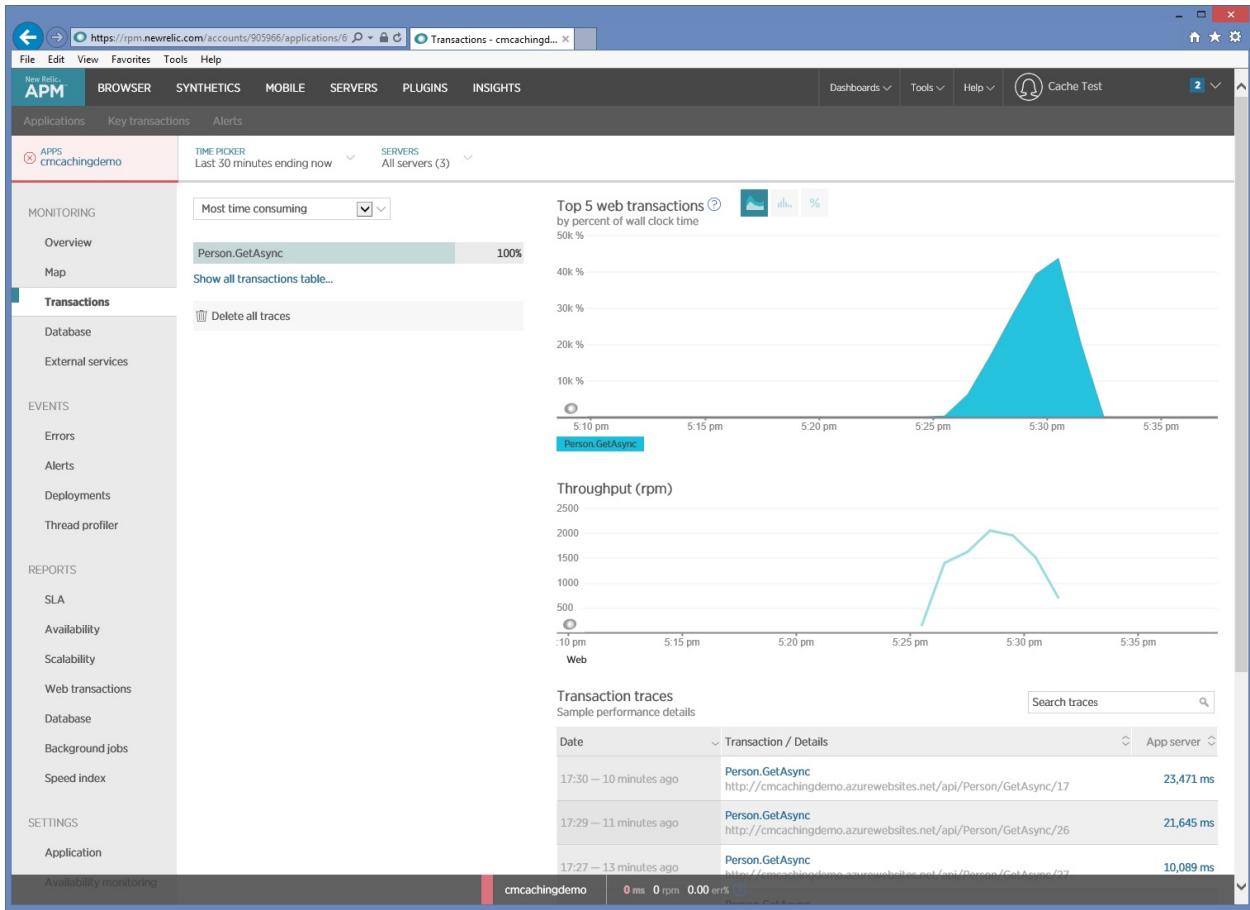
Example diagnosis

The following sections apply these steps to the sample application described earlier.

Instrument the application and monitor the live system

Instrument the application and monitor it to get information about the specific requests that users make while the application is in production.

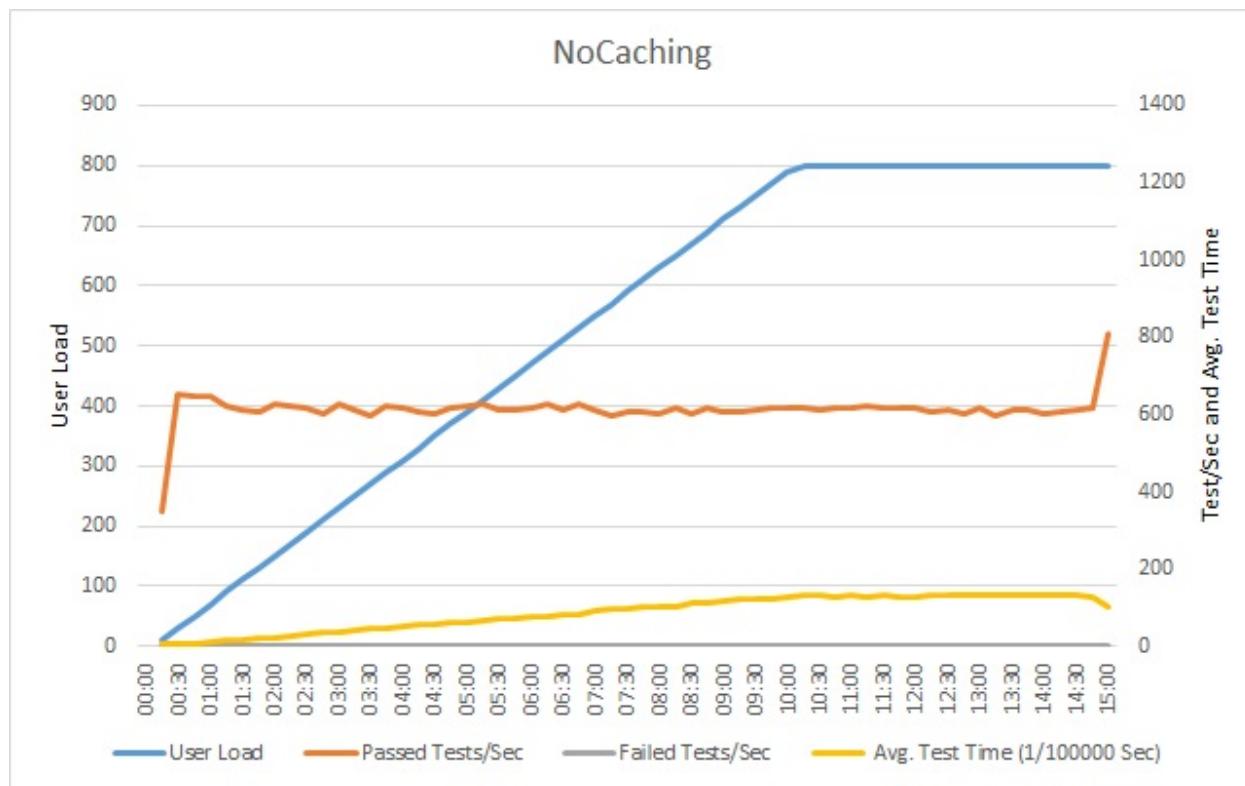
The following image shows monitoring data captured by [New Relic](#) during a load test. In this case, the only HTTP GET operation performed is `Person/GetAsync`. But in a live production environment, knowing the relative frequency that each request is performed can give you insight into which resources should be cached.



If you need a deeper analysis, you can use a profiler to capture low-level performance data in a test environment (not the production system). Look at metrics such as I/O request rates, memory usage, and CPU utilization. These metrics may show a large number of requests to a data store or service, or repeated processing that performs the same calculation.

Load test the application

The following graph shows the results of load testing the sample application. The load test simulates a step load of up to 800 users performing a typical series of operations.



The number of successful tests performed each second reaches a plateau, and additional requests are slowed as a result. The average test time steadily increases with the workload. The response time levels off once the user load peaks.

Examine data access statistics

Data access statistics and other information provided by a data store can give useful information, such as which queries are repeated most frequently. For example, in Microsoft SQL Server, the `sys.dm_exec_query_stats` management view has statistical information for recently executed queries. The text for each query is available in the `sys.dm_exec_query_plan` view. You can use a tool such as SQL Server Management Studio to run the following SQL query and determine how frequently queries are performed.

SQL

```
SELECT UseCounts, Text, Query_Plan
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)
```

The `UseCount` column in the results indicates how frequently each query is run. The following image shows that the third query was run more than 250,000 times, significantly more than any other query.

```

SELECT UseCounts, Text, Query_Plan
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)

```

100 % <

Results Messages

	UseCounts	Text	Query_Plan
1	1	SELECT UseCounts, Text, Query_Plan FROM sys.dm...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
2	1	select is_federation_member from sys.databases where ...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
3	256049	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
4	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
5	2	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
6	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
7	2	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
8	3	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
9	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
10	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
11	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
12	3	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
13	2	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
14	2	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
15	2	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
16	2	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
17	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
18	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
19	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
20	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
21	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
22	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
23	2	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."
24	1	(@p_linq_0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com..."

Here is the SQL query that is causing so many database requests:

SQL

```

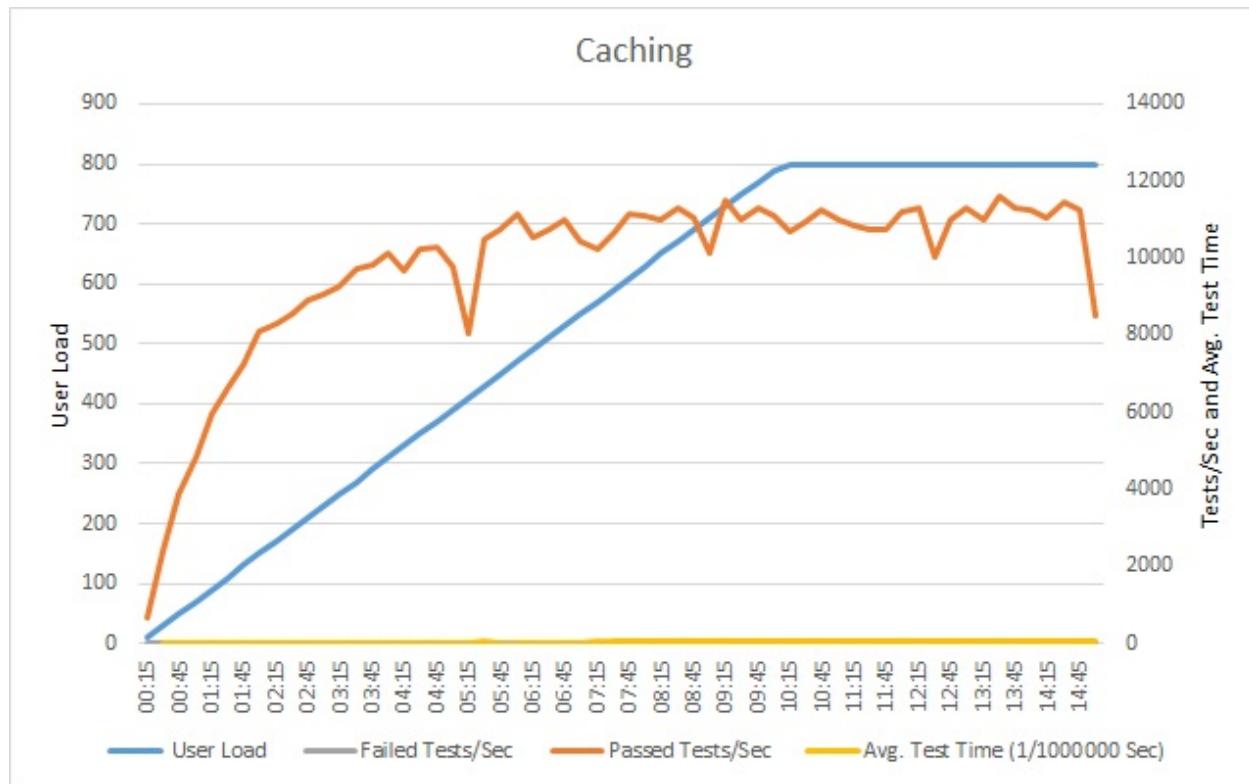
(@p_linq_0 int)SELECT TOP (2)
[Extent1].[BusinessEntityId] AS [BusinessEntityId],
[Extent1].[FirstName] AS [FirstName],
[Extent1].[LastName] AS [LastName]
FROM [Person].[Person] AS [Extent1]
WHERE [Extent1].[BusinessEntityId] = @p_linq_0

```

This is the query that Entity Framework generates in `GetByIdAsync` method shown earlier.

Implement the cache strategy solution and verify the result

After you incorporate a cache, repeat the load tests and compare the results to the earlier load tests without a cache. Here are the load test results after adding a cache to the sample application.



The volume of successful tests still reaches a plateau, but at a higher user load. The request rate at this load is significantly higher than earlier. Average test time still increases with load, but the maximum response time is 0.05 ms, compared with 1 ms earlier—a 20× improvement.

Related resources

- [API implementation best practices](#)
- [Cache-Aside pattern](#)
- [Caching best practices](#)
- [Circuit Breaker pattern](#)

Noisy Neighbor antipattern

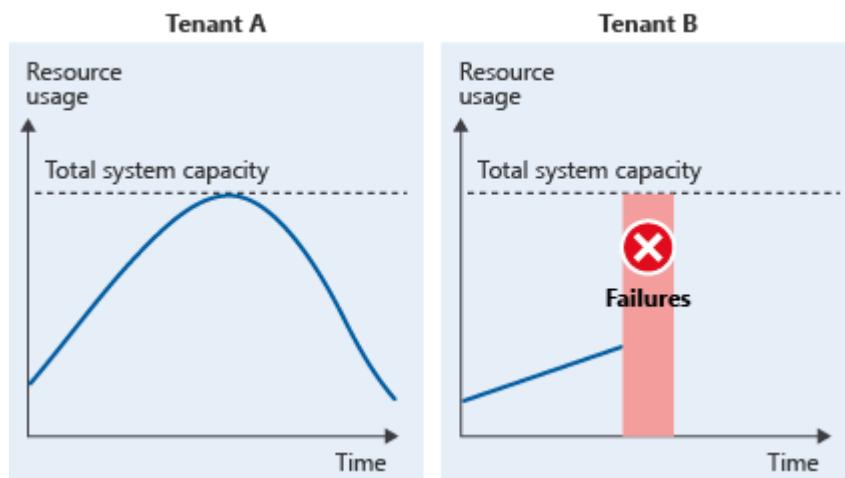
Azure

Multitenant systems share resources between tenants. Because tenants use the same shared resources, the activity of one tenant can have a negative impact on another tenant's use of the system.

Problem description

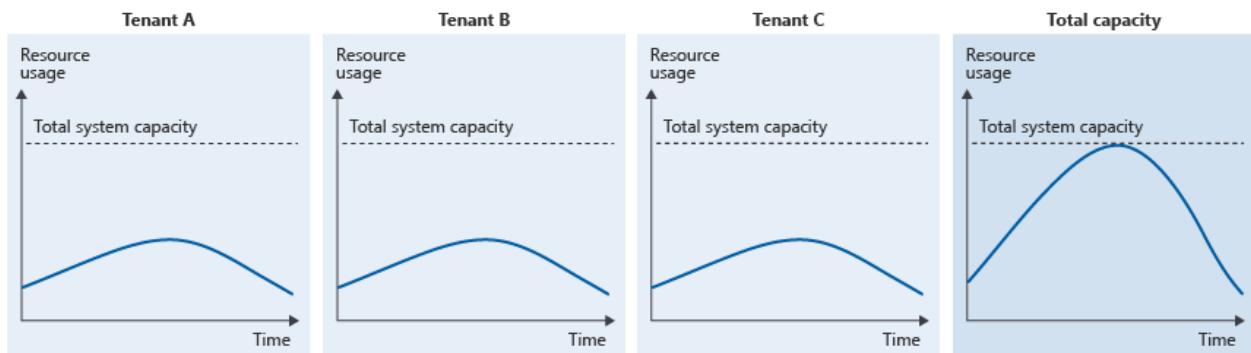
When you build a service to be shared by multiple customers or tenants, you can build it to be *multitenanted*. A benefit of multitenant systems is that resources can be pooled and shared among tenants. This often results in lower costs and improved efficiency. However, if a single tenant uses a disproportionate amount of the resources available in the system, the overall performance of the system can suffer. The *noisy neighbor* problem occurs when one tenant's performance is degraded because of the activities of another tenant.

Consider an example multitenant system with two tenants. Tenant A's usage patterns and tenant B's usage patterns coincide, which means that at peak times, the total resource usage is higher than the capacity of the system:



It's likely that whichever tenant's request that arrived first will take precedence. Then the other tenant will experience a noisy neighbor problem. Alternatively, both tenants might find their performance suffers.

The noisy neighbor problem also occurs even when each individual tenant is consuming relatively small amounts of the system's capacity, but the collective resource usage of many tenants results in a peak in overall usage:



This can happen when you have multiple tenants that all have similar usage patterns, or where you haven't provisioned sufficient capacity for the collective load on the system.

How to fix the problem

Noisy neighbor problems are an inherent risk in multitenant systems, and it's not possible to completely eliminate the possibility of being affected by a noisy neighbor. However, there are some steps that both clients and service providers can take to reduce the likelihood of noisy neighbor problems, or to mitigate their effects when they're observed.

Actions that clients can take

- Ensure your application handles [service throttling](#), to reduce making unnecessary requests to the service. Ensure that your application follows good practices to [retry requests that received a transient failure response](#).
- Purchase reserved capacity, if available. For example, when using Azure Cosmos DB, purchase [reserved throughput](#), and when using ExpressRoute, [provision separate circuits for environments that are sensitive to performance](#).
- Migrate to a single-tenant instance of the service, or to a service tier with stronger isolation guarantees. For example, when using Service Bus, [migrate to the premium tier](#), and when using Azure Cache for Redis, [provision a standard or premium tier cache](#).

Actions that service providers can take

- Monitor the resource usage for your system. Monitor both the overall resource usage and the resources that each tenant uses. Configure alerts to detect spikes in resource usage, and if possible, configure automation to automatically mitigate known issues by [scaling up or out](#).
- Apply resource governance. Consider applying policies that avoid a single tenant overwhelming the system and reducing the capacity available to others. This step

might take the form of quota enforcement, through the [Throttling pattern](#) or the [Rate Limiting pattern](#).

- **Provision more infrastructure.** This process might involve scaling up by upgrading some of your solution components, or it might involve scaling out by provisioning additional shards, if you follow the [Sharding pattern](#), or stamps, if you follow the [Deployment Stamps pattern](#).
- **Enable tenants to purchase pre-provisioned or reserved capacity.** This capacity provides tenants with more certainty that your solution adequately handles their workload.
- **Smooth out tenants' resource usage.** For example, you might try one of the following approaches:
 - If you host multiple instances of your solution, consider rebalancing tenants across the instances or stamps. For example, consider placing tenants with predictable and similar usage patterns across multiple stamps, to flatten the peaks in their usage.
 - Consider whether you have background processes or resource-intensive workloads that aren't time-sensitive. Run these workloads asynchronously at off-peak times, to preserve your peak resource capacity for time-sensitive workloads.
- **Check whether your downstream services provide controls to mitigate noisy neighbor problems.** For example, when using Kubernetes, [consider using pod limits](#), and when using Service Fabric, [consider using the built-in governance capabilities](#).
- **Restrict the operations that tenants can perform.** For example, prevent tenants from executing operations that will run very large database queries, such as by specifying a maximum returnable record count or time limit on queries. This action mitigates the risk of tenants taking actions that might negatively impact other tenants.
- **Provide a Quality of Service (QoS) system.** When you apply QoS, you prioritize some processes or workloads ahead of others. By factoring QoS into your design and architecture, you can ensure that high-priority operations take precedence when there's pressure on your resources.

Considerations

In most cases, individual tenants don't intend to cause noisy neighbor issues. Individual tenants might not even be aware that their workloads cause noisy neighbor issues for others. However, it's also possible that some tenants might exploit vulnerabilities in shared components to attack a service, either individually or by executing a distributed denial of service (DDoS) attack.

Regardless of the cause, it's important to treat these problems as resource governance issues, and to apply usage quotas, throttling, and governance controls to mitigate the problem.

Note

Make sure that you tell your clients about any throttling that you apply, or any usage quotas on your service. It's important that they reliably handle failed requests, and that they aren't surprised by any limitations or quotas you apply.

How to detect the problem

From a client's perspective, the noisy neighbor problem typically manifests as failed server requests, or requests that take a long time to complete. In particular, if the same request succeeds at other times and appears to fail randomly, there might be a noisy neighbor issue. Client applications should record telemetry to track the success rate and performance of the requests to services, and the applications should also record baseline performance metrics for comparison purposes.

From a service's perspective, the noisy neighbor issue might appear in several ways:

- **Spikes in resource usage.** It's important to have a clear understanding of your normal baseline resource usage, and to configure monitoring and alerts to detect spikes in resource usage. Ensure you consider all of the resources that could affect your service's performance or availability. These resources include metrics like server CPU and memory usage, disk IO, database usage, network traffic, and metrics that are exposed by managed services, such as the number of requests and the synthetic and abstract performance metrics, such as the Azure Cosmos DB request units.
- **Failures when performing an operation for a tenant.** In particular, look for failures that occur when a tenant isn't using a large portion of the system's resources. Such a pattern might indicate that the tenant is a victim of the noisy neighbor problem. Consider tracking the resource consumption by tenant. For example, when using Azure Cosmos DB, consider logging the request units used for each request, and add the tenant's identifier as a dimension to the telemetry, so that you can aggregate the request unit consumption for each tenant.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [John Downs](#) | Principal Customer Engineer, FastTrack for Azure

Other contributors:

- [Chad Kittel](#) | Principal Software Engineer
- [Paolo Salvatori](#) | Principal Customer Engineer, FastTrack for Azure
- [Daniel Scott-Raynsford](#) | Partner Technology Strategist
- [Arsen Vladimirskiy](#) | Principal Customer Engineer, FastTrack for Azure

To see non-public LinkedIn profiles, sign in to LinkedIn.

Related resources

- [Architectural considerations for a multitenant solution](#)
- [Transient fault handling best practices](#)

Retry Storm antipattern

Article • 11/15/2022

When a service is unavailable or busy, having clients retry their connections too frequently can cause the service to struggle to recover, and can make the problem worse. It also doesn't make sense to retry forever, since requests are typically only valid for a defined period of time.

Problem description

In the cloud, services sometimes experience problems and become unavailable to clients, or have to throttle or rate limit their clients. While it's a good practice for clients to retry failed connections to services, it's important they do not retry too frequently or for too long. Retries within a short period of time are unlikely to succeed since the services likely will not have recovered. Also, services can be put under even more stress when lots of connection attempts are made while they're trying to recover, and repeated connection attempts may even overwhelm the service and make the underlying problem worse.

The following example illustrates a scenario where a client connects to a server-based API. If the request doesn't succeed, then the client retries immediately, and keeps retrying forever. Often this sort of behavior is more subtle than in this example, but the same principle applies.

C#

```
public async Task<string> GetDataFromServer()
{
    while(true)
    {
        var result = await
httpClient.GetAsync(string.Format("http://{0}:8080/api/...", hostName));
        if (result.IsSuccessStatusCode) break;
    }

    // ... Process result.
}
```

How to fix the problem

Client applications should follow some best practices to avoid causing a retry storm.

- Cap the number of retry attempts, and don't keep retrying for a long period of time. While it might seem easy to write a `while(true)` loop, you almost certainly don't want to actually retry for a long period of time, since the situation that led to the request being initiated has probably changed. In most applications, retrying for a few seconds or minutes is sufficient.
- Pause between retry attempts. If a service is unavailable, retrying immediately is unlikely to succeed. Gradually increase the amount of time you wait between attempts, for example by using an [exponential backoff strategy](#).
- Gracefully handle errors. If the service isn't responding, consider whether it makes sense to abort the attempt and return an error back to the user or caller of your component. Consider these failure scenarios when designing your application.
- Consider using the [Circuit Breaker pattern](#), which is designed specifically to help avoid retry storms.
- If the server provides a `retry-after` response header, make sure you don't attempt to retry until the specified time period has elapsed.
- Use official SDKs when communicating to Azure services. These SDKs generally have built-in retry policies and protections against causing or contributing to retry storms. If you're communicating with a service that doesn't have an SDK, or where the SDK doesn't handle retry logic correctly, consider using a library like [Polly](#) (for .NET) or [retry](#) (for JavaScript) to handle your retry logic correctly and avoid writing the code yourself.
- If you're running in an environment that supports it, use a service mesh (or another abstraction layer) to send outbound calls. Typically these tools, such as [Dapr](#), support retry policies and automatically follow best practices, like backing off after repeated attempts. This approach means you don't have to write retry code yourself.
- Consider batching requests and using request pooling where available. Many SDKs handle request batching and connection pooling on your behalf, which will reduce the total number of outbound connection attempts your application makes, although you still need to be careful not to retry these connections too frequently.

Services should also protect themselves against retry storms.

- Add a gateway layer so you can shut off connections during an incident. This is an example of the [Bulkhead pattern](#). Azure provides many different gateway services for different types of solutions including [Front Door](#), [Application Gateway](#), and [API Management](#).
- Throttle requests at your gateway, which ensures you won't accept so many requests that your back-end components can't continue to operate.
- If you're throttling, send back a `retry-after` header to help clients understand when to reattempt their connections.

Considerations

- Clients should consider the type of error returned. Some error types don't indicate a failure of the service, but instead indicate that the client sent an invalid request. For example, if a client application receives a `400 Bad Request` error response, retrying the same request probably is not going to help since the server is telling you that your request is not valid.
- Clients should consider the length of time that makes sense to reattempt connections. The length of time you should retry for will be driven by your business requirements and whether you can reasonably propagate an error back to a user or caller. In most applications, retrying for a few seconds or minutes is sufficient.

How to detect the problem

From a client's perspective, symptoms of this problem could include very long response or processing times, along with telemetry that indicates repeated attempts to retry the connection.

From a service's perspective, symptoms of this problem could include a large number of requests from one client within a short period of time, or a large number of requests from a single client while recovering from outages. Symptoms could also include difficulty when recovering the service, or ongoing cascading failures of the service right after a fault has been repaired.

Example diagnosis

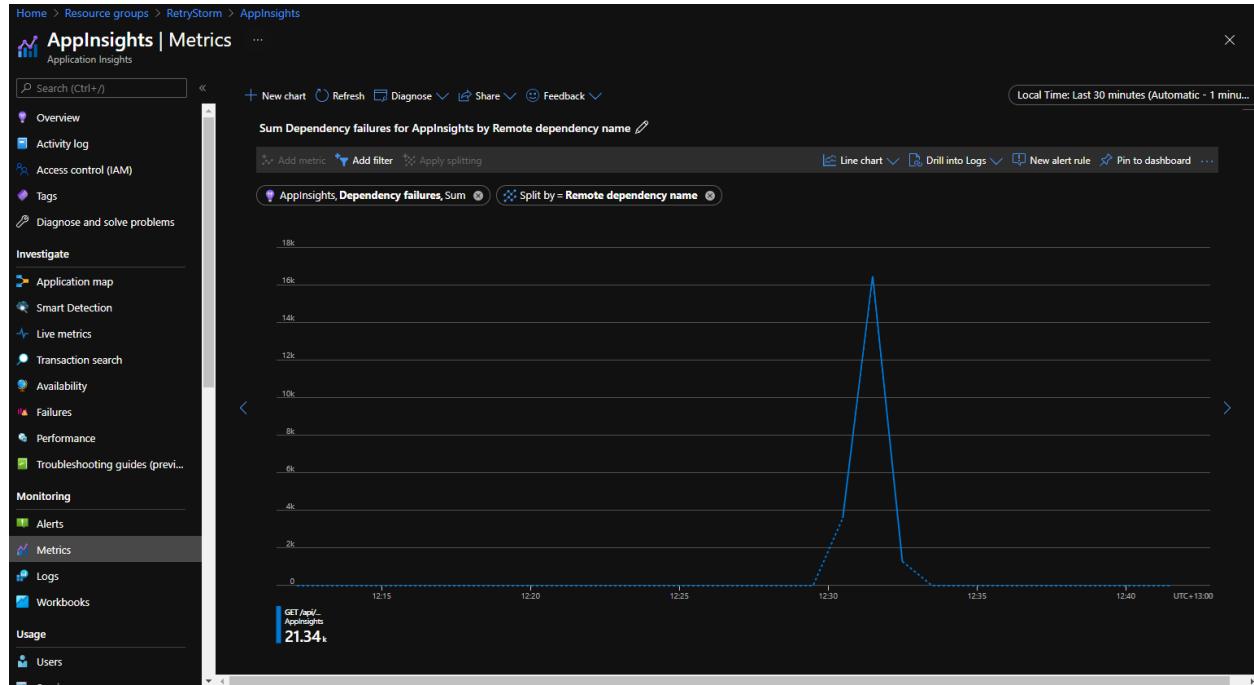
The following sections illustrate one approach to detecting a potential retry storm, both on the client side and the service side.

Identifying from client telemetry

[Azure Application Insights](#) records telemetry from applications and makes the data available for querying and visualization. Outbound connections are tracked as dependencies, and information about them can be accessed and charted to identify when a client is making a large number of outbound requests to the same service.

The following graph was taken from the Metrics tab within the Application Insights portal, and displaying the *Dependency failures* metric split by *Remote dependency name*.

This illustrates a scenario where there were a large number (over 21,000) of failed connection attempts to a dependency within a short time.



Identifying from server telemetry

Server applications may be able to detect large numbers of connections from a single client. In the following example, Azure Front Door acts as a gateway for an application, and [has been configured to log](#) all requests to a Log Analytics workspace.

The following Kusto query can be executed against Log Analytics. It will identify client IP addresses that have sent large numbers of requests to the application within the last day.

```
Kusto

AzureDiagnostics
| where ResourceType == "FRONDOORS" and Category == "FrontdoorAccessLog"
| where TimeGenerated > ago(1d)
| summarize count() by bin(TimeGenerated, 1h), clientIp_s
| order by count_ desc
```

Executing this query during a retry storm shows a large number of connection attempts from a single IP address.

The screenshot shows the Azure Log Analytics workspace interface. On the left, there's a sidebar with navigation links like 'Tables', 'Queries', and 'Favorites'. The main area displays a query in the editor:

```
1 AzureDiagnostics
2 | where ResourceType == "FRONDOORS" and Category == "FrontdoorAccessLog"
3 | where TimeGenerated > ago(1d)
4 | summarize count() by bin(TimeGenerated, 1h), clientIp_s
5 | order by count_ desc
```

Below the editor, the results pane shows a table titled 'Completed' with the following data:

TimeGenerated [UTC]	clientIp_s	count_
23/02/2021, 11:00:00.000 pm	[REDACTED]	81,608
23/02/2021, 11:00:00.000 pm	[REDACTED]	5

Related resources

- [Retry pattern](#)
- [Circuit Breaker pattern](#)
- [Transient fault handling best practices](#)
- [Service-specific retry guidance](#)

Synchronous I/O antipattern

Article • 12/16/2022

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.

C#

```
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`.

C#

```
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.

C#

```

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.

C#

```

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.

C#

```
// Asynchronous wrapper around synchronous library method
private async Task<int> LibraryI0operationAsync()
{
    return await Task.Run(() => LibraryI0operation());
}
```

Now the calling code can await on the wrapper:

C#

```
// Invoke the asynchronous wrapper using a task
await LibraryI0operationAsync();
```

Considerations

- I/O operations that are expected to be very short lived and are unlikely to cause contention might be more performant as synchronous operations. An example might be reading small files on an SSD drive. The overhead of dispatching a task to another thread, and synchronizing with that thread when the task completes, might outweigh the benefits of asynchronous I/O. However, these cases are relatively rare, and most I/O operations should be done asynchronously.
- Improving I/O performance may cause other parts of the system to become bottlenecks. For example, unblocking threads might result in a higher volume of concurrent requests to shared resources, leading in turn to resource starvation or throttling. If that becomes a problem, you might need to scale out the number of web servers or partition data stores to reduce contention.

How to detect the problem

For users, the application may seem unresponsive periodically. The application might fail with timeout exceptions. These failures could also return HTTP 500 (Internal Server)

errors. On the server, incoming client requests might be blocked until a thread becomes available, resulting in excessive request queue lengths, manifested as HTTP 503 (Service Unavailable) errors.

You can perform the following steps to help identify the problem:

1. Monitor the production system and determine whether blocked worker threads are constraining throughput.
2. If requests are being blocked due to lack of threads, review the application to determine which operations may be performing I/O synchronously.
3. Perform controlled load testing of each operation that is performing synchronous I/O, to find out whether those operations are affecting system performance.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Monitor web server performance

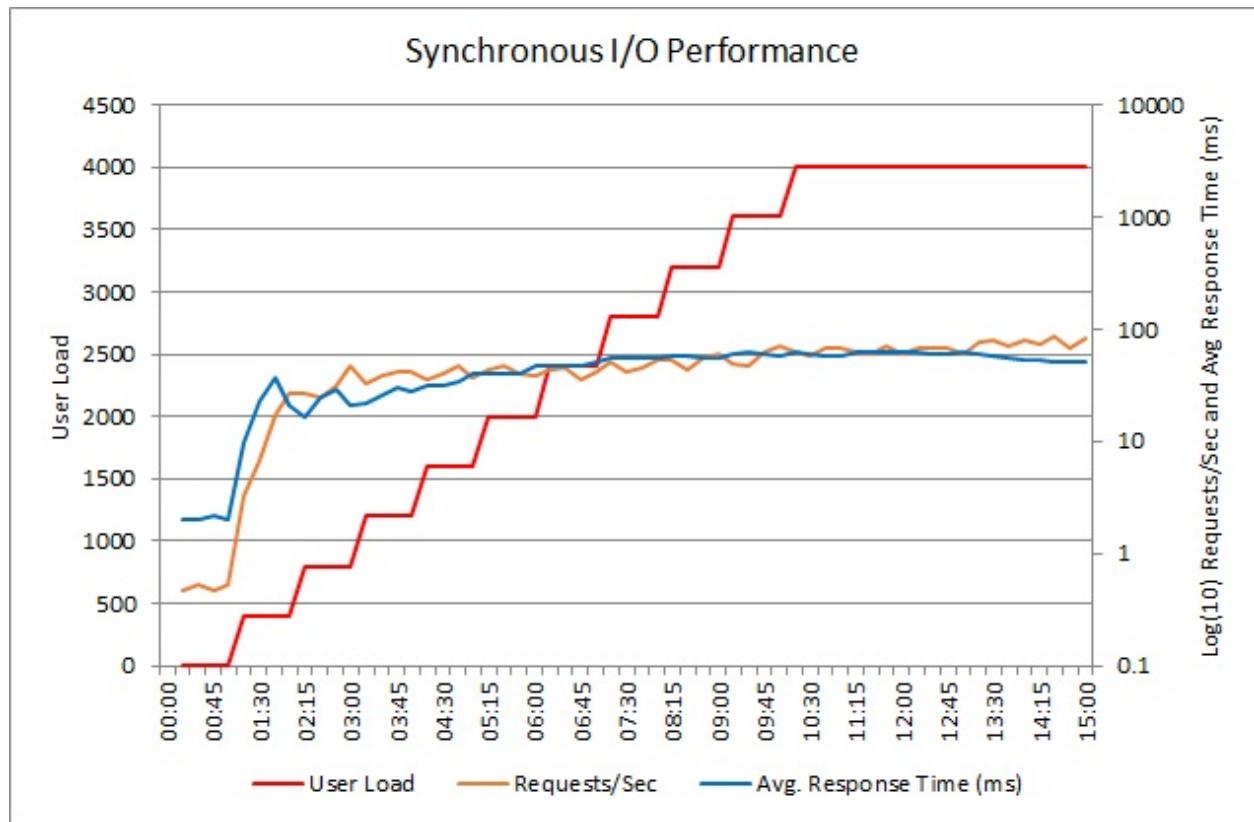
For Azure web applications and web roles, it's worth monitoring the performance of the IIS web server. In particular, pay attention to the request queue length to establish whether requests are being blocked waiting for available threads during periods of high activity. You can gather this information by enabling Azure diagnostics. For more information, see:

- [Monitor Apps in Azure App Service](#)
- [Create and use performance counters in an Azure application](#)

Instrument the application to see how requests are handled once they have been accepted. Tracing the flow of a request can help to identify whether it is performing slow-running calls and blocking the current thread. Thread profiling can also highlight requests that are being blocked.

Load test the application

The following graph shows the performance of the synchronous `GetUserProfile` method shown earlier, under varying loads of up to 4000 concurrent users. The application is an ASP.NET application running in an Azure Cloud Service web role.



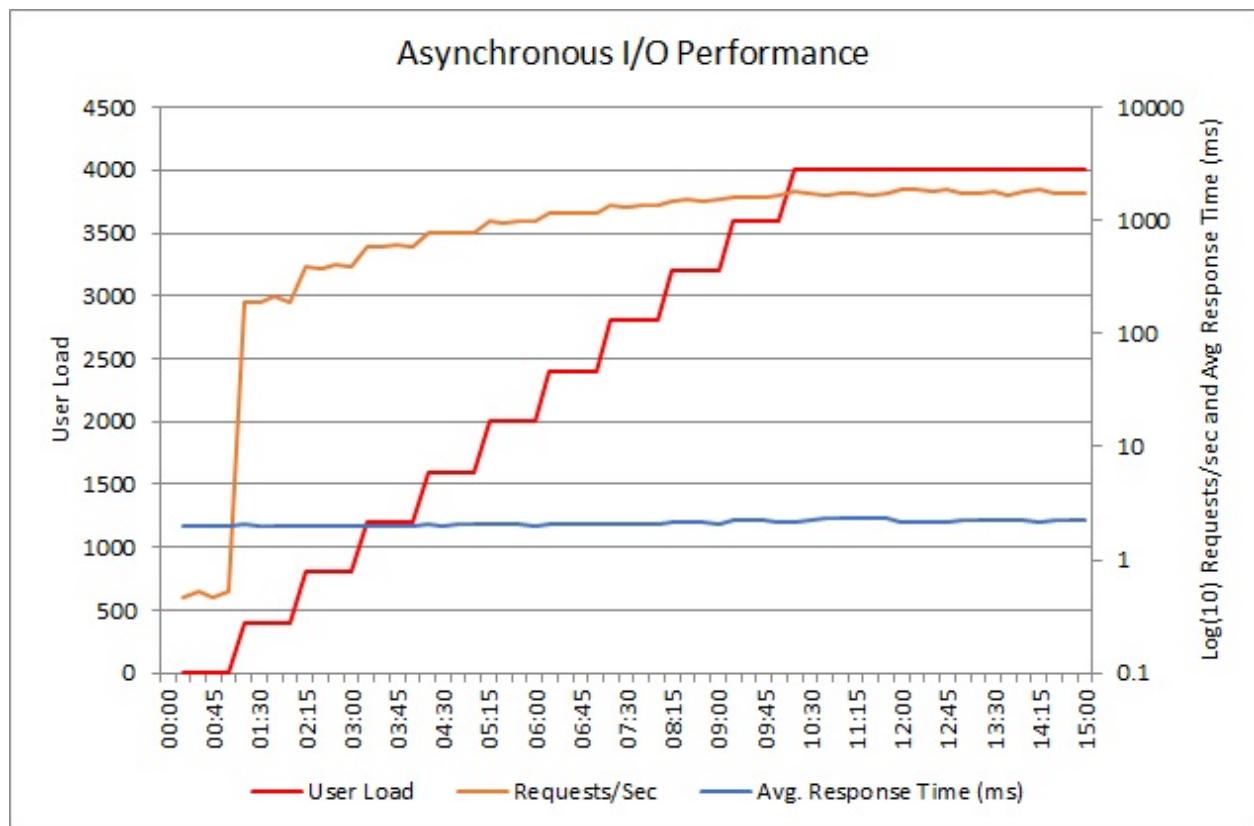
The synchronous operation is hard-coded to sleep for 2 seconds, to simulate synchronous I/O, so the minimum response time is slightly over 2 seconds. When the load reaches approximately 2500 concurrent users, the average response time reaches a plateau, although the volume of requests per second continues to increase. Note that the scale for these two measures is logarithmic. The number of requests per second doubles between this point and the end of the test.

In isolation, it's not necessarily clear from this test whether the synchronous I/O is a problem. Under heavier load, the application may reach a tipping point where the web server can no longer process requests in a timely manner, causing client applications to receive time-out exceptions.

Incoming requests are queued by the IIS web server and handed to a thread running in the ASP.NET thread pool. Because each operation performs I/O synchronously, the thread is blocked until the operation completes. As the workload increases, eventually all of the ASP.NET threads in the thread pool are allocated and blocked. At that point, any further incoming requests must wait in the queue for an available thread. As the queue length grows, requests start to time out.

Implement the solution and verify the result

The next graph shows the results from load testing the asynchronous version of the code.



Throughput is far higher. Over the same duration as the previous test, the system successfully handles a nearly tenfold increase in throughput, as measured in requests per second. Moreover, the average response time is relatively constant and remains approximately 25 times smaller than the previous test.

Responsible innovation: a best practices toolkit

Article • 01/24/2023

Responsible innovation is a toolkit that helps developers become good stewards for the future of science and its effect on society. This toolkit provides a set of practices in development for anticipating and addressing the potential negative impacts of technology on people. We're sharing this as an early-stage practice for feedback and learning.

Judgment Call

[Judgment Call](#) is an award-winning responsible innovation game and team-based activity that puts Microsoft's AI principles of fairness, privacy and security, reliability and safety, transparency, inclusion, and accountability into action. The game provides an easy-to-use method for cultivating stakeholder empathy through *scenario-imagining*. Game participants write product reviews from the perspective of a particular stakeholder, describing what kind of impact and harm the technology could produce from their point of view.

Harms Modeling

[Harms Modeling](#) is a framework for product teams, grounded in four core pillars of responsible innovation that examines how technology can negatively impact people's lives:

- Injuries
- Denial of consequential services
- Infringement on human rights
- Erosion of democratic & societal structures

Similar to *Security Threat Modeling*, Harms Modeling enables product teams to anticipate potential real-world impacts of technology, which is a cornerstone of responsible development.

Community Jury

[Community Jury](#) is a technique that brings together diverse stakeholders impacted by technology. It's an adaptation of the [citizen jury](#). The stakeholders can learn from

experts about a project, deliberate together, and give feedback on use cases and product design. In addition, this responsible innovation technique allows project teams to collaborate with researchers to identify stakeholder values and understand the perceptions and concerns of impacted stakeholders.

Sustainability

The [Azure Well-Architected Framework sustainability workload guidance](#) are recommendations and principles to increase cloud efficiency across Azure workloads. This guidance is developed in partnership with the Green Software Foundation for an improved sustainability posture to help create new business value while reducing the operational footprint. This guide helps drive innovation responsibly by evaluating the impact workload design decisions have on the climate and gives concrete advice on how to build for a better tomorrow.

Judgment Call

Article • 10/11/2022

Judgment Call is a game and team-based activity that puts Microsoft's AI principles of fairness, privacy and security, reliability and safety, transparency, inclusion, and accountability into action. The game provides an easy-to-use method for cultivating stakeholder empathy by imagining their scenarios. Game participants write product reviews from the perspective of a particular stakeholder, describing what kind of impact and harms the technology could produce from their point of view.

<https://www.youtube-nocookie.com/embed/LXqIAXEMGIO>

Benefits

Technology builders need practical methods to incorporate ethics in product development, by considering the values of diverse stakeholders and how technology may uphold or not uphold those values. The goal of the game is to imagine potential outcomes of a product or platform by gaining a better understanding of stakeholders, and what they need and expect.

The game helps people discuss challenging topics in a safe space within the context of gameplay, and gives technology creator a vocabulary to facilitate ethics discussions during product development. It gives managers and designers an interactive tool to lead ethical dialogue with their teams to incorporate ethical design in their products.

The theoretical basis and initial outcomes of the Judgment Call game were presented at the 2019 ACM Design Interactive Systems conference and the game was a finalist in the Fast Company 2019 Innovation by Design Awards (Social Goods category). The game has been presented to thousands of people in the US and internationally. While the largest group facilitated in game play has been over 100, each card deck can involve 1-10 players. This game is not intended to be a substitute for performing robust user research. Rather, it's an exercise to help tech builders learn how to exercise their moral imagination.

Preparation

Judgment Call uses role play to surface ethical concerns in product development so players will anticipate societal impact, write product reviews, and explore mitigations. Players think of what kind of harms and impacts the technology might produce by writing product reviews from the point of view of a stakeholder.

To prepare for this game, download the [printable Judgment Call game kit](#).

During the Game

The players are expected to go through the following steps in this game:

1. Identify the product

Start by identifying a product that your team is building, or a scenario you are working on. For example, if your team is developing synthetic voice technology, your scenario could be developing a voice assistant for online learning.

2. Pass the cards

Once you have decided on the scenario, pass out the cards. Each player receives a stakeholder card, an ethical principle card, a rating card, and a review form. The following is a description of these cards:

- **Stakeholder:** This represents the group of people impacted by the technology or scenario you've chosen. Stakeholders can also include the makers (those proposing and working on the technology) as well as the internal project sponsors (managers and executives who are supportive of the project).
- **Principle:** This includes Microsoft's ethical principles of fairness, privacy and security, reliability and safety, transparency, inclusion, and accountability.
- **Rating:** Star cards give a rating: 1-star is poor, 3-star is mediocre, 5-star is excellent.

3. Brainstorm and identify stakeholders

As a group, brainstorm all the stakeholders that could be directly or indirectly affected by the technology. Then assign each player a stakeholder from the list. In some cases, the Judgment Call decks may have pre-populated stakeholder cards, but players can add additional cards relevant to their situation.

The deck provides following categories of stakeholders:

- **Direct:** These use the technology and/or the technology is used on them.
- **Indirect:** These feel the effects of the technology.
- **Special populations:** A wide range of categories that includes those who cannot use the technology or choose not to use it, and organizations that

may be interested in its deployment like advocacy groups or the government.

There are a range of harms that can be felt or perpetuated onto each group of stakeholder. Types of stakeholders can include end users of the product, administrators, internal teams, advocacy groups, manufacturers, someone who might be excluded from using the product. Stakeholders can also include the makers who propose and work on the technology, as well as the internal project sponsors, that is, the managers and executives who are supportive of the project.

4. Moderator selection

The moderator describes scenarios for gameplay, guides discussion, and collects comment cards at the end. They will also give an overview of the ethical principles considered in the deck.

5. Presentation of a product

The moderator introduces a product, real or imaginary, for the players to role play. They can start with questions about the product with prompts such as:

- Who will use the product?
- What features will it have?
- When will it be used?
- Where will it be used?
- Why is it useful?
- How will it be used?

6. Writing reviews

Players each write a review of the product or scenario based on the cards they've been dealt. Players take turns reading their reviews aloud. Listen carefully for any issues or themes that might come up. Take note of any changes you might make to your product based on insights discussed.

- The stakeholder card tells them who they are for the round.
- The ethical principle card is one of the six Microsoft AI principles; it tells them what area they're focusing on in their review.
- The rating card tells them how much their stakeholder likes the technology (1, 3, or 5 stars).

The player then writes a review of the technology, from the perspective of their stakeholder, of why they like or dislike the product from the perspective of the ethical principle they received.

Discussion

The moderator has each player read their reviews. Everyone is invited to discuss the different perspectives and other considerations that may not have come up.

Potential moderator questions include:

- Who is the most impacted?
- What features are problematic?
- What are the potential harms?

Harms mitigation

Finally, the group picks a thread from the discussion to begin identifying design or process changes that can mitigate a particular harm. At the end of each round, the decks are shuffled, and another round can begin with the same or a different scenario.

Next steps

Once you have enough understanding of potential harms or negative impact your product or scenario may cause, proceed to learn [how to model these harms](#) so you can devise effective mitigations.

Foundations of assessing harm

Article • 05/06/2022

Harms Modeling is a practice designed to help you anticipate the potential for harm, identify gaps in product that could put people at risk, and ultimately create approaches that proactively address harm.

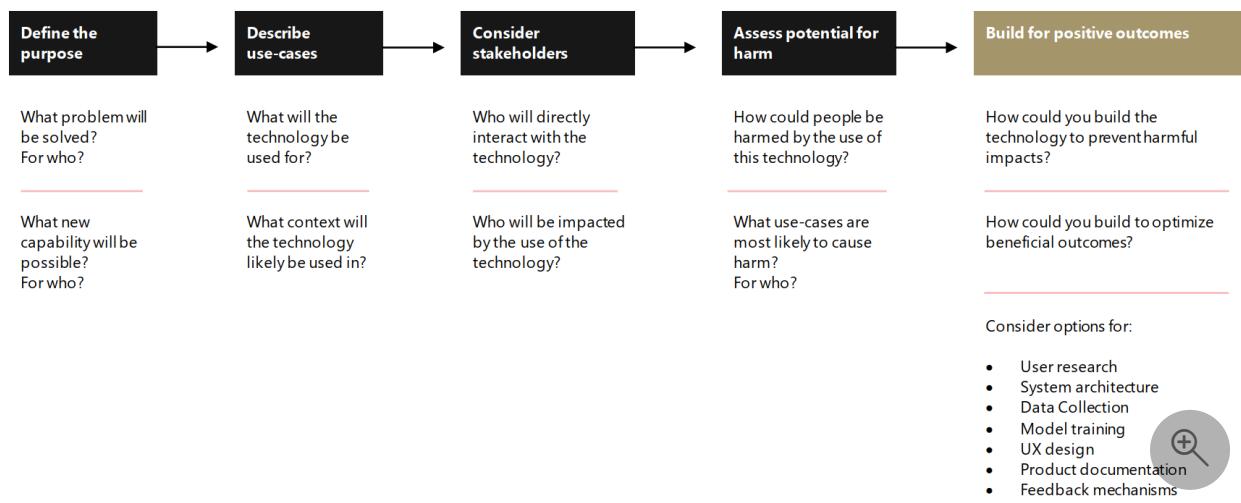
Why harms modeling?

As technology builders, your work is global. Designing AI to be trustworthy requires creating solutions that reflect ethical principles deeply rooted in important and timeless values. In the process of designing and building your technologies, it is essential to evaluate not only ideal outcomes, but possible negative ones as well.

Technology and human rights

It is as important as ever to be aware of how digital technology could impact human rights. In addition to continuing to protect privacy and security, we must address the risks of AI and other emerging technologies, such as facial recognition. History teaches us that human rights violations not only result from the nefarious use of technology, but also from a lack of awareness amongst those who have good intentions. As a part of our company's dedication to the protection of human rights, Microsoft forged a partnership with important stakeholders outside of our industry, including the United Nations (UN).

An important set of UN principles that our company firmly supports, which was ratified by over 250 nations, is the Universal Declaration of Human Rights (UDHR). The UDHR is a milestone document. Drafted by a diverse global group of legal and cultural experts, the Declaration was proclaimed by the United Nations General Assembly in 1948 as a common standard of achievements for all peoples and all nations. It sets out, for the first time, fundamental human rights to be universally protected. It has been translated into over 500 languages. Additionally, Microsoft is one of 4,700 corporate signatories to the UN Global Compact, an international business initiative designed to promote responsible corporate citizenship.



[Download in Microsoft Word](#)

Human understanding

In addition to appreciating the importance of human rights, building trustworthy systems requires us to consider many people's perspectives. Asking who the stakeholders are, what they value, how they could benefit, and how they could be hurt by our technology, is a powerful step that allows us to design and build better products.

Who does the technology impact?

Who are the customers?

- What do they value?
- How should they benefit?
- How could tech harm them?

Who are the non-customer stakeholders?

- What do they value?
- How should they benefit?
- How could tech harm them?

Asking these questions is a practice in [Value Sensitive Design](#) and is the beginning to better understanding what is important to stakeholders, and how it plays into their relationship with the product.

Types of stakeholders

Project sponsors

Backers, decision makers, and owners make up this category. Their values are articulated in the project strategy and goals.

Tech builders

Designers, developers, project managers, and people working directly on designing systems make up this group. They bring their own ethical standards and profession-specific values to the system.

Direct & indirect stakeholders

These stakeholders are significantly impacted by the system. Those who are impacted include end users, software staff, clients, bystanders, interfacing institutions, and even past or future generations. Non-human factors such as places, for example, historic buildings or sacred spaces, may also be included.

Marginalized populations

This category is made up of the population frequently considered a minority, vulnerable, or stigmatized. This category includes children, the elderly, members of the LGBTQ+ community, ethnic minorities, and other populations who often experience unique and disproportionate consequences.

Assessing Harm

Once you have defined the technology purpose, use cases, and stakeholders, conduct a Harms Modeling exercise to evaluate potential ways the use of a technology you are building could result in negative outcomes for people and society.

CATEGORY	TYPE OF HARM	Severity	Scale	Probability	Frequency	POTENTIAL
Risk of injury	Physical or infrastructure damage	■	■	■	■	LOW
	Emotional or psychological distress	▲	■	▲	▲	HIGH
Denial of consequential services	Opportunity loss	■	■	■	■	LOW
	Economic loss	■	■	■	■	LOW
Infringement on human rights	Dignity loss	■	■	■	■	MODERATE
	Liberty loss	■	■	■	■	LOW
	Privacy loss	▲	■	▲	▲	HIGH
	Environmental impact	■	■	■	■	LOW
Erosion of social & democratic structures	Manipulation	▲	■	■	▲	HIGH
	Social detriment	■	■	■	■	MODERATE

NOTE: This summary represents the outcome of a qualitative assessment and is used to inform prioritization of responsible innovation mitigations.



[Download in Microsoft Word](#)

The diagram above is an example of a harms evaluation. This model is a qualitative approach that's used to understand the potential magnitude of harm.

You can complete this ideation activity individually, but ideally it is conducted as collaboration between developers, data scientists, designers, user researcher, business decision-makers, and other disciplines that are involved in building the technology.

Suggestions for harm description statements:

- Intended use: If [feature] was used for [use case], then [stakeholder] could experience [harm description].
- Unintended use: If [user] tried to use [feature] for [use case], then [stakeholder] could experience [harm description].
- System error: If [feature] failed to function properly when used for [use case], then [stakeholder] could experience [harm description].
- Misuse: [Malicious actor] could potentially use [feature], to cause [harm description] to [stakeholder].

Use the categories, questions, and examples described in the [Types of harm](#) to generate specific ideas for how harm could occur. The article lists categories of harms, that are based upon common negative impact areas. Adapt and adopt additional categories that are relevant to you.

Next Steps

Read [Types of harm](#) for further harms analysis.

Types of harm

Article • 01/24/2023

This article creates awareness of the different types of harm, so that appropriate mitigation steps can be implemented.

Risk of injury

Physical injury

Consider how technology could hurt people or create dangerous environments.

Harm	Description	Consideration(s)	Example
Overreliance on safety features	This points out the dependence on technology to make decisions without adequate human oversight.	How might people rely on this technology to keep them safe? How could this technology reduce appropriate human oversight?	A healthcare agent could misdiagnose illness, leading to unnecessary treatment.
Inadequate fail-safes	Real-world testing may insufficiently consider a diverse set of users and scenarios.	If this technology fails or is misused, how would people be impacted? At what point could a human intervene? Are there alternative uses that have not been tested for? How would a system failure impact users?	If an automatic door failed to detect a wheelchair during an emergency evacuation, a person could be trapped if there isn't an accessible override button.
Exposure to unhealthy agents	Manufacturing, as well as disposal of technology, can jeopardize the health and well-being of workers and nearby inhabitants.	What negative outcomes could come from the manufacturing of your components or devices?	Inadequate safety measures could expose workers to toxins during digital component manufacturing.

Emotional or psychological injury

Misused technology can lead to severe emotional and psychological distress.

Harm	Description	Consideration(s)	Example
Overreliance on automation	Misguided beliefs can lead users to trust the reliability of a digital agent over that of a human.	How could this technology reduce direct interpersonal feedback? How might this technology interface with trusted sources of information? How could sole dependence on an artificial agent impact a person?	A chatbot could be relied upon for relationship advice or mental health counseling instead of a trained professional.
Distortion of reality or gaslighting	When intentionally misused, technology can undermine trust and distort someone's sense of reality.	Could this be used to modify digital media or physical environments?	An IoT device could enable monitoring and controlling of an ex-intimate partner from afar.
Reduced self-esteem/reputation damage	Some shared content can be harmful, false, misleading, or denigrating.	How could this technology be used to share personal information inappropriately? How could it be manipulated to misuse information and misrepresent people?	Synthetic media "revenge porn" can swap faces, creating the illusion of a person participating in a video who did not.
Addiction/attention hijacking	Technology could be designed for prolonged interaction, without regard for well-being.	In what ways might this technology reward or encourage continued interaction beyond delivering user value?	Variable drop rates in video game loot boxes could cause players to keep playing and neglect self-care.
Identity theft	Identity theft may lead to loss of control over personal credentials, reputation, and representation.	How might an individual be impersonated with this technology? How might this technology mistakenly recognize the wrong individual as an authentic user?	Synthetic voice font could mimic the sound of a person's voice and be used to access a bank account.

Harm	Description	Consideration(s)	Example
Misattribution	This includes crediting a person with an action or content they are not responsible for.	In what ways might this technology attribute an action to an individual or group? How could someone be affected if an action was incorrectly attributed to them?	Facial recognition can misidentify an individual during a police investigation.

Denial of consequential services

Opportunity loss

Automated decisions could limit access to resources, services, and opportunities essential to well-being.

Harm	Description	Consideration(s)	Example
Employment discrimination	Some people may be denied access to apply for or secure a job based on characteristics unrelated to merit.	Are there ways in which this technology could impact recommendations or decisions related to employment?	Hiring AI could recommend fewer candidates with female-sounding names for interviews.
Housing discrimination	This includes denying people access to housing or the ability to apply for housing.	How could this technology impact recommendations or decisions related to housing?	Public housing queuing algorithm could cause people with international-sounding names to wait longer for vouchers.

Harm	Description	Consideration(s)	Example
Insurance and benefit discrimination	This includes denying people insurance, social assistance, or access to a medical trial due to biased standards.	Could this technology be used to determine access, cost, allocation of insurance or social benefits?	Insurance company might charge higher rates for drivers working night shifts due to algorithmic predictions suggesting increased drunk driving risk.
Educational discrimination	Access to education may be denied because of an unchangeable characteristic.	How might this technology be used to determine access, cost, accommodations, or other outcomes related to education?	Emotion classifier could incorrectly report that students of color are less engaged than their white counterparts, leading to lower grades.
Digital divide/technological discrimination	Disproportionate access to the benefits of technology may leave some people less informed or equipped to participate in society.	What prerequisite skills, equipment, or connectivity are necessary to get the most out of this technology? What might be the impact of select people gaining earlier access to this technology than others, in terms of equipment, connectivity, or other product functionality?	Content throttling could prevent rural students from accessing classroom instruction video feeds.
Loss of choice/network and filter bubble	Presenting people with only information that conforms to and reinforces their beliefs.	How might this technology affect the choices and information available to people? What past behaviors or preferences might this technology rely on to predict future behaviors or preferences?	News feed could only present information that confirms existing beliefs.

Economic loss

Automating decisions related to financial instruments, economic opportunity, and resources can amplify existing societal inequities and obstruct well-being.

Harm	Description	Consideration(s)	Example
Credit discrimination	People may be denied access to financial instruments based on characteristics unrelated to economic merit.	How might this technology rely on existing credit structures to make decisions? How might this technology affect the ability of an individual or group to obtain or maintain a credit score?	Higher introductory rate offers could be sent only to homes in lower socioeconomic postal codes.
Differential pricing of goods and services	Goods or services may be offered at different prices for reasons unrelated to the cost of production or delivery.	How could this technology be used to determine the pricing of goods or services? What are the criteria for determining the cost to people for the use of this tech?	More could be charged for products based on designation for men or women.
Economic exploitation	People might be compelled or misled to work on something that impacts their dignity or well-being.	What role did human labor play in producing training data for this technology? How was this workforce acquired? What role does human labor play in supporting this technology? Where is this workforce expected to come from?	Paying financially destitute people for their biometric data to train AI systems.
Devaluation of individual expertise	Technology may supplant the use of paid human expertise or labor.	How might this technology impact the need to employ an existing workforce?	AI agents replace doctors/radiographers for evaluation of medical imaging.

Infringement on human rights

Dignity loss

Technology can influence how people perceive the world and recognize, engage, and value one another. The exchange of honor and respect between people can be

interfered with.

Harm	Description	Consideration(s)	Example
Dehumanization	Removing, reducing, or obscuring the visibility of a person's humanity.	How might this technology be used to simplify or abstract the way a person is represented? How might this technology reduce the distinction between humans and the digital world?	Entity recognition and virtual overlays in drone surveillance could reduce the perceived accountability of human actions.
Public shaming	This may mean exposing people's private, sensitive, or socially inappropriate material.	How might movements or actions be revealed through data aggregation?	A fitness app could reveal a user's GPS location on social media, indicating attendance at an Alcoholics Anonymous meeting.

Liberty loss

Automating legal, judicial, and social systems can reinforce biases and lead to detrimental consequences.

Harm	Description	Consideration(s)	Example
Predictive policing	Inferring suspicious behavior or criminal intent based on historical records.	How could this support or replace human policing or criminal justice decision-making?	An algorithm can predict several area arrests, so police make sure they match or exceed that number.
Social control	Conformity may be reinforced or encouraged by publicly designating human behaviors as positive or negative.	What types of personal or behavioral data might feed this technology? How would it be obtained? What outputs would be derived from this data? Is this technology likely to be used to encourage or discourage certain behaviors?	Authoritarian government uses social media and e-commerce data to determine a "trustworthy" score based on where people shop and whom they spend time with.

Harm	Description	Consideration(s)	Example
Loss of effective remedy	This means an inability to explain the rationale or lack of opportunity to contest a decision.	How might people understand the reasoning for decisions made by this technology? How might an individual relying on this technology explain its decisions? How could people contest or question a decision this technology makes?	Automated prison sentence or pre-trial release decision is not explained to the accused.

Privacy loss

The information generated by our use of technology can be used to determine facts or make assumptions about someone without their knowledge.

Harm	Description	Consideration(s)	Example
Interference with private life	Revealing information a person has not chosen to share.	How could this technology use information to infer portions of private life? How could decisions based upon these inferences expose things that a person does not want made public?	Task-tracking AI could monitor personal patterns from which it infers an extramarital affair.
Forced association	Requiring participation in the use of technology or surveillance to take part in society.	How might the use of this technology be required for participation in society or organization membership?	Biometric enrollment in a company's meeting room transcription AI is a stipulated requirement in the job offer letter.
Inability to freely and fully develop personality	This may mean restriction of one's ability to truthfully express themselves or explore external avenues for self-development.	In what way does the system or product ascribe positive vs. negative connotations toward particular personality traits? How can using the product or system reveal information to entities such as the government or employer that inhibits free expression?	Intelligent meeting system could record all discussions between colleagues, including personal coaching and mentorship sessions.

Harm	Description	Consideration(s)	Example
Never forgiven	Digital files or records may never be deleted.	What and where is data being stored from this product, and who can access it? How long is user data stored after technology interaction? How is user data updated or deleted?	A teenager's social media history could remain searchable long after they have outgrown the platform.
Loss of freedom of movement or assembly	This means an inability to navigate the physical or virtual world with desired anonymity.	In what ways might this technology monitor people across physical and virtual space?	A real name could be required to sign up for a video game enabling real-world stalking.

Environmental impact

The environment can be impacted by every decision in a system or product life cycle, from the amount of cloud computing needed to retail packaging. Environmental changes can affect entire communities.

Harm	Description	Consideration(s)	Example
Exploitation or depletion of resources	Obtaining the raw materials for technology, including how it's powered, leads to negative consequences to the environment and its inhabitants.	What materials are needed to build or run this technology? What energy requirements are needed to build or run this technology?	A local community could be displaced due to harvesting rare earth minerals and metals required for some electronic manufacturing.
Electronic waste	Reduced quality of collective well-being because of the inability to repair, recycle, or otherwise responsibly dispose of electronics.	How might this technology reduce electronic waste by recycling materials or allowing users to self-repair? How might this technology contribute to electronic waste when new versions are released or when current/past versions stop working?	Toxic materials inside discarded electronic devices could leach into the water supply, making local populations ill.

Harm	Description	Consideration(s)	Example
Carbon emissions	Running cloud solutions that are not optimized can lead to unnecessary carbon emissions and electricity waste, causing harm to the climate.	Do you have insights into how optimized your cloud workloads and solutions are? What impact does your solution have on the climate, and does it differ based on the region where you deploy your workloads?	Running solutions that are not optimized or properly designed for cloud efficiency can lead to a heavier toll on the climate, causing unnecessary carbon emissions and electricity waste.

Erosion of social & democratic structures

Manipulation

Technology's ability to create highly personalized and manipulative experiences can undermine an informed citizenry and trust in societal structures.

Harm	Description	Consideration(s)	Example
Misinformation	Disguising fake information as legitimate or credible information.	How might this technology be used to generate misinformation? How could it be used to spread credible misinformation?	Generation of synthetic speech of a political leader sways an election.
Behavioral exploitation	This means exploiting personal preferences or patterns of behavior to induce a desired reaction.	How might this technology be used to observe behavior patterns? How could this technology be used to encourage dysfunctional or maladaptive behaviors?	Monitoring shopping habits in the connected retail environment leads to personalized incentives for impulse shoppers and hoarders.

Social detriment

At scale, the way technology impacts people shape social and economic structures within communities. It can further ingrain elements that include or benefit some while excluding others.

Harm	Description	Consideration(s)	Example
------	-------------	------------------	---------

Harm	Description	Consideration(s)	Example
Amplification of power inequality	This may perpetuate existing class or privilege disparities.	How might this technology be used in contexts where there are existing social, economic or class disparities? How might people with more power or privilege disproportionately influence the technology?	Requiring a residential address & phone number to register on a job website could prevent a homeless person from applying.
Stereotype reinforcement	This may perpetuate uninformed "conventional wisdom" about historically or statistically underrepresented people.	How might this technology be used to reinforce or amplify existing social norms or cultural stereotypes? How might the data used by this technology cause it to reflect biases or stereotypes?	Results of an image search for "CEO" could primarily show photos of Caucasian men.
Loss of individuality	This may be an inability to express a unique perspective.	How might this technology amplify majority opinions or "group-think"? Conversely, how might unique forms of expression be suppressed? In what ways might the data gathered by this technology be used in feedback to people?	Limited customization options in designing a video game avatar inhibit self-expression of a player's diversity.
Loss of representation	Broad categories of generalization obscure, diminish, or erase real identities.	How could this technology constrain identity options? Could it be used to label or categorize people automatically?	Automatic photo caption assigns incorrect gender identity and age to the subject.
Skill degradation and complacency	Overreliance on automation lead to atrophy of manual skills.	In what ways might this technology reduce the accessibility and ability to use manual controls?	Overreliance on automation could lead to an inability to gauge the airplane's true orientation because the pilots have been trained to rely on instruments only.

Evaluate harms

Once you've generated a broad list of potential harms, you should complete your Harms Model by evaluating the potential magnitude for each harm category. This will allow you to prioritize your areas of focus. See the following example harms model for reference:

Contributing factor	Definition
Severity	How acutely could the technology impact an individual or group's well-being?
Scale	How broadly could the impact on well-being be experienced across populations or groups?
Probability	How likely is the technology to impact individual or group's well-being?
Frequency	How often would an individual or group experience an impact on their well-being from the technology?

Next Steps

Use the Harms Model you developed to guide your product development work:

- Seek more information from stakeholders that you identified as potentially experiencing harm.
- Develop and validate the hypothesis for addressing the areas you identified as having the highest potential for harm.
- Integrate the insights into your decisions throughout the technology development process: data collection and model training, system architecture, user experience design, product documentation, feedback loops, and communication capabilities and limitations of the technology.
- Explore [Community Jury](#).
- Assess and mitigate unfairness using Azure Machine Learning and the open-source [FairLearn package](#).

Other Responsible AI tools:

- [Responsible AI resource center](#)
- [Guidelines for Human AI interaction](#)
- [Conversational AI guidelines](#)
- [Inclusive Design guidelines](#)
- [AI Fairness checklist](#)

Additional references:

- [Downloadable booklet for assessing harms](#)

- Value Sensitive Design ↗

Community jury

Article • 12/29/2022

Community jury, an adaptation of the [citizen jury](#), is a technique where diverse stakeholders impacted by a technology are provided an opportunity to learn about a project, deliberate together, and give feedback on use cases and product design. This technique allows project teams to understand the perceptions and concerns of impacted stakeholders for effective collaboration.

A community jury is different from a focus group or market research; it allows the impacted stakeholders to hear directly from the subject matter experts in the product team, and cocreate collaborative solutions to challenging problems with them.

Benefits

This section discusses some important benefits of community jury.

Expert testimony

Members of a community jury hear details about the technology under consideration, directly from experts on the product team. These experts help highlight the capabilities in particular applications.

Proximity

A community jury allows decision-makers to hear directly from the community, and to learn about their values, concerns, and ideas regarding a particular issue or problem. It also provides a valuable opportunity to better understand the reasons for their conclusions.

Consensus

By bringing impacted individuals together and providing collaborative solutions and an opportunity for them to learn and discuss key aspects of the technology, a community jury can identify areas of agreement and build common ground solutions to challenging problems.

Community jury contributors

Product team

This group comprises owners who will bring the product to market, representative project managers, engineers, designers, data scientists, and others involved in the making of a product. Additionally, subject matter experts are included who can answer in-depth questions about the product.

Preparation

Effective deliberation and cocreation require ready-to-use answers to technical questions. As product owners, it is important to ensure technical details are collected prior to the start of the jury session.

Relevant artifacts could include:

- Documentation of existing protections, planned or in place.
- Data Flows, for example, what data types collected, who will have access, for how long, with what protections, and so on.
- Mockups or prototypes of proposed stakeholder collaborative solutions.

During the session

Along with providing an accessible presentation of technical details, product team witnesses should describe certain applications of the technology.

Relevant artifacts could include:

- Customer benefits
- Harms assessment and socio-technical impact
- Storyboards
- Competitive solutions and analyses
- Academic or popular media reports

Moderator

Bring on a neutral user researcher to ensure everyone is heard, avoiding domination of the communications by any one member. The moderator will facilitate brainstorms and deliberations, as well as educate jury members in uncovering bias, and ways to ask difficult questions. If a user researcher is not available, choose a moderator who is skilled at facilitating group discussions. Following the session, the moderator is responsible for the following:

- ensure that the agreed-upon compensation is provided to the jury members;
- produce a report that describes key insights, concerns, and recommendations, and
- share key insights and next steps with the jury, and thank them for their participation.

Preparation

- Structure the sessions so that there is ample time for learning, deliberation, and cocreation. This could mean having multiple sessions that go in-depth on different topics or having longer sessions.
- Pilot the jury with a smaller sample of community members to work out the procedural and content issues prior to the actual sessions.

During the session

- Ensure that all perspectives are heard, including those of the project team and those of the jury members. This minimizes group-thinking as well as one or two individuals dominating the discussion.
- Reinforce the value of the juror participation by communicating the plan for integrating jury feedback into the product planning process. Ensure that the project team is prepared to hear criticisms from the jury.

Jury members

These are direct and indirect stakeholders impacted by the technology, representative of the diverse community in which the technology will be deployed.

Sample size

A jury should be large enough to represent the diversity and collective will of the community, but not so large that the voices of quieter jurors are drowned out. It is ideal to get feedback from at least 16-20 individuals total who meet the criteria below. You can split the community jury over multiple sessions so that no more than 8-10 individuals are part of one session.

Recruitment criteria

Stratify a randomly selected participant pool so that the jury includes the demographic diversity of the community. Based on the project objectives, relevant recruitment criteria

may include balancing across gender identity, age, [privacy index](#). Recruitment should follow good user research recruitment procedures and reach a wider community.

Session structure

Sessions typically last 2-3 hours. Add more or longer deep dive sessions, as needed, if aspects of the project require more learning, deliberation, and cocreation.

1. **Overview, introduction, and Q&A:** The moderator provides a session overview, then introduces the project team and explains the product's purpose, along with potential use cases, benefits, and harms. Questions are then accepted from community members. This session should be between 30 to 60 minutes long.
2. **Discussion of key themes:** Jury members ask in-depth questions about aspects of the project, fielded by the moderator. This session should also be between 30 to 60 minutes in length.
3. This step can be any one of the following options:
 - **Deliberation and cocreation:** This option is preferable. Jury members deliberate and co-create effective collaboration solutions with the project team. This is typically 30 to 60 minutes long.
 - **Individual anonymous survey:** Alternatively, conduct an anonymous survey of jury members. Anonymity may allow issues to be raised that would not otherwise be expressed. Use this 30-minute session if there are time constraints.
4. **Following the session:** The moderator produces a study report that describes key insights, concerns, and potential solutions to the concerns.

If the values of different stakeholders were in conflict with each other during the session and the value tensions were left unresolved, the product team would need to brainstorm solutions, and conduct a follow-up session with the jury to determine if the solutions adequately resolve their concerns.

Tips to run a successful, effective, and collaborative jury

- Ensure alignment of goals and outcomes with the project team before planning begins, including deliverables, recruitment strategy, and timeline. Consider including additional subject-matter experts relevant to the project, to address open questions/concerns.

- The consent to audio and video recording of the jury should follow your company's standard procedures for non-disclosure and consent that is obtained from participants during user research studies.
- Provide fair compensation to participants for the time they devote to participation. Whenever possible, participants should also be reimbursed for costs incurred as a result of study participation, for example, parking and transportation costs. Experienced user research recruiters can help determine fair gratuities for various participant profiles.
- Ensure that all perspectives are heard, minimizing group-thinking as well as one or two individuals dominating the discussion. This should include those of the project team and the jury members.
- Structure the sessions so that there is ample time for learning, deliberation, and cocreation. This could mean having multiple sessions going in-depth on different topics or having longer sessions.
- Pilot the jury with a smaller sample of community members to work out the procedural and content issues prior to the actual sessions.
- If the topic being discussed is related to personal data privacy, aim to balance the composition of the community jury to include individuals with different [privacy indices](#).
- Consider including session breaks and providing snacks/refreshments for sessions that are two hours or longer.
- Reinforce the value of the juror participation by communicating the plan for integrating jury feedback into the product planning process. Also ensure that the project team is prepared to hear criticisms from the jury.
- After the jury, in addition to publishing the research report, send out a thank you note to the volunteer participants of the jury, along with an executive summary of the key insights.

Additional information

Privacy index

The [Privacy Index](#) is an approximate measure for an individual's concern about personal data privacy, and is gauged using the following:

1. Consumers have lost all control over how personal information is collected and used by companies.
2. Most businesses handle the personal information they collect about consumers in a proper and confidential way.

3. Existing laws and organizational practices provide a reasonable level of protection for consumer privacy today.

Participants are asked to provide responses to the above concerns using the scale of: 1 - Strongly Agree, 2 - Somewhat Agree, 3 - Somewhat Disagree, 4- Strongly Disagree, and classified into the categories below.

High/Fundamentalist => IF A = 1 or 2 AND B & C = 3 or 4

Low/unconcerned => IF A = 3 or 4 AND B & C = 1 or 2

Medium/pragmatist => All other responses

Next steps

Explore the following references for detailed information on this method:

- Jefferson center: creator of the Citizen's Jury method <https://jefferson-center.org/about-us/how-we-work/>
- Case study: [Connected Health Cities \(UK\)](#)
- Case study: [Community Jury at Microsoft](#)

Responsible AI

Article • 02/28/2023

Driven by ethical principles that put people first, Microsoft is committed to advancing AI. We want to partner with you to support this endeavor.

Responsible AI principles

As you implement AI solutions, consider the following principles in your solution:

- **Fairness:** AI systems should treat all people fairly.
- **Reliability and safety:** AI systems should perform reliably and safely.
- **Privacy and security:** AI systems should be secure and respect privacy.
- **Inclusiveness:** AI systems should empower everyone and engage people.
- **Transparency:** AI systems should be understandable.
- **Accountability:** People should be accountable for AI systems.

Establish a responsible AI strategy

Learn how to develop your own responsible AI strategy and principles based on the values of your organization.

- [Get started at AI Business School ↗](#)

Guidelines to develop AI responsibly

Put responsible AI into practice with these guidelines, designed to help you anticipate and address potential issues throughout the software development lifecycle.

- [Human-AI interaction guidelines ↗](#)
- [Conversational AI guidelines ↗](#)
- [Inclusive design guidelines ↗](#)
- [AI fairness checklist ↗](#)
- [Datasheets for Datasets template ↗](#)
- [AI security engineering guidance ↗](#)

Tools for responsible AI

Tools are available to help developers and data scientists understand, protect, and control AI systems. These tools can come from a variety of sources, including Azure Machine Learning, open-source projects, and research.

- **Understand:** AI systems can behave unexpectedly for a variety of reasons. Software tools can help you understand the behavior of your AI systems so that you can better tailor them to your needs. Examples of this type of tool include [InterpretML](#), [Error Analysis](#) and [Fairlearn](#).
- **Protect:** AI systems rely on data. Software tools can help you protect that data by preserving privacy and ensuring confidentiality. Examples of this type of tool include [Confidential Computing for Machine Learning](#), [SmartNoise differential privacy toolkit](#), [SEAL Homomorphic Encryption toolkit](#), and the [Presidio data de-identification toolkit](#).
- **Control:** Responsible AI needs governance and control through the development cycle. Azure Machine Learning enables an audit trail for better traceability, lineage, and control to meet regulatory requirements. Examples include audit trail and traceability.

Next steps

For more information about responsible solution development, visit:

- [Responsible AI overview](#)
- [Responsible AI resources](#)
- [Responsible bots: 10 guidelines for developers of conversational AI](#)

Responsible and trusted AI

Article • 07/28/2023

Microsoft outlines six key principles for responsible AI: accountability, inclusiveness, reliability and safety, fairness, transparency, and privacy and security. These principles are essential to creating responsible and trustworthy AI as it moves into mainstream products and services. They're guided by two perspectives: ethical and explainable.



The principles of responsible AI



Ethical

From an ethical perspective, AI should:

- Be fair and inclusive in its assertions.
- Be accountable for its decisions.
- Not discriminate or hinder different races, disabilities, or backgrounds.

In 2017, Microsoft established an advisory committee for AI, ethics, and effects in engineering and research ([Aether](#)). The core responsibility of the committee is to advise on issues, technologies, processes, and best practices for responsible AI. To learn more, see [Understanding the Microsoft governance model - Aether + Office of Responsible AI](#).

Accountability

Accountability is an essential pillar of responsible AI. The people who design and deploy an AI system need to be accountable for its actions and decisions, especially as we

progress toward more autonomous systems.

Organizations should consider establishing an internal review body that provides oversight, insights, and guidance about developing and deploying AI systems. This guidance might vary depending on the company and region, and it should reflect an organization's AI journey.

Inclusiveness

Inclusiveness mandates that AI should consider all human races and experiences. Inclusive design practices can help developers understand and address potential barriers that could unintentionally exclude people. Where possible, organizations should use speech-to-text, text-to-speech, and visual recognition technology to empower people who have hearing, visual, and other impairments.

Reliability and safety

For AI systems to be trusted, they need to be reliable and safe. It's important for a system to perform as it was originally designed and to respond safely to new situations. Its inherent resilience should resist intended or unintended manipulation.

An organization should establish rigorous testing and validation for operating conditions to ensure that the system responds safely to edge cases. It should integrate A/B testing and champion/challenger methods into the evaluation process.

An AI system's performance can degrade over time. An organization needs to establish a robust monitoring and model-tracking process to reactively and proactively measure the model's performance (and retrain it for modernization, as necessary).

Explainable

Explainability helps data scientists, auditors, and business decision makers ensure that AI systems can justify their decisions and how they reach their conclusions. Explainability also helps ensure compliance with company policies, industry standards, and government regulations.

A data scientist should be able to explain to a stakeholder how they achieved certain levels of accuracy and what influenced the outcome. Likewise, to comply with the company's policies, an auditor needs a tool that validates the model. A business decision maker needs to gain trust by providing a transparent model.

Explainability tools

Microsoft has developed [InterpretML](#), an open-source toolkit that helps organizations achieve model explainability. It supports glass-box and black-box models:

- Glass-box models are interpretable because of their structure. For these models, Explainable Boosting Machine (EBM) provides the state of the algorithm based on a decision tree or linear models. EBM provides lossless explanations and is editable by domain experts.
- Black-box models are more challenging to interpret because of a complex internal structure, the neural network. Explainers like local interpretable model-agnostic explanations (LIME) or SHapley Additive exPlanations (SHAP) interpret these models by analyzing the relationship between the input and output.

[Fairlearn](#) is an Azure Machine Learning integration and an open-source toolkit for the SDK and the AutoML graphical user interface. It uses explainers to understand what mainly influences the model, and it uses domain experts to validate these influences.

To learn more about explainability, explore [model interpretability in Azure Machine Learning](#).

Fairness

Fairness is a core ethical principle that all humans aim to understand and apply. This principle is even more important when AI systems are being developed. Key checks and balances need to make sure that the system's decisions don't discriminate against, or express a bias toward, a group or individual based on gender, race, sexual orientation, or religion.

Microsoft provides an [AI fairness checklist](#) that offers guidance and solutions for AI systems. These solutions are loosely categorized into five stages: envision, prototype, build, launch, and evolve. Each stage lists recommended due-diligence activities that help minimize the impact of unfairness in the system.

Fairlearn integrates with Azure Machine Learning and supports data scientists and developers to assess and improve the fairness of their AI systems. It provides unfairness-mitigation algorithms and an interactive dashboard that visualizes the fairness of the model. An organization should use the toolkit and closely assess the fairness of the model while it's being built. This activity should be an integral part of the data science process.

Learn how to [mitigate unfairness in machine learning models](#).

Transparency

Achieving transparency helps the team understand:

- The data and algorithms that were used to train the model.
- The transformation logic that was applied to the data.
- The final model that was generated.
- The model's associated assets.

This information offers insights about how the model was created, so the team can reproduce it in a transparent way. Snapshots within [Azure Machine Learning workspaces](#) support transparency by recording or retraining all training-related assets and metrics involved in the experiment.

Privacy and security

A data holder is obligated to protect the data in an AI system. Privacy and security are an integral part of this system.

Personal data needs to be secured, and access to it shouldn't compromise an individual's privacy. [Azure differential privacy](#) helps protect and preserve privacy by randomizing data and adding noise to conceal personal information from data scientists.

Human AI guidelines

Human AI design guidelines consist of 18 principles that occur over four periods: initially, during interaction, when wrong, and over time. These principles help an organization produce a more inclusive and human-centric AI system.

Initially

- **Clarify what the system can do.** If the AI system uses or generates metrics, it's important to show them all and how they're tracked.
- **Clarify how well the system can do what it does.** Help users understand that AI isn't completely accurate. Set expectations for when the AI system might make mistakes.

During interaction

- **Show contextually relevant information.** Provide visual information related to the user's current context and environment, such as nearby hotels. Return details close to the target destination and date.
- **Mitigate social biases.** Make sure that the language and behavior don't introduce unintended stereotypes or biases. For example, an autocomplete feature needs to be inclusive of gender identity.

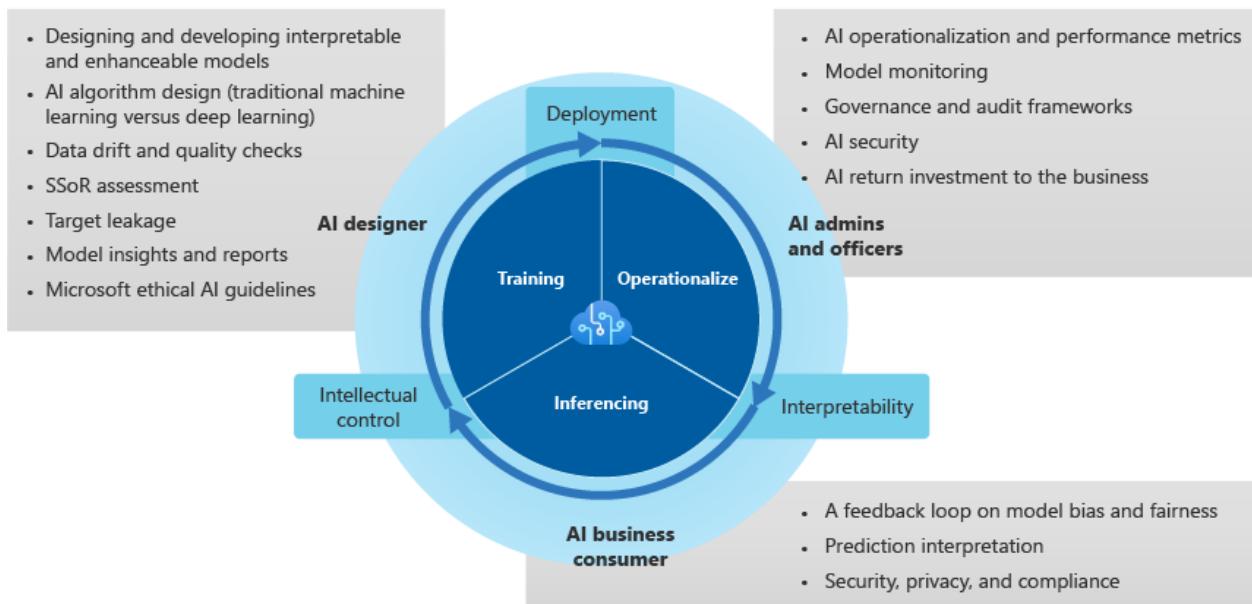
When wrong

- **Support efficient dismissal.** Provide an easy mechanism to ignore or dismiss undesirable features or services.
- **Support efficient correction.** Provide an intuitive way of making it easier to edit, refine, or recover.
- **Make clear why the system did what it did.** Optimize explainable AI to offer insights about the AI system's assertions.

Over time

- **Remember recent interactions.** Retain a history of interactions for future reference.
- **Learn from user behavior.** Personalize the interaction based on the user's behavior.
- **Update and adapt cautiously.** Limit disruptive changes, and update based on the user's profile.
- **Encourage granular feedback.** Gather user feedback from their interactions with the AI system.

Trusted AI framework



AI designer

The AI designer builds the model and is responsible for:

- Data drift and quality checks. The designer detects outliers and performs data quality checks to identify missing values. The designer also standardizes distribution, scrutinizes data, and produces use case and project reports.
- Assessing data in the system's source to identify potential bias.
- Designing AI algorithms to minimize data biases. These efforts include discovering how binning, grouping, and normalization (especially in traditional machine learning models like tree-based ones) can eliminate minority groups from data. Categorical AI design reiterates data biases by grouping social, racial, and gender classes in industry verticals that rely on protected health information (PHI) and personal data.
- Optimizing monitoring and alerts to identify target leakage and strengthen the model's development.
- Establishing best practices for reporting and insights that offer a granular understanding of the model. The designer avoids black-box approaches that use feature or vector importance, Uniform Manifold Approximation and Projection (UMAP) clustering, Friedman's H-statistic, feature effects, and related techniques. Identification metrics help to define predictive influence, relationships, and dependencies between correlations in complex and modern datasets.

AI administrator and officers

The AI administrator and officers oversee AI, governance, and audit framework operations and performance metrics. They also oversee how AI security is implemented and the business's return on investment. Their tasks include:

- Monitoring a tracking dashboard that assists model monitoring and combines model metrics for production models. The dashboard focuses on accuracy, model degradation, data drift, deviation, and changes in speed/error of inference.
- Implementing flexible deployment and redeployment (preferably, through a REST API) that allows models to be implemented into open, agnostic architecture. The architecture integrates the model with business processes and generates value for feedback loops.
- Working toward building model governance and access to set boundaries and mitigate negative business and operational impact. Role-based access control (RBAC) standards determine security controls, which preserve restricted production environments and the IP.
- Using AI audit and compliance frameworks to track how models develop and change to uphold industry-specific standards. Interpretable and responsible AI is founded on explainability measures, concise features, model visualizations, and industry-vertical language.

AI business consumers

AI business consumers (business experts) close the feedback loop and provide input for the AI designer. Predictive decision-making and potential bias implications like fairness and ethical measures, privacy and compliance, and business efficiency help to evaluate AI systems. Here are some considerations for business consumers:

- Feedback loops belong to a business's ecosystem. Data that shows a model's bias, errors, prediction speed, and fairness establishes trust and balance between the AI designer, administrator, and officers. Human-centric assessment should gradually improve AI over time.

Minimizing AI learning from multidimensional, complex data can help prevent biased learning. This technique is called less-than-one-shot (LO-shot) learning.

- Using interpretability design and tools holds AI systems accountable for potential biases. Model bias and fairness issues should be flagged and fed to an alerting and anomaly detection system that learns from this behavior and automatically addresses biases.

- Each predictive value should be broken down into individual features or vectors by importance or impact. It should deliver thorough prediction explanations that can be exported into a business report for audit and compliance reviews, customer transparency, and business readiness.
- Due to increasing global security and privacy risks, best practices for resolving data violations during inference require complying with regulations in individual industry verticals. Examples include alerts about noncompliance with PHI and personal data, or alerts about violation of national/regional security laws.

Next steps

Explore [human AI guidelines](#) to learn more about responsible AI.

Introduction to guidelines for human-AI interaction

Article • 04/14/2023

This article provides background on 18 recommended guidelines for human-AI interaction design from Microsoft and how to apply them. Microsoft experts in the field of AI have drawn on years of research and thinking to develop this guidance.

Why do we need guidelines for human-AI interaction?

We need guidelines for human-AI interaction because AI systems may demonstrate unpredictable behaviors that can be disruptive, confusing, offensive, or even dangerous. For these reasons, AI systems often violate traditional human-computer interaction design principles.

When a traditional application or product doesn't behave consistently, it would be judged to have a design deficiency or bug. However, inconsistency and uncertainty are inherent in AI-infused systems because of their probabilistic nature and because they change over time as they learn with new data.

Attributes of AI services, including their accuracy, failure modes, and ability to be understood raise new challenges and opportunities for product, service, and application developers.

What are the guidelines?

Microsoft proposes 18 generally applicable design guidelines for human-AI interaction.

These guidelines synthesize more than two decades of thinking and research about how to make AI user-friendly. The [Microsoft team](#) ran them through three rounds of validation to ensure they are specific, observable, and easy to understand. You can read the [original research paper](#).

INITIALLY	1 Make clear what the system can do.	2 Make clear how well the system can do what it can do.					
DURING INTERACTION	3 Time services based on context.	4 Show contextually relevant information.	5 Match relevant social norms.	6 Mitigate social biases.			
WHEN WRONG	7 Support efficient invocation.	8 Support efficient dismissal.	9 Support efficient correction.	10 Scope services when in doubt.	11 Make clear why the system did what it did.		
OVER TIME	12 Remember recent interactions.	13 Learn from user behavior.	14 Update and adapt cautiously.	15 Encourage granular feedback.	16 Convey the consequences of user actions.	17 Provide global controls.	18 Notify users about changes.

 Microsoft

The 18 guidelines provide recommendations on how to create meaningful AI-infused experiences that leave users feeling in control and that respect their values, goals, and attention.

The guidelines are grouped into four categories, depending on the phase of user interaction to which they apply.

The initial phase

Upon initial exposure to the system, users should learn what to expect. What is this system capable of? And how well can it perform? Setting unreasonably high expectations can result in frustration and product abandonment, so it's important to communicate honestly what the product can do, and how well. Therefore, the guidelines in the first category are about setting expectations:

1. **Make clear what the system can do.** Help the user understand what the AI system is capable of doing.
2. **Make clear how well the system can do what it can do.** Help the user understand how often the AI system may make mistakes.

During interaction

This subset of guidelines is about context. Whether it's the larger social and cultural context or the local context of a user's setting, current task, and attention, AI systems make inferences about people and their needs that depend on context.

3. **Time services based on context.** Time when to act or interrupt based on the user's current task and environment.
4. **Show contextually relevant information.** Display information relevant to the user's current task and environment.
5. **Match relevant social norms.** Ensure the experience is delivered in a way that users would expect, given their social and cultural context.
6. **Mitigate social biases.** Ensure the AI system's language and behaviors don't reinforce undesirable and unfair stereotypes and biases.

Guidelines 5 and 6 in this group remind us to consider social norms and biases. Is the data set representative of the population? Is the model learning and replicating undesirable social biases? To apply these guidelines effectively, ensure your team has enough diversity to cover each other's blind spots.

When the system is wrong

AI-infused systems will inevitably be wrong, and you need to plan for it. The system might not trigger when expected or might trigger at the wrong time, so it should be easy to invoke and dismiss. When the system is wrong, it should be easy to correct it, and when it's uncertain, the user should be able to complete the task on their own. For example, the AI system can gracefully fade out or ask the user for clarification.

When the system is wrong, users are puzzled and frustrated, and understandably so. How can you prevent such feelings? Providing an explanation of why the system did what it did can help users understand how the system works, empathize with it, and reduce feelings of frustration. But to be able to do that, you need to build explainability into your system, and not work with an opaque box.

7. **Support efficient invocation.** Make it easy to invoke or request the AI system's services when needed.
8. **Support efficient dismissal.** Make it easy to dismiss or ignore undesired AI system services.
9. **Support efficient correction.** Make it easy to edit, refine, or recover when the AI system is wrong.

10. **Scope services when in doubt.** Engage in disambiguation or gracefully degrade the AI system's services when uncertain about a user's goals.
11. **Make clear why the system did what it did.** Enable the user to access an explanation of why the AI system behaved as it did.

Over time

AI systems learn and improve over time. An AI system should learn from user behavior. The guidelines in this group encourage users to teach the system, by providing granular feedback. As users provide feedback, convey the consequences of their actions by at least acknowledging the feedback was recorded, and perhaps indicating how it will affect their experience in the future.

However, as the system learns and improves, and the model updates, be cautious about introducing large disruptive changes to the user experience. Plan ahead how to roll out those changes gradually and notify users about them.

12. **Remember recent interactions.** Maintain short-term memory and allow the user to make efficient references to that memory.
13. **Learn from user behavior.** Personalize the user's experience by learning from their actions over time.
14. **Update and adapt cautiously.** Limit disruptive changes when updating and adapting the AI system's behaviors.
15. **Encourage granular feedback.** Enable the user to provide feedback indicating their preferences during regular interaction with the AI system.
16. **Convey the consequences of user actions.** Immediately update or convey how user actions will impact future behaviors of the AI system.
17. **Provide global controls.** Allow the user to globally customize what the AI system monitors and how it behaves.
18. **Notify users about changes.** Inform the user when the AI system adds or updates its capabilities.

Resources

- Original human-AI interaction guidelines research paper ↗
- Printable cards ↗

- Blog post: Guidelines for Human-AI Interaction ↗
- Blog post: AI Guidelines in the Creative Process ↗
- Microsoft Research: Guidelines for human-AI interaction design ↗

Responsible use of AI with Azure AI services

Article • 01/20/2024

Azure AI services provides information and guidelines on how to responsibly use artificial intelligence in applications. Below are the links to articles that provide this guidance for the different services within the Azure AI services suite.

Anomaly Detector

- Transparency note and use cases
- Data, privacy, and security

Azure AI Vision - OCR

- Transparency note and use cases
- Characteristics and limitations
- Integration and responsible use
- Data, privacy, and security

Azure AI Vision - Image Analysis

- Transparency note
- Characteristics and limitations
- Integration and responsible use
- Data, privacy, and security
- Limited Access features

Azure AI Vision - Face

- Transparency note and use cases
- Characteristics and limitations
- Integration and responsible use
- Data privacy and security
- Limited Access features

Azure AI Vision - Spatial Analysis

- Transparency note and use cases
- Responsible use in AI deployment
- Disclosure design guidelines
- Research insights
- Data, privacy, and security

Custom Vision

- Transparency note and use cases
- Characteristics and limitations
- Integration and responsible use
- Data, privacy, and security

Language service

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Custom text classification

- Transparency note
- Integration and responsible use
- Characteristics and limitations
- Data, privacy, and security

Language - Named entity recognition

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Custom named entity recognition

- Transparency note
- Integration and responsible use
- Characteristics and limitations
- Data, privacy, and security

Language - Entity linking

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Language detection

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Key phrase extraction

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Personally identifiable information detection

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Question Answering

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Sentiment Analysis and opinion mining

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Text Analytics for health

- Transparency note
- Integration and responsible use
- Data, privacy, and security

Language - Summarization

- Transparency note
- Integration and responsible use
- Characteristics and limitations
- Data, privacy, and security

Language Understanding

- Transparency note and use cases
- Characteristics and limitations
- Integration and responsible use
- Data, privacy, and security

Azure OpenAI Service

- Transparency note
- Limited access
- Code of conduct
- Data, privacy, and security

Personalizer

- Transparency note and use cases
- Characteristics and limitations
- Integration and responsible use
- Data and privacy

QnA Maker

- Transparency note and use cases
- Characteristics and limitations
- Integration and responsible use

- Data, privacy, and security

Speech - Pronunciation Assessment

- Transparency note and use cases
- Characteristics and limitations

Speech - Speaker Recognition

- Transparency note and use cases
- Characteristics and limitations
- Limited access
- General guidelines
- Data, privacy, and security

Speech - Custom Neural Voice

- Transparency note and use cases
- Characteristics and limitations
- Limited access
- Responsible deployment of synthetic speech
- Disclosure of voice talent
- Disclosure of design guidelines
- Disclosure of design patterns
- Code of conduct
- Data, privacy, and security

Speech - Text to speech

- Transparency note and use cases

Speech - Speech to text

- Transparency note and use cases
- Characteristics and limitations
- Integration and responsible use
- Data, privacy, and security

What is Responsible AI?

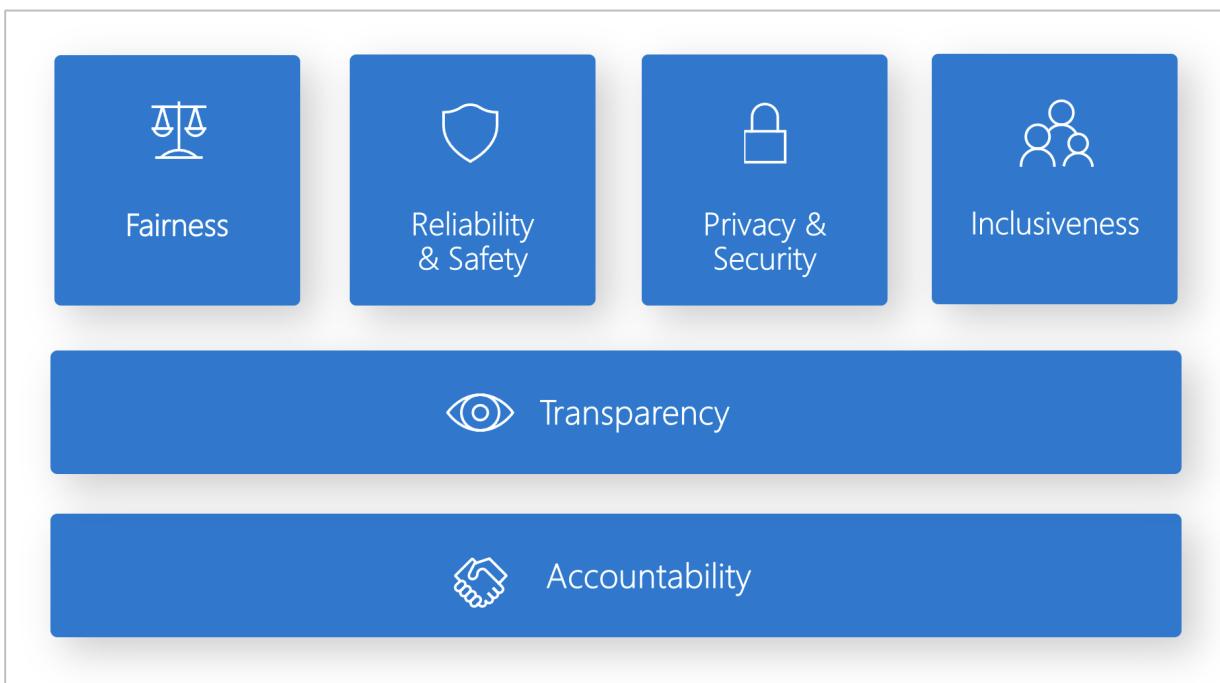
Article • 01/31/2024

APPLIES TO:  Azure CLI ml extension v2 (current)  Python SDK azure-ai-ml v2 (current) 

Responsible Artificial Intelligence (Responsible AI) is an approach to developing, assessing, and deploying AI systems in a safe, trustworthy, and ethical way. AI systems are the product of many decisions made by those who develop and deploy them. From system purpose to how people interact with AI systems, Responsible AI can help proactively guide these decisions toward more beneficial and equitable outcomes. That means keeping people and their goals at the center of system design decisions and respecting enduring values like fairness, reliability, and transparency.

Microsoft developed a [Responsible AI Standard](#). It's a framework for building AI systems according to six principles: fairness, reliability and safety, privacy and security, inclusiveness, transparency, and accountability. For Microsoft, these principles are the cornerstone of a responsible and trustworthy approach to AI, especially as intelligent technology becomes more prevalent in products and services that people use every day.

This article demonstrates how Azure Machine Learning supports tools for enabling developers and data scientists to implement and operationalize the six principles.



Fairness and inclusiveness

AI systems should treat everyone fairly and avoid affecting similarly situated groups of people in different ways. For example, when AI systems provide guidance on medical treatment, loan applications, or employment, they should make the same recommendations to everyone who has similar symptoms, financial circumstances, or professional qualifications.

Fairness and inclusiveness in Azure Machine Learning: The [fairness assessment](#) component of the [Responsible AI dashboard](#) enables data scientists and developers to assess model fairness across sensitive groups defined in terms of gender, ethnicity, age, and other characteristics.

Reliability and safety

To build trust, it's critical that AI systems operate reliably, safely, and consistently. These systems should be able to operate as they were originally designed, respond safely to unanticipated conditions, and resist harmful manipulation. How they behave and the variety of conditions they can handle reflect the range of situations and circumstances that developers anticipated during design and testing.

Reliability and safety in Azure Machine Learning: The [error analysis](#) component of the [Responsible AI dashboard](#) enables data scientists and developers to:

- Get a deep understanding of how failure is distributed for a model.
- Identify cohorts (subsets) of data with a higher error rate than the overall benchmark.

These discrepancies might occur when the system or model underperforms for specific demographic groups or for infrequently observed input conditions in the training data.

Transparency

When AI systems help inform decisions that have tremendous impacts on people's lives, it's critical that people understand how those decisions were made. For example, a bank might use an AI system to decide whether a person is creditworthy. A company might use an AI system to determine the most qualified candidates to hire.

A crucial part of transparency is *interpretability*: the useful explanation of the behavior of AI systems and their components. Improving interpretability requires stakeholders to comprehend how and why AI systems function the way they do. The stakeholders can then identify potential performance issues, fairness issues, exclusionary practices, or unintended outcomes.

Transparency in Azure Machine Learning: The [model interpretability](#) and [counterfactual what-if](#) components of the [Responsible AI dashboard](#) enable data scientists and developers to generate human-understandable descriptions of the predictions of a model.

The model interpretability component provides multiple views into a model's behavior:

- *Global explanations.* For example, what features affect the overall behavior of a loan allocation model?
- *Local explanations.* For example, why was a customer's loan application approved or rejected?
- *Model explanations for a selected cohort of data points.* For example, what features affect the overall behavior of a loan allocation model for low-income applicants?

The counterfactual what-if component enables understanding and debugging a machine learning model in terms of how it reacts to feature changes and perturbations.

Azure Machine Learning also supports a [Responsible AI scorecard](#). The scorecard is a customizable PDF report that developers can easily configure, generate, download, and share with their technical and non-technical stakeholders to educate them about their datasets and models health, achieve compliance, and build trust. This scorecard can also be used in audit reviews to uncover the characteristics of machine learning models.

Privacy and security

As AI becomes more prevalent, protecting privacy and securing personal and business information are becoming more important and complex. With AI, privacy and data security require close attention because access to data is essential for AI systems to make accurate and informed predictions and decisions about people. AI systems must comply with privacy laws that:

- Require transparency about the collection, use, and storage of data.
- Mandate that consumers have appropriate controls to choose how their data is used.

Privacy and security in Azure Machine Learning: Azure Machine Learning enables administrators and developers to [create a secure configuration that complies](#) with their companies' policies. With Azure Machine Learning and the Azure platform, users can:

- Restrict access to resources and operations by user account or group.
- Restrict incoming and outgoing network communications.
- Encrypt data in transit and at rest.
- Scan for vulnerabilities.

- Apply and audit configuration policies.

Microsoft also created two open-source packages that can enable further implementation of privacy and security principles:

- [SmartNoise](#): Differential privacy is a set of systems and practices that help keep the data of individuals safe and private. In machine learning solutions, differential privacy might be required for regulatory compliance. SmartNoise is an open-source project (co-developed by Microsoft) that contains components for building differentially private systems that are global.
- [Counterfit](#): Counterfit is an open-source project that comprises a command-line tool and generic automation layer to allow developers to simulate cyberattacks against AI systems. Anyone can download the tool and deploy it through Azure Cloud Shell to run in a browser, or deploy it locally in an Anaconda Python environment. It can assess AI models hosted in various cloud environments, on-premises, or in the edge. The tool is agnostic to AI models and supports various data types, including text, images, or generic input.

Accountability

The people who design and deploy AI systems must be accountable for how their systems operate. Organizations should draw upon industry standards to develop accountability norms. These norms can ensure that AI systems aren't the final authority on any decision that affects people's lives. They can also ensure that humans maintain meaningful control over otherwise highly autonomous AI systems.

Accountability in Azure Machine Learning: [Machine learning operations \(MLOps\)](#) is based on DevOps principles and practices that increase the efficiency of AI workflows. Azure Machine Learning provides the following MLOps capabilities for better accountability of your AI systems:

- Register, package, and deploy models from anywhere. You can also track the associated metadata that's required to use the model.
- Capture the governance data for the end-to-end machine learning lifecycle. The logged lineage information can include who is publishing models, why changes were made, and when models were deployed or used in production.
- Notify and alert on events in the machine learning lifecycle. Examples include experiment completion, model registration, model deployment, and data drift detection.
- Monitor applications for operational issues and issues related to machine learning. Compare model inputs between training and inference, explore model-specific

metrics, and provide monitoring and alerts on your machine learning infrastructure.

Besides the MLOps capabilities, the [Responsible AI scorecard](#) in Azure Machine Learning creates accountability by enabling cross-stakeholder communications. The scorecard also creates accountability by empowering developers to configure, download, and share their model health insights with their technical and non-technical stakeholders about AI data and model health. Sharing these insights can help build trust.

The machine learning platform also enables decision-making by informing business decisions through:

- Data-driven insights, to help stakeholders understand causal treatment effects on an outcome, by using historical data only. For example, "How would a medicine affect a patient's blood pressure?" These insights are provided through the [causal inference](#) component of the [Responsible AI dashboard](#).
- Model-driven insights, to answer users' questions (such as "What can I do to get a different outcome from your AI next time?") so they can take action. Such insights are provided to data scientists through the [counterfactual what-if](#) component of the [Responsible AI dashboard](#).

Next steps

- For more information on how to implement Responsible AI in Azure Machine Learning, see [Responsible AI dashboard](#).
- Learn how to generate the Responsible AI dashboard via [CLI and SDK](#) or [Azure Machine Learning studio UI](#).
- Learn how to generate a [Responsible AI scorecard](#) based on the insights observed in your Responsible AI dashboard.
- Learn about the [Responsible AI Standard](#) ↗ for building AI systems according to six key principles.

Model interpretability

Article • 05/23/2023

This article describes methods you can use for model interpretability in Azure Machine Learning.

Important

With the release of the Responsible AI dashboard, which includes model interpretability, we recommend that you migrate to the new experience, because the older SDK v1 preview model interpretability dashboard will no longer be actively maintained.

Why model interpretability is important to model debugging

When you're using machine learning models in ways that affect people's lives, it's critically important to understand what influences the behavior of models. Interpretability helps answer questions in scenarios such as:

- Model debugging: Why did my model make this mistake? How can I improve my model?
- Human-AI collaboration: How can I understand and trust the model's decisions?
- Regulatory compliance: Does my model satisfy legal requirements?

The interpretability component of the [Responsible AI dashboard](#) contributes to the "diagnose" stage of the model lifecycle workflow by generating human-understandable descriptions of the predictions of a machine learning model. It provides multiple views into a model's behavior:

- Global explanations: For example, what features affect the overall behavior of a loan allocation model?
- Local explanations: For example, why was a customer's loan application approved or rejected?

You can also observe model explanations for a selected cohort as a subgroup of data points. This approach is valuable when, for example, you're assessing fairness in model predictions for individuals in a particular demographic group. The **Local explanation** tab of this component also represents a full data visualization, which is great for general

eyeballing of the data and looking at differences between correct and incorrect predictions of each cohort.

The capabilities of this component are founded by the [InterpretML](#) package, which generates model explanations.

Use interpretability when you need to:

- Determine how trustworthy your AI system's predictions are by understanding what features are most important for the predictions.
- Approach the debugging of your model by understanding it first and identifying whether the model is using healthy features or merely false correlations.
- Uncover potential sources of unfairness by understanding whether the model is basing predictions on sensitive features or on features that are highly correlated with them.
- Build user trust in your model's decisions by generating local explanations to illustrate their outcomes.
- Complete a regulatory audit of an AI system to validate models and monitor the impact of model decisions on humans.

How to interpret your model

In machine learning, *features* are the data fields you use to predict a target data point. For example, to predict credit risk, you might use data fields for age, account size, and account age. Here, age, account size, and account age are features. Feature importance tells you how each data field affects the model's predictions. For example, although you might use age heavily in the prediction, account size and account age might not affect the prediction values significantly. Through this process, data scientists can explain resulting predictions in ways that give stakeholders visibility into the model's most important features.

By using the classes and methods in the Responsible AI dashboard and by using SDK v2 and CLI v2, you can:

- Explain model prediction by generating feature-importance values for the entire model (global explanation) or individual data points (local explanation).
- Achieve model interpretability on real-world datasets at scale.
- Use an interactive visualization dashboard to discover patterns in your data and its explanations at training time.

By using the classes and methods in the SDK v1, you can:

- Explain model prediction by generating feature-importance values for the entire model or individual data points.
- Achieve model interpretability on real-world datasets at scale during training and inference.
- Use an interactive visualization dashboard to discover patterns in your data and its explanations at training time.

 **Note**

Model interpretability classes are made available through the SDK v1 package. For more information, see [Install SDK packages for Azure Machine Learning](#) and `azureml.interpret`.

Supported model interpretability techniques

The Responsible AI dashboard and `azureml-interpret` use the interpretability techniques that were developed in [Interpret-Community](#), an open-source Python package for training interpretable models and helping to explain opaque-box AI systems. Opaque-box models are those for which we have no information about their internal workings.

Interpret-Community serves as the host for the following supported explainers, and currently supports the interpretability techniques presented in the next sections.

Supported in Responsible AI dashboard in Python SDK v2 and CLI v2

Interpretability technique	Description	Type
Mimic Explainer (Global Surrogate) + SHAP tree	<p>Mimic Explainer is based on the idea of training global surrogate models to mimic opaque-box models. A global surrogate model is an intrinsically interpretable model that's trained to approximate the predictions of any opaque-box model as accurately as possible.</p> <p>Data scientists can interpret the surrogate model to draw conclusions about the opaque-box model. The Responsible AI dashboard uses LightGBM (<code>LGBMExplainableModel</code>), paired with the SHAP (<code>SHapley Additive exPlanations</code>) Tree Explainer, which is a specific explainer to trees and ensembles of trees. The combination of LightGBM and SHAP tree provides model-</p>	Model-agnostic

Interpretability technique	Description	Type
	agnostic global and local explanations of your machine learning models.	

Supported model interpretability techniques for text models

Interpretability technique	Description	Type	Text Task
SHAP text	SHAP (SHapley Additive exPlanations) is a popular explanation method for deep neural networks that provides insights into the contribution of each input feature to a given prediction. It's based on the concept of Shapley values, which is a method for assigning credit to individual players in a cooperative game. SHAP applies this concept to the input features of a neural network by computing the average contribution of each feature to the model's output across all possible combinations of features. For text specifically, SHAP splits on words in a hierarchical manner, treating each word or token as a feature. This produces a set of attribution values that quantify the importance of each word or token for the given prediction. The final attribution map is generated by visualizing these values as a heatmap over the original text document. SHAP is a model-agnostic method and can be used to explain a wide range of deep learning models, including CNNs, RNNs, and transformers. Additionally, it provides several desirable properties, such as consistency, accuracy, and fairness, making it a reliable and interpretable technique for understanding the decision-making process of a model.	Model Agnostic	Text Multi-class Classification, Text Multi-label Classification

Supported model interpretability techniques for image models

Interpretability technique	Description	Type	Vision Task
SHAP vision	<p>SHAP (SHapley Additive exPlanations) is a popular explanation method for deep neural networks that provides insights into the contribution of each input feature to a given prediction. It's based on the concept of Shapley values, which is a method for assigning credit to individual players in a cooperative game. SHAP applies this concept to the input features of a neural network by computing the average contribution of each feature to the model's output across all possible combinations of features. For vision specifically, SHAP splits on the image in a hierarchical manner, treating superpixel areas of the image as each feature. This produces a set of attribution values that quantify the importance of each superpixel or image area for the given prediction. The final attribution map is generated by visualizing these values as a heatmap. SHAP is a model-agnostic method and can be used to explain a wide range of deep learning models, including CNNs, RNNs, and transformers.</p> <p>Additionally, it provides several desirable properties, such as consistency, accuracy, and fairness, making it a reliable and interpretable technique for understanding the decision-making process of a model.</p>	Model Agnostic	Image Multi-class Classification, Image Multi-label Classification
Guided Backprop	<p>Guided-backprop is a popular explanation method for deep neural networks that provides insights into the learned representations of the model. It generates a visualization of the input features that activate a particular neuron in the model, by computing the gradient of the output with respect to the input image. Unlike other gradient-based methods, guided-backprop only backpropagates through positive gradients and uses a modified ReLU activation function to ensure that negative gradients don't influence the visualization. This results in a more interpretable and high-resolution saliency map that highlights the most important features in the input image for a given prediction. Guided-backprop can be used to explain a wide range of deep learning models, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers.</p>	AutoML	Image Multi-class Classification, Image Multi-label Classification

Interpretability technique	Description	Type	Vision Task
Guided gradCAM	<p>Guided GradCAM is a popular explanation method for deep neural networks that provides insights into the learned representations of the model. It generates a visualization of the input features that contribute most to a particular output class, by combining the gradient-based approach of guided backpropagation with the localization approach of GradCAM. Specifically, it computes the gradients of the output class with respect to the feature maps of the last convolutional layer in the network, and then weights each feature map according to the importance of its activation for that class. This produces a high-resolution heatmap that highlights the most discriminative regions of the input image for the given output class. Guided GradCAM can be used to explain a wide range of deep learning models, including CNNs, RNNs, and transformers. Additionally, by incorporating guided backpropagation, it ensures that the visualization is meaningful and interpretable, avoiding spurious activations and negative contributions.</p>	AutoML	Image Multi-class Classification, Image Multi-label Classification
Integrated Gradients	<p>Integrated Gradients is a popular explanation method for deep neural networks that provides insights into the contribution of each input feature to a given prediction. It computes the integral of the gradient of the output class with respect to the input image, along a straight path between a baseline image and the actual input image. This path is typically chosen to be a linear interpolation between the two images, with the baseline being a neutral image that has no salient features. By integrating the gradient along this path, Integrated Gradients provides a measure of how each input feature contributes to the prediction, allowing for an attribution map to be generated. This map highlights the most influential input features, and can be used to gain insights into the model's decision-making process. Integrated Gradients can be used to explain a wide range of deep learning models, including CNNs, RNNs, and transformers. Additionally, it's a theoretically grounded technique that satisfies a set of desirable</p>	AutoML	Image Multi-class Classification, Image Multi-label Classification

Interpretability technique	Description	Type	Vision Task
	properties, such as sensitivity, implementation invariance, and completeness.		
XRAI	XRAI is a novel region-based saliency method based on Integrated Gradients (IG). It over-segments the image and iteratively tests the importance of each region, coalescing smaller regions into larger segments based on attribution scores. This strategy yields high quality, tightly bounded saliency regions that outperform existing saliency techniques. XRAI can be used with any DNN-based model as long as there's a way to cluster the input features into segments through some similarity metric.	AutoML	Image Multi-class Classification, Image Multi-label Classification
D-RISE	D-RISE is a model agnostic method for creating visual explanations for the predictions of object detection models. By accounting for both the localization and categorization aspects of object detection, D-RISE can produce saliency maps that highlight parts of an image that most contribute to the prediction of the detector. Unlike gradient-based methods, D-RISE is more general and doesn't need access to the inner workings of the object detector; it only requires access to the inputs and outputs of the model. The method can be applied to one-stage detectors (for example, YOLOv3), two-stage detectors (for example, Faster-RCNN), and Vision Transformers (for example, DETR, OWL-ViT). D-Rise provides the saliency map by creating random masks of the input image and will send it to the object detector with the random masks of the input image. By assessing the change of the object detector's score, it aggregates all the detections with each mask and produce a final saliency map.	Model Agnostic	Object Detection

Supported in Python SDK v1

Interpretability technique	Description	Type
SHAP Tree Explainer	The SHAP Tree Explainer, which focuses on a polynomial, time-fast, SHAP value-estimation algorithm that's specific to <i>trees and forests</i> .	Model-specific

Interpretability technique	Description	Type
	<i>ensembles of trees.</i>	
SHAP Deep Explainer	Based on the explanation from SHAP, Deep Explainer is a "high-speed approximation algorithm for SHAP values in deep learning models that builds on a connection with DeepLIFT described in the SHAP NIPS paper . TensorFlow models and Keras models using the TensorFlow back end are supported (there's also preliminary support for PyTorch)."	Model-specific
SHAP Linear Explainer	The SHAP Linear Explainer computes SHAP values for a <i>linear model</i> , optionally accounting for inter-feature correlations.	Model-specific
SHAP Kernel Explainer	The SHAP Kernel Explainer uses a specially weighted local linear regression to estimate SHAP values for <i>any model</i> .	Model-agnostic
Mimic Explainer (Global Surrogate)	Mimic Explainer is based on the idea of training global surrogate models to mimic opaque-box models. A global surrogate model is an intrinsically interpretable model that's trained to approximate the predictions of <i>any opaque-box model</i> as accurately as possible. Data scientists can interpret the surrogate model to draw conclusions about the opaque-box model. You can use one of the following interpretable models as your surrogate model: LightGBM (LGBMExplainableModel), Linear Regression (LinearExplainableModel), Stochastic Gradient Descent explainable model (SGDExplainableModel), or Decision Tree (DecisionTreeExplainableModel).	Model-agnostic
Permutation Feature Importance Explainer	Permutation Feature Importance (PFI) is a technique used to explain classification and regression models that's inspired by Breiman's Random Forests paper (see section 10). At a high level, the way it works is by randomly shuffling data one feature at a time for the entire dataset and calculating how much the performance metric of interest changes. The larger the change, the more important that feature is. PFI can explain the overall behavior of <i>any underlying model</i> but doesn't explain individual predictions.	Model-agnostic

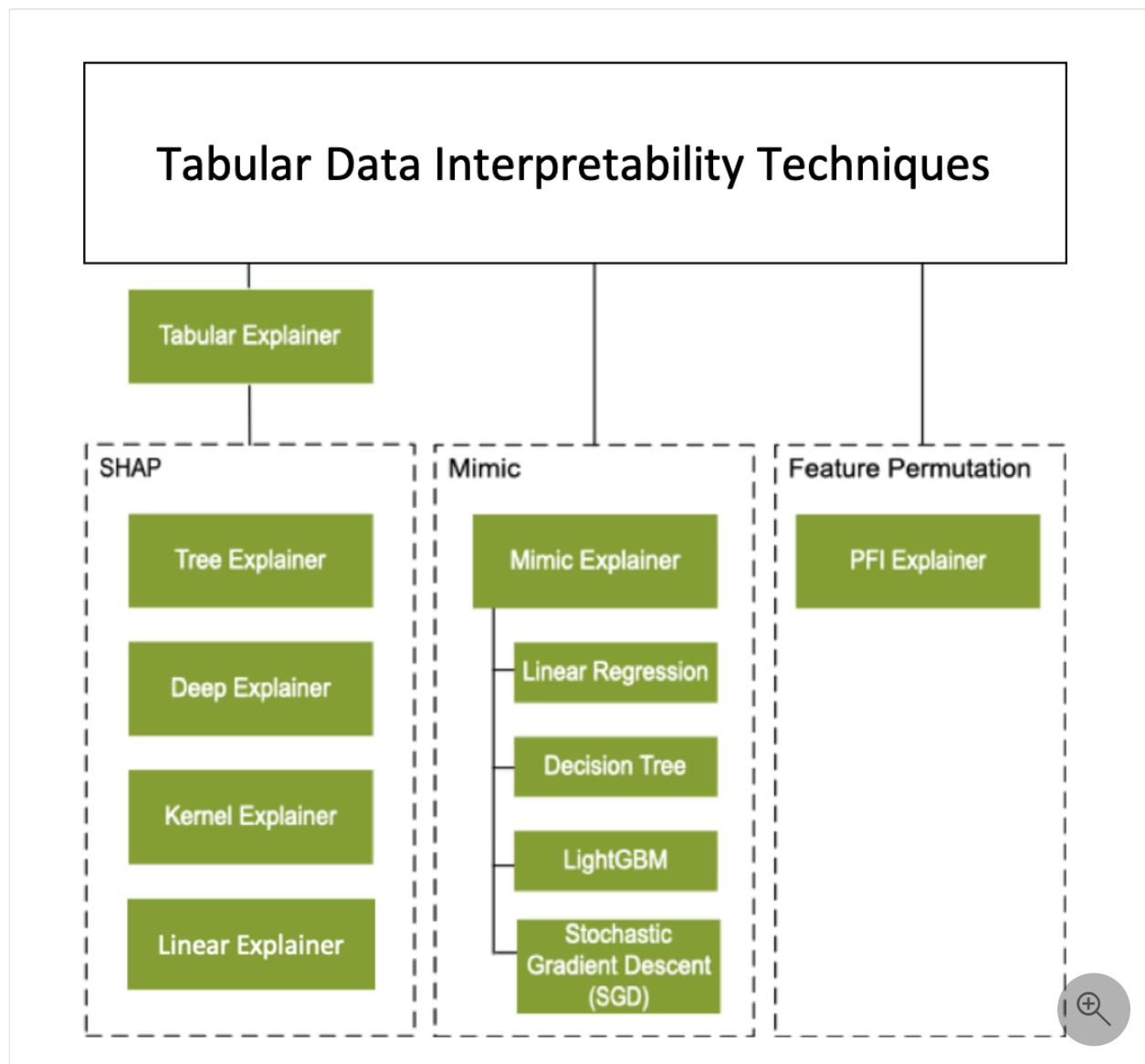
Besides the interpretability techniques described above, we support another SHAP-based explainer, called Tabular Explainer. Depending on the model, Tabular Explainer uses one of the supported SHAP explainers:

- Tree Explainer for all tree-based models
- Deep Explainer for deep neural network (DNN) models
- Linear Explainer for linear models
- Kernel Explainer for all other models

Tabular Explainer has also made significant feature and performance enhancements over the direct SHAP explainers:

- **Summarization of the initialization dataset:** When speed of explanation is most important, we summarize the initialization dataset and generate a small set of representative samples. This approach speeds up the generation of overall and individual feature importance values.
- **Sampling the evaluation data set:** If you pass in a large set of evaluation samples but don't actually need all of them to be evaluated, you can set the sampling parameter to `true` to speed up the calculation of overall model explanations.

The following diagram shows the current structure of supported explainers:



Supported machine learning models

The `azureml.interpret` package of the SDK supports models that are trained with the following dataset formats:

- `numpy.array`
- `pandas.DataFrame`
- `iml.datatypes.DenseData`
- `scipy.sparse.csr_matrix`

The explanation functions accept both models and pipelines as input. If a model is provided, it must implement the prediction function `predict` or `predict_proba` that conforms to the Scikit convention. If your model doesn't support this, you can wrap it in a function that generates the same outcome as `predict` or `predict_proba` in Scikit and use that wrapper function with the selected explainer.

If you provide a pipeline, the explanation function assumes that the running pipeline script returns a prediction. When you use this wrapping technique, `azureml.interpret` can support models that are trained via PyTorch, TensorFlow, and Keras deep learning frameworks as well as classic machine learning models.

Local and remote compute target

The `azureml.interpret` package is designed to work with both local and remote compute targets. If you run the package locally, the SDK functions won't contact any Azure services.

You can run the explanation remotely on Azure Machine Learning Compute and log the explanation info into the Azure Machine Learning Run History Service. After this information is logged, reports and visualizations from the explanation are readily available on Azure Machine Learning studio for analysis.

Next steps

- Learn how to generate the Responsible AI dashboard via [CLI v2 and SDK v2](#) or the [Azure Machine Learning studio UI](#).
- Explore the [supported interpretability visualizations](#) of the Responsible AI dashboard.
- Learn how to generate a [Responsible AI scorecard](#) based on the insights observed in the Responsible AI dashboard.
- Learn how to enable [interpretability for automated machine learning models \(SDK v1\)](#).

Model performance and fairness

Article • 02/27/2023 • 5 minutes to read

This article describes methods that you can use to understand your model performance and fairness in Azure Machine Learning.

What is machine learning fairness?

Artificial intelligence and machine learning systems can display unfair behavior. One way to define unfair behavior is by its harm, or its impact on people. AI systems can give rise to many types of harm. To learn more, see the [NeurIPS 2017 keynote by Kate Crawford ↗](#).

Two common types of AI-caused harms are:

- **Harm of allocation:** An AI system extends or withholds opportunities, resources, or information for certain groups. Examples include hiring, school admissions, and lending, where a model might be better at picking good candidates among a specific group of people than among other groups.
- **Harm of quality-of-service:** An AI system doesn't work as well for one group of people as it does for another. For example, a voice recognition system might fail to work as well for women as it does for men.

To reduce unfair behavior in AI systems, you have to assess and mitigate these harms. The *model overview* component of the [Responsible AI dashboard](#) contributes to the identification stage of the model lifecycle by generating model performance metrics for your entire dataset and your identified cohorts of data. It generates these metrics across subgroups identified in terms of sensitive features or sensitive attributes.

Note

Fairness is a socio-technical challenge. Quantitative fairness metrics don't capture many aspects of fairness, such as justice and due process. Also, many quantitative fairness metrics can't all be satisfied simultaneously.

The goal of the Fairlearn open-source package is to enable humans to assess the impact and mitigation strategies. Ultimately, it's up to the humans who build AI and machine learning models to make trade-offs that are appropriate for their scenarios.

In this component of the Responsible AI dashboard, fairness is conceptualized through an approach known as *group fairness*. This approach asks: "Which groups of individuals are at risk for experiencing harm?" The term *sensitive features* suggests that the system designer should be sensitive to these features when assessing group fairness.

During the assessment phase, fairness is quantified through *disparity metrics*. These metrics can evaluate and compare model behavior across groups either as ratios or as differences. The Responsible AI dashboard supports two classes of disparity metrics:

- **Disparity in model performance:** These sets of metrics calculate the disparity (difference) in the values of the selected performance metric across subgroups of data. Here are a few examples:
 - Disparity in accuracy rate
 - Disparity in error rate
 - Disparity in precision
 - Disparity in recall
 - Disparity in mean absolute error (MAE)
- **Disparity in selection rate:** This metric contains the difference in selection rate (favorable prediction) among subgroups. An example of this is disparity in loan approval rate. Selection rate means the fraction of data points in each class classified as 1 (in binary classification) or distribution of prediction values (in regression).

The fairness assessment capabilities of this component come from the [Fairlearn](#) package. Fairlearn provides a collection of model fairness assessment metrics and unfairness mitigation algorithms.

Note

A fairness assessment is not a purely technical exercise. The Fairlearn open-source package can identify quantitative metrics to help you assess the fairness of a model, but it won't perform the assessment for you. You must perform a qualitative analysis to evaluate the fairness of your own models. The sensitive features noted earlier are an example of this kind of qualitative analysis.

Parity constraints for mitigating unfairness

After you understand your model's fairness issues, you can use the mitigation algorithms in the [Fairlearn](#) open-source package to mitigate those issues. These

algorithms support a set of constraints on the predictor's behavior called *parity constraints* or criteria.

Parity constraints require some aspects of the predictor's behavior to be comparable across the groups that sensitive features define (for example, different races). The mitigation algorithms in the Fairlearn open-source package use such parity constraints to mitigate the observed fairness issues.

ⓘ Note

The unfairness mitigation algorithms in the Fairlearn open-source package can provide suggested mitigation strategies to reduce unfairness in a machine learning model, but those strategies don't eliminate unfairness. Developers might need to consider other parity constraints or criteria for their machine learning models. Developers who use Azure Machine Learning must determine for themselves if the mitigation sufficiently reduces unfairness in their intended use and deployment of machine learning models.

The Fairlearn package supports the following types of parity constraints:

Parity constraint	Purpose	Machine learning task
Demographic parity	Mitigate allocation harms	Binary classification, regression
Equalized odds	Diagnose allocation and quality-of-service harms	Binary classification
Equal opportunity	Diagnose allocation and quality-of-service harms	Binary classification
Bounded group loss	Mitigate quality-of-service harms	Regression

Mitigation algorithms

The Fairlearn open-source package provides two types of unfairness mitigation algorithms:

- **Reduction:** These algorithms take a standard black-box machine learning estimator (for example, a LightGBM model) and generate a set of retrained models by using a sequence of reweighted training datasets.

For example, applicants of a certain gender might be upweighted or downweighted to retrain models and reduce disparities across gender groups. Users can then pick a model that provides the best trade-off between accuracy (or another performance metric) and disparity, based on their business rules and cost calculations.

- **Post-processing:** These algorithms take an existing classifier and a sensitive feature as input. They then derive a transformation of the classifier's prediction to enforce the specified fairness constraints. The biggest advantage of one post-processing algorithm, threshold optimization, is its simplicity and flexibility because it doesn't need to retrain the model.

Algorithm	Description	Machine learning task	Sensitive features	Supported parity constraints	Algorithm type
ExponentiatedGradient	Black-box approach to fair classification described in A Reductions Approach to Fair Classification .	Binary classification	Categorical	Demographic parity, equalized odds	Reduction
GridSearch	Black-box approach described in A Reductions Approach to Fair Classification .	Binary classification	Binary	Demographic parity, equalized odds	Reduction

Algorithm	Description	Machine learning task	Sensitive features	Supported parity constraints	Algorithm type
<code>GridSearch</code>	<p>Black-box approach that implements a grid-search variant of fair regression with the algorithm for bounded group loss described in Fair Regression: Quantitative Definitions and Reduction-based Algorithms.</p>	Regression	Binary	Bounded group loss	Reduction
<code>ThresholdOptimizer</code>	<p>Postprocessing algorithm based on the paper Equality of Opportunity in Supervised Learning. This technique takes as input an existing classifier and a sensitive feature. Then, it derives a monotone transformation of the classifier's prediction to enforce the specified parity constraints.</p>	Binary classification	Categorical	Demographic parity, equalized odds	Post-processing

Next steps

- Learn how to generate the Responsible AI dashboard via [CLI](#) and [SDK](#) or [Azure Machine Learning studio UI](#).
 - Explore the [supported model overview](#) and [fairness assessment visualizations](#) of the Responsible AI dashboard.
 - Learn how to generate a [Responsible AI scorecard](#) based on the insights observed in the Responsible AI dashboard.
 - Learn how to use the components by checking out Fairlearn's [GitHub repository](#), [user guide](#), [examples](#), and [sample notebooks](#).
-

Additional resources

Documentation

[Model explainability in automated ML \(preview\) - Azure Machine Learning](#)

Learn how to get explanations for how your automated ML model determines feature importance and makes predictions when using the Azure Machine Learning SDK.

[Model interpretability - Azure Machine Learning](#)

Learn how your machine learning model makes predictions during training and inferencing by using the Azure Machine Learning CLI and Python SDK.

[Avoid overfitting & imbalanced data with AutoML - Azure Machine Learning](#)

Identify and manage common pitfalls of ML models with Azure Machine Learning's automated machine learning solutions.

[Evaluate AutoML experiment results - Azure Machine Learning](#)

Learn how to view and evaluate charts and metrics for each of your automated machine learning experiment jobs.

[Counterfactuals analysis and what-if - Azure Machine Learning](#)

Generate diverse counterfactual examples with feature perturbations to see minimal changes required to achieve desired prediction with the Responsible AI dashboard's integration of DiCE machine learning.

[Use pipeline parameters to retrain models in the designer - Azure Machine Learning](#)

Retrain models with published pipelines and pipeline parameters in Azure Machine Learning designer.

[Use the Responsible AI dashboard in Azure Machine Learning studio - Azure Machine Learning](#)

Learn how to use the various tools and visualization charts in the Responsible AI dashboard in Azure Machine Learning.

[Use Python to interpret & explain models \(preview\) - Azure Machine Learning](#)

Learn how to get explanations for how your machine learning model determines feature importance and makes predictions when using the Azure Machine Learning SDK.

[Show 5 more](#)

Training

Learning paths and modules

[Detect and mitigate unfairness in models with Azure Machine Learning - Training](#)

Detect and mitigate unfairness in models with Azure Machine Learning

What is Responsible AI?

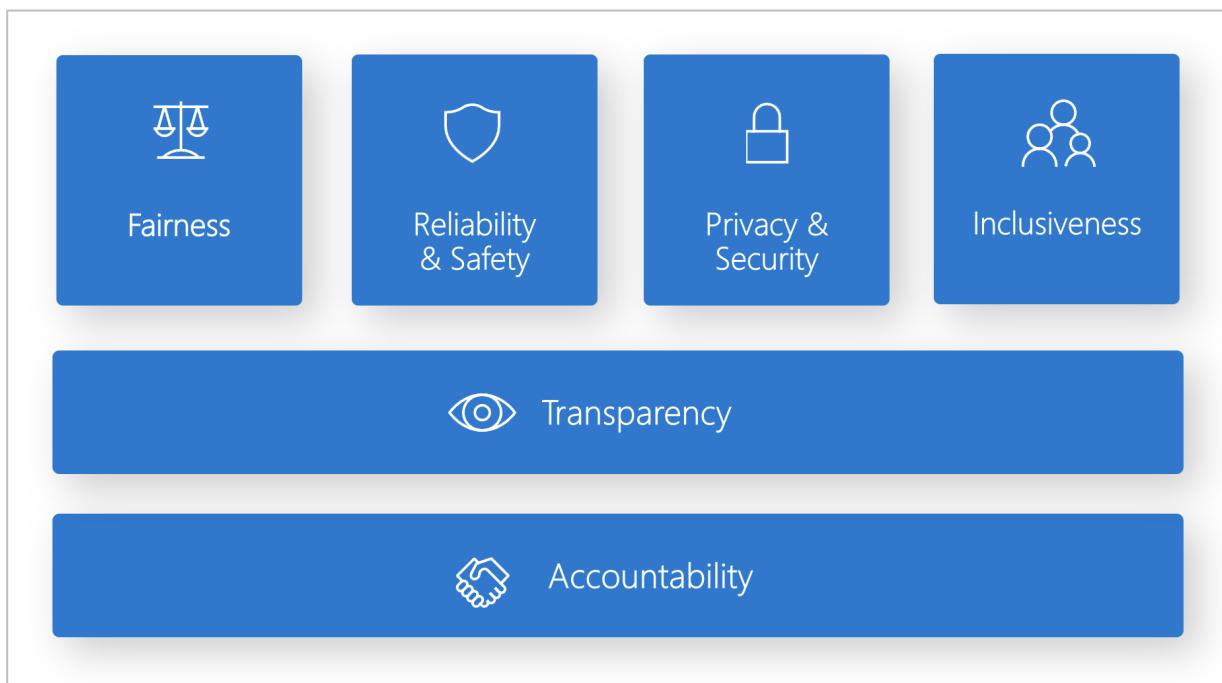
Article • 01/31/2024

APPLIES TO:  Azure CLI ml extension v2 (current)  Python SDK azure-ai-ml v2 (current) 

Responsible Artificial Intelligence (Responsible AI) is an approach to developing, assessing, and deploying AI systems in a safe, trustworthy, and ethical way. AI systems are the product of many decisions made by those who develop and deploy them. From system purpose to how people interact with AI systems, Responsible AI can help proactively guide these decisions toward more beneficial and equitable outcomes. That means keeping people and their goals at the center of system design decisions and respecting enduring values like fairness, reliability, and transparency.

Microsoft developed a [Responsible AI Standard](#). It's a framework for building AI systems according to six principles: fairness, reliability and safety, privacy and security, inclusiveness, transparency, and accountability. For Microsoft, these principles are the cornerstone of a responsible and trustworthy approach to AI, especially as intelligent technology becomes more prevalent in products and services that people use every day.

This article demonstrates how Azure Machine Learning supports tools for enabling developers and data scientists to implement and operationalize the six principles.



Fairness and inclusiveness

AI systems should treat everyone fairly and avoid affecting similarly situated groups of people in different ways. For example, when AI systems provide guidance on medical treatment, loan applications, or employment, they should make the same recommendations to everyone who has similar symptoms, financial circumstances, or professional qualifications.

Fairness and inclusiveness in Azure Machine Learning: The [fairness assessment](#) component of the [Responsible AI dashboard](#) enables data scientists and developers to assess model fairness across sensitive groups defined in terms of gender, ethnicity, age, and other characteristics.

Reliability and safety

To build trust, it's critical that AI systems operate reliably, safely, and consistently. These systems should be able to operate as they were originally designed, respond safely to unanticipated conditions, and resist harmful manipulation. How they behave and the variety of conditions they can handle reflect the range of situations and circumstances that developers anticipated during design and testing.

Reliability and safety in Azure Machine Learning: The [error analysis](#) component of the [Responsible AI dashboard](#) enables data scientists and developers to:

- Get a deep understanding of how failure is distributed for a model.
- Identify cohorts (subsets) of data with a higher error rate than the overall benchmark.

These discrepancies might occur when the system or model underperforms for specific demographic groups or for infrequently observed input conditions in the training data.

Transparency

When AI systems help inform decisions that have tremendous impacts on people's lives, it's critical that people understand how those decisions were made. For example, a bank might use an AI system to decide whether a person is creditworthy. A company might use an AI system to determine the most qualified candidates to hire.

A crucial part of transparency is *interpretability*: the useful explanation of the behavior of AI systems and their components. Improving interpretability requires stakeholders to comprehend how and why AI systems function the way they do. The stakeholders can then identify potential performance issues, fairness issues, exclusionary practices, or unintended outcomes.

Transparency in Azure Machine Learning: The [model interpretability](#) and [counterfactual what-if](#) components of the [Responsible AI dashboard](#) enable data scientists and developers to generate human-understandable descriptions of the predictions of a model.

The model interpretability component provides multiple views into a model's behavior:

- *Global explanations.* For example, what features affect the overall behavior of a loan allocation model?
- *Local explanations.* For example, why was a customer's loan application approved or rejected?
- *Model explanations for a selected cohort of data points.* For example, what features affect the overall behavior of a loan allocation model for low-income applicants?

The counterfactual what-if component enables understanding and debugging a machine learning model in terms of how it reacts to feature changes and perturbations.

Azure Machine Learning also supports a [Responsible AI scorecard](#). The scorecard is a customizable PDF report that developers can easily configure, generate, download, and share with their technical and non-technical stakeholders to educate them about their datasets and models health, achieve compliance, and build trust. This scorecard can also be used in audit reviews to uncover the characteristics of machine learning models.

Privacy and security

As AI becomes more prevalent, protecting privacy and securing personal and business information are becoming more important and complex. With AI, privacy and data security require close attention because access to data is essential for AI systems to make accurate and informed predictions and decisions about people. AI systems must comply with privacy laws that:

- Require transparency about the collection, use, and storage of data.
- Mandate that consumers have appropriate controls to choose how their data is used.

Privacy and security in Azure Machine Learning: Azure Machine Learning enables administrators and developers to [create a secure configuration that complies](#) with their companies' policies. With Azure Machine Learning and the Azure platform, users can:

- Restrict access to resources and operations by user account or group.
- Restrict incoming and outgoing network communications.
- Encrypt data in transit and at rest.
- Scan for vulnerabilities.

- Apply and audit configuration policies.

Microsoft also created two open-source packages that can enable further implementation of privacy and security principles:

- [SmartNoise](#): Differential privacy is a set of systems and practices that help keep the data of individuals safe and private. In machine learning solutions, differential privacy might be required for regulatory compliance. SmartNoise is an open-source project (co-developed by Microsoft) that contains components for building differentially private systems that are global.
- [Counterfit](#): Counterfit is an open-source project that comprises a command-line tool and generic automation layer to allow developers to simulate cyberattacks against AI systems. Anyone can download the tool and deploy it through Azure Cloud Shell to run in a browser, or deploy it locally in an Anaconda Python environment. It can assess AI models hosted in various cloud environments, on-premises, or in the edge. The tool is agnostic to AI models and supports various data types, including text, images, or generic input.

Accountability

The people who design and deploy AI systems must be accountable for how their systems operate. Organizations should draw upon industry standards to develop accountability norms. These norms can ensure that AI systems aren't the final authority on any decision that affects people's lives. They can also ensure that humans maintain meaningful control over otherwise highly autonomous AI systems.

Accountability in Azure Machine Learning: [Machine learning operations \(MLOps\)](#) is based on DevOps principles and practices that increase the efficiency of AI workflows. Azure Machine Learning provides the following MLOps capabilities for better accountability of your AI systems:

- Register, package, and deploy models from anywhere. You can also track the associated metadata that's required to use the model.
- Capture the governance data for the end-to-end machine learning lifecycle. The logged lineage information can include who is publishing models, why changes were made, and when models were deployed or used in production.
- Notify and alert on events in the machine learning lifecycle. Examples include experiment completion, model registration, model deployment, and data drift detection.
- Monitor applications for operational issues and issues related to machine learning. Compare model inputs between training and inference, explore model-specific

metrics, and provide monitoring and alerts on your machine learning infrastructure.

Besides the MLOps capabilities, the [Responsible AI scorecard](#) in Azure Machine Learning creates accountability by enabling cross-stakeholder communications. The scorecard also creates accountability by empowering developers to configure, download, and share their model health insights with their technical and non-technical stakeholders about AI data and model health. Sharing these insights can help build trust.

The machine learning platform also enables decision-making by informing business decisions through:

- Data-driven insights, to help stakeholders understand causal treatment effects on an outcome, by using historical data only. For example, "How would a medicine affect a patient's blood pressure?" These insights are provided through the [causal inference](#) component of the [Responsible AI dashboard](#).
- Model-driven insights, to answer users' questions (such as "What can I do to get a different outcome from your AI next time?") so they can take action. Such insights are provided to data scientists through the [counterfactual what-if](#) component of the [Responsible AI dashboard](#).

Next steps

- For more information on how to implement Responsible AI in Azure Machine Learning, see [Responsible AI dashboard](#).
- Learn how to generate the Responsible AI dashboard via [CLI and SDK](#) or [Azure Machine Learning studio UI](#).
- Learn how to generate a [Responsible AI scorecard](#) based on the insights observed in your Responsible AI dashboard.
- Learn about the [Responsible AI Standard](#) ↗ for building AI systems according to six key principles.

SaaS and multitenant solution architecture

Article • 04/21/2023

This series of articles provides guidance and resources for organizations that build software as a service (SaaS), including startups. It also provides extensive guidance about architecting multitenant solutions on Azure.

Key concepts

The key concepts in this series of articles are *SaaS*, *startups*, and *multitenancy*. These terms are related but distinct. SaaS and startup are business concepts, and multitenancy is an architecture concept.

SaaS is a business model. An organization can choose to provide their software product as a service to their customers. Commonly, SaaS products are sold to businesses (business-to-business, or B2B), or to consumers (business-to-consumer, or B2C). SaaS products are distinct from products that customers install and manage themselves. Many SaaS solutions use a multitenant architecture, but some don't. SaaS solutions might also use different multitenancy models or approaches.

Startups are businesses in an early stage of their lifecycle. Many software startups build SaaS solutions, but some might provide software in other ways instead. Startups often have specific concerns, including rapid innovation, finding a product and market fit, and anticipating scale and growth.

Multitenancy is a way of architecting a solution to share components between multiple customers, or *tenants*. Multitenant architectures are frequently used in SaaS solutions, but there are also some places where they're used outside of SaaS, such as in enterprises who build a platform for multiple business units to share. Multitenancy doesn't imply that every component in a solution is shared. Rather, it implies that at least *some* components of a solution are reused across multiple tenants.

Next steps

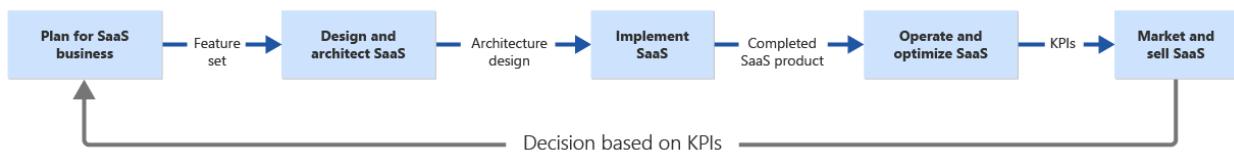
- [Plan your own journey to SaaS](#)
- [Understand how startups architect their solutions](#)
- [Learn about multitenant architectural approaches](#)

Plan your journey to SaaS

Article • 04/21/2023

Building and operating software as a service (SaaS) presents both a unique opportunity and a unique challenge for a business at any stage. The considerations of SaaS are important to keep in mind not only when planning your SaaS offering, but also on a daily basis as you operate your business.

The following diagram depicts the typical journey a company goes through while building a SaaS product. Understanding this process is helpful for knowing which resources apply to you at your current stage. The rest of the article provides a brief description of each stage of the SaaS journey and any links that are relevant to a business currently in that stage.



1. Plan for a SaaS business model

The first stage in the SaaS journey is centered around business decisions. Business decisions must be thought through carefully before making any technical decisions, as they eventually become the software requirements for your application. At a minimum, consider these issues:

- **Identify the problem you're trying to solve.** SaaS solutions are designed to solve business problems. Identify the business problem you're trying to solve before designing a solution around it.
- **Know how your solution solves the problem.** Understand clearly how your designed SaaS solution solves the problem you identified.
- **Know your pricing model.** SaaS solutions are ultimately designed to generate revenue. Understand the various [pricing models](#) and which one aligns best with the solution you're designing.
- **Understand your customers and how they will interact with your application.** Know who your customers are and what features they care about. Knowing this up front saves you precious time and energy so that you're not developing features that are underutilized.

In addition to your application requirements, also consider these few things that relate to your overall business:

- **Ensure your business is ready to take on the responsibility of operating a SaaS application.** Operating a SaaS business means customers depend solely on your company for things like support. Make sure you have the ability to provide support for the application, potentially on a 24/7 basis.
- **Ensure you have a smooth path for migration from legacy offerings.** If you plan on migrating from a different business model, make sure you have a plan in place for migrating your customers without too much disruption.
- **Understand how the processes you establish will scale.** As you're planning, proceed with the understanding that processes need to change over time as your business grows. You might be able to do some things manually when you only have a handful of customers, but this approach doesn't scale well. For more information, see these articles:
 - [Foundations of SaaS](#) - A Microsoft Learn module about the foundations of SaaS
 - [Accelerate and de-risk your journey to SaaS](#) - A video from Microsoft Ignite 2021 that outlines the key considerations, challenges, and other lessons for SaaS migration and modernization projects.
 - [Microsoft SaaS Academy](#) - Free SaaS learning courses.
 - [Pricing model considerations](#) - Important technical considerations to keep in mind when deciding on a pricing strategy.
 - [Microsoft for Startups Founders Hub](#) - A resource center for startups building solutions on Azure that provides business and technical mentoring, such as Microsoft software for running your business including LinkedIn, Microsoft 365 and GitHub Enterprise, and Azure credits.
 - [Microsoft SaaS Stories](#) - A series of video interviews with some of Microsoft's ISV partners that highlight their experiences building SaaS.

2. Design and architect a SaaS solution

After deciding what your business requirements are, the next stage in the journey is to design your application to support your requirements. SaaS products typically need to take into account the concept of multitenancy, and there are many considerations that come into play. The output of this step should be an application architecture that addresses your specific requirements and any considerations. For more information, see these articles:

- [Architect multitenant solutions on Azure](#) - An introduction to multitenant applications on Azure.
- [Multitenant architecture considerations](#) - Key considerations of designing a multitenant architecture.
- [Tenancy models](#) - An overview of the main tenancy models and the differences between them.
- [Independent software vendor \(ISV\) considerations for Azure landing zones](#) - A comparison between different landing zones for ISV scenarios.
- [Azure Well-Architected Framework](#) - A set of guiding tenets that help improve the quality of a workload.
- [SaaS journey review](#) - An assessment of your SaaS product examining your knowledge of multitenant architecture.
- [Technical guide to building SaaS apps on Azure](#) - An E-book created for ISVs, technical professionals, and technical business leaders that outlines several SaaS technical decision points.
- [Architecture for startups](#) - An introduction to architectures for startups.

3. Implement a SaaS solution

You need to implement the architecture you developed. In this stage, you develop and iterate on your SaaS product using the normal software development life cycle (SDLC) process. It's important in this stage to not put too many requirements into development at one time. Try to figure out which features would provide the most benefit to your customers and start from a minimum viable product (MVP). More iterations with smaller improvements over time are easier to implement than larger chunks of development.

For more information, see these articles:

- [SaaS starter web app architecture](#) - A reference architecture for a starter web-based SaaS application.
- [Azure SaaS Development Kit \(ASDK\)](#) - A modular implementation of the architecture designed to provide a starting place for building a SaaS application in .NET.

4. Operate your SaaS solution

In this stage, you begin to onboard customers to your new SaaS product and begin operating as a SaaS provider with users in production. Have your SaaS product close to completion and have a strategy to migrate existing customers or onboard new ones. Have a plan in place to support your customers if problems arise. It's also important to begin identifying key performance indicator (KPI) metrics that you can collect, which

help drive various business and technical decisions later on. For more information, see these articles:

- [Deploy multitenant applications](#) - Considerations for maintaining and deploying to your multitenant application.
- [Measure tenant consumption](#) - Considerations for collecting consumption data from your multitenant application.

5. Market and sell your SaaS solution

In this stage, you begin to market and sell your SaaS solution. Explore all avenues available to you for selling your application, including but not limited to the [Azure Marketplace](#). This stage is also when you begin to take the KPI data from the previous stage and use it to analyze how your customers are interacting with your SaaS application. Then use that analysis to make business and technical decisions about the roadmap of your SaaS product. For more information, see these articles:

- [Mastering the marketplace](#) - Learning content that is focused around how to best take advantage of the Azure Marketplace.
- [Marketplace publishing guide](#) - The offer types that are available in the Azure Marketplace and the key differences between them.
- [Marketing best practices](#) - A comprehensive guide for using the Azure Marketplace to market and sell your application.
- [Plan a SaaS marketplace offer](#) - The documentation page for how to plan a SaaS offer on the Azure Marketplace.
- [Co-sell with Microsoft sales teams](#) - An overview of how to Co-sell with Microsoft sales teams.
- [Join the Microsoft partner network](#) - The Microsoft partner network. Here, you register your company as a Microsoft partner and obtain information about the various partner programs.

6. Repeat the process

Developing SaaS solutions is a cyclical journey. To get the most out of your SaaS product, you must constantly iterate and adapt to the needs of your customers and the market. After you have made your decisions about the current direction of your product, the process starts over at stage one. For more information, see these articles:

- [Azure well-architected review](#) - An assessment of your workload against the Azure Well Architected Framework that results in curated and personalized guidance for

your scenario. Complete this review regularly to identify areas of your application you can improve.

- [SaaS journey review](#) - An assessment of your SaaS product examining your knowledge of multitenant architecture and evaluating adherence to SaaS operation best practices.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Landon Pierce](#) | Customer Engineer, FastTrack for Azure
- [Arsen Vladimirsy](#) | Principal Customer Engineer, FastTrack for Azure

Other contributors:

- [John Downs](#) | Principal Customer Engineer, FastTrack for Azure
- [Irina Kostina](#) | Software Engineer, FastTrack for Azure
- [Nick Ward](#) | Senior Cloud Solution Architect

Next steps

- [Foundations of SaaS](#)
- [Technical guide to building SaaS apps on Azure](#)
- [Azure Well-Architected Framework](#)

Related resources

- [SaaS and multitenant solution architecture](#)
- [Understand how startups architect their solutions](#)
- [Learn about multitenant architectural approaches](#)

Microsoft SaaS stories video interviews

Article • 11/01/2023

Microsoft SaaS stories is a collection of insightful interviews featuring Microsoft partners who have embarked on their unique software as a service (SaaS) journey. These interviews delve into the experiences of these partners as they develop and publish their SaaS solutions on the Microsoft commercial marketplace, while also highlighting their growth strategies and successes. By showcasing a diverse range of case studies, this series aims to inspire and inform businesses looking to build and scale their own SaaS offerings within the Azure ecosystem.

Episode 1: Basis Theory

In our first episode, we talk with Basis Theory features CTO Brandon Weber. Basis Theory shares how they built confidence with their customers by creating an easy-to-use SaaS platform that scales while remaining reliable and secure. Learn the challenges they encountered running a 24/7 service while evolving the service and handling customer growth.

<https://www.youtube-nocookie.com/embed/f0pJ9FljpGI> ↗

Episode 2: Zammo.ai

In our second episode with Zammo's Stacey Kyler and Nicholas Spagnola, we learn about their significant growth in business and faster time to close based on having their products in the marketplace. They share their experience building for Azure and running a No-Code Conversational AI Software SaaS platform.

<https://www.youtube-nocookie.com/embed/ngdWwIIXbJc> ↗

Episode 3: Wolfpack

In the third episode with Wolfpack's Koen den Hollander, we learn how they built their SaaS application for retail customers, and how connecting engineers directly to customers enables them to deliver value at scale.

https://www.youtube-nocookie.com/embed/NDcpqd9Df_s ↗

Episode 4: Vocean

In this episode, we're exploring how Vocean built their SaaS application that changes the way organizations make decisions. They share the importance of taking time to plan, learn, and listen to experts around you before rushing to build features.

<https://www.youtube-nocookie.com/embed/VsAHRPvbeBg> ↗

Episode 5: Access Infinity

In this episode, we talk to Access Infinity's Managing Director, Keshav Nagaraja. We explore how Access Infinity saw an opportunity in their consulting business to create platforms that help their customers at scale, and how they came up with a pricing model that drives positive user behaviors.

<https://www.youtube-nocookie.com/embed/28OQ0rVAbLY> ↗

Episode 6: Sage

In this episode, we learn how Sage embraced the opportunities to shift their application to SaaS, how they used SaaS as an opportunity to simplify their pricing model, and how they use a simple set of principles to guide complex changes.

<https://www.youtube-nocookie.com/embed/Oeh2y-giKmw> ↗

Episode 7: CallCabinet

In this episode, we learn how CallCabinet moved from tapes to servers to the cloud, dramatically reducing user costs while focusing on innovative features. We discuss why they adopted a SaaS model and their biggest challenges in doing so, how they built for scale, how they are using artificial intelligence (AI), and more.

<https://www.youtube-nocookie.com/embed/8nwGrCrTDWE> ↗

Episode 8: Octopai

In this episode, we explore how Octopai embraced a global SaaS model and are providing innovative value to their customers through new AI technologies. We speak

with Zinette Ezra, VP of Product at Octopai to learn more about their pricing model, challenges they have overcome, and how they built for scale and security.

<https://www.youtube-nocookie.com/embed/zD6l370JEt8> ↗

Next steps

- Plan your own journey to SaaS
- Learn about multitenant architectural approaches

The journey to SaaS: Dynamics 365

Article • 11/14/2023

Many independent software vendors (ISVs) think about moving from on-premises software delivery to a cloud-based and software-as-a-service (SaaS)-based delivery model. At Microsoft, we've been through this journey with many of our products, and we get asked to share our real-life experiences and the key lessons that we learned along the way.

Our goal with this article is to give an overview of how this journey played out when we built Microsoft Dynamics 365. We describe the thought process we went through and the key drivers for each of the major decisions we made. We hope that this document provides a sense of the evolution of our product as we moved from delivering on-premises software to a hyperscale SaaS product used by millions of users across thousands of organizations. We hope that by reading this document, you can learn from our experiences and plan your own journey to SaaS. While the Microsoft journey with Dynamics 365 might be unique, we believe that the lessons and principles we learned can still provide valuable insights for organizations of any size that are planning their own transition to a SaaS model.

A brief history of our journey

Microsoft Dynamics has a deep history as a set of on-premises products. We adopted the cloud for all of the many benefits it offered, and we knew that our technology and business model would need to adapt as we moved toward providing SaaS.

The first decision that we confronted was the choice between building something new or evolving our on-premises applications into cloud services. For Dynamics 365, we believed two things. First, there was sufficient value in the data models and business logic that we'd created and validated with thousands of customers that it was worth trying to evolve our existing solutions. Second, the layered architecture and platform framework of our on-premises products provided the right levers to allow us to evolve to a great cloud architecture more quickly than starting from scratch. The combination of value and speed, along with the understanding that we could evolve to the correct cloud-native principles, made evolution to cloud-based SaaS and continuous improvement the right choice for Dynamics 365. Other organizations might have different priorities and come to a different strategy.

Early on, we decided to focus on building the best product we could build on the Microsoft Azure platform. Cloud platforms evolve rapidly, and we wanted to take

advantage of the richness of one platform, instead of spreading our resources across multiple clouds. Other SaaS vendors might make different decisions based on their own situations. For example, a horizontal platform provider might build software for customers to use across multiple clouds, so it makes sense for them to have a presence on each of those cloud platforms. But a SaaS application developer can make a choice to focus on one cloud and gain the benefits of focusing on that one cloud provider and their evolution. For Dynamics 365, we knew that going all-in on Microsoft Azure would give us a more integrated and seamless experience while maintaining robust security and high performance.

As we started to deeply explore the Azure platform and plan our SaaS journey, we learned how to operate and scale a massive enterprise resource planning (ERP) platform in the cloud. At the same time, Azure has become richer and more capable, and it introduced new capabilities that we couldn't have imagined. We knew the constant evolution of the cloud meant that our migration wouldn't be a one-time thing. Instead, we thought about continuous improvement at every step of our journey. Continuous improvement affects everything we do, and in the early stages of our journey, we had to make significant changes—everything from our overall architecture to how we dealt with database queries. We're constantly evolving our solution to make the most of what the cloud enables. We've embraced microservice architecture, and we use generative AI as part of our ongoing evolution. These approaches and technologies are easier to build, deploy, and operate when you use a powerful cloud platform like Microsoft Azure.

An essential ingredient of continuous improvement in the cloud is telemetry. With effective telemetry, you can understand how the application is used and how it performs—even down to the level of individual features. Telemetry provides insights that let you solve problems by knowing what's happened instead of following traditional on-premises approaches like reproducing the problem and debugging. Telemetry also allows you to make engineering decisions based on real data and to confirm product hypotheses through experimentation and data. Creating a telemetry infrastructure *along with the right policies around what data is retained and for how long, and how it's managed* is something that should be done as early as possible.

Understanding data classification and retention policies for the data managed by your application also requires extra attention as part of the journey to SaaS. As a SaaS provider, your responsibilities under current and emerging data privacy laws are different than when you supplied software that customers deployed and ran themselves. It's essential that you understand the data privacy regulations that apply to you as a SaaS provider. You also need to get your data classification and retention processes in place at the start of your cloud journey.

How we prepared for the journey

Scoping an MVP

Our strategy focused on building a minimum viable product (MVP) so that we could get the solution to customers as quickly as possible and begin learning about the unique challenges and opportunities of SaaS. This focus was a strategic choice. We believe that rapid learning and iteration is essential in the cloud, and defining the MVP is a place to start.

The term *minimum viable product* is often misunderstood. It's important to consider that both attributes of the MVP are equally important:

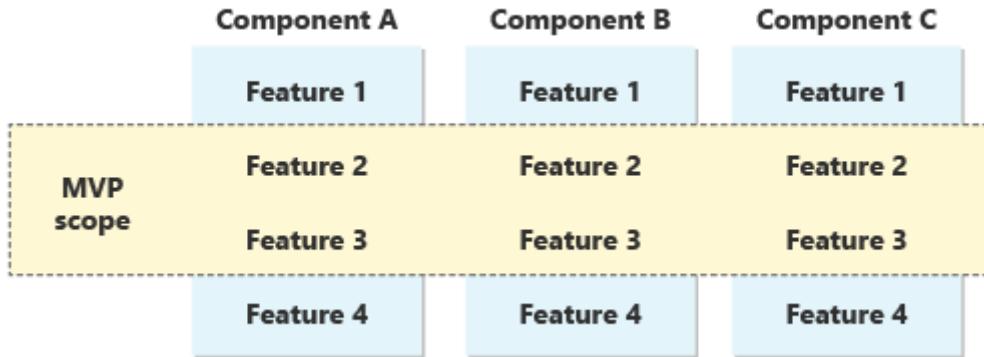
- **Minimum:** Figure out the quickest way to start generating value for your customers. The sooner your customers use your solution, the sooner you start learning how they use it and how you can continue to improve it.
- **Viable:** It's critical that you scope your product so that it's complete enough for somebody to get real value from it. Initially, you pick a subset of the overall capabilities that you expect to build. Picking the right subset is important—if you're too *minimum* to be viable, then customers don't use the product, and you don't get the feedback you need to evolve.

For Dynamics 365, we focused on making a product that was ready for customers to use right from its launch. Then, we learned how customers got value from it, and we got a large amount of feedback and telemetry. We used that data to inform our product journey, iterating and making it better and better as we progressed.

Our strategy was to build a product that new customers would love. While we were intentional about making migrations from existing on-premises customers easier, migration was a secondary focus compared to building a great modern product. This strategy meant that our new customers would have a complete experience in Dynamics 365 right from the start. They also provided us with invaluable feedback, and they did so with the benefit of fresh eyes. They weren't already heavily invested in the on-premises Dynamics products, so they helped us to build a product that truly was cloud native and a full SaaS offering. As we continued to improve and expand our capabilities, we eventually reached the point where the feature set was a superset of the on-premises products. At that point, we could start to support the transition of our existing customers from the on-premises version to the more advanced Dynamics 365.

We consciously chose this strategy because we knew it would work well for an ERP system. In other products, it might be possible to pick a subset of the product's features to move first, and then add more features over time. But in an ERP, components are

tightly interconnected. The product isn't useful until there's a slice of functionality across all these components, providing a useful end-to-end experience to customers. The MVP scope is a horizontal slice of features across each component. We decided to select a cross-cutting set of functionalities that would support new customers' use cases:



For other solutions, it might make sense to instead scope an MVP as a whole component. It's important to make a conscious decision about the strategy you follow when you embark on your own journey to SaaS. The key is that the initial deliverable to market should be as small as possible, while still being complete enough to get real usage.

As we planned and improved, we kept in mind that customer expectations were continuously evolving, too. Rather than migrating our product in its exact current state, we instead planned for what customers would need by the time we had a product ready for them to use. A journey to SaaS coupled with a cloud migration is often a long-term endeavor, taking months or even years. It's important not to lose sight of changes in customer demand during this time. Otherwise, you can spend significant effort building something that doesn't fully address customer needs when it finally arrives.

Usage, customer satisfaction, and costs

On-premises software revenue is typically recognized at the point that the sales transaction occurs, and the responsibility for successful deployment and adoption rests with the customer. With a cloud SaaS subscription model, customers often begin by licensing a few seats and only expand their subscription after the solution has been proven. Any seats that they purchase but don't use are a risk because they might be canceled at the next subscription anniversary. As a result, with the transition to SaaS, we changed the top metrics that we used to drive our business.

In the on-premises world of licensed software, the primary focus was revenue. In the cloud, the focus was on usage and customer satisfaction. These metrics became the forward indicators of revenue and revenue growth. We spent effort on minimizing the time to successful deployment, providing visibility of purchased but unused licenses,

and maintaining high satisfaction across user and business roles. From the start, our focus has been on creating products that customers love to use. We know from experience that when customers get value from using the product, revenue follows. By prioritizing customer experience and usage, we set the foundations for a successful business strategy.

When you build SaaS, the cost of goods sold (COGS) matters a lot, especially as you scale and your costs grow, too. But it's better to prioritize satisfaction and usage first. If you provide a good customer experience, you can optimize the costs of delivering the service by making more efficient use of your resources and taking advantage of new platform capabilities. If the experience isn't good enough, usage will be lower and you'll have fewer customers to satisfy. So when we review our progress, we focus on three key performance indicators, in order of importance:

- **Customer satisfaction:** Do our customers like the experience of using the product? What's their feedback?
- **Usage:** How many users do we have? How many subscriptions do we have? Is our usage accelerating? What's the time between purchase and usage? How can we encourage customers to use all the subscriptions they purchase?
- **COGS:** How much does it cost to serve our customers?

It's also important to think about how any SaaS product generates revenue. Customers need to understand how they pay for the service, and the pricing model needs to make sense to them. In many business-to-business SaaS solutions, the number of users that the customer has is a great indicator of the resources consumed when the users use the system. The more users who actively use the system, the more system resources are needed to give them a good experience. The cost to the customer reflects that fact. Customers have an intuitive understanding of a user-based pricing structure.

However, there are some situations where user counts don't give a good indication of the resources that the customer consumes. For example, when a customer's marketing team sends out a large number of messages for an email campaign, there might just be one user sending millions of email messages. Similarly, a background process, not a user, imports order details. It's important that customers understand the metrics they're charged for and can predict their bill. You might choose to use meters like the number of contacts they send email messages to or the number of order lines they process each month.

The architecture of Dynamics 365

Identity, authentication, and authorization

Business applications like Dynamics 365 manage high-value business data and automate mission-critical business activities. It's essential to ensure that only authorized users have access to data and the system's actions. By using Microsoft Entra ID, enterprises can manage access to Dynamics 365 with the same tools and platforms that they already use across their IT estate. Customers can take advantage of advanced security features like Conditional Access without more work on our part. The capabilities to secure their Dynamics system continue to evolve with the ongoing investment by Microsoft in the Entra platform.

Dynamics 365 assigns users to roles and assigns permissions for specific data and actions to those roles. This approach follows a common pattern for managing authorization beyond the user authentication provided by Entra. This approach also provides the capability for Dynamics 365 to enforce best-practice business requirements like the separation of duties.

Tenancy model

Each organization that uses Dynamics 365 expects their data to be kept secure and isolated from access by other organizations. We model each organization as a *tenant*, and each tenant has many users that can each use the products and work with the organization's data. Sharing resources reduces cost components of running the services, but sharing must be balanced against the requirements to ensure the expected levels of tenant isolation. Fortunately, the Azure platform provides rich capabilities to enable application providers to balance cost as they deliver required isolation.

For example, we felt that it was important to keep each tenant's business data in a separate SQL database. This separation allows us, among other capabilities, to implement Azure SQL transparent data encryption (TDE) with customer-managed keys—an important component of our enterprise trust promises regarding data. Azure SQL, specifically including elastic pools, gives us cost efficiency while still allowing a separate database per customer. In addition to increasing infrastructure costs, the decision to keep a separate database per tenant increases the management complexity. There aren't enough database administrators (DBAs) to manage databases manually at the scale of the Dynamics service, and that led to significant investment in automation of management tasks. For more information about how Dynamics 365 works with databases at scale, see [Running 1M databases on Azure SQL for a large SaaS provider: Microsoft Dynamics 365 and Power Platform](#).

For every tier of our solution, our strategy has been to use native Azure platform capabilities to enforce tenant isolation and deliver scale and resiliency while simultaneously gaining cost efficiencies where we can. We're always looking at places