

Contents

[Azure Architecture Center](#)

[Azure Cloud Adoption Guide](#)

[Foundational Adoption Stage](#)

[Intermediate Adoption Stage](#)

[Application Architecture Guide](#)

[Architecture styles](#)

[Choosing a compute service](#)

[Choosing a data store](#)

[Design principles](#)

[Pillars of software quality](#)

[Design patterns](#)

[Data Architecture Guide](#)

[Reference Architectures](#)

[App Service web applications](#)

[Hybrid networks](#)

[Identity management](#)

[Jenkins server](#)

[N-tier application](#)

[Network DMZ](#)

[SAP on Azure](#)

[SharePoint Server 2016](#)

[Cloud Design Patterns](#)

[Availability patterns](#)

[Data management patterns](#)

[Design and Implementation patterns](#)

[Management and Monitoring patterns](#)

[Messaging patterns](#)

[Performance and scalability patterns](#)

[Resiliency patterns](#)

Security patterns

Best Practices

API design

API implementation

Autoscaling

Background jobs

Caching

Content Delivery Network

Data partitioning

Monitoring and diagnostics

Naming conventions

Transient fault handling

Retry guidance for specific services

Performance Antipatterns

Busy Database

Busy Front End

Chatty I/O

Extraneous Fetching

Improper Instantiation

Monolithic Persistence

No Caching

Synchronous I/O

Design Review Checklists

Availability

DevOps

Resiliency (general)

Resiliency (Azure services)

Scalability

Design for Resiliency

Designing resilient applications

Resiliency checklist

Failure mode analysis

Scenario guides

[Azure for AWS Professionals](#)

[Build microservices on Azure](#)

[Manage identity in multitenant applications](#)

[Migrate to Service Fabric from Cloud Services](#)

[Migrate a Cloud Services application to Service Fabric](#)

[Refactor a Service Fabric application](#)

[Extend Azure Resource Manager template functionality](#)

Resources

[Azure Roadmap](#)

[Microsoft Trust Center for Azure](#)

[Icons and diagrams](#)

[Calendar of Azure updates](#)

[Training](#)

[SQLCAT blog](#)

[AzureCAT blog](#)

[White papers](#)

Azure Application Architecture Guide

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

Reference Architectures

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

Cloud Design Patterns

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



Building Microservices on Azure

This multi-part series takes you through the process of designing and building a microservices architecture on Azure. A reference implementation is included.

Azure Data Architecture Guide

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

Best Practices for Cloud Applications

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

Designing for Resiliency

Learn how to design resilient applications for Azure.

Azure Building Blocks

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

Design Review Checklists

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

Azure Virtual Datacenter

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

Azure for AWS Professionals

Leverage your AWS experiences in Microsoft Azure.

Performance Antipatterns

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

□

Run SharePoint Server 2016 on Azure

Deploy and run a high availability SharePoint Server 2016 farm on Azure.

Run SAP HANA on Azure

Deploy and run SAP NetWeaver and SAP HANA in a high availability environment on Azure.

Identity Management for Multitenant Applications

Understand the best practices for multitenancy, when using Azure AD for identity management.

Azure Customer Advisory Team

The AzureCAT team's blog

SQL Server Customer Advisory Team

The SQLCAT team's blog

The cloud presents a fundamental shift in the way that enterprises procure and utilize technology resources. In the past, enterprises assumed ownership and responsibility of all levels of technology from infrastructure to software. Now, the cloud offer the potential to transform the way enterprises utilize technology by provisioning and consuming resources as needed.

While the cloud offers nearly unlimited flexibility in terms of design choices, enterprises seek proven and consistent methodology for the adoption of cloud technologies. And, each enterprise has different goals and timelines for cloud adoption, making a one-size-fits-all approach to adoption nearly impossible.

The process of adopting cloud technologies is not a linear process, and this is especially true for large enterprises with many different teams. Some teams may be responsible for a large set of existing workloads and have a requirement to modernize them by adding cloud technologies or migrating them completely. Other teams may have the opportunity to innovate by beginning new development from scratch in the cloud. Yet other teams may not be ready to adopt cloud technologies in production but are ready to learn about and experiment with the cloud.

Each of these different teams requires different approaches to adopting the cloud, but the core knowledge necessary to begin the process is common to all. For this reason, this guide approaches enterprise cloud adoption from the perspective of organizational readiness. The stages of organizational readiness are:

1. **Foundational Azure adopters:** for enterprises with little to no experience with Azure. At the end of this stage, an enterprise is capable of deploying a basic Azure Web Site or a virtual network and an infrastructure as a service (IaaS) virtual machine.
2. **Intermediate Azure adopters:** for enterprises with foundational experience in Azure. At the end of this stage, an enterprise is capable of tracking costs and usage across multiple business units, connecting an on-premises network to an Azure virtual network, deploying single region n-tier workloads, and extending their security boundary to include Azure.
3. **Advanced Azure adopters:** for enterprises with intermediate experience in Azure. At the end of this stage, an enterprise is capable of managing multiple devops environments, implementing complex hybrid networking scenarios for increased security, making workloads highly available to multiple regions around the globe.
4. **Modernizing on-premises applications:** for enterprises with intermediate to advanced experience in Azure who want to integrate Azure technologies with on-premises work loads. At the end of this stage, an enterprise is capable of enumerating and stack-ranking on-premises workloads for migration, performing the migration, and integrating migrated workloads into on-premises IT management and monitoring processes.
5. **Optimizing migrated applications for the cloud:** for enterprises at the modernizing stage who want to efficiently use cloud resources. At the end of this stage, an enterprise is capable of evaluating Azure usage to identify unused or underutilized resources, scaling workloads, and scheduling workloads with instance reservation.
6. **Innovating in the cloud:** for enterprises with intermediate to advanced experience in Azure who want to develop native cloud applications from scratch. At the end of this stage, an enterprise is capable of developing and deploying resilient, highly available workloads that use native cloud services, microservices, and other cutting edge Azure services.

How to use the Azure Cloud Adoption Guide

The Azure Cloud Adoption Guide is organized into a series of stages. Each stage includes a list of articles. The articles fall into one of three categories: articles that **explain** how something works, articles that describe **how to** accomplish a specific task, or articles that provide **guidance** to assist you in making technology choices and design decisions.

The explainer articles provide background knowledge that makes it easier to understand how to accomplish a specific task. Some of the articles include a short explainer video that you can watch instead of reading the articles. As you work through the list of articles, when you reach a 'how to' document, follow the steps to accomplish the task before proceeding. The guidance articles provide best practice recommendations for the task.

The articles are listed in order, and they are progressive. If you are already familiar with the subject matter in a article, you can move on to the next article in the list.

Audience

The audience for the Azure Cloud Adoption Guide includes Enterprise Administrators, Finance, IT operations, IT security and

compliance, workload development owners, and workload operations owners.

Next steps

- If you are new to Azure, start with the [foundational Azure adoption](#) section of this guide.

Adopting Azure: Foundational

6/11/2018 • 2 minutes to read • [Edit Online](#)

Adopting Azure is the first stage of organizational maturity for an enterprise. By the end of this stage, people in your organization can deploy simple workloads to Azure.

The list below includes the tasks for completing the foundational adoption stage. The list is progressive so complete each task in order. If you have previously completed the task, move on the next task in the list.

1. Understand Azure internals:

- [Explainer: how does Azure work?](#)
- [Explainer: what is cloud resource governance?](#)

2. Understand enterprise digital identity in Azure:

- [Explainer: what is an Azure Active Directory Tenant?](#)
- [How to: get an Azure Active Directory Tenant](#)
- [Guidance: Azure AD tenant design guidance](#)
- [How to: add new users to Azure Active Directory](#)

3. Understand subscriptions in Azure:

- [Explainer: what is an Azure subscription?](#)
- [Guidance: Azure subscription design](#)

4. Understand resource management in Azure:

- [Explainer: what is Azure Resource Manager?](#)
- [Explainer: what is an Azure resource group?](#)
- [Explainer: understanding resource access in Azure](#)
- [How to: create an Azure resource group using the Azure portal](#)
- [Guidance: Azure resource group design guidance](#)
- [Guidance: naming conventions for Azure resources](#)

5. Deploy a basic Azure architecture:

- Learn about the different types of Azure compute options such as Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) in the [overview of Azure compute options](#).
- Now that you understand the different types of Azure compute options, pick either a web application (PaaS) or virtual machine (IaaS) as your first resource in Azure:
- PaaS: Introduction to Platform as a Service:
 - [How to: deploy a basic web application to Azure](#)
 - [Guidance: proven practices for deploying a basic web application to Azure](#)
- IaaS: Introduction to Virtual Networking:
 - [Explainer: Azure virtual network](#)
 - [How to: deploy a Virtual Network to Azure using the portal](#)
- IaaS: Deploy a single virtual machine(VM) workload (Windows and Linux):
 - [How to: deploy a Windows VM to Azure with the portal](#)
 - [Guidance: proven practices for running a Windows VM on Azure](#)
 - [How to: deploy a Linux VM to Azure with the portal](#)
 - [Guidance: proven practices for running a Linux VM on Azure](#)

Azure Cloud Adoption Guide: Intermediate Overview

6/19/2018 • 8 minutes to read • [Edit Online](#)

In the foundational adoption stage, you were introduced to the basic concepts of Azure resource governance. The foundational stage was designed to get you started with your Azure adoption journey, and it walked you through how to deploy a simple workload with a single small team. In reality, most large organizations have many teams that are working on many different workloads at the same time. As you would expect, a simple governance model is not sufficient to manage more complex organizational and development scenarios.

The focus of the intermediate stage of Azure adoption is designing your governance model for multiple teams working on multiple new Azure development workloads.

The audience for this stage of the guide is the following personas within your organization:

- *Finance*: owner of the financial commitment to Azure, responsible for developing policies and procedures for tracking resource consumption costs including billing and chargeback.
- *Central IT*: responsible for governing your organization's cloud resources including resource management and access, as well as workload health and monitoring.
- *Shared infrastructure owner*: technical roles responsible for network connectivity from on-premises to cloud.
- *Security operations*: responsible for implementing security policy necessary to extend on-premises security boundary to include Azure. May also own security infrastructure in Azure for storing secrets.
- *Workload owner*: responsible for publishing a workload to Azure. Depending on the structure of your organization's development teams, this could be a development lead, a program management lead, or build engineering lead. Part of the publishing process may include the deployment of resources to Azure.
 - *Workload contributor*: responsible for contributing to the publishing of a workload to Azure. May require read access to Azure resources for performance monitoring or tuning. Does not require permission to create, update, or delete resources.

Section 1: Azure concepts for multiple workloads and multiple teams

In the foundational adoption stage, you learned some basics about Azure internals and how resources are created, read, updated, and deleted. You also learned about identity and that Azure only trusts Azure Active Directory (AD) to authenticate and authorize users who need access to those resources.

You also started learning about how to configure Azure's governance tools to manage your organization's use of Azure resources. In the foundational stage we looked at how to govern a single team's access to the resources necessary to deploy a simple workload. In reality, your organization is going to be made up of multiple teams working on multiple workloads simultaneously.

Before we begin, let's take a look at what the term **workload** actually means. It's a term that is typically understood to define an arbitrary unit of functionality such as an application or service. We think about a workload in terms of the code artifacts that are deployed to a server as well as any other services, such as a database, that are necessary. This is a useful definition for an on-premises application or service but in the cloud we need to expand on it.

In the cloud, a workload not only encompasses all the artifacts but also includes the cloud resources as well. We include cloud resources as part of our definition because of a concept known as **infrastructure-as-code**. As you learned in the "how does Azure work" explainer, resources in Azure are deployed by an orchestrator service. The orchestrator service exposes this functionality through a web API, and this web API can be called using several tools such as Powershell, the Azure command line interface (CLI), and the Azure portal. This means that we can specify our resources in a machine-readable file that can be stored along with the code artifacts associated with our application.

This enables us to define a workload in terms of code artifacts and the necessary cloud resources, and this further enables us to **isolate** our workloads. Workloads may be isolated by the way resources are organized, by network topology, or by other attributes. The goal of workload isolation is to associate a workload's specific resources to a team so the team can independently manage all aspects of those resources. This enables multiple teams to share resource management services in Azure while preventing the unintentional deletion or modification of each other's resources.

This isolation also enables another concept known as **DevOps**. DevOps includes the software development practices that include both software development and IT operations above, but adds the use of automation as much as possible. One of the principles of DevOps is known as continuous integration and continuous delivery (CI/CD). Continuous integration refers to the automated build processes that are run each time a developer commits a code change, and continuous delivery refers to the automated processes that deploy this code to various **environments** such as a **development environment** for testing or a **production environment** for final deployment.

Section 2: Governance design for multiple teams and multiple workloads

In the [foundational stage](#) of the Azure cloud adoption guide, you were introduced to the concept of cloud governance. You learned how to design a simple governance model for a single team working on a single workload.

In the intermediate stage, the [governance design guide](#) expands on the foundational concepts to add multiple teams, multiple workloads, and multiple environments. Once you've gone through the examples in the document you can apply the design principles to designing and implementing your organization's governance model.

Section 3: Implementing a resource management model

Your organization's cloud governance model represents the intersection between Azure's resource access management tools, your people, and the access management rules you've defined. In the governance design guide, you learned about several different models for governing access to Azure resources. Now we'll walk through the steps necessary to implement the resource management model with one subscription for each of the **shared infrastructure**, **production**, and **development** environments from the design guide. We'll have one **subscription owner** for all three environments. Each workload will be isolated in a **resource group** with a **workload owner** added with the **contributor** role.

NOTE

Read [understanding resource access in Azure](#) to learn more about the relationship between Azure Accounts and subscriptions.

Follow these steps:

1. Create an [Azure account](#) if your organization doesn't already have one. The person who signs up for the Azure account becomes the Azure account administrator, and your organization's leadership must select an individual to assume this role. This individual will be responsible for:
 - Creating subscriptions, and
 - Creating and administering [Azure Active Directory \(AD\)](#) tenants that store user identity for those subscriptions.
2. Your organization's leadership team decides which people are responsible for:
 - Management of user identity; an [Azure AD tenant](#) is created by default when your organization's Azure Account is created, and the account administrator is added as the [Azure AD global administrator](#) by default. Your organization can choose another user to manage user identity by [assigning the Azure AD](#)

global administrator role to that user.

- Subscriptions, which means these users:
 - Manage costs associated with resource usage in that subscription,
 - Implement and maintain least permission model for resource access, and
 - Keep track of service limits.
- Shared infrastructure services (if your organization decides to use this model), which means this user is responsible for:
 - On-premises to Azure network connectivity, and
 - Ownership of network connectivity within Azure through virtual network peering.
- Workload owners.

3. The Azure AD global administrator [creates the new user accounts](#) for:

- The person who will be the **subscription owner** for each subscription associated with each environment.
Note that this is necessary only if the subscription **service administrator** will not be tasked with managing resource access for each subscription/environment.
- The person who will be the **network operations user**, and
- The people who are **workload owner(s)**.

4. The Azure account administrator creates the following three subscriptions using the [Azure account portal](#):

- A subscription for the **shared infrastructure** environment,
- A subscription for the **production** environment, and
- A subscription for the **development** environment.

5. The Azure account administrator [adds the subscription service owner to each subscription](#).

6. Create an approval process for **workload owners** to request the creation of resource groups. The approval process can be implemented in many ways, such as over email, or you can use a process management tool such as [Sharepoint workflows](#). The approval process can follow these steps:

- The **workload owner** prepares a bill of materials for required Azure resources in either the **development** environment, **production** environment, or both, and submits it to the **subscription owner**.
- The **subscription owner** reviews the bill of materials and validates the requested resources to ensure that the requested resources are appropriate for their planned use - for example, checking that the requested [virtual machine sizes](#) are correct.
- If the request is not approved, the **workload owner** is notified. If the request is approved, the **subscription owner** [creates the requested resource group](#) following your organization's [naming conventions](#), [adds the workload owner](#) with the **contributor role** and sends notification to the **workload owner** that the resource group has been created.

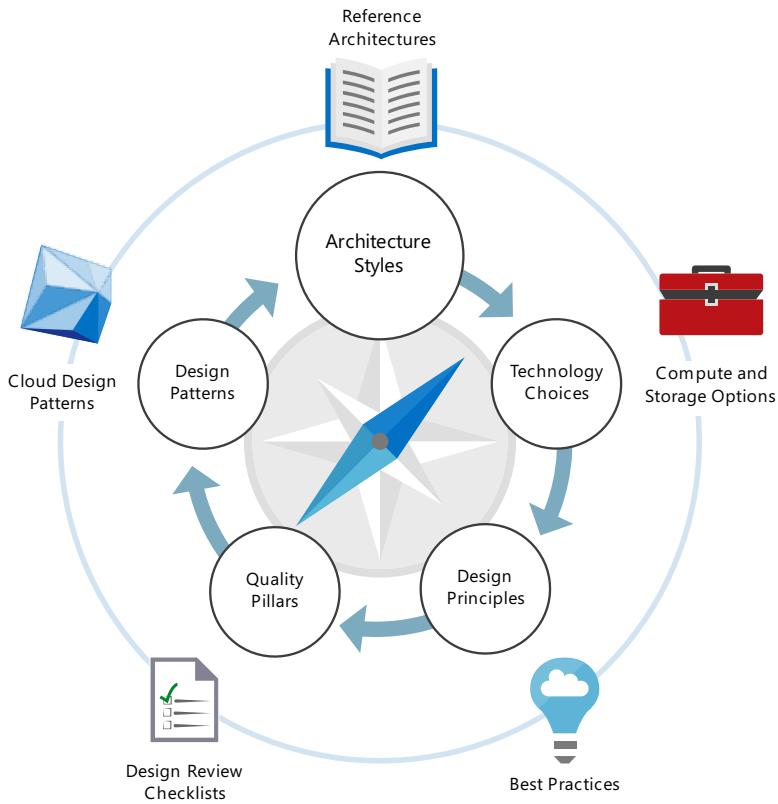
7. Create an approval process for workload owners to request a virtual network peering connection from the shared infrastructure owner. As with the previous step, this approval process can be implemented using email or a process management tool.

Now that you've implemented your governance model, you can deploy your shared infrastructure services.

Section 4: deploy shared infrastructure services

There are several [hybrid network reference architectures](#) that your organization can use to connect your on-premises network to Azure. Each of these reference architectures includes a deployment that requires a subscription identifier. During deployment, specify the subscription identifier for the subscription associated with your **shared infrastructure** environment. You will also need to edit the template files to specify the resource group that is managed by your **network operations** user, or, you can use the default resource groups in the deployment and add the **network operations** user with the **contributor** role to them.

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



Introduction

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

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

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

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

How this guide is structured

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

Architecture styles

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

Learn more:

- [Architecture styles](#)
- [Azure reference architectures](#)

Technology choices

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

Learn more:

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

Design principles

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

Learn more:

- [Design principles for Azure applications](#)
- [Best practices when building for the cloud](#)

Quality pillars

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

Learn more:

- [Pillars of software quality](#)
- [Design review checklists](#)

Cloud design patterns

Design patterns are general solutions to common software design problem. We have identified a set of design patterns that are especially useful when designing distributed applications for the cloud.

Learn more:

- [Catalog of cloud design patterns](#)

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

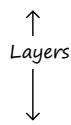
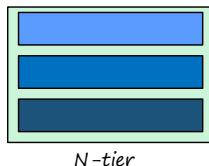
We have identified a set of architecture styles that are commonly found in cloud applications. The article for each style includes:

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

A quick tour of the styles

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

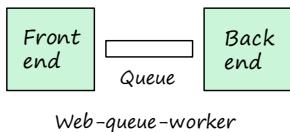
N-tier



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

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

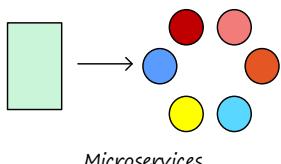
Web-Queue-Worker



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

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

Microservices

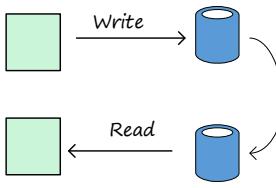


If your application has a more complex domain, consider moving to a [Microservices](#) architecture. A microservices application is composed of many small, independent services. Each service implements a single business capability. Services are loosely coupled, communicating through API contracts.

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

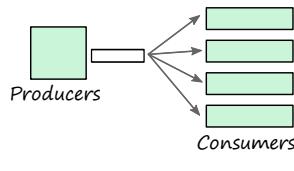
CQRS

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



CQRS makes the most sense when it's applied to a subsystem of a larger architecture. Generally, you shouldn't impose it across the entire application, as that will just create unneeded complexity. Consider it for collaborative domains where many users access the same data.

Event-driven architecture



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

Consider an event-driven architecture for applications that ingest and process a large volume of data with very low latency, such as IoT

solutions. The style is also useful when different subsystems must perform different types of processing on the same event data.

Big Data, Big Compute

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

Architecture styles as constraints

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

For example, the constraints in microservices include:

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

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

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

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

ARCHITECTURE STYLE	DEPENDENCY MANAGEMENT	DOMAIN TYPE
N-tier	Horizontal tiers divided by subnet	Traditional business domain. Frequency of updates is low.
Web-Queue-Worker	Front and backend jobs, decoupled by async messaging.	Relatively simple domain with some resource intensive tasks.
Microservices	Vertically (functionally) decomposed services that call each other through APIs.	Complicated domain. Frequent updates.
CQRS	Read/write segregation. Schema and scale are optimized separately.	Collaborative domain where lots of users access the same data.

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

Consider challenges and benefits

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

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

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

Overview of Azure compute options

6/15/2018 • 3 minutes to read • [Edit Online](#)

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

Overview

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

Platform-as-a-Service (PaaS) provides a managed hosting environment, where you can deploy your application without needing to manage VMs or networking resources. For example, instead of creating individual VMs, you specify an instance count, and the service will provision, configure, and manage the necessary resources. Azure App Service is an example of a PaaS service.

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

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

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

Azure compute options

Here are the main compute options currently available in Azure:

- [Virtual Machines](#) are an IaaS service, allowing you to deploy and manage VMs inside a virtual network (VNet).
- [App Service](#) is a managed PaaS offering for hosting web apps, mobile app back ends, RESTful APIs, or automated business processes.
- [Service Fabric](#) is a distributed systems platform that can run in many environments, including Azure or on premises. Service Fabric is an orchestrator of microservices across a cluster of machines.
- [Azure Container Service](#) lets you create, configure, and manage a cluster of VMs that are preconfigured to run containerized applications.
- [Azure Container Instances](#) offer the fastest and simplest way to run a container in Azure, without having to provision any virtual machines and without having to adopt a higher-level service.
- [Azure Functions](#) is a managed FaaS service.
- [Azure Batch](#) is a managed service for running large-scale parallel and high-performance computing (HPC) applications.
- [Cloud Services](#) is a managed service for running cloud applications. It uses a PaaS hosting model.

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

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

Next steps

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

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

Choose the right data store

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

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

This heterogeneity means that a single data store is usually not the best approach. Instead, it's often better to store different types of data in different data stores, each focused towards a specific workload or usage pattern. The term *polyglot persistence* is used to describe solutions that use a mix of data store technologies.

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

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

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

Relational database management systems

Relational databases organize data as a series of two-dimensional tables with rows and columns. Each table has its own columns, and every row in a table has the same set of columns. This model is mathematically based, and most vendors provide a dialect of the Structured Query Language (SQL) for retrieving and managing data. An RDBMS typically implements a transactionally consistent mechanism that conforms to the ACID (Atomic, Consistent, Isolated, Durable) model for updating information.

An RDBMS typically supports a schema-on-write model, where the data structure is defined ahead of time, and all read or write operations must use the schema. This is in contrast to most NoSQL data stores, particularly key/value types, where the schema-on-read model assumes that the client will be imposing its own interpretive schema on data coming out of the database, and is agnostic to the data format being written.

An RDBMS is very useful when strong consistency guarantees are important — where all changes are atomic, and transactions always leave the data in a consistent state. However, the underlying structures do not lend themselves to scaling out by distributing storage and processing across machines. Also, information stored in an RDBMS, must be put into a relational structure by following the normalization process. While this process is well understood, it can lead to inefficiencies, because of the need to disassemble logical entities into rows in separate tables, and then reassemble the data when running queries.

Relevant Azure service:

- [Azure SQL Database](#)

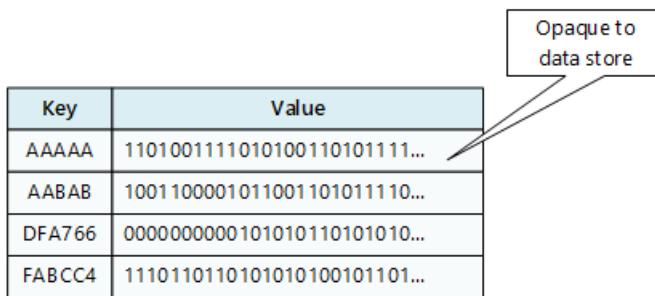
- [Azure Database for MySQL](#)
- [Azure Database for PostgreSQL](#)

Key/value stores

A key/value store is essentially a large hash table. You associate each data value with a unique key, and the key/value store uses this key to store the data by using an appropriate hashing function. The hashing function is selected to provide an even distribution of hashed keys across the data storage.

Most key/value stores only support simple query, insert, and delete operations. To modify a value (either partially or completely), an application must overwrite the existing data for the entire value. In most implementations, reading or writing a single value is an atomic operation. If the value is large, writing may take some time.

An application can store arbitrary data as a set of values, although some key/value stores impose limits on the maximum size of values. The stored values are opaque to the storage system software. Any schema information must be provided and interpreted by the application. Essentially, values are blobs and the key/value store simply retrieves or stores the value by key.



The diagram shows a table with four rows. The first row has a blue header with 'Key' and 'Value'. The subsequent three rows have black headers. The 'Key' column contains binary strings: 'AAAAAA', 'AABAB', 'DFA766', and 'FABCC4'. The 'Value' column contains binary strings: '1101001111010100110101111...', '1001100001011001101011110...', '0000000000101010110101010...', and '1110110110101010100101101...'. A callout box with the text 'Opaque to data store' points to the 'Value' column.

Key	Value
AAAAAA	1101001111010100110101111...
AABAB	1001100001011001101011110...
DFA766	0000000000101010110101010...
FABCC4	1110110110101010100101101...

Key/value stores are highly optimized for applications performing simple lookups, but are less suitable for systems that need to query data across different key/value stores. Key/value stores are also not optimized for scenarios where querying by value is important, rather than performing lookups based only on keys. For example, with a relational database, you can find a record by using a WHERE clause, but key/values stores usually do not have this type of lookup capability for values.

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

Relevant Azure services:

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

Document databases

A document database is conceptually similar to a key/value store, except that it stores a collection of named fields and data (known as documents), each of which could be simple scalar items or compound elements such as lists and child collections. The data in the fields of a document can be encoded in a variety of ways, including XML, YAML, JSON, BSON, or even stored as plain text. Unlike key/value stores, the fields in documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

Typically, a document contains the entire data for an entity. What items constitute an entity are application specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document may contain information that would be spread across several relational tables in an RDBMS.

A document store does not require that all documents have the same structure. This free-form approach provides a great deal of flexibility. Applications can store different data in documents as business requirements change.

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

The application can retrieve documents by using the document key. This is a unique identifier for the document, which is often hashed, to help distribute data evenly. Some document databases create the document key automatically. Others enable you to specify an attribute of the document to use as the key. The application can also query documents based on the value of one or more fields. Some document databases support indexing to facilitate fast lookup of documents based on one or more indexed fields.

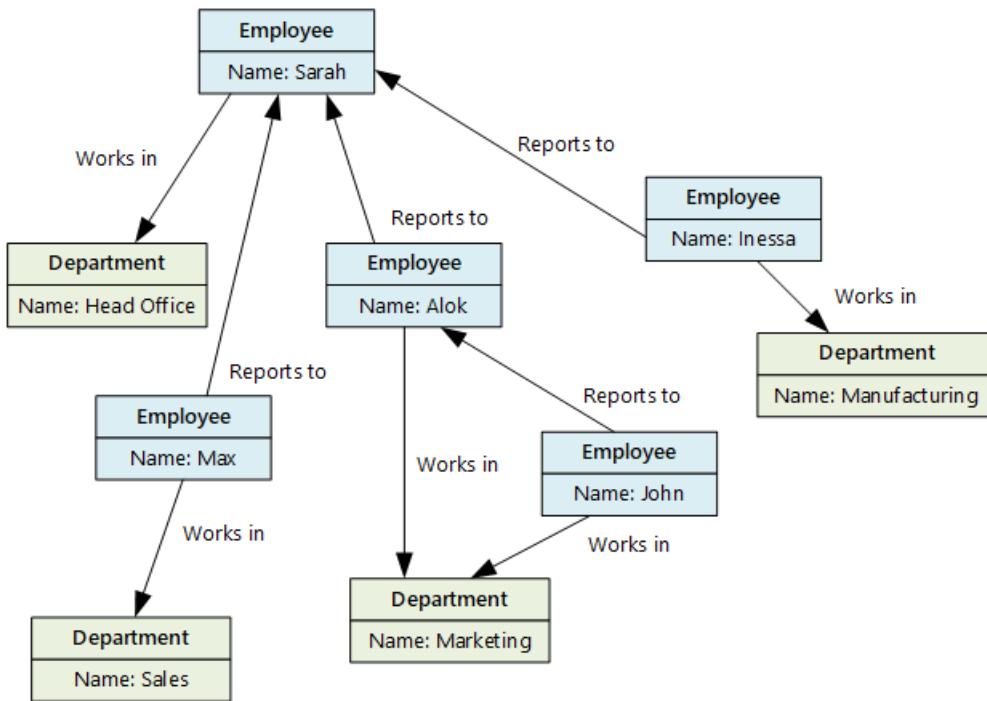
Many document databases support in-place updates, enabling an application to modify the values of specific fields in a document without rewriting the entire document. Read and write operations over multiple fields in a single document are usually atomic.

Relevant Azure service: [Cosmos DB](#)

Graph databases

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

The purpose of a graph database is to allow an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. The following diagram shows an organization's personnel database structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.



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

Relevant Azure service: [Cosmos DB](#)

Column-family databases

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

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

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

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

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

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

family. Indexes let you retrieve data by columns value, rather than row key.

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

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

Data analytics

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

Relevant Azure services:

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

Search Engine Databases

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

The key characteristics of a search engine database are the ability to store and index information very quickly, and provide fast response times for search requests. Indexes can be multi-dimensional and may support free-text searches across large volumes of text data. Indexing can be performed using a pull model, triggered by the search engine database, or using a push model, initiated by external application code.

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

Relevant Azure service: [Azure Search](#)

Time Series Databases

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

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

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

Object storage

Object storage is optimized for storing and retrieving large binary objects (images, files, video and audio streams, large application data objects and documents, virtual machine disk images). Objects in these store types are composed of the stored data, some metadata, and a unique ID for accessing the object. Object stores enables the management of extremely large amounts of unstructured data.

Relevant Azure service: [Blob Storage](#)

Shared files

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

Relevant Azure service: [File Storage](#)

Design principles for Azure applications

6/15/2018 • 2 minutes to read • [Edit Online](#)

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

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

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

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

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

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

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

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

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

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

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

Pillars of software quality

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

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

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

Scalability

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

Horizontal scaling has significant advantages over vertical scaling:

- True cloud scale. Applications can be designed to run on hundreds or even thousands of nodes, reaching scales that are not possible on a single node.
- Horizontal scale is elastic. You can add more instances if load increases, or remove them during quieter periods.
- Scaling out can be triggered automatically, either on a schedule or in response to changes in load.
- Scaling out may be cheaper than scaling up. Running several small VMs can cost less than a single large VM.
- Horizontal scaling can also improve resiliency, by adding redundancy. If an instance goes down, the application keeps running.

An advantage of vertical scaling is that you can do it without making any changes to the application. But at some point you'll hit a limit, where you can't scale any up any more. At that point, any further scaling must be horizontal.

Horizontal scale must be designed into the system. For example, you can scale out VMs by placing them behind a load balancer. But each VM in the pool must be able to handle any client request, so the application must be stateless or store state externally (say, in a distributed cache). Managed PaaS services often have horizontal scaling and auto-scaling built in. The ease of scaling these services is a major advantage of using PaaS services.

Just adding more instances doesn't mean an application will scale, however. It might simply push the bottleneck somewhere else. For example, if you scale a web front-end to handle more client requests, that might trigger lock contentions in the database. You would then need to consider additional measures, such as optimistic concurrency or data partitioning, to enable more throughput to the database.

Always conduct performance and load testing to find these potential bottlenecks. The stateful parts of a system,

such as databases, are the most common cause of bottlenecks, and require careful design to scale horizontally. Resolving one bottleneck may reveal other bottlenecks elsewhere.

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

Scalability guidance

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

Availability

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

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

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

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

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

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

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

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

Availability guidance

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

Resiliency

Resiliency is the ability of the system to recover from failures and continue to function. The goal of resiliency is to return the application to a fully functioning state after a failure occurs. Resiliency is closely related to availability.

In traditional application development, there has been a focus on reducing mean time between failures (MTBF). Effort was spent trying to prevent the system from failing. In cloud computing, a different mindset is required, due to several factors:

- Distributed systems are complex, and a failure at one point can potentially cascade throughout the system.
- Costs for cloud environments are kept low through the use of commodity hardware, so occasional hardware failures must be expected.
- Applications often depend on external services, which may become temporarily unavailable or throttle high-volume users.
- Today's users expect an application to be available 24/7 without ever going offline.

All of these factors mean that cloud applications must be designed to expect occasional failures and recover from them. Azure has many resiliency features already built into the platform. For example,

- Azure Storage, SQL Database, and Cosmos DB all provide built-in data replication, both within a region and across regions.
- Azure Managed Disks are automatically placed in different storage scale units, to limit the effects of hardware failures.
- VMs in an availability set are spread across several fault domains. A fault domain is a group of VMs that share a common power source and network switch. Spreading VMs across fault domains limits the impact of physical hardware failures, network outages, or power interruptions.

That said, you still need to build resiliency into your application. Resiliency strategies can be applied at all levels of the architecture. Some mitigations are more tactical in nature — for example, retrying a remote call after a transient network failure. Other mitigations are more strategic, such as failing over the entire application to a secondary region. Tactical mitigations can make a big difference. While it's rare for an entire region to experience a disruption, transient problems such as network congestion are more common — so target these first. Having the right monitoring and diagnostics is also important, both to detect failures when they happen, and to find the root causes.

When designing an application to be resilient, you must understand your availability requirements. How much downtime is acceptable? This is partly a function of cost. How much will potential downtime cost your business? How much should you invest in making the application highly available?

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

Resiliency guidance

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

Management and DevOps

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

Deployments must be reliable and predictable. They should be automated to reduce the chance of human error. They should be a fast and routine process, so they don't slow down the release of new features or bug fixes. Equally important, you must be able to quickly roll back or roll forward if an update has problems.

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

control of the infrastructure or, in some cases, the operating system. In a large application, it's not practical to log into VMs to troubleshoot an issue or sift through log files. With PaaS services, there may not even be a dedicated VM to log into. Monitoring and diagnostics give insight into the system, so that you know when and where failures occur. All systems must be observable. Use a common and consistent logging schema that lets you correlate events across systems.

The monitoring and diagnostics process has several distinct phases:

- Instrumentation. Generating the raw data, from application logs, web server logs, diagnostics built into the Azure platform, and other sources.
- Collection and storage. Consolidating the data into one place.
- Analysis and diagnosis. To troubleshoot issues and see the overall health.
- Visualization and alerts. Using telemetry data to spot trends or alert the operations team.

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

Management and DevOps guidance

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

Security

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

Here are some broad security areas to consider.

Identity management

Consider using Azure Active Directory (Azure AD) to authenticate and authorize users. Azure AD is a fully managed identity and access management service. You can use it to create domains that exist purely on Azure, or integrate with your on-premises Active Directory identities. Azure AD also integrates with Office365, Dynamics CRM Online, and many third-party SaaS applications. For consumer-facing applications, Azure Active Directory B2C lets users authenticate with their existing social accounts (such as Facebook, Google, or LinkedIn), or create a new user account that is managed by Azure AD.

If you want to integrate an on-premises Active Directory environment with an Azure network, several approaches are possible, depending on your requirements. For more information, see our [Identity Management](#) reference architectures.

Protecting your infrastructure

Control access to the Azure resources that you deploy. Every Azure subscription has a [trust relationship](#) with an Azure AD tenant. Use [Role-Based Access Control](#) (RBAC) to grant users within your organization the correct permissions to Azure resources. Grant access by assigning RBAC role to users or groups at a certain scope. The scope can be a subscription, a resource group, or a single resource. [Audit](#) all changes to infrastructure.

Application security

In general, the security best practices for application development still apply in the cloud. These include things like using SSL everywhere, protecting against CSRF and XSS attacks, preventing SQL injection attacks, and so on.

Cloud applications often use managed services that have access keys. Never check these into source control. Consider storing application secrets in Azure Key Vault.

Data sovereignty and encryption

Make sure that your data remains in the correct geopolitical zone when using Azure's highly available. Azure's geo-replicated storage uses the concept of a [paired region](#) in the same geopolitical region.

Use Key Vault to safeguard cryptographic keys and secrets. By using Key Vault, you can encrypt keys and secrets by using keys that are protected by hardware security modules (HSMs). Many Azure storage and DB services support data encryption at rest, including [Azure Storage](#), [Azure SQL Database](#), [Azure SQL Data Warehouse](#), and [Cosmos DB](#).

Security resources

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

Cloud Design Patterns

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

These design patterns are useful for building reliable, scalable, secure applications in the cloud.

Each pattern describes the problem that the pattern addresses, considerations for applying the pattern, and an example based on Microsoft Azure. Most of the patterns include code samples or snippets that show how to implement the pattern on Azure. However, most of the patterns are relevant to any distributed system, whether hosted on Azure or on other cloud platforms.

Challenges in cloud development

Availability



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

Data Management



Data management is the key element of cloud applications, and influences most of the quality attributes. Data is typically hosted in different locations and across multiple servers for reasons such as performance, scalability or availability, and this can present a range of challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations.

Design and Implementation



Good design encompasses factors such as consistency and coherence in component design and deployment, maintainability to simplify administration and development, and reusability to allow components and subsystems to be used in other applications and in other scenarios. Decisions made during the design and implementation phase have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.

Messaging



The distributed nature of cloud applications requires a messaging infrastructure that connects the components and services, ideally in a loosely coupled manner in order to maximize scalability. Asynchronous messaging is widely used, and provides many benefits, but also brings challenges such as the ordering of messages, poison message management, idempotency, and more.

Management and Monitoring



Cloud applications run in in a remote datacenter where you do not have full control of the infrastructure or, in some cases, the operating system. This can make management and monitoring more difficult than an on-premises deployment. Applications must expose runtime information that administrators and operators can use to manage and monitor the system, as well as supporting changing business requirements and customization without requiring the application to be stopped or redeployed.

	<p>Performance and Scalability</p> <p>Performance is an indication of the responsiveness of a system to execute any action within a given time interval, while scalability is ability of a system either to handle increases in load without impact on performance or for the available resources to be readily increased. Cloud applications typically encounter variable workloads and peaks in activity. Predicting these, especially in a multi-tenant scenario, is almost impossible. Instead, applications should be able to scale out within limits to meet peaks in demand, and scale in when demand decreases. Scalability concerns not just compute instances, but other elements such as data storage, messaging infrastructure, and more.</p>
	<p>Resiliency</p> <p>Resiliency is the ability of a system to gracefully handle and recover from failures. The nature of cloud hosting, where applications are often multi-tenant, use shared platform services, compete for resources and bandwidth, communicate over the Internet, and run on commodity hardware means there is an increased likelihood that both transient and more permanent faults will arise. Detecting failures, and recovering quickly and efficiently, is necessary to maintain resiliency.</p>
	<p>Security</p> <p>Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. Cloud applications are exposed on the Internet outside trusted on-premises boundaries, are often open to the public, and may serve untrusted users. Applications must be designed and deployed in a way that protects them from malicious attacks, restricts access to only approved users, and protects sensitive data.</p>

Catalog of patterns

PATTERN	SUMMARY
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
Backends for Frontends	Create separate backend services to be consumed by specific frontend applications or interfaces.
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Cache-Aside	Load data on demand into a cache from a data store
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation.
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel.

PATTERN	SUMMARY
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Federated Identity	Delegate authentication to an external identity provider.
Gatekeeper	Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed.

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

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

Introduction

The cloud is changing the way applications are designed, including how data is processed and stored. Instead of a single general-purpose database that handles all of a solution's data, *Polyglot persistence* solutions use multiple, specialized data stores, each optimized to provide specific capabilities. The perspective on data in the solution changes as a result. There are no longer multiple layers of business logic that read and write to a single data layer. Instead, solutions are designed around a *data pipeline* that describes how data flows through a solution, where it is processed, where it is stored, and how it is consumed by the next component in the pipeline.

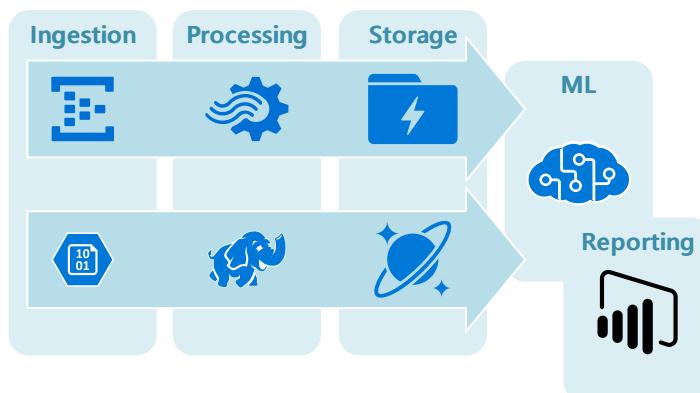
How this guide is structured

This guide is structured around two general categories of data solution, *Traditional RDBMS workloads* and *big data solutions*.

Traditional RDBMS workloads. These workloads include online transaction processing (OLTP) and online analytical processing (OLAP). Data in OLTP systems is typically relational data with a pre-defined schema and a set of constraints to maintain referential integrity. Often, data from multiple sources in the organization may be consolidated into a data warehouse, using an ETL process to move and transform the source data.



Big data solutions. A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems. The data may be processed in batch or in real time. Big data solutions typically involve a large amount of non-relational data, such as key-value data, JSON documents, or time series data. Often traditional RDBMS systems are not well-suited to store this type of data. The term *NoSQL* refers to a family of databases designed to hold non-relational data. (The term isn't quite accurate, because many non-relational data stores support SQL compatible queries.)

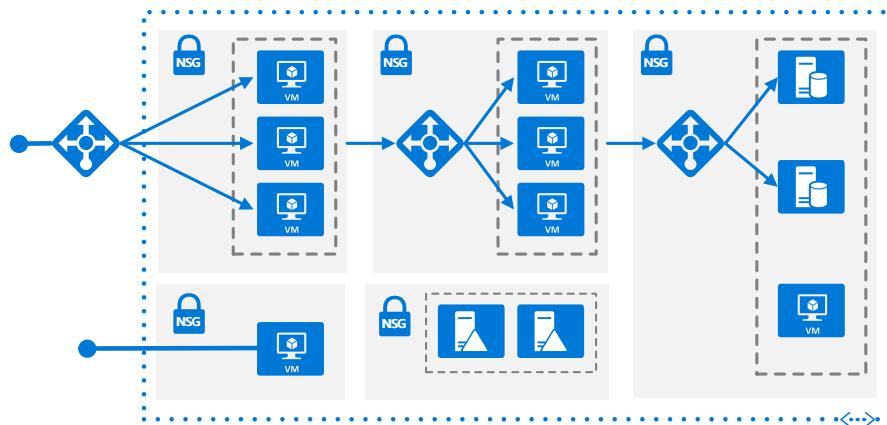


These two categories are not mutually exclusive, and there is overlap between them, but we feel that it's a useful way to frame the discussion. Within each category, the guide discusses **common scenarios**, including relevant Azure services and the appropriate architecture for the scenario. In addition, the guide compares **technology choices** for data solutions in Azure, including open source options. Within each category, we describe the key selection criteria and a capability matrix, to help you choose the right technology for your scenario.

This guide is not intended to teach you data science or database theory — you can find entire books on those subjects. Instead, th

goal is to help you select the right data architecture or data pipeline for your scenario, and then select the Azure services and technologies that best fit your requirements. If you already have an architecture in mind, you can skip directly to the technology choices.

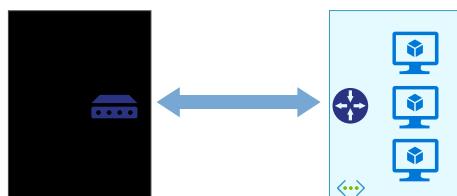
Our reference architectures are arranged by scenario, with related architectures grouped together. Each architecture includes recommended practices, along with considerations for scalability, availability, manageability, and security. Most also include a deployable solution.



N-tier application

Deploy an N-tier application to Azure, for Windows or Linux.

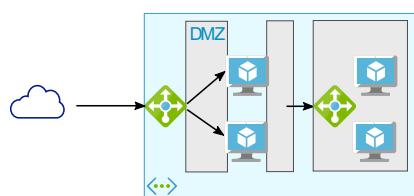
Configurations are shown for SQL Server and Apache Cassandra. For high availability, deploy an active-passive configuration in two regions.



Hybrid network

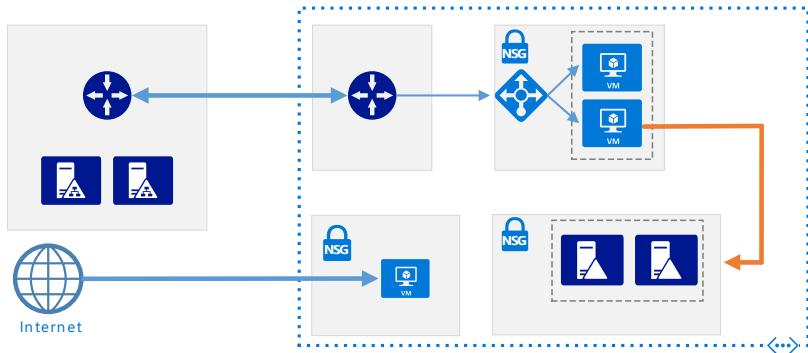
This series shows options for creating a network connection between an on-premises network and Azure.

Configurations include site-to-site VPN or Azure ExpressRoute for a private, dedicated connection.



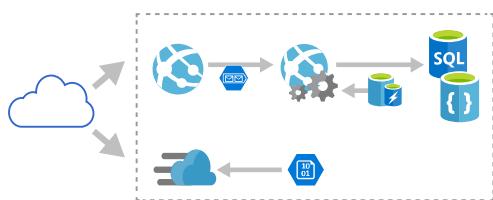
Network DMZ

This series shows how to create a network DMZ to protect the boundary between an Azure virtual network and an on-premises network or the Internet.



Identity management

This series shows options for integrating your on-premises Active Directory (AD) environment with an Azure network.



App Service web application

This series shows best practices for web applications that use Azure App Service.



Jenkins build server

Deploy and operate a scalable, enterprise-grade Jenkins server on Azure.

SharePoint Server 2016 farm

Deploy and run a high availability SharePoint Server 2016 farm on Azure with SQL Server Always On Availability Groups.



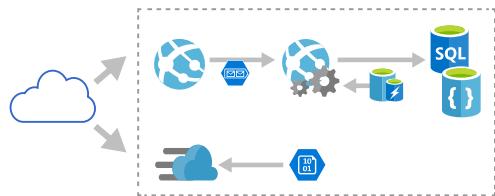
Run SAP on Azure

Deploy and run SAP in a high availability environment on Azure.

These reference architectures show proven practices for web applications that use Azure App Service and other managed service in Azure.

Basic web application

A basic web application that uses Azure App Service and Azure SQL Database.



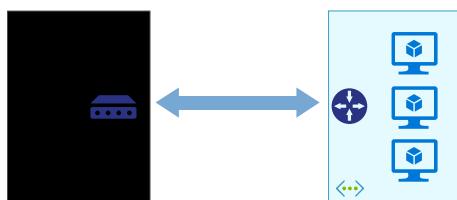
Improved scalability

Improve scalability and performance by adding cache, CDN, and WebJobs for background tasks.

Run in multiple regions

Run a web application in multiple regions to achieve high availability.

These reference architectures show proven practices for creating a robust network connection between an on-premises network and Azure. [Which should I choose?](#)



VPN

Extend an on-premises network to Azure using a site-to-site virtual private network (VPN).

ExpressRoute

Extend an on-premises network to Azure using Azure ExpressRoute.

ExpressRoute with VPN failover

Extend an on-premises network to Azure using Azure ExpressRoute, with a VPN as a failover connection.

Hub-spoke topology

The hub is a central point of connectivity to your on-premises network. The spokes are VNets that peer with the hub, and can be used to isolate workloads.

Hub-spoke topology with shared services

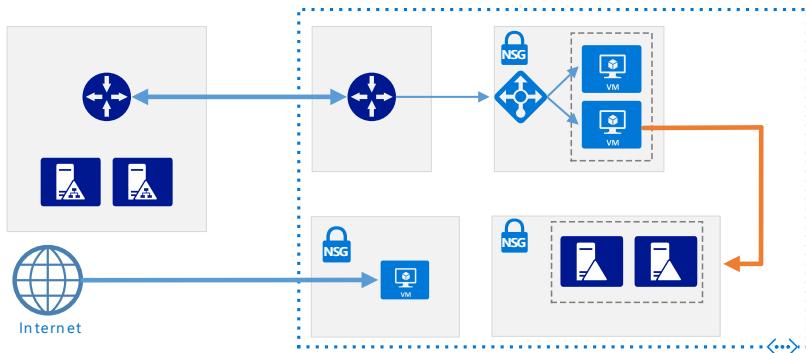
Deploy a hub-spoke topology that includes shared services, including Active Directory services and a network virtual appliance (NVA). Shared services can be consumed by all spokes.

These reference architectures show options for integrating your on-premises Active Directory (AD) environment with an Azure network.

Which should I choose?

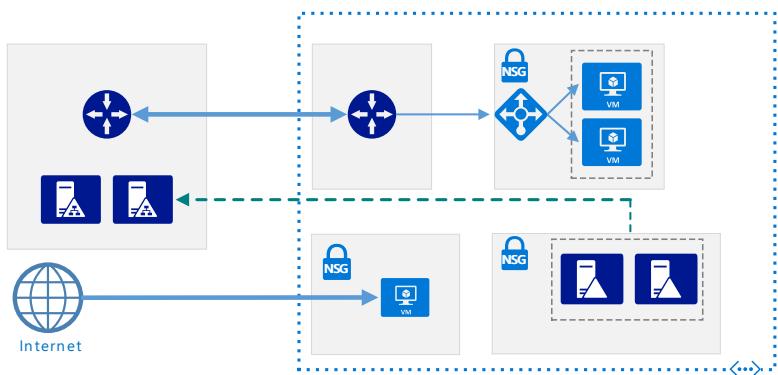
Integrate with Azure Active Directory

Integrate on-premises Active Directory domains and forests with Azure AD.



Extend AD DS to Azure

Extend your Active Directory environment to Azure using Active Directory Domain Services.



Create an AD DS forest in Azure

Create a separate AD domain in Azure that is trusted by domains in your on-premises forest.

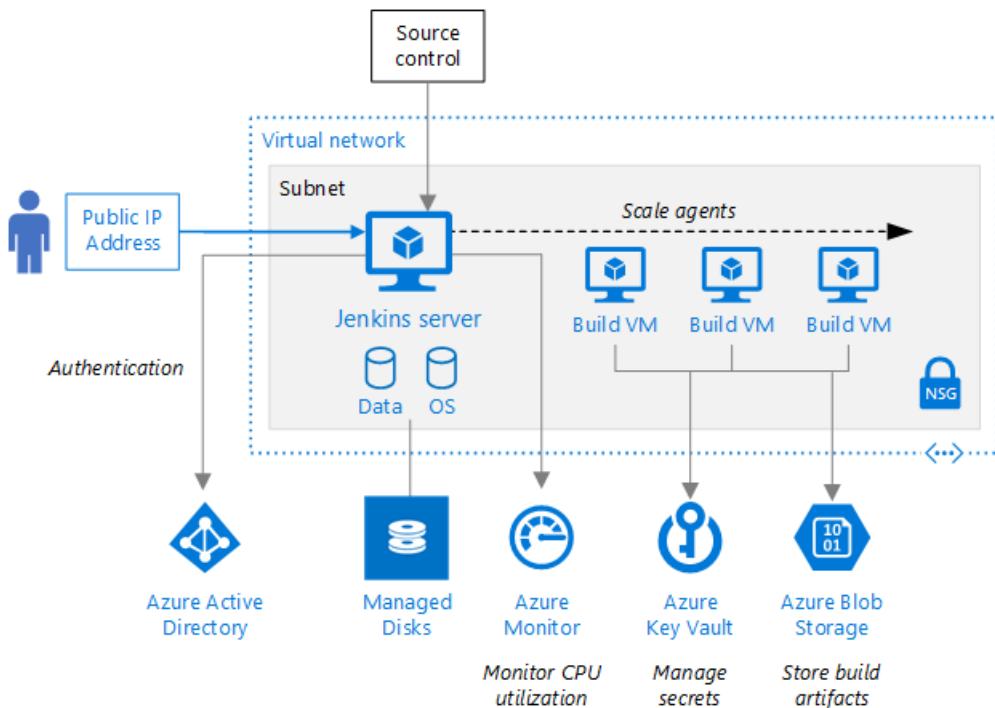
Extend AD FS to Azure

Use Active Directory Federation Services for federated authentication and authorization in Azure.

Run a Jenkins server on Azure

5/2/2018 • 12 minutes to read • [Edit Online](#)

This reference architecture shows how to deploy and operate a scalable, enterprise-grade Jenkins server on Azure secured with single sign-on (SSO). The architecture also uses Azure Monitor to monitor the state of the Jenkins server. [Deploy this solution.](#)



Download a [Visio file](#) that contains this architecture diagram.

This architecture supports disaster recovery with Azure services but does not cover more advanced scale-out scenarios involving multiple masters or high availability (HA) with no downtime. For general insights about the various Azure components, including a step-by-step tutorial about building out a CI/CD pipeline on Azure, see [Jenkins on Azure](#).

The focus of this document is on the core Azure operations needed to support Jenkins, including the use of Azure Storage to maintain build artifacts, the security items needed for SSO, other services that can be integrated, and scalability for the pipeline. The architecture is designed to work with an existing source control repository. For example, a common scenario is to start Jenkins jobs based on GitHub commits.

Architecture

The architecture consists of the following components:

- **Resource group.** A [resource group](#) is used to group Azure assets so they can be managed by lifetime, owner, and other criteria. Use resource groups to deploy and monitor Azure assets as a group and track billing costs by resource group. You can also delete resources as a set, which is very useful for test deployments.
- **Jenkins server.** A virtual machine is deployed to run [Jenkins](#) as an automation server and serve as Jenkins Master. This reference architecture uses the [solution template for Jenkins on Azure](#), installed on a Linux (Ubuntu 16.04 LTS) virtual machine on Azure. Other Jenkins offerings are available in the Azure Marketplace.

NOTE

Nginx is installed on the VM to act as a reverse proxy to Jenkins. You can configure Nginx to enable SSL for the Jenkins server.

- **Virtual network.** A [virtual network](#) connects Azure resources to each other and provides logical isolation. In this architecture, the Jenkins server runs in a virtual network.
- **Subnets.** The Jenkins server is isolated in a [subnet](#) to make it easier to manage and segregate network traffic without impacting performance.
- **NSGs.** Use [network security groups](#) (NSGs) to restrict network traffic from the Internet to the subnet of a virtual network.
- **Managed disks.** A [managed disk](#) is a persistent virtual hard disk (VHD) used for application storage and also to maintain the state of the Jenkins server and provide disaster recovery. Data disks are stored in Azure Storage. For high performance, [premium storage](#) is recommended.
- **Azure Blob Storage.** The [Windows Azure Storage plugin](#) uses Azure Blob Storage to store the build artifacts that are created and shared with other Jenkins builds.
- **Azure Active Directory (Azure AD).** [Azure AD](#) supports user authentication, allowing you to set up SSO. Azure AD [service principals](#) define the policy and permissions for each role authorization in the workflow, using [role-based access control](#) (RBAC). Each service principal is associated with a Jenkins job.
- **Azure Key Vault.** To manage secrets and cryptographic keys used to provision Azure resources when secrets are required, this architecture uses [Key Vault](#). For added help storing secrets associated with the application in the pipeline, see also the [Azure Credentials](#) plugin for Jenkins.
- **Azure monitoring services.** This service [monitors](#) the Azure virtual machine hosting Jenkins. This deployment monitors the virtual machine status and CPU utilization and sends alerts.

Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

Azure AD

The [Azure AD](#) tenant for your Azure subscription is used to enable SSO for Jenkins users and set up [service principals](#) that enable Jenkins jobs to access Azure resources.

SSO authentication and authorization are implemented by the Azure AD plugin installed on the Jenkins server. SSO allows you to authenticate using your organization credentials from Azure AD when logging on to the Jenkins server. When configuring the Azure AD plugin, you can specify the level of a user's authorized access to the Jenkins server.

To provide Jenkins jobs with access to Azure resources, an Azure AD administrator creates service principals. These grant applications—in this case, the Jenkins jobs—[authenticated, authorized access](#) to Azure resources.

[RBAC](#) further defines and controls access to Azure resources for users or service principals through their assigned role. Both built-in and custom roles are supported. Roles also help secure the pipeline and ensure that a user's or agent's responsibilities are assigned and authorized correctly. In addition, RBAC can be set up to limit access to Azure assets. For example, a user can be limited to working with only the assets in a particular resource group.

Storage

Use the Jenkins [Windows Azure Storage plugin](#), which is installed from the Azure Marketplace, to store build

artifacts that can be shared with other builds and tests. An Azure Storage account must be configured before this plugin can be used by the Jenkins jobs.

Jenkins Azure plugins

The solution template for Jenkins on Azure installs several Azure plugins. The Azure DevOps Team builds and maintains the solution template and the following plugins, which work with other Jenkins offerings in Azure Marketplace as well as any Jenkins master set up on premises:

- [Azure AD plugin](#) allows the Jenkins server to support SSO for users based on Azure AD.
- [Azure VM Agents](#) plugin uses an Azure Resource Manager template to create Jenkins agents in Azure virtual machines.
- [Azure Credentials](#) plugin allows you to store Azure service principal credentials in Jenkins.
- [Windows Azure Storage plugin](#) uploads build artifacts to, or downloads build dependencies from, [Azure Blob storage](#).

We also recommend reviewing the growing list of all available Azure plugins that work with Azure resources. To see all the latest list, visit [Jenkins Plugin Index](#) and search for Azure. For example, the following plugins are available for deployment:

- [Azure Container Agents](#) helps you to run a container as an agent in Jenkins.
- [Kubernetes Continuous Deploy](#) deploys resource configurations to a Kubernetes cluster.
- [Azure Container Service](#) deploys configurations to Azure Container Service with Kubernetes, DC/OS with Marathon, or Docker Swarm.
- [Azure Functions](#) deploys your project to Azure Function.
- [Azure App Service](#) deploys to Azure App Service.

Scalability considerations

Jenkins can scale to support very large workloads. For elastic builds, do not run builds on the Jenkins master server. Instead, offload build tasks to Jenkins agents, which can be elastically scaled in and out as need. Consider two options for scaling agents:

- Use the [Azure VM Agents](#) plugin to create Jenkins agents that run in Azure VMs. This plugin enables elastic scale-out for agents and can use distinct types of virtual machines. You can select a different base image from Azure Marketplace or use a custom image. For details about how the Jenkins agents scale, see [Architecting for Scale](#) in the Jenkins documentation.
- Use the [Azure Container Agents](#) plugin to run a container as an agent in either [Azure Container Service with Kubernetes](#), or [Azure Container Instances](#).

Virtual machines generally cost more to scale than containers. To use containers for scaling, however, your build process must run with containers.

Also, use Azure Storage to share build artifacts that may be used in the next stage of the pipeline by other build agents.

Scaling the Jenkins server

You can scale the Jenkins server VM up or down by changing the VM size. The [solution template for Jenkins on Azure](#) specifies the DS2 v2 size (with two CPUs, 7 GB) by default. This size handles a small to medium team workload. Change the VM size by choosing a different option when building out the server.

Selecting the correct server size depends on the size of the expected workload. The Jenkins community maintains a

[selection guide](#) to help identify the configuration that best meets your requirements. Azure offers many [sizes for Linux VMs](#) to meet any requirements. For more information about scaling the Jenkins master, refer to the Jenkins community of [best practices](#), which also includes details about scaling Jenkins master.

Availability considerations

Availability in the context of a Jenkins server means being able to recover any state information associated with your workflow, such as test results, libraries you have created, or other artifacts. Critical workflow state or artifacts must be maintained to recover the workflow if the Jenkins server goes down. To assess your availability requirements, consider two common metrics:

- Recovery Time Objective (RTO) specifies how long you can go without Jenkins.
- Recovery Point Objective (RPO) indicates how much data you can afford to lose if a disruption in service affects Jenkins.

In practice, RTO and RPO imply redundancy and backup. Availability is not a question of hardware recovery—that is part of Azure—but rather ensuring you maintain the state of your Jenkins server. Microsoft offers a [service level agreement](#) (SLA) for single VM instances. If this SLA doesn't meet your uptime requirements, make sure you have a plan for disaster recovery, or consider using a [multi-master Jenkins server](#) deployment (not covered in this document).

Consider using the disaster recovery [scripts](#) in step 7 of the deployment to create an Azure Storage account with managed disks to store the Jenkins server state. If Jenkins goes down, it can be restored to the state stored in this separate storage account.

Security considerations

Use the following approaches to help lock down security on a basic Jenkins server, since in its basic state, it is not secure.

- Set up a secure way to log into the Jenkins server. This architecture uses HTTP and has a public IP, but HTTP is not secure by default. Consider setting up [HTTPS on the Nginx server](#) being used for a secure logon.

NOTE

When adding SSL to your server, create an NSG rule for the Jenkins subnet to open port 443. For more information, see [How to open ports to a virtual machine with the Azure portal](#).

- Ensure that the Jenkins configuration prevents cross site request forgery (Manage Jenkins > Configure Global Security). This is the default for Microsoft Jenkins Server.
- Configure read-only access to the Jenkins dashboard by using the [Matrix Authorization Strategy Plugin](#).
- Install the [Azure Credentials](#) plugin to use Key Vault to handle secrets for the Azure assets, the agents in the pipeline, and third-party components.
- Use RBAC to restrict the access of the service principal to the minimum required to run the jobs. This helps limit the scope of damage from a rogue job.

Jenkins jobs often require secrets to access Azure services that require authorization, such as Azure Container Service. Use [Key Vault](#) along with the [Azure Credential plugin](#) to manage these secrets securely. Use Key Vault to store service principal credentials, passwords, tokens, and other secrets.

To get a central view of the security state of your Azure resources, use [Azure Security Center](#). Security Center monitors potential security issues and provides a comprehensive picture of the security health of your deployment.

Security Center is configured per Azure subscription. Enable security data collection as described in the [Azure Security Center quick start guide](#). When data collection is enabled, Security Center automatically scans any virtual machines created under that subscription.

The Jenkins server has its own user management system, and the Jenkins community provides best practices for [securing a Jenkins instance on Azure](#). The solution template for Jenkins on Azure implements these best practices.

Manageability considerations

Use resource groups to organize the Azure resources that are deployed. Deploy production environments and development/test environments in separate resource groups, so that you can monitor each environment's resources and roll up billing costs by resource group. You can also delete resources as a set, which is very useful for test deployments.

Azure provides several features for [monitoring and diagnostics](#) of the overall infrastructure. To monitor CPU usage, this architecture deploys Azure Monitor. For example, you can use Azure Monitor to monitor CPU utilization, and send a notification if CPU usage exceeds 80 percent. (High CPU usage indicates that you might want to scale up the Jenkins server VM.) You can also notify a designated user if the VM fails or becomes unavailable.

Communities

Communities can answer questions and help you set up a successful deployment. Consider the following:

- [Jenkins Community Blog](#)
- [Azure Forum](#)
- [Stack Overflow Jenkins](#)

For more best practices from the Jenkins community, visit [Jenkins best practices](#).

Deploy the solution

To deploy this architecture, follow the steps below to install the [solution template for Jenkins on Azure](#), then install the scripts that set up monitoring and disaster recovery in the steps below.

Prerequisites

- This reference architecture requires an Azure subscription.
- To create an Azure service principal, you must have admin rights to the Azure AD tenant that is associated with the deployed Jenkins server.
- These instructions assume that the Jenkins administrator is also an Azure user with at least Contributor privileges.

Step 1: Deploy the Jenkins server

1. Open the [Azure Marketplace image for Jenkins](#) in your web browser and select **GET IT NOW** from the left side of the page.
2. Review the pricing details and select **Continue**, then select **Create** to configure the Jenkins server in the Azure portal.

For detailed instructions, see [Create a Jenkins server on an Azure Linux VM from the Azure portal](#). For this reference architecture, it is sufficient to get the server up and running with the admin logon. Then you can provision it to use various other services.

Step 2: Set up SSO

The step is run by the Jenkins administrator, who must also have a user account in the subscription's Azure AD

directory and must be assigned the Contributor role.

Use the [Azure AD Plugin](#) from the Jenkins Update Center in the Jenkins server and follow the instructions to set up SSO.

Step 3: Provision Jenkins server with Azure VM Agent plugin

The step is run by the Jenkins administrator to set up the Azure VM Agent plugin, which is already installed.

[Follow these steps to configure the plugin](#). For a tutorial about setting up service principals for the plugin, see [Scale your Jenkins deployments to meet demand with Azure VM agents](#).

Step 4: Provision Jenkins server with Azure Storage

The step is run by the Jenkins administrator, who sets up the Windows Azure Storage Plugin, which is already installed.

[Follow these steps to configure the plugin](#).

Step 5: Provision Jenkins server with Azure Credential plugin

The step is run by the Jenkins administrator to set up the Azure Credential plugin, which is already installed.

[Follow these steps to configure the plugin](#).

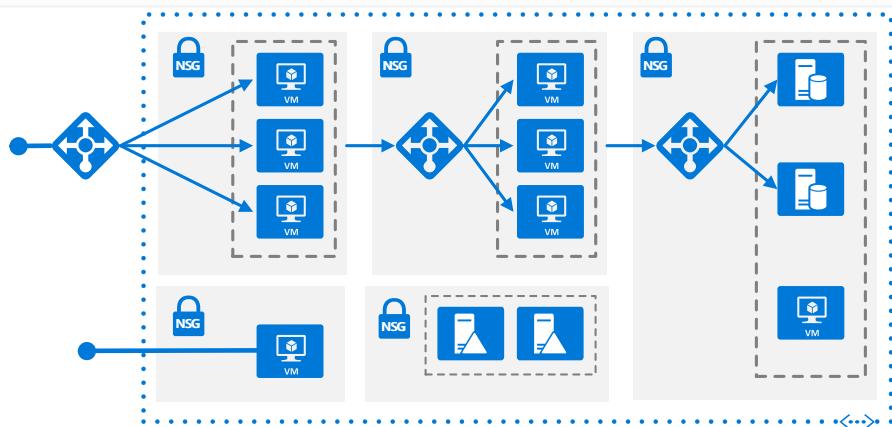
Step 6: Provision Jenkins server for monitoring by the Azure Monitor Service

To set up monitoring for your Jenkins server, follow the instructions in [Create metric alerts in Azure Monitor for Azure services](#).

Step 7: Provision Jenkins server with Managed Disks for disaster recovery

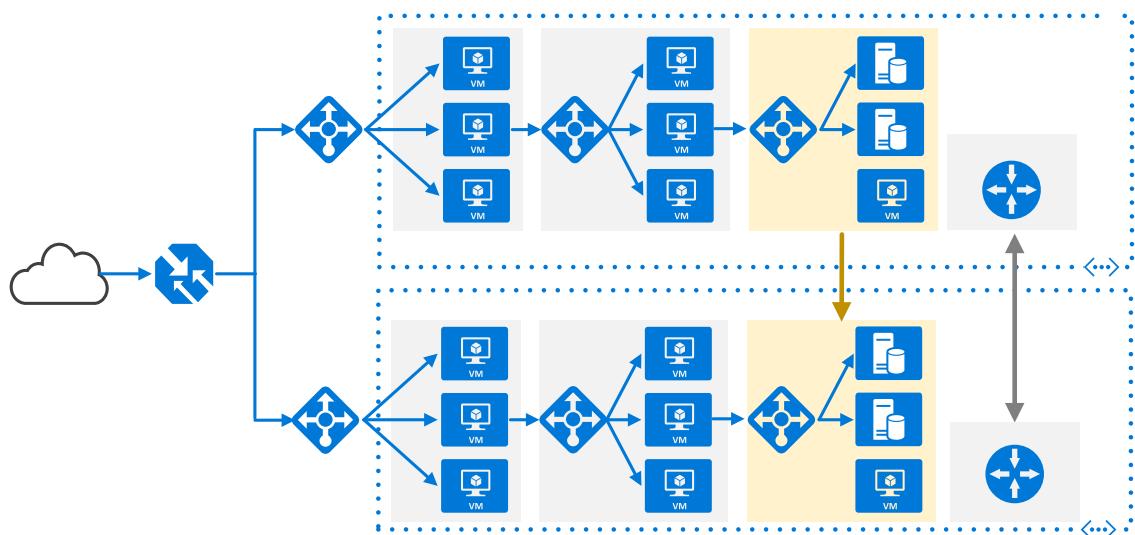
The Microsoft Jenkins product group has created disaster recovery scripts that build a managed disk used to save the Jenkins state. If the server goes down, it can be restored to its latest state.

Download and run the disaster recovery scripts from [GitHub](#).



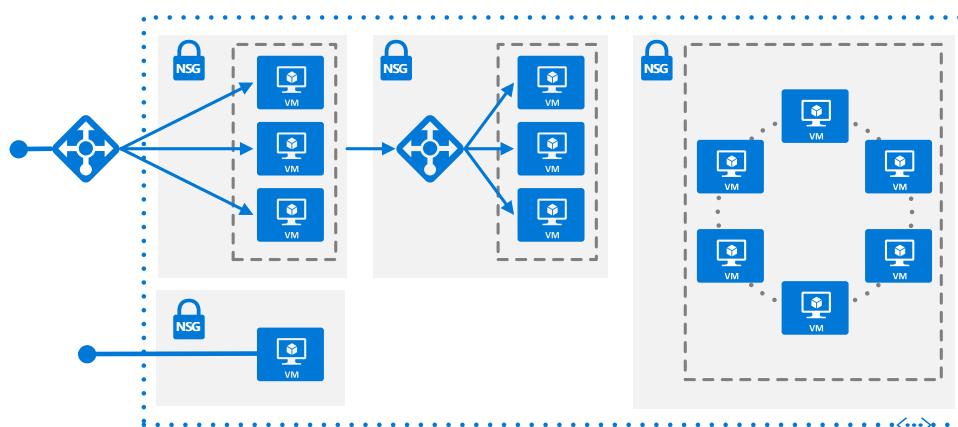
N-tier application with SQL Server

Deploy VMs and a virtual network configured for an N-tier application using SQL Server on Windows.



Multi-region N-tier application with SQL Server

Deploy an N-tier application to two regions for high availability, using SQL Server Always On Availability Groups.



N-tier application with Cassandra

Deploy Linux VMs and a virtual network configured for an N-tier application using Apache Cassandra.



Windows VM

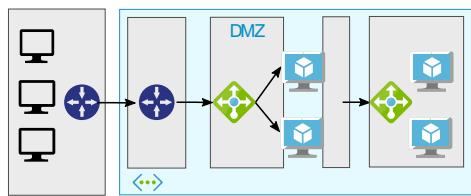
Baseline recommendations for running a Windows VM in Azure.



Linux VM

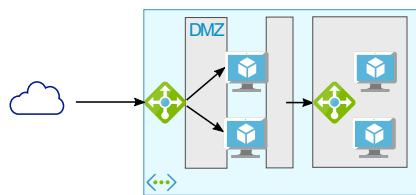
Baseline recommendations for running a Linux VM in Azure.

These reference architectures show proven practices for creating a network DMZ that protects the boundary between an Azure virtual network and an on-premises network or the Internet.



DMZ between Azure and on-premises

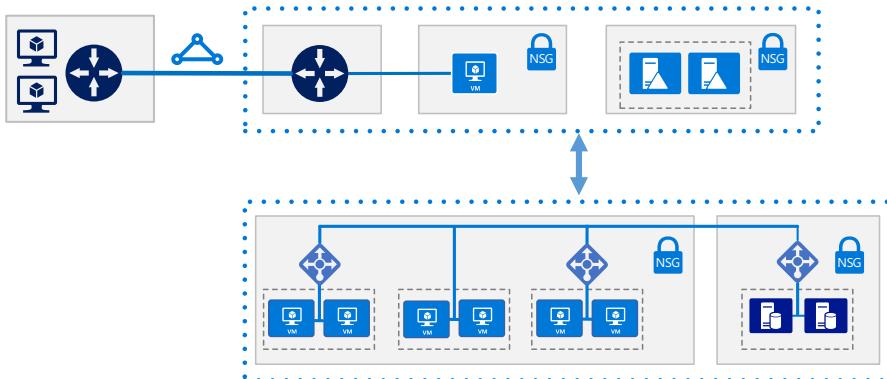
Implements a secure hybrid network that extends an on-premises network to Azure.



DMZ between Azure and the Internet

Implements a secure network that accepts Internet traffic to Azure.

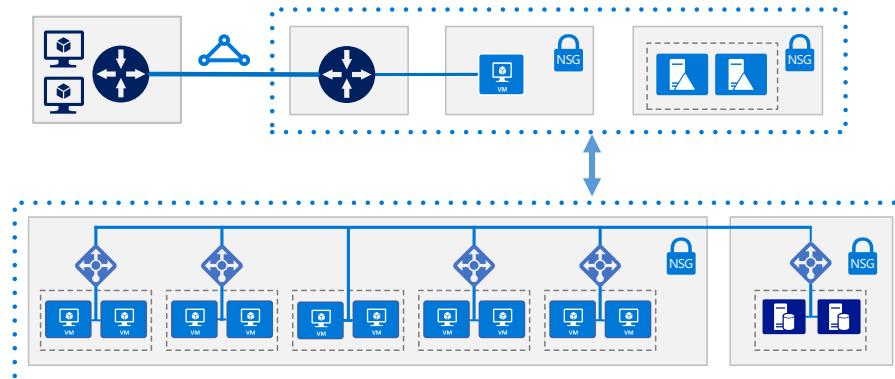
These reference architectures show proven practices for running a production SAP deployment on Azure. The architectures are configured for high availability and disaster recovery (HADR).



SAP NetWeaver for AnyDB

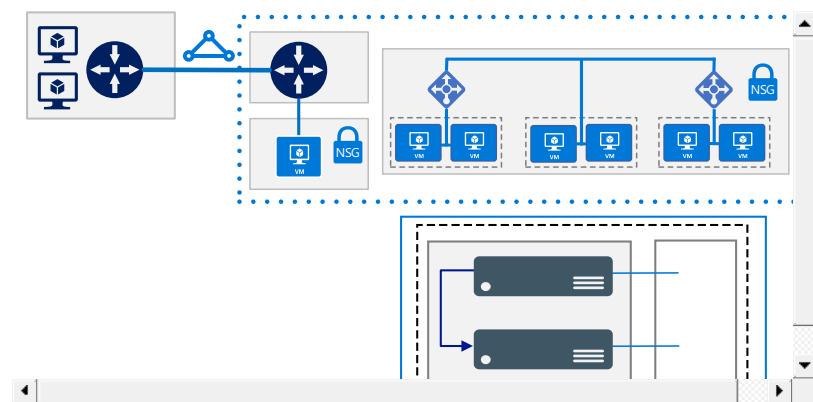
Run SAP NetWeaver in a Windows environment.

This reference architecture targets AnyDB, the SAP term for any supported DBMS besides SAP HANA.



SAP S/4HANA

Run SAP S/4HANA in a Linux environment.



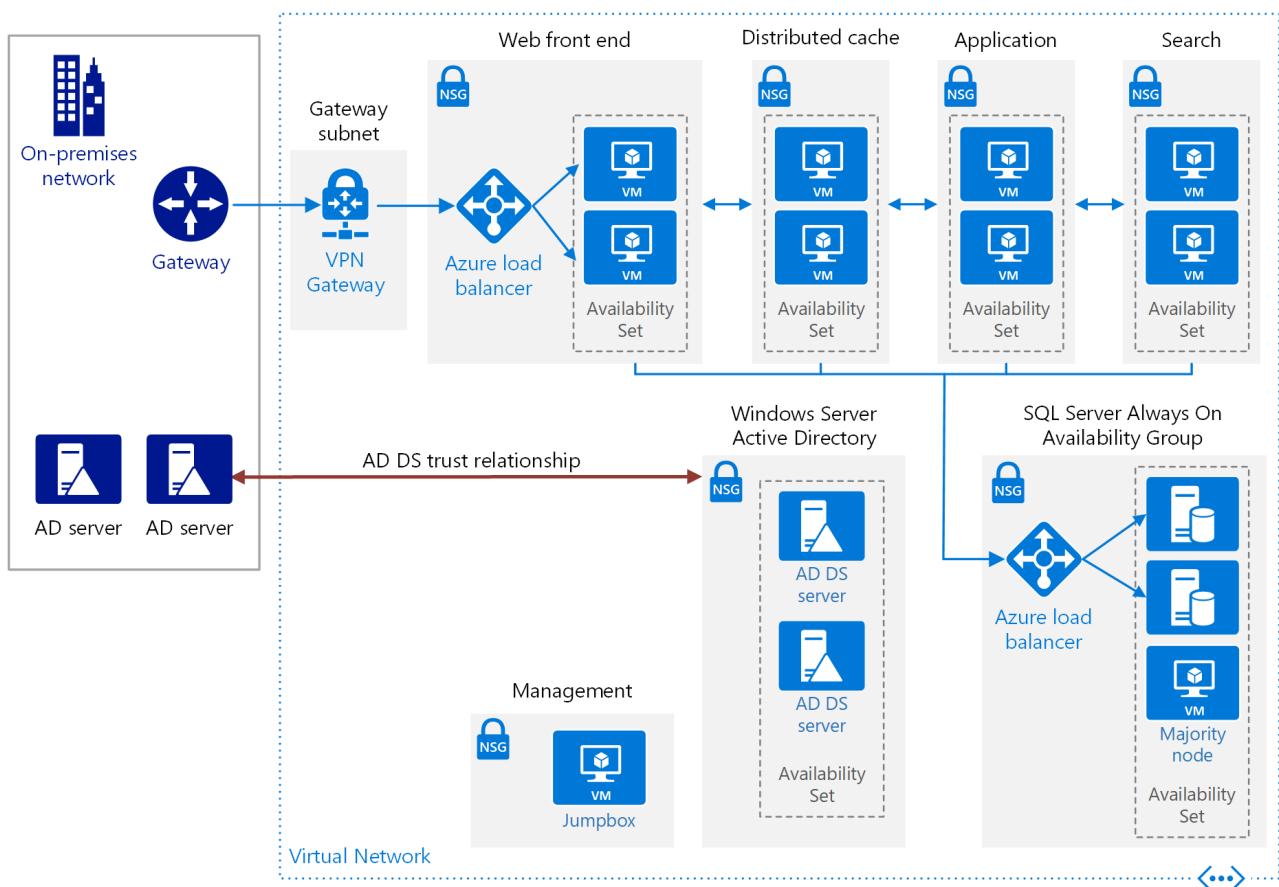
SAP HANA on Azure Large Instances

HANA Large Instances are deployed on physical servers in Azure regions.

Run a high availability SharePoint Server 2016 farm in Azure

4/11/2018 • 14 minutes to read • [Edit Online](#)

This reference architecture shows a set of proven practices for setting up a high availability SharePoint Server 2016 farm on Azure, using MinRole topology and SQL Server Always On availability groups. The SharePoint farm is deployed in a secured virtual network with no Internet-facing endpoint or presence. [Deploy this solution.](#)



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

Architecture

This architecture builds on the one shown in [Run Windows VMs for an N-tier application](#). It deploys a SharePoint Server 2016 farm with high availability inside an Azure virtual network (VNet). This architecture is suitable for a test or production environment, a SharePoint hybrid infrastructure with Office 365, or as the basis for a disaster recovery scenario.

The architecture consists of the following components:

- **Resource groups.** A [resource group](#) is a container that holds related Azure resources. One resource group is used for the SharePoint servers, and another resource group is used for infrastructure components that are independent of VMs, such as the virtual network and load balancers.
- **Virtual network (VNet).** The VMs are deployed in a VNet with a unique intranet address space. The VNet is further subdivided into subnets.
- **Virtual machines (VMs).** The VMs are deployed into the VNet, and private static IP addresses are

assigned to all of the VMs. Static IP addresses are recommended for the VMs running SQL Server and SharePoint Server 2016, to avoid issues with IP address caching and changes of addresses after a restart.

- **Availability sets.** Place the VMs for each SharePoint role into separate [availability sets](#), and provision at least two virtual machines (VMs) for each role. This makes the VMs eligible for a higher service level agreement (SLA).
- **Internal load balancer.** The [load balancer](#) distributes SharePoint request traffic from the on-premises network to the front-end web servers of the SharePoint farm.
- **Network security groups (NSGs).** For each subnet that contains virtual machines, a [network security group](#) is created. Use NSGs to restrict network traffic within the VNet, in order to isolate subnets.
- **Gateway.** The gateway provides a connection between your on-premises network and the Azure virtual network. Your connection can use ExpressRoute or site-to-site VPN. For more information, see [Connect an on-premises network to Azure](#).
- **Windows Server Active Directory (AD) domain controllers.** Because SharePoint Server 2016 does not support using Azure Active Directory Domain Services, you must deploy Windows Server AD domain controllers. These domain controllers run in the Azure VNet and have a trust relationship with the on-premises Windows Server AD forest. Client web requests for SharePoint farm resources are authenticated in the VNet rather than sending that authentication traffic across the gateway connection to the on-premises network. In DNS, intranet A or CNAME records are created so that intranet users can resolve the name of the SharePoint farm to the private IP address of the internal load balancer.
- **SQL Server Always On Availability Group.** For high availability of the SQL Server database, we recommend [SQL Server Always On Availability Groups](#). Two virtual machines are used for SQL Server. One contains the primary database replica and the other contains the secondary replica.
- **Majority node VM.** This VM allows the failover cluster to establish quorum. For more information, see [Understanding Quorum Configurations in a Failover Cluster](#).
- **SharePoint servers.** The SharePoint servers perform the web front-end, caching, application, and search roles.
- **Jumpbox.** Also called a [bastion host](#). This is a secure VM on the network that administrators use to connect to the other VMs. The jumpbox has an NSG that allows remote traffic only from public IP addresses on a safe list. The NSG should permit remote desktop (RDP) traffic.

Recommendations

Your requirements might differ from the architecture described here. Use these recommendations as a starting point.

Resource group recommendations

We recommend separating resource groups according to the server role, and having a separate resource group for infrastructure components that are global resources. In this architecture, the SharePoint resources form one group, while the SQL Server and other utility assets form another.

Virtual network and subnet recommendations

Use one subnet for each SharePoint role, plus a subnet for the gateway and one for the jumpbox.

The gateway subnet must be named *GatewaySubnet*. Assign the gateway subnet address space from the last part of the virtual network address space. For more information, see [Connect an on-premises network to Azure using a VPN gateway](#).

VM recommendations

Based on Standard DSv2 virtual machine sizes, this architecture requires a minimum of 38 cores:

- 8 SharePoint servers on Standard_DS3_v2 (4 cores each) = 32 cores
- 2 Active Directory domain controllers on Standard_DS1_v2 (1 core each) = 2 cores
- 2 SQL Server VMs on Standard_DS1_v2 = 2 cores
- 1 majority node on Standard_DS1_v2 = 1 core
- 1 management server on Standard_DS1_v2 = 1 core

The total number of cores will depend on the VM sizes that you select. For more information, see [SharePoint Server recommendations](#) below.

Make sure your Azure subscription has enough VM core quota for the deployment, or the deployment will fail. See [Azure subscription and service limits, quotas, and constraints](#).

NSG recommendations

We recommend having one NSG for each subnet that contains VMs, to enable subnet isolation. If you want to configure subnet isolation, add NSG rules that define the allowed or denied inbound or outbound traffic for each subnet. For more information, see [Filter network traffic with network security groups](#).

Do not assign an NSG to the gateway subnet, or the gateway will stop functioning.

Storage recommendations

The storage configuration of the VMs in the farm should match the appropriate best practices used for on-premises deployments. SharePoint servers should have a separate disk for logs. SharePoint servers hosting search index roles require additional disk space for the search index to be stored. For SQL Server, the standard practice is to separate data and logs. Add more disks for database backup storage, and use a separate disk for `tempdb`.

For best reliability, we recommend using [Azure Managed Disks](#). Managed disks ensure that the disks for VMs within an availability set are isolated to avoid single points of failure.

NOTE

Currently the Resource Manager template for this reference architecture does not use managed disks. We are planning to update the template to use managed disks.

Use Premium managed disks for all SharePoint and SQL Server VMs. You can use Standard managed disks for the majority node server, the domain controllers, and the management server.

SharePoint Server recommendations

Before configuring the SharePoint farm, make sure you have one Windows Server Active Directory service account per service. For this architecture, you need at a minimum the following domain-level accounts to isolate privilege per role:

- SQL Server Service account
- Setup User account
- Server Farm account
- Search Service account
- Content Access account
- Web App Pool accounts
- Service App Pool accounts
- Cache Super User account
- Cache Super Reader account

For all roles except the Search Indexer, we recommended using the [Standard_DS3_v2](#) VM size. The Search Indexer

should be at least the [Standard_DS13_v2](#) size.

NOTE

The Resource Manager template for this reference architecture uses the smaller DS3 size for the Search Indexer, for purposes of testing the deployment. For a production deployment, use the DS13 size or larger.

For production workloads, see [Hardware and software requirements for SharePoint Server 2016](#).

To meet the support requirement for disk throughput of 200 MB per second minimum, make sure to plan the Search architecture. See [Plan enterprise search architecture in SharePoint Server 2013](#). Also follow the guidelines in [Best practices for crawling in SharePoint Server 2016](#).

In addition, store the search component data on a separate storage volume or partition with high performance. To reduce load and improve throughput, configure the object cache user accounts, which are required in this architecture. Split the Windows Server operating system files, the SharePoint Server 2016 program files, and diagnostics logs across three separate storage volumes or partitions with normal performance.

For more information about these recommendations, see [Initial deployment administrative and service accounts in SharePoint Server 2016](#).

Hybrid workloads

This reference architecture deploys a SharePoint Server 2016 farm that can be used as a [SharePoint hybrid environment](#) — that is, extending SharePoint Server 2016 to Office 365 SharePoint Online. If you have Office Online Server, see [Office Web Apps and Office Online Server supportability in Azure](#).

The default service applications in this deployment are designed to support hybrid workloads. All SharePoint Server 2016 and Office 365 hybrid workloads can be deployed to this farm without changes to the SharePoint infrastructure, with one exception: The Cloud Hybrid Search Service Application must not be deployed onto servers hosting an existing search topology. Therefore, one or more search-role-based VMs must be added to the farm to support this hybrid scenario.

SQL Server Always On Availability Groups

This architecture uses SQL Server virtual machines because SharePoint Server 2016 cannot use Azure SQL Database. To support high availability in SQL Server, we recommend using Always On Availability Groups, which specify a set of databases that fail over together, making them highly-available and recoverable. In this reference architecture, the databases are created during deployment, but you must manually enable Always On Availability Groups and add the SharePoint databases to an availability group. For more information, see [Create the availability group and add the SharePoint databases](#).

We also recommend adding a listener IP address to the cluster, which is the private IP address of the internal load balancer for the SQL Server virtual machines.

For recommended VM sizes and other performance recommendations for SQL Server running in Azure, see [Performance best practices for SQL Server in Azure Virtual Machines](#). Also follow the recommendations in [Best practices for SQL Server in a SharePoint Server 2016 farm](#).

We recommend that the majority node server reside on a separate computer from the replication partners. The server enables the secondary replication partner server in a high-safety mode session to recognize whether to initiate an automatic failover. Unlike the two partners, the majority node server doesn't serve the database but rather supports automatic failover.

Scalability considerations

To scale up the existing servers, simply change the VM size.

With the [MinRoles](#) capability in SharePoint Server 2016, you can scale out servers based on the server's role and also remove servers from a role. When you add servers to a role, you can specify any of the single roles or one of the combined roles. If you add servers to the Search role, however, you must also reconfigure the search topology using PowerShell. You can also convert roles using MinRoles. For more information, see [Managing a MinRole Server Farm in SharePoint Server 2016](#).

Note that SharePoint Server 2016 doesn't support using virtual machine scale sets for auto-scaling.

Availability considerations

This reference architecture supports high availability within an Azure region, because each role has at least two VMs deployed in an availability set.

To protect against a regional failure, create a separate disaster recovery farm in a different Azure region. Your recovery time objectives (RTOs) and recovery point objectives (RPOs) will determine the setup requirements. For details, see [Choose a disaster recovery strategy for SharePoint 2016](#). The secondary region should be a *paired region* with the primary region. In the event of a broad outage, recovery of one region is prioritized out of every pair. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

Manageability considerations

To operate and maintain servers, server farms, and sites, follow the recommended practices for SharePoint operations. For more information, see [Operations for SharePoint Server 2016](#).

The tasks to consider when managing SQL Server in a SharePoint environment may differ from the ones typically considered for a database application. A best practice is to fully back up all SQL databases weekly with incremental nightly backups. Back up transaction logs every 15 minutes. Another practice is to implement SQL Server maintenance tasks on the databases while disabling the built-in SharePoint ones. For more information, see [Storage and SQL Server capacity planning and configuration](#).

Security considerations

The domain-level service accounts used to run SharePoint Server 2016 require Windows Server AD domain controllers for domain-join and authentication processes. Azure Active Directory Domain Services can't be used for this purpose. To extend the Windows Server AD identity infrastructure already in place in the intranet, this architecture uses two Windows Server AD replica domain controllers of an existing on-premises Windows Server AD forest.

In addition, it's always wise to plan for security hardening. Other recommendations include:

- Add rules to NSGs to isolate subnets and roles.
- Don't assign public IP addresses to VMs.
- For intrusion detection and analysis of payloads, consider using a network virtual appliance in front of the front-end web servers instead of an internal Azure load balancer.
- As an option, use IPsec policies for encryption of cleartext traffic between servers. If you are also doing subnet isolation, update your network security group rules to allow IPsec traffic.
- Install anti-malware agents for the VMs.

Deploy the solution

The deployment scripts for this reference architecture are available on [GitHub](#).

You can deploy this architecture incrementally or all at once. The first time, we recommend an incremental deployment, so that you can see what each deployment does. Specify the increment using one of the following *mode* parameters.

MODE	WHAT IT DOES
onprem	(Optional) Deploys a simulated on-premises network environment, for testing or evaluation. This step does not connect to an actual on-premises network.
infrastructure	Deploys the SharePoint 2016 network infrastructure and jumpbox to Azure.
createvpn	Deploys a virtual network gateway for both the SharePoint and on-premises networks and connects them. Run this step only if you ran the <code>onprem</code> step.
workload	Deploys the SharePoint servers to the SharePoint network.
security	Deploys the network security group to the SharePoint network.
all	Deploys all the preceding deployments.

To deploy the architecture incrementally with a simulated on-premises network environment, run the following steps in order:

1. onprem
2. infrastructure
3. createvpn
4. workload
5. security

To deploy the architecture incrementally without a simulated on-premises network environment, run the following steps in order:

1. infrastructure
2. workload
3. security

To deploy everything in one step, use `all`. Note that the entire process may take several hours.

Prerequisites

- Install the latest version of [Azure PowerShell](#).
- Before deploying this reference architecture, verify that your subscription has sufficient quota—at least 38 cores. If you don't have enough, use the Azure portal to submit a support request for more quota.
- To estimate the cost of this deployment, see the [Azure Pricing Calculator](#).

Deploy the reference architecture

1. Download or clone the [GitHub repo](#) to your local computer.
2. Open a PowerShell window and navigate to the `/sharepoint/sharepoint-2016` folder.
3. Run the following PowerShell command. For `<subscription id>`, use your Azure subscription ID. For `<location>`, specify an Azure region, such as `eastus` or `westus`. For `<mode>`, specify `onprem`, `infrastructure`, `createvpn`, `workload`, `security`, OR `all`.

```
. \Deploy-ReferenceArchitecture.ps1 <subscription id> <location> <mode>
```

4. When prompted, log on to your Azure account. The deployment scripts can take up to several hours to complete, depending on the mode you selected.
5. When the deployment completes, run the scripts to configure SQL Server Always On Availability Groups. See the [readme](#) for details.

WARNING

The parameter files include a hard-coded password (`AweS0me@PW`) in various places. Change these values before you deploy.

Validate the deployment

After you deploy this reference architecture, the following resource groups are listed under the Subscription that you used:

RESOURCE GROUP	PURPOSE
ra-onprem-sp2016-rg	Simulated on-premises network with Active Directory, federated with the SharePoint 2016 network
ra-sp2016-network-rg	Infrastructure to support SharePoint deployment
ra-sp2016-workload-rg	SharePoint and supporting resources

Validate access to the SharePoint site from the on-premises network

1. In the [Azure portal](#), under **Resource groups**, select the `ra-onprem-sp2016-rg` resource group.
2. In the list of resources, select the VM resource named `ra-adds-user-vm1`.
3. Connect to the VM, as described in [Connect to virtual machine](#). The user name is `\onpremuser`.
4. When the remote connection to the VM is established, open a browser in the VM and navigate to `http://portal.contoso.local`.
5. In the **Windows Security** box, log on to the SharePoint portal using `contoso.local\testuser` for the user name.

This logon tunnels from the Fabrikam.com domain used by the on-premises network to the contoso.local domain used by the SharePoint portal. When the SharePoint site opens, you'll see the root demo site.

Validate jumpbox access to VMs and check configuration settings

1. In [Azure portal](#), under **Resource groups**, select the `ra-sp2016-network-rg` resource group.
2. In the list of resources, select the VM resource named `ra-sp2016-jb-vm1`, which is the jumpbox.
3. Connect to the VM, as described in [Connect to virtual machine](#). The user name is `testuser`.
4. After you log onto the jumpbox, open an RDP session from the jumpbox. Connect to any other VMs in the VNet. The username is `testuser`. You can ignore the warning about the remote computer's security certificate.
5. When the remote connection to the VM opens, review the configuration and make changes using the administrative tools such as Server Manager.

The following table shows the VMs that are deployed.

RESOURCE NAME	PURPOSE	RESOURCE GROUP	VM NAME
Ra-sp2016-ad-vm1	Active Directory + DNS	Ra-sp2016-network-rg	Ad1.contoso.local
Ra-sp2016-ad-vm2	Active Directory + DNS	Ra-sp2016-network-rg	Ad2.contoso.local
Ra-sp2016-fsw-vm1	SharePoint	Ra-sp2016-network-rg	Fsw1.contoso.local
Ra-sp2016-jb-vm1	Jumpbox	Ra-sp2016-network-rg	Jb (use public IP to log on)
Ra-sp2016-sql-vm1	SQL Always On - Failover	Ra-sp2016-network-rg	Sq1.contoso.local
Ra-sp2016-sql-vm2	SQL Always On - Primary	Ra-sp2016-network-rg	Sq2.contoso.local
Ra-sp2016-app-vm1	SharePoint 2016 Application MinRole	Ra-sp2016-workload-rg	App1.contoso.local
Ra-sp2016-app-vm2	SharePoint 2016 Application MinRole	Ra-sp2016-workload-rg	App2.contoso.local
Ra-sp2016-dch-vm1	SharePoint 2016 Distributed Cache MinRole	Ra-sp2016-workload-rg	Dch1.contoso.local
Ra-sp2016-dch-vm2	SharePoint 2016 Distributed Cache MinRole	Ra-sp2016-workload-rg	Dch2.contoso.local
Ra-sp2016-srch-vm1	SharePoint 2016 Search MinRole	Ra-sp2016-workload-rg	Srch1.contoso.local
Ra-sp2016-srch-vm2	SharePoint 2016 Search MinRole	Ra-sp2016-workload-rg	Srch2.contoso.local
Ra-sp2016-wfe-vm1	SharePoint 2016 Web Front End MinRole	Ra-sp2016-workload-rg	Wfe1.contoso.local
Ra-sp2016-wfe-vm2	SharePoint 2016 Web Front End MinRole	Ra-sp2016-workload-rg	Wfe2.contoso.local

Contributors to this reference architecture — Joe Davies, Bob Fox, Neil Hodgkinson, Paul Stork

Cloud Design Patterns

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

These design patterns are useful for building reliable, scalable, secure applications in the cloud.

Each pattern describes the problem that the pattern addresses, considerations for applying the pattern, and an example based on Microsoft Azure. Most of the patterns include code samples or snippets that show how to implement the pattern on Azure. However, most of the patterns are relevant to any distributed system, whether hosted on Azure or on other cloud platforms.

Challenges in cloud development

	<h3>Availability</h3> <p>Availability is the proportion of time that the system is functional and working, usually measured as a percentage of uptime. It can be affected by system errors, infrastructure problems, malicious attacks, and system load. Cloud applications typically provide users with a service level agreement (SLA), so applications must be designed to maximize availability.</p>
	<h3>Data Management</h3> <p>Data management is the key element of cloud applications, and influences most of the quality attributes. Data is typically hosted in different locations and across multiple servers for reasons such as performance, scalability or availability, and this can present a range of challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations.</p>
	<h3>Design and Implementation</h3> <p>Good design encompasses factors such as consistency and coherence in component design and deployment, maintainability to simplify administration and development, and reusability to allow components and subsystems to be used in other applications and in other scenarios. Decisions made during the design and implementation phase have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.</p>
	<h3>Messaging</h3> <p>The distributed nature of cloud applications requires a messaging infrastructure that connects the components and services, ideally in a loosely coupled manner in order to maximize scalability. Asynchronous messaging is widely used, and provides many benefits, but also brings challenges such as the ordering of messages, poison message management, idempotency, and more</p>
	<h3>Management and Monitoring</h3> <p>Cloud applications run in a remote datacenter where you do not have full control of the infrastructure or, in some cases, the operating system. This can make management and monitoring more difficult than an on-premises deployment. Applications must expose runtime information that administrators and operators can use to manage and monitor the system, as well as supporting changing business requirements and customization without requiring the application to be stopped or redeployed.</p>

	<p>Performance and Scalability</p> <p>Performance is an indication of the responsiveness of a system to execute any action within a given time interval, while scalability is ability of a system either to handle increases in load without impact on performance or for the available resources to be readily increased. Cloud applications typically encounter variable workloads and peaks in activity. Predicting these, especially in a multi-tenant scenario, is almost impossible. Instead, applications should be able to scale out within limits to meet peaks in demand, and scale in when demand decreases. Scalability concerns not just compute instances, but other elements such as data storage, messaging infrastructure, and more.</p>
	<p>Resiliency</p> <p>Resiliency is the ability of a system to gracefully handle and recover from failures. The nature of cloud hosting, where applications are often multi-tenant, use shared platform services, compete for resources and bandwidth, communicate over the Internet, and run on commodity hardware means there is an increased likelihood that both transient and more permanent faults will arise. Detecting failures, and recovering quickly and efficiently, is necessary to maintain resiliency.</p>
	<p>Security</p> <p>Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. Cloud applications are exposed on the Internet outside trusted on-premises boundaries, are often open to the public, and may serve untrusted users. Applications must be designed and deployed in a way that protects them from malicious attacks, restricts access to only approved users, and protects sensitive data.</p>

Catalog of patterns

PATTERN	SUMMARY
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
Backends for Frontends	Create separate backend services to be consumed by specific frontend applications or interfaces.
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Cache-Aside	Load data on demand into a cache from a data store
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation.
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel.

PATTERN	SUMMARY
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Federated Identity	Delegate authentication to an external identity provider.
Gatekeeper	Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed.

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

Availability patterns

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

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

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

Data Management patterns

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

Data management is the key element of cloud applications, and influences most of the quality attributes. Data is typically hosted in different locations and across multiple servers for reasons such as performance, scalability or availability, and this can present a range of challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations.

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

Design and Implementation patterns

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

Good design encompasses factors such as consistency and coherence in component design and deployment, maintainability to simplify administration and development, and reusability to allow components and subsystems to be used in other applications and in other scenarios. Decisions made during the design and implementation phase have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.

PATTERN	SUMMARY
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
Backends for Frontends	Create separate backend services to be consumed by specific frontend applications or interfaces.
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.

PATTERN	SUMMARY
Strangler	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

Management and Monitoring patterns

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

Cloud applications run in a remote datacenter where you do not have full control of the infrastructure or, in some cases, the operating system. This can make management and monitoring more difficult than an on-premises deployment. Applications must expose runtime information that administrators and operators can use to manage and monitor the system, as well as supporting changing business requirements and customization without requiring the application to be stopped or redeployed.

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

Messaging patterns

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

The distributed nature of cloud applications requires a messaging infrastructure that connects the components and services, ideally in a loosely coupled manner in order to maximize scalability. Asynchronous messaging is widely used, and provides many benefits, but also brings challenges such as the ordering of messages, poison message management, idempotency, and more.

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

Performance and Scalability patterns

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

Performance is an indication of the responsiveness of a system to execute any action within a given time interval, while scalability is ability of a system either to handle increases in load without impact on performance or for the available resources to be readily increased. Cloud applications typically encounter variable workloads and peaks in activity. Predicting these, especially in a multi-tenant scenario, is almost impossible. Instead, applications should be able to scale out within limits to meet peaks in demand, and scale in when demand decreases. Scalability concerns not just compute instances, but other elements such as data storage, messaging infrastructure, and more.

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

Resiliency patterns

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

Resiliency is the ability of a system to gracefully handle and recover from failures. The nature of cloud hosting, where applications are often multi-tenant, use shared platform services, compete for resources and bandwidth, communicate over the Internet, and run on commodity hardware means there is an increased likelihood that both transient and more permanent faults will arise. Detecting failures, and recovering quickly and efficiently, is necessary to maintain resiliency.

PATTERN	SUMMARY
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed.
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources.

Security patterns

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

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

PATTERN	SUMMARY
Federated Identity	Delegate authentication to an external identity provider.
Gatekeeper	Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them.
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

API design

6/19/2018 • 28 minutes to read • [Edit Online](#)

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

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

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

Introduction to REST

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

A primary advantage of REST over HTTP is that it uses open standards, and does not bind the implementation of the API or the client applications 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://adventure-works.com/orders/1
```

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

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

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

clients and specific servers. Any server can handle any request from any client. That said, other factors can limit scalability. For example, many web services write to a backend data store, which may be hard to scale out. (The article [Data Partitioning](#) describes strategies to scale out a data store.)

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

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

In 2008, Leonard Richardson proposed the following [maturity model](#) for web APIs:

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

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

Organize the API around resources

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

```
http://adventure-works.com/orders // Good  
http://adventure-works.com/create-order // Avoid
```

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

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

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

Sending an HTTP GET request to the collection URI retrieves a list of items in the collection. Each item in the collection also has its own unique URI. An HTTP GET request to the item's URI returns the details of that item.

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

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

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

TIP

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

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

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

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

Define operations in terms of HTTP methods

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

- **GET** retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.
- **POST** creates a new resource at the specified URI. The body of the request message provides the details of the

new resource. Note that POST can also be used to trigger operations that don't actually create resources.

- **PUT** either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
- **PATCH** performs a partial update of a resource. The request body specifies the set of changes to apply to the resource.
- **DELETE** removes the resource at the specified URI.

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

RESOURCE	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

The differences between POST, PUT, and PATCH can be confusing.

- A POST request creates a resource. The server assigns a URI for the new resource, and returns that URI to the client. In the REST model, you frequently apply POST requests to collections. The new resource is added to the collection. A POST request can also be used to submit data for processing to an existing resource, without any new resource being created.
- A PUT request creates a resource or updates an existing resource. The client specifies the URI for the resource. The request body contains a complete representation of the resource. If a resource with this URI already exists, it is replaced. Otherwise a new resource is created, if the server supports doing so. PUT requests are most frequently applied to resources that are individual items, such as a specific customer, rather than collections. A server might support updates but not creation via PUT. Whether to support creation via PUT depends on whether the client can meaningfully assign a URI to a resource before it exists. If not, then use POST to create resources and PUT or PATCH to update.
- A PATCH request performs a *partial update* to an existing resource. The client specifies the URI for the resource. The request body specifies a set of *changes* to apply to the resource. This can be more efficient than using PUT, because the client only sends the changes, not the entire representation of the resource. Technically PATCH can also create a new resource (by specifying a set of updates to a "null" resource), if the server supports this.

PUT requests must be idempotent. If a client submits the same PUT request multiple times, the results should always be the same (the same resource will be modified with the same values). POST and PATCH requests are not guaranteed to be idempotent.

Conform to HTTP semantics

This section describes some typical considerations for designing an API that conforms to the HTTP specification. However, it doesn't cover every possible detail or scenario. When in doubt, consult the HTTP specifications.

Media types

As mentioned earlier, clients and servers exchange representations of resources. For example, in a POST request, the request body contains a representation of the resource to create. In a GET request, the response body contains

a representation of the fetched resource.

In the HTTP protocol, formats are specified through the use of *media types*, also called MIME types. For non-binary data, most web APIs support JSON (media type = application/json) and possibly XML (media type = application/xml).

The Content-Type header in a request or response specifies the format of the representation. Here is an example of a POST request that includes JSON data:

```
POST http://adventure-works.com/orders HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: 57

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

If the server doesn't support the media type, it should return HTTP status code 415 (Unsupported Media Type).

A client request can include an Accept header that contains a list of media types the client will accept from the server in the response message. For example:

```
GET http://adventure-works.com/orders/2 HTTP/1.1
Accept: application/json
```

If the server cannot match any of the media type(s) listed, it should return HTTP status code 406 (Not Acceptable).

GET methods

A successful GET method typically returns HTTP status code 200 (OK). If the resource cannot be found, the method should return 404 (Not Found).

POST methods

If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

PUT methods

If a PUT method creates a new resource, it returns HTTP status code 201 (Created), as with a POST method. If the method updates an existing resource, it returns either 200 (OK) or 204 (No Content). In some cases, it might not be possible to update an existing resource. In that case, consider returning HTTP status code 409 (Conflict).

Consider implementing bulk HTTP PUT operations that can batch updates to multiple resources in a collection. The PUT request should specify the URI of the collection, and the request body should specify the details of the resources to be modified. This approach can help to reduce chattiness and improve performance.

PATCH methods

With a PATCH request, the client sends a set of updates to an existing resource, in the form of a *patch document*. The server processes the patch document to perform the update. The patch document doesn't describe the whole resource, only a set of changes to apply. The specification for the PATCH method ([RFC 5789](#)) doesn't define a particular format for patch documents. The format must be inferred from the media type in the request.

JSON is probably the most common data format for web APIs. There are two main JSON-based patch formats,

called *JSON patch* and *JSON merge patch*.

JSON merge patch is somewhat simpler. The patch document has the same structure as the original JSON resource, but includes just the subset of fields that should be changed or added. In addition, a field can be deleted by specifying `null` for the field value in the patch document. (That means merge patch is not suitable if the original resource can have explicit null values.)

For example, suppose the original resource has the following JSON representation:

```
{  
  "name": "gizmo",  
  "category": "widgets",  
  "color": "blue",  
  "price": 10  
}
```

Here is a possible JSON merge patch for this resource:

```
{  
  "price": 12,  
  "color": null,  
  "size": "small"  
}
```

This tells the server to update "price", delete "color", and add "size". "Name" and "category" are not modified. For the exact details of JSON merge patch, see [RFC 7396](#). The media type for JSON merge patch is "application/merge-patch+json".

Merge patch is not suitable if the original resource can contain explicit null values, due to the special meaning of `null` in the patch document. Also, the patch document doesn't specify the order that the server should apply the updates. That may or may not matter, depending on the data and the domain. JSON patch, defined in [RFC 6902](#), is more flexible. It specifies the changes as a sequence of operations to apply. Operations include add, remove, replace, copy, and test (to validate values). The media type for JSON patch is "application/json-patch+json".

Here are some typical error conditions that might be encountered when processing a PATCH request, along with the appropriate HTTP status code.

ERROR CONDITION	HTTP STATUS CODE
The patch document format isn't supported.	415 (Unsupported Media Type)
Malformed patch document.	400 (Bad Request)
The patch document is valid, but the changes can't be applied to the resource in its current state.	409 (Conflict)

DELETE methods

If the delete operation is successful, the web server should respond with HTTP status code 204, indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn't exist, the web server can return HTTP 404 (Not Found).

Asynchronous operations

Sometimes a POST, PUT, PATCH, or DELETE operation might require processing that takes awhile to complete. If you wait for completion before sending a response to the client, it may cause unacceptable latency. If so, consider making the operation asynchronous. Return HTTP status code 202 (Accepted) to indicate the request was accepted for processing but is not completed.

You should expose an endpoint that returns the status of an asynchronous request, so the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response. For example:

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

If the client sends a GET request to this endpoint, the response should contain the current status of the request. Optionally, it could also include an estimated time to completion or a link to cancel the operation.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "In progress",
  "link": { "rel": "cancel", "method": "delete", "href": "/api/status/12345" }
}
```

If the asynchronous operation creates a new resource, the status endpoint should return status code 303 (See Other) after the operation completes. In the 303 response, include a Location header that gives the URI of the new resource:

```
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

For more information, see [Asynchronous operations in REST](#).

Filter and paginate data

Exposing a collection of resources through a single URI can lead to applications fetching large amounts of data when only a subset of the information is required. For example, suppose a client application needs to find all orders with a cost over a specific value. It might retrieve all orders from the `/orders` URI and then filter these orders on the client side. Clearly this process is highly inefficient. It wastes network bandwidth and processing power on the server hosting the web API.

Instead, the API can allow passing a filter in the query string of the URI, such as `/orders?minCost=n`. The web API is then responsible for parsing and handling the `minCost` parameter in the query string and returning the filtered results on the sever side.

GET requests over collection resources can potentially return a large number of items. You should design a web API to limit the amount of data returned by any single request. Consider supporting query strings that specify the maximum number of items to retrieve and a starting offset into the collection. For example:

```
/orders?limit=25&offset=50
```

Also consider imposing an upper limit on the number of items returned, to help prevent Denial of Service attacks. To assist client applications, GET requests that return paginated data should also include some form of metadata that indicate the total number of resources available in the collection. You might also consider other intelligent paging strategies; for more information, see [API Design Notes: Smart Paging](#)

You can use a similar strategy to sort data as it is fetched, by providing a sort parameter that takes a field name as the value, such as `/orders?sort=ProductID`. However, this approach can have a negative effect on caching, because query string parameters form part of the resource identifier used by many cache implementations as the key to

cached data.

You can extend this approach to limit the fields returned for each item, if each item contains a large amount of data. For example, you could use a query string parameter that accepts a comma-delimited list of fields, such as `/orders?fields=ProductId,Quantity`.

Give all optional parameters in query strings meaningful defaults. For example, set the `limit` parameter to 10 and the `offset` parameter to 0 if you implement pagination, set the `sort` parameter to the key of the resource if you implement ordering, and set the `fields` parameter to all fields in the resource if you support projections.

Support partial responses for large binary resources

A resource may contain large binary fields, such as files or images. To overcome problems caused by unreliable and intermittent connections and to improve response times, consider enabling such resources to be retrieved in chunks. To do this, the web API should support the `Accept-Ranges` header for GET requests for large resources. This header indicates that the GET operation supports partial requests. The client application can submit GET requests that return a subset of a resource, specified as a range of bytes.

Also, consider implementing HTTP HEAD requests for these resources. A HEAD request is similar to a GET request, except that it only returns the HTTP headers that describe the resource, with an empty message body. A client application can issue a HEAD request to determine whether to fetch a resource by using partial GET requests. For example:

```
HEAD http://adventure-works.com/products/10?fields=productImage HTTP/1.1
```

Here is an example response message:

```
HTTP/1.1 200 OK

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 4580
```

The `Content-Length` header gives the total size of the resource, and the `Accept-Ranges` header indicates that the corresponding GET operation supports partial results. The client application can use this information to retrieve the image in smaller chunks. The first request fetches the first 2500 bytes by using the `Range` header:

```
GET http://adventure-works.com/products/10?fields=productImage HTTP/1.1
Range: bytes=0-2499
```

The response message indicates that this is a partial response by returning HTTP status code 206. The `Content-Length` header specifies the actual number of bytes returned in the message body (not the size of the resource), and the `Content-Range` header indicates which part of the resource this is (bytes 0-2499 out of 4580):

```
HTTP/1.1 206 Partial Content

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2500
Content-Range: bytes 0-2499/4580

[...]
```

A subsequent request from the client application can retrieve the remainder of the resource.

Use HATEOAS to enable navigation to related resources

One of the primary motivations behind REST is that it should be possible to navigate the entire set of resources without requiring prior knowledge of the URI scheme. Each HTTP GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks included in the response, and it should also be provided with information that describes the operations available on each of these resources. This principle is known as HATEOAS, or Hypertext as the Engine of Application State. The system is effectively a finite state machine, and the response to each request contains the information necessary to move from one state to another; no other information should be necessary.

NOTE

Currently there are no standards or specifications that define how to model the HATEOAS principle. The examples shown in this section illustrate one possible solution.

For example, to handle the relationship between an order and a customer, the representation of an order could include links that identify the available operations for the customer of the order. Here is a possible representation:

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"http://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"http://adventure-works.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"http://adventure-works.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"http://adventure-works.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"http://adventure-works.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"http://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 (`http://adventure-works.com/customers/3`), the HTTP method, and the supported MIME types. This is all the information that a client application needs to be able to invoke the operation.

The `links` array also includes self-referencing information about the resource itself that has been retrieved. These have the relationship `self`.

The set of links that are returned may change, depending on the state of the resource. This is what is meant by hypertext being the "engine of application state."

Versioning a RESTful web API

It is highly unlikely that a web API will remain static. As business requirements change new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. While updating a web API to handle new or differing requirements is a relatively straightforward process, you must consider the effects that such changes will have on client applications consuming the web API. The issue is that although the developer designing and implementing a web API has full control over that API, the developer does not have the same degree of control over client applications which may be built by third party organizations operating remotely. The primary imperative is to enable existing client applications to continue functioning unchanged while allowing new client applications to take advantage of new features and resources.

Versioning enables a web API to indicate the features and resources that it exposes, and a client application can submit requests that are directed to a specific version of a feature or resource. The following sections describe several different approaches, each of which has its own benefits and trade-offs.

No versioning

This is the simplest approach, and may be acceptable for some internal APIs. Big changes could be represented as new resources or new links. Adding content to existing resources might not present a breaking change as client applications that are not expecting to see this content will simply ignore it.

For example, a request to the URI `http://adventure-works.com/customers/3` should return the details of a single customer containing `id`, `name`, and `address` fields expected by the client application:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

NOTE

For simplicity, the example responses shown in this section do not include HATEOAS links.

If the `DateCreated` field is added to the schema of the customer resource, then the response would look like this:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":"1 Microsoft Way Redmond WA 98053"}
```

Existing client applications might continue functioning correctly if they are capable of ignoring unrecognized fields, while new client applications can be designed to handle this new field. However, if more radical changes to the

schema of resources occur (such as removing or renaming fields) or the relationships between resources change then these may constitute breaking changes that prevent existing client applications from functioning correctly. In these situations you should consider one of the following approaches.

URI versioning

Each time you modify the web API or change the schema of resources, you add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

Extending the previous example, if the `address` field is restructured into sub-fields containing each constituent part of the address (such as `streetAddress`, `city`, `state`, and `zipCode`), this version of the resource could be exposed through a URI containing a version number, such as <http://adventure-works.com/v2/customers/3>:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"1
Microsoft Way","city":"Redmond","state":"WA","zipCode":98053}}
```

This versioning mechanism is very simple but depends on the server routing the request to the appropriate endpoint. However, it can become unwieldy as the web API matures through several iterations and the server has to support a number of different versions. Also, from a purist's point of view, in all cases the client applications are fetching the same data (customer 3), so the URI should not really be different depending on the version. This scheme also complicates implementation of HATEOAS as all links will need to include the version number in their URIs.

Query string versioning

Rather than providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as <http://adventure-works.com/customers/3?version=2>. The version parameter should default to a meaningful value such as 1 if it is omitted by older client applications.

This approach has the semantic advantage that the same resource is always retrieved from the same URI, but it depends on the code that handles the request to parse the query string and send back the appropriate HTTP response. This approach also suffers from the same complications for implementing HATEOAS as the URI versioning mechanism.

NOTE

Some older web browsers and web proxies will not cache responses for requests that include a query string in the URI. This can have an adverse impact on performance for web applications that use a web API and that run from within such a web browser.

Header versioning

Rather than appending the version number as a query string parameter, you could implement a custom header that indicates the version of the resource. This approach requires that the client application adds the appropriate header to any requests, although the code handling the client request could use a default value (version 1) if the version header is omitted. The following examples utilize a custom header named *Custom-Header*. The value of this header indicates the version of web API.

Version 1:

```
GET http://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=1
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

Version 2:

```
GET http://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=2
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"1
Microsoft Way","city":"Redmond","state":"WA","zipCode":98053}}
```

Note that as with the previous two approaches, implementing HATEOAS requires including the appropriate custom header in any links.

Media type versioning

When a client application sends an HTTP GET request to a web server it should stipulate the format of the content that it can handle by using an *Accept* header, as described earlier in this guidance. Frequently the purpose of the *Accept* header is to allow the client application to specify whether the body of the response should be XML, JSON, or some other common format that the client can parse. However, it is possible to define custom media types that include information enabling the client application to indicate which version of a resource it is expecting. The following example shows a request that specifies an *Accept* header with the value *application/vnd.adventure-works.v1+json*. The *vnd.adventure-works.v1* element indicates to the web server that it should return version 1 of the resource, while the *json* element specifies that the format of the response body should be JSON:

```
GET http://adventure-works.com/customers/3 HTTP/1.1
Accept: application/vnd.adventure-works.v1+json
```

The code handling the request is responsible for processing the *Accept* header and honoring it as far as possible (the client application may specify multiple formats in the *Accept* header, in which case the web server can choose the most appropriate format for the response body). The web server confirms the format of the data in the response body by using the *Content-Type* header:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.adventure-works.v1+json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

If the *Accept* header does not specify any known media types, the web server could generate an HTTP 406 (Not Acceptable) response message or return a message with a default media type.

This approach is arguably the purest of the versioning mechanisms and lends itself naturally to HATEOAS, which can include the MIME type of related data in resource links.

NOTE

When you select a versioning strategy, you should also consider the implications on performance, especially caching on the web server. The URI versioning and Query String versioning schemes are cache-friendly inasmuch as the same URI/query string combination refers to the same data each time.

The Header versioning and Media Type versioning mechanisms typically require additional logic to examine the values in the custom header or the Accept header. In a large-scale environment, many clients using different versions of a web API can result in a significant amount of duplicated data in a server-side cache. This issue can become acute if a client application communicates with a web server through a proxy that implements caching, and that only forwards a request to the web server if it does not currently hold a copy of the requested data in its cache.

Open API Initiative

The [Open API Initiative](#) was created by an industry consortium to standardize REST API descriptions across vendors. As part of this initiative, the Swagger 2.0 specification was renamed the OpenAPI Specification (OAS) and brought under the Open API Initiative.

You may want to adopt OpenAPI for your web APIs. Some points to consider:

- The OpenAPI Specification comes with a set of opinionated guidelines on how a REST API should be designed. That has advantages for interoperability, but requires more care when designing your API to conform to the specification.
- OpenAPI promotes a contract-first approach, rather than an implementation-first approach. Contract-first means you design the API contract (the interface) first and then write code that implements the contract.
- Tools like Swagger can generate client libraries or documentation from API contracts. For example, see [ASP.NET Web API Help Pages using Swagger](#).

More information

- [Microsoft REST API Guidelines](#). Detailed recommendations for designing public REST APIs.
- [The REST Cookbook](#). Introduction to building RESTful APIs.
- [Web API Checklist](#). A useful list of items to consider when designing and implementing a web API.
- [Open API Initiative](#). Documentation and implementation details on Open API.

API implementation

1/4/2018 • 46 minutes to read • [Edit Online](#)

A carefully-designed RESTful web API defines the resources, relationships, and navigation schemes that are accessible to client applications. When you implement and deploy a web API, you should consider the physical requirements of the environment hosting the web API and the way in which the web API is constructed rather than the logical structure of the data. This guidance focusses on best practices for implementing a web API and publishing it to make it available to client applications. For detailed information about web API design, see [API Design Guidance](#).

Processing requests

Consider the following points when you implement the code to handle requests.

GET, PUT, DELETE, HEAD, and PATCH actions should be idempotent

The code that implements these requests should not impose any side-effects. The same request repeated over the same resource should result in the same state. For example, sending multiple DELETE requests to the same URI should have the same effect, although the HTTP status code in the response messages may be different. The first DELETE request might return status code 204 (No Content), while a subsequent DELETE request might return status code 404 (Not Found).

NOTE

The article [Idempotency Patterns](#) on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.

POST actions that create new resources should not have unrelated side-effects

If a POST request is intended to create a new resource, the effects of the request should be limited to the new resource (and possibly any directly related resources if there is some sort of linkage involved). For example, in an ecommerce system, a POST request that creates a new order for a customer might also amend inventory levels and generate billing information, but it should not modify information not directly related to the order or have any other side-effects on the overall state of the system.

Avoid implementing chatty POST, PUT, and DELETE operations

Support POST, PUT and DELETE requests over resource collections. A POST request can contain the details for multiple new resources and add them all to the same collection, a PUT request can replace the entire set of resources in a collection, and a DELETE request can remove an entire collection.

The OData support included in ASP.NET Web API 2 provides the ability to batch requests. A client application can package up several web API requests and send them to the server in a single HTTP request, and receive a single HTTP response that contains the replies to each request. For more information, [Introducing Batch Support in Web API and Web API OData](#).

Follow the HTTP specification when sending a response

A web API must return messages that contain the correct HTTP status code to enable the client to determine how to handle the result, the appropriate HTTP headers so that the client understands the nature of the result, and a suitably formatted body to enable the client to parse the result.

For example, a POST operation should return status code 201 (Created) and the response message should include the URI of the newly created resource in the Location header of the response message.

Support content negotiation

The body of a response message may contain data in a variety of formats. For example, an HTTP GET request could return data in JSON, or XML format. When the client submits a request, it can include an Accept header that specifies the data formats that it can handle. These formats are specified as media types. For example, a client that issues a GET request that retrieves an image can specify an Accept header that lists the media types that the client can handle, such as "image/jpeg, image/gif, image/png". When the web API returns the result, it should format the data by using one of these media types and specify the format in the Content-Type header of the response.

If the client does not specify an Accept header, then use a sensible default format for the response body. As an example, the ASP.NET Web API framework defaults to JSON for text-based data.

Provide links to support HATEOAS-style navigation and discovery of resources

The HATEOAS approach enables a client to navigate and discover resources from an initial starting point. This is achieved by using links containing URLs; when a client issues an HTTP GET request to obtain a resource, the response should contain URLs that enable a client application to quickly locate any directly related resources. For example, in a web API that supports an e-commerce solution, a customer may have placed many orders. When a client application retrieves the details for a customer, the response should include links that enable the client application to send HTTP GET requests that can retrieve these orders. Additionally, HATEOAS-style links should describe the other operations (POST, PUT, DELETE, and so on) that each linked resource supports together with the corresponding URI to perform each request. This approach is described in more detail in [API Design](#).

Currently there are no standards that govern the implementation of HATEOAS, but the following example illustrates one possible approach. In this example, an HTTP GET request that finds the details for a customer returns a response that include HATEOAS links that reference the orders for that customer:

```
GET http://adventure-works.com/customers/2 HTTP/1.1
Accept: text/json
...
```

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
>{"CustomerID":2,"CustomerName":"Bert","Links": [
    {"rel":"self",
     "href":"http://adventure-works.com/customers/2",
     "action":"GET",
     "types":["text/xml","application/json"]},
    {"rel":"self",
     "href":"http://adventure-works.com/customers/2",
     "action":"PUT",
     "types":["application/x-www-form-urlencoded"]},
    {"rel":"self",
     "href":"http://adventure-works.com/customers/2",
     "action":"DELETE",
     "types":[]},
    {"rel":"orders",
     "href":"http://adventure-works.com/customers/2/orders",
     "action":"GET",
     "types":["text/xml","application/json"]},
    {"rel":"orders",
     "href":"http://adventure-works.com/customers/2/orders",
     "action":"POST",
     "types":["application/x-www-form-urlencoded"]}]
}
```

In this example, the customer data is represented by the `Customer` class shown in the following code snippet. The

HATEOAS links are held in the `Links` collection property:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public List<Link> Links { get; set; }
    ...
}

public class Link
{
    public string Rel { get; set; }
    public string Href { get; set; }
    public string Action { get; set; }
    public string [] Types { get; set; }
}
```

The HTTP GET operation retrieves the customer data from storage and constructs a `Customer` object, and then populates the `Links` collection. The result is formatted as a JSON response message. Each link comprises the following fields:

- The relationship between the object being returned and the object described by the link. In this case "self" indicates that the link is a reference back to the object itself (similar to a `this` pointer in many object-oriented languages), and "orders" is the name of a collection containing the related order information.
- The hyperlink (`Href`) for the object being described by the link in the form of a URI.
- The type of HTTP request (`Action`) that can be sent to this URI.
- The format of any data (`Types`) that should be provided in the HTTP request or that can be returned in the response, depending on the type of the request.

The HATEOAS links shown in the example HTTP response indicate that a client application can perform the following operations:

- An HTTP GET request to the URI `http://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 `http://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 `http://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 `http://adventure-works.com/customers/2/orders` to find all the orders for the customer. The data can be returned as XML or JSON.
- An HTTP PUT request to the URI `http://adventure-works.com/customers/2/orders` to create a new order for this customer. The data must be provided in the request message in x-www-form-urlencoded format.

Handling exceptions

Consider the following points if an operation throws an uncaught exception.

Capture exceptions and return a meaningful response to clients

The code that implements an HTTP operation should provide comprehensive exception handling rather than letting uncaught exceptions propagate to the framework. If an exception makes it impossible to complete the operation successfully, the exception can be passed back in the response message, but it should include a meaningful description of the error that caused the exception. The exception should also include the appropriate HTTP status code rather than simply returning status code 500 for every situation. For example, if a user request causes a database update that violates a constraint (such as attempting to delete a customer that has outstanding

orders), you should return status code 409 (Conflict) and a message body indicating the reason for the conflict. If some other condition renders the request unachievable, you can return status code 400 (Bad Request). You can find a full list of HTTP status codes on the [Status Code Definitions](#) page on the W3C website.

The code example traps different conditions and returns an appropriate response.

```
[HttpDelete]
[Route("customers/{id:int}")]
public IHttpActionResult DeleteCustomer(int id)
{
    try
    {
        // Find the customer to be deleted in the repository
        var customerToDelete = repository.GetCustomer(id);

        // If there is no such customer, return an error response
        // with status code 404 (Not Found)
        if (customerToDelete == null)
        {
            return NotFound();
        }

        // Remove the customer from the repository
        // The DeleteCustomer method returns true if the customer
        // was successfully deleted
        if (repository.DeleteCustomer(id))
        {
            // Return a response message with status code 204 (No Content)
            // To indicate that the operation was successful
            return StatusCode(HttpStatusCode.NoContent);
        }
        else
        {
            // Otherwise return a 400 (Bad Request) error response
            return BadRequest(Strings.CustomerNotDeleted);
        }
    }
    catch
    {
        // If an uncaught exception occurs, return an error response
        // with status code 500 (Internal Server Error)
        return InternalServerError();
    }
}
```

TIP

Do not include information that could be useful to an attacker attempting to penetrate your API.

Many web servers trap error conditions themselves before they reach the web API. For example, if you configure authentication for a web site and the user fails to provide the correct authentication information, the web server should respond with status code 401 (Unauthorized). Once a client has been authenticated, your code can perform its own checks to verify that the client should be able access the requested resource. If this authorization fails, you should return status code 403 (Forbidden).

Handle exceptions consistently and log information about errors

To handle exceptions in a consistent manner, consider implementing a global error handling strategy across the entire web API. You should also incorporate error logging which captures the full details of each exception; this error log can contain detailed information as long as it is not made accessible over the web to clients.

Distinguish between client-side errors and server-side errors

The HTTP protocol distinguishes between errors that occur due to the client application (the HTTP 4xx status codes), and errors that are caused by a mishap on the server (the HTTP 5xx status codes). Make sure that you respect this convention in any error response messages.

Optimizing client-side data access

In a distributed environment such as that involving a web server and client applications, one of the primary sources of concern is the network. This can act as a considerable bottleneck, especially if a client application is frequently sending requests or receiving data. Therefore you should aim to minimize the amount of traffic that flows across the network. Consider the following points when you implement the code to retrieve and maintain data:

Support client-side caching

The HTTP 1.1 protocol supports caching in clients and intermediate servers through which a request is routed by the use of the Cache-Control header. When a client application sends an HTTP GET request to the web API, the response can include a Cache-Control header that indicates whether the data in the body of the response can be safely cached by the client or an intermediate server through which the request has been routed, and for how long before it should expire and be considered out-of-date. The following example shows an HTTP GET request and the corresponding response that includes a Cache-Control header:

```
GET http://adventure-works.com/orders/2 HTTP/1.1
```

```
HTTP/1.1 200 OK
...
Cache-Control: max-age=600, private
Content-Type: text/json; charset=utf-8
Content-Length: ...
>{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

In this example, the Cache-Control header specifies that the data returned should be expired after 600 seconds, and is only suitable for a single client and must not be stored in a shared cache used by other clients (it is *private*). The Cache-Control header could specify *public* rather than *private* in which case the data can be stored in a shared cache, or it could specify *no-store* in which case the data must **not** be cached by the client. The following code example shows how to construct a Cache-Control header in a response message:

```

public class OrdersController : ApiController
{
    ...
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...

        // Create a Cache-Control header for the response
        var cacheControlHeader = new CacheControlHeaderValue();
        cacheControlHeader.Private = true;
        cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
        ...

        // Return a response message containing the order and the cache control header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>(order, this)
        {
            CacheControlHeader = cacheControlHeader
        };
        return response;
    }
    ...
}

```

This code makes use of a custom `IHttpActionResult` class named `OkResultWithCaching`. This class enables the controller to set the cache header contents:

```

public class OkResultWithCaching<T> : OkNegotiatedContentResult<T>
{
    public OkResultWithCaching(T content, ApiController controller)
        : base(content, controller) { }

    public OkResultWithCaching(T content, IContentNegotiator contentNegotiator, HttpRequestMessage request,
        IEnumerable<MediaTypeFormatter> formatters)
        : base(content, contentNegotiator, request, formatters) { }

    public CacheControlHeaderValue CacheControlHeader { get; set; }
    public EntityTagHeaderValue ETag { get; set; }

    public override async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        HttpResponseMessage response;
        try
        {
            response = await base.ExecuteAsync(cancellationToken);
            response.Headers.CacheControl = this.CacheControlHeader;
            response.Headers.ETag = ETag;
        }
        catch (OperationCanceledException)
        {
            response = new HttpResponseMessage(HttpStatusCode.Conflict) { ReasonPhrase = "Operation was
cancelled" };
        }
        return response;
    }
}

```

NOTE

The HTTP protocol also defines the *no-cache* directive for the Cache-Control header. Rather confusingly, this directive does not mean "do not cache" but rather "revalidate the cached information with the server before returning it"; the data can still be cached, but it is checked each time it is used to ensure that it is still current.

Cache management is the responsibility of the client application or intermediate server, but if properly implemented it can save bandwidth and improve performance by removing the need to fetch data that has already been recently retrieved.

The *max-age* value in the Cache-Control header is only a guide and not a guarantee that the corresponding data won't change during the specified time. The web API should set the max-age to a suitable value depending on the expected volatility of the data. When this period expires, the client should discard the object from the cache.

NOTE

Most modern web browsers support client-side caching by adding the appropriate cache-control headers to requests and examining the headers of the results, as described. However, some older browsers will not cache the values returned from a URL that includes a query string. This is not usually an issue for custom client applications which implement their own cache management strategy based on the protocol discussed here.

Some older proxies exhibit the same behavior and might not cache requests based on URLs with query strings. This could be an issue for custom client applications that connect to a web server through such a proxy.

Provide ETags to optimize query processing

When a client application retrieves an object, the response message can also include an *ETag* (Entity Tag). An ETag is an opaque string that indicates the version of a resource; each time a resource changes the Etag is also modified. This ETag should be cached as part of the data by the client application. The following code example shows how to add an ETag as part of the response to an HTTP GET request. This code uses the `GetHashCode` method of an object to generate a numeric value that identifies the object (you can override this method if necessary and generate your own hash using an algorithm such as MD5) :

```
public class OrdersController : ApiController
{
    ...
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...

        var hashedOrder = order.GetHashCode();
        string hashedOrderEtag = $"\"{hashedOrder}\"";
        var eTag = new EntityTagHeaderValue(hashedOrderEtag);

        // Return a response message containing the order and the cache control header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>(order, this)
        {
            ...
            ETag = eTag
        };
        return response;
    }
    ...
}
```

The response message posted by the web API looks like this:

```
HTTP/1.1 200 OK
...
Cache-Control: max-age=600, private
Content-Type: text/json; charset=utf-8
ETag: "2147483648"
Content-Length: ...
>{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

TIP

For security reasons, do not allow sensitive data or data returned over an authenticated (HTTPS) connection to be cached.

A client application can issue a subsequent GET request to retrieve the same resource at any time, and if the resource has changed (it has a different ETag) the cached version should be discarded and the new version added to the cache. If a resource is large and requires a significant amount of bandwidth to transmit back to the client, repeated requests to fetch the same data can become inefficient. To combat this, the HTTP protocol defines the following process for optimizing GET requests that you should support in a web API:

- The client constructs a GET request containing the ETag for the currently cached version of the resource referenced in an If-None-Match HTTP header:

```
GET http://adventure-works.com/orders/2 HTTP/1.1
If-None-Match: "2147483648"
```

- The GET operation in the web API obtains the current ETag for the requested data (order 2 in the above example), and compares it to the value in the If-None-Match header.
- If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should return an HTTP response with an empty message body and a status code of 304 (Not Modified).
- If the current ETag for the requested data does not match the ETag provided by the request, then the data has changed and the web API should return an HTTP response with the new data in the message body and a status code of 200 (OK).
- If the requested data no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).
- The client uses the status code to maintain the cache. If the data has not changed (status code 304) then the object can remain cached and the client application should continue to use this version of the object. If the data has changed (status code 200) then the cached object should be discarded and the new one inserted. If the data is no longer available (status code 404) then the object should be removed from the cache.

NOTE

If the response header contains the Cache-Control header no-store then the object should always be removed from the cache regardless of the HTTP status code.

The code below shows the `FindOrderByID` method extended to support the If-None-Match header. Notice that if the If-None-Match header is omitted, the specified order is always retrieved:

```

public class OrdersController : ApiController
{
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        try
        {
            // Find the matching order
            Order order = ...;

            // If there is no such order then return NotFound
            if (order == null)
            {
                return NotFound();
            }

            // Generate the ETag for the order
            var hashedOrder = order.GetHashCode();
            string hashedOrderEtag = $"\"{hashedOrder}\"";

            // Create the Cache-Control and ETag headers for the response
            IHttpActionResult response;
            var cacheControlHeader = new CacheControlHeaderValue();
            cacheControlHeader.Public = true;
            cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
            var eTag = new EntityTagHeaderValue(hashedOrderEtag);

            // Retrieve the If-None-Match header from the request (if it exists)
            var nonMatchEtags = Request.Headers.IfNoneMatch;

            // If there is an ETag in the If-None-Match header and
            // this ETag matches that of the order just retrieved,
            // then create a Not Modified response message
            if (nonMatchEtags.Count > 0 &&
                String.CompareOrdinal(nonMatchEtags.First().Tag, hashedOrderEtag) == 0)
            {
                response = new EmptyResultWithCaching()
                {
                    StatusCode = HttpStatusCode.NotModified,
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag
                };
            }
            // Otherwise create a response message that contains the order details
            else
            {
                response = new OkResultWithCaching<Order>(order, this)
                {
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag
                };
            }
        }

        return response;
    }
    catch
    {
        return InternalServerError();
    }
}
...
}

```

This example incorporates an additional custom `IHttpActionResult` class named `EmptyResultWithCaching`. This class simply acts as a wrapper around an `HttpResponseMessage` object that does not contain a response body:

```

public class EmptyResultWithCaching : IHttpActionResult
{
    public CacheControlHeaderValue CacheControlHeader { get; set; }
    public EntityTagHeaderValue ETag { get; set; }
    public HttpStatusCode StatusCode { get; set; }
    public Uri Location { get; set; }

    public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        HttpResponseMessage response = new HttpResponseMessage(HttpStatusCode);
        response.Headers.CacheControl = this.CacheControlHeader;
        response.Headers.ETag = this.ETag;
        response.Headers.Location = this.Location;
        return response;
    }
}

```

TIP

In this example, the ETag for the data is generated by hashing the data retrieved from the underlying data source. If the ETag can be computed in some other way, then the process can be optimized further and the data only needs to be fetched from the data source if it has changed. This approach is especially useful if the data is large or accessing the data source can result in significant latency (for example, if the data source is a remote database).

Use ETags to Support Optimistic Concurrency

To enable updates over previously cached data, the HTTP protocol supports an optimistic concurrency strategy. If, after fetching and caching a resource, the client application subsequently sends a PUT or DELETE request to change or remove the resource, it should include in If-Match header that references the ETag. The web API can then use this information to determine whether the resource has already been changed by another user since it was retrieved and send an appropriate response back to the client application as follows:

- The client constructs a PUT request containing the new details for the resource and the ETag for the currently cached version of the resource referenced in an If-Match HTTP header. The following example shows a PUT request that updates an order:

```

PUT http://adventure-works.com/orders/1 HTTP/1.1
If-Match: "2282343857"
Content-Type: application/x-www-form-urlencoded
Content-Length: ...
productID=3&quantity=5&orderValue=250

```

- The PUT operation in the web API obtains the current ETag for the requested data (order 1 in the above example), and compares it to the value in the If-Match header.
- If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should perform the update, returning a message with HTTP status code 204 (No Content) if it is successful. The response can include Cache-Control and ETag headers for the updated version of the resource. The response should always include the Location header that references the URI of the newly updated resource.
- If the current ETag for the requested data does not match the ETag provided by the request, then the data has been changed by another user since it was fetched and the web API should return an HTTP response with an empty message body and a status code of 412 (Precondition Failed).
- If the resource to be updated no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).

- The client uses the status code and response headers to maintain the cache. If the data has been updated (status code 204) then the object can remain cached (as long as the Cache-Control header does not specify no-store) but the ETag should be updated. If the data was changed by another user changed (status code 412) or not found (status code 404) then the cached object should be discarded.

The next code example shows an implementation of the PUT operation for the Orders controller:

```

public class OrdersController : ApiController
{
    [HttpPut]
    [Route("api/orders/{id:int}")]
    public IHttpActionResult UpdateExistingOrder(int id, DTOOrder order)
    {
        try
        {
            var baseUri = Constants.GetUriFromConfig();
            var orderToUpdate = this.ordersRepository.GetOrder(id);
            if (orderToUpdate == null)
            {
                return NotFound();
            }

            var hashedOrder = orderToUpdate.GetHashCode();
            string hashedOrderEtag = $"\"{hashedOrder}\"";

            // Retrieve the If-Match header from the request (if it exists)
            var matchEtags = Request.Headers.IfMatch;

            // If there is an Etag in the If-Match header and
            // this etag matches that of the order just retrieved,
            // or if there is no etag, then update the Order
            if (((matchEtags.Count > 0 &&
                  String.CompareOrdinal(matchEtags.First().Tag, hashedOrderEtag) == 0)) ||
                matchEtags.Count == 0)
            {
                // Modify the order
                orderToUpdate.OrderValue = order.OrderValue;
                orderToUpdate.ProductID = order.ProductID;
                orderToUpdate.Quantity = order.Quantity;

                // Save the order back to the data store
                // ...

                // Create the No Content response with Cache-Control, ETag, and Location headers
                var cacheControlHeader = new CacheControlHeaderValue();
                cacheControlHeader.Private = true;
                cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);

                hashedOrder = order.GetHashCode();
                hashedOrderEtag = $"\"{hashedOrder}\"";
                var eTag = new EntityTagHeaderValue(hashedOrderEtag);

                var location = new Uri($"{baseUri}/{Constants.ORDERS}/{id}");
                var response = new EmptyResultWithCaching()
                {
                    StatusCode = HttpStatusCode.NoContent,
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag,
                    Location = location
                };

                return response;
            }
        }
    }
}
// Otherwise return a Precondition Failed response
return StatusCode(HttpStatusCode.PreconditionFailed);
}

```

```
        }
    catch
    {
        return InternalServerError();
    }
}
...
}
```

TIP

Use of the If-Match header is entirely optional, and if it is omitted the web API will always attempt to update the specified order, possibly blindly overwriting an update made by another user. To avoid problems due to lost updates, always provide an If-Match header.

Handling large requests and responses

There may be occasions when a client application needs to issue requests that send or receive data that may be several megabytes (or bigger) in size. Waiting while this amount of data is transmitted could cause the client application to become unresponsive. Consider the following points when you need to handle requests that include significant amounts of data:

Optimize requests and responses that involve large objects

Some resources may be large objects or include large fields, such as graphics images or other types of binary data. A web API should support streaming to enable optimized uploading and downloading of these resources.

The HTTP protocol provides the chunked transfer encoding mechanism to stream large data objects back to a client. When the client sends an HTTP GET request for a large object, the web API can send the reply back in piecemeal *chunks* over an HTTP connection. The length of the data in the reply may not be known initially (it might be generated), so the server hosting the web API should send a response message with each chunk that specifies the Transfer-Encoding: Chunked header rather than a Content-Length header. The client application can receive each chunk in turn to build up the complete response. The data transfer completes when the server sends back a final chunk with zero size.

A single request could conceivably result in a massive object that consumes considerable resources. If, during the streaming process, the web API determines that the amount of data in a request has exceeded some acceptable bounds, it can abort the operation and return a response message with status code 413 (Request Entity Too Large).

You can minimize the size of large objects transmitted over the network by using HTTP compression. This approach helps to reduce the amount of network traffic and the associated network latency, but at the cost of requiring additional processing at the client and the server hosting the web API. For example, a client application that expects to receive compressed data can include an Accept-Encoding: gzip request header (other data compression algorithms can also be specified). If the server supports compression it should respond with the content held in gzip format in the message body and the Content-Encoding: gzip response header.

You can combine encoded compression with streaming; compress the data first before streaming it, and specify the gzip content encoding and chunked transfer encoding in the message headers. Also note that some web servers (such as Internet Information Server) can be configured to automatically compress HTTP responses regardless of whether the web API compresses the data or not.

Implement partial responses for clients that do not support asynchronous operations

As an alternative to asynchronous streaming, a client application can explicitly request data for large objects in chunks, known as partial responses. The client application sends an HTTP HEAD request to obtain information about the object. If the web API supports partial responses it should respond to the HEAD request with a response message that contains an Accept-Ranges header and a Content-Length header that indicates the total size of the object, but the body of the message should be empty. The client application can use this information to construct a

series of GET requests that specify a range of bytes to receive. The web API should return a response message with HTTP status 206 (Partial Content), a Content-Length header that specifies the actual amount of data included in the body of the response message, and a Content-Range header that indicates which part (such as bytes 4000 to 8000) of the object this data represents.

HTTP HEAD requests and partial responses are described in more detail in [API Design](#).

Avoid sending unnecessary 100-Continue status messages in client applications

A client application that is about to send a large amount of data to a server may determine first whether the server is actually willing to accept the request. Prior to sending the data, the client application can submit an HTTP request with an Expect: 100-Continue header, a Content-Length header that indicates the size of the data, but an empty message body. If the server is willing to handle the request, it should respond with a message that specifies the HTTP status 100 (Continue). The client application can then proceed and send the complete request including the data in the message body.

If you are hosting a service by using IIS, the HTTP.sys driver automatically detects and handles Expect: 100-Continue headers before passing requests to your web application. This means that you are unlikely to see these headers in your application code, and you can assume that IIS has already filtered any messages that it deems to be unfit or too large.

If you are building client applications by using the .NET Framework, then all POST and PUT messages will first send messages with Expect: 100-Continue headers by default. As with the server-side, the process is handled transparently by the .NET Framework. However, this process results in each POST and PUT request causing two round-trips to the server, even for small requests. If your application is not sending requests with large amounts of data, you can disable this feature by using the `ServicePointManager` class to create `ServicePoint` objects in the client application. A `ServicePoint` object handles the connections that the client makes to a server based on the scheme and host fragments of URIs that identify resources on the server. You can then set the `Expect100Continue` property of the `ServicePoint` object to false. All subsequent POST and PUT requests made by the client through a URI that matches the scheme and host fragments of the `ServicePoint` object will be sent without Expect: 100-Continue headers. The following code shows how to configure a `ServicePoint` object that configures all requests sent to URIs with a scheme of `http` and a host of `www.contoso.com`.

```
Uri uri = new Uri("http://www.contoso.com/");
ServicePoint sp = ServicePointManager.FindServicePoint(uri);
sp.Expect100Continue = false;
```

You can also set the static `Expect100Continue` property of the `ServicePointManager` class to specify the default value of this property for all subsequently created `ServicePoint` objects. For more information, see [ServicePoint Class](#).

Support pagination for requests that may return large numbers of objects

If a collection contains a large number of resources, issuing a GET request to the corresponding URI could result in significant processing on the server hosting the web API affecting performance, and generate a significant amount of network traffic resulting in increased latency.

To handle these cases, the web API should support query strings that enable the client application to refine requests or fetch data in more manageable, discrete blocks (or pages). The code below shows the `GetAllOrders` method in the `Orders` controller. This method retrieves the details of orders. If this method was unconstrained, it could conceivably return a large amount of data. The `limit` and `offset` parameters are intended to reduce the volume of data to a smaller subset, in this case only the first 10 orders by default:

```

public class OrdersController : ApiController
{
    ...
    [Route("api/orders")]
    [HttpGet]
    public IEnumerable<Order> GetAllOrders(int limit=10, int offset=0)
    {
        // Find the number of orders specified by the limit parameter
        // starting with the order specified by the offset parameter
        var orders = ...;
        return orders;
    }
    ...
}

```

A client application can issue a request to retrieve 30 orders starting at offset 50 by using the URI

<http://www.adventure-works.com/api/orders?limit=30&offset=50>.

TIP

Avoid enabling client applications to specify query strings that result in a URI that is more than 2000 characters long. Many web clients and servers cannot handle URLs that are this long.

Maintaining responsiveness, scalability, and availability

The same web API might be utilized by many client applications running anywhere in the world. It is important to ensure that the web API is implemented to maintain responsiveness under a heavy load, to be scalable to support a highly varying workload, and to guarantee availability for clients that perform business-critical operations.

Consider the following points when determining how to meet these requirements:

Provide asynchronous support for long-running requests

A request that might take a long time to process should be performed without blocking the client that submitted the request. The web API can perform some initial checking to validate the request, initiate a separate task to perform the work, and then return a response message with HTTP code 202 (Accepted). The task could run asynchronously as part of the web API processing, or it could be offloaded to a background task.

The web API should also provide a mechanism to return the results of the processing to the client application. You can achieve this by providing a polling mechanism for client applications to periodically query whether the processing has finished and obtain the result, or enabling the web API to send a notification when the operation has completed.

You can implement a simple polling mechanism by providing a *polling* URI that acts as a virtual resource using the following approach:

1. The client application sends the initial request to the web API.
2. The web API stores information about the request in a table held in table storage or Microsoft Azure Cache, and generates a unique key for this entry, possibly in the form of a GUID.
3. The web API initiates the processing as a separate task. The web API records the state of the task in the table as *Running*.
4. The web API returns a response message with HTTP status code 202 (Accepted), and the GUID of the table entry in the body of the message.
5. When the task has completed, the web API stores the results in the table, and sets the state of the task to *Complete*. Note that if the task fails, the web API could also store information about the failure and set the status to *Failed*.
6. While the task is running, the client can continue performing its own processing. It can periodically send a

request to the URI `/polling/{guid}` where `{guid}` is the GUID returned in the 202 response message by the web API.

7. The web API at the `/polling/{guid}` URI queries the state of the corresponding task in the table and returns a response message with HTTP status code 200 (OK) containing this state (*Running*, *Complete*, or *Failed*). If the task has completed or failed, the response message can also include the results of the processing or any information available about the reason for the failure.

Options for implementing notifications include:

- Using an Azure Notification Hub to push asynchronous responses to client applications. For more information, see [Azure Notification Hubs Notify Users](#).
- Using the Comet model to retain a persistent network connection between the client and the server hosting the web API, and using this connection to push messages from the server back to the client. The MSDN magazine article [Building a Simple Comet Application in the Microsoft .NET Framework](#) describes an example solution.
- Using SignalR to push data in real-time from the web server to the client over a persistent network connection. SignalR is available for ASP.NET web applications as a NuGet package. You can find more information on the [ASP.NET SignalR](#) website.

Ensure that each request is stateless

Each request should be considered atomic. There should be no dependencies between one request made by a client application and any subsequent requests submitted by the same client. This approach assists in scalability; instances of the web service can be deployed on a number of servers. Client requests can be directed at any of these instances and the results should always be the same. It also improves availability for a similar reason; if a web server fails requests can be routed to another instance (by using Azure Traffic Manager) while the server is restarted with no ill effects on client applications.

Track clients and implement throttling to reduce the chances of DOS attacks

If a specific client makes a large number of requests within a given period of time it might monopolize the service and affect the performance of other clients. To mitigate this issue, a web API can monitor calls from client applications either by tracking the IP address of all incoming requests or by logging each authenticated access. You can use this information to limit resource access. If a client exceeds a defined limit, the web API can return a response message with status 503 (Service Unavailable) and include a `Retry-After` header that specifies when the client can send the next request without it being declined. This strategy can help to reduce the chances of a Denial Of Service (DOS) attack from a set of clients stalling the system.

Manage persistent HTTP connections carefully

The HTTP protocol supports persistent HTTP connections where they are available. The HTTP 1.0 specification added the `Connection:Keep-Alive` header that enables a client application to indicate to the server that it can use the same connection to send subsequent requests rather than opening new ones. The connection closes automatically if the client does not reuse the connection within a period defined by the host. This behavior is the default in HTTP 1.1 as used by Azure services, so there is no need to include `Keep-Alive` headers in messages.

Keeping a connection open can help to improve responsiveness by reducing latency and network congestion, but it can be detrimental to scalability by keeping unnecessary connections open for longer than required, limiting the ability of other concurrent clients to connect. It can also affect battery life if the client application is running on a mobile device; if the application only makes occasional requests to the server, maintaining an open connection can cause the battery to drain more quickly. To ensure that a connection is not made persistent with HTTP 1.1, the client can include a `Connection:Close` header with messages to override the default behavior. Similarly, if a server is handling a very large number of clients it can include a `Connection:Close` header in response messages which should close the connection and save server resources.

NOTE

Persistent HTTP connections are a purely optional feature to reduce the network overhead associated with repeatedly establishing a communications channel. Neither the web API nor the client application should depend on a persistent HTTP connection being available. Do not use persistent HTTP connections to implement Comet-style notification systems; instead you should utilize sockets (or websockets if available) at the TCP layer. Finally, note Keep-Alive headers are of limited use if a client application communicates with a server via a proxy; only the connection with the client and the proxy will be persistent.

Publishing and managing a web API

To make a web API available for client applications, the web API must be deployed to a host environment. This environment is typically a web server, although it may be some other type of host process. You should consider the following points when publishing a web API:

- All requests must be authenticated and authorized, and the appropriate level of access control must be enforced.
- A commercial web API might be subject to various quality guarantees concerning response times. It is important to ensure that host environment is scalable if the load can vary significantly over time.
- It may be necessary to meter requests for monetization purposes.
- It might be necessary to regulate the flow of traffic to the web API, and implement throttling for specific clients that have exhausted their quotas.
- Regulatory requirements might mandate logging and auditing of all requests and responses.
- To ensure availability, it may be necessary to monitor the health of the server hosting the web API and restart it if necessary.

It is useful to be able to decouple these issues from the technical issues concerning the implementation of the web API. For this reason, consider creating a [façade](#), running as a separate process and that routes requests to the web API. The façade can provide the management operations and forward validated requests to the web API. Using a façade can also bring many functional advantages, including:

- Acting as an integration point for multiple web APIs.
- Transforming messages and translating communications protocols for clients built by using varying technologies.
- Caching requests and responses to reduce load on the server hosting the web API.

Testing a web API

A web API should be tested as thoroughly as any other piece of software. You should consider creating unit tests to validate the functionality of The nature of a web API brings its own additional requirements to verify that it operates correctly. You should pay particular attention to the following aspects:

- Test all routes to verify that they invoke the correct operations. Be especially aware of HTTP status code 405 (Method Not Allowed) being returned unexpectedly as this can indicate a mismatch between a route and the HTTP methods (GET, POST, PUT, DELETE) that can be dispatched to that route.

Send HTTP requests to routes that do not support them, such as submitting a POST request to a specific resource (POST requests should only be sent to resource collections). In these cases, the only valid response *should* be status code 405 (Not Allowed).

- Verify that all routes are protected properly and are subject to the appropriate authentication and authorization checks.

NOTE

Some aspects of security such as user authentication are most likely to be the responsibility of the host environment rather than the web API, but it is still necessary to include security tests as part of the deployment process.

- Test the exception handling performed by each operation and verify that an appropriate and meaningful HTTP response is passed back to the client application.
- Verify that request and response messages are well-formed. For example, if an HTTP POST request contains the data for a new resource in x-www-form-urlencoded format, confirm that the corresponding operation correctly parses the data, creates the resources, and returns a response containing the details of the new resource, including the correct Location header.
- Verify all links and URIs in response messages. For example, an HTTP POST message should return the URI of the newly-created resource. All HATEOAS links should be valid.
- Ensure that each operation returns the correct status codes for different combinations of input. For example:
 - If a query is successful, it should return status code 200 (OK).
 - If a resource is not found, the operation should return HTTP status code 404 (Not Found).
 - If the client sends a request that successfully deletes a resource, the status code should be 204 (No Content).
 - If the client sends a request that creates a new resource, the status code should be 201 (Created)

Watch out for unexpected response status codes in the 5xx range. These messages are usually reported by the host server to indicate that it was unable to fulfill a valid request.

- Test the different request header combinations that a client application can specify and ensure that the web API returns the expected information in response messages.
- Test query strings. If an operation can take optional parameters (such as pagination requests), test the different combinations and order of parameters.
- Verify that asynchronous operations complete successfully. If the web API supports streaming for requests that return large binary objects (such as video or audio), ensure that client requests are not blocked while the data is streamed. If the web API implements polling for long-running data modification operations, verify that the operations report their status correctly as they proceed.

You should also create and run performance tests to check that the web API operates satisfactorily under duress. You can build a web performance and load test project by using Visual Studio Ultimate. For more information, see [Run performance tests on an application before a release](#).

Using Azure API Management

On Azure, consider using [Azure API Management](#) to publish and manage a web API. Using this facility, you can generate a service that acts as a façade for one or more web APIs. The service is itself a scalable web service that you can create and configure by using the Azure Management portal. You can use this service to publish and manage a web API as follows:

1. Deploy the web API to a website, Azure cloud service, or Azure virtual machine.
2. Connect the API management service to the web API. Requests sent to the URL of the management API are mapped to URIs in the web API. The same API management service can route requests to more than one web API. This enables you to aggregate multiple web APIs into a single management service. Similarly, the same web API can be referenced from more than one API management service if you need to restrict or partition the functionality available to different applications.

NOTE

The URIs in HATEOAS links generated as part of the response for HTTP GET requests should reference the URL of the API management service and not the web server hosting the web API.

3. For each web API, specify the HTTP operations that the web API exposes together with any optional parameters that an operation can take as input. You can also configure whether the API management service should cache the response received from the web API to optimize repeated requests for the same data. Record the details of the HTTP responses that each operation can generate. This information is used to generate documentation for developers, so it is important that it is accurate and complete.

You can either define operations manually using the wizards provided by the Azure Management portal, or you can import them from a file containing the definitions in WADL or Swagger format.

4. Configure the security settings for communications between the API management service and the web server hosting the web API. The API management service currently supports Basic authentication and mutual authentication using certificates, and OAuth 2.0 user authorization.
5. Create a product. A product is the unit of publication; you add the web APIs that you previously connected to the management service to the product. When the product is published, the web APIs become available to developers.

NOTE

Prior to publishing a product, you can also define user-groups that can access the product and add users to these groups. This gives you control over the developers and applications that can use the web API. If a web API is subject to approval, prior to being able to access it a developer must send a request to the product administrator. The administrator can grant or deny access to the developer. Existing developers can also be blocked if circumstances change.

6. Configure policies for each web API. Policies govern aspects such as whether cross-domain calls should be allowed, how to authenticate clients, whether to convert between XML and JSON data formats transparently, whether to restrict calls from a given IP range, usage quotas, and whether to limit the call rate. Policies can be applied globally across the entire product, for a single web API in a product, or for individual operations in a web API.

For more information, see the [API Management Documentation](#).

TIP

Azure provides the Azure Traffic Manager which enables you to implement failover and load-balancing, and reduce latency across multiple instances of a web site hosted in different geographic locations. You can use Azure Traffic Manager in conjunction with the API Management Service; the API Management Service can route requests to instances of a web site through Azure Traffic Manager. For more information, see [Traffic Manager routing Methods](#).

In this structure, if you are using custom DNS names for your web sites, you should configure the appropriate CNAME record for each web site to point to the DNS name of the Azure Traffic Manager web site.

Supporting client-side developers

Developers constructing client applications typically require information on how to access the web API, and documentation concerning the parameters, data types, return types, and return codes that describe the different requests and responses between the web service and the client application.

Document the REST operations for a web API

The Azure API Management Service includes a developer portal that describes the REST operations exposed by a web API. When a product has been published it appears on this portal. Developers can use this portal to sign up for access; the administrator can then approve or deny the request. If the developer is approved, they are assigned a subscription key that is used to authenticate calls from the client applications that they develop. This key must be provided with each web API call otherwise it will be rejected.

This portal also provides:

- Documentation for the product, listing the operations that it exposes, the parameters required, and the different responses that can be returned. Note that this information is generated from the details provided in step 3 in the list in the Publishing a web API by using the Microsoft Azure API Management Service section.
- Code snippets that show how to invoke operations from several languages, including JavaScript, C#, Java, Ruby, Python, and PHP.
- A developers' console that enables a developer to send an HTTP request to test each operation in the product and view the results.
- A page where the developer can report any issues or problems found.

The Azure Management portal enables you to customize the developer portal to change the styling and layout to match the branding of your organization.

Implement a client SDK

Building a client application that invokes REST requests to access a web API requires writing a significant amount of code to construct each request and format it appropriately, send the request to the server hosting the web service, and parse the response to work out whether the request succeeded or failed and extract any data returned. To insulate the client application from these concerns, you can provide an SDK that wraps the REST interface and abstracts these low-level details inside a more functional set of methods. A client application uses these methods, which transparently convert calls into REST requests and then convert the responses back into method return values. This is a common technique that is implemented by many services, including the Azure SDK.

Creating a client-side SDK is a considerable undertaking as it has to be implemented consistently and tested carefully. However, much of this process can be made mechanical, and many vendors supply tools that can automate many of these tasks.

Monitoring a web API

Depending on how you have published and deployed your web API you can monitor the web API directly, or you can gather usage and health information by analyzing the traffic that passes through the API Management service.

Monitoring a web API directly

If you have implemented your web API by using the ASP.NET Web API template (either as a Web API project or as a Web role in an Azure cloud service) and Visual Studio 2013, you can gather availability, performance, and usage data by using ASP.NET Application Insights. Application Insights is a package that transparently tracks and records information about requests and responses when the web API is deployed to the cloud; once the package is installed and configured, you don't need to amend any code in your web API to use it. When you deploy the web API to an Azure web site, all traffic is examined and the following statistics are gathered:

- Server response time.
- Number of server requests and the details of each request.
- The top slowest requests in terms of average response time.
- The details of any failed requests.
- The number of sessions initiated by different browsers and user agents.
- The most frequently viewed pages (primarily useful for web applications rather than web APIs).
- The different user roles accessing the web API.

You can view this data in real time from the Azure Management portal. You can also create webtests that monitor the health of the web API. A webtest sends a periodic request to a specified URI in the web API and captures the response. You can specify the definition of a successful response (such as HTTP status code 200), and if the request does not return this response you can arrange for an alert to be sent to an administrator. If necessary, the administrator can restart the server hosting the web API if it has failed.

For more information, see [Application Insights - Get started with ASP.NET](#).

Monitoring a web API through the API Management Service

If you have published your web API by using the API Management service, the API Management page on the Azure Management portal contains a dashboard that enables you to view the overall performance of the service. The Analytics page enables you to drill down into the details of how the product is being used. This page contains the following tabs:

- **Usage.** This tab provides information about the number of API calls made and the bandwidth used to handle these calls over time. You can filter usage details by product, API, and operation.
- **Health.** This tab enables you to view the outcome of API requests (the HTTP status codes returned), the effectiveness of the caching policy, the API response time, and the service response time. Again, you can filter health data by product, API, and operation.
- **Activity.** This tab provides a text summary of the numbers of successful calls, failed calls, blocked calls, average response time, and response times for each product, web API, and operation. This page also lists the number of calls made by each developer.
- **At a glance.** This tab displays a summary of the performance data, including the developers responsible for making the most API calls, and the products, web APIs, and operations that received these calls.

You can use this information to determine whether a particular web API or operation is causing a bottleneck, and if necessary scale the host environment and add more servers. You can also ascertain whether one or more applications are using a disproportionate volume of resources and apply the appropriate policies to set quotas and limit call rates.

NOTE

You can change the details for a published product, and the changes are applied immediately. For example, you can add or remove an operation from a web API without requiring that you republish the product that contains the web API.

More information

- [ASP.NET Web API OData](#) contains examples and further information on implementing an OData web API by using ASP.NET.
- [Introducing Batch Support in Web API and Web API OData](#) describes how to implement batch operations in a web API by using OData.
- [Idempotency Patterns](#) on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.
- [Status Code Definitions](#) on the W3C website contains a full list of HTTP status codes and their descriptions.
- [Run background tasks with WebJobs](#) provides information and examples on using WebJobs to perform background operations.
- [Azure Notification Hubs Notify Users](#) shows how to use an Azure Notification Hub to push asynchronous responses to client applications.
- [API Management](#) describes how to publish a product that provides controlled and secure access to a web API.
- [Azure API Management REST API Reference](#) describes how to use the API Management REST API to build custom management applications.

- [Traffic Manager Routing Methods](#) summarizes how Azure Traffic Manager can be used to load-balance requests across multiple instances of a website hosting a web API.
- [Application Insights - Get started with ASP.NET](#) provides detailed information on installing and configuring Application Insights in an ASP.NET Web API project.

Autoscaling

2/22/2018 • 13 minutes to read • [Edit Online](#)

Autoscaling is the process of dynamically allocating resources to match performance requirements. As the volume of work grows, an application may need additional resources to maintain the desired performance levels and satisfy service-level agreements (SLAs). As demand slackens and the additional resources are no longer needed, they can be de-allocated to minimize costs.

Autoscaling takes advantage of the elasticity of cloud-hosted environments while easing management overhead. It reduces the need for an operator to continually monitor the performance of a system and make decisions about adding or removing resources.

There are two main ways that an application can scale:

- **Vertical scaling**, also called scaling up and down, means changing the capacity of a resource. For example, you could move an application to a larger VM size. Vertical scaling often requires making the system temporarily unavailable while it is being redeployed. Therefore, it's less common to automate vertical scaling.
- **Horizontal scaling**, also called scaling out and in, means adding or removing instances of a resource. The application continues running without interruption as new resources are provisioned. When the provisioning process is complete, the solution is deployed on these additional resources. If demand drops, the additional resources can be shut down cleanly and deallocated.

Many cloud-based systems, including Microsoft Azure, support automatic horizontal scaling. The rest of this article focuses on horizontal scaling.

NOTE

Autoscaling mostly applies to compute resources. While it's possible to horizontally scale a database or message queue, this usually involves [data partitioning](#), which is generally not automated.

Overview

An autoscaling strategy typically involves the following pieces:

- Instrumentation and monitoring systems at the application, service, and infrastructure levels. These systems capture key metrics, such as response times, queue lengths, CPU utilization, and memory usage.
- Decision-making logic that evaluates these metrics against predefined thresholds or schedules, and decides whether to scale.
- Components that scale the system.
- Testing, monitoring, and tuning of the autoscaling strategy to ensure that it functions as expected.

Azure provides built-in autoscaling mechanisms that address common scenarios. If a particular service or technology does not have built-in autoscaling functionality, or if you have specific autoscaling requirements beyond its capabilities, you might consider a custom implementation. A custom implementation would collect operational and system metrics, analyze the metrics, and then scale resources accordingly.

Configure autoscaling for an Azure solution

Azure provides built-in autoscaling for most compute options.

- **Virtual Machines** support autoscaling through the use of [VM Scale Sets](#), which are a way to manage a set

of Azure virtual machines as a group. See [How to use automatic scaling and Virtual Machine Scale Sets](#).

- **Service Fabric** also supports auto-scaling through VM Scale Sets. Every node type in a Service Fabric cluster is set up as a separate VM scale set. That way, each node type can be scaled in or out independently. See [Scale a Service Fabric cluster in or out using auto-scale rules](#).
- **Azure App Service** has built-in autoscaling. Autoscale settings apply to all of the apps within an App Service. See [Scale instance count manually or automatically](#).
- **Azure Cloud Services** has built-in autoscaling at the role level. See [How to configure auto scaling for a Cloud Service in the portal](#).

These compute options all use [Azure Monitor autoscale](#) to provide a common set of autoscaling functionality.

- **Azure Functions** differs from the previous compute options, because you don't need to configure any autoscale rules. Instead, Azure Functions automatically allocates compute power when your code is running, scaling out as necessary to handle load. For more information, see [Choose the correct hosting plan for Azure Functions](#).

Finally, a custom autoscaling solution can sometimes be useful. For example, you could use Azure diagnostics and application-based metrics, along with custom code to monitor and export the application metrics. Then you could define custom rules based on these metrics, and use Resource Manager REST APIs to trigger autoscaling. However, a custom solution is not simple to implement, and should be considered only if none of the previous approaches can fulfill your requirements.

Use the built-in autoscaling features of the platform, if they meet your requirements. If not, carefully consider whether you really need more complex scaling features. Examples of additional requirements may include more granularity of control, different ways to detect trigger events for scaling, scaling across subscriptions, and scaling other types of resources.

Use Azure Monitor autoscale

[Azure Monitor autoscale](#) provide a common set of autoscaling functionality for VM Scale Sets, Azure App Service, and Azure Cloud Service. Scaling can be performed on a schedule, or based on a runtime metric, such as CPU or memory usage. Examples:

- Scale out to 10 instances on weekdays, and scale in to 4 instances on Saturday and Sunday.
- Scale out by one instance if average CPU usage is above 70%, and scale in by one instance if CPU usage falls below 50%.
- Scale out by one instance if the number of messages in a queue exceeds a certain threshold.

For a list of built-in metrics, see [Azure Monitor autoscaling common metrics](#). You can also implement custom metrics by using Application Insights.

You can configure autoscaling by using PowerShell, the Azure CLI, an Azure Resource Manager template, or the Azure portal. For more detailed control, use the [Azure Resource Manager REST API](#). The [Azure Monitoring Service Management Library](#) and the [Microsoft Insights Library](#) (in preview) are SDKs that allow collecting metrics from different resources, and perform autoscaling by making use of the REST APIs. For resources where Azure Resource Manager support isn't available, or if you are using Azure Cloud Services, the Service Management REST API can be used for autoscaling. In all other cases, use Azure Resource Manager.

Consider the following points when using Azure autoscale:

- Consider whether you can predict the load on the application well enough to use scheduled autoscaling, adding and removing instances to meet anticipated peaks in demand. If this isn't possible, use reactive autoscaling based on runtime metrics, in order to handle unpredictable changes in demand. Typically, you can combine these approaches. For example, create a strategy that adds resources based on a schedule of the times when

you know the application is most busy. This helps to ensure that capacity is available when required, without any delay from starting new instances. For each scheduled rule, define metrics that allow reactive autoscaling during that period to ensure that the application can handle sustained but unpredictable peaks in demand.

- It's often difficult to understand the relationship between metrics and capacity requirements, especially when an application is initially deployed. Provision a little extra capacity at the beginning, and then monitor and tune the autoscaling rules to bring the capacity closer to the actual load.
- Configure the autoscaling rules, and then monitor the performance of your application over time. Use the results of this monitoring to adjust the way in which the system scales if necessary. However, keep in mind that autoscaling is not an instantaneous process. It takes time to react to a metric such as average CPU utilization exceeding (or falling below) a specified threshold.
- Autoscaling rules that use a detection mechanism based on a measured trigger attribute (such as CPU usage or queue length) use an aggregated value over time, rather than instantaneous values, to trigger an autoscaling action. By default, the aggregate is an average of the values. This prevents the system from reacting too quickly, or causing rapid oscillation. It also allows time for new instances that are auto-started to settle into running mode, preventing additional autoscaling actions from occurring while the new instances are starting up. For Azure Cloud Services and Azure Virtual Machines, the default period for the aggregation is 45 minutes, so it can take up to this period of time for the metric to trigger autoscaling in response to spikes in demand. You can change the aggregation period by using the SDK, but be aware that periods of fewer than 25 minutes may cause unpredictable results (for more information, see [Auto Scaling Cloud Services on CPU Percentage with the Azure Monitoring Services Management Library](#)). For Web Apps, the averaging period is much shorter, allowing new instances to be available in about five minutes after a change to the average trigger measure.
- If you configure autoscaling using the SDK rather than the portal, you can specify a more detailed schedule during which the rules are active. You can also create your own metrics and use them with or without any of the existing ones in your autoscaling rules. For example, you may wish to use alternative counters, such as the number of requests per second or the average memory availability, or use custom counters that measure specific business processes.
- When autoscaling Service Fabric, the node types in your cluster are made of VM scale sets at the backend, so you need to set up auto-scale rules for each node type. Take into account the number of nodes that you must have before you set up auto-scaling. The minimum number of nodes that you must have for the primary node type is driven by the reliability level you have chosen. For more info, see [scale a Service Fabric cluster in or out using auto-scale rules](#).
- You can use the portal to link resources such as SQL Database instances and queues to a Cloud Service instance. This allows you to more easily access the separate manual and automatic scaling configuration options for each of the linked resources. For more information, see [How to: Link a resource to a cloud service](#).
- When you configure multiple policies and rules, they could conflict with each other. Autoscale uses the following conflict resolution rules to ensure that there is always a sufficient number of instances running:
 - Scale out operations always take precedence over scale in operations.
 - When scale out operations conflict, the rule that initiates the largest increase in the number of instances takes precedence.
 - When scale in operations conflict, the rule that initiates the smallest decrease in the number of instances takes precedence.
- In an App Service Environment any worker pool or front-end metrics can be used to define autoscale rules. For more information, see [Autoscaling and App Service Environment](#).

Application design considerations

Autoscaling isn't an instant solution. Simply adding resources to a system or running more instances of a process doesn't guarantee that the performance of the system will improve. Consider the following points when designing an autoscaling strategy:

- The system must be designed to be horizontally scalable. Avoid making assumptions about instance affinity; do

not design solutions that require that the code is always running in a specific instance of a process. When scaling a cloud service or web site horizontally, don't assume that a series of requests from the same source will always be routed to the same instance. For the same reason, design services to be stateless to avoid requiring a series of requests from an application to always be routed to the same instance of a service. When designing a service that reads messages from a queue and processes them, don't make any assumptions about which instance of the service handles a specific message. Autoscaling could start additional instances of a service as the queue length grows. The [Competing Consumers Pattern](#) describes how to handle this scenario.

- If the solution implements a long-running task, design this task to support both scaling out and scaling in. Without due care, such a task could prevent an instance of a process from being shut down cleanly when the system scales in, or it could lose data if the process is forcibly terminated. Ideally, refactor a long-running task and break up the processing that it performs into smaller, discrete chunks. The [Pipes and Filters Pattern](#) provides an example of how you can achieve this.
- Alternatively, you can implement a checkpoint mechanism that records state information about the task at regular intervals, and save this state in durable storage that can be accessed by any instance of the process running the task. In this way, if the process is shutdown, the work that it was performing can be resumed from the last checkpoint by using another instance.
- When background tasks run on separate compute instances, such as in worker roles of a cloud services hosted application, you may need to scale different parts of the application using different scaling policies. For example, you may need to deploy additional user interface (UI) compute instances without increasing the number of background compute instances, or the opposite of this. If you offer different levels of service (such as basic and premium service packages), you may need to scale out the compute resources for premium service packages more aggressively than those for basic service packages in order to meet SLAs.
- Consider using the length of the queue over which UI and background compute instances communicate as a criterion for your autoscaling strategy. This is the best indicator of an imbalance or difference between the current load and the processing capacity of the background task.
- If you base your autoscaling strategy on counters that measure business processes, such as the number of orders placed per hour or the average execution time of a complex transaction, ensure that you fully understand the relationship between the results from these types of counters and the actual compute capacity requirements. It may be necessary to scale more than one component or compute unit in response to changes in business process counters.
- To prevent a system from attempting to scale out excessively, and to avoid the costs associated with running many thousands of instances, consider limiting the maximum number of instances that can be automatically added. Most autoscaling mechanisms allow you to specify the minimum and maximum number of instances for a rule. In addition, consider gracefully degrading the functionality that the system provides if the maximum number of instances have been deployed, and the system is still overloaded.
- Keep in mind that autoscaling might not be the most appropriate mechanism to handle a sudden burst in workload. It takes time to provision and start new instances of a service or add resources to a system, and the peak demand may have passed by the time these additional resources have been made available. In this scenario, it may be better to throttle the service. For more information, see the [Throttling Pattern](#).
- Conversely, if you do need the capacity to process all requests when the volume fluctuates rapidly, and cost isn't a major contributing factor, consider using an aggressive autoscaling strategy that starts additional instances more quickly. You can also use a scheduled policy that starts a sufficient number of instances to meet the maximum load before that load is expected.
- The autoscaling mechanism should monitor the autoscaling process, and log the details of each autoscaling event (what triggered it, what resources were added or removed, and when). If you create a custom autoscaling mechanism, ensure that it incorporates this capability. Analyze the information to help measure the effectiveness of the autoscaling strategy, and tune it if necessary. You can tune both in the short term, as the usage patterns become more obvious, and over the long term, as the business expands or the requirements of the application evolve. If an application reaches the upper limit defined for autoscaling, the mechanism might also alert an operator who could manually start additional resources if necessary. Note that, under these circumstances, the operator may also be responsible for manually removing these resources after the workload

eases.

Related patterns and guidance

The following patterns and guidance may also be relevant to your scenario when implementing autoscaling:

- [Throttling Pattern](#). This pattern describes how an application can continue to function and meet SLAs when an increase in demand places an extreme load on resources. Throttling can be used with autoscaling to prevent a system from being overwhelmed while the system scales out.
- [Competing Consumers Pattern](#). This pattern describes how to implement a pool of service instances that can handle messages from any application instance. Autoscaling can be used to start and stop service instances to match the anticipated workload. This approach enables a system to process multiple messages concurrently to optimize throughput, improve scalability and availability, and balance the workload.
- [Monitoring and diagnostics](#). Instrumentation and telemetry are vital for gathering the information that can drive the autoscaling process.

Background jobs

6/28/2018 • 33 minutes to read • [Edit Online](#)

Many types of applications require background tasks that run independently of the user interface (UI). Examples include batch jobs, intensive processing tasks, and long-running processes such as workflows. Background jobs can be executed without requiring user interaction--the application can start the job and then continue to process interactive requests from users. This can help to minimize the load on the application UI, which can improve availability and reduce interactive response times.

For example, if an application is required to generate thumbnails of images that are uploaded by users, it can do this as a background job and save the thumbnail to storage when it is complete--without the user needing to wait for the process to be completed. In the same way, a user placing an order can initiate a background workflow that processes the order, while the UI allows the user to continue browsing the web app. When the background job is complete, it can update the stored orders data and send an email to the user that confirms the order.

When you consider whether to implement a task as a background job, the main criteria is whether the task can run without user interaction and without the UI needing to wait for the job to be completed. Tasks that require the user or the UI to wait while they are completed might not be appropriate as background jobs.

Types of background jobs

Background jobs typically include one or more of the following types of jobs:

- CPU-intensive jobs, such as mathematical calculations or structural model analysis.
- I/O-intensive jobs, such as executing a series of storage transactions or indexing files.
- Batch jobs, such as nightly data updates or scheduled processing.
- Long-running workflows, such as order fulfillment, or provisioning services and systems.
- Sensitive-data processing where the task is handed off to a more secure location for processing. For example, you might not want to process sensitive data within a web app. Instead, you might use a pattern such as [Gatekeeper](#) to transfer the data to an isolated background process that has access to protected storage.

Triggers

Background jobs can be initiated in several different ways. They fall into one of the following categories:

- **Event-driven triggers.** The task is started in response to an event, typically an action taken by a user or a step in a workflow.
- **Schedule-driven triggers.** The task is invoked on a schedule based on a timer. This might be a recurring schedule or a one-off invocation that is specified for a later time.

Event-driven triggers

Event-driven invocation uses a trigger to start the background task. Examples of using event-driven triggers include:

- The UI or another job places a message in a queue. The message contains data about an action that has taken place, such as the user placing an order. The background task listens on this queue and detects the arrival of a new message. It reads the message and uses the data in it as the input to the background job.
- The UI or another job saves or updates a value in storage. The background task monitors the storage and detects changes. It reads the data and uses it as the input to the background job.
- The UI or another job makes a request to an endpoint, such as an HTTPS URI, or an API that is exposed as a

web service. It passes the data that is required to complete the background task as part of the request. The endpoint or web service invokes the background task, which uses the data as its input.

Typical examples of tasks that are suited to event-driven invocation include image processing, workflows, sending information to remote services, sending email messages, and provisioning new users in multitenant applications.

Schedule-driven triggers

Schedule-driven invocation uses a timer to start the background task. Examples of using schedule-driven triggers include:

- A timer that is running locally within the application or as part of the application's operating system invokes a background task on a regular basis.
- A timer that is running in a different application, or a timer service such as Azure Scheduler, sends a request to an API or web service on a regular basis. The API or web service invokes the background task.
- A separate process or application starts a timer that causes the background task to be invoked once after a specified time delay, or at a specific time.

Typical examples of tasks that are suited to schedule-driven invocation include batch-processing routines (such as updating related-products lists for users based on their recent behavior), routine data processing tasks (such as updating indexes or generating accumulated results), data analysis for daily reports, data retention cleanup, and data consistency checks.

If you use a schedule-driven task that must run as a single instance, be aware of the following:

- If the compute instance that is running the scheduler (such as a virtual machine using Windows scheduled tasks) is scaled, you will have multiple instances of the scheduler running. These could start multiple instances of the task.
- If tasks run for longer than the period between scheduler events, the scheduler may start another instance of the task while the previous one is still running.

Returning results

Background jobs execute asynchronously in a separate process, or even in a separate location, from the UI or the process that invoked the background task. Ideally, background tasks are "fire and forget" operations, and their execution progress has no impact on the UI or the calling process. This means that the calling process does not wait for completion of the tasks. Therefore, it cannot automatically detect when the task ends.

If you require a background task to communicate with the calling task to indicate progress or completion, you must implement a mechanism for this. Some examples are:

- Write a status indicator value to storage that is accessible to the UI or caller task, which can monitor or check this value when required. Other data that the background task must return to the caller can be placed into the same storage.
- Establish a reply queue that the UI or caller listens on. The background task can send messages to the queue that indicate status and completion. Data that the background task must return to the caller can be placed into the messages. If you are using Azure Service Bus, you can use the **ReplyTo** and **CorrelationId** properties to implement this capability. For more information, see [Correlation in Service Bus Brokered Messaging](#).
- Expose an API or endpoint from the background task that the UI or caller can access to obtain status information. Data that the background task must return to the caller can be included in the response.
- Have the background task call back to the UI or caller through an API to indicate status at predefined points or on completion. This might be through events raised locally or through a publish-and-subscribe mechanism. Data that the background task must return to the caller can be included in the request or event payload.

Hosting environment

You can host background tasks by using a range of different Azure platform services:

- **Azure Web Apps and WebJobs.** You can use WebJobs to execute custom jobs based on a range of different types of scripts or executable programs within the context of a web app.
- **Azure Virtual Machines.** If you have a Windows service or want to use the Windows Task Scheduler, it is common to host your background tasks within a dedicated virtual machine.
- **Azure Batch.** Batch is a platform service that schedules compute-intensive work to run on a managed collection of virtual machines. It can automatically scale compute resources.
- **Azure Container Service.** Azure Container Service provides a container hosting environment on Azure.
- **Azure Cloud Services.** You can write code within a role that executes as a background task.

The following sections describe each of these options in more detail, and include considerations to help you choose the appropriate option.

Azure Web Apps and WebJobs

You can use Azure WebJobs to execute custom jobs as background tasks within an Azure Web App. WebJobs run within the context of your web app as a continuous process. WebJobs also run in response to a trigger event from Azure Scheduler or external factors, such as changes to storage blobs and message queues. Jobs can be started and stopped on demand, and shut down gracefully. If a continuously running WebJob fails, it is automatically restarted. Retry and error actions are configurable.

When you configure a WebJob:

- If you want the job to respond to an event-driven trigger, you should configure it as **Run continuously**. The script or program is stored in the folder named site/wwwroot/app_data/jobs/continuous.
- If you want the job to respond to a schedule-driven trigger, you should configure it as **Run on a schedule**. The script or program is stored in the folder named site/wwwroot/app_data/jobs/triggered.
- If you choose the **Run on demand** option when you configure a job, it will execute the same code as the **Run on a schedule** option when you start it.

Azure WebJobs run within the sandbox of the web app. This means that they can access environment variables and share information, such as connection strings, with the web app. The job has access to the unique identifier of the machine that is running the job. The connection string named **AzureWebJobsStorage** provides access to Azure storage queues, blobs, and tables for application data, and access to Service Bus for messaging and communication. The connection string named **AzureWebJobsDashboard** provides access to the job action log files.

Azure WebJobs have the following characteristics:

- **Security:** WebJobs are protected by the deployment credentials of the web app.
- **Supported file types:** You can define WebJobs by using command scripts (.cmd), batch files (.bat), PowerShell scripts (.ps1), bash shell scripts (.sh), PHP scripts (.php), Python scripts (.py), JavaScript code (.js), and executable programs (.exe, .jar, and more).
- **Deployment:** You can deploy scripts and executables by using the [Azure portal](#), by using [Visual Studio](#), by using the [Azure WebJobs SDK](#), or by copying them directly to the following locations:
 - For triggered execution: site/wwwroot/app_data/jobs/triggered/{job name}
 - For continuous execution: site/wwwroot/app_data/jobs/continuous/{job name}
- **Logging:** Console.Out is treated (marked) as INFO. Console.Error is treated as ERROR. You can access monitoring and diagnostics information by using the Azure portal. You can download log files directly from the site. They are saved in the following locations:
 - For triggered execution: Vfs/data/jobs/triggered/jobName
 - For continuous execution: Vfs/data/jobs/continuous/jobName
- **Configuration:** You can configure WebJobs by using the portal, the REST API, and PowerShell. You can use a

configuration file named settings.job in the same root directory as the job script to provide configuration information for a job. For example:

- { "stopping_wait_time": 60 }
- { "is_singleton": true }

Considerations

- By default, WebJobs scale with the web app. However, you can configure jobs to run on single instance by setting the **is_singleton** configuration property to **true**. Single instance WebJobs are useful for tasks that you do not want to scale or run as simultaneous multiple instances, such as reindexing, data analysis, and similar tasks.
- To minimize the impact of jobs on the performance of the web app, consider creating an empty Azure Web App instance in a new App Service plan to host WebJobs that may be long running or resource intensive.

More information

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

Azure Virtual Machines

Background tasks might be implemented in a way that prevents them from being deployed to Azure Web Apps or Cloud Services, or these options might not be convenient. Typical examples are Windows services, and third-party utilities and executable programs. Another example might be programs written for an execution environment that is different than that hosting the application. For example, it might be a Unix or Linux program that you want to execute from a Windows or .NET application. You can choose from a range of operating systems for an Azure virtual machine, and run your service or executable on that virtual machine.

To help you choose when to use Virtual Machines, see [Azure App Services, Cloud Services and Virtual Machines comparison](#). For information about the options for Virtual Machines, see [Virtual Machine and Cloud Service sizes for Azure](#). For more information about the operating systems and prebuilt images that are available for Virtual Machines, see [Azure Virtual Machines Marketplace](#).

To initiate the background task in a separate virtual machine, you have a range of options:

- You can execute the task on demand directly from your application by sending a request to an endpoint that the task exposes. This passes in any data that the task requires. This endpoint invokes the task.
- You can configure the task to run on a schedule by using a scheduler or timer that is available in your chosen operating system. For example, on Windows you can use Windows Task Scheduler to execute scripts and tasks. Or, if you have SQL Server installed on the virtual machine, you can use the SQL Server Agent to execute scripts and tasks.
- You can use Azure Scheduler to initiate the task by adding a message to a queue that the task listens on, or by sending a request to an API that the task exposes.

See the earlier section [Triggers](#) for more information about how you can initiate background tasks.

Considerations

Consider the following points when you are deciding whether to deploy background tasks in an Azure virtual machine:

- Hosting background tasks in a separate Azure virtual machine provides flexibility and allows precise control over initiation, execution, scheduling, and resource allocation. However, it will increase runtime cost if a virtual machine must be deployed just to run background tasks.
- There is no facility to monitor the tasks in the Azure portal and no automated restart capability for failed tasks-- although you can monitor the basic status of the virtual machine and manage it by using the [Azure Resource Manager Cmdlets](#). However, there are no facilities to control processes and threads in compute nodes. Typically, using a virtual machine will require additional effort to implement a mechanism that collects data from instrumentation in the task, and from the operating system in the virtual machine. One solution that might be

appropriate is to use the [System Center Management Pack for Azure](#).

- You might consider creating monitoring probes that are exposed through HTTP endpoints. The code for these probes could perform health checks, collect operational information and statistics--or collate error information and return it to a management application. For more information, see [Health Endpoint Monitoring Pattern](#).

More information

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

Azure Batch

Consider [Azure Batch](#) if you need to run large, parallel high-performance computing (HPC) workloads across tens, hundreds, or thousands of VMs.

The Batch service provisions the VMs, assign tasks to the VMs, runs the tasks, and monitors the progress. Batch can automatically scale out the VMs in response to the workload. Batch also provides job scheduling. Azure Batch supports both Linux and Windows VMs.

Considerations

Batch works well with intrinsically parallel workloads. It can also perform parallel calculations with a reduce step at the end, or run [Message Passing Interface \(MPI\) applications](#) for parallel tasks that require message passing between nodes.

An Azure Batch job runs on a pool of nodes (VMs). One approach is to allocate a pool only when needed and then delete it after the job completes. This maximizes utilization, because nodes are not idle, but the job must wait for nodes to be allocated. Alternatively, you can create a pool ahead of time. That approach minimizes the time that it takes for a job to start, but can result in having nodes that sit idle. For more information, see [Pool and compute node lifetime](#).

More information

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

Azure Container Service

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

Containers can be useful for running background jobs. Some of the benefits include:

- Containers support high-density hosting. You can isolate a background task in a container, while placing multiple containers in each VM.
- The container orchestrator handles internal load balancing, configuring the internal network, and other configuration tasks.
- Containers can be started and stopped as needed.
- Azure Container Registry allows you to register your containers inside Azure boundaries. This comes with security, privacy, and proximity benefits.

Considerations

- Requires an understanding of how to use a container orchestrator. Depending on the skillset of your DevOps team, this may or may not be an issue.
- Container Service runs in an IaaS environment. It provisions a cluster of VMs inside a dedicated VNet.

More information

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

Azure Cloud Services

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

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

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

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

Running background tasks in a worker role has several advantages:

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

Considerations

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

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

hosted:

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

Lifecycle

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

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

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

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

Implementation considerations

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

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

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

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

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

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

More information

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

Partitioning

If you decide to include background tasks within an existing compute instance (such as a web app, web role, existing worker role, or virtual machine), you must consider how this will affect the quality attributes of the compute instance and the background task itself. These factors will help you to decide whether to colocate the tasks with the existing compute instance or separate them out into a separate compute instance:

- **Availability:** Background tasks might not need to have the same level of availability as other parts of the application, in particular the UI and other parts that are directly involved in user interaction. Background tasks might be more tolerant of latency, retried connection failures, and other factors that affect availability because the operations can be queued. However, there must be sufficient capacity to prevent the backup of requests that could block queues and affect the application as a whole.
- **Scalability:** Background tasks are likely to have a different scalability requirement than the UI and the interactive parts of the application. Scaling the UI might be necessary to meet peaks in demand, while outstanding background tasks might be completed during less busy times by a fewer number of compute instances.
- **Resiliency:** Failure of a compute instance that just hosts background tasks might not fatally affect the application as a whole if the requests for these tasks can be queued or postponed until the task is available again. If the compute instance and/or tasks can be restarted within an appropriate interval, users of the application might not be affected.
- **Security:** Background tasks might have different security requirements or restrictions than the UI or other parts

of the application. By using a separate compute instance, you can specify a different security environment for the tasks. You can also use patterns such as Gatekeeper to isolate the background compute instances from the UI in order to maximize security and separation.

- **Performance:** You can choose the type of compute instance for background tasks to specifically match the performance requirements of the tasks. This might mean using a less expensive compute option if the tasks do not require the same processing capabilities as the UI, or a larger instance if they require additional capacity and resources.
- **Manageability:** Background tasks might have a different development and deployment rhythm from the main application code or the UI. Deploying them to a separate compute instance can simplify updates and versioning.
- **Cost:** Adding compute instances to execute background tasks increases hosting costs. You should carefully consider the trade-off between additional capacity and these extra costs.

For more information, see [Leader Election Pattern](#) and [Competing Consumers Pattern](#).

Conflicts

If you have multiple instances of a background job, it is possible that they will compete for access to resources and services, such as databases and storage. This concurrent access can result in resource contention, which might cause conflicts in availability of the services and in the integrity of data in storage. You can resolve resource contention by using a pessimistic locking approach. This prevents competing instances of a task from concurrently accessing a service or corrupting data.

Another approach to resolve conflicts is to define background tasks as a singleton, so that there is only ever one instance running. However, this eliminates the reliability and performance benefits that a multiple-instance configuration can provide. This is especially true if the UI can supply sufficient work to keep more than one background task busy.

It is vital to ensure that the background task can automatically restart and that it has sufficient capacity to cope with peaks in demand. You can achieve this by allocating a compute instance with sufficient resources, by implementing a queueing mechanism that can store requests for later execution when demand decreases, or by using a combination of these techniques.

Coordination

The background tasks might be complex and might require multiple individual tasks to execute to produce a result or to fulfil all the requirements. It is common in these scenarios to divide the task into smaller discreet steps or subtasks that can be executed by multiple consumers. Multistep jobs can be more efficient and more flexible because individual steps might be reusable in multiple jobs. It is also easy to add, remove, or modify the order of the steps.

Coordinating multiple tasks and steps can be challenging, but there are three common patterns that you can use to guide your implementation of a solution:

- **Decomposing a task into multiple reusable steps.** An application might be required to perform a variety of tasks of varying complexity on the information that it processes. A straightforward but inflexible approach to implementing this application might be to perform this processing as a monolithic module. However, this approach is likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing are required elsewhere within the application. For more information, see [Pipes and Filters Pattern](#).
- **Managing execution of the steps for a task.** An application might perform tasks that comprise a number of steps (some of which might invoke remote services or access remote resources). The individual steps might be independent of each other, but they are orchestrated by the application logic that implements the task. For more information, see [Scheduler Agent Supervisor Pattern](#).
- **Managing recovery for task steps that fail.** An application might need to undo the work that is performed

by a series of steps (which together define an eventually consistent operation) if one or more of the steps fail. For more information, see [Compensating Transaction Pattern](#).

Resiliency considerations

Background tasks must be resilient in order to provide reliable services to the application. When you are planning and designing background tasks, consider the following points:

- Background tasks must be able to gracefully handle role or service restarts without corrupting data or introducing inconsistency into the application. For long-running or multistep tasks, consider using *check pointing* by saving the state of jobs in persistent storage, or as messages in a queue if this is appropriate. For example, you can persist state information in a message in a queue and incrementally update this state information with the task progress so that the task can be processed from the last known good checkpoint-- instead of restarting from the beginning. When using Azure Service Bus queues, you can use message sessions to enable the same scenario. Sessions allow you to save and retrieve the application processing state by using the [SetState](#) and [GetState](#) methods. For more information about designing reliable multistep processes and workflows, see [Scheduler Agent Supervisor Pattern](#).
- When you use web or worker roles to host multiple background tasks, design your override of the **Run** method to monitor for failed or stalled tasks, and restart them. Where this is not practical, and you are using a worker role, force the worker role to restart by exiting from the **Run** method.
- When you use queues to communicate with background tasks, the queues can act as a buffer to store requests that are sent to the tasks while the application is under higher than usual load. This allows the tasks to catch up with the UI during less busy periods. It also means that recycling the role will not block the UI. For more information, see [Queue-Based Load Leveling Pattern](#). If some tasks are more important than others, consider implementing the [Priority Queue Pattern](#) to ensure that these tasks run before less important ones.
- Background tasks that are initiated by messages or process messages must be designed to handle inconsistencies, such as messages arriving out of order, messages that repeatedly cause an error (often referred to as *poison messages*), and messages that are delivered more than once. Consider the following:
 - Messages that must be processed in a specific order, such as those that change data based on the existing data value (for example, adding a value to an existing value), might not arrive in the original order in which they were sent. Alternatively, they might be handled by different instances of a background task in a different order due to varying loads on each instance. Messages that must be processed in a specific order should include a sequence number, key, or some other indicator that background tasks can use to ensure that they are processed in the correct order. If you are using Azure Service Bus, you can use message sessions to guarantee the order of delivery. However, it is usually more efficient, where possible, to design the process so that the message order is not important.
 - Typically, a background task will peek at messages in the queue, which temporarily hides them from other message consumers. Then it deletes the messages after they have been successfully processed. If a background task fails when processing a message, that message will reappear on the queue after the peek time-out expires. It will be processed by another instance of the task or during the next processing cycle of this instance. If the message consistently causes an error in the consumer, it will block the task, the queue, and eventually the application itself when the queue becomes full. Therefore, it is vital to detect and remove poison messages from the queue. If you are using Azure Service Bus, messages that cause an error can be moved automatically or manually to an associated dead letter queue.
 - Queues are guaranteed at *least once* delivery mechanisms, but they might deliver the same message more than once. In addition, if a background task fails after processing a message but before deleting it from the queue, the message will become available for processing again. Background tasks should be idempotent, which means that processing the same message more than once does not cause an error or inconsistency in the application's data. Some operations are naturally idempotent, such as setting a stored value to a specific new value. However, operations such as adding a value to an existing stored value without checking that the stored value is still the same as when the message was originally sent will cause inconsistencies. Azure Service Bus queues can be configured to automatically remove duplicated

messages.

- Some messaging systems, such as Azure storage queues and Azure Service Bus queues, support a de-queue count property that indicates the number of times a message has been read from the queue. This can be useful in handling repeated and poison messages. For more information, see [Asynchronous Messaging Primer](#) and [Idempotency Patterns](#).

Scaling and performance considerations

Background tasks must offer sufficient performance to ensure they do not block the application, or cause inconsistencies due to delayed operation when the system is under load. Typically, performance is improved by scaling the compute instances that host the background tasks. When you are planning and designing background tasks, consider the following points around scalability and performance:

- Azure supports autoscaling (both scaling out and scaling back in) based on current demand and load--or on a predefined schedule, for Web Apps, Cloud Services web and worker roles, and Virtual Machines hosted deployments. Use this feature to ensure that the application as a whole has sufficient performance capabilities while minimizing runtime costs.
- Where background tasks have a different performance capability from the other parts of a Cloud Services application (for example, the UI or components such as the data access layer), hosting the background tasks together in a separate worker role allows the UI and background task roles to scale independently to manage the load. If multiple background tasks have significantly different performance capabilities from each other, consider dividing them into separate worker roles and scaling each role type independently. However, note that this might increase runtime costs compared to combining all the tasks into fewer roles.
- Simply scaling the roles might not be sufficient to prevent loss of performance under load. You might also need to scale storage queues and other resources to prevent a single point of the overall processing chain from becoming a bottleneck. Also, consider other limitations, such as the maximum throughput of storage and other services that the application and the background tasks rely on.
- Background tasks must be designed for scaling. For example, they must be able to dynamically detect the number of storage queues in use in order to listen on or send messages to the appropriate queue.
- By default, WebJobs scale with their associated Azure Web Apps instance. However, if you want a WebJob to run as only a single instance, you can create a Settings.job file that contains the JSON data `{ "is_singleton": true }`. This forces Azure to only run one instance of the WebJob, even if there are multiple instances of the associated web app. This can be a useful technique for scheduled jobs that must run as only a single instance.

Related patterns

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

More information

- [Scaling Azure Applications with Worker Roles](#)
- [Executing Background Tasks](#)
- [Azure Role Startup Life Cycle \(blog post\)](#)
- [Azure Cloud Services Role Lifecycle \(video\)](#)
- [What is the Azure WebJobs SDK](#)
- [Run Background tasks with WebJobs](#)
- [Azure Queues and Service Bus Queues - Compared and Contrasted](#)
- [How to Enable Diagnostics in a Cloud Service](#)

Caching

1/19/2018 • 56 minutes to read • [Edit Online](#)

Caching is a common technique that aims to improve the performance and scalability of a system. It does this by temporarily copying frequently accessed data to fast storage that's located close to the application. If this fast data storage is located closer to the application than the original source, then caching can significantly improve response times for client applications by serving data more quickly.

Caching is most effective when a client instance repeatedly reads the same data, especially if all the following conditions apply to the original data store:

- It remains relatively static.
- It's slow compared to the speed of the cache.
- It's subject to a high level of contention.
- It's far away when network latency can cause access to be slow.

Caching in distributed applications

Distributed applications typically implement either or both of the following strategies when caching data:

- Using a private cache, where data is held locally on the computer that's running an instance of an application or service.
- Using a shared cache, serving as a common source which can be accessed by multiple processes and/or machines.

In both cases, caching can be performed client-side and/or server-side. Client-side caching is done by the process that provides the user interface for a system, such as a web browser or desktop application. Server-side caching is done by the process that provides the business services that are running remotely.

Private caching

The most basic type of cache is an in-memory store. It's held in the address space of a single process and accessed directly by the code that runs in that process. This type of cache is very quick to access. It can also provide an extremely effective means for storing modest amounts of static data, since the size of a cache is typically constrained by the volume of memory that's available on the machine hosting the process.

If you need to cache more information than is physically possible in memory, you can write cached data to the local file system. This will be slower to access than data that's held in-memory, but should still be faster and more reliable than retrieving data across a network.

If you have multiple instances of an application that uses this model running concurrently, each application instance has its own independent cache holding its own copy of the data.

Think of a cache as a snapshot of the original data at some point in the past. If this data is not static, it is likely that different application instances hold different versions of the data in their caches. Therefore, the same query performed by these instances can return different results, as shown in Figure 1.

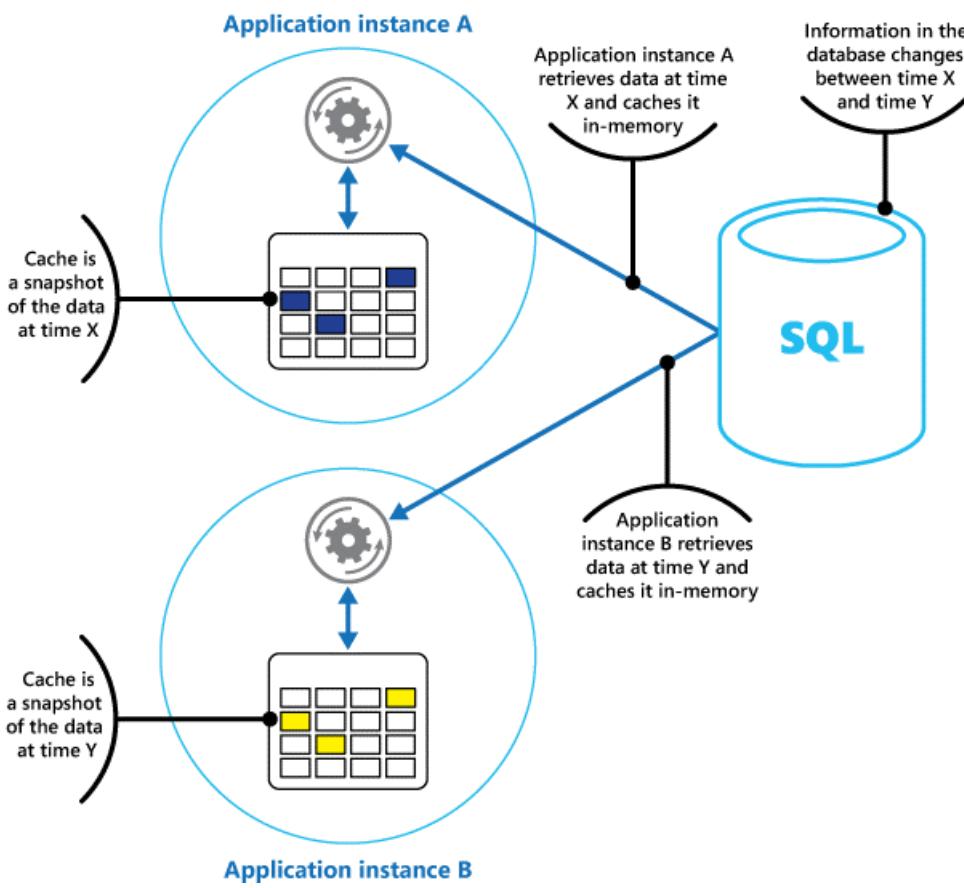


Figure 1: Using an in-memory cache in different instances of an application

Shared caching

Using a shared cache can help alleviate concerns that data might differ in each cache, which can occur with in-memory caching. Shared caching ensures that different application instances see the same view of cached data. It does this by locating the cache in a separate location, typically hosted as part of a separate service, as shown in Figure 2.

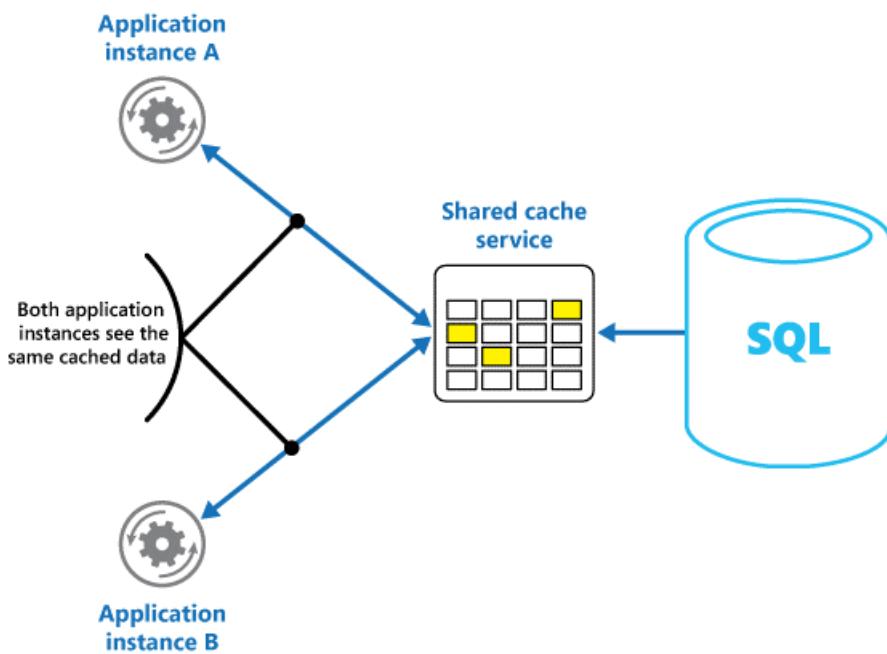


Figure 2: Using a shared cache

An important benefit of the shared caching approach is the scalability it provides. Many shared cache services are implemented by using a cluster of servers, and utilize software that distributes the data across the cluster in a transparent manner. An application instance simply sends a request to the cache service. The underlying

infrastructure is responsible for determining the location of the cached data in the cluster. You can easily scale the cache by adding more servers.

There are two main disadvantages of the shared caching approach:

- The cache is slower to access because it is no longer held locally to each application instance.
- The requirement to implement a separate cache service might add complexity to the solution.

Considerations for using caching

The following sections describe in more detail the considerations for designing and using a cache.

Decide when to cache data

Caching can dramatically improve performance, scalability, and availability. The more data that you have and the larger the number of users that need to access this data, the greater the benefits of caching become. That's because caching reduces the latency and contention that's associated with handling large volumes of concurrent requests in the original data store.

For example, a database might support a limited number of concurrent connections. Retrieving data from a shared cache, however, rather than the underlying database, makes it possible for a client application to access this data even if the number of available connections is currently exhausted. Additionally, if the database becomes unavailable, client applications might be able to continue by using the data that's held in the cache.

Consider caching data that is read frequently but modified infrequently (for example, data that has a higher proportion of read operations than write operations). However, we don't recommend that you use the cache as the authoritative store of critical information. Instead, ensure that all changes that your application cannot afford to lose are always saved to a persistent data store. This means that if the cache is unavailable, your application can still continue to operate by using the data store, and you won't lose important information.

Determine how to cache data effectively

The key to using a cache effectively lies in determining the most appropriate data to cache, and caching it at the appropriate time. The data can be added to the cache on demand the first time it is retrieved by an application. This means that the application needs to fetch the data only once from the data store, and that subsequent access can be satisfied by using the cache.

Alternatively, a cache can be partially or fully populated with data in advance, typically when the application starts (an approach known as seeding). However, it might not be advisable to implement seeding for a large cache because this approach can impose a sudden, high load on the original data store when the application starts running.

Often an analysis of usage patterns can help you decide whether to fully or partially prepopulate a cache, and to choose the data to cache. For example, it can be useful to seed the cache with the static user profile data for customers who use the application regularly (perhaps every day), but not for customers who use the application only once a week.

Caching typically works well with data that is immutable or that changes infrequently. Examples include reference information such as product and pricing information in an e-commerce application, or shared static resources that are costly to construct. Some or all of this data can be loaded into the cache at application startup to minimize demand on resources and to improve performance. It might also be appropriate to have a background process that periodically updates reference data in the cache to ensure it is up to date, or that refreshes the cache when reference data changes.

Caching is less useful for dynamic data, although there are some exceptions to this consideration (see the section Cache highly dynamic data later in this article for more information). When the original data changes regularly, either the cached information becomes stale very quickly or the overhead of synchronizing the cache with the original data store reduces the effectiveness of caching.

Note that a cache does not have to include the complete data for an entity. For example, if a data item represents a multivalued object such as a bank customer with a name, address, and account balance, some of these elements might remain static (such as the name and address), while others (such as the account balance) might be more dynamic. In these situations, it can be useful to cache the static portions of the data and retrieve (or calculate) only the remaining information when it is required.

We recommend that you carry out performance testing and usage analysis to determine whether pre-population or on-demand loading of the cache, or a combination of both, is appropriate. The decision should be based on the volatility and usage pattern of the data. Cache utilization and performance analysis is particularly important in applications that encounter heavy loads and must be highly scalable. For example, in highly scalable scenarios it might make sense to seed the cache to reduce the load on the data store at peak times.

Caching can also be used to avoid repeating computations while the application is running. If an operation transforms data or performs a complicated calculation, it can save the results of the operation in the cache. If the same calculation is required afterward, the application can simply retrieve the results from the cache.

An application can modify data that's held in a cache. However, we recommend thinking of the cache as a transient data store that could disappear at any time. Do not store valuable data in the cache only; make sure that you maintain the information in the original data store as well. This means that if the cache becomes unavailable, you minimize the chance of losing data.

Cache highly dynamic data

When you store rapidly-changing information in a persistent data store, it can impose an overhead on the system. For example, consider a device that continually reports status or some other measurement. If an application chooses not to cache this data on the basis that the cached information will nearly always be outdated, then the same consideration could be true when storing and retrieving this information from the data store. In the time it takes to save and fetch this data, it might have changed.

In a situation such as this, consider the benefits of storing the dynamic information directly in the cache instead of in the persistent data store. If the data is non-critical and does not require auditing, then it doesn't matter if the occasional change is lost.

Manage data expiration in a cache

In most cases, data that's held in a cache is a copy of data that's held in the original data store. The data in the original data store might change after it was cached, causing the cached data to become stale. Many caching systems enable you to configure the cache to expire data and reduce the period for which data may be out of date.

When cached data expires, it's removed from the cache, and the application must retrieve the data from the original data store (it can put the newly-fetched information back into cache). You can set a default expiration policy when you configure the cache. In many cache services, you can also stipulate the expiration period for individual objects when you store them programmatically in the cache. Some caches enable you to specify the expiration period as an absolute value, or as a sliding value that causes the item to be removed from the cache if it is not accessed within the specified time. This setting overrides any cache-wide expiration policy, but only for the specified objects.

NOTE

Consider the expiration period for the cache and the objects that it contains carefully. If you make it too short, objects will expire too quickly and you will reduce the benefits of using the cache. If you make the period too long, you risk the data becoming stale.

It's also possible that the cache might fill up if data is allowed to remain resident for a long time. In this case, any requests to add new items to the cache might cause some items to be forcibly removed in a process known as eviction. Cache services typically evict data on a least-recently-used (LRU) basis, but you can usually override this policy and prevent items from being evicted. However, if you adopt this approach, you risk exceeding the memory that's available in the cache. An application that attempts to add an item to the cache will fail with an exception.

Some caching implementations might provide additional eviction policies. There are several types of eviction policies. These include:

- A most-recently-used policy (in the expectation that the data will not be required again).
- A first-in-first-out policy (oldest data is evicted first).
- An explicit removal policy based on a triggered event (such as the data being modified).

Invalidate data in a client-side cache

Data that's held in a client-side cache is generally considered to be outside the auspices of the service that provides the data to the client. A service cannot directly force a client to add or remove information from a client-side cache.

This means that it's possible for a client that uses a poorly configured cache to continue using outdated information. For example, if the expiration policies of the cache aren't properly implemented, a client might use outdated information that's cached locally when the information in the original data source has changed.

If you are building a web application that serves data over an HTTP connection, you can implicitly force a web client (such as a browser or web proxy) to fetch the most recent information. You can do this if a resource is updated by a change in the URI of that resource. Web clients typically use the URI of a resource as the key in the client-side cache, so if the URI changes, the web client ignores any previously cached versions of a resource and fetches the new version instead.

Managing concurrency in a cache

Caches are often designed to be shared by multiple instances of an application. Each application instance can read and modify data in the cache. Consequently, the same concurrency issues that arise with any shared data store also apply to a cache. In a situation where an application needs to modify data that's held in the cache, you might need to ensure that updates made by one instance of the application do not overwrite the changes made by another instance.

Depending on the nature of the data and the likelihood of collisions, you can adopt one of two approaches to concurrency:

- **Optimistic.** Immediately prior to updating the data, the application checks to see whether the data in the cache has changed since it was retrieved. If the data is still the same, the change can be made. Otherwise, the application has to decide whether to update it. (The business logic that drives this decision will be application-specific.) This approach is suitable for situations where updates are infrequent, or where collisions are unlikely to occur.
- **Pessimistic.** When it retrieves the data, the application locks it in the cache to prevent another instance from changing it. This process ensures that collisions cannot occur, but they can also block other instances that need to process the same data. Pessimistic concurrency can affect the scalability of a solution and is recommended only for short-lived operations. This approach might be appropriate for situations where collisions are more likely, especially if an application updates multiple items in the cache and must ensure that these changes are applied consistently.

Implement high availability and scalability, and improve performance

Avoid using a cache as the primary repository of data; this is the role of the original data store from which the cache is populated. The original data store is responsible for ensuring the persistence of the data.

Be careful not to introduce critical dependencies on the availability of a shared cache service into your solutions. An application should be able to continue functioning if the service that provides the shared cache is unavailable. The application should not hang or fail while waiting for the cache service to resume.

Therefore, the application must be prepared to detect the availability of the cache service and fall back to the original data store if the cache is inaccessible. The [Circuit-Breaker pattern](#) is useful for handling this scenario. The service that provides the cache can be recovered, and once it becomes available, the cache can be repopulated as

data is read from the original data store, following a strategy such as the [Cache-aside pattern](#).

However, there might be a scalability impact on the system if the application falls back to the original data store when the cache is temporarily unavailable. While the data store is being recovered, the original data store could be swamped with requests for data, resulting in timeouts and failed connections.

Consider implementing a local, private cache in each instance of an application, together with the shared cache that all application instances access. When the application retrieves an item, it can check first in its local cache, then in the shared cache, and finally in the original data store. The local cache can be populated using the data in either the shared cache, or in the database if the shared cache is unavailable.

This approach requires careful configuration to prevent the local cache from becoming too stale with respect to the shared cache. However, the local cache acts as a buffer if the shared cache is unreachable. Figure 3 shows this structure.

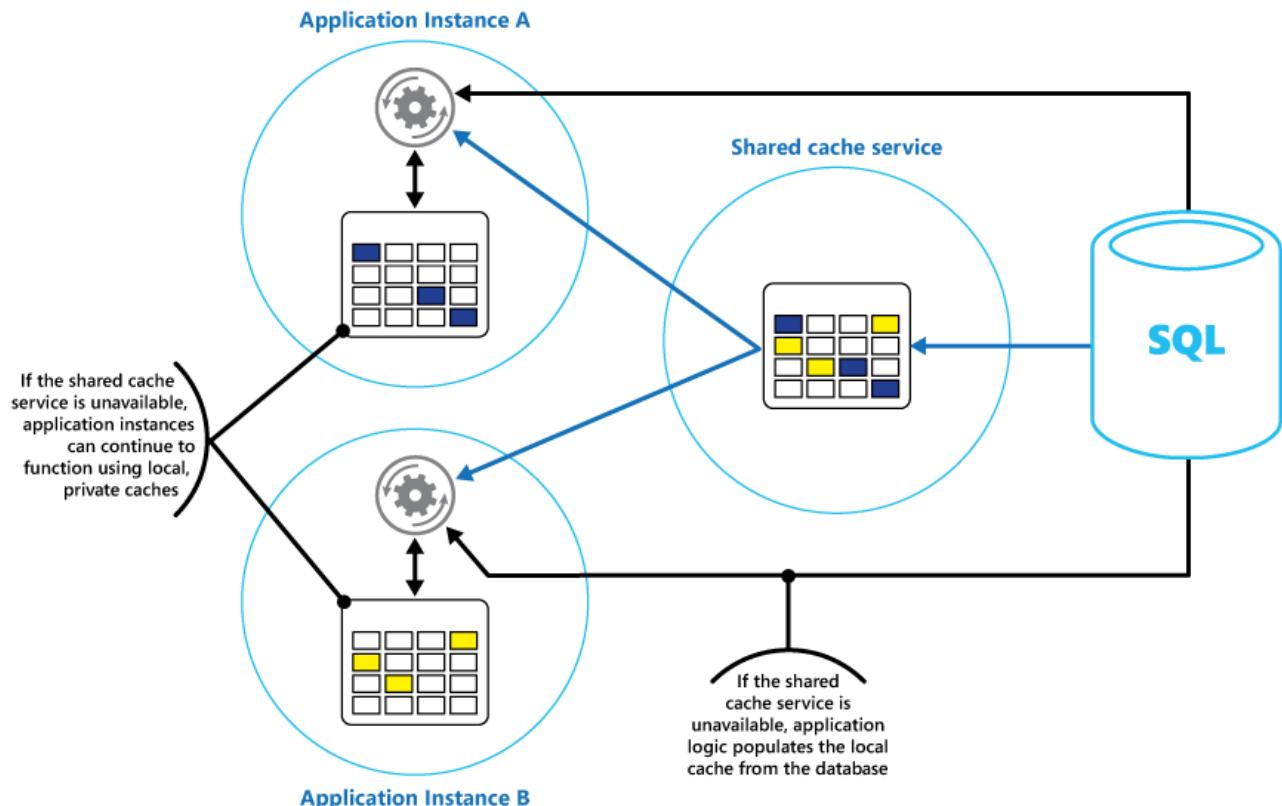


Figure 3: Using a local, private cache with a shared cache

To support large caches that hold relatively long-lived data, some cache services provide a high-availability option that implements automatic failover if the cache becomes unavailable. This approach typically involves replicating the cached data that's stored on a primary cache server to a secondary cache server, and switching to the secondary server if the primary server fails or connectivity is lost.

To reduce the latency that's associated with writing to multiple destinations, the replication to the secondary server might occur asynchronously when data is written to the cache on the primary server. This approach leads to the possibility that some cached information might be lost in the event of a failure, but the proportion of this data should be small compared to the overall size of the cache.

If a shared cache is large, it might be beneficial to partition the cached data across nodes to reduce the chances of contention and improve scalability. Many shared caches support the ability to dynamically add (and remove) nodes and rebalance the data across partitions. This approach might involve clustering, in which the collection of nodes is presented to client applications as a seamless, single cache. Internally, however, the data is dispersed between nodes following a predefined distribution strategy that balances the load evenly. The [Data partitioning guidance document](#) on the Microsoft website provides more information about possible partitioning strategies.

Clustering can also increase the availability of the cache. If a node fails, the remainder of the cache is still accessible.

Clustering is frequently used in conjunction with replication and failover. Each node can be replicated, and the replica can be quickly brought online if the node fails.

Many read and write operations are likely to involve single data values or objects. However, at times it might be necessary to store or retrieve large volumes of data quickly. For example, seeding a cache could involve writing hundreds or thousands of items to the cache. An application might also need to retrieve a large number of related items from the cache as part of the same request.

Many large-scale caches provide batch operations for these purposes. This enables a client application to package up a large volume of items into a single request and reduces the overhead that's associated with performing a large number of small requests.

Caching and eventual consistency

For the cache-aside pattern to work, the instance of the application that populates the cache must have access to the most recent and consistent version of the data. In a system that implements eventual consistency (such as a replicated data store) this might not be the case.

One instance of an application could modify a data item and invalidate the cached version of that item. Another instance of the application might attempt to read this item from a cache, which causes a cache-miss, so it reads the data from the data store and adds it to the cache. However, if the data store has not been fully synchronized with the other replicas, the application instance could read and populate the cache with the old value.

For more information about handling data consistency, see the [Data consistency primer](#).

Protect cached data

Irrespective of the cache service you use, consider how to protect the data that's held in the cache from unauthorized access. There are two main concerns:

- The privacy of the data in the cache.
- The privacy of data as it flows between the cache and the application that's using the cache.

To protect data in the cache, the cache service might implement an authentication mechanism that requires that applications specify the following:

- Which identities can access data in the cache.
- Which operations (read and write) that these identities are allowed to perform.

To reduce overhead that's associated with reading and writing data, after an identity has been granted write and/or read access to the cache, that identity can use any data in the cache.

If you need to restrict access to subsets of the cached data, you can do one of the following:

- Split the cache into partitions (by using different cache servers) and only grant access to identities for the partitions that they should be allowed to use.
- Encrypt the data in each subset by using different keys, and provide the encryption keys only to identities that should have access to each subset. A client application might still be able to retrieve all of the data in the cache, but it will only be able to decrypt the data for which it has the keys.

You must also protect the data as it flows in and out of the cache. To do this, you depend on the security features provided by the network infrastructure that client applications use to connect to the cache. If the cache is implemented using an on-site server within the same organization that hosts the client applications, then the isolation of the network itself might not require you to take additional steps. If the cache is located remotely and requires a TCP or HTTP connection over a public network (such as the Internet), consider implementing SSL.

Considerations for implementing caching with Microsoft Azure

[Azure Redis Cache](#) is an implementation of the open source Redis cache that runs as a service in an Azure datacenter. It provides a caching service that can be accessed from any Azure application, whether the application is implemented as a cloud service, a website, or inside an Azure virtual machine. Caches can be shared by client applications that have the appropriate access key.

Azure Redis Cache is a high-performance caching solution that provides availability, scalability and security. It typically runs as a service spread across one or more dedicated machines. It attempts to store as much information as it can in memory to ensure fast access. This architecture is intended to provide low latency and high throughput by reducing the need to perform slow I/O operations.

Azure Redis Cache is compatible with many of the various APIs that are used by client applications. If you have existing applications that already use Azure Redis Cache running on-premises, the Azure Redis Cache provides a quick migration path to caching in the cloud.

Features of Redis

Redis is more than a simple cache server. It provides a distributed in-memory database with an extensive command set that supports many common scenarios. These are described later in this document, in the section [Using Redis caching](#). This section summarizes some of the key features that Redis provides.

Redis as an in-memory database

Redis supports both read and write operations. In Redis, writes can be protected from system failure either by being stored periodically in a local snapshot file or in an append-only log file. This is not the case in many caches (which should be considered transitory data stores).

All writes are asynchronous and do not block clients from reading and writing data. When Redis starts running, it reads the data from the snapshot or log file and uses it to construct the in-memory cache. For more information, see [Redis persistence](#) on the Redis website.

NOTE

Redis does not guarantee that all writes will be saved in the event of a catastrophic failure, but at worst you might lose only a few seconds worth of data. Remember that a cache is not intended to act as an authoritative data source, and it is the responsibility of the applications using the cache to ensure that critical data is saved successfully to an appropriate data store. For more information, see the [cache-aside pattern](#).

Redis data types

Redis is a key-value store, where values can contain simple types or complex data structures such as hashes, lists, and sets. It supports a set of atomic operations on these data types. Keys can be permanent or tagged with a limited time-to-live, at which point the key and its corresponding value are automatically removed from the cache. For more information about Redis keys and values, visit the page [An introduction to Redis data types and abstractions](#) on the Redis website.

Redis replication and clustering

Redis supports master/subordinate replication to help ensure availability and maintain throughput. Write operations to a Redis master node are replicated to one or more subordinate nodes. Read operations can be served by the master or any of the subordinates.

In the event of a network partition, subordinates can continue to serve data and then transparently resynchronize with the master when the connection is reestablished. For further details, visit the [Replication](#) page on the Redis website.

Redis also provides clustering, which enables you to transparently partition data into shards across servers and spread the load. This feature improves scalability, because new Redis servers can be added and the data repartitioned as the size of the cache increases.

Furthermore, each server in the cluster can be replicated by using master/subordinate replication. This ensures

availability across each node in the cluster. For more information about clustering and sharding, visit the [Redis cluster tutorial page](#) on the Redis website.

Redis memory use

A Redis cache has a finite size that depends on the resources available on the host computer. When you configure a Redis server, you can specify the maximum amount of memory it can use. You can also configure a key in a Redis cache to have an expiration time, after which it is automatically removed from the cache. This feature can help prevent the in-memory cache from filling with old or stale data.

As memory fills up, Redis can automatically evict keys and their values by following a number of policies. The default is LRU (least recently used), but you can also select other policies such as evicting keys at random or turning off eviction altogether (in which case attempts to add items to the cache fail if it is full). The page [Using Redis as an LRU cache](#) provides more information.

Redis transactions and batches

Redis enables a client application to submit a series of operations that read and write data in the cache as an atomic transaction. All the commands in the transaction are guaranteed to run sequentially, and no commands issued by other concurrent clients will be interwoven between them.

However, these are not true transactions as a relational database would perform them. Transaction processing consists of two stages--the first is when the commands are queued, and the second is when the commands are run. During the command queuing stage, the commands that comprise the transaction are submitted by the client. If some sort of error occurs at this point (such as a syntax error, or the wrong number of parameters) then Redis refuses to process the entire transaction and discards it.

During the run phase, Redis performs each queued command in sequence. If a command fails during this phase, Redis continues with the next queued command and does not roll back the effects of any commands that have already been run. This simplified form of transaction helps to maintain performance and avoid performance problems that are caused by contention.

Redis does implement a form of optimistic locking to assist in maintaining consistency. For detailed information about transactions and locking with Redis, visit the [Transactions page](#) on the Redis website.

Redis also supports non-transactional batching of requests. The Redis protocol that clients use to send commands to a Redis server enables a client to send a series of operations as part of the same request. This can help to reduce packet fragmentation on the network. When the batch is processed, each command is performed. If any of these commands are malformed, they will be rejected (which doesn't happen with a transaction), but the remaining commands will be performed. There is also no guarantee about the order in which the commands in the batch will be processed.

Redis security

Redis is focused purely on providing fast access to data, and is designed to run inside a trusted environment that can be accessed only by trusted clients. Redis supports a limited security model based on password authentication. (It is possible to remove authentication completely, although we don't recommend this.)

All authenticated clients share the same global password and have access to the same resources. If you need more comprehensive sign-in security, you must implement your own security layer in front of the Redis server, and all client requests should pass through this additional layer. Redis should not be directly exposed to untrusted or unauthenticated clients.

You can restrict access to commands by disabling them or renaming them (and by providing only privileged clients with the new names).

Redis does not directly support any form of data encryption, so all encoding must be performed by client applications. Additionally, Redis does not provide any form of transport security. If you need to protect data as it flows across the network, we recommend implementing an SSL proxy.

For more information, visit the [Redis security](#) page on the Redis website.

NOTE

Azure Redis Cache provides its own security layer through which clients connect. The underlying Redis servers are not exposed to the public network.

Azure Redis cache

Azure Redis Cache provides access to Redis servers that are hosted at an Azure datacenter. It acts as a façade that provides access control and security. You can provision a cache by using the Azure portal.

The portal provides a number of predefined configurations. These range from a 53 GB cache running as a dedicated service that supports SSL communications (for privacy) and master/subordinate replication with an SLA of 99.9% availability, down to a 250 MB cache without replication (no availability guarantees) running on shared hardware.

Using the Azure portal, you can also configure the eviction policy of the cache, and control access to the cache by adding users to the roles provided. These roles, which define the operations that members can perform, include Owner, Contributor, and Reader. For example, members of the Owner role have complete control over the cache (including security) and its contents, members of the Contributor role can read and write information in the cache, and members of the Reader role can only retrieve data from the cache.

Most administrative tasks are performed through the Azure portal. For this reason, many of the administrative commands that are available in the standard version of Redis are not available, including the ability to modify the configuration programmatically, shut down the Redis server, configure additional subordinates, or forcibly save data to disk.

The Azure portal includes a convenient graphical display that enables you to monitor the performance of the cache. For example, you can view the number of connections being made, the number of requests being performed, the volume of reads and writes, and the number of cache hits versus cache misses. Using this information, you can determine the effectiveness of the cache and if necessary, switch to a different configuration or change the eviction policy.

Additionally, you can create alerts that send email messages to an administrator if one or more critical metrics fall outside of an expected range. For example, you might want to alert an administrator if the number of cache misses exceeds a specified value in the last hour, because it means the cache might be too small or data might be being evicted too quickly.

You can also monitor the CPU, memory, and network usage for the cache.

For further information and examples showing how to create and configure an Azure Redis Cache, visit the page [Lap around Azure Redis Cache](#) on the Azure blog.

Caching session state and HTML output

If you're building ASP.NET web applications that run by using Azure web roles, you can save session state information and HTML output in an Azure Redis Cache. The session state provider for Azure Redis Cache enables you to share session information between different instances of an ASP.NET web application, and is very useful in web farm situations where client-server affinity is not available and caching session data in-memory would not be appropriate.

Using the session state provider with Azure Redis Cache delivers several benefits, including:

- Sharing session state with a large number of instances of ASP.NET web applications.
- Providing improved scalability.
- Supporting controlled, concurrent access to the same session state data for multiple readers and a single writer.

- Using compression to save memory and improve network performance.

For more information, see [ASP.NET session state provider for Azure Redis Cache](#).

NOTE

Do not use the session state provider for Azure Redis Cache with ASP.NET applications that run outside of the Azure environment. The latency of accessing the cache from outside of Azure can eliminate the performance benefits of caching data.

Similarly, the output cache provider for Azure Redis Cache enables you to save the HTTP responses generated by an ASP.NET web application. Using the output cache provider with Azure Redis Cache can improve the response times of applications that render complex HTML output. Application instances that generate similar responses can make use of the shared output fragments in the cache rather than generating this HTML output afresh. For more information, see [ASP.NET output cache provider for Azure Redis Cache](#).

Building a custom Redis cache

Azure Redis Cache acts as a façade to the underlying Redis servers. Currently it supports a fixed set of configurations but does not provide for Redis clustering. If you require an advanced configuration that is not covered by the Azure Redis cache (such as a cache bigger than 53 GB) you can build and host your own Redis servers by using Azure virtual machines.

This is a potentially complex process because you might need to create several VMs to act as master and subordinate nodes if you want to implement replication. Furthermore, if you wish to create a cluster, then you need multiple masters and subordinate servers. A minimal clustered replication topology that provides a high degree of availability and scalability comprises at least six VMs organized as three pairs of master/subordinate servers (a cluster must contain at least three master nodes).

Each master/subordinate pair should be located close together to minimize latency. However, each set of pairs can be running in different Azure datacenters located in different regions, if you wish to locate cached data close to the applications that are most likely to use it. For an example of building and configuring a Redis node running as an Azure VM, see [Running Redis on a CentOS Linux VM in Azure](#).

NOTE

Please note that if you implement your own Redis cache in this way, you are responsible for monitoring, managing, and securing the service.

Partitioning a Redis cache

Partitioning the cache involves splitting the cache across multiple computers. This structure gives you several advantages over using a single cache server, including:

- Creating a cache that is much bigger than can be stored on a single server.
- Distributing data across servers, improving availability. If one server fails or becomes inaccessible, the data that it holds is unavailable, but the data on the remaining servers can still be accessed. For a cache, this is not crucial because the cached data is only a transient copy of the data that's held in a database. Cached data on a server that becomes inaccessible can be cached on a different server instead.
- Spreading the load across servers, thereby improving performance and scalability.
- Geolocating data close to the users that access it, thus reducing latency.

For a cache, the most common form of partitioning is sharding. In this strategy, each partition (or shard) is a Redis cache in its own right. Data is directed to a specific partition by using sharding logic, which can use a variety of

approaches to distribute the data. The [Sharding pattern](#) provides more information about implementing sharding.

To implement partitioning in a Redis cache, you can take one of the following approaches:

- *Server-side query routing.* In this technique, a client application sends a request to any of the Redis servers that comprise the cache (probably the closest server). Each Redis server stores metadata that describes the partition that it holds, and also contains information about which partitions are located on other servers. The Redis server examines the client request. If it can be resolved locally, it will perform the requested operation. Otherwise it will forward the request on to the appropriate server. This model is implemented by Redis clustering, and is described in more detail on the [Redis cluster tutorial](#) page on the Redis website. Redis clustering is transparent to client applications, and additional Redis servers can be added to the cluster (and the data re-partitioned) without requiring that you reconfigure the clients.
- *Client-side partitioning.* In this model, the client application contains logic (possibly in the form of a library) that routes requests to the appropriate Redis server. This approach can be used with Azure Redis Cache. Create multiple Azure Redis Caches (one for each data partition) and implement the client-side logic that routes the requests to the correct cache. If the partitioning scheme changes (if additional Azure Redis Caches are created, for example), client applications might need to be reconfigured.
- *Proxy-assisted partitioning.* In this scheme, client applications send requests to an intermediary proxy service which understands how the data is partitioned and then routes the request to the appropriate Redis server. This approach can also be used with Azure Redis Cache; the proxy service can be implemented as an Azure cloud service. This approach requires an additional level of complexity to implement the service, and requests might take longer to perform than using client-side partitioning.

The page [Partitioning: how to split data among multiple Redis instances](#) on the Redis website provides further information about implementing partitioning with Redis.

Implement Redis cache client applications

Redis supports client applications written in numerous programming languages. If you are building new applications by using the .NET Framework, the recommended approach is to use the StackExchange.Redis client library. This library provides a .NET Framework object model that abstracts the details for connecting to a Redis server, sending commands, and receiving responses. It is available in Visual Studio as a NuGet package. You can use this same library to connect to an Azure Redis Cache, or a custom Redis cache hosted on a VM.

To connect to a Redis server you use the static `Connect` method of the `ConnectionMultiplexer` class. The connection that this method creates is designed to be used throughout the lifetime of the client application, and the same connection can be used by multiple concurrent threads. Do not reconnect and disconnect each time you perform a Redis operation because this can degrade performance.

You can specify the connection parameters, such as the address of the Redis host and the password. If you are using Azure Redis Cache, the password is either the primary or secondary key that is generated for Azure Redis Cache by using the Azure Management portal.

After you have connected to the Redis server, you can obtain a handle on the Redis database that acts as the cache. The Redis connection provides the `GetDatabase` method to do this. You can then retrieve items from the cache and store data in the cache by using the `StringGet` and `StringSet` methods. These methods expect a key as a parameter, and return the item either in the cache that has a matching value (`StringGet`) or add the item to the cache with this key (`StringSet`).

Depending on the location of the Redis server, many operations might incur some latency while a request is transmitted to the server and a response is returned to the client. The StackExchange library provides asynchronous versions of many of the methods that it exposes to help client applications remain responsive. These methods support the [Task-based Asynchronous Pattern](#) in the .NET Framework.

The following code snippet shows a method named `RetrieveItem`. It illustrates an implementation of the cache-aside pattern based on Redis and the StackExchange library. The method takes a string key value and attempts to

retrieve the corresponding item from the Redis cache by calling the `StringGetAsync` method (the asynchronous version of `StringGet`).

If the item is not found, it is fetched from the underlying data source using the `GetItemFromDataSourceAsync` method (which is a local method and not part of the StackExchange library). It's then added to the cache by using the `StringSetAsync` method so it can be retrieved more quickly next time.

```
// Connect to the Azure Redis cache
ConfigurationOptions config = new ConfigurationOptions();
config.EndPoints.Add("<your DNS name>.redis.cache.windows.net");
config.Password = "<Redis cache key from management portal>";
ConnectionMultiplexer redisHostConnection = ConnectionMultiplexer.Connect(config);
IDatabase cache = redisHostConnection.GetDatabase();
...
private async Task<string> RetrieveItem(string itemKey)
{
    // Attempt to retrieve the item from the Redis cache
    string itemValue = await cache.StringGetAsync(itemKey);

    // If the value returned is null, the item was not found in the cache
    // So retrieve the item from the data source and add it to the cache
    if (itemValue == null)
    {
        itemValue = await GetItemFromDataSourceAsync(itemKey);
        await cache.StringSetAsync(itemKey, itemValue);
    }

    // Return the item
    return itemValue;
}
```

The `StringGet` and `StringSet` methods are not restricted to retrieving or storing string values. They can take any item that is serialized as an array of bytes. If you need to save a .NET object, you can serialize it as a byte stream and use the `StringSet` method to write it to the cache.

Similarly, you can read an object from the cache by using the `StringGet` method and deserializing it as a .NET object. The following code shows a set of extension methods for the `IDatabase` interface (the `GetDatabase` method of a Redis connection returns an `IDatabase` object), and some sample code that uses these methods to read and write a `BlogPost` object to the cache:

```

public static class RedisCacheExtensions
{
    public static async Task<T> GetAsync<T>(this IDatabase cache, string key)
    {
        return Deserialize<T>(await cache.StringGetAsync(key));
    }

    public static async Task<object> GetAsync(this IDatabase cache, string key)
    {
        return Deserialize<object>(await cache.StringGetAsync(key));
    }

    public static async Task SetAsync(this IDatabase cache, string key, object value)
    {
        await cache.StringSetAsync(key, Serialize(value));
    }

    static byte[] Serialize(object o)
    {
        byte[] objectDataAsStream = null;

        if (o != null)
        {
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            using (MemoryStream memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, o);
                objectDataAsStream = memoryStream.ToArray();
            }
        }

        return objectDataAsStream;
    }

    static T Deserialize<T>(byte[] stream)
    {
        T result = default(T);

        if (stream != null)
        {
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            using (MemoryStream memoryStream = new MemoryStream(stream))
            {
                result = (T)binaryFormatter.Deserialize(memoryStream);
            }
        }

        return result;
    }
}

```

The following code illustrates a method named `RetrieveBlogPost` that uses these extension methods to read and write a serializable `BlogPost` object to the cache following the cache-aside pattern:

```

// The BlogPost type
[Serializable]
public class BlogPost
{
    private HashSet<string> tags;

    public BlogPost(int id, string title, int score, IEnumerable<string> tags)
    {
        this.Id = id;
        this.Title = title;
        this.Score = score;
        this.tags = new HashSet<string>(tags);
    }

    public int Id { get; set; }
    public string Title { get; set; }
    public int Score { get; set; }
    public ICollection<string> Tags => this.tags;
}

...
private async Task<BlogPost> RetrieveBlogPost(string blogPostKey)
{
    BlogPost blogPost = await cache.GetAsync<BlogPost>(blogPostKey);
    if (blogPost == null)
    {
        blogPost = await GetBlogPostFromDataSourceAsync(blogPostKey);
        await cache.SetAsync(blogPostKey, blogPost);
    }

    return blogPost;
}

```

Redis supports command pipelining if a client application sends multiple asynchronous requests. Redis can multiplex the requests using the same connection rather than receiving and responding to commands in a strict sequence.

This approach helps to reduce latency by making more efficient use of the network. The following code snippet shows an example that retrieves the details of two customers concurrently. The code submits two requests and then performs some other processing (not shown) before waiting to receive the results. The `Wait` method of the cache object is similar to the .NET Framework `Task.Wait` method:

```

ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
var task1 = cache.StringGetAsync("customer:1");
var task2 = cache.StringGetAsync("customer:2");
...
var customer1 = cache.Wait(task1);
var customer2 = cache.Wait(task2);

```

For additional information on writing client applications that can the Azure Redis Cache, see [Azure Redis Cache documentation](#). More information is also available at [StackExchange.Redis](#).

The page [Pipelines and multiplexers](#) on the same website provides more information about asynchronous operations and pipelining with Redis and the StackExchange library. The next section in this article, Using Redis Caching, provides examples of some of the more advanced techniques that you can apply to data that's held in a Redis cache.

Using Redis caching

The simplest use of Redis for caching concerns is key-value pairs where the value is an uninterpreted string of

arbitrary length that can contain any binary data. (It is essentially an array of bytes that can be treated as a string). This scenario was illustrated in the section Implement Redis Cache client applications earlier in this article.

Note that keys also contain uninterpreted data, so you can use any binary information as the key. The longer the key is, however, the more space it will take to store, and the longer it will take to perform lookup operations. For usability and ease of maintenance, design your keyspace carefully and use meaningful (but not verbose) keys.

For example, use structured keys such as "customer:100" to represent the key for the customer with ID 100 rather than simply "100". This scheme enables you to easily distinguish between values that store different data types. For example, you could also use the key "orders:100" to represent the key for the order with ID 100.

Apart from one-dimensional binary strings, a value in a Redis key-value pair can also hold more structured information, including lists, sets (sorted and unsorted), and hashes. Redis provides a comprehensive command set that can manipulate these types, and many of these commands are available to .NET Framework applications through a client library such as StackExchange. The page [An introduction to Redis data types and abstractions](#) on the Redis website provides a more detailed overview of these types and the commands that you can use to manipulate them.

This section summarizes some common use cases for these data types and commands.

Perform atomic and batch operations

Redis supports a series of atomic get-and-set operations on string values. These operations remove the possible race hazards that might occur when using separate `GET` and `SET` commands. The operations that are available include:

- `INCR`, `INCRBY`, `DECR`, and `DECRBY`, which perform atomic increment and decrement operations on integer numeric data values. The StackExchange library provides overloaded versions of the `IDatabase.StringIncrementAsync` and `IDatabase.StringDecrementAsync` methods to perform these operations and return the resulting value that is stored in the cache. The following code snippet illustrates how to use these methods:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
await cache.StringSetAsync("data:counter", 99);
...
long oldValue = await cache.StringIncrementAsync("data:counter");
// Increment by 1 (the default)
// oldValue should be 100

long newValue = await cache.StringDecrementAsync("data:counter", 50);
// Decrement by 50
// newValue should be 50
```

- `GETSET`, which retrieves the value that's associated with a key and changes it to a new value. The StackExchange library makes this operation available through the `IDatabase.StringGetSetAsync` method. The code snippet below shows an example of this method. This code returns the current value that's associated with the key "data:counter" from the previous example. Then it resets the value for this key back to zero, all as part of the same operation:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string oldValue = await cache.StringGetSetAsync("data:counter", 0);
```

- `MGET` and `MSET`, which can return or change a set of string values as a single operation. The `IDatabase.StringGetAsync` and `IDatabase.StringSetAsync` methods are overloaded to support this

functionality, as shown in the following example:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Create a list of key-value pairs
var keysAndValues =
    new List<KeyValuePair<RedisKey, RedisValue>>()
{
    new KeyValuePair<RedisKey, RedisValue>("data:key1", "value1"),
    new KeyValuePair<RedisKey, RedisValue>("data:key99", "value2"),
    new KeyValuePair<RedisKey, RedisValue>("data:key322", "value3")
};

// Store the list of key-value pairs in the cache
cache.StringSet(keysAndValues.ToArray());
...
// Find all values that match a list of keys
RedisKey[] keys = { "data:key1", "data:key99", "data:key322" };
// values should contain { "value1", "value2", "value3" }
RedisValue[] values = cache.StringGet(keys);
```

You can also combine multiple operations into a single Redis transaction as described in the Redis transactions and batches section earlier in this article. The StackExchange library provides support for transactions through the `ITransaction` interface.

You create an `ITransaction` object by using the `IDatabase.CreateTransaction` method. You invoke commands to the transaction by using the methods provided by the `ITransaction` object.

The `ITransaction` interface provides access to a set of methods that's similar to those accessed by the `IDatabase` interface, except that all the methods are asynchronous. This means that they are only performed when the `ITransaction.Execute` method is invoked. The value that's returned by the `ITransaction.Execute` method indicates whether the transaction was created successfully (true) or if it failed (false).

The following code snippet shows an example that increments and decrements two counters as part of the same transaction:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
ITransaction transaction = cache.CreateTransaction();
var tx1 = transaction.StringIncrementAsync("data:counter1");
var tx2 = transaction.StringDecrementAsync("data:counter2");
bool result = transaction.Execute();
Console.WriteLine("Transaction {0}", result ? "succeeded" : "failed");
Console.WriteLine("Result of increment: {0}", tx1.Result);
Console.WriteLine("Result of decrement: {0}", tx2.Result);
```

Remember that Redis transactions are unlike transactions in relational databases. The `Execute` method simply queues all the commands that comprise the transaction to be run, and if any of them is malformed then the transaction is stopped. If all the commands have been queued successfully, each command runs asynchronously.

If any command fails, the others still continue processing. If you need to verify that a command has completed successfully, you must fetch the results of the command by using the `Result` property of the corresponding task, as shown in the example above. Reading the `Result` property will block the calling thread until the task has completed.

For more information, see the [Transactions in Redis](#) page on the StackExchange.Redis website.

When performing batch operations, you can use the `IBatch` interface of the StackExchange library. This interface

provides access to a set of methods similar to those accessed by the `IDatabase` interface, except that all the methods are asynchronous.

You create an `IBatch` object by using the `IDatabase.CreateBatch` method, and then run the batch by using the `IBatch.Execute` method, as shown in the following example. This code simply sets a string value, increments and decrements the same counters used in the previous example, and displays the results:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
IBatch batch = cache.CreateBatch();
batch.StringSetAsync("data:key1", 11);
var t1 = batch.StringIncrementAsync("data:counter1");
var t2 = batch.StringDecrementAsync("data:counter2");
batch.Execute();
Console.WriteLine("{0}", t1.Result);
Console.WriteLine("{0}", t2.Result);
```

It is important to understand that unlike a transaction, if a command in a batch fails because it is malformed, the other commands might still run. The `IBatch.Execute` method does not return any indication of success or failure.

Perform fire and forget cache operations

Redis supports fire and forget operations by using command flags. In this situation, the client simply initiates an operation but has no interest in the result and does not wait for the command to be completed. The example below shows how to perform the INCR command as a fire and forget operation:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
await cache.StringSetAsync("data:key1", 99);
...
cache.StringIncrement("data:key1", flags: CommandFlags.FireAndForget);
```

Specify automatically expiring keys

When you store an item in a Redis cache, you can specify a timeout after which the item will be automatically removed from the cache. You can also query how much more time a key has before it expires by using the `TTL` command. This command is available to StackExchange applications by using the `IDatabase.KeyTimeToLive` method.

The following code snippet shows how to set an expiration time of 20 seconds on a key, and query the remaining lifetime of the key:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Add a key with an expiration time of 20 seconds
await cache.StringSetAsync("data:key1", 99, TimeSpan.FromSeconds(20));
...
// Query how much time a key has left to live
// If the key has already expired, the KeyTimeToLive function returns a null
TimeSpan? expiry = cache.KeyTimeToLive("data:key1");
```

You can also set the expiration time to a specific date and time by using the EXPIRE command, which is available in the StackExchange library as the `KeyExpireAsync` method:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Add a key with an expiration date of midnight on 1st January 2015
await cache.StringSetAsync("data:key1", 99);
await cache.KeyExpireAsync("data:key1",
    new DateTime(2015, 1, 1, 0, 0, DateTimeKind.Utc));
...
```

TIP

You can manually remove an item from the cache by using the DEL command, which is available through the StackExchange library as the `IDatabase.KeyDeleteAsync` method.

Use tags to cross-correlate cached items

A Redis set is a collection of multiple items that share a single key. You can create a set by using the SADD command. You can retrieve the items in a set by using the SMEMBERS command. The StackExchange library implements the SADD command with the `IDatabase.SetAddAsync` method, and the SMEMBERS command with the `IDatabase.SetMembersAsync` method.

You can also combine existing sets to create new sets by using the SDIFF (set difference), SINTER (set intersection), and SUNION (set union) commands. The StackExchange library unifies these operations in the `IDatabase.SetCombineAsync` method. The first parameter to this method specifies the set operation to perform.

The following code snippets show how sets can be useful for quickly storing and retrieving collections of related items. This code uses the `BlogPost` type that was described in the section Implement Redis Cache Client Applications earlier in this article.

A `BlogPost` object contains four fields—an ID, a title, a ranking score, and a collection of tags. The first code snippet below shows the sample data that's used for populating a C# list of `BlogPost` objects:

```

List<string[]> tags = new List<string[]>
{
    new[] { "iot", "csharp" },
    new[] { "iot", "azure", "csharp" },
    new[] { "csharp", "git", "big data" },
    new[] { "iot", "git", "database" },
    new[] { "database", "git" },
    new[] { "csharp", "database" },
    new[] { "iot" },
    new[] { "iot", "database", "git" },
    new[] { "azure", "database", "big data", "git", "csharp" },
    new[] { "azure" }
};

List<BlogPost> posts = new List<BlogPost>();
int blogKey = 1;
int numberOfPosts = 20;
Random random = new Random();
for (int i = 0; i < numberOfPosts; i++)
{
    blogKey++;
    posts.Add(new BlogPost(
        blogKey, // Blog post ID
        string.Format(CultureInfo.InvariantCulture, "Blog Post #{0}",
            blogKey), // Blog post title
        random.Next(100, 10000), // Ranking score
        tags[i % tags.Count])); // Tags--assigned from a collection
        // in the tags list
}

```

You can store the tags for each `BlogPost` object as a set in a Redis cache and associate each set with the ID of the `BlogPost`. This enables an application to quickly find all the tags that belong to a specific blog post. To enable searching in the opposite direction and find all blog posts that share a specific tag, you can create another set that holds the blog posts referencing the tag ID in the key:

```

ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Tags are easily represented as Redis Sets
foreach (BlogPost post in posts)
{
    string redisKey = string.Format(CultureInfo.InvariantCulture,
        "blog:posts:{0}:tags", post.Id);
    // Add tags to the blog post in Redis
    await cache.SetAddAsync(
        redisKey, post.Tags.Select(s => (RedisValue)s.ToArray()));

    // Now do the inverse so we can figure how which blog posts have a given tag
    foreach (var tag in post.Tags)
    {
        await cache.SetAddAsync(string.Format(CultureInfo.InvariantCulture,
            "tag:{0}:blog:posts", tag), post.Id);
    }
}

```

These structures enable you to perform many common queries very efficiently. For example, you can find and display all of the tags for blog post 1 like this:

```
// Show the tags for blog post #1
foreach (var value in await cache.SetMembersAsync("blog:posts:1:tags"))
{
    Console.WriteLine(value);
}
```

You can find all tags that are common to blog post 1 and blog post 2 by performing a set intersection operation, as follows:

```
// Show the tags in common for blog posts #1 and #2
foreach (var value in await cache.SetCombineAsync(SetOperation.Intersect, new RedisKey[]
    { "blog:posts:1:tags", "blog:posts:2:tags" }))
{
    Console.WriteLine(value);
}
```

And you can find all blog posts that contain a specific tag:

```
// Show the ids of the blog posts that have the tag "iot".
foreach (var value in await cache.SetMembersAsync("tag:iot:blog:posts"))
{
    Console.WriteLine(value);
}
```

Find recently accessed items

A common task required of many applications is to find the most recently accessed items. For example, a blogging site might want to display information about the most recently read blog posts.

You can implement this functionality by using a Redis list. A Redis list contains multiple items that share the same key. The list acts as a double-ended queue. You can push items to either end of the list by using the LPUSH (left push) and RPUSH (right push) commands. You can retrieve items from either end of the list by using the LPOP and RPOP commands. You can also return a set of elements by using the LRANGE and RRANGE commands.

The code snippets below show how you can perform these operations by using the StackExchange library. This code uses the `BlogPost` type from the previous examples. As a blog post is read by a user, the `IDatabase.ListLeftPushAsync` method pushes the title of the blog post onto a list that's associated with the key "blog:recent_posts" in the Redis cache.

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string redisKey = "blog:recent_posts";
BlogPost blogPost = ...; // Reference to the blog post that has just been read
await cache.ListLeftPushAsync(
    redisKey, blogPost.Title); // Push the blog post onto the list
```

As more blog posts are read, their titles are pushed onto the same list. The list is ordered by the sequence in which the titles have been added. The most recently read blog posts are towards the left end of the list. (If the same blog post is read more than once, it will have multiple entries in the list.)

You can display the titles of the most recently read posts by using the `IDatabase.ListRange` method. This method takes the key that contains the list, a starting point, and an ending point. The following code retrieves the titles of the 10 blog posts (items from 0 to 9) at the left-most end of the list:

```
// Show latest ten posts
foreach (string postTitle in await cache.ListRangeAsync(redisKey, 0, 9))
{
    Console.WriteLine(postTitle);
}
```

Note that the `ListRangeAsync` method does not remove items from the list. To do this, you can use the `IDatabase.ListLeftPopAsync` and `IDatabase.ListRightPopAsync` methods.

To prevent the list from growing indefinitely, you can periodically cull items by trimming the list. The code snippet below shows you how to remove all but the five left-most items from the list:

```
await cache.ListTrimAsync(redisKey, 0, 5);
```

Implement a leader board

By default, the items in a set are not held in any specific order. You can create an ordered set by using the ZADD command (the `IDatabase.SortedSetAdd` method in the StackExchange library). The items are ordered by using a numeric value called a score, which is provided as a parameter to the command.

The following code snippet adds the title of a blog post to an ordered list. In this example, each blog post also has a score field that contains the ranking of the blog post.

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string redisKey = "blog:post_rankings";
BlogPost blogPost = ...; // Reference to a blog post that has just been rated
await cache.SortedSetAddAsync(redisKey, blogPost.Title, blogPost.Score);
```

You can retrieve the blog post titles and scores in ascending score order by using the `IDatabase.SortedSetRangeByRankWithScores` method:

```
foreach (var post in await cache.SortedSetRangeByRankWithScoresAsync(redisKey))
{
    Console.WriteLine(post);
}
```

NOTE

The StackExchange library also provides the `IDatabase.SortedSetRangeByRankAsync` method, which returns the data in score order, but does not return the scores.

You can also retrieve items in descending order of scores, and limit the number of items that are returned by providing additional parameters to the `IDatabase.SortedSetRangeByRankWithScoresAsync` method. The next example displays the titles and scores of the top 10 ranked blog posts:

```
foreach (var post in await cache.SortedSetRangeByRankWithScoresAsync(
            redisKey, 0, 9, Order.Descending))
{
    Console.WriteLine(post);
}
```

The next example uses the `IDatabase.SortedSetRangeByScoreWithScoresAsync` method, which you can use to limit the

items that are returned to those that fall within a given score range:

```
// Blog posts with scores between 5000 and 100000
foreach (var post in await cache.SortedSetRangeByScoreWithScoresAsync(
    redisKey, 5000, 100000))
{
    Console.WriteLine(post);
}
```

Message by using channels

Apart from acting as a data cache, a Redis server provides messaging through a high-performance publisher/subscriber mechanism. Client applications can subscribe to a channel, and other applications or services can publish messages to the channel. Subscribing applications will then receive these messages and can process them.

Redis provides the SUBSCRIBE command for client applications to use to subscribe to channels. This command expects the name of one or more channels on which the application will accept messages. The StackExchange library includes the `ISubscription` interface, which enables a .NET Framework application to subscribe and publish to channels.

You create an `ISubscription` object by using the `GetSubscriber` method of the connection to the Redis server. Then you listen for messages on a channel by using the `SubscribeAsync` method of this object. The following code example shows how to subscribe to a channel named "messages:blogPosts":

```
ConnectionMultiplexer redisHostConnection = ...;
ISubscriber subscriber = redisHostConnection.GetSubscriber();
...
await subscriber.SubscribeAsync("messages:blogPosts", (channel, message) => Console.WriteLine("Title is: {0}",
    message));
```

The first parameter to the `Subscribe` method is the name of the channel. This name follows the same conventions that are used by keys in the cache. The name can contain any binary data, although it is advisable to use relatively short, meaningful strings to help ensure good performance and maintainability.

Note also that the namespace used by channels is separate from that used by keys. This means you can have channels and keys that have the same name, although this may make your application code more difficult to maintain.

The second parameter is an Action delegate. This delegate runs asynchronously whenever a new message appears on the channel. This example simply displays the message on the console (the message will contain the title of a blog post).

To publish to a channel, an application can use the Redis PUBLISH command. The StackExchange library provides the `IServer.PublishAsync` method to perform this operation. The next code snippet shows how to publish a message to the "messages:blogPosts" channel:

```
ConnectionMultiplexer redisHostConnection = ...;
ISubscriber subscriber = redisHostConnection.GetSubscriber();
...
BlogPost blogPost = ...;
subscriber.PublishAsync("messages:blogPosts", blogPost.Title);
```

There are several points you should understand about the publish/subscribe mechanism:

- Multiple subscribers can subscribe to the same channel, and they will all receive the messages that are published to that channel.

- Subscribers only receive messages that have been published after they have subscribed. Channels are not buffered, and once a message is published, the Redis infrastructure pushes the message to each subscriber and then removes it.
- By default, messages are received by subscribers in the order in which they are sent. In a highly active system with a large number of messages and many subscribers and publishers, guaranteed sequential delivery of messages can slow performance of the system. If each message is independent and the order is unimportant, you can enable concurrent processing by the Redis system, which can help to improve responsiveness. You can achieve this in a StackExchange client by setting the `PreserveAsyncOrder` of the connection used by the subscriber to false:

```
ConnectionMultiplexer redisHostConnection = ...;
redisHostConnection.PreserveAsyncOrder = false;
ISubscriber subscriber = redisHostConnection.GetSubscriber();
```

Serialization considerations

When you choose a serialization format, consider tradeoffs between performance, interoperability, versioning, compatibility with existing systems, data compression, and memory overhead. When you are evaluating performance, remember that benchmarks are highly dependent on context. They may not reflect your actual workload, and may not consider newer libraries or versions. There is no single "fastest" serializer for all scenarios.

Some options to consider include:

- [Protocol Buffers](#) (also called protobuf) is a serialization format developed by Google for serializing structured data efficiently. It uses strongly-typed definition files to define message structures. These definition files are then compiled to language-specific code for serializing and deserializing messages. Protobuf can be used over existing RPC mechanisms, or it can generate an RPC service.
- [Apache Thrift](#) uses a similar approach, with strongly typed definition files and a compilation step to generate the serialization code and RPC services.
- [Apache Avro](#) provides similar functionality to Protocol Buffers and Thrift, but there is no compilation step. Instead, serialized data always includes a schema that describes the structure.
- [JSON](#) is an open standard that uses human-readable text fields. It has broad cross-platform support. JSON does not use message schemas. Being a text-based format, it is not very efficient over the wire. In some cases, however, you may be returning cached items directly to a client via HTTP, in which case storing JSON could save the cost of deserializing from another format and then serializing to JSON.
- [BSON](#) is a binary serialization format that uses a structure similar to JSON. BSON was designed to be lightweight, easy to scan, and fast to serialize and deserialize, relative to JSON. Payloads are comparable in size to JSON. Depending on the data, a BSON payload may be smaller or larger than a JSON payload. BSON has some additional data types that are not available in JSON, notably BinData (for byte arrays) and Date.
- [MessagePack](#) is a binary serialization format that is designed to be compact for transmission over the wire. There are no message schemas or message type checking.
- [Bond](#) is a cross-platform framework for working with schematized data. It supports cross-language serialization and deserialization. Notable differences from other systems listed here are support for inheritance, type aliases, and generics.
- [gRPC](#) is an open source RPC system developed by Google. By default, it uses Protocol Buffers as its definition language and underlying message interchange format.

Related patterns and guidance

The following pattern might also be relevant to your scenario when you implement caching in your applications:

- [Cache-aside pattern](#): This pattern describes how to load data on demand into a cache from a data store. This pattern also helps to maintain consistency between data that's held in the cache and the data in the original data store.
- The [Sharding pattern](#) provides information about implementing horizontal partitioning to help improve scalability when storing and accessing large volumes of data.

More information

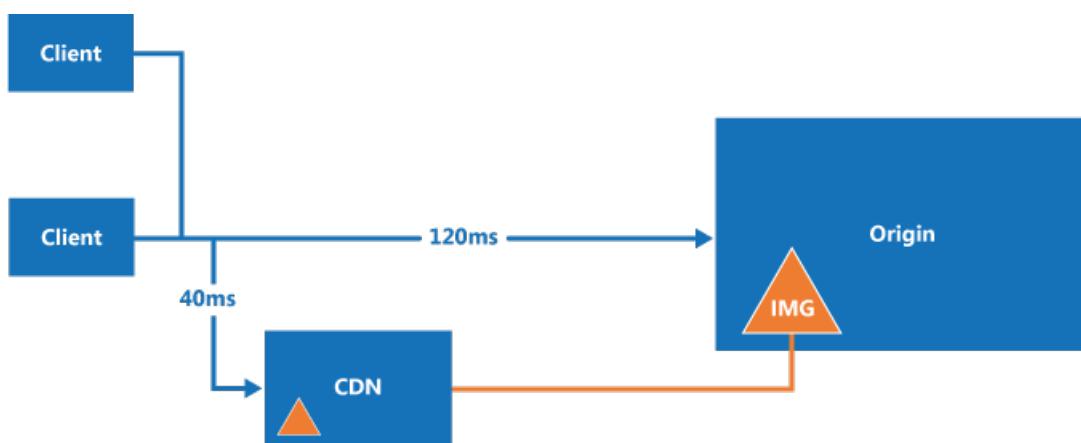
- The [MemoryCache class](#) page on the Microsoft website
- The [Azure Redis Cache documentation](#) page on the Microsoft website
- The [Azure Redis Cache FAQ](#) page on the Microsoft website
- The [Configuration model](#) page on the Microsoft website
- The [Task-based Asynchronous Pattern](#) page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

Best practices for using content delivery networks (CDNs)

2/27/2018 • 8 minutes to read • [Edit Online](#)

A content delivery network (CDN) is a distributed network of servers that can efficiently deliver web content to users. CDNs store cached content on edge servers that are close to end-users, to minimize latency.

CDNs are typically used to deliver static content such as images, style sheets, documents, client-side scripts, and HTML pages. The major advantages of using a CDN are lower latency and faster delivery of content to users, regardless of their geographical location in relation to the datacenter where the application is hosted. CDNs can also help to reduce load on a web application, because the application does not have to service requests for the content that is hosted in the CDN.



In Azure, the [Azure Content Delivery Network](#) is a global CDN solution for delivering high-bandwidth content that is hosted in Azure or any other location. Using Azure CDN, you can cache publicly available objects loaded from Azure blob storage, a web application, virtual machine, any publicly accessible web server.

This topic describes some general best practices and considerations when using a CDN. To learn more about using Azure CDN, see [CDN Documentation](#).

How and why a CDN is used

Typical uses for a CDN include:

- Delivering static resources for client applications, often from a website. These resources can be images, style sheets, documents, files, client-side scripts, HTML pages, HTML fragments, or any other content that the server does not need to modify for each request. The application can create items at runtime and make them available to the CDN (for example, by creating a list of current news headlines), but it does not do so for each request.
- Delivering public static and shared content to devices such as mobile phones and tablet computers. The application itself is a web service that offers an API to clients running on the various devices. The CDN can also deliver static datasets (via the web service) for the clients to use, perhaps to generate the client UI. For example, the CDN could be used to distribute JSON or XML documents.
- Serving entire websites that consist of only public static content to clients, without requiring any dedicated compute resources.
- Streaming video files to the client on demand. Video benefits from the low latency and reliable connectivity available from the globally located datacenters that offer CDN connections. Microsoft Azure Media Services (AMS) integrates with Azure CDN to deliver content directly to the CDN for further distribution. For more information, see [Streaming endpoints overview](#).

- Generally improving the experience for users, especially those located far from the datacenter hosting the application. These users might otherwise suffer higher latency. A large proportion of the total size of the content in a web application is often static, and using the CDN can help to maintain performance and overall user experience while eliminating the requirement to deploy the application to multiple data centers. For a list of Azure CDN node locations, see [Azure CDN POP Locations](#).
- Supporting IoT (Internet of Things) solutions. The huge numbers of devices and appliances involved in an IoT solution could easily overwhelm an application if it had to distribute firmware updates directly to each device.
- Coping with peaks and surges in demand without requiring the application to scale, avoiding the consequent increased running costs. For example, when an update to an operating system is released for a hardware device such as a specific model of router, or for a consumer device such as a smart TV, there will be a huge peak in demand as it is downloaded by millions of users and devices over a short period.

Challenges

There are several challenges to take into account when planning to use a CDN.

- **Deployment.** Decide the origin from which the CDN fetches the content, and whether you need to deploy the content in more than one storage system. Take into account the process for deploying static content and resources. For example, you may need to implement a separate step to load content into Azure blob storage.
- **Versioning and cache-control.** Consider how you will update static content and deploy new versions. Understand how the CDN performs caching and time-to-live (TTL). For Azure CDN, see [How caching works](#).
- **Testing.** It can be difficult to perform local testing of your CDN settings when developing and testing an application locally or in a staging environment.
- **Search engine optimization (SEO).** Content such as images and documents are served from a different domain when you use the CDN. This can have an effect on SEO for this content.
- **Content security.** Not all CDNs offer any form of access control for the content. Some CDN services, including Azure CDN, support token-based authentication to protect CDN content. For more information, see [Securing Azure Content Delivery Network assets with token authentication](#).
- **Client security.** Clients might connect from an environment that does not allow access to resources on the CDN. This could be a security-constrained environment that limits access to only a set of known sources, or one that prevents loading of resources from anything other than the page origin. A fallback implementation is required to handle these cases.
- **Resilience.** The CDN is a potential single point of failure for an application.

Scenarios where CDN may be less useful include:

- If the content has a low hit rate, it might be accessed only few times while it is valid (determined by its time-to-live setting).
- If the data is private, such as for large enterprises or supply chain ecosystems.

General guidelines and good practices

Using a CDN is a good way to minimize the load on your application, and maximize availability and performance. Consider adopting this strategy for all of the appropriate content and resources your application uses. Consider the points in the following sections when designing your strategy to use a CDN.

Deployment

Static content may need to be provisioned and deployed independently from the application if you do not include it in the application deployment package or process. Consider how this will affect the versioning approach you use to manage both the application components and the static resource content.

Consider using bundling and minification techniques to reduce load times for clients. Bundling combines multiple files into a single file. Minification removes unnecessary characters from scripts and CSS files without altering

functionality.

If you need to deploy the content to an additional location, this will be an extra step in the deployment process. If the application updates the content for the CDN, perhaps at regular intervals or in response to an event, it must store the updated content in any additional locations as well as the endpoint for the CDN.

Consider how you will handle local development and testing when some static content is expected to be served from a CDN. For example, you could pre-deploy the content to the CDN as part of your build script. Alternatively, use compile directives or flags to control how the application loads the resources. For example, in debug mode, the application could load static resources from a local folder. In release mode, the application would use the CDN.

Consider the options for file compression, such as gzip (GNU zip). Compression may be performed on the origin server by the web application hosting or directly on the edge servers by the CDN. For more information, see [Improve performance by compressing files in Azure CDN](#).

Routing and versioning

You may need to use different CDN instances at various times. For example, when you deploy a new version of the application you may want to use a new CDN and retain the old CDN (holding content in an older format) for previous versions. If you use Azure blob storage as the content origin, you can create a separate storage account or a separate container and point the CDN endpoint to it.

Do not use the query string to denote different versions of the application in links to resources on the CDN because, when retrieving content from Azure blob storage, the query string is part of the resource name (the blob name). This approach can also affect how the client caches resources.

Deploying new versions of static content when you update an application can be a challenge if the previous resources are cached on the CDN. For more information, see the section on cache control, below.

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

Cache control

Consider how to manage caching within the system. For example, in Azure CDN, you can set global caching rules, and then set custom caching for particular origin endpoints. You can also control how caching is performed in a CDN by sending cache-directive headers at the origin.

For more information, see [How caching works](#).

To prevent objects from being available on the CDN, you can delete them from the origin, remove or delete the CDN endpoint, or in the case of blob storage, make the container or blob private. However, items are not removed from the until the time-to-live expires. You can also manually purge a CDN endpoint.

Security

The CDN can deliver content over HTTPS (SSL), by using the certificate provided by the CDN, as well as over standard HTTP. To avoid browser warnings about mixed content, you might need to use HTTPS to request static content that is displayed in pages loaded through HTTPS.

If you deliver static assets such as font files by using the CDN, you might encounter same-origin policy issues if you use an `XMLHttpRequest` call to request these resources from a different domain. Many web browsers prevent cross-origin resource sharing (CORS) unless the web server is configured to set the appropriate response headers. You can configure the CDN to support CORS by using one of the following methods:

- Configure the CDN to add CORS headers to the responses. For more information, see [Using Azure CDN with CORS](#).
- If the origin is Azure blob storage, add CORS rules to the storage endpoint. For more information, see [Cross-Origin Resource Sharing \(CORS\) Support for the Azure Storage Services](#).

- Configure the application to set the CORS headers. For example, see [Enabling Cross-Origin Requests \(CORS\)](#) in the ASP.NET Core documentation.

CDN fallback

Consider how your application will cope with a failure or temporary unavailability of the CDN. Client applications may be able to use copies of the resources that were cached locally (on the client) during previous requests, or you can include code that detects failure and instead requests resources from the origin (the application folder or Azure blob container that holds the resources) if the CDN is unavailable.

Data partitioning

2/7/2018 • 65 minutes to read • [Edit Online](#)

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

Why partition data?

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

In this article, the term *partitioning* means the process of physically dividing data into separate data stores. It is not the same as SQL Server table partitioning.

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

- **Improve scalability.** When you scale up a single database system, it will eventually reach a physical hardware limit. If you divide data across multiple partitions, each of which is hosted on a separate server, you can scale out the system almost indefinitely.
- **Improve performance.** Data access operations on each partition take place over a smaller volume of data. Provided that the data is partitioned in a suitable way, partitioning can make your system more efficient. Operations that affect more than one partition can run in parallel. Each partition can be located near the application that uses it to minimize network latency.
- **Improve availability.** Separating data across multiple servers avoids a single point of failure. If a server fails, or is undergoing planned maintenance, only the data in that partition is unavailable. Operations on other partitions can continue. Increasing the number of partitions reduces the relative impact of a single server failure by reducing the percentage of data that will be unavailable. Replicating each partition can further reduce the chance of a single partition failure affecting operations. It also makes it possible to separate critical data that must be continually and highly available from low-value data that has lower availability requirements (log files, for example).
- **Improve security.** Depending on the nature of the data and how it is partitioned, it might be possible to separate sensitive and non-sensitive data into different partitions, and therefore into different servers or data stores. Security can then be specifically optimized for the sensitive data.
- **Provide operational flexibility.** Partitioning offers many opportunities for fine tuning operations, maximizing administrative efficiency, and minimizing cost. For example, you can define different strategies for management, monitoring, backup and restore, and other administrative tasks based on the importance of the data in each partition.
- **Match the data store to the pattern of use.** Partitioning allows each partition to be deployed on a different type of data store, based on cost and the built-in features that data store offers. For example, large binary data can be stored in a blob data store, while more structured data can be held in a document database. For more information, see [Building a polyglot solution](#) in the patterns & practices guide and [Data access for highly-scalable solutions: Using SQL, NoSQL, and polyglot persistence](#) on the Microsoft website.

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

reasons for this rationale include:

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

Designing partitions

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

NOTE

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

Partitioning strategies

The three typical strategies for partitioning data are:

- **Horizontal partitioning** (often called *sharding*). In this strategy, each partition is a data store in its own right, but all partitions have the same schema. Each partition is known as a *shard* and holds a specific subset of the data, such as all the orders for a specific set of customers in an e-commerce application.
- **Vertical partitioning**. In this strategy, each partition holds a subset of the fields for items in the data store. The fields are divided according to their pattern of use. For example, frequently accessed fields might be placed in one vertical partition and less frequently accessed fields in another.
- **Functional partitioning**. In this strategy, data is aggregated according to how it is used by each bounded context in the system. For example, an e-commerce system that implements separate business functions for invoicing and managing product inventory might store invoice data in one partition and product inventory data in another.

It's important to note that the three strategies described here can be combined. They are not mutually exclusive, and we recommend that you consider them all when you design a partitioning scheme. For example, you might divide data into shards and then use vertical partitioning to further subdivide the data in each shard. Similarly, the data in a functional partition can be split into shards (which can also be vertically partitioned).

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

Horizontal partitioning (sharding)

Figure 1 shows an overview of horizontal partitioning or sharding. In this example, product inventory data is divided into shards based on the product key. Each shard holds the data for a contiguous range of shard keys (A-G and H-Z), organized alphabetically.

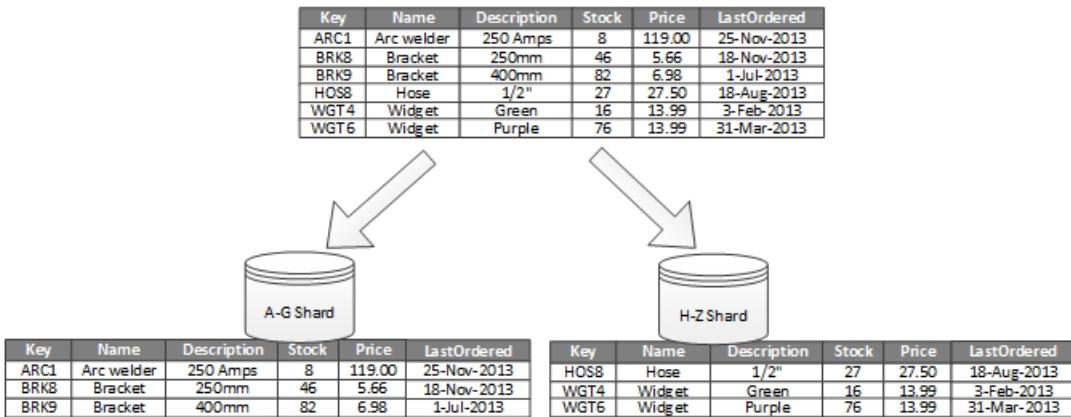


Figure 1. Horizontally partitioning (sharding) data based on a partition key

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

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

Note that different shards do not have to contain similar volumes of data. Rather, the more important consideration is to balance the number of requests. Some shards might be very large, but each item is the subject of a low number of access operations. Other shards might be smaller, but each item is accessed much more frequently. It is also important to ensure that a single shard does not exceed the scale limits (in terms of capacity and processing resources) of the data store that's being used to host that shard.

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

Choose a sharding key that minimizes any future requirements to split large shards into smaller pieces, coalesce small shards into larger partitions, or change the schema that describes the data stored in a set of partitions. These operations can be very time consuming, and might require taking one or more shards offline while they are performed.

If shards are replicated, it might be possible to keep some of the replicas online while others are split, merged, or reconfigured. However, the system might need to limit the operations that can be performed on the data in these shards while the reconfiguration is taking place. For example, the data in the replicas can be marked as read-only to limit the scope of inconsistencies that might occur while shards are being restructured.

For more detailed information and guidance about many of these considerations, and good practice techniques for designing data stores that implement horizontal partitioning, see [Sharding pattern](#).

Vertical partitioning

The most common use for vertical partitioning is to reduce the I/O and performance costs associated with fetching the items that are accessed most frequently. Figure 2 shows an example of vertical partitioning. In this example, different properties for each data item are held in different partitions. One partition holds data that is accessed more frequently, including the name, description, and price information for products. Another holds the volume in stock and the last ordered date.

The diagram shows a top-level wide table being partitioned into two narrower tables below it. The top table has columns: Key, Name, Description, Stock, Price, and LastOrdered. The bottom-left table has columns: Key, Name, Description, and Price. The bottom-right table has columns: Key, Stock, and LastOrdered.

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

Key	Name	Description	Price
ARC1	Arc welder	250Amps	119.00
BRK8	Bracket	250mm	5.66
BRK9	Bracket	400mm	6.98
HOS8	Hose	1/2"	27.50
WGT4	Widget	Green	13.99
WGT6	Widget	Purple	13.99

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

Figure 2. Vertically partitioning data by its pattern of use

In this example, the application regularly queries the product name, description, and price when displaying the product details to customers. The stock level and date when the product was last ordered from the manufacturer are held in a separate partition because these two items are commonly used together.

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

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

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

Vertical partitioning operates at the entity level within a data store, partially normalizing an entity to break it down from a *wide* item to a set of *narrow* items. It is ideally suited for column-oriented data stores such as HBase and Cassandra. If the data in a collection of columns is unlikely to change, you can also consider using column stores in SQL Server.

Functional partitioning

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

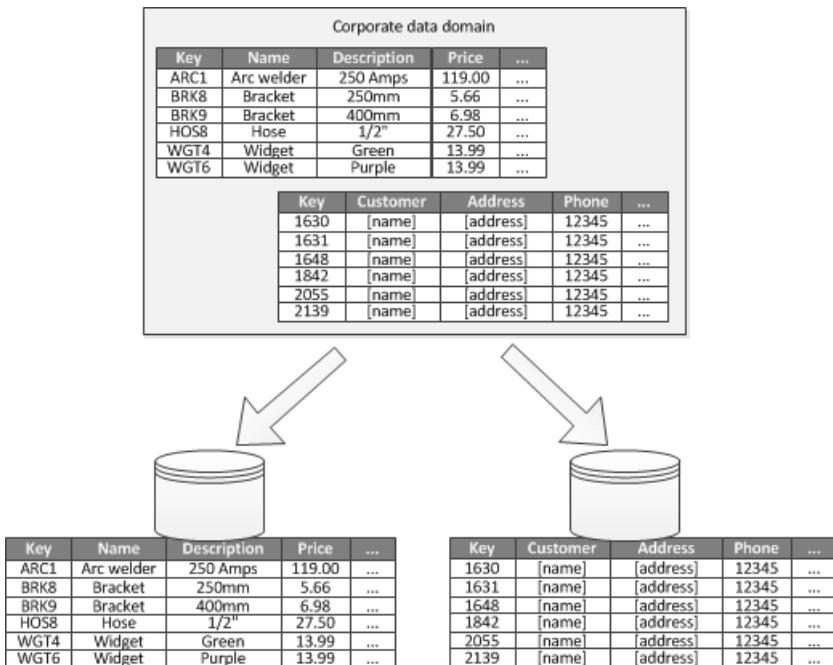


Figure 3. Functionally partitioning data by bounded context or subdomain

This partitioning strategy can help reduce data access contention across different parts of a system.

Designing partitions for scalability

It's vital to consider size and workload for each partition and balance them so that data is distributed to achieve maximum scalability. However, you must also partition the data so that it does not exceed the scaling limits of a single partition store.

Follow these steps when designing partitions for scalability:

1. Analyze the application to understand the data access patterns, such as the size of the result set returned by each query, the frequency of access, the inherent latency, and the server-side compute processing requirements. In many cases, a few major entities will demand most of the processing resources.
2. Use this analysis to determine the current and future scalability targets, such as data size and workload. Then distribute the data across the partitions to meet the scalability target. In the horizontal partitioning strategy, choosing the appropriate shard key is important to make sure distribution is even. For more information, see the [Sharding pattern](#).
3. Make sure that the resources available to each partition are sufficient to handle the scalability requirements in terms of data size and throughput. For example, the node that's hosting a partition might impose a hard limit on the amount of storage space, processing power, or network bandwidth that it provides. If the data storage and processing requirements are likely to exceed these limits, it might be necessary to refine your partitioning strategy or split data out further. For example, one scalability approach might be to separate logging data from the core application features. You do this by using separate data stores to prevent the total data storage requirements from exceeding the scaling limit of the node. If the total number of data stores exceeds the node limit, it might be necessary to use separate storage nodes.
4. Monitor the system under use to verify that the data is distributed as expected and that the partitions can handle the load that is imposed on them. It's possible that the usage does not match the usage that's anticipated by the analysis. In that case, it might be possible to rebalance the partitions. Failing that, it might be necessary to redesign some parts of the system to gain the required balance.

Note that some cloud environments allocate resources in terms of infrastructure boundaries. Ensure that the limits of your selected boundary provide enough room for any anticipated growth in the volume of data, in terms of data storage, processing power, and bandwidth.

For example, if you use Azure table storage, a busy shard might require more resources than are available to a

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

If this is the case, the shard might need to be repartitioned to spread the load. If the total size or throughput of these tables exceeds the capacity of a storage account, it might be necessary to create additional storage accounts and spread the tables across these accounts. If the number of storage accounts exceeds the number of accounts that are available to a subscription, then it might be necessary to use multiple subscriptions.

Designing partitions for query performance

Query performance can often be boosted by using smaller data sets and by running parallel queries. Each partition should contain a small proportion of the entire data set. This reduction in volume can improve the performance of queries. However, partitioning is not an alternative for designing and configuring a database appropriately. For example, make sure that you have the necessary indexes in place if you are using a relational database.

Follow these steps when designing partitions for query performance:

1. Examine the application requirements and performance:
 - Use the business requirements to determine the critical queries that must always perform quickly.
 - Monitor the system to identify any queries that perform slowly.
 - Establish which queries are performed most frequently. A single instance of each query might have minimal cost, but the cumulative consumption of resources could be significant. It might be beneficial to separate the data that's retrieved by these queries into a distinct partition, or even a cache.
2. Partition the data that is causing slow performance:
 - Limit the size of each partition so that the query response time is within target.
 - Design the shard key so that the application can easily find the partition if you are implementing horizontal partitioning. This prevents the query from having to scan through every partition.
 - Consider the location of a partition. If possible, try to keep data in partitions that are geographically close to the applications and users that access it.
3. If an entity has throughput and query performance requirements, use functional partitioning based on that entity. If this still doesn't satisfy the requirements, apply horizontal partitioning as well. In most cases a single partitioning strategy will suffice, but in some cases it is more efficient to combine both strategies.
4. Consider using asynchronous queries that run in parallel across partitions to improve performance.

Designing partitions for availability

Partitioning data can improve the availability of applications by ensuring that the entire dataset does not constitute a single point of failure and that individual subsets of the dataset can be managed independently. Replicating partitions that contain critical data can also improve availability.

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

- **How critical the data is to business operations.** Some data might include critical business information such as invoice details or bank transactions. Other data might include less critical operational data, such as log files, performance traces, and so on. After identifying each type of data, consider:
 - Storing critical data in highly-available partitions with an appropriate backup plan.
 - Establishing separate management and monitoring mechanisms or procedures for the different criticalities of each dataset. Place data that has the same level of criticality in the same partition so that it can be backed up together at an appropriate frequency. For example, partitions that hold data for bank transactions might need to be backed up more frequently than partitions that hold logging or trace information.

- **How individual partitions can be managed.** Designing partitions to support independent management and maintenance provides several advantages. For example:
 - If a partition fails, it can be recovered independently without affecting instances of applications that access data in other partitions.
 - Partitioning data by geographical area allows scheduled maintenance tasks to occur at off-peak hours for each location. Ensure that partitions are not too big to prevent any planned maintenance from being completed during this period.
- **Whether to replicate critical data across partitions.** This strategy can improve availability and performance, although it can also introduce consistency issues. It takes time for changes made to data in a partition to be synchronized with every replica. During this period, different partitions will contain different data values.

Understanding how partitioning affects design and development

Using partitioning adds complexity to the design and development of your system. Consider partitioning as a fundamental part of system design even if the system initially only contains a single partition. If you address partitioning as an afterthought, when the system starts to suffer performance and scalability issues, the complexity increases because you already have a live system to maintain.

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

In some cases, partitioning is not considered important because the initial dataset is small and can be easily handled by a single server. This might be true in a system that is not expected to scale beyond its initial size, but many commercial systems need to expand as the number of users increases. This expansion is typically accompanied by a growth in the volume of data.

It's also important to understand that partitioning is not always a function of large data stores. For example, a small data store might be heavily accessed by hundreds of concurrent clients. Partitioning the data in this situation can help to reduce contention and improve throughput.

Consider the following points when you design a data partitioning scheme:

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

logic ensures that the updates are all completed successfully. It also handles the inconsistencies that can arise from querying data while an eventually consistent operation is running. For more information about implementing eventual consistency, see the [Data consistency primer](#).

- **Consider how queries locate the correct partition.** If a query must scan all partitions to locate the required data, there is a significant impact on performance, even when multiple parallel queries are running. Queries that are used with vertical and functional partitioning strategies can naturally specify the partitions. However, horizontal partitioning (sharding) can make locating an item difficult because every shard has the same schema. A typical solution for sharding is to maintain a map that can be used to look up the shard location for specific items of data. This map can be implemented in the sharding logic of the application, or maintained by the data store if it supports transparent sharding.
- **When using a horizontal partitioning strategy, consider periodically rebalancing the shards.** This helps distribute the data evenly by size and by workload to minimize hotspots, maximize query performance, and work around physical storage limitations. However, this is a complex task that often requires the use of a custom tool or process.
- **If you replicate each partition, it provides additional protection against failure.** If a single replica fails, queries can be directed towards a working copy.
- **If you reach the physical limits of a partitioning strategy, you might need to extend the scalability to a different level.** For example, if partitioning is at the database level, you might need to locate or replicate partitions in multiple databases. If partitioning is already at the database level, and physical limitations are an issue, it might mean that you need to locate or replicate partitions in multiple hosting accounts.
- **Avoid transactions that access data in multiple partitions.** Some data stores implement transactional consistency and integrity for operations that modify data, but only when the data is located in a single partition. If you need transactional support across multiple partitions, you will probably need to implement this as part of your application logic because most partitioning systems do not provide native support.

All data stores require some operational management and monitoring activity. The tasks can range from loading data, backing up and restoring data, reorganizing data, and ensuring that the system is performing correctly and efficiently.

Consider the following factors that affect operational management:

- **How to implement appropriate management and operational tasks when the data is partitioned.** These tasks might include backup and restore, archiving data, monitoring the system, and other administrative tasks. For example, maintaining logical consistency during backup and restore operations can be a challenge.
- **How to load the data into multiple partitions and add new data that's arriving from other sources.** Some tools and utilities might not support sharded data operations such as loading data into the correct partition. This means that you might have to create or obtain new tools and utilities.
- **How to archive and delete the data on a regular basis.** To prevent the excessive growth of partitions, you need to archive and delete data on a regular basis (perhaps monthly). It might be necessary to transform the data to match a different archive schema.
- **How to locate data integrity issues.** Consider running a periodic process to locate any data integrity issues such as data in one partition that references missing information in another. The process can either attempt to fix these issues automatically or raise an alert to an operator to correct the problems manually. For example, in an e-commerce application, order information might be held in one partition but the line items that constitute each order might be held in another. The process of placing an order needs to add data to other partitions. If this process fails, there might be line items stored for which there is no corresponding order.

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

Partitioning strategies for Azure SQL Database

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

Horizontal partitioning with Elastic Database

A single SQL database has a limit to the volume of data that it can contain. Throughput is constrained by architectural factors and the number of concurrent connections that it supports. The Elastic Database feature of SQL Database supports horizontal scaling for a SQL database. Using Elastic Database, you can partition your data into shards that are spread across multiple SQL databases. You can also add or remove shards as the volume of data that you need to handle grows and shrinks. Using Elastic Database can also help reduce contention by distributing the load across databases.

NOTE

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

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

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

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

NOTE

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

An alternative approach is to use Azure SQL Data Sync or an Azure Data Factory pipeline to replicate the shard map manager database across regions. This form of replication runs periodically and is more suitable if the shard map changes infrequently. Additionally, the shard map manager database does not have to be created by using a Premium pricing tier.

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

- A **list shard map** describes an association between a single key and a shardlet. For example, in a multitenant system, the data for each tenant can be associated with a unique key and stored in its own shardlet. To guarantee privacy and isolation (that is, to prevent one tenant from exhausting the data storage resources

available to others), each shardlet can be held within its own shard.

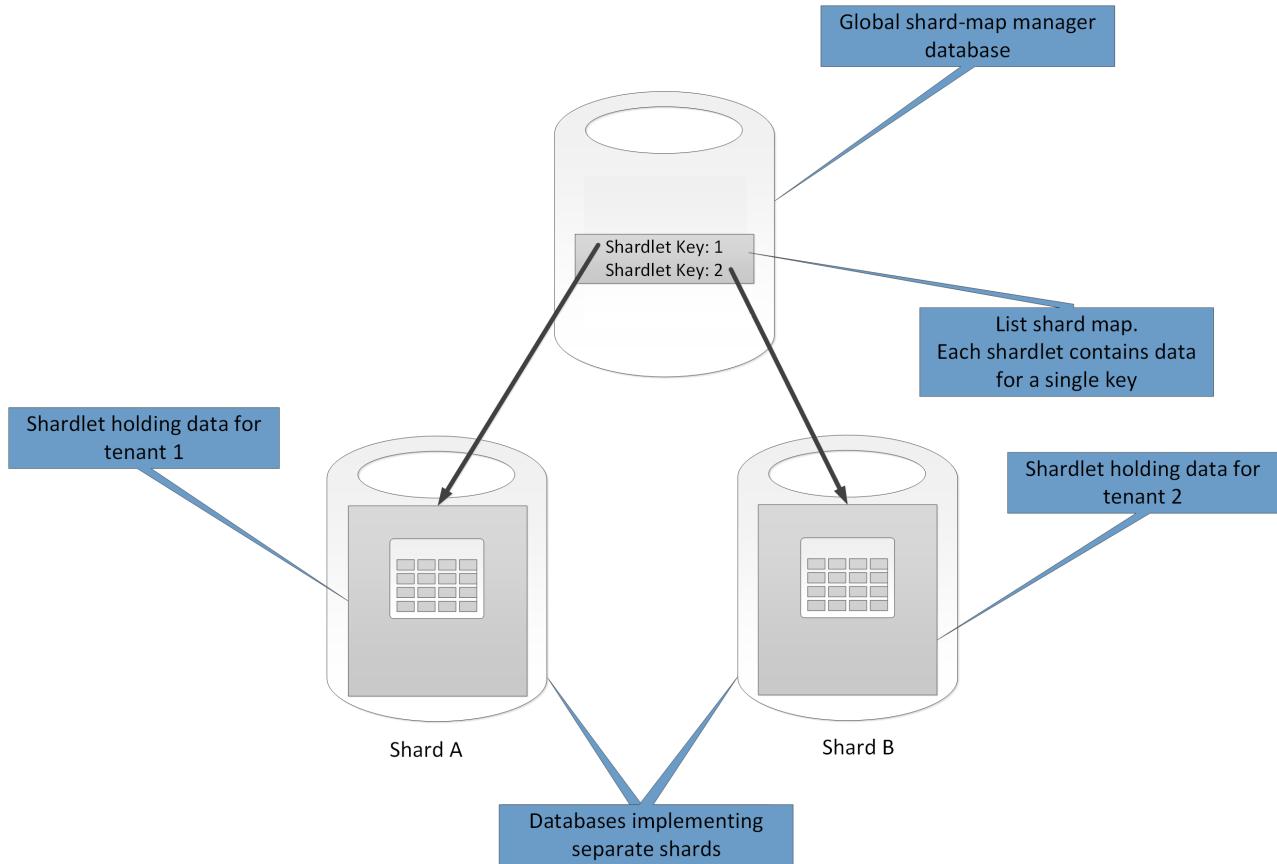


Figure 4. Using a list shard map to store tenant data in separate shards

- A **range shard map** describes an association between a set of contiguous key values and a shardlet. In the multitenant example described previously, as an alternative to implementing dedicated shardlets, you can group the data for a set of tenants (each with their own key) within the same shardlet. This scheme is less expensive than the first (because tenants share data storage resources), but it also creates a risk of reduced data privacy and isolation.

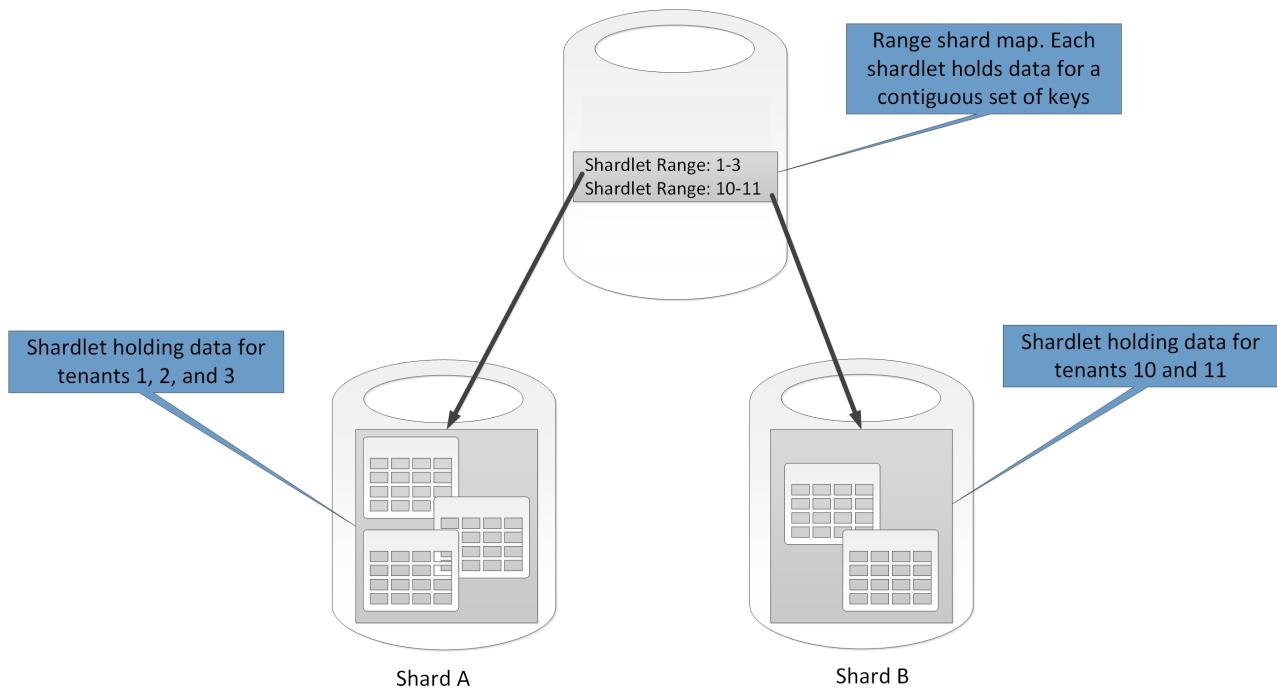


Figure 5. Using a range shard map to store data for a range of tenants in a shard

Note that a single shard can contain the data for several shardlets. For example, you can use list shardlets to store

data for different non-contiguous tenants in the same shard. You can also mix range shardlets and list shardlets in the same shard, although they will be addressed through different maps in the global shard map manager database. (The global shard map manager database can contain multiple shard maps.) Figure 6 depicts this approach.

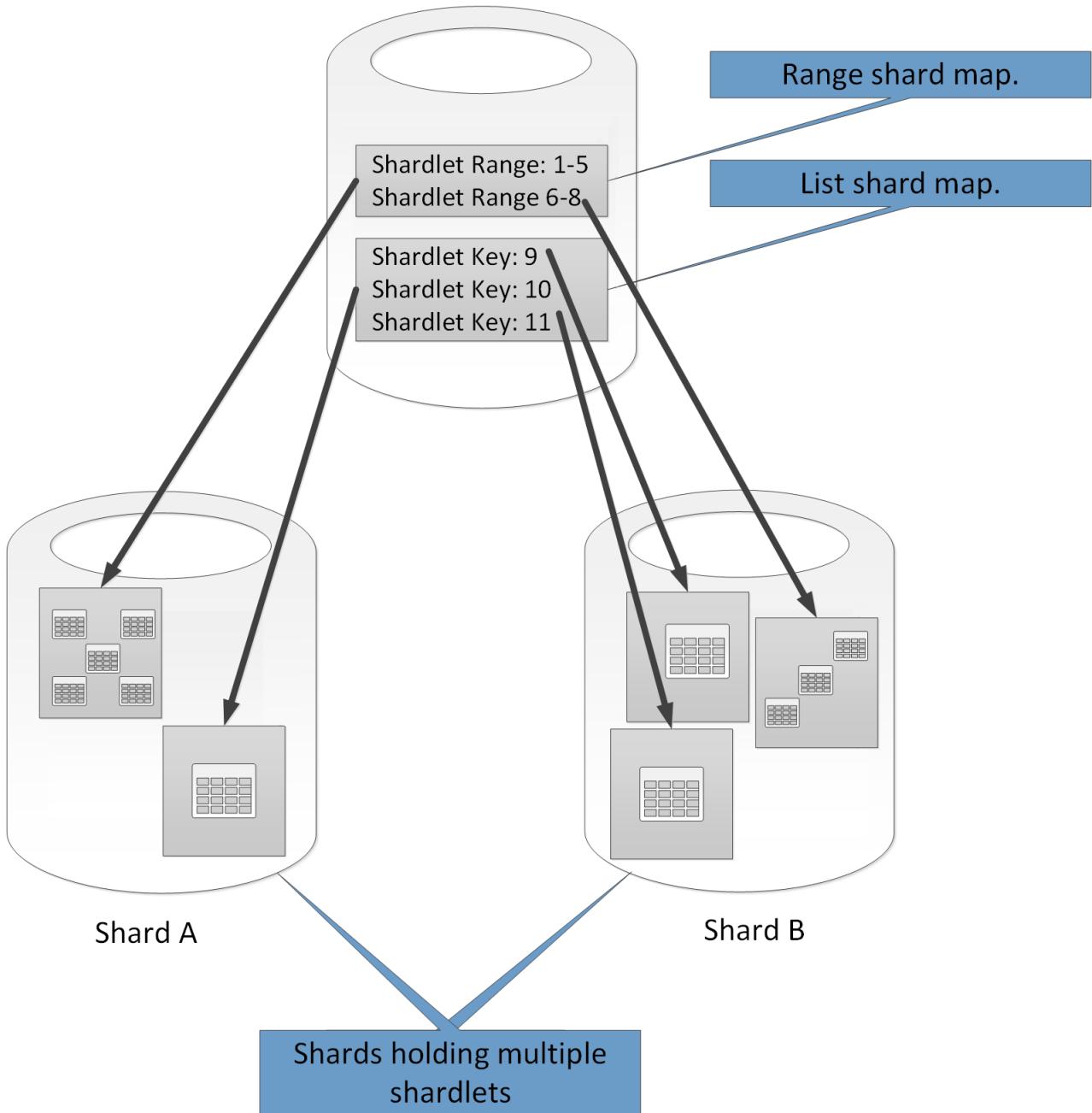


Figure 6. Implementing multiple shard maps

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

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

of it becoming stale.

NOTE

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

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

NOTE

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

- Transactional operations are only supported for data that's held within the same shard, and not across shards. Transactions can span shardlets as long as they are part of the same shard. Therefore, if your business logic needs to perform transactions, either store the affected data in the same shard or implement eventual consistency. For more information, see the [Data consistency primer](#).
- Place shards close to the users that access the data in those shards (in other words, geo-locate the shards). This strategy helps reduce latency.
- Avoid having a mixture of highly active (hotspots) and relatively inactive shards. Try to spread the load evenly across shards. This might require hashing the shardlet keys.
- If you are geo-locating shards, make sure that the hashed keys map to shardlets held in shards stored close to the users that access that data.
- Currently, only a limited set of SQL data types are supported as shardlet keys; *int*, *bigint*, *varbinary*, and *uniqueidentifier*. The SQL *int* and *bigint* types correspond to the *int* and *long* data types in C#, and have the same ranges. The SQL *varbinary* type can be handled by using a *Byte* array in C#, and the SQL *uniqueidentier* type corresponds to the *Guid* class in the .NET Framework.

As the name implies, Elastic Database makes it possible for a system to add and remove shards as the volume of data shrinks and grows. The APIs in the Azure SQL Database Elastic Database client library enable an application to create and delete shards dynamically (and transparently update the shard map manager). However, removing a shard is a destructive operation that also requires deleting all the data in that shard.

If an application needs to split a shard into two separate shards or combine shards, Elastic Database provides a separate split-merge service. This service runs in a cloud-hosted service (which must be created by the developer) and migrates data safely between shards. For more information, see the topic [Scaling using the Elastic Database split-merge tool](#) on the Microsoft website.

Partitioning strategies for Azure Storage

Azure storage provides four abstractions for managing data:

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

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

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

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

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

Partitioning Azure table storage

Azure table storage is a key-value store that's designed around partitioning. All entities are stored in a partition, and partitions are managed internally by Azure table storage. Each entity that's stored in a table must provide a two-part key that includes:

- **The partition key**. This is a string value that determines in which partition Azure table storage will place the entity. All entities with the same partition key will be stored in the same partition.
- **The row key**. This is another string value that identifies the entity within the partition. All entities within a partition are sorted lexically, in ascending order, by this key. The partition key/row key combination must be unique for each entity and cannot exceed 1 KB in length.

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

If you are attempting to store entities that exceed this capacity, then consider splitting them into multiple tables.

Use vertical partitioning to divide the fields into the groups that are most likely to be accessed together.

Figure 7 shows the logical structure of an example storage account (Contoso Data) for a fictitious e-commerce application. The storage account contains three tables: Customer Info, Product Info, and Order Info. Each table has multiple partitions.

In the Customer Info table, the data is partitioned according to the city in which the customer is located, and the row key contains the customer ID. In the Product Info table, the products are partitioned by product category, and the row key contains the product number. In the Order Info table, the orders are partitioned by the date on which they were placed, and the row key specifies the time the order was received. Note that all data is ordered by the row key in each partition.

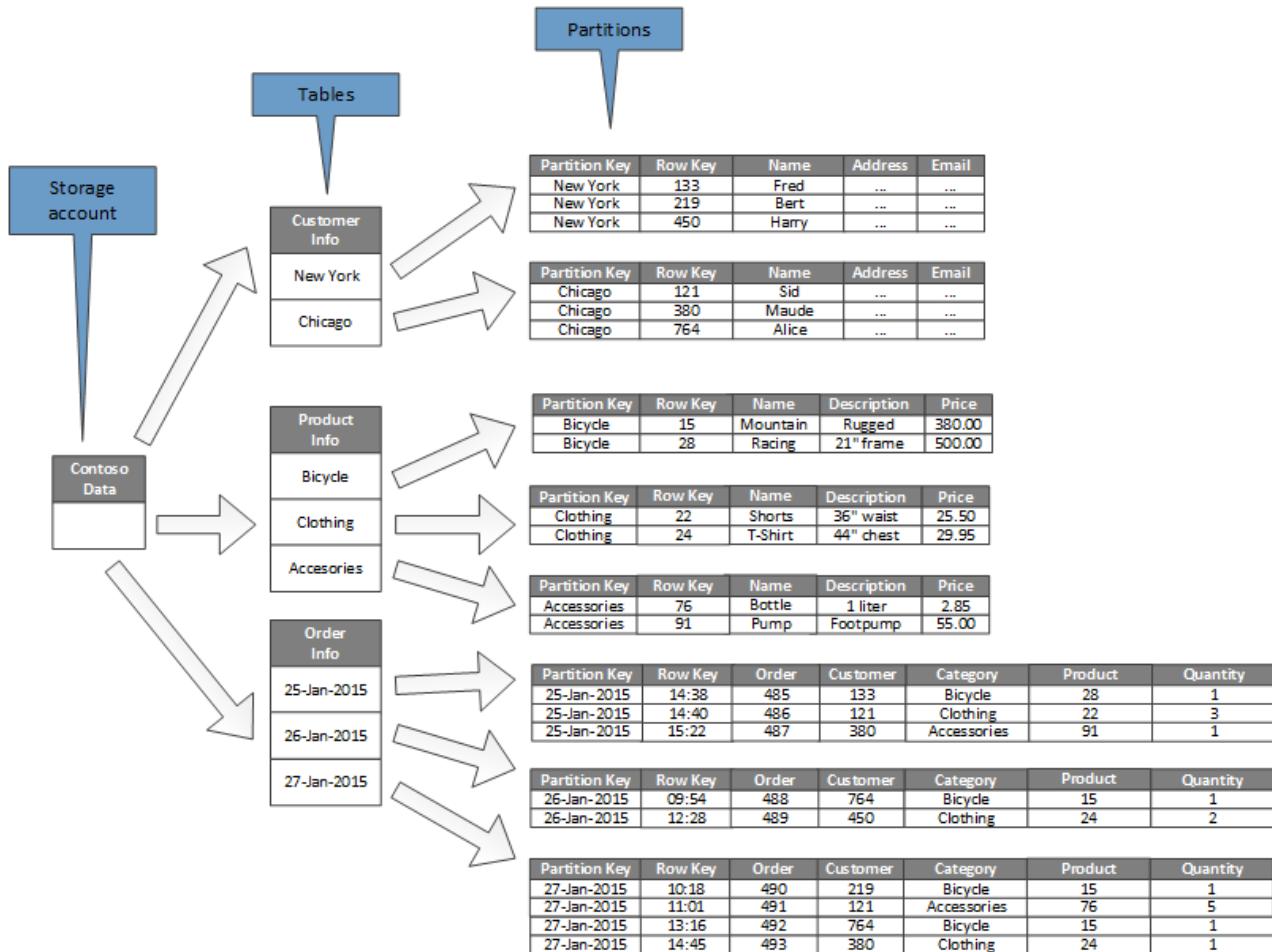


Figure 7. The tables and partitions in an example storage account

NOTE

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

Azure table storage uses the partition key to determine how to store the data. If an entity is added to a table with a previously unused partition key, Azure table storage creates a new partition for this entity. Other entities with the same partition key will be stored in the same partition.

This mechanism effectively implements an automatic scale-out strategy. Each partition is stored on a single server in an Azure datacenter to help ensure that queries that retrieve data from a single partition run quickly. However,

different partitions can be distributed across multiple servers. Additionally, a single server can host multiple partitions if these partitions are limited in size.

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

- The selection of partition key and row key values should be driven by the way in which the data is accessed. Choose a partition key/row key combination that supports the majority of your queries. The most efficient queries retrieve data by specifying the partition key and the row key. Queries that specify a partition key and a range of row keys can be completed by scanning a single partition. This is relatively fast because the data is held in row key order. If queries don't specify which partition to scan, the partition key might require Azure table storage to scan every partition for your data.

TIP

If an entity has one natural key, then use it as the partition key and specify an empty string as the row key. If an entity has a composite key comprising two properties, select the slowest changing property as the partition key and the other as the row key. If an entity has more than two key properties, use a concatenation of properties to provide the partition and row keys.

- If you regularly perform queries that look up data by using fields other than the partition and row keys, consider implementing the [index table pattern](#).
- If you generate partition keys by using a monotonic increasing or decreasing sequence (such as "0001", "0002", "0003", and so on) and each partition only contains a limited amount of data, then Azure table storage can physically group these partitions together on the same server. This mechanism assumes that the application is most likely to perform queries across a contiguous range of partitions (range queries) and is optimized for this case. However, this approach can lead to hotspots focused on a single server because all insertions of new entities are likely to be concentrated at one end or the other of the contiguous ranges. It can also reduce scalability. To spread the load more evenly across servers, consider hashing the partition key to make the sequence more random.
- Azure table storage supports transactional operations for entities that belong to the same partition. This means that an application can perform multiple insert, update, delete, replace, or merge operations as an atomic unit (as long as the transaction doesn't include more than 100 entities and the payload of the request doesn't exceed 4 MB). Operations that span multiple partitions are not transactional, and might require you to implement eventual consistency as described by the [Data consistency primer](#). For more information about table storage and transactions, go to the page [Performing entity group transactions](#) on the Microsoft website.
- Give careful attention to the granularity of the partition key because of the following reasons:
 - Using the same partition key for every entity causes the table storage service to create a single large partition that's held on one server. This prevents it from scaling out and instead focuses the load on a single server. As a result, this approach is only suitable for systems that manage a small number of entities. However, this approach does ensure that all entities can participate in entity group transactions.
 - Using a unique partition key for every entity causes the table storage service to create a separate partition for each entity, possibly resulting in a large number of small partitions (depending on the size of the entities). This approach is more scalable than using a single partition key, but entity group transactions are not possible. Also, queries that fetch more than one entity might involve reading from more than one server. However, if the application performs range queries, then using a monotonic sequence to generate the partition keys might help to optimize these queries.
 - Sharing the partition key across a subset of entities makes it possible for you to group related entities in the same partition. Operations that involve related entities can be performed by using entity group transactions, and queries that fetch a set of related entities can be satisfied by accessing a single server.

For additional information about partitioning data in Azure table storage, see the article [Azure storage table design guide](#) on the Microsoft website.

Partitioning Azure blob storage

Azure blob storage makes it possible to hold large binary objects--currently up to 5 TB in size for block blobs or 1 TB for page blobs. (For the most recent information, go to the page [Azure Storage scalability and performance targets](#) on the Microsoft website.) Use block blobs in scenarios such as streaming where you need to upload or download large volumes of data quickly. Use page blobs for applications that require random rather than serial access to parts of the data.

Each blob (either block or page) is held in a container in an Azure storage account. You can use containers to group related blobs that have the same security requirements. This grouping is logical rather than physical. Inside a container, each blob has a unique name.

The partition key for a blob is account name + container name + blob name. This means each blob can have its own partition if load on the blob demands it. Blobs can be distributed across many servers in order to scale out access to them, but a single blob can only be served by a single server.

The actions of writing a single block (block blob) or page (page blob) are atomic, but operations that span blocks, pages, or blobs are not. If you need to ensure consistency when performing write operations across blocks, pages, and blobs, take out a write lock by using a blob lease.

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

Partitioning Azure storage queues

Azure storage queues enable you to implement asynchronous messaging between processes. An Azure storage account can contain any number of queues, and each queue can contain any number of messages. The only limitation is the space that's available in the storage account. The maximum size of an individual message is 64 KB. If you require messages bigger than this, then consider using Azure Service Bus queues instead.

Each storage queue has a unique name within the storage account that contains it. Azure partitions queues based on the name. All messages for the same queue are stored in the same partition, which is controlled by a single server. Different queues can be managed by different servers to help balance the load. The allocation of queues to servers is transparent to applications and users.

In a large-scale application, don't use the same storage queue for all instances of the application because this approach might cause the server that's hosting the queue to become a hotspot. Instead, use different queues for different functional areas of the application. Azure storage queues do not support transactions, so directing messages to different queues should have little impact on messaging consistency.

An Azure storage queue can handle up to 2,000 messages per second. If you need to process messages at a greater rate than this, consider creating multiple queues. For example, in a global application, create separate storage queues in separate storage accounts to handle application instances that are running in each region.

Partitioning strategies for Azure Service Bus

Azure Service Bus uses a message broker to handle messages that are sent to a Service Bus queue or topic. By default, all messages that are sent to a queue or topic are handled by the same message broker process. This architecture can place a limitation on the overall throughput of the message queue. However, you can also partition a queue or topic when it is created. You do this by setting the *EnablePartitioning* property of the queue or topic description to *true*.

A partitioned queue or topic is divided into multiple fragments, each of which is backed by a separate message store and message broker. Service Bus takes responsibility for creating and managing these fragments. When an application posts a message to a partitioned queue or topic, Service Bus assigns the message to a fragment for that queue or topic. When an application receives a message from a queue or subscription, Service Bus checks each fragment for the next available message and then passes it to the application for processing.

This structure helps distribute the load across message brokers and message stores, increasing scalability and improving availability. If the message broker or message store for one fragment is temporarily unavailable, Service Bus can retrieve messages from one of the remaining available fragments.

Service Bus assigns a message to a fragment as follows:

- If the message belongs to a session, all messages with the same value for the * `SessionId`* property are sent to the same fragment.
- If the message does not belong to a session, but the sender has specified a value for the `PartitionKey` property, then all messages with the same `PartitionKey` value are sent to the same fragment.

NOTE

If the `SessionId` and `PartitionKey` properties are both specified, then they must be set to the same value or the message will be rejected.

- If the `SessionId` and `PartitionKey` properties for a message are not specified, but duplicate detection is enabled, the `MessageId` property will be used. All messages with the same `MessageId` will be directed to the same fragment.
- If messages do not include a `SessionId`, `PartitionKey`, or `MessageId` property, then Service Bus assigns messages to fragments sequentially. If a fragment is unavailable, Service Bus will move on to the next. This means that a temporary fault in the messaging infrastructure does not cause the message-send operation to fail.

Consider the following points when deciding if or how to partition a Service Bus message queue or topic:

- Service Bus queues and topics are created within the scope of a Service Bus namespace. Service Bus currently allows up to 100 partitioned queues or topics per namespace.
- Each Service Bus namespace imposes quotas on the available resources, such as the number of subscriptions per topic, the number of concurrent send and receive requests per second, and the maximum number of concurrent connections that can be established. These quotas are documented on the Microsoft website on the page [Service Bus quotas](#). If you expect to exceed these values, then create additional namespaces with their own queues and topics, and spread the work across these namespaces. For example, in a global application, create separate namespaces in each region and configure application instances to use the queues and topics in the nearest namespace.
- Messages that are sent as part of a transaction must specify a partition key. This can be a `SessionId`, `PartitionKey`, or `MessageId` property. All messages that are sent as part of the same transaction must specify the same partition key because they must be handled by the same message broker process. You cannot send messages to different queues or topics within the same transaction.
- Partitioned queues and topics can't be configured to be automatically deleted when they become idle.
- Partitioned queues and topics can't currently be used with the Advanced Message Queuing Protocol (AMQP) if you are building cross-platform or hybrid solutions.

Partitioning strategies for Cosmos DB

Azure Cosmos DB is a NoSQL database that can store JSON documents using the [Azure Cosmos DB SQL API](#). A document in a Cosmos DB database is a JSON-serialized representation of an object or other piece of data. No

fixed schemas are enforced except that every document must contain a unique ID.

Documents are organized into collections. You can group related documents together in a collection. For example, in a system that maintains blog postings, you can store the contents of each blog post as a document in a collection. You can also create collections for each subject type. Alternatively, in a multitenant application, such as a system where different authors control and manage their own blog posts, you can partition blogs by author and create separate collections for each author. The storage space that's allocated to collections is elastic and can shrink or grow as needed.

Cosmos DB supports automatic partitioning of data based on an application-defined partition key. A *logical partition* is a partition that stores all the data for a single partition key value. All documents that share the same value for the partition key are placed within the same logical partition. Cosmos DB distributes values according to hash of the partition key. A logical partition has a maximum size of 10 GB. Therefore, the choice of the partition key is an important decision at design time. Choose a property with a wide range of values and even access patterns. For more information, see [Partition and scale in Azure Cosmos DB](#).

NOTE

Each Cosmos DB database has a *performance level* that determines the amount of resources it gets. A performance level is associated with a *request unit* (RU) rate limit. The RU rate limit specifies the volume of resources that's reserved and available for exclusive use by that collection. The cost of a collection depends on the performance level that's selected for that collection. The higher the performance level (and RU rate limit) the higher the charge. You can adjust the performance level of a collection by using the Azure portal. For more information, see [Request Units in Azure Cosmos DB](#).

If the partitioning mechanism that Cosmos DB provides is not sufficient, you may need to shard the data at the application level. Document collections provide a natural mechanism for partitioning data within a single database. The simplest way to implement sharding is to create a collection for each shard. Containers are logical resources and can span one or more servers. Fixed-size containers have a maximum limit of 10 GB and 10,000 RU/s throughput. Unlimited containers do not have a maximum storage size, but must specify a partition key. With application sharding, the client application must direct requests to the appropriate shard, usually by implementing its own mapping mechanism based on some attributes of the data that define the shard key.

All databases are created in the context of a Cosmos DB database account. A single account can contain several databases, and it specifies in which regions the databases are created. Each account also enforces its own access control. You can use Cosmos DB accounts to geo-locate shards (collections within databases) close to the users who need to access them, and enforce restrictions so that only those users can connect to them.

Consider the following points when deciding how to partition data with the Cosmos DB SQL API:

- **The resources available to a Cosmos DB database are subject to the quota limitations of the account.** Each database can hold a number of collections, and each collection is associated with a performance level that governs the RU rate limit (reserved throughput) for that collection. For more information, see [Azure subscription and service limits, quotas, and constraints](#).
- **Each document must have an attribute that can be used to uniquely identify that document within the collection in which it is held.** This attribute is different from the shard key, which defines which collection holds the document. A collection can contain a large number of documents. In theory, it's limited only by the maximum length of the document ID. The document ID can be up to 255 characters.
- **All operations against a document are performed within the context of a transaction. Transactions are scoped to the collection in which the document is contained.** If an operation fails, the work that it has performed is rolled back. While a document is subject to an operation, any changes that are made are subject to snapshot-level isolation. This mechanism guarantees that if, for example, a request to create a new document fails, another user who's querying the database simultaneously will not see a partial document that is then removed.
- **Database queries are also scoped to the collection level.** A single query can retrieve data from only one

collection. If you need to retrieve data from multiple collections, you must query each collection individually and merge the results in your application code.

- **Cosmos DB supports programmable items that can all be stored in a collection alongside documents.**

These include stored procedures, user-defined functions, and triggers (written in JavaScript). These items can access any document within the same collection. Furthermore, these items run either inside the scope of the ambient transaction (in the case of a trigger that fires as the result of a create, delete, or replace operation performed against a document), or by starting a new transaction (in the case of a stored procedure that is run as the result of an explicit client request). If the code in a programmable item throws an exception, the transaction is rolled back. You can use stored procedures and triggers to maintain integrity and consistency between documents, but these documents must all be part of the same collection.

- **The collections that you intend to hold in the databases should be unlikely to exceed the throughput limits defined by the performance levels of the collections.** For more information, see [Request Units in Azure Cosmos DB](#). If you anticipate reaching these limits, consider splitting collections across databases in different accounts to reduce the load per collection.

Partitioning strategies for Azure Search

The ability to search for data is often the primary method of navigation and exploration that's provided by many web applications. It helps users find resources quickly (for example, products in an e-commerce application) based on combinations of search criteria. The Azure Search service provides full-text search capabilities over web content, and includes features such as type-ahead, suggested queries based on near matches, and faceted navigation. A full description of these capabilities is available on the page [What is Azure Search?](#) on the Microsoft website.

Azure Search stores searchable content as JSON documents in a database. You define indexes that specify the searchable fields in these documents and provide these definitions to Azure Search. When a user submits a search request, Azure Search uses the appropriate indexes to find matching items.

To reduce contention, the storage that's used by Azure Search can be divided into 1, 2, 3, 4, 6, or 12 partitions, and each partition can be replicated up to 6 times. The product of the number of partitions multiplied by the number of replicas is called the *search unit* (SU). A single instance of Azure Search can contain a maximum of 36 SUs (a database with 12 partitions only supports a maximum of 3 replicas).

You are billed for each SU that is allocated to your service. As the volume of searchable content increases or the rate of search requests grows, you can add SUs to an existing instance of Azure Search to handle the extra load. Azure Search itself distributes the documents evenly across the partitions. No manual partitioning strategies are currently supported.

Each partition can contain a maximum of 15 million documents or occupy 300 GB of storage space (whichever is smaller). You can create up to 50 indexes. The performance of the service varies and depends on the complexity of the documents, the available indexes, and the effects of network latency. On average, a single replica (1 SU) should be able to handle 15 queries per second (QPS), although we recommend performing benchmarking with your own data to obtain a more precise measure of throughput. For more information, see the page [Service limits in Azure Search](#) on the Microsoft website.

NOTE

You can store a limited set of data types in searchable documents, including strings, Booleans, numeric data, datetime data, and some geographical data. For more details, see the page [Supported data types \(Azure Search\)](#) on the Microsoft website.

You have limited control over how Azure Search partitions data for each instance of the service. However, in a global environment you might be able to improve performance and reduce latency and contention further by partitioning the service itself using either of the following strategies:

- Create an instance of Azure Search in each geographic region, and ensure that client applications are directed towards the nearest available instance. This strategy requires that any updates to searchable content are replicated in a timely manner across all instances of the service.
- Create two tiers of Azure Search:
 - A local service in each region that contains the data that's most frequently accessed by users in that region. Users can direct requests here for fast but limited results.
 - A global service that encompasses all the data. Users can direct requests here for slower but more complete results.

This approach is most suitable when there is a significant regional variation in the data that's being searched.

Partitioning strategies for Azure Redis Cache

Azure Redis Cache provides a shared caching service in the cloud that's based on the Redis key-value data store. As its name implies, Azure Redis Cache is intended as a caching solution. Use it only for holding transient data and not as a permanent data store. Applications that utilize Azure Redis Cache should be able to continue functioning if the cache is unavailable. Azure Redis Cache supports primary/secondary replication to provide high availability, but currently limits the maximum cache size to 53 GB. If you need more space than this, you must create additional caches. For more information, go to the page [Azure Redis Cache](#) on the Microsoft website.

Partitioning a Redis data store involves splitting the data across instances of the Redis service. Each instance constitutes a single partition. Azure Redis Cache abstracts the Redis services behind a façade and does not expose them directly. The simplest way to implement partitioning is to create multiple Azure Redis Cache instances and spread the data across them.

You can associate each data item with an identifier (a partition key) that specifies which cache stores the data item. The client application logic can then use this identifier to route requests to the appropriate partition. This scheme is very simple, but if the partitioning scheme changes (for example, if additional Azure Redis Cache instances are created), client applications might need to be reconfigured.

Native Redis (not Azure Redis Cache) supports server-side partitioning based on Redis clustering. In this approach, you can divide the data evenly across servers by using a hashing mechanism. Each Redis server stores metadata that describes the range of hash keys that the partition holds, and also contains information about which hash keys are located in the partitions on other servers.

Client applications simply send requests to any of the participating Redis servers (probably the closest one). The Redis server examines the client request. If it can be resolved locally, it performs the requested operation. Otherwise it forwards the request on to the appropriate server.

This model is implemented by using Redis clustering, and is described in more detail on the [Redis cluster tutorial](#) page on the Redis website. Redis clustering is transparent to client applications. Additional Redis servers can be added to the cluster (and the data can be re-partitioned) without requiring that you reconfigure the clients.

IMPORTANT

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

The page [Partitioning: how to split data among multiple Redis instances](#) on the Redis website provides more information about implementing partitioning with Redis. The remainder of this section assumes that you are implementing client-side or proxy-assisted partitioning.

Consider the following points when deciding how to partition data with Azure Redis Cache:

- Azure Redis Cache is not intended to act as a permanent data store, so whatever partitioning scheme you implement, your application code must be able to retrieve data from a location that's not the cache.
- Data that is frequently accessed together should be kept in the same partition. Redis is a powerful key-value store that provides several highly optimized mechanisms for structuring data. These mechanisms can be one of the following:
 - Simple strings (binary data up to 512 MB in length)
 - Aggregate types such as lists (which can act as queues and stacks)
 - Sets (ordered and unordered)
 - Hashes (which can group related fields together, such as the items that represent the fields in an object)
- The aggregate types enable you to associate many related values with the same key. A Redis key identifies a list, set, or hash rather than the data items that it contains. These types are all available with Azure Redis Cache and are described by the [Data types](#) page on the Redis website. For example, in part of an e-commerce system that tracks the orders that are placed by customers, the details of each customer can be stored in a Redis hash that is keyed by using the customer ID. Each hash can hold a collection of order IDs for the customer. A separate Redis set can hold the orders, again structured as hashes, and keyed by using the order ID. Figure 8 shows this structure. Note that Redis does not implement any form of referential integrity, so it is the developer's responsibility to maintain the relationships between customers and orders.

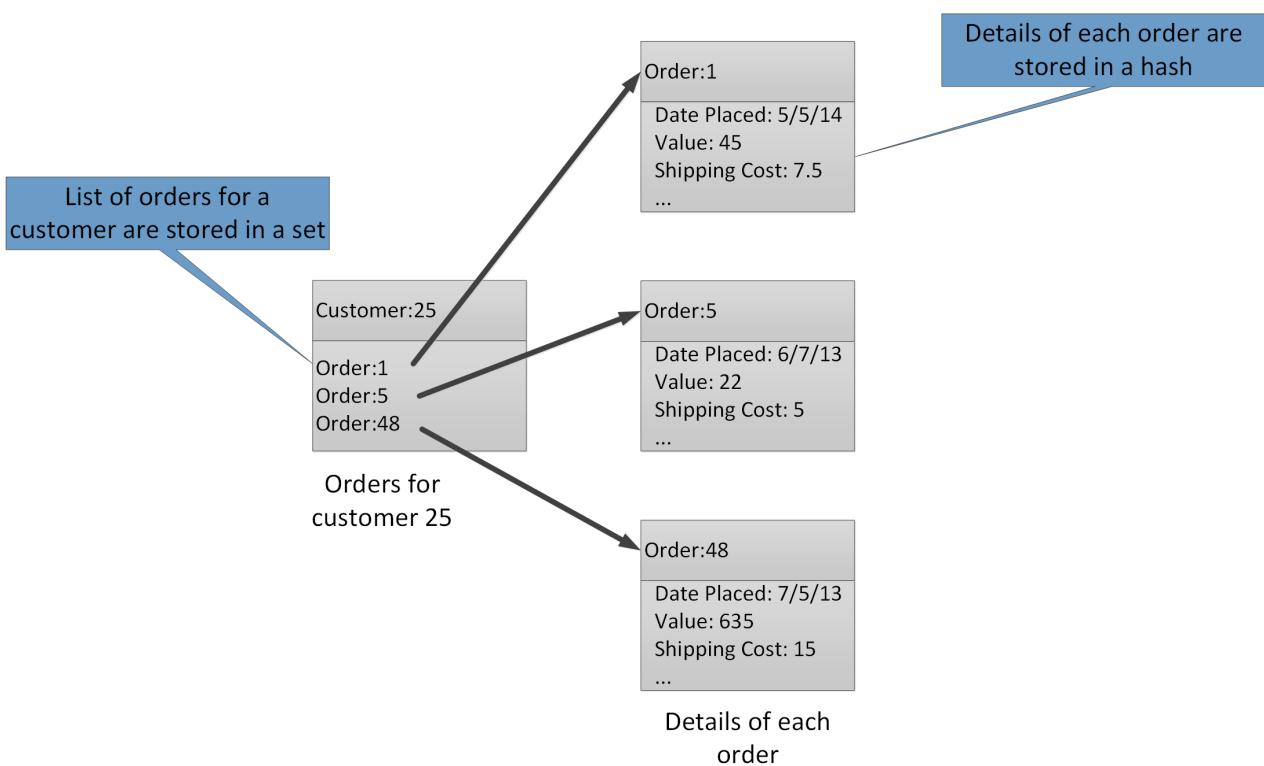


Figure 8. Suggested structure in Redis storage for recording customer orders and their details

NOTE

In Redis, all keys are binary data values (like Redis strings) and can contain up to 512 MB of data. In theory, a key can contain almost any information. However, we recommend adopting a consistent naming convention for keys that is descriptive of the type of data and that identifies the entity, but is not excessively long. A common approach is to use keys of the form "entity_type:ID". For example, you can use "customer:99" to indicate the key for a customer with the ID 99.

- You can implement vertical partitioning by storing related information in different aggregations in the same database. For example, in an e-commerce application, you can store commonly accessed information about

products in one Redis hash and less frequently used detailed information in another. Both hashes can use the same product ID as part of the key. For example, you can use "product: nn" (where *nn* is the product ID) for the product information and "product_details: nn" for the detailed data. This strategy can help reduce the volume of data that most queries are likely to retrieve.

- You can repartition a Redis data store, but keep in mind that it's a complex and time-consuming task. Redis clustering can repartition data automatically, but this capability is not available with Azure Redis Cache. Therefore, when you design your partitioning scheme, try to leave sufficient free space in each partition to allow for expected data growth over time. However, remember that Azure Redis Cache is intended to cache data temporarily, and that data held in the cache can have a limited lifetime specified as a time-to-live (TTL) value. For relatively volatile data, the TTL can be short, but for static data the TTL can be a lot longer. Avoid storing large amounts of long-lived data in the cache if the volume of this data is likely to fill the cache. You can specify an eviction policy that causes Azure Redis Cache to remove data if space is at a premium.

NOTE

When you use Azure Redis cache, you specify the maximum size of the cache (from 250 MB to 53 GB) by selecting the appropriate pricing tier. However, after an Azure Redis Cache has been created, you cannot increase (or decrease) its size.

- Redis batches and transactions cannot span multiple connections, so all data that is affected by a batch or transaction should be held in the same database (shard).

NOTE

A sequence of operations in a Redis transaction is not necessarily atomic. The commands that compose a transaction are verified and queued before they run. If an error occurs during this phase, the entire queue is discarded. However, after the transaction has been successfully submitted, the queued commands run in sequence. If any command fails, only that command stops running. All previous and subsequent commands in the queue are performed. For more information, go to the [Transactions](#) page on the Redis website.

- Redis supports a limited number of atomic operations. The only operations of this type that support multiple keys and values are MGET and MSET operations. MGET operations return a collection of values for a specified list of keys, and MSET operations store a collection of values for a specified list of keys. If you need to use these operations, the key-value pairs that are referenced by the MSET and MGET commands must be stored within the same database.

Partitioning Strategies for Azure Service Fabric

Azure Service Fabric is a microservices platform that provides a runtime for distributed applications in the cloud. Service Fabric supports .Net guest executables, stateful and stateless services, and containers. Stateful services provide a [reliable collection](#) to persistently store data in a key-value collection within the Service Fabric cluster. For more information about strategies for partitioning keys in a reliable collection, see [guidelines and recommendations for reliable collections in Azure Service Fabric](#).

More information

- [Overview of Azure Service Fabric](#) is an introduction to Azure Service Fabric.
- [Partition Service Fabric reliable services](#) provides more information about reliable services in Azure Service Fabric.

Partitioning strategies for Azure Event Hubs

[Azure Event Hubs](#) is designed for data streaming at massive scale, and partitioning is built into the service to

enable horizontal scaling. Each consumer only reads a specific partition of the message stream.

The event publisher is only aware of its partition key, not the partition to which the events are published. This decoupling of key and partition insulates the sender from needing to know too much about the downstream processing. (It's also possible send events directly to a given partition, but generally that's not recommended.)

Consider long-term scale when you select the partition count. After an event hub is created, you can't change the number of partitions.

For more information about using partitions in Event Hubs, see [What is Event Hubs?](#).

For considerations about trade-offs between availability and consistency, see [Availability and consistency in Event Hubs](#).

Rebalancing partitions

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

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

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

NOTE

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

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

Offline migration

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

Conceptually, this process includes the following steps:

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

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

Online migration

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

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

Related patterns and guidance

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

- The [Data consistency primer](#) page on the Microsoft website describes strategies for maintaining consistency in a distributed environment such as the cloud.
- The [Data partitioning guidance](#) page on the Microsoft website provides a general overview of how to design partitions to meet various criteria in a distributed solution.
- The [sharding pattern](#) as described on the Microsoft website summarizes some common strategies for sharding data.
- The [index table pattern](#) as described on the Microsoft website illustrates how to create secondary indexes over data. An application can quickly retrieve data with this approach, by using queries that do not reference the primary key of a collection.
- The [materialized view pattern](#) as described on the Microsoft website describes how to generate pre-populated views that summarize data to support fast query operations. This approach can be useful in a partitioned data store if the partitions that contain the data being summarized are distributed across multiple sites.
- The [Using Azure Content Delivery Network](#) article on the Microsoft website provides additional guidance on configuring and using Content Delivery Network with Azure.

More information

- The page [What is Azure SQL Database?](#) on the Microsoft website provides detailed documentation that describes how to create and use SQL databases.
- The page [Elastic Database features overview](#) on the Microsoft website provides a comprehensive introduction to Elastic Database.
- The page [Scaling using the Elastic Database split-merge tool](#) on the Microsoft website contains information about using the split-merge service to manage Elastic Database shards.
- The page [Azure storage scalability and performance targets](#) on the Microsoft website documents the current sizing and throughput limits of Azure Storage.
- The page [Performing entity group transactions](#) on the Microsoft website provides detailed information about implementing transactional operations over entities that are stored in Azure table storage.
- The article [Azure Storage table design guide](#) on the Microsoft website contains detailed information about partitioning data in Azure table storage.
- The page [Using Azure Content Delivery Network](#) on the Microsoft website describes how to replicate data that's held in Azure blob storage by using the Azure Content Delivery Network.
- The page [What is Azure Search?](#) on the Microsoft website provides a full description of the capabilities that are available in Azure Search.
- The page [Service limits in Azure Search](#) on the Microsoft website contains information about the capacity of each instance of Azure Search.
- The page [Supported data types \(Azure Search\)](#) on the Microsoft website summarizes the data types that you

can use in searchable documents and indexes.

- The page [Azure Redis Cache](#) on the Microsoft website provides an introduction to Azure Redis Cache.
- The [Partitioning: how to split data among multiple Redis instances](#) page on the Redis website provides information about how to implement partitioning with Redis.
- The page [Running Redis on a CentOS Linux VM in Azure](#) on the Microsoft website walks through an example that shows you how to build and configure a Redis node running as an Azure VM.
- The [Data types](#) page on the Redis website describes the data types that are available with Redis and Azure Redis Cache.

Monitoring and diagnostics

8/14/2017 • 69 minutes to read • [Edit Online](#)

Distributed applications and services running in the cloud are, by their nature, complex pieces of software that comprise many moving parts. In a production environment, it's important to be able to track the way in which users utilize your system, trace resource utilization, and generally monitor the health and performance of your system. You can use this information as a diagnostic aid to detect and correct issues, and also to help spot potential problems and prevent them from occurring.

Monitoring and diagnostics scenarios

You can use monitoring to gain an insight into how well a system is functioning. Monitoring is a crucial part of maintaining quality-of-service targets. Common scenarios for collecting monitoring data include:

- Ensuring that the system remains healthy.
- Tracking the availability of the system and its component elements.
- Maintaining performance to ensure that the throughput of the system does not degrade unexpectedly as the volume of work increases.
- Guaranteeing that the system meets any service-level agreements (SLAs) established with customers.
- Protecting the privacy and security of the system, users, and their data.
- Tracking the operations that are performed for auditing or regulatory purposes.
- Monitoring the day-to-day usage of the system and spotting trends that might lead to problems if they're not addressed.
- Tracking issues that occur, from initial report through to analysis of possible causes, rectification, consequent software updates, and deployment.
- Tracing operations and debugging software releases.

NOTE

This list is not intended to be comprehensive. This document focuses on these scenarios as the most common situations for performing monitoring. There might be others that are less common or are specific to your environment.

The following sections describe these scenarios in more detail. The information for each scenario is discussed in the following format:

1. A brief overview of the scenario
2. The typical requirements of this scenario
3. The raw instrumentation data that's required to support the scenario, and possible sources of this information
4. How this raw data can be analyzed and combined to generate meaningful diagnostic information

Health monitoring

A system is healthy if it is running and capable of processing requests. The purpose of health monitoring is to generate a snapshot of the current health of the system so that you can verify that all components of the system are functioning as expected.

Requirements for health monitoring

An operator should be alerted quickly (within a matter of seconds) if any part of the system is deemed to be

unhealthy. The operator should be able to ascertain which parts of the system are functioning normally, and which parts are experiencing problems. System health can be highlighted through a traffic-light system:

- Red for unhealthy (the system has stopped)
- Yellow for partially healthy (the system is running with reduced functionality)
- Green for completely healthy

A comprehensive health-monitoring system enables an operator to drill down through the system to view the health status of subsystems and components. For example, if the overall system is depicted as partially healthy, the operator should be able to zoom in and determine which functionality is currently unavailable.

Data sources, instrumentation, and data-collection requirements

The raw data that's required to support health monitoring can be generated as a result of:

- Tracing execution of user requests. This information can be used to determine which requests have succeeded, which have failed, and how long each request takes.
- Synthetic user monitoring. This process simulates the steps performed by a user and follows a predefined series of steps. The results of each step should be captured.
- Logging exceptions, faults, and warnings. This information can be captured as a result of trace statements embedded into the application code, as well as retrieving information from the event logs of any services that the system references.
- Monitoring the health of any third-party services that the system uses. This monitoring might require retrieving and parsing health data that these services supply. This information might take a variety of formats.
- Endpoint monitoring. This mechanism is described in more detail in the "Availability monitoring" section.
- Collecting ambient performance information, such as background CPU utilization or I/O (including network) activity.

Analyzing health data

The primary focus of health monitoring is to quickly indicate whether the system is running. Hot analysis of the immediate data can trigger an alert if a critical component is detected as unhealthy. (It fails to respond to a consecutive series of pings, for example.) The operator can then take the appropriate corrective action.

A more advanced system might include a predictive element that performs a cold analysis over recent and current workloads. A cold analysis can spot trends and determine whether the system is likely to remain healthy or whether the system will need additional resources. This predictive element should be based on critical performance metrics, such as:

- The rate of requests directed at each service or subsystem.
- The response times of these requests.
- The volume of data flowing into and out of each service.

If the value of any metric exceeds a defined threshold, the system can raise an alert to enable an operator or autoscaling (if available) to take the preventative actions necessary to maintain system health. These actions might involve adding resources, restarting one or more services that are failing, or applying throttling to lower-priority requests.

Availability monitoring

A truly healthy system requires that the components and subsystems that compose the system are available. Availability monitoring is closely related to health monitoring. But whereas health monitoring provides an immediate view of the current health of the system, availability monitoring is concerned with tracking the availability of the system and its components to generate statistics about the uptime of the system.

In many systems, some components (such as a database) are configured with built-in redundancy to permit rapid failover in the event of a serious fault or loss of connectivity. Ideally, users should not be aware that such a failure

has occurred. But from an availability monitoring perspective, it's necessary to gather as much information as possible about such failures to determine the cause and take corrective actions to prevent them from recurring.

The data that's required to track availability might depend on a number of lower-level factors. Many of these factors might be specific to the application, system, and environment. An effective monitoring system captures the availability data that corresponds to these low-level factors and then aggregates them to give an overall picture of the system. For example, in an e-commerce system, the business functionality that enables a customer to place orders might depend on the repository where order details are stored and the payment system that handles the monetary transactions for paying for these orders. The availability of the order-placement part of the system is therefore a function of the availability of the repository and the payment subsystem.

Requirements for availability monitoring

An operator should also be able to view the historical availability of each system and subsystem, and use this information to spot any trends that might cause one or more subsystems to periodically fail. (Do services start to fail at a particular time of day that corresponds to peak processing hours?)

A monitoring solution should provide an immediate and historical view of the availability or unavailability of each subsystem. It should also be capable of quickly alerting an operator when one or more services fail or when users can't connect to services. This is a matter of not only monitoring each service, but also examining the actions that each user performs if these actions fail when they attempt to communicate with a service. To some extent, a degree of connectivity failure is normal and might be due to transient errors. But it might be useful to allow the system to raise an alert for the number of connectivity failures to a specified subsystem that occur during a specific period.

Data sources, instrumentation, and data-collection requirements

As with health monitoring, the raw data that's required to support availability monitoring can be generated as a result of synthetic user monitoring and logging any exceptions, faults, and warnings that might occur. In addition, availability data can be obtained from performing endpoint monitoring. The application can expose one or more health endpoints, each testing access to a functional area within the system. The monitoring system can ping each endpoint by following a defined schedule and collect the results (success or fail).

All timeouts, network connectivity failures, and connection retry attempts must be recorded. All data should be time-stamped.

Analyzing availability data

The instrumentation data must be aggregated and correlated to support the following types of analysis:

- The immediate availability of the system and subsystems.
- The availability failure rates of the system and subsystems. Ideally, an operator should be able to correlate failures with specific activities: what was happening when the system failed?
- A historical view of failure rates of the system or any subsystems across any specified period, and the load on the system (number of user requests, for example) when a failure occurred.
- The reasons for unavailability of the system or any subsystems. For example, the reasons might be service not running, connectivity lost, connected but timing out, and connected but returning errors.

You can calculate the percentage availability of a service over a period of time by using the following formula:

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

This is useful for SLA purposes. ([SLA monitoring](#) is described in more detail later in this guidance.) The definition of *downtime* depends on the service. For example, Visual Studio Team Services Build Service defines downtime as the period (total accumulated minutes) during which Build Service is unavailable. A minute is considered unavailable if all continuous HTTP requests to Build Service to perform customer-initiated operations throughout the minute either result in an error code or do not return a response.

Performance monitoring

As the system is placed under more and more stress (by increasing the volume of users), the size of the datasets that these users access grows and the possibility of failure of one or more components becomes more likely. Frequently, component failure is preceded by a decrease in performance. If you're able to detect such a decrease, you can take proactive steps to remedy the situation.

System performance depends on a number of factors. Each factor is typically measured through key performance indicators (KPIs), such as the number of database transactions per second or the volume of network requests that are successfully serviced in a specified time frame. Some of these KPIs might be available as specific performance measures, whereas others might be derived from a combination of metrics.

NOTE

Determining poor or good performance requires that you understand the level of performance at which the system should be capable of running. This requires observing the system while it's functioning under a typical load and capturing the data for each KPI over a period of time. This might involve running the system under a simulated load in a test environment and gathering the appropriate data before deploying the system to a production environment.

You should also ensure that monitoring for performance purposes does not become a burden on the system. You might be able to dynamically adjust the level of detail for the data that the performance monitoring process gathers.

Requirements for performance monitoring

To examine system performance, an operator typically needs to see information that includes:

- The response rates for user requests.
- The number of concurrent user requests.
- The volume of network traffic.
- The rates at which business transactions are being completed.
- The average processing time for requests.

It can also be helpful to provide tools that enable an operator to help spot correlations, such as:

- The number of concurrent users versus request latency times (how long it takes to start processing a request after the user has sent it).
- The number of concurrent users versus the average response time (how long it takes to complete a request after it has started processing).
- The volume of requests versus the number of processing errors.

Along with this high-level functional information, an operator should be able to obtain a detailed view of the performance for each component in the system. This data is typically provided through low-level performance counters that track information such as:

- Memory utilization.
- Number of threads.
- CPU processing time.
- Request queue length.
- Disk or network I/O rates and errors.
- Number of bytes written or read.
- Middleware indicators, such as queue length.

All visualizations should allow an operator to specify a time period. The displayed data might be a snapshot of the current situation and/or a historical view of the performance.

An operator should be able to raise an alert based on any performance measure for any specified value during any

specified time interval.

Data sources, instrumentation, and data-collection requirements

You can gather high-level performance data (throughput, number of concurrent users, number of business transactions, error rates, and so on) by monitoring the progress of users' requests as they arrive and pass through the system. This involves incorporating tracing statements at key points in the application code, together with timing information. All faults, exceptions, and warnings should be captured with sufficient data for correlating them with the requests that caused them. The Internet Information Services (IIS) log is another useful source.

If possible, you should also capture performance data for any external systems that the application uses. These external systems might provide their own performance counters or other features for requesting performance data. If this is not possible, record information such as the start time and end time of each request made to an external system, together with the status (success, fail, or warning) of the operation. For example, you can use a stopwatch approach to time requests: start a timer when the request starts and then stop the timer when the request finishes.

Low-level performance data for individual components in a system might be available through features and services such as Windows performance counters and Azure Diagnostics.

Analyzing performance data

Much of the analysis work consists of aggregating performance data by user request type and/or the subsystem or service to which each request is sent. An example of a user request is adding an item to a shopping cart or performing the checkout process in an e-commerce system.

Another common requirement is summarizing performance data in selected percentiles. For example, an operator might determine the response times for 99 percent of requests, 95 percent of requests, and 70 percent of requests. There might be SLA targets or other goals set for each percentile. The ongoing results should be reported in near real time to help detect immediate issues. The results should also be aggregated over the longer time for statistical purposes.

In the case of latency issues affecting performance, an operator should be able to quickly identify the cause of the bottleneck by examining the latency of each step that each request performs. The performance data must therefore provide a means of correlating performance measures for each step to tie them to a specific request.

Depending on the visualization requirements, it might be useful to generate and store a data cube that contains views of the raw data. This data cube can allow complex ad hoc querying and analysis of the performance information.

Security monitoring

All commercial systems that include sensitive data must implement a security structure. The complexity of the security mechanism is usually a function of the sensitivity of the data. In a system that requires users to be authenticated, you should record:

- All sign-in attempts, whether they fail or succeed.
- All operations performed by--and the details of all resources accessed by--an authenticated user.
- When a user ends a session and signs out.

Monitoring might be able to help detect attacks on the system. For example, a large number of failed sign-in attempts might indicate a brute-force attack. An unexpected surge in requests might be the result of a distributed denial-of-service (DDoS) attack. You must be prepared to monitor all requests to all resources regardless of the source of these requests. A system that has a sign-in vulnerability might accidentally expose resources to the outside world without requiring a user to actually sign in.

Requirements for security monitoring

The most critical aspects of security monitoring should enable an operator to quickly:

- Detect attempted intrusions by an unauthenticated entity.
- Identify attempts by entities to perform operations on data for which they have not been granted access.
- Determine whether the system, or some part of the system, is under attack from outside or inside. (For example, a malicious authenticated user might be attempting to bring the system down.)

To support these requirements, an operator should be notified:

- If one account makes repeated failed sign-in attempts within a specified period.
- If one authenticated account repeatedly tries to access a prohibited resource during a specified period.
- If a large number of unauthenticated or unauthorized requests occur during a specified period.

The information that's provided to an operator should include the host address of the source for each request. If security violations regularly arise from a particular range of addresses, these hosts might be blocked.

A key part in maintaining the security of a system is being able to quickly detect actions that deviate from the usual pattern. Information such as the number of failed and/or successful sign-in requests can be displayed visually to help detect whether there is a spike in activity at an unusual time. (An example of this activity is users signing in at 3:00 AM and performing a large number of operations when their working day starts at 9:00 AM). This information can also be used to help configure time-based autoscaling. For example, if an operator observes that a large number of users regularly sign in at a particular time of day, the operator can arrange to start additional authentication services to handle the volume of work, and then shut down these additional services when the peak has passed.

Data sources, instrumentation, and data-collection requirements

Security is an all-encompassing aspect of most distributed systems. The pertinent data is likely to be generated at multiple points throughout a system. You should consider adopting a Security Information and Event Management (SIEM) approach to gather the security-related information that results from events raised by the application, network equipment, servers, firewalls, antivirus software, and other intrusion-prevention elements.

Security monitoring can incorporate data from tools that are not part of your application. These tools can include utilities that identify port-scanning activities by external agencies, or network filters that detect attempts to gain unauthenticated access to your application and data.

In all cases, the gathered data must enable an administrator to determine the nature of any attack and take the appropriate countermeasures.

Analyzing security data

A feature of security monitoring is the variety of sources from which the data arises. The different formats and level of detail often require complex analysis of the captured data to tie it together into a coherent thread of information. Apart from the simplest of cases (such as detecting a large number of failed sign-ins, or repeated attempts to gain unauthorized access to critical resources), it might not be possible to perform any complex automated processing of security data. Instead, it might be preferable to write this data, time-stamped but otherwise in its original form, to a secure repository to allow for expert manual analysis.

SLA monitoring

Many commercial systems that support paying customers make guarantees about the performance of the system in the form of SLAs. Essentially, SLAs state that the system can handle a defined volume of work within an agreed time frame and without losing critical information. SLA monitoring is concerned with ensuring that the system can meet measurable SLAs.

NOTE

SLA monitoring is closely related to performance monitoring. But whereas performance monitoring is concerned with ensuring that the system functions *optimally*, SLA monitoring is governed by a contractual obligation that defines what *optimally* actually means.

SLAs are often defined in terms of:

- Overall system availability. For example, an organization might guarantee that the system will be available for 99.9 percent of the time. This equates to no more than 9 hours of downtime per year, or approximately 10 minutes a week.
- Operational throughput. This aspect is often expressed as one or more high-water marks, such as guaranteeing that the system can support up to 100,000 concurrent user requests or handle 10,000 concurrent business transactions.
- Operational response time. The system might also make guarantees for the rate at which requests are processed. An example is that 99 percent of all business transactions will finish within 2 seconds, and no single transaction will take longer than 10 seconds.

NOTE

Some contracts for commercial systems might also include SLAs for customer support. An example is that all help-desk requests will elicit a response within 5 minutes, and that 99 percent of all problems will be fully addressed within 1 working day. Effective [issue tracking](#) (described later in this section) is key to meeting SLAs such as these.

Requirements for SLA monitoring

At the highest level, an operator should be able to determine at a glance whether the system is meeting the agreed SLAs or not. And if not, the operator should be able to drill down and examine the underlying factors to determine the reasons for substandard performance.

Typical high-level indicators that can be depicted visually include:

- The percentage of service uptime.
- The application throughput (measured in terms of successful transactions and/or operations per second).
- The number of successful/failing application requests.
- The number of application and system faults, exceptions, and warnings.

All of these indicators should be capable of being filtered by a specified period of time.

A cloud application will likely comprise a number of subsystems and components. An operator should be able to select a high-level indicator and see how it's composed from the health of the underlying elements. For example, if the uptime of the overall system falls below an acceptable value, an operator should be able to zoom in and determine which elements are contributing to this failure.

NOTE

System uptime needs to be defined carefully. In a system that uses redundancy to ensure maximum availability, individual instances of elements might fail, but the system can remain functional. System uptime as presented by health monitoring should indicate the aggregate uptime of each element and not necessarily whether the system has actually halted.

Additionally, failures might be isolated. So even if a specific system is unavailable, the remainder of the system might remain available, although with decreased functionality. (In an e-commerce system, a failure in the system might prevent a customer from placing orders, but the customer might still be able to browse the product catalog.)

For alerting purposes, the system should be able to raise an event if any of the high-level indicators exceed a

specified threshold. The lower-level details of the various factors that compose the high-level indicator should be available as contextual data to the alerting system.

Data sources, instrumentation, and data-collection requirements

The raw data that's required to support SLA monitoring is similar to the raw data that's required for performance monitoring, together with some aspects of health and availability monitoring. (See those sections for more details.) You can capture this data by:

- Performing endpoint monitoring.
- Logging exceptions, faults, and warnings.
- Tracing the execution of user requests.
- Monitoring the availability of any third-party services that the system uses.
- Using performance metrics and counters.

All data must be timed and time-stamped.

Analyzing SLA data

The instrumentation data must be aggregated to generate a picture of the overall performance of the system. Aggregated data must also support drill-down to enable examination of the performance of the underlying subsystems. For example, you should be able to:

- Calculate the total number of user requests during a specified period and determine the success and failure rate of these requests.
- Combine the response times of user requests to generate an overall view of system response times.
- Analyze the progress of user requests to break down the overall response time of a request into the response times of the individual work items in that request.
- Determine the overall availability of the system as a percentage of uptime for any specific period.
- Analyze the percentage time availability of the individual components and services in the system. This might involve parsing logs that third-party services have generated.

Many commercial systems are required to report real performance figures against agreed SLAs for a specified period, typically a month. This information can be used to calculate credits or other forms of repayments for customers if the SLAs are not met during that period. You can calculate availability for a service by using the technique described in the section [Analyzing availability data](#).

For internal purposes, an organization might also track the number and nature of incidents that caused services to fail. Learning how to resolve these issues quickly, or eliminate them completely, will help to reduce downtime and meet SLAs.

Auditing

Depending on the nature of the application, there might be statutory or other legal regulations that specify requirements for auditing users' operations and recording all data access. Auditing can provide evidence that links customers to specific requests. Non-repudiation is an important factor in many e-business systems to help maintain trust between a customer and the organization that's responsible for the application or service.

Requirements for auditing

An analyst must be able to trace the sequence of business operations that users are performing so that you can reconstruct users' actions. This might be necessary simply as a matter of record, or as part of a forensic investigation.

Audit information is highly sensitive. It will likely include data that identifies the users of the system, together with the tasks that they're performing. For this reason, audit information will most likely take the form of reports that are available only to trusted analysts rather than as an interactive system that supports drill-down of graphical

operations. An analyst should be able to generate a range of reports. For example, reports might list all users' activities occurring during a specified time frame, detail the chronology of activity for a single user, or list the sequence of operations performed against one or more resources.

Data sources, instrumentation, and data-collection requirements

The primary sources of information for auditing can include:

- The security system that manages user authentication.
- Trace logs that record user activity.
- Security logs that track all identifiable and unidentifiable network requests.

The format of the audit data and the way in which it's stored might be driven by regulatory requirements. For example, it might not be possible to clean the data in any way. (It must be recorded in its original format.) Access to the repository where it's held must be protected to prevent tampering.

Analyzing audit data

An analyst must be able to access the raw data in its entirety, in its original form. Aside from the requirement to generate common audit reports, the tools for analyzing this data are likely to be specialized and kept external to the system.

Usage monitoring

Usage monitoring tracks how the features and components of an application are used. An operator can use the gathered data to:

- Determine which features are heavily used and determine any potential hotspots in the system. High-traffic elements might benefit from functional partitioning or even replication to spread the load more evenly. An operator can also use this information to ascertain which features are infrequently used and are possible candidates for retirement or replacement in a future version of the system.
- Obtain information about the operational events of the system under normal use. For example, in an e-commerce site, you can record the statistical information about the number of transactions and the volume of customers that are responsible for them. This information can be used for capacity planning as the number of customers grows.
- Detect (possibly indirectly) user satisfaction with the performance or functionality of the system. For example, if a large number of customers in an e-commerce system regularly abandon their shopping carts, this might be due to a problem with the checkout functionality.
- Generate billing information. A commercial application or multitenant service might charge customers for the resources that they use.
- Enforce quotas. If a user in a multitenant system exceeds their paid quota of processing time or resource usage during a specified period, their access can be limited or processing can be throttled.

Requirements for usage monitoring

To examine system usage, an operator typically needs to see information that includes:

- The number of requests that are processed by each subsystem and directed to each resource.
- The work that each user is performing.
- The volume of data storage that each user occupies.
- The resources that each user is accessing.

An operator should also be able to generate graphs. For example, a graph might display the most resource-hungry users, or the most frequently accessed resources or system features.

Data sources, instrumentation, and data-collection requirements

Usage tracking can be performed at a relatively high level. It can note the start and end times of each request and

the nature of the request (read, write, and so on, depending on the resource in question). You can obtain this information by:

- Tracing user activity.
- Capturing performance counters that measure the utilization for each resource.
- Monitoring the resource consumption by each user.

For metering purposes, you also need to be able to identify which users are responsible for performing which operations, and the resources that these operations utilize. The gathered information should be detailed enough to enable accurate billing.

Issue tracking

Customers and other users might report issues if unexpected events or behavior occurs in the system. Issue tracking is concerned with managing these issues, associating them with efforts to resolve any underlying problems in the system, and informing customers of possible resolutions.

Requirements for issue tracking

Operators often perform issue tracking by using a separate system that enables them to record and report the details of problems that users report. These details can include the tasks that the user was trying to perform, symptoms of the problem, the sequence of events, and any error or warning messages that were issued.

Data sources, instrumentation, and data-collection requirements

The initial data source for issue-tracking data is the user who reported the issue in the first place. The user might be able to provide additional data such as:

- A crash dump (if the application includes a component that runs on the user's desktop).
- A screen snapshot.
- The date and time when the error occurred, together with any other environmental information such as the user's location.

This information can be used to help the debugging effort and help construct a backlog for future releases of the software.

Analyzing issue-tracking data

Different users might report the same problem. The issue-tracking system should associate common reports.

The progress of the debugging effort should be recorded against each issue report. When the problem is resolved, the customer can be informed of the solution.

If a user reports an issue that has a known solution in the issue-tracking system, the operator should be able to inform the user of the solution immediately.

Tracing operations and debugging software releases

When a user reports an issue, the user is often only aware of the immediate impact that it has on their operations. The user can only report the results of their own experience back to an operator who is responsible for maintaining the system. These experiences are usually just a visible symptom of one or more fundamental problems. In many cases, an analyst will need to dig through the chronology of the underlying operations to establish the root cause of the problem. This process is called *root cause analysis*.

NOTE

Root cause analysis might uncover inefficiencies in the design of an application. In these situations, it might be possible to rework the affected elements and deploy them as part of a subsequent release. This process requires careful control, and the updated components should be monitored closely.

Requirements for tracing and debugging

For tracing unexpected events and other problems, it's vital that the monitoring data provides enough information to enable an analyst to trace back to the origins of these issues and reconstruct the sequence of events that occurred. This information must be sufficient to enable an analyst to diagnose the root cause of any problems. A developer can then make the necessary modifications to prevent them from recurring.

Data sources, instrumentation, and data-collection requirements

Troubleshooting can involve tracing all the methods (and their parameters) invoked as part of an operation to build up a tree that depicts the logical flow through the system when a customer makes a specific request. Exceptions and warnings that the system generates as a result of this flow need to be captured and logged.

To support debugging, the system can provide hooks that enable an operator to capture state information at crucial points in the system. Or, the system can deliver detailed step-by-step information as selected operations progress. Capturing data at this level of detail can impose an additional load on the system and should be a temporary process. An operator uses this process mainly when a highly unusual series of events occurs and is difficult to replicate, or when a new release of one or more elements into a system requires careful monitoring to ensure that the elements function as expected.

The monitoring and diagnostics pipeline

Monitoring a large-scale distributed system poses a significant challenge. Each of the scenarios described in the previous section should not necessarily be considered in isolation. There is likely to be a significant overlap in the monitoring and diagnostic data that's required for each situation, although this data might need to be processed and presented in different ways. For these reasons, you should take a holistic view of monitoring and diagnostics.

You can envisage the entire monitoring and diagnostics process as a pipeline that comprises the stages shown in Figure 1.

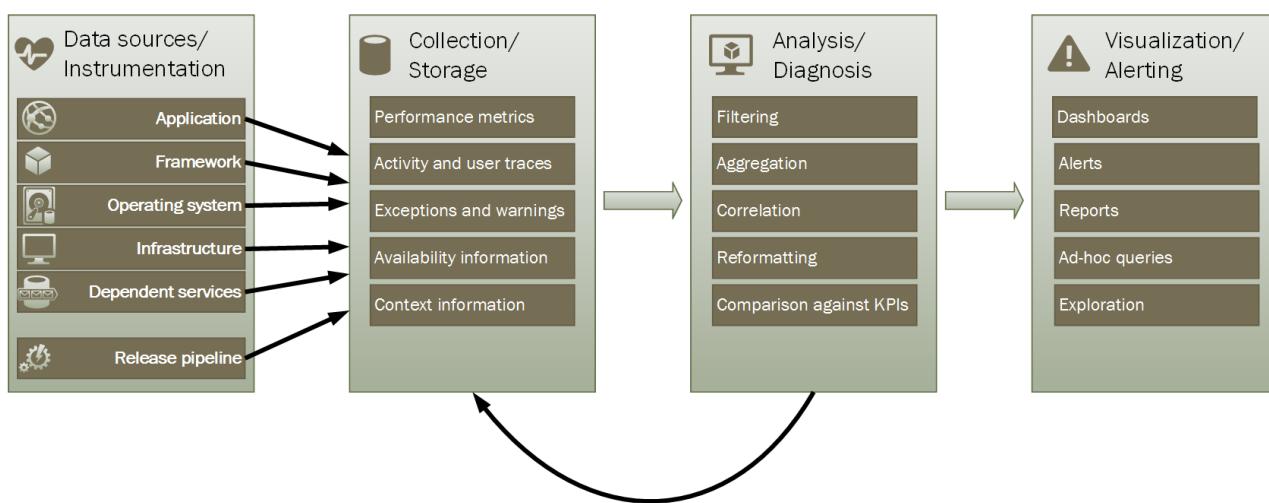


Figure 1. The stages in the monitoring and diagnostics pipeline

Figure 1 highlights how the data for monitoring and diagnostics can come from a variety of data sources. The instrumentation and collection stages are concerned with identifying the sources from where the data needs to be captured, determining which data to capture, how to capture it, and how to format this data so that it can be easily examined. The analysis/diagnosis stage takes the raw data and uses it to generate meaningful information that an

operator can use to determine the state of the system. The operator can use this information to make decisions about possible actions to take, and then feed the results back into the instrumentation and collection stages. The visualization/alerting stage phase presents a consumable view of the system state. It can display information in near real time by using a series of dashboards. And it can generate reports, graphs, and charts to provide a historical view of the data that can help identify long-term trends. If information indicates that a KPI is likely to exceed acceptable bounds, this stage can also trigger an alert to an operator. In some cases, an alert can also be used to trigger an automated process that attempts to take corrective actions, such as autoscaling.

Note that these steps constitute a continuous-flow process where the stages are happening in parallel. Ideally, all the phases should be dynamically configurable. At some points, especially when a system has been newly deployed or is experiencing problems, it might be necessary to gather extended data on a more frequent basis. At other times, it should be possible to revert to capturing a base level of essential information to verify that the system is functioning properly.

Additionally, the entire monitoring process should be considered a live, ongoing solution that's subject to fine-tuning and improvements as a result of feedback. For example, you might start with measuring many factors to determine system health. Analysis over time might lead to a refinement as you discard measures that aren't relevant, enabling you to more precisely focus on the data that you need while minimizing background noise.

Sources of monitoring and diagnostic data

The information that the monitoring process uses can come from several sources, as illustrated in Figure 1. At the application level, information comes from trace logs incorporated into the code of the system. Developers should follow a standard approach for tracking the flow of control through their code. For example, an entry to a method can emit a trace message that specifies the name of the method, the current time, the value of each parameter, and any other pertinent information. Recording the entry and exit times can also prove useful.

You should log all exceptions and warnings, and ensure that you retain a full trace of any nested exceptions and warnings. Ideally, you should also capture information that identifies the user who is running the code, together with activity correlation information (to track requests as they pass through the system). And you should log attempts to access all resources such as message queues, databases, files, and other dependent services. This information can be used for metering and auditing purposes.

Many applications make use of libraries and frameworks to perform common tasks such as accessing a data store or communicating over a network. These frameworks might be configurable to provide their own trace messages and raw diagnostic information, such as transaction rates and data transmission successes and failures.

NOTE

Many modern frameworks automatically publish performance and trace events. Capturing this information is simply a matter of providing a means to retrieve and store it where it can be processed and analyzed.

The operating system where the application is running can be a source of low-level system-wide information, such as performance counters that indicate I/O rates, memory utilization, and CPU usage. Operating system errors (such as the failure to open a file correctly) might also be reported.

You should also consider the underlying infrastructure and components on which your system runs. Virtual machines, virtual networks, and storage services can all be sources of important infrastructure-level performance counters and other diagnostic data.

If your application uses other external services, such as a web server or database management system, these services might publish their own trace information, logs, and performance counters. Examples include SQL Server Dynamic Management Views for tracking operations performed against a SQL Server database, and IIS trace logs for recording requests made to a web server.

As the components of a system are modified and new versions are deployed, it's important to be able to attribute issues, events, and metrics to each version. This information should be tied back to the release pipeline so that problems with a specific version of a component can be tracked quickly and rectified.

Security issues might occur at any point in the system. For example, a user might attempt to sign in with an invalid user ID or password. An authenticated user might try to obtain unauthorized access to a resource. Or a user might provide an invalid or outdated key to access encrypted information. Security-related information for successful and failing requests should always be logged.

The section [Instrumenting an application](#) contains more guidance on the information that you should capture. But you can use a variety of strategies to gather this information:

- **Application/system monitoring.** This strategy uses internal sources within the application, application frameworks, operating system, and infrastructure. The application code can generate its own monitoring data at notable points during the lifecycle of a client request. The application can include tracing statements that might be selectively enabled or disabled as circumstances dictate. It might also be possible to inject diagnostics dynamically by using a diagnostics framework. These frameworks typically provide plug-ins that can attach to various instrumentation points in your code and capture trace data at these points.

Additionally, your code and/or the underlying infrastructure might raise events at critical points. Monitoring agents that are configured to listen for these events can record the event information.

- **Real user monitoring.** This approach records the interactions between a user and the application and observes the flow of each request and response. This information can have a two-fold purpose: it can be used for metering usage by each user, and it can be used to determine whether users are receiving a suitable quality of service (for example, fast response times, low latency, and minimal errors). You can use the captured data to identify areas of concern where failures occur most often. You can also use the data to identify elements where the system slows down, possibly due to hotspots in the application or some other form of bottleneck. If you implement this approach carefully, it might be possible to reconstruct users' flows through the application for debugging and testing purposes.

IMPORTANT

You should consider the data that's captured by monitoring real users to be highly sensitive because it might include confidential material. If you save captured data, store it securely. If you want to use the data for performance monitoring or debugging purposes, strip out all personally identifiable information first.

- **Synthetic user monitoring.** In this approach, you write your own test client that simulates a user and performs a configurable but typical series of operations. You can track the performance of the test client to help determine the state of the system. You can also use multiple instances of the test client as part of a load-testing operation to establish how the system responds under stress, and what sort of monitoring output is generated under these conditions.

NOTE

You can implement real and synthetic user monitoring by including code that traces and times the execution of method calls and other critical parts of an application.

- **Profiling.** This approach is primarily targeted at monitoring and improving application performance. Rather than operating at the functional level of real and synthetic user monitoring, it captures lower-level information as the application runs. You can implement profiling by using periodic sampling of the execution state of an application (determining which piece of code that the application is running at a given point in time). You can also use instrumentation that inserts probes into the code at important junctures (such as the start and end of a method call) and records which methods were invoked, at what time, and

how long each call took. You can then analyze this data to determine which parts of the application might cause performance problems.

- **Endpoint monitoring.** This technique uses one or more diagnostic endpoints that the application exposes specifically to enable monitoring. An endpoint provides a pathway into the application code and can return information about the health of the system. Different endpoints can focus on various aspects of the functionality. You can write your own diagnostics client that sends periodic requests to these endpoints and assimilate the responses. For more information, see the [Health Endpoint Monitoring Pattern](#).

For maximum coverage, you should use a combination of these techniques.

Instrumenting an application

Instrumentation is a critical part of the monitoring process. You can make meaningful decisions about the performance and health of a system only if you first capture the data that enables you to make these decisions. The information that you gather by using instrumentation should be sufficient to enable you to assess performance, diagnose problems, and make decisions without requiring you to sign in to a remote production server to perform tracing (and debugging) manually. Instrumentation data typically comprises metrics and information that's written to trace logs.

The contents of a trace log can be the result of textual data that's written by the application or binary data that's created as the result of a trace event (if the application is using Event Tracing for Windows--ETW). They can also be generated from system logs that record events arising from parts of the infrastructure, such as a web server. Textual log messages are often designed to be human-readable, but they should also be written in a format that enables an automated system to parse them easily.

You should also categorize logs. Don't write all trace data to a single log, but use separate logs to record the trace output from different operational aspects of the system. You can then quickly filter log messages by reading from the appropriate log rather than having to process a single lengthy file. Never write information that has different security requirements (such as audit information and debugging data) to the same log.

NOTE

A log might be implemented as a file on the file system, or it might be held in some other format, such as a blob in blob storage. Log information might also be held in more structured storage, such as rows in a table.

Metrics will generally be a measure or count of some aspect or resource in the system at a specific time, with one or more associated tags or dimensions (sometimes called a *sample*). A single instance of a metric is usually not useful in isolation. Instead, metrics have to be captured over time. The key issue to consider is which metrics you should record and how frequently. Generating data for metrics too often can impose a significant additional load on the system, whereas capturing metrics infrequently might cause you to miss the circumstances that lead to a significant event. The considerations will vary from metric to metric. For example, CPU utilization on a server might vary significantly from second to second, but high utilization becomes an issue only if it's long-lived over a number of minutes.

Information for correlating data

You can easily monitor individual system-level performance counters, capture metrics for resources, and obtain application trace information from various log files. But some forms of monitoring require the analysis and diagnostics stage in the monitoring pipeline to correlate the data that's retrieved from several sources. This data might take several forms in the raw data, and the analysis process must be provided with sufficient instrumentation data to be able to map these different forms. For example, at the application framework level, a task might be identified by a thread ID. Within an application, the same work might be associated with the user ID for the user who is performing that task.

Also, there's unlikely to be a 1:1 mapping between threads and user requests, because asynchronous operations might reuse the same threads to perform operations on behalf of more than one user. To complicate matters further, a single request might be handled by more than one thread as execution flows through the system. If possible, associate each request with a unique activity ID that's propagated through the system as part of the request context. (The technique for generating and including activity IDs in trace information depends on the technology that's used to capture the trace data.)

All monitoring data should be time-stamped in the same way. For consistency, record all dates and times by using Coordinated Universal Time. This will help you more easily trace sequences of events.

NOTE

Computers operating in different time zones and networks might not be synchronized. Don't depend on using time stamps alone for correlating instrumentation data that spans multiple machines.

Information to include in the instrumentation data

Consider the following points when you're deciding which instrumentation data you need to collect:

- Make sure that information captured by trace events is machine and human readable. Adopt well-defined schemas for this information to facilitate automated processing of log data across systems, and to provide consistency to operations and engineering staff reading the logs. Include environmental information, such as the deployment environment, the machine on which the process is running, the details of the process, and the call stack.
- Enable profiling only when necessary because it can impose a significant overhead on the system. Profiling by using instrumentation records an event (such as a method call) every time it occurs, whereas sampling records only selected events. The selection can be time-based (once every n seconds), or frequency-based (once every n requests). If events occur very frequently, profiling by instrumentation might cause too much of a burden and itself affect overall performance. In this case, the sampling approach might be preferable. However, if the frequency of events is low, sampling might miss them. In this case, instrumentation might be the better approach.
- Provide sufficient context to enable a developer or administrator to determine the source of each request. This might include some form of activity ID that identifies a specific instance of a request. It might also include information that can be used to correlate this activity with the computational work performed and the resources used. Note that this work might cross process and machine boundaries. For metering, the context should also include (either directly or indirectly via other correlated information) a reference to the customer who caused the request to be made. This context provides valuable information about the application state at the time that the monitoring data was captured.
- Record all requests, and the locations or regions from which these requests are made. This information can assist in determining whether there are any location-specific hotspots. This information can also be useful in determining whether to repartition an application or the data that it uses.
- Record and capture the details of exceptions carefully. Often, critical debug information is lost as a result of poor exception handling. Capture the full details of exceptions that the application throws, including any inner exceptions and other context information. Include the call stack if possible.
- Be consistent in the data that the different elements of your application capture, because this can assist in analyzing events and correlating them with user requests. Consider using a comprehensive and configurable logging package to gather information, rather than depending on developers to adopt the same approach as they implement different parts of the system. Gather data from key performance counters, such as the volume of I/O being performed, network utilization, number of requests, memory use, and CPU utilization. Some infrastructure services might provide their own specific performance counters, such as the number of connections to a database, the rate at which transactions are being performed, and the number of transactions that succeed or fail. Applications might also define their own specific performance counters.
- Log all calls made to external services, such as database systems, web services, or other system-level services

that are part of the infrastructure. Record information about the time taken to perform each call, and the success or failure of the call. If possible, capture information about all retry attempts and failures for any transient errors that occur.

Ensuring compatibility with telemetry systems

In many cases, the information that instrumentation produces is generated as a series of events and passed to a separate telemetry system for processing and analysis. A telemetry system is typically independent of any specific application or technology, but it expects information to follow a specific format that's usually defined by a schema. The schema effectively specifies a contract that defines the data fields and types that the telemetry system can ingest. The schema should be generalized to allow for data arriving from a range of platforms and devices.

A common schema should include fields that are common to all instrumentation events, such as the event name, the event time, the IP address of the sender, and the details that are required for correlating with other events (such as a user ID, a device ID, and an application ID). Remember that any number of devices might raise events, so the schema should not depend on the device type. Additionally, various devices might raise events for the same application; the application might support roaming or some other form of cross-device distribution.

The schema might also include domain fields that are relevant to a particular scenario that's common across different applications. This might be information about exceptions, application start and end events, and success and/or failure of web service API calls. All applications that use the same set of domain fields should emit the same set of events, enabling a set of common reports and analytics to be built.

Finally, a schema might contain custom fields for capturing the details of application-specific events.

Best practices for instrumenting applications

The following list summarizes best practices for instrumenting a distributed application running in the cloud.

- Make logs easy to read and easy to parse. Use structured logging where possible. Be concise and descriptive in log messages.
- In all logs, identify the source and provide context and timing information as each log record is written.
- Use the same time zone and format for all time stamps. This will help to correlate events for operations that span hardware and services running in different geographic regions.
- Categorize logs and write messages to the appropriate log file.
- Do not disclose sensitive information about the system or personal information about users. Scrub this information before it's logged, but ensure that the relevant details are retained. For example, remove the ID and password from any database connection strings, but write the remaining information to the log so that an analyst can determine that the system is accessing the correct database. Log all critical exceptions, but enable the administrator to turn logging on and off for lower levels of exceptions and warnings. Also, capture and log all retry logic information. This data can be useful in monitoring the transient health of the system.
- Trace out of process calls, such as requests to external web services or databases.
- Don't mix log messages with different security requirements in the same log file. For example, don't write debug and audit information to the same log.
- With the exception of auditing events, make sure that all logging calls are fire-and-forget operations that do not block the progress of business operations. Auditing events are exceptional because they are critical to the business and can be classified as a fundamental part of business operations.
- Make sure that logging is extensible and does not have any direct dependencies on a concrete target. For example, rather than writing information by using *System.Diagnostics.Trace*, define an abstract interface (such as *ILogger*) that exposes logging methods and that can be implemented through any appropriate means.
- Make sure that all logging is fail-safe and never triggers any cascading errors. Logging must not throw any exceptions.
- Treat instrumentation as an ongoing iterative process and review logs regularly, not just when there is a problem.

Collecting and storing data

The collection stage of the monitoring process is concerned with retrieving the information that instrumentation generates, formatting this data to make it easier for the analysis/diagnosis stage to consume, and saving the transformed data in reliable storage. The instrumentation data that you gather from different parts of a distributed system can be held in a variety of locations and with varying formats. For example, your application code might generate trace log files and generate application event log data, whereas performance counters that monitor key aspects of the infrastructure that your application uses can be captured through other technologies. Any third-party components and services that your application uses might provide instrumentation information in different formats, by using separate trace files, blob storage, or even a custom data store.

Data collection is often performed through a collection service that can run autonomously from the application that generates the instrumentation data. Figure 2 illustrates an example of this architecture, highlighting the instrumentation data-collection subsystem.

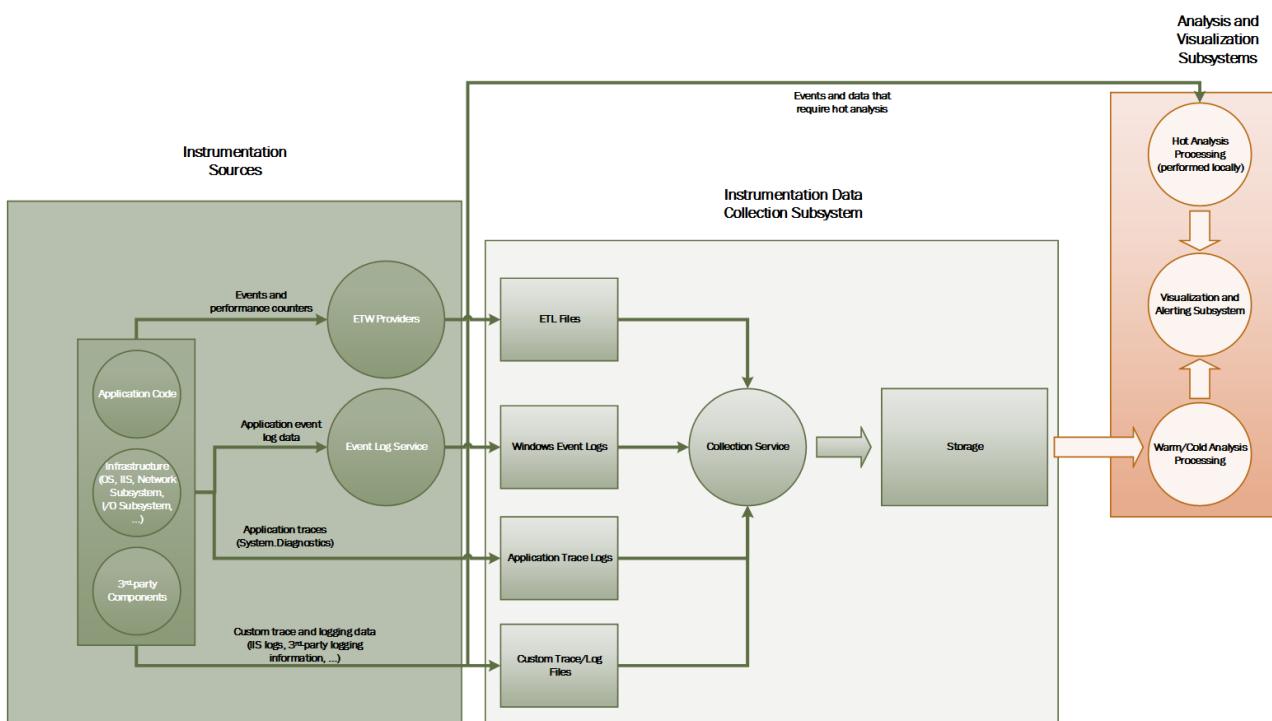


Figure 2. Collecting instrumentation data

Note that this is a simplified view. The collection service is not necessarily a single process and might comprise many constituent parts running on different machines, as described in the following sections. Additionally, if the analysis of some telemetry data must be performed quickly (hot analysis, as described in the section [Supporting hot, warm, and cold analysis](#) later in this document), local components that operate outside the collection service might perform the analysis tasks immediately. Figure 2 depicts this situation for selected events. After analytical processing, the results can be sent directly to the visualization and alerting subsystem. Data that's subjected to warm or cold analysis is held in storage while it awaits processing.

For Azure applications and services, Azure Diagnostics provides one possible solution for capturing data. Azure Diagnostics gathers data from the following sources for each compute node, aggregates it, and then uploads it to Azure Storage:

- IIS logs
- IIS Failed Request logs
- Windows event logs
- Performance counters
- Crash dumps
- Azure Diagnostics infrastructure logs

- Custom error logs
- .NET EventSource
- Manifest-based ETW

For more information, see the article [Azure: Telemetry Basics and Troubleshooting](#).

Strategies for collecting instrumentation data

Considering the elastic nature of the cloud, and to avoid the necessity of manually retrieving telemetry data from every node in the system, you should arrange for the data to be transferred to a central location and consolidated. In a system that spans multiple datacenters, it might be useful to first collect, consolidate, and store data on a region-by-region basis, and then aggregate the regional data into a single central system.

To optimize the use of bandwidth, you can elect to transfer less urgent data in chunks, as batches. However, the data must not be delayed indefinitely, especially if it contains time-sensitive information.

Pulling and pushing instrumentation data

The instrumentation data-collection subsystem can actively retrieve instrumentation data from the various logs and other sources for each instance of the application (the *pull model*). Or, it can act as a passive receiver that waits for the data to be sent from the components that constitute each instance of the application (the *push model*).

One approach to implementing the pull model is to use monitoring agents that run locally with each instance of the application. A monitoring agent is a separate process that periodically retrieves (pulls) telemetry data collected at the local node and writes this information directly to centralized storage that all instances of the application share. This is the mechanism that Azure Diagnostics implements. Each instance of an Azure web or worker role can be configured to capture diagnostic and other trace information that's stored locally. The monitoring agent that runs alongside each instance copies the specified data to Azure Storage. The article [Enabling Diagnostics in Azure Cloud Services and Virtual Machines](#) provides more details on this process. Some elements, such as IIS logs, crash dumps, and custom error logs, are written to blob storage. Data from the Windows event log, ETW events, and performance counters is recorded in table storage. Figure 3 illustrates this mechanism.

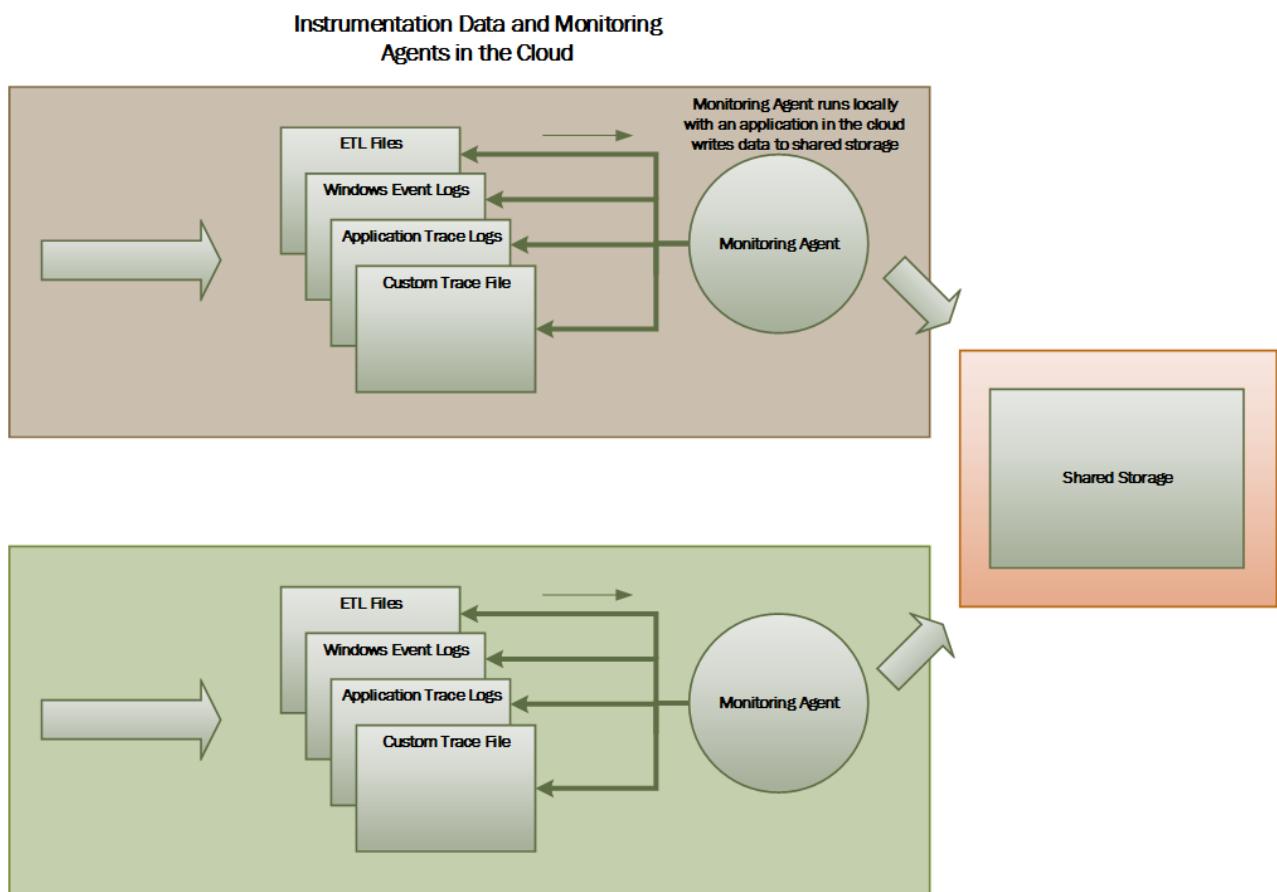


Figure 3. Using a monitoring agent to pull information and write to shared storage

NOTE

Using a monitoring agent is ideally suited to capturing instrumentation data that's naturally pulled from a data source. An example is information from SQL Server Dynamic Management Views or the length of an Azure Service Bus queue.

It's feasible to use the approach just described to store telemetry data for a small-scale application running on a limited number of nodes in a single location. However, a complex, highly scalable, global cloud application might generate huge volumes of data from hundreds of web and worker roles, database shards, and other services. This flood of data can easily overwhelm the I/O bandwidth available with a single, central location. Therefore, your telemetry solution must be scalable to prevent it from acting as a bottleneck as the system expands. Ideally, your solution should incorporate a degree of redundancy to reduce the risks of losing important monitoring information (such as auditing or billing data) if part of the system fails.

To address these issues, you can implement queuing, as shown in Figure 4. In this architecture, the local monitoring agent (if it can be configured appropriately) or custom data-collection service (if not) posts data to a queue. A separate process running asynchronously (the storage writing service in Figure 4) takes the data in this queue and writes it to shared storage. A message queue is suitable for this scenario because it provides "at least once" semantics that help ensure that queued data will not be lost after it's posted. You can implement the storage writing service by using a separate worker role.

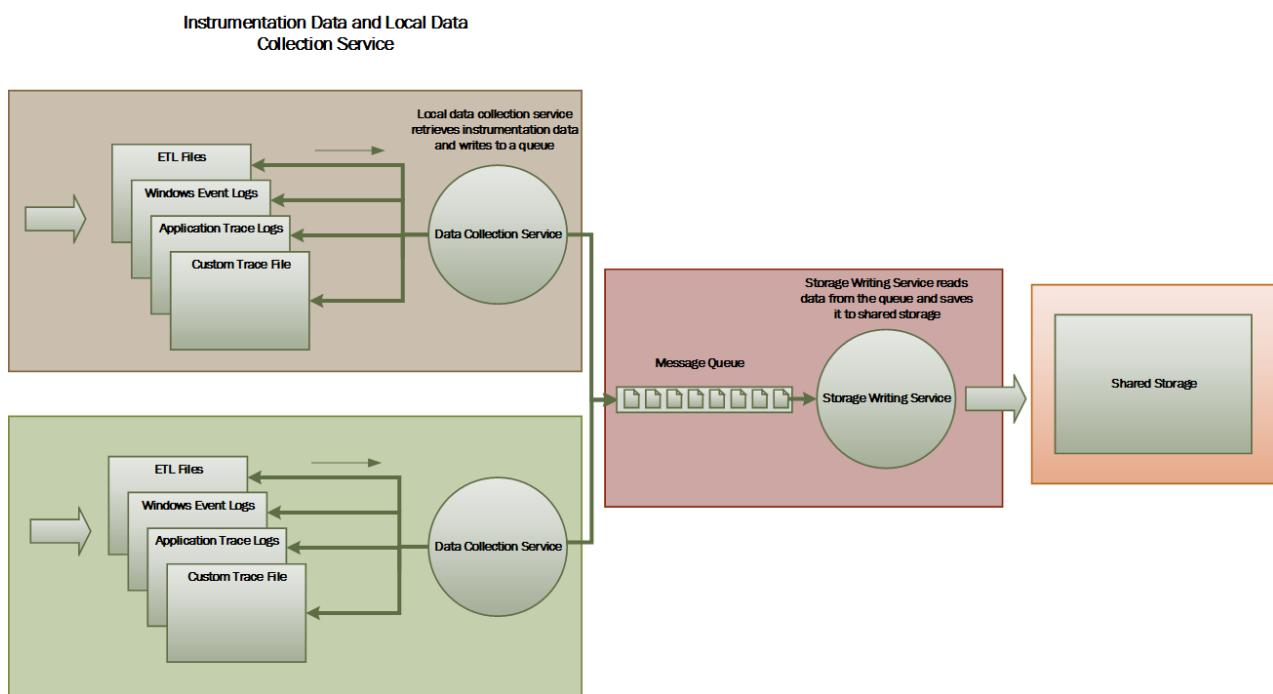


Figure 4. Using a queue to buffer instrumentation data

The local data-collection service can add data to a queue immediately after it's received. The queue acts as a buffer, and the storage writing service can retrieve and write the data at its own pace. By default, a queue operates on a first-in, first-out basis. But you can prioritize messages to accelerate them through the queue if they contain data that must be handled more quickly. For more information, see the [Priority Queue](#) pattern. Alternatively, you can use different channels (such as Service Bus topics) to direct data to different destinations depending on the form of analytical processing that's required.

For scalability, you can run multiple instances of the storage writing service. If there is a high volume of events, you can use an event hub to dispatch the data to different compute resources for processing and storage.

Consolidating instrumentation data

The instrumentation data that the data-collection service retrieves from a single instance of an application gives a localized view of the health and performance of that instance. To assess the overall health of the system, it's necessary to consolidate some aspects of the data in the local views. You can perform this after the data has been

stored, but in some cases, you can also achieve it as the data is collected. Rather than being written directly to shared storage, the instrumentation data can pass through a separate data consolidation service that combines data and acts as a filter and cleanup process. For example, instrumentation data that includes the same correlation information such as an activity ID can be amalgamated. (It's possible that a user starts performing a business operation on one node and then gets transferred to another node in the event of node failure, or depending on how load balancing is configured.) This process can also detect and remove any duplicated data (always a possibility if the telemetry service uses message queues to push instrumentation data out to storage). Figure 5 illustrates an example of this structure.

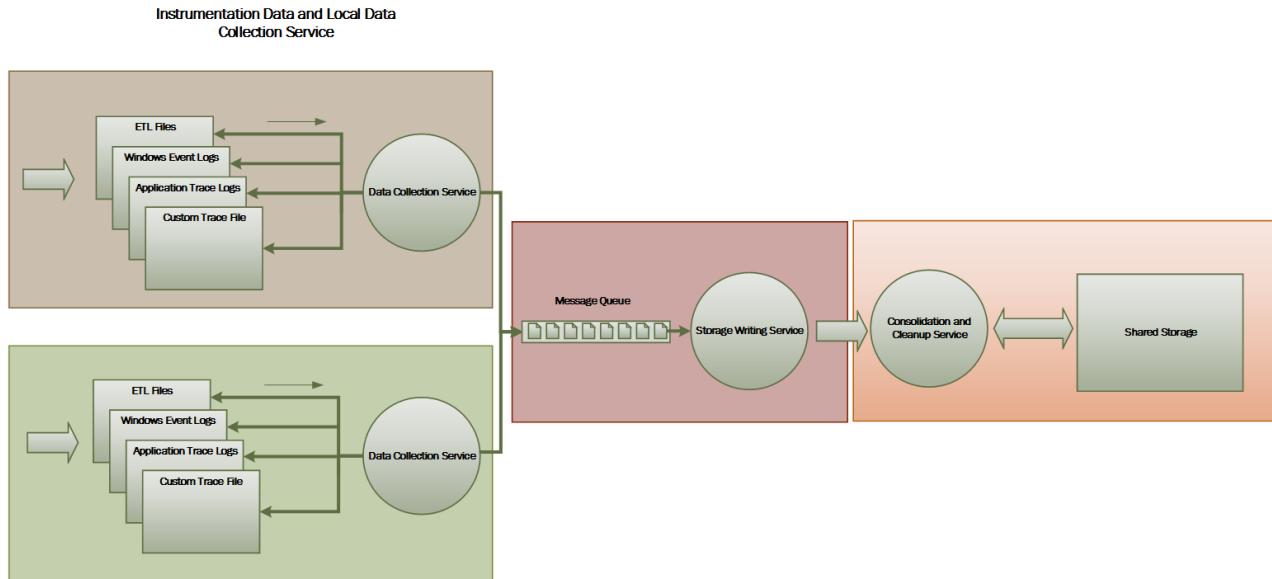


Figure 5. Using a separate service to consolidate and clean up instrumentation data

Storing instrumentation data

The previous discussions have depicted a rather simplistic view of the way in which instrumentation data is stored. In reality, it can make sense to store the different types of information by using technologies that are most appropriate to the way in which each type is likely to be used.

For example, Azure blob and table storage have some similarities in the way in which they're accessed. But they have limitations in the operations that you can perform by using them, and the granularity of the data that they hold is quite different. If you need to perform more analytical operations or require full-text search capabilities on the data, it might be more appropriate to use data storage that provides capabilities that are optimized for specific types of queries and data access. For example:

- Performance counter data can be stored in a SQL database to enable ad hoc analysis.
- Trace logs might be better stored in Azure Cosmos DB.
- Security information can be written to HDFS.
- Information that requires full-text search can be stored through Elasticsearch (which can also speed searches by using rich indexing).

You can implement an additional service that periodically retrieves the data from shared storage, partitions and filters the data according to its purpose, and then writes it to an appropriate set of data stores as shown in Figure 6. An alternative approach is to include this functionality in the consolidation and cleanup process and write the data directly to these stores as it's retrieved rather than saving it in an intermediate shared storage area. Each approach has its advantages and disadvantages. Implementing a separate partitioning service lessens the load on the consolidation and cleanup service, and it enables at least some of the partitioned data to be regenerated if necessary (depending on how much data is retained in shared storage). However, it consumes additional resources. Also, there might be a delay between the receipt of instrumentation data from each application instance and the conversion of this data into actionable information.

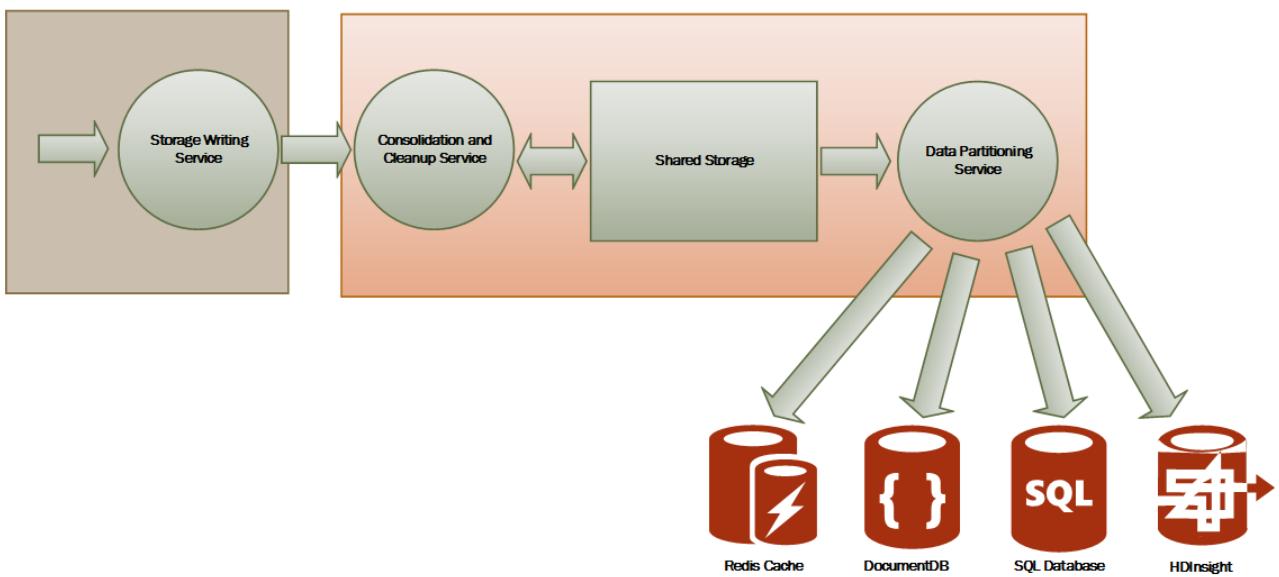


Figure 6. Partitioning data according to analytical and storage requirements

The same instrumentation data might be required for more than one purpose. For example, performance counters can be used to provide a historical view of system performance over time. This information might be combined with other usage data to generate customer billing information. In these situations, the same data might be sent to more than one destination, such as a document database that can act as a long-term store for holding billing information, and a multidimensional store for handling complex performance analytics.

You should also consider how urgently the data is required. Data that provides information for alerting must be accessed quickly, so it should be held in fast data storage and indexed or structured to optimize the queries that the alerting system performs. In some cases, it might be necessary for the telemetry service that gathers the data on each node to format and save data locally so that a local instance of the alerting system can quickly notify you of any issues. The same data can be dispatched to the storage writing service shown in the previous diagrams and stored centrally if it's also required for other purposes.

Information that's used for more considered analysis, for reporting, and for spotting historical trends is less urgent and can be stored in a manner that supports data mining and ad hoc queries. For more information, see the section [Supporting hot, warm, and cold analysis](#) later in this document.

Log rotation and data retention

Instrumentation can generate considerable volumes of data. This data can be held in several places, starting with the raw log files, trace files, and other information captured at each node to the consolidated, cleaned, and partitioned view of this data held in shared storage. In some cases, after the data has been processed and transferred, the original raw source data can be removed from each node. In other cases, it might be necessary or simply useful to save the raw information. For example, data that's generated for debugging purposes might be best left available in its raw form but can then be discarded quickly after any bugs have been rectified.

Performance data often has a longer life so that it can be used for spotting performance trends and for capacity planning. The consolidated view of this data is usually kept online for a finite period to enable fast access. After that, it can be archived or discarded. Data gathered for metering and billing customers might need to be saved indefinitely. Additionally, regulatory requirements might dictate that information collected for auditing and security purposes also needs to be archived and saved. This data is also sensitive and might need to be encrypted or otherwise protected to prevent tampering. You should never record users' passwords or other information that might be used to commit identity fraud. Such details should be scrubbed from the data before it's stored.

Down-sampling

It's useful to store historical data so you can spot long-term trends. Rather than saving old data in its entirety, it might be possible to down-sample the data to reduce its resolution and save storage costs. As an example, rather than saving minute-by-minute performance indicators, you can consolidate data that's more than a month old to form an hour-by-hour view.

Best practices for collecting and storing logging information

The following list summarizes best practices for capturing and storing logging information:

- The monitoring agent or data-collection service should run as an out-of-process service and should be simple to deploy.
- All output from the monitoring agent or data-collection service should be an agnostic format that's independent of the machine, operating system, or network protocol. For example, emit information in a self-describing format such as JSON, MessagePack, or Protobuf rather than ETL/ETW. Using a standard format enables the system to construct processing pipelines; components that read, transform, and send data in the agreed format can be easily integrated.
- The monitoring and data-collection process must be fail-safe and must not trigger any cascading error conditions.
- In the event of a transient failure in sending information to a data sink, the monitoring agent or data-collection service should be prepared to reorder telemetry data so that the newest information is sent first. (The monitoring agent/data-collection service might elect to drop the older data, or save it locally and transmit it later to catch up, at its own discretion.)

Analyzing data and diagnosing issues

An important part of the monitoring and diagnostics process is analyzing the gathered data to obtain a picture of the overall well-being of the system. You should have defined your own KPIs and performance metrics, and it's important to understand how you can structure the data that has been gathered to meet your analysis requirements. It's also important to understand how the data that's captured in different metrics and log files is correlated, because this information can be key to tracking a sequence of events and help diagnose problems that arise.

As described in the section [Consolidating instrumentation data](#), the data for each part of the system is typically captured locally, but it generally needs to be combined with data generated at other sites that participate in the system. This information requires careful correlation to ensure that data is combined accurately. For example, the usage data for an operation might span a node that hosts a website to which a user connects, a node that runs a separate service accessed as part of this operation, and data storage held on another node. This information needs to be tied together to provide an overall view of the resource and processing usage for the operation. Some pre-processing and filtering of data might occur on the node on which the data is captured, whereas aggregation and formatting are more likely to occur on a central node.

Supporting hot, warm, and cold analysis

Analyzing and reformatting data for visualization, reporting, and alerting purposes can be a complex process that consumes its own set of resources. Some forms of monitoring are time-critical and require immediate analysis of data to be effective. This is known as *hot analysis*. Examples include the analyses that are required for alerting and some aspects of security monitoring (such as detecting an attack on the system). Data that's required for these purposes must be quickly available and structured for efficient processing. In some cases, it might be necessary to move the analysis processing to the individual nodes where the data is held.

Other forms of analysis are less time-critical and might require some computation and aggregation after the raw data has been received. This is called *warm analysis*. Performance analysis often falls into this category. In this case, an isolated, single performance event is unlikely to be statistically significant. (It might be caused by a sudden spike or glitch.) The data from a series of events should provide a more reliable picture of system performance.

Warm analysis can also be used to help diagnose health issues. A health event is typically processed through hot analysis and can raise an alert immediately. An operator should be able to drill into the reasons for the health event by examining the data from the warm path. This data should contain information about the events leading up to the issue that caused the health event.

Some types of monitoring generate more long-term data. This analysis can be performed at a later date, possibly

according to a predefined schedule. In some cases, the analysis might need to perform complex filtering of large volumes of data captured over a period of time. This is called *cold analysis*. The key requirement is that the data is stored safely after it has been captured. For example, usage monitoring and auditing require an accurate picture of the state of the system at regular points in time, but this state information does not have to be available for processing immediately after it has been gathered.

An operator can also use cold analysis to provide the data for predictive health analysis. The operator can gather historical information over a specified period and use it in conjunction with the current health data (retrieved from the hot path) to spot trends that might soon cause health issues. In these cases, it might be necessary to raise an alert so that corrective action can be taken.

Correlating data

The data that instrumentation captures can provide a snapshot of the system state, but the purpose of analysis is to make this data actionable. For example:

- What has caused an intense I/O loading at the system level at a specific time?
- Is it the result of a large number of database operations?
- Is this reflected in the database response times, the number of transactions per second, and application response times at the same juncture?

If so, one remedial action that might reduce the load might be to shard the data over more servers. In addition, exceptions can arise as a result of a fault in any level of the system. An exception in one level often triggers another fault in the level above.

For these reasons, you need to be able to correlate the different types of monitoring data at each level to produce an overall view of the state of the system and the applications that are running on it. You can then use this information to make decisions about whether the system is functioning acceptably or not, and determine what can be done to improve the quality of the system.

As described in the section [Information for correlating data](#), you must ensure that the raw instrumentation data includes sufficient context and activity ID information to support the required aggregations for correlating events. Additionally, this data might be held in different formats, and it might be necessary to parse this information to convert it into a standardized format for analysis.

Troubleshooting and diagnosing issues

Diagnosis requires the ability to determine the cause of faults or unexpected behavior, including performing root cause analysis. The information that's required typically includes:

- Detailed information from event logs and traces, either for the entire system or for a specified subsystem during a specified time window.
- Complete stack traces resulting from exceptions and faults of any specified level that occur within the system or a specified subsystem during a specified period.
- Crash dumps for any failed processes either anywhere in the system or for a specified subsystem during a specified time window.
- Activity logs recording the operations that are performed either by all users or for selected users during a specified period.

Analyzing data for troubleshooting purposes often requires a deep technical understanding of the system architecture and the various components that compose the solution. As a result, a large degree of manual intervention is often required to interpret the data, establish the cause of problems, and recommend an appropriate strategy to correct them. It might be appropriate simply to store a copy of this information in its original format and make it available for cold analysis by an expert.

Visualizing data and raising alerts

An important aspect of any monitoring system is the ability to present the data in such a way that an operator can quickly spot any trends or problems. Also important is the ability to quickly inform an operator if a significant event has occurred that might require attention.

Data presentation can take several forms, including visualization by using dashboards, alerting, and reporting.

Visualization by using dashboards

The most common way to visualize data is to use dashboards that can display information as a series of charts, graphs, or some other illustration. These items can be parameterized, and an analyst should be able to select the important parameters (such as the time period) for any specific situation.

Dashboards can be organized hierarchically. Top-level dashboards can give an overall view of each aspect of the system but enable an operator to drill down to the details. For example, a dashboard that depicts the overall disk I/O for the system should allow an analyst to view the I/O rates for each individual disk to ascertain whether one or more specific devices account for a disproportionate volume of traffic. Ideally, the dashboard should also display related information, such as the source of each request (the user or activity) that's generating this I/O. This information can then be used to determine whether (and how) to spread the load more evenly across devices, and whether the system would perform better if more devices were added.

A dashboard might also use color-coding or some other visual cues to indicate values that appear anomalous or that are outside an expected range. Using the previous example:

- A disk with an I/O rate that's approaching its maximum capacity over an extended period (a hot disk) can be highlighted in red.
- A disk with an I/O rate that periodically runs at its maximum limit over short periods (a warm disk) can be highlighted in yellow.
- A disk that's exhibiting normal usage can be displayed in green.

Note that for a dashboard system to work effectively, it must have the raw data to work with. If you are building your own dashboard system, or using a dashboard developed by another organization, you must understand which instrumentation data you need to collect, at what levels of granularity, and how it should be formatted for the dashboard to consume.

A good dashboard does not only display information, it also enables an analyst to pose ad hoc questions about that information. Some systems provide management tools that an operator can use to perform these tasks and explore the underlying data. Alternatively, depending on the repository that's used to hold this information, it might be possible to query this data directly, or import it into tools such as Microsoft Excel for further analysis and reporting.

NOTE

You should restrict access to dashboards to authorized personnel, because this information might be commercially sensitive. You should also protect the underlying data for dashboards to prevent users from changing it.

Raising alerts

Alerting is the process of analyzing the monitoring and instrumentation data and generating a notification if a significant event is detected.

Alerting helps ensure that the system remains healthy, responsive, and secure. It's an important part of any system that makes performance, availability, and privacy guarantees to the users where the data might need to be acted on immediately. An operator might need to be notified of the event that triggered the alert. Alerting can also be used to invoke system functions such as autoscaling.

Alerting usually depends on the following instrumentation data:

- Security events. If the event logs indicate that repeated authentication and/or authorization failures are

occurring, the system might be under attack and an operator should be informed.

- Performance metrics. The system must quickly respond if a particular performance metric exceeds a specified threshold.
- Availability information. If a fault is detected, it might be necessary to quickly restart one or more subsystems, or fail over to a backup resource. Repeated faults in a subsystem might indicate more serious concerns.

Operators might receive alert information by using many delivery channels such as email, a pager device, or an SMS text message. An alert might also include an indication of how critical a situation is. Many alerting systems support subscriber groups, and all operators who are members of the same group can receive the same set of alerts.

An alerting system should be customizable, and the appropriate values from the underlying instrumentation data can be provided as parameters. This approach enables an operator to filter data and focus on those thresholds or combinations of values that are of interest. Note that in some cases, the raw instrumentation data can be provided to the alerting system. In other situations, it might be more appropriate to supply aggregated data. (For example, an alert can be triggered if the CPU utilization for a node has exceeded 90 percent over the last 10 minutes). The details provided to the alerting system should also include any appropriate summary and context information. This data can help reduce the possibility that false-positive events will trip an alert.

Reporting

Reporting is used to generate an overall view of the system. It might incorporate historical data in addition to current information. Reporting requirements themselves fall into two broad categories: operational reporting and security reporting.

Operational reporting typically includes the following aspects:

- Aggregating statistics that you can use to understand resource utilization of the overall system or specified subsystems during a specified time window
- Identifying trends in resource usage for the overall system or specified subsystems during a specified period
- Monitoring the exceptions that have occurred throughout the system or in specified subsystems during a specified period
- Determining the efficiency of the application in terms of the deployed resources, and understanding whether the volume of resources (and their associated cost) can be reduced without affecting performance unnecessarily

Security reporting is concerned with tracking customers' use of the system. It can include:

- Auditing user operations. This requires recording the individual requests that each user performs, together with dates and times. The data should be structured to enable an administrator to quickly reconstruct the sequence of operations that a user performs over a specified period.
- Tracking resource use by user. This requires recording how each request for a user accesses the various resources that compose the system, and for how long. An administrator must be able to use this data to generate a utilization report by user over a specified period, possibly for billing purposes.

In many cases, batch processes can generate reports according to a defined schedule. (Latency is not normally an issue.) But they should also be available for generation on an ad hoc basis if needed. As an example, if you are storing data in a relational database such as Azure SQL Database, you can use a tool such as SQL Server Reporting Services to extract and format data and present it as a set of reports.

Related patterns and guidance

- [Autoscaling guidance](#) describes how to decrease management overhead by reducing the need for an operator to continually monitor the performance of a system and make decisions about adding or removing resources.
- [Health Endpoint Monitoring Pattern](#) describes how to implement functional checks within an application that external tools can access through exposed endpoints at regular intervals.

- [Priority Queue Pattern](#) shows how to prioritize queued messages so that urgent requests are received and can be processed before less urgent messages.

More information

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

Naming conventions

6/19/2018 • 9 minutes to read • [Edit Online](#)

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

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

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

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

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

Naming subscriptions

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

A recommended pattern for naming subscriptions is:

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

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

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

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

For more information on how to organize subscriptions for larger enterprises, read our [prescriptive subscription governance guidance](#).

Use affixes to avoid ambiguity

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

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

- SvcCalculationEngine (prefix)
- CalculationEngineSvc (suffix)

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

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

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

Naming rules and restrictions

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

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

characters will cause most validation rules to fail.

General

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

Compute

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

NOTE

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

Storage

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Storage account name (data)	Global	3-24	Lowercase	Alphanumeric	<globally unique name><number> (use a function to calculate a unique guid for naming storage accounts)	profxdata001
Storage account name (disks)	Global	3-24	Lowercase	Alphanumeric	<vm name without hyphens>st<number>	profxsql001st0

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
--------	-------	--------	--------	------------------	-------------------	---------

Container name	Storage account	3-63	Lowercase	Alphanumeric and hyphen	<context>	logs
Blob name	Container	1-1024	Case sensitive	Any URL characters	<variable based on blob usage>	<variable based on blob usage>
Queue name	Storage account	3-63	Lowercase	Alphanumeric and hyphen	<service short name>-<context>-<num>	awesomeservice-messages-001
Table name	Storage account	3-63	Case insensitive	Alphanumeric	<service short name><context>	awesomeservicelogs
File name	Storage account	3-63	Lowercase	Alphanumeric	<variable based on blob usage>	<variable based on blob usage>
Data Lake Store	Global	3-24	Lowercase	Alphanumeric	<name>dls	telemetrydls

Networking

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Virtual Network (VNet)	Resource Group	2-64	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service short name>-vnet	profx-vnet
Subnet	Parent VNet	2-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<descriptive context>	web
Network Interface	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<vmname>-nic<num>	profx-sql1-nic1
Network Security Group	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service short name>-<context>-nsg	profx-app-nsg
Network Security Group Rule	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<descriptive context>	sql-allow

ENTITY	SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Public IP Address	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<vm or service name>-pip	profx-sql1-pip
Load Balancer	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service or role>-lb	profx-lb
Load Balanced Rules Config	Load Balancer	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<descriptive context>	http
Azure Application Gateway	Resource Group	1-80	Case insensitive	Alphanumeric, hyphen, underscore, and period	<service or role>-agw	profx-agw
Traffic Manager Profile	Resource Group	1-63	Case insensitive	Alphanumeric, hyphen, and period	<descriptive context>	app1

Organize resources with tags

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

TIP

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

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

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

Some of the common tagging use cases are:

- **Billing**; Grouping resources and associating them with billing or charge back codes.
- **Service Context Identification**; Identify groups of resources across Resource Groups for common operations and grouping
- **Access Control and Security Context**; Administrative role identification based on portfolio, system, service, app, instance, etc.

TIP

Tag early - tag often. Better to have a baseline tagging scheme in place and adjust over time rather than having to retrofit after the fact.

An example of some common tagging approaches:

TAG NAME	KEY	EXAMPLE	COMMENT
Bill To / Internal Chargeback ID	billTo	IT-Chargeback-1234	An internal I/O or billing code
Operator or Directly Responsible Individual (DRI)	managedBy	joe@contoso.com	Alias or email address
Project Name	projectName	myproject	Name of the project or product line
Project Version	projectVersion	3.4	Version of the project or product line
Environment	environment	<Production, Staging, QA >	Environmental identifier
Tier	tier	Front End, Back End, Data	Tier or role/context identification
Data Profile	dataProfile	Public, Confidential, Restricted, Internal	Sensitivity of data stored in the resource

Tips and tricks

Some types of resources may require additional care on naming and conventions.

Virtual machines

Especially in larger topologies, carefully naming virtual machines streamlines identifying the role and purpose of each machine, and enabling more predictable scripting.

Storage accounts and storage entities

There are two primary use cases for storage accounts - backing disks for VMs, and storing data in blobs, queues and tables. Storage accounts used for VM disks should follow the naming convention of associating them with the parent VM name (and with the potential need for multiple storage accounts for high-end VM SKUs, also apply a number suffix).

TIP

Storage accounts - whether for data or disks - should follow a naming convention that allows for multiple storage accounts to be leveraged (i.e. always using a numeric suffix).

It's possible to configure a custom domain name for accessing blob data in your Azure Storage account. The default endpoint for the Blob service is <https://<name>.blob.core.windows.net>.

But if you map a custom domain (such as www.contoso.com) to the blob endpoint for your storage account, you can also access blob data in your storage account by using that domain. For example, with a custom domain name,

`http://mystorage.blob.core.windows.net/mycontainer/myblob` could be accessed as
`http://www.contoso.com/mycontainer/myblob`.

For more information about configuring this feature, refer to [Configure a custom domain name for your Blob storage endpoint](#).

For more information on naming blobs, containers and tables, refer to the following list:

- [Naming and Referencing Containers, Blobs, and Metadata](#)
- [Naming Queues and Metadata](#)
- [Naming Tables](#)

A blob name can contain any combination of characters, but reserved URL characters must be properly escaped. Avoid blob names that end with a period (.), a forward slash (/), or a sequence or combination of the two. By convention, the forward slash is the **virtual** directory separator. Do not use a backward slash (\) in a blob name. The client APIs may allow it, but then fail to hash properly, and the signatures will not match.

It is not possible to modify the name of a storage account or container after it has been created. If you want to use a new name, you must delete it and create a new one.

TIP

We recommend that you establish a naming convention for all storage accounts and types before embarking on the development of a new service or application.

Transient fault handling

11/22/2017 • 19 minutes to read • [Edit Online](#)

All applications that communicate with remote services and resources must be sensitive to transient faults. This is especially the case for applications that run in the cloud, where the nature of the environment and connectivity over the Internet means these types of faults are likely to be encountered more often. Transient faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy. These faults are often self-correcting, and if the action is repeated after a suitable delay it is likely succeed.

This document covers general guidance for transient fault handling. For information about handling transient faults when using Microsoft Azure services, see [Azure service-specific retry guidelines](#).

Why do transient faults occur in the cloud?

Transient faults can occur in any environment, on any platform or operating system, and in any kind of application. In solutions that run on local, on-premises infrastructure, performance and availability of the application and its components is typically maintained through expensive and often under-used hardware redundancy, and components and resources are located close to each other. While this makes a failure less likely, it can still result in transient faults - and even an outage through unforeseen events such as external power supply or network issues, or other disaster scenarios.

Cloud hosting, including private cloud systems, can offer a higher overall availability by using shared resources, redundancy, automatic failover, and dynamic resource allocation across a huge number of commodity compute nodes. However, the nature of these environments can mean that transient faults are more likely to occur. There are several reasons for this:

- Many resources in a cloud environment are shared, and access to these resources is subject to throttling in order to protect the resource. Some services will refuse connections when the load rises to a specific level, or a maximum throughput rate is reached, in order to allow processing of existing requests and to maintain performance of the service for all users. Throttling helps to maintain the quality of service for neighbors and other tenants using the shared resource.
- Cloud environments are built using vast numbers of commodity hardware units. They deliver performance by dynamically distributing the load across multiple computing units and infrastructure components, and deliver reliability by automatically recycling or replacing failed units. This dynamic nature means that transient faults and temporary connection failures may occasionally occur.
- There are often more hardware components, including network infrastructure such as routers and load balancers, between the application and the resources and services it uses. This additional infrastructure can occasionally introduce additional connection latency and transient connection faults.
- Network conditions between the client and the server may be variable, especially when communication crosses the Internet. Even in on-premises locations, very heavy traffic loads may slow communication and cause intermittent connection failures.

Challenges

Transient faults can have a huge impact on the perceived availability of an application, even if it has been thoroughly tested under all foreseeable circumstances. To ensure that cloud-hosted applications operate reliably, they must be able to respond to the following challenges:

- The application must be able to detect faults when they occur, and determine if these faults are likely to be

transient, more long-lasting, or are terminal failures. Different resources are likely to return different responses when a fault occurs, and these responses may also vary depending on the context of the operation; for example, the response for an error when reading from storage may be different from response for an error when writing to storage. Many resources and services have well-documented transient failure contracts. However, where such information is not available, it may be difficult to discover the nature of the fault and whether it is likely to be transient.

- The application must be able to retry the operation if it determines that the fault is likely to be transient and keep track of the number of times the operation was retried.
- The application must use an appropriate strategy for the retries. This strategy specifies the number of times it should retry, the delay between each attempt, and the actions to take after a failed attempt. The appropriate number of attempts and the delay between each one are often difficult to determine, and vary based on the type of resource as well as the current operating conditions of the resource and the application itself.

General guidelines

The following guidelines will help you to design a suitable transient fault handing mechanism for your applications:

- **Determine if there is a built-in retry mechanism:**

- Many services provide an SDK or client library that contains a transient fault handling mechanism. The retry policy it uses is typically tailored to the nature and requirements of the target service. Alternatively, REST interfaces for services may return information that is useful in determining whether a retry is appropriate, and how long to wait before the next retry attempt.
- Use the built-in retry mechanism where one is available unless you have specific and well-understood requirements that mean a different retry behavior is more appropriate.

- **Determine if the operation is suitable for retrying:**

- You should only retry operations where the faults are transient (typically indicated by the nature of the error), and if there is at least some likelihood that the operation will succeed when reattempted. There is no point in reattempting operations that indicate an invalid operation such as a database update to an item that does not exist, or requests to a service or resource that has suffered a fatal error
- In general, you should implement retries only where the full impact of this can be determined, and the conditions are well understood and can be validated. If not, leave it to the calling code to implement retries. Remember that the errors returned from resources and services outside your control may evolve over time, and you may need to revisit your transient fault detection logic.
- When you create services or components, consider implementing error codes and messages that will help clients determine whether they should retry failed operations. In particular, indicate if the client should retry the operation (perhaps by returning an **isTransient** value) and suggest a suitable delay before the next retry attempt. If you build a web service, consider returning custom errors defined within your service contracts. Even though generic clients may not be able to read these, they will be useful when building custom clients.

- **Determine an appropriate retry count and interval:**

- It is vital to optimize the retry count and the interval to the type of use case. If you do not retry a sufficient number of times, the application will be unable to complete the operation and is likely to experience a failure. If you retry too many times, or with too short an interval between tries, the application can potentially hold resources such as threads, connections, and memory for long periods, which will adversely affect the health of the application.
- The appropriate values for the time interval and the number of retry attempts depend on the type of operation being attempted. For example, if the operation is part of a user interaction, the interval should be short and only a few retries attempted to avoid making users wait for a response (which holds open connections and can reduce availability for other users). If the operation is part of a long running or critical workflow, where cancelling and restarting the process is expensive or time-consuming, it is

appropriate to wait longer between attempts and retry more times.

- Determining the appropriate intervals between retries is the most difficult part of designing a successful strategy. Typical strategies use the following types of retry interval:
 - **Exponential back-off.** The application waits a short time before the first retry, and then exponentially increasing times between each subsequent retry. For example, it may retry the operation after 3 seconds, 12 seconds, 30 seconds, and so on.
 - **Incremental intervals.** The application waits a short time before the first retry, and then incrementally increasing times between each subsequent retry. For example, it may retry the operation after 3 seconds, 7 seconds, 13 seconds, and so on.
 - **Regular intervals.** The application waits for the same period of time between each attempt. For example, it may retry the operation every 3 seconds.
 - **Immediate retry.** Sometimes a transient fault is extremely short, perhaps caused by an event such as a network packet collision or a spike in a hardware component. In this case, retrying the operation immediately is appropriate because it may succeed if the fault has cleared in the time it takes the application to assemble and send the next request. However, there should never be more than one immediate retry attempt, and you should switch to alternative strategies, such as such as exponential back-off or fallback actions, if the immediate retry fails.
 - **Randomization.** Any of the retry strategies listed above may include a randomization to prevent multiple instances of the client sending subsequent retry attempts at the same time. For example, one instance may retry the operation after 3 seconds, 11 seconds, 28 seconds, and so on while another instance may retry the operation after 4 seconds, 12 seconds, 26 seconds, and so on. Randomization is a useful technique that may be combined with other strategies.
- As a general guideline, use an exponential back-off strategy for background operations, and immediate or regular interval retry strategies for interactive operations. In both cases, you should choose the delay and the retry count so that the maximum latency for all retry attempts is within the required end-to-end latency requirement.
- Take into account the combination of all the factors that contribute to the overall maximum timeout for a retried operation. These factors include the time taken for a failed connection to produce a response (typically set by a timeout value in the client) as well as the delay between retry attempts and the maximum number of retries. The total of all these times can result in very large overall operation times, especially when using an exponential delay strategy where the interval between retries grows rapidly after each failure. If a process must meet a specific service level agreement (SLA), the overall operation time, including all timeouts and delays, must be within that defined in the SLA
- Over-aggressive retry strategies, which have too short intervals or too many retries, can have an adverse effect on the target resource or service. This may prevent the resource or service from recovering from its overloaded state, and it will continue to block or refuse requests. This results in a vicious circle where more and more requests are sent to the resource or service, and consequently its ability to recover is further reduced.
- Take into account the timeout of the operations when choosing the retry intervals to avoid launching a subsequent attempt immediately (for example, if the timeout period is similar to the retry interval). Also consider if you need to keep the total possible period (the timeout plus the retry intervals) to below a specific total time. Operations that have unusually short or very long timeouts may influence how long to wait, and how often to retry the operation.
- Use the type of the exception and any data it contains, or the error codes and messages returned from the service, to optimize the interval and the number of retries. For example, some exceptions or error codes (such as the HTTP code 503 Service Unavailable with a Retry-After header in the response) may indicate how long the error might last, or that the service has failed and will not respond to any subsequent attempt.

- **Avoid anti-patterns:**

- In the vast majority of cases, you should avoid implementations that include duplicated layers of retry

code. Avoid designs that include cascading retry mechanisms, or that implement retry at every stage of an operation that involves a hierarchy of requests, unless you have specific requirements that demand this. In these exceptional circumstances, use policies that prevent excessive numbers of retries and delay periods, and make sure you understand the consequences. For example, if one component makes a request to another, which then accesses the target service, and you implement retry with a count of three on both calls there will be nine retry attempts in total against the service. Many services and resources implement a built-in retry mechanism and you should investigate how you can disable or modify this if you need to implement retries at a higher level.

- Never implement an endless retry mechanism. This is likely to prevent the resource or service recovering from overload situations, and cause throttling and refused connections to continue for a longer period. Use a finite number of retries, or implement a pattern such as [Circuit Breaker](#) to allow the service to recover.
- Never perform an immediate retry more than once.
- Avoid using a regular retry interval, especially when you have a large number of retry attempts, when accessing services and resources in Azure. The optimum approach in this scenario is an exponential back-off strategy with a circuit-breaking capability.
- Prevent multiple instances of the same client, or multiple instances of different clients, from sending retries at the same times. If this is likely to occur, introduce randomization into the retry intervals.

- **Test your retry strategy and implementation:**

- Ensure you fully test your retry strategy implementation under as wide a set of circumstances as possible, especially when both the application and the target resources or services it uses are under extreme load. To check behavior during testing, you can:
 - Inject transient and non-transient faults into the service. For example, send invalid requests or add code that detects test requests and responds with different types of errors. For an example using TestApi, see [Fault Injection Testing with TestApi](#) and [Introduction to TestApi – Part 5: Managed Code Fault Injection APIs](#).
 - Create a mock of the resource or service that returns a range of errors that the real service may return. Ensure you cover all the types of error that your retry strategy is designed to detect.
 - Force transient errors to occur by temporarily disabling or overloading the service if it is a custom service that you created and deployed (you should not, of course, attempt to overload any shared resources or shared services within Azure).
 - For HTTP-based APIs, consider using the FiddlerCore library in your automated tests to change the outcome of HTTP requests, either by adding extra roundtrip times or by changing the response (such as the HTTP status code, headers, body, or other factors). This enables deterministic testing of a subset of the failure conditions, whether transient faults or other types of failure. For more information, see [FiddlerCore](#). For examples of how to use the library, particularly the **HttpMangler** class, examine the [source code for the Azure Storage SDK](#).
 - Perform high load factor and concurrent tests to ensure that the retry mechanism and strategy works correctly under these conditions, and does not have an adverse effect on the operation of the client or cause cross-contamination between requests.

- **Manage retry policy configurations:**

- A *retry policy* is a combination of all of the elements of your retry strategy. It defines the detection mechanism that determines whether a fault is likely to be transient, the type of interval to use (such as regular, exponential back-off, and randomization), the actual interval value(s), and the number of times to retry.
- Retries must be implemented in many places within even the simplest application, and in every layer of more complex applications. Rather than hard-coding the elements of each policy at multiple locations, consider using a central point for storing all the policies. For example, store the values such as the interval and retry count in application configuration files, read them at runtime, and programmatically

build the retry policies. This makes it easier to manage the settings, and to modify and fine tune the values in order to respond to changing requirements and scenarios. However, design the system to store the values rather than rereading a configuration file every time, and ensure suitable defaults are used if the values cannot be obtained from configuration.

- In an Azure Cloud Services application, consider storing the values that are used to build the retry policies at runtime in the service configuration file so that they can be changed without needing to restart the application.
- Take advantage of built-in or default retry strategies available in the client APIs you use, but only where they are appropriate for your scenario. These strategies are typically general-purpose. In some scenarios they may be all that is required, but in other scenarios they may not offer the full range of options to suit your specific requirements. You must understand how the settings will affect your application through testing to determine the most appropriate values.

- **Log and track transient and non-transient faults:**

- As part of your retry strategy, include exception handling and other instrumentation that logs when retry attempts are made. While an occasional transient failure and retry are to be expected, and do not indicate a problem, regular and increasing numbers of retries are often an indicator of an issue that may cause a failure, or is currently impacting application performance and availability.
- Log transient faults as Warning entries rather than Error entries so that monitoring systems do not detect them as application errors that may trigger false alerts.
- Consider storing a value in your log entries that indicates if the retries were caused by throttling in the service, or by other types of faults such as connection failures, so that you can differentiate them during analysis of the data. An increase in the number of throttling errors is often an indicator of a design flaw in the application or the need to switch to a premium service that offers dedicated hardware.
- Consider measuring and logging the overall time taken for operations that include a retry mechanism. This is a good indicator of the overall effect of transient faults on user response times, process latency, and the efficiency of the application use cases. Also log the number of retries occurred in order to understand the factors that contributed to the response time.
- Consider implementing a telemetry and monitoring system that can raise alerts when the number and rate of failures, the average number of retries, or the overall times taken for operations to succeed, is increasing.

- **Manage operations that continually fail:**

- There will be circumstances where the operation continues to fail at every attempt, and it is vital to consider how you will handle this situation:
 - Although a retry strategy will define the maximum number of times that an operation should be retried, it does not prevent the application repeating the operation again, with the same number of retries. For example, if an order processing service fails with a fatal error that puts it out of action permanently, the retry strategy may detect a connection timeout and consider it to be a transient fault. The code will retry the operation a specified number of times and then give up. However, when another customer places an order, the operation will be attempted again - even though it is sure to fail every time.
 - To prevent continual retries for operations that continually fail, consider implementing the [Circuit Breaker pattern](#). In this pattern, if the number of failures within a specified time window exceeds the threshold, requests are returned to the caller immediately as errors, without attempting to access the failed resource or service.
 - The application can periodically test the service, on an intermittent basis and with very long intervals between requests, to detect when it becomes available. An appropriate interval will depend on the scenario, such as the criticality of the operation and the nature of the service, and might be anything between a few minutes and several hours. At the point where the test succeeds, the application can resume normal operations and pass requests to the newly recovered service.

- In the meantime, it may be possible to fall back to another instance of the service (perhaps in a different datacenter or application), use a similar service that offers compatible (perhaps simpler) functionality, or perform some alternative operations in the hope that the service will become available soon. For example, it may be appropriate to store requests for the service in a queue or data store and replay them later. Otherwise you might be able to redirect the user to an alternative instance of the application, degrade the performance of the application but still offer acceptable functionality, or just return a message to the user indicating that the application is not available at present.

- **Other considerations**

- When deciding on the values for the number of retries and the retry intervals for a policy, consider if the operation on the service or resource is part of a long-running or multi-step operation. It may be difficult or expensive to compensate all the other operational steps that have already succeeded when one fails. In this case, a very long interval and a large number of retries may be acceptable as long as it does not block other operations by holding or locking scarce resources.
- Consider if retrying the same operation may cause inconsistencies in data. If some parts of a multi-step process are repeated, and the operations are not idempotent, it may result in an inconsistency. For example, an operation that increments a value, if repeated, will produce an invalid result. Repeating an operation that sends a message to a queue may cause an inconsistency in the message consumer if it cannot detect duplicate messages. To prevent this, ensure that you design each step as an idempotent operation. For more information about idempotency, see [Idempotency Patterns](#).
- Consider the scope of the operations that will be retried. For example, it may be easier to implement retry code at a level that encompasses several operations, and retry them all if one fails. However, doing this may result in idempotency issues or unnecessary rollback operations.
- If you choose a retry scope that encompasses several operations, take into account the total latency of all of them when determining the retry intervals, when monitoring the time taken, and before raising alerts for failures.
- Consider how your retry strategy may affect neighbors and other tenants in a shared application, or when using shared resources and services. Aggressive retry policies can cause an increasing number of transient faults to occur for these other users and for applications that share the resources and services. Likewise, your application may be affected by the retry policies implemented by other users of the resources and services. For mission-critical applications, you may decide to use premium services that are not shared. This provides you with much more control over the load and consequent throttling of these resources and services, which can help to justify the additional cost.

More information

- [Azure service-specific retry guidelines](#)
- [The Transient Fault Handling Application Block](#)
- [Circuit Breaker Pattern](#)
- [Compensating Transaction Pattern](#)
- [Idempotency Patterns](#)

Retry guidance for specific services

6/22/2018 • 40 minutes to read • [Edit Online](#)

Most Azure services and client SDKs include a retry mechanism. However, these differ because each service has different characteristics and requirements, and so each retry mechanism is tuned to a specific service. This guide summarizes the retry mechanism features for the majority of Azure services, and includes information to help you use, adapt, or extend the retry mechanism for that service.

For general guidance on handling transient faults, and retrying connections and operations against services and resources, see [Retry guidance](#).

The following table summarizes the retry features for the Azure services described in this guidance.

Service	Retry Capabilities	Policy Configuration	Scope	Telemetry Features
Azure Active Directory	Native in ADAL library	Embedded into ADAL library	Internal	None
Cosmos DB	Native in service	Non-configurable	Global	TraceSource
Event Hubs	Native in client	Programmatic	Client	None
Redis Cache	Native in client	Programmatic	Client	TextWriter
Search	Native in client	Programmatic	Client	ETW or Custom
Service Bus	Native in client	Programmatic	Namespace Manager, Messaging Factory, and Client	ETW
Service Fabric	Native in client	Programmatic	Client	None
SQL Database with ADO.NET	Polly	Declarative and programmatic	Single statements or blocks of code	Custom
SQL Database with Entity Framework	Native in client	Programmatic	Global per AppDomain	None
SQL Database with Entity Framework Core	Native in client	Programmatic	Global per AppDomain	None
Storage	Native in client	Programmatic	Client and individual operations	TraceSource

Note

For most of the Azure built-in retry mechanisms, there is currently no way apply a different retry policy for different types of error or exception. You should configure a policy that provides the optimum average performance and availability. One way to fine-tune the policy is to analyze log files to determine the type of transient faults that are occurring.

Azure Active Directory

Azure Active Directory (Azure AD) is a comprehensive identity and access management cloud solution that combines core directory services, advanced identity governance, security, and application access management. Azure AD also offers developers an identity management platform to deliver access control to their applications, based on centralized policy and rules.

NOTE

For retry guidance on Managed Service Identity endpoints, see [How to use an Azure VM Managed Service Identity \(MSI\) for token acquisition](#).

Retry mechanism

There is a built-in retry mechanism for Azure Active Directory in the Active Directory Authentication Library (ADAL). To avoid unexpected lockouts, we recommend that third party libraries and application code do **not** retry failed connections, but allow ADAL to handle retries.

Retry usage guidance

Consider the following guidelines when using Azure Active Directory:

- When possible, use the ADAL library and the built-in support for retries.
- If you are using the REST API for Azure Active Directory, retry the operation if the result code is 429 (Too Many Requests) or an error in the 5xx range. Do not retry for any other errors.
- An exponential back-off policy is recommended for use in batch scenarios with Azure Active Directory.

Consider starting with the following settings for retrying operations. These are general purpose settings, and you should monitor the operations and fine tune the values to suit your own scenario.

CONTEXT	SAMPLE TARGET E2E MAX LATENCY	RETRY STRATEGY	SETTINGS	VALUES	HOW IT WORKS
Interactive, UI, or foreground	2 sec	FixedInterval	Retry count Retry interval First fast retry	3 500 ms true	Attempt 1 - delay 0 sec Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	60 sec	ExponentialBackoff	Retry count Min back-off Max back-off Delta back-off First fast retry	5 0 sec 60 sec 2 sec false	Attempt 1 - delay 0 sec Attempt 2 - delay ~2 sec Attempt 3 - delay ~6 sec Attempt 4 - delay ~14 sec Attempt 5 - delay ~30 sec

More information

- [Azure Active Directory Authentication Libraries](#)

Cosmos DB

Cosmos DB is a fully-managed multi-model database that supports schema-less JSON data. It offers configurable

and reliable performance, native JavaScript transactional processing, and is built for the cloud with elastic scale.

Retry mechanism

The `DocumentClient` class automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. Exceptions that the client raises are either beyond the retry policy or are not transient errors.

If Cosmos DB throttles the client, it returns an HTTP 429 error. Check the status code in the `DocumentClientException`.

Policy configuration

The following table shows the default settings for the `RetryOptions` class.

SETTING	DEFAULT VALUE	DESCRIPTION
MaxRetryAttemptsOnThrottledRequests	9	The maximum number of retries if the request fails because Cosmos DB applied rate limiting on the client.
MaxRetryWaitTimeInSeconds	30	The maximum retry time in seconds.

Example

```
DocumentClient client = new DocumentClient(new Uri(endpoint), authKey); ;
var options = client.ConnectionPolicy.RetryOptions;
options.MaxRetryAttemptsOnThrottledRequests = 5;
options.MaxRetryWaitTimeInSeconds = 15;
```

Telemetry

Retry attempts are logged as unstructured trace messages through a .NET **TraceSource**. You must configure a **TraceListener** to capture the events and write them to a suitable destination log.

For example, if you add the following to your App.config file, traces will be generated in a text file in the same location as the executable:

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="SourceSwitch" value="Verbose"/>
    </switches>
    <sources>
      <source name="DocDBTrace" switchName="SourceSwitch" switchType="System.Diagnostics.SourceSwitch" >
        <listeners>
          <add name="MyTextListener" type="System.Diagnostics.TextWriterTraceListener"
            traceOutputOptions="DateTime,ProcessId,ThreadId" initializeData="CosmosDBTrace.txt"></add>
        </listeners>
      </source>
    </sources>
  </system.diagnostics>
</configuration>
```

Event Hubs

Azure Event Hubs is a hyper-scale telemetry ingestion service that collects, transforms, and stores millions of events.

Retry mechanism

Retry behavior in the Azure Event Hubs Client Library is controlled by the `RetryPolicy` property on the `EventHubClient` class. The default policy retries with exponential backoff when Azure Event Hub returns a transient `EventHubsException` or an `OperationCanceledException`.

Example

```
EventHubClient client = EventHubClient.CreateFromConnectionString("[event_hub_connection_string]");
client.RetryPolicy = RetryPolicy.Default;
```

More information

[.NET Standard client library for Azure Event Hubs](#)

Azure Redis Cache

Azure Redis Cache is a fast data access and low latency cache service based on the popular open source Redis Cache. It is secure, managed by Microsoft, and is accessible from any application in Azure.

The guidance in this section is based on using the `StackExchange.Redis` client to access the cache. A list of other suitable clients can be found on the [Redis website](#), and these may have different retry mechanisms.

Note that the `StackExchange.Redis` client uses multiplexing through a single connection. The recommended usage is to create an instance of the client at application startup and use this instance for all operations against the cache. For this reason, the connection to the cache is made only once, and so all of the guidance in this section is related to the retry policy for this initial connection—and not for each operation that accesses the cache.

Retry mechanism

The `StackExchange.Redis` client uses a connection manager class that is configured through a set of options, including:

- **ConnectRetry**. The number of times a failed connection to the cache will be retried.
- **ReconnectRetryPolicy**. The retry strategy to use.
- **ConnectTimeout**. The maximum waiting time in milliseconds.

Policy configuration

Retry policies are configured programmatically by setting the options for the client before connecting to the cache. This can be done by creating an instance of the **ConfigurationOptions** class, populating its properties, and passing it to the **Connect** method.

The built-in classes support linear (constant) delay and exponential backoff with randomized retry intervals. You can also create a custom retry policy by implementing the **IReconnectRetryPolicy** interface.

The following example configures a retry strategy using exponential backoff.

```
var deltaBackOffInMilliseconds = TimeSpan.FromSeconds(5).Milliseconds;
var maxDeltaBackOffInMilliseconds = TimeSpan.FromSeconds(20).Milliseconds;
var options = new ConfigurationOptions
{
    EndPoints = {"localhost"},
    ConnectRetry = 3,
    ReconnectRetryPolicy = new ExponentialRetry(deltaBackOffInMilliseconds, maxDeltaBackOffInMilliseconds),
    ConnectTimeout = 2000
};
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

Alternatively, you can specify the options as a string, and pass this to the **Connect** method. Note that the **ReconnectRetryPolicy** property cannot be set this way, only through code.

```
var options = "localhost,connectRetry=3,connectTimeout=2000";
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

You can also specify options directly when you connect to the cache.

```
var conn = ConnectionMultiplexer.Connect("redis0:6380,redis1:6380,connectRetry=3");
```

For more information, see [Stack Exchange Redis Configuration](#) in the StackExchange.Redis documentation.

The following table shows the default settings for the built-in retry policy.

CONTEXT	SETTING	DEFAULT VALUE (V 1.2.2)	MEANING
ConfigurationOptions	ConnectRetry	3	The number of times to repeat connect attempts during the initial connection operation.
	ConnectTimeout	Maximum 5000 ms plus SyncTimeout	Timeout (ms) for connect operations. Not a delay between retry attempts.
	SyncTimeout	1000	Time (ms) to allow for synchronous operations.
	ReconnectRetryPolicy	LinearRetry 5000 ms	Retry every 5000 ms.

NOTE

For synchronous operations, `SyncTimeout` can add to the end-to-end latency, but setting the value too low can cause excessive timeouts. See [How to troubleshoot Azure Redis Cache](#). In general, avoid using synchronous operations, and use asynchronous operations instead. For more information see [Pipelines and Multiplexers](#).

Retry usage guidance

Consider the following guidelines when using Azure Redis Cache:

- The StackExchange Redis client manages its own retries, but only when establishing a connection to the cache when the application first starts. You can configure the connection timeout, the number of retry attempts, and the time between retries to establish this connection, but the retry policy does not apply to operations against the cache.
- Instead of using a large number of retry attempts, consider falling back by accessing the original data source instead.

Telemetry

You can collect information about connections (but not other operations) using a **TextWriter**.

```
var writer = new StringWriter();
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

An example of the output this generates is shown below.

```
localhost:6379,connectTimeout=2000,connectRetry=3
1 unique nodes specified
Requesting tie-break from localhost:6379 > __Booksleeve_TieBreak...
Allowing endpoints 00:00:02 to respond...
localhost:6379 faulted: SocketFailure on PING
localhost:6379 failed to nominate (Faulted)
> UnableToResolvePhysicalConnection on GET
No masters detected
localhost:6379: Standalone v2.0.0, master; keep-alive: 00:01:00; int: Connecting; sub: Connecting; not in use:
DidNotRespond
localhost:6379: int ops=0, qu=0, qs=0, qc=1, wr=0, sync=1, socks=2; sub ops=0, qu=0, qs=0, qc=0, wr=0, socks=2
Circular op-count snapshot; int: 0 (0.00 ops/s; spans 10s); sub: 0 (0.00 ops/s; spans 10s)
Sync timeouts: 0; fire and forget: 0; last heartbeat: -1s ago
resetting failing connections to retry...
retrying; attempts left: 2...
...
...
```

Examples

The following code example configures a constant (linear) delay between retries when initializing the StackExchange.Redis client. This example shows how to set the configuration using a **ConfigurationOptions** instance.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StackExchange.Redis;

namespace RetryCodeSamples
{
    class CacheRedisCodeSamples
    {
        public async static Task Samples()
        {
            var writer = new StringWriter();
            {
                try
                {
                    var retryTimeInMilliseconds = TimeSpan.FromSeconds(4).Milliseconds; // delay between
retries

                    // Using object-based configuration.
                    var options = new ConfigurationOptions
                    {
                        EndPoints = { "localhost" },
                        ConnectRetry = 3,
                        ReconnectRetryPolicy = new LinearRetry(retryTimeInMilliseconds)
                    };
                    ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);

                    // Store a reference to the multiplexer for use in the application.
                }
                catch
                {
                    Console.WriteLine(writer.ToString());
                    throw;
                }
            }
        }
    }
}
```

The next example sets the configuration by specifying the options as a string. The connection timeout is the maximum period of time to wait for a connection to the cache, not the delay between retry attempts. Note that the **ReconnectRetryPolicy** property can only be set by code.

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StackExchange.Redis;

namespace RetryCodeSamples
{
    class CacheRedisCodeSamples
    {
        public async static Task Samples()
        {
            var writer = new StringWriter();
            {
                try
                {
                    // Using string-based configuration.
                    var options = "localhost,connectRetry=3,connectTimeout=2000";
                    ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);

                    // Store a reference to the multiplexer for use in the application.
                }
                catch
                {
                    Console.WriteLine(writer.ToString());
                    throw;
                }
            }
        }
    }
}
```

For more examples, see [Configuration](#) on the project website.

More information

- [Redis website](#)

Azure Search

Azure Search can be used to add powerful and sophisticated search capabilities to a website or application, quickly and easily tune search results, and construct rich and fine-tuned ranking models.

Retry mechanism

Retry behavior in the Azure Search SDK is controlled by the `SetRetryPolicy` method on the [SearchServiceClient](#) and [SearchIndexClient](#) classes. The default policy retries with exponential backoff when Azure Search returns a 5xx or 408 (Request Timeout) response.

Telemetry

Trace with ETW or by registering a custom trace provider. For more information, see the [AutoRest documentation](#).

Service Bus

Service Bus is a cloud messaging platform that provides loosely coupled message exchange with improved scale and resiliency for components of an application, whether hosted in the cloud or on-premises.

Retry mechanism

Service Bus implements retries using implementations of the [RetryPolicy](#) base class. All of the Service Bus clients expose a **RetryPolicy** property that can be set to one of the implementations of the **RetryPolicy** base class. The built-in implementations are:

- The [RetryExponential Class](#). This exposes properties that control the back-off interval, the retry count, and the **TerminationTimeBuffer** property that is used to limit the total time for the operation to complete.
- The [NoRetry Class](#). This is used when retries at the Service Bus API level are not required, such as when retries are managed by another process as part of a batch or multiple step operation.

Service Bus actions can return a range of exceptions, as listed in [Service Bus messaging exceptions](#). The list provides information about which if these indicate that retrying the operation is appropriate. For example, a **ServerBusyException** indicates that the client should wait for a period of time, then retry the operation. The occurrence of a **ServerBusyException** also causes Service Bus to switch to a different mode, in which an extra 10-second delay is added to the computed retry delays. This mode is reset after a short period.

The exceptions returned from Service Bus expose the **IsTransient** property that indicates if the client should retry the operation. The built-in **RetryExponential** policy relies on the **IsTransient** property in the **MessagingException** class, which is the base class for all Service Bus exceptions. If you create custom implementations of the **RetryPolicy** base class you could use a combination of the exception type and the **IsTransient** property to provide more fine-grained control over retry actions. For example, you could detect a **QuotaExceededException** and take action to drain the queue before retrying sending a message to it.

Policy configuration

Retry policies are set programmatically, and can be set as a default policy for a **NamespaceManager** and for a **MessagingFactory**, or individually for each messaging client. To set the default retry policy for a messaging session you set the **RetryPolicy** of the **NamespaceManager**.

```
namespaceManager.Settings.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                                               maxBackoff: TimeSpan.FromSeconds(30),
                                                               maxRetryCount: 3);
```

To set the default retry policy for all clients created from a messaging factory, you set the **RetryPolicy** of the **MessagingFactory**.

```
messagingFactory.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                                       maxBackoff: TimeSpan.FromSeconds(30),
                                                       maxRetryCount: 3);
```

To set the retry policy for a messaging client, or to override its default policy, you set its **RetryPolicy** property using an instance of the required policy class:

```
client.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                           maxBackoff: TimeSpan.FromSeconds(30),
                                           maxRetryCount: 3);
```

The retry policy cannot be set at the individual operation level. It applies to all operations for the messaging client. The following table shows the default settings for the built-in retry policy.

SETTING	DEFAULT VALUE	MEANING
Policy	Exponential	Exponential back-off.

SETTING	DEFAULT VALUE	MEANING
MinimalBackoff	0	Minimum back-off interval. This is added to the retry interval computed from deltaBackoff.
MaximumBackoff	30 seconds	Maximum back-off interval. MaximumBackoff is used if the computed retry interval is greater than MaxBackoff.
DeltaBackoff	3 seconds	Back-off interval between retries. Multiples of this timespan will be used for subsequent retry attempts.
TimeBuffer	5 seconds	The termination time buffer associated with the retry. Retry attempts will be abandoned if the remaining time is less than TimeBuffer.
MaxRetryCount	10	The maximum number of retries.
ServerBusyBaseSleepTime	10 seconds	If the last exception encountered was ServerBusyException , this value will be added to the computed retry interval. This value cannot be changed.

Retry usage guidance

Consider the following guidelines when using Service Bus:

- When using the built-in **RetryExponential** implementation, do not implement a fallback operation as the policy reacts to Server Busy exceptions and automatically switches to an appropriate retry mode.
- Service Bus supports a feature called Paired Namespaces, which implements automatic failover to a backup queue in a separate namespace if the queue in the primary namespace fails. Messages from the secondary queue can be sent back to the primary queue when it recovers. This feature helps to address transient failures. For more information, see [Asynchronous Messaging Patterns and High Availability](#).

Consider starting with following settings for retrying operations. These are general purpose settings, and you should monitor the operations and fine tune the values to suit your own scenario.

CONTEXT	EXAMPLE MAXIMUM LATENCY	RETRY POLICY	SETTINGS	HOW IT WORKS
Interactive, UI, or foreground	2 seconds*	Exponential	MinimumBackoff = 0 MaximumBackoff = 30 sec. DeltaBackoff = 300 msec. TimeBuffer = 300 msec. MaxRetryCount = 2	Attempt 1: Delay 0 sec. Attempt 2: Delay ~300 msec. Attempt 3: Delay ~900 msec.

CONTEXT	EXAMPLE MAXIMUM LATENCY	RETRY POLICY	SETTINGS	HOW IT WORKS
Background or batch	30 seconds	Exponential	MinimumBackoff = 1 MaximumBackoff = 30 sec. DeltaBackoff = 1.75 sec. TimeBuffer = 5 sec. MaxRetryCount = 3	Attempt 1: Delay ~1 sec. Attempt 2: Delay ~3 sec. Attempt 3: Delay ~6 msec. Attempt 4: Delay ~13 msec.

* Not including additional delay that is added if a Server Busy response is received.

Telemetry

Service Bus logs retries as ETW events using an **EventSource**. You must attach an **EventListener** to the event source to capture the events and view them in Performance Viewer, or write them to a suitable destination log. You could use the [Semantic Logging Application Block](#) to do this. The retry events are of the following form:

```
Microsoft-ServiceBus-Client/RetryPolicyIteration
ThreadId="14,500"
FormattedMessage="[TrackingId:] RetryExponential: Operation Get:https://retry-
tests.servicebus.windows.net/TestQueue/?api-version=2014-05 at iteration 0 is retrying after 00:00:00.1000000
sleep because of Microsoft.ServiceBus.Messaging.MessagingCommunicationException: The remote name could not be
resolved: 'retry-tests.servicebus.windows.net'.TrackingId:6a26f99c-dc6d-422e-8565-f89fdd0d4fe3,
TimeStamp:9/5/2014 10:00:13 PM."
trackingId=""
policyType="RetryExponential"
operation="Get:https://retry-tests.servicebus.windows.net/TestQueue/?api-version=2014-05"
iteration="0"
iterationSleep="00:00:00.1000000"
lastExceptionType="Microsoft.ServiceBus.Messaging.MessagingCommunicationException"
exceptionMessage="The remote name could not be resolved: 'retry-
tests.servicebus.windows.net'.TrackingId:6a26f99c-dc6d-422e-8565-f89fdd0d4fe3,TimeStamp:9/5/2014 10:00:13 PM"
```

Examples

The following code example shows how to set the retry policy for:

- A namespace manager. The policy applies to all operations on that manager, and cannot be overridden for individual operations.
- A messaging factory. The policy applies to all clients created from that factory, and cannot be overridden when creating individual clients.
- An individual messaging client. After a client has been created, you can set the retry policy for that client. The policy applies to all operations on that client.

```
using System;
using System.Threading.Tasks;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;

namespace RetryCodeSamples
{
    class ServiceBusCodeSamples
    {
        private const string connectionString =
            @"Endpoint=sb://[my-namespace].servicebus.windows.net/;
                SharedAccessKeyName=RootManageSharedAccessKey;
                SharedAccessKey=C99.....Mk=";

        public async static Task Samples()
```

```

    {
        const string QueueName = "TestQueue";

        ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http;

        var namespaceManager = NamespaceManager.CreateFromConnectionString(connectionString);

        // The namespace manager will have a default exponential policy with 10 retry attempts
        // and a 3 second delay delta.
        // Retry delays will be approximately 0 sec, 3 sec, 9 sec, 25 sec and the fixed 30 sec,
        // with an extra 10 sec added when receiving a ServiceBusyException.

        {
            // Set different values for the retry policy, used for all operations on the namespace
            manager.

            namespaceManager.Settings.RetryPolicy =
                new RetryExponential(
                    minBackoff: TimeSpan.FromSeconds(0),
                    maxBackoff: TimeSpan.FromSeconds(30),
                    maxRetryCount: 3);

            // Policies cannot be specified on a per-operation basis.
            if (!await namespaceManager.QueueExistsAsync(QueueName))
            {
                await namespaceManager.CreateQueueAsync(QueueName);
            }
        }

        var messagingFactory = MessagingFactory.Create(
            namespaceManager.Address, namespaceManager.Settings.TokenProvider);
        // The messaging factory will have a default exponential policy with 10 retry attempts
        // and a 3 second delay delta.
        // Retry delays will be approximately 0 sec, 3 sec, 9 sec, 25 sec and the fixed 30 sec,
        // with an extra 10 sec added when receiving a ServiceBusyException.

        {
            // Set different values for the retry policy, used for clients created from it.
            messagingFactory.RetryPolicy =
                new RetryExponential(
                    minBackoff: TimeSpan.FromSeconds(1),
                    maxBackoff: TimeSpan.FromSeconds(30),
                    maxRetryCount: 3);

            // Policies cannot be specified on a per-operation basis.
            var session = await messagingFactory.AcceptMessageSessionAsync();
        }

        {
            var client = messagingFactory.CreateQueueClient(QueueName);
            // The client inherits the policy from the factory that created it.

            // Set different values for the retry policy on the client.
            client.RetryPolicy =
                new RetryExponential(
                    minBackoff: TimeSpan.FromSeconds(0.1),
                    maxBackoff: TimeSpan.FromSeconds(30),
                    maxRetryCount: 3);

            // Policies cannot be specified on a per-operation basis.
            var session = await client.AcceptMessageSessionAsync();
        }
    }
}

```

More information

- [Asynchronous Messaging Patterns and High Availability](#)

Service Fabric

Distributing reliable services in a Service Fabric cluster guards against most of the potential transient faults discussed in this article. Some transient faults are still possible, however. For example, the naming service might be in the middle of a routing change when it gets a request, causing it to throw an exception. If the same request comes 100 milliseconds later, it will probably succeed.

Internally, Service Fabric manages this kind of transient fault. You can configure some settings by using the `OperationRetrySettings` class while setting up your services. The following code shows an example. In most cases, this should not be necessary, and the default settings will be fine.

```
FabricTransportRemotingSettings transportSettings = new FabricTransportRemotingSettings
{
    OperationTimeout = TimeSpan.FromSeconds(30)
};

var retrySettings = new OperationRetrySettings(TimeSpan.FromSeconds(15), TimeSpan.FromSeconds(1), 5);

var clientFactory = new FabricTransportServiceRemotingClientFactory(transportSettings);

var serviceProxyFactory = new ServiceProxyFactory((c) => clientFactory, retrySettings);

var client = serviceProxyFactory.CreateServiceProxy<ISomeService>(
    new Uri("fabric:/SomeApp/SomeStatefulReliableService"),
    new ServicePartitionKey(0));
```

More information

- [Remote Exception Handling](#)

SQL Database using ADO.NET

SQL Database is a hosted SQL database available in a range of sizes and as both a standard (shared) and premium (non-shared) service.

Retry mechanism

SQL Database has no built-in support for retries when accessed using ADO.NET. However, the return codes from requests can be used to determine why a request failed. For more information about SQL Database throttling, see [Azure SQL Database resource limits](#). For a list of relevant error codes, see [SQL error codes for SQL Database client applications](#).

You can use the Polly library to implement retries for SQL Database. See [Transient fault handling with Polly](#).

Retry usage guidance

Consider the following guidelines when accessing SQL Database using ADO.NET:

- Choose the appropriate service option (shared or premium). A shared instance may suffer longer than usual connection delays and throttling due to the usage by other tenants of the shared server. If more predictable performance and reliable low latency operations are required, consider choosing the premium option.
- Ensure that you perform retries at the appropriate level or scope to avoid non-idempotent operations causing inconsistency in the data. Ideally, all operations should be idempotent so that they can be repeated without causing inconsistency. Where this is not the case, the retry should be performed at a level or scope that allows all related changes to be undone if one operation fails; for example, from within a transactional scope. For more information, see [Cloud Service Fundamentals Data Access Layer – Transient Fault Handling](#).
- A fixed interval strategy is not recommended for use with Azure SQL Database except for interactive scenarios

where there are only a few retries at very short intervals. Instead, consider using an exponential back-off strategy for the majority of scenarios.

- Choose a suitable value for the connection and command timeouts when defining connections. Too short a timeout may result in premature failures of connections when the database is busy. Too long a timeout may prevent the retry logic working correctly by waiting too long before detecting a failed connection. The value of the timeout is a component of the end-to-end latency; it is effectively added to the retry delay specified in the retry policy for every retry attempt.
- Close the connection after a certain number of retries, even when using an exponential back off retry logic, and retry the operation on a new connection. Retrying the same operation multiple times on the same connection can be a factor that contributes to connection problems. For an example of this technique, see [Cloud Service Fundamentals Data Access Layer – Transient Fault Handling](#).
- When connection pooling is in use (the default) there is a chance that the same connection will be chosen from the pool, even after closing and reopening a connection. If this is the case, a technique to resolve it is to call the **ClearPool** method of the **SqlConnection** class to mark the connection as not reusable. However, you should do this only after several connection attempts have failed, and only when encountering the specific class of transient failures such as SQL timeouts (error code -2) related to faulty connections.
- If the data access code uses transactions initiated as **TransactionScope** instances, the retry logic should reopen the connection and initiate a new transaction scope. For this reason, the retryable code block should encompass the entire scope of the transaction.

Consider starting with following settings for retrying operations. These are general purpose settings, and you should monitor the operations and fine tune the values to suit your own scenario.

CONTEXT	SAMPLE TARGET E2E MAX LATENCY	RETRY STRATEGY	SETTINGS	VALUES	HOW IT WORKS
Interactive, UI, or foreground	2 sec	FixedInterval	Retry count Retry interval First fast retry	3 500 ms true	Attempt 1 - delay 0 sec Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	30 sec	ExponentialBackoff	Retry count Min back-off Max back-off Delta back-off First fast retry	5 0 sec 60 sec 2 sec false	Attempt 1 - delay 0 sec Attempt 2 - delay ~2 sec Attempt 3 - delay ~6 sec Attempt 4 - delay ~14 sec Attempt 5 - delay ~30 sec

NOTE

The end-to-end latency targets assume the default timeout for connections to the service. If you specify longer connection timeouts, the end-to-end latency will be extended by this additional time for every retry attempt.

Examples

This section shows how you can use Polly to access Azure SQL Database using a set of retry policies configured in the `Policy` class.

The following code shows an extension method on the `SqlCommand` class that calls `ExecuteAsync` with exponential

backoff.

```
public async static Task<SqlDataReader> ExecuteReaderWithRetryAsync(this SqlCommand command)
{
    GuardConnectionIsNotNull(command);

    var policy = Policy.Handle<Exception>().WaitAndRetryAsync(
        retryCount: 3, // Retry 3 times
        sleepDurationProvider: attempt => TimeSpan.FromMilliseconds(200 * Math.Pow(2, attempt - 1)), // Exponential backoff based on an initial 200ms delay.
        onRetry: (exception, attempt) =>
    {
        // Capture some info for logging/telemetry.
        logger.LogWarning($"ExecuteReaderWithRetryAsync: Retry {attempt} due to {exception}.");
    });

    // Retry the following call according to the policy.
    await policy.ExecuteAsync<SqlDataReader>(async token =>
    {
        // This code is executed within the Policy

        if (conn.State != System.Data.ConnectionState.Open) await conn.OpenAsync(token);
        return await command.ExecuteReaderAsync(System.Data.CommandBehavior.Default, token);

    }, cancellationToken);
}
```

This asynchronous extension method can be used as follows.

```
var sqlCommand = sqlConnection.CreateCommand();
sqlCommand.CommandText = "[some query]";

using (var reader = await sqlCommand.ExecuteReaderWithRetryAsync())
{
    // Do something with the values
}
```

More information

- [Cloud Service Fundamentals Data Access Layer – Transient Fault Handling](#)

For general guidance on getting the most from SQL Database, see [Azure SQL Database Performance and Elasticity Guide](#).

SQL Database using Entity Framework 6

SQL Database is a hosted SQL database available in a range of sizes and as both a standard (shared) and premium (non-shared) service. Entity Framework is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write.

Retry mechanism

Retry support is provided when accessing SQL Database using Entity Framework 6.0 and higher through a mechanism called [Connection Resiliency / Retry Logic](#). The main features of the retry mechanism are:

- The primary abstraction is the **IDbExecutionStrategy** interface. This interface:
 - Defines synchronous and asynchronous **Execute*** methods.
 - Defines classes that can be used directly or can be configured on a database context as a default strategy, mapped to provider name, or mapped to a provider name and server name. When configured on a context, retries occur at the level of individual database operations, of which there might be several for a

- given context operation.
- Defines when to retry a failed connection, and how.
- It includes several built-in implementations of the **IDbExecutionStrategy** interface:
 - Default - no retrying.
 - Default for SQL Database (automatic) - no retrying, but inspects exceptions and wraps them with suggestion to use the SQL Database strategy.
 - Default for SQL Database - exponential (inherited from base class) plus SQL Database detection logic.
- It implements an exponential back-off strategy that includes randomization.
- The built-in retry classes are stateful and are not thread safe. However, they can be reused after the current operation is completed.
- If the specified retry count is exceeded, the results are wrapped in a new exception. It does not bubble up the current exception.

Policy configuration

Retry support is provided when accessing SQL Database using Entity Framework 6.0 and higher. Retry policies are configured programmatically. The configuration cannot be changed on a per-operation basis.

When configuring a strategy on the context as the default, you specify a function that creates a new strategy on demand. The following code shows how you can create a retry configuration class that extends the **DbConfiguration** base class.

```
public class BloggingContextConfiguration : DbConfiguration
{
    public BlogConfiguration()
    {
        // Set up the execution strategy for SQL Database (exponential) with 5 retries and 4 sec delay
        this.SetExecutionStrategy(
            "System.Data.SqlClient", () => new SqlAzureExecutionStrategy(5, TimeSpan.FromSeconds(4)));
    }
}
```

You can then specify this as the default retry strategy for all operations using the **SetConfiguration** method of the **DbConfiguration** instance when the application starts. By default, EF will automatically discover and use the configuration class.

```
DbConfiguration.SetConfiguration(new BloggingContextConfiguration());
```

You can specify the retry configuration class for a context by annotating the context class with a **DbConfigurationType** attribute. However, if you have only one configuration class, EF will use it without the need to annotate the context.

```
[DbConfigurationType(typeof(BloggingContextConfiguration))]
public class BloggingContext : DbContext
```

If you need to use different retry strategies for specific operations, or disable retries for specific operations, you can create a configuration class that allows you to suspend or swap strategies by setting a flag in the **CallContext**. The configuration class can use this flag to switch strategies, or disable the strategy you provide and use a default strategy. For more information, see [Suspend Execution Strategy](#) in the page Limitations with Retrying Execution Strategies (EF6 onwards).

Another technique for using specific retry strategies for individual operations is to create an instance of the required strategy class and supply the desired settings through parameters. You then invoke its **ExecuteAsync** method.

```

var executionStrategy = new SqlAzureExecutionStrategy(5, TimeSpan.FromSeconds(4));
var blogs = await executionStrategy.ExecuteAsync(
    async () =>
    {
        using (var db = new BloggingContext("Blogs"))
        {
            // Acquire some values asynchronously and return them
        }
    },
    new CancellationToken()
);

```

The simplest way to use a **DbConfiguration** class is to locate it in the same assembly as the **DbContext** class. However, this is not appropriate when the same context is required in different scenarios, such as different interactive and background retry strategies. If the different contexts execute in separate AppDomains, you can use the built-in support for specifying configuration classes in the configuration file or set it explicitly using code. If the different contexts must execute in the same AppDomain, a custom solution will be required.

For more information, see [Code-Based Configuration \(EF6 onwards\)](#).

The following table shows the default settings for the built-in retry policy when using EF6.

SETTING	DEFAULT VALUE	MEANING
Policy	Exponential	Exponential back-off.
MaxRetryCount	5	The maximum number of retries.
MaxDelay	30 seconds	The maximum delay between retries. This value does not affect how the series of delays are computed. It only defines an upper bound.
DefaultCoefficient	1 second	The coefficient for the exponential back-off computation. This value cannot be changed.
DefaultRandomFactor	1.1	The multiplier used to add a random delay for each entry. This value cannot be changed.
DefaultExponentialBase	2	The multiplier used to calculate the next delay. This value cannot be changed.

Retry usage guidance

Consider the following guidelines when accessing SQL Database using EF6:

- Choose the appropriate service option (shared or premium). A shared instance may suffer longer than usual connection delays and throttling due to the usage by other tenants of the shared server. If predictable performance and reliable low latency operations are required, consider choosing the premium option.
- A fixed interval strategy is not recommended for use with Azure SQL Database. Instead, use an exponential back-off strategy because the service may be overloaded, and longer delays allow more time for it to recover.
- Choose a suitable value for the connection and command timeouts when defining connections. Base the timeout on both your business logic design and through testing. You may need to modify this value over time as the volumes of data or the business processes change. Too short a timeout may result in premature failures of connections when the database is busy. Too long a timeout may prevent the retry logic working correctly by

waiting too long before detecting a failed connection. The value of the timeout is a component of the end-to-end latency, although you cannot easily determine how many commands will execute when saving the context. You can change the default timeout by setting the **CommandTimeout** property of the **DbContext** instance.

- Entity Framework supports retry configurations defined in configuration files. However, for maximum flexibility on Azure you should consider creating the configuration programmatically within the application. The specific parameters for the retry policies, such as the number of retries and the retry intervals, can be stored in the service configuration file and used at runtime to create the appropriate policies. This allows the settings to be changed without requiring the application to be restarted.

Consider starting with the following settings for retrying operations. You cannot specify the delay between retry attempts (it is fixed and generated as an exponential sequence). You can specify only the maximum values, as shown here; unless you create a custom retry strategy. These are general purpose settings, and you should monitor the operations and fine tune the values to suit your own scenario.

CONTEXT	SAMPLE TARGET E2E MAX LATENCY	RETRY POLICY	SETTINGS	VALUES	HOW IT WORKS
Interactive, UI, or foreground	2 seconds	Exponential	MaxRetryCount MaxDelay	3 750 ms	Attempt 1 - delay 0 sec Attempt 2 - delay 750 ms Attempt 3 – delay 750 ms
Background or batch	30 seconds	Exponential	MaxRetryCount MaxDelay	5 12 seconds	Attempt 1 - delay 0 sec Attempt 2 - delay ~1 sec Attempt 3 - delay ~3 sec Attempt 4 - delay ~7 sec Attempt 5 - delay 12 sec

NOTE

The end-to-end latency targets assume the default timeout for connections to the service. If you specify longer connection timeouts, the end-to-end latency will be extended by this additional time for every retry attempt.

Examples

The following code example defines a simple data access solution that uses Entity Framework. It sets a specific retry strategy by defining an instance of a class named **BlogConfiguration** that extends **DbConfiguration**.

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.SqlServer;
using System.Threading.Tasks;

namespace RetryCodeSamples
{
    public class BlogConfiguration : DbConfiguration
    {
        public BlogConfiguration()
        {
            // Set up the execution strategy for SQL Database (exponential) with 5 retries and 12 sec delay.
            // These values could be loaded from configuration rather than being hard-coded.
            this.SetExecutionStrategy(
                "System.Data.SqlClient", () => new SqlAzureExecutionStrategy(5,
TimeSpan.FromSeconds(12)));
        }
    }

    // Specify the configuration type if more than one has been defined.
    // [DbConfigurationType(typeof(BlogConfiguration))]
    public class BloggingContext : DbContext
    {
        // Definition of content goes here.
    }

    class EF6CodeSamples
    {
        public async static Task Samples()
        {
            // Execution strategy configured by DbConfiguration subclass, discovered automatically or
            // or explicitly indicated through configuration or with an attribute. Default is no retries.
            using (var db = new BloggingContext("Blogs"))
            {
                // Add, edit, delete blog items here, then:
                await db.SaveChangesAsync();
            }
        }
    }
}

```

More examples of using the Entity Framework retry mechanism can be found in [Connection Resiliency / Retry Logic](#).

More information

- [Azure SQL Database Performance and Elasticity Guide](#)

SQL Database using Entity Framework Core

[Entity Framework Core](#) is an object-relational mapper that enables .NET Core developers to work with data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write. This version of Entity Framework was written from the ground up, and doesn't automatically inherit all the features from EF6.x.

Retry mechanism

Retry support is provided when accessing SQL Database using Entity Framework Core through a mechanism called [Connection Resiliency](#). Connection resiliency was introduced in EF Core 1.1.0.

The primary abstraction is the `IExecutionStrategy` interface. The execution strategy for SQL Server, including SQL Azure, is aware of the exception types that can be retried and has sensible defaults for maximum retries, delay between retries, and so on.

Examples

The following code enables automatic retries when configuring the DbContext object, which represents a session with the database.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=
(localdb)\mssqllocaldb;Database=EFMiscellaneous.ConnectionResiliency;Trusted_Connection=True;",
            options => options.EnableRetryOnFailure());
}
```

The following code shows how to execute a transaction with automatic retries, by using an execution strategy. The transaction is defined in a delegate. If a transient failure occurs, the execution strategy will invoke the delegate again.

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var transaction = db.Database.BeginTransaction())
        {
            db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            db.SaveChanges();

            db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });
            db.SaveChanges();

            transaction.Commit();
        }
    });
}
```

Azure Storage

Azure storage services include table and blob storage, files, and storage queues.

Retry mechanism

Retries occur at the individual REST operation level and are an integral part of the client API implementation. The client storage SDK uses classes that implement the [IExtendedRetryPolicy Interface](#).

There are different implementations of the interface. Storage clients can choose from policies specifically designed for accessing tables, blobs, and queues. Each implementation uses a different retry strategy that essentially defines the retry interval and other details.

The built-in classes provide support for linear (constant delay) and exponential with randomization retry intervals. There is also a no retry policy for use when another process is handling retries at a higher level. However, you can implement your own retry classes if you have specific requirements not provided by the built-in classes.

Alternate retries switch between primary and secondary storage service location if you are using read access geo-redundant storage (RA-GRS) and the result of the request is a retryable error. See [Azure Storage Redundancy Options](#) for more information.

Policy configuration

Retry policies are configured programmatically. A typical procedure is to create and populate a

TableRequestOptions, **BlobRequestOptions**, **FileRequestOptions**, or **QueueRequestOptions** instance.

```
TableRequestOptions interactiveRequestOption = new TableRequestOptions()
{
    RetryPolicy = new LinearRetry(TimeSpan.FromMilliseconds(500), 3),
    // For Read-access geo-redundant storage, use PrimaryThenSecondary.
    // Otherwise set this to PrimaryOnly.
    LocationMode = LocationMode.PrimaryThenSecondary,
    // Maximum execution time based on the business use case.
    MaximumExecutionTime = TimeSpan.FromSeconds(2)
};
```

The request options instance can then be set on the client, and all operations with the client will use the specified request options.

```
client.DefaultRequestOptions = interactiveRequestOption;
var stats = await client.GetServiceStatsAsync();
```

You can override the client request options by passing a populated instance of the request options class as a parameter to operation methods.

```
var stats = await client.GetServiceStatsAsync(interactiveRequestOption, operationContext: null);
```

You use an **OperationContext** instance to specify the code to execute when a retry occurs and when an operation has completed. This code can collect information about the operation for use in logs and telemetry.

```
// Set up notifications for an operation
var context = new OperationContext();
context.ClientRequestID = "some request id";
context.Retrying += (sender, args) =>
{
    /* Collect retry information */
};
context.RequestCompleted += (sender, args) =>
{
    /* Collect operation completion information */
};
var stats = await client.GetServiceStatsAsync(null, context);
```

In addition to indicating whether a failure is suitable for retry, the extended retry policies return a **RetryContext** object that indicates the number of retries, the results of the last request, whether the next retry will happen in the primary or secondary location (see table below for details). The properties of the **RetryContext** object can be used to decide if and when to attempt a retry. For more details, see [IExtendedRetryPolicy.Evaluate Method](#).

The following tables show the default settings for the built-in retry policies.

Request options

SETTING	DEFAULT VALUE	MEANING
MaximumExecutionTime	None	Maximum execution time for the request, including all potential retry attempts. If it is not specified, then the amount of time that a request is permitted to take is unlimited. In other words, the request might hang.

SETTING	DEFAULT VALUE	MEANING
ServerTimeout	None	Server timeout interval for the request (value is rounded to seconds). If not specified, it will use the default value for all requests to the server. Usually, the best option is to omit this setting so that the server default is used.
LocationMode	None	If the storage account is created with the Read access geo-redundant storage (RA-GRS) replication option, you can use the location mode to indicate which location should receive the request. For example, if PrimaryThenSecondary is specified, requests are always sent to the primary location first. If a request fails, it is sent to the secondary location.
RetryPolicy	ExponentialPolicy	See below for details of each option.

Exponential policy

SETTING	DEFAULT VALUE	MEANING
maxAttempt	3	Number of retry attempts.
deltaBackoff	4 seconds	Back-off interval between retries. Multiples of this timespan, including a random element, will be used for subsequent retry attempts.
MinBackoff	3 seconds	Added to all retry intervals computed from deltaBackoff. This value cannot be changed.
MaxBackoff	120 seconds	MaxBackoff is used if the computed retry interval is greater than MaxBackoff. This value cannot be changed.

Linear policy

SETTING	DEFAULT VALUE	MEANING
maxAttempt	3	Number of retry attempts.
deltaBackoff	30 seconds	Back-off interval between retries.

Retry usage guidance

Consider the following guidelines when accessing Azure storage services using the storage client API:

- Use the built-in retry policies from the `Microsoft.WindowsAzure.Storage.RetryPolicies` namespace where they are appropriate for your requirements. In most cases, these policies will be sufficient.
- Use the **ExponentialRetry** policy in batch operations, background tasks, or non-interactive scenarios. In these scenarios, you can typically allow more time for the service to recover—with a consequently increased chance of the operation eventually succeeding.

- Consider specifying the **MaximumExecutionTime** property of the **RequestOptions** parameter to limit the total execution time, but take into account the type and size of the operation when choosing a timeout value.
- If you need to implement a custom retry, avoid creating wrappers around the storage client classes. Instead, use the capabilities to extend the existing policies through the **IExtendedRetryPolicy** interface.
- If you are using read access geo-redundant storage (RA-GRS) you can use the **LocationMode** to specify that retry attempts will access the secondary read-only copy of the store should the primary access fail. However, when using this option you must ensure that your application can work successfully with data that may be stale if the replication from the primary store has not yet completed.

Consider starting with following settings for retrying operations. These are general purpose settings, and you should monitor the operations and fine tune the values to suit your own scenario.

CONTEXT	SAMPLE TARGET E2E MAX LATENCY	RETRY POLICY	SETTINGS	VALUES	HOW IT WORKS
Interactive, UI, or foreground	2 seconds	Linear	maxAttempt deltaBackoff	3 500 ms	Attempt 1 - delay 500 ms Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	30 seconds	Exponential	maxAttempt deltaBackoff	5 4 seconds	Attempt 1 - delay ~3 sec Attempt 2 - delay ~7 sec Attempt 3 - delay ~15 sec

Telemetry

Retry attempts are logged to a **TraceSource**. You must configure a **TraceListener** to capture the events and write them to a suitable destination log. You can use the **TextWriterTraceListener** or **XmlWriterTraceListener** to write the data to a log file, the **EventLogTraceListener** to write to the Windows Event Log, or the **EventProviderTraceListener** to write trace data to the ETW subsystem. You can also configure auto-flushing of the buffer, and the verbosity of events that will be logged (for example, Error, Warning, Informational, and Verbose). For more information, see [Client-side Logging with the .NET Storage Client Library](#).

Operations can receive an **OperationContext** instance, which exposes a **Retrying** event that can be used to attach custom telemetry logic. For more information, see [OperationContext.Retrying Event](#).

Examples

The following code example shows how to create two **TableRequestOptions** instances with different retry settings; one for interactive requests and one for background requests. The example then sets these two retry policies on the client so that they apply for all requests, and also sets the interactive strategy on a specific request so that it overrides the default settings applied to the client.

```
using System;
using System.Threading.Tasks;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.RetryPolicies;
using Microsoft.WindowsAzure.Storage.Table;

namespace RetryCodeSamples
{
    class AzureStorageCodeSamples
    {
        private const string connectionString = "UseDevelopmentStorage=true";
```

```

public async static Task Samples()
{
    var storageAccount = CloudStorageAccount.Parse(connectionString);

    TableRequestOptions interactiveRequestOption = new TableRequestOptions()
    {
        RetryPolicy = new LinearRetry(TimeSpan.FromMilliseconds(500), 3),
        // For Read-access geo-redundant storage, use PrimaryThenSecondary.
        // Otherwise set this to PrimaryOnly.
        LocationMode = LocationMode.PrimaryThenSecondary,
        // Maximum execution time based on the business use case.
        MaximumExecutionTime = TimeSpan.FromSeconds(2)
    };

    TableRequestOptions backgroundRequestOption = new TableRequestOptions()
    {
        // Client has a default exponential retry policy with 4 sec delay and 3 retry attempts
        // Retry delays will be approximately 3 sec, 7 sec, and 15 sec
        MaximumExecutionTime = TimeSpan.FromSeconds(30),
        // PrimaryThenSecondary in case of Read-access geo-redundant storage, else set this to
PrimaryOnly
        LocationMode = LocationMode.PrimaryThenSecondary
    };

    var client = storageAccount.CreateCloudTableClient();
    // Client has a default exponential retry policy with 4 sec delay and 3 retry attempts
    // Retry delays will be approximately 3 sec, 7 sec, and 15 sec
    // ServerTimeout and MaximumExecutionTime are not set

    {
        // Set properties for the client (used on all requests unless overridden)
        // Different exponential policy parameters for background scenarios
        client.DefaultRequestOptions = backgroundRequestOption;
        // Linear policy for interactive scenarios
        client.DefaultRequestOptions = interactiveRequestOption;
    }

    {
        // set properties for a specific request
        var stats = await client.GetServiceStatsAsync(interactiveRequestOption, operationContext:
null);
    }

    {
        // Set up notifications for an operation
        var context = new OperationContext();
        context.ClientRequestID = "some request id";
        context.Retrying += (sender, args) =>
        {
            /* Collect retry information */
        };
        context.RequestCompleted += (sender, args) =>
        {
            /* Collect operation completion information */
        };
        var stats = await client.GetServiceStatsAsync(null, context);
    }
}
}

```

More information

- [Azure Storage Client Library Retry Policy Recommendations](#)
- [Storage Client Library 2.0 – Implementing Retry Policies](#)

General REST and retry guidelines

Consider the following when accessing Azure or third party services:

- Use a systematic approach to managing retries, perhaps as reusable code, so that you can apply a consistent methodology across all clients and all solutions.
- Consider using a retry framework such as the Transient Fault Handling Application Block to manage retries if the target service or client has no built-in retry mechanism. This will help you implement a consistent retry behavior, and it may provide a suitable default retry strategy for the target service. However, you may need to create custom retry code for services that have non-standard behavior, that do not rely on exceptions to indicate transient failures, or if you want to use a **Retry-Response** reply to manage retry behavior.
- The transient detection logic will depend on the actual client API you use to invoke the REST calls. Some clients, such as the newer **HttpClient** class, will not throw exceptions for completed requests with a non-success HTTP status code. This improves performance but prevents the use of the Transient Fault Handling Application Block. In this case you could wrap the call to the REST API with code that produces exceptions for non-success HTTP status codes, which can then be processed by the block. Alternatively, you can use a different mechanism to drive the retries.
- 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. The Transient Fault Handling Application Block cannot use the standard or any custom "retry-after" headers.
- Do not retry for status codes representing client errors (errors in the 4xx range) except for a 408 Request Timeout.
- Thoroughly test your retry strategies and mechanisms under a range of conditions, such as different network states and varying system loadings.

Retry strategies

The following are the typical types of retry strategy intervals:

- **Exponential.** A retry policy that performs a specified number of retries, using a randomized exponential back off approach to determine the interval between retries. For example:

```
var random = new Random();

var delta = (int)((Math.Pow(2.0, currentRetryCount) - 1.0) *
    random.Next((int)(this.deltaBackoff.TotalMilliseconds * 0.8),
    (int)(this.deltaBackoff.TotalMilliseconds * 1.2)));
var interval = (int)Math.Min(checked(this.minBackoff.TotalMilliseconds + delta),
    this.maxBackoff.TotalMilliseconds);
retryInterval = TimeSpan.FromMilliseconds(interval);
```

- **Incremental.** A retry strategy with a specified number of retry attempts and an incremental time interval between retries. For example:

```
retryInterval = TimeSpan.FromMilliseconds(this.initialInterval.TotalMilliseconds +
    (this.increment.TotalMilliseconds * currentRetryCount));
```

- **LinearRetry.** A retry policy that performs a specified number of retries, using a specified fixed time interval between retries. For example:

```
retryInterval = this.deltaBackoff;
```

Transient fault handling with Polly

Polly is a library to programmatically handle retries and [circuit breaker](#) strategies. The Polly project is a member of the [.NET Foundation](#). For services where the client does not natively support retries, Polly is a valid alternative and avoids the need to write custom retry code, which can be hard to implement correctly. Polly also provides a way to trace errors when they occur, so that you can log retries.

More information

- [Connection Resiliency](#)
- [Data Points - EF Core 1.1](#)

Performance antipatterns for cloud applications

6/23/2017 • 2 minutes to read • [Edit Online](#)

A *performance antipattern* is a common practice that is likely to cause scalability problems when an application is under pressure.

Here is a common scenario: An application behaves well during performance testing. It's released to production, and begins to handle real workloads. At that point, it starts to perform poorly — rejecting user requests, stalling, or throwing exceptions. The development team is then faced with two questions:

- Why didn't this behavior show up during testing?
- How do we fix it?

The answer to the first question is straightforward. It's very difficult in a test environment to simulate real users, their behavior patterns, and the volumes of work they might perform. The only completely sure way to understand how a system behaves under load is to observe it in production. To be clear, we aren't suggesting that you should skip performance testing. Performance tests are crucial for getting baseline performance metrics. But you must be prepared to observe and correct performance issues when they arise in the live system.

The answer to the second question, how to fix the problem, is less straightforward. Any number of factors might contribute, and sometimes the problem only manifests under certain circumstances. Instrumentation and logging are key to finding the root cause, but you also have to know what to look for.

Based on our engagements with Microsoft Azure customers, we've identified some of the most common performance issues that customers see in production. For each antipattern, we describe why the antipattern typically occurs, symptoms of the antipattern, and techniques for resolving the problem. We also provide sample code that illustrates both the antipattern and a suggested solution.

Some of these antipatterns may seem obvious when you read the descriptions, but they occur more often than you might think. Sometimes an application inherits a design that worked on-premises, but doesn't scale in the cloud. Or an application might start with a very clean design, but as new features are added, one or more of these antipatterns creeps in. Regardless, this guide will help you to identify and fix these antipatterns.

Here is the list of the antipatterns that we've identified:

ANTIPATTERN	DESCRIPTION
Busy Database	Offloading too much processing to a data store.
Busy Front End	Moving resource-intensive tasks onto background threads.
Chatty I/O	Continually sending many small network requests.
Extraneous Fetching	Retrieving more data than is needed, resulting in unnecessary I/O.
Improper Instantiation	Repeatedly creating and destroying objects that are designed to be shared and reused.
Monolithic Persistence	Using the same data store for data with very different usage patterns.

ANTIPATTERN	DESCRIPTION
No Caching	Failing to cache data.
Synchronous I/O	Blocking the calling thread while I/O completes.

Busy Database antipattern

6/23/2017 • 7 minutes to read • [Edit Online](#)

Offloading processing to a database server can cause it to spend a significant proportion of time running code, rather than responding to requests to store and retrieve data.

Problem description

Many database systems can run code. Examples include stored procedures and triggers. Often, it's more efficient to perform this processing close to the data, rather than transmitting the data to a client application for processing. However, overusing these features can hurt performance, for several reasons:

- The database server may spend too much time processing, rather than accepting new client requests and fetching data.
- A database is usually a shared resource, so it can become a bottleneck during periods of high use.
- Runtime costs may be excessive if the data store is metered. That's particularly true of managed database services. For example, Azure SQL Database charges for [Database Transaction Units](#) (DTUs).
- Databases have finite capacity to scale up, and it's not trivial to scale a database horizontally. Therefore, it may be better to move processing into a compute resource, such as a VM or App Service app, that can easily scale out.

This antipattern typically occurs because:

- The database is viewed as a service rather than a repository. An application might use the database server to format data (for example, converting to XML), manipulate string data, or perform complex calculations.
- Developers try to write queries whose results can be displayed directly to users. For example a query might combine fields, or format dates, times, and currency according to locale.
- Developers are trying to correct the [Extraneous Fetching](#) antipattern by pushing computations to the database.
- Stored procedures are used to encapsulate business logic, perhaps because they are considered easier to maintain and update.

The following example retrieves the 20 most valuable orders for a specified sales territory and formats the results as XML. It uses Transact-SQL functions to parse the data and convert the results to XML. You can find the complete sample [here](#).

```

SELECT TOP 20
    soh.[SalesOrderNumber] AS '@OrderNumber',
    soh.[Status] AS '@Status',
    soh.[ShipDate] AS '@ShipDate',
    YEAR(soh.[OrderDate]) AS '@OrderDateYear',
    MONTH(soh.[OrderDate]) AS '@OrderDateMonth',
    soh.[DueDate] AS '@DueDate',
    FORMAT(ROUND(soh.[SubTotal],2),'C')
        AS '@SubTotal',
    FORMAT(ROUND(soh.[TaxAmt],2),'C')
        AS '@TaxAmt',
    FORMAT(ROUND(soh.[TotalDue],2),'C')
        AS '@TotalDue',
    CASE WHEN soh.[TotalDue] > 5000 THEN 'Y' ELSE 'N' END
        AS '@ReviewRequired',
(
SELECT
    c.[AccountNumber] AS '@AccountNumber',
    UPPER(LTRIM(RTRIM(REPLACE(
        CONCAT( p.[Title], ' ', p.[FirstName], ' ', p.[MiddleName], ' ', p.[LastName], ' ', p.[Suffix]),
        ' ', ' ')))) AS '@FullName'
FROM [Sales].[Customer] c
INNER JOIN [Person].[Person] p
ON c.[PersonID] = p.[BusinessEntityID]
WHERE c.[CustomerID] = soh.[CustomerID]
FOR XML PATH ('Customer'), TYPE
),
(
SELECT
    sod.[OrderQty] AS '@Quantity',
    FORMAT(sod.[UnitPrice],'C')
        AS '@UnitPrice',
    FORMAT(ROUND(sod.[LineTotal],2),'C')
        AS '@LineTotal',
    sod.[ProductID] AS '@ProductId',
    CASE WHEN (sod.[ProductID] >= 710) AND (sod.[ProductID] <= 720) AND (sod.[OrderQty] >= 5) THEN 'Y' ELSE
    'N' END
        AS '@InventoryCheckRequired'

    FROM [Sales].[SalesOrderDetail] sod
    WHERE sod.[SalesOrderID] = soh.[SalesOrderID]
    ORDER BY sod.[SalesOrderDetailID]
    FOR XML PATH ('LineItem'), TYPE, ROOT('OrderLineItems')
)
FROM [Sales].[SalesOrderHeader] soh
WHERE soh.[TerritoryId] = @TerritoryId
ORDER BY soh.[TotalDue] DESC
FOR XML PATH ('Order'), ROOT('Orders')

```

Clearly, this is complex query. As we'll see later, it turns out to use significant processing resources on the database server.

How to fix the problem

Move processing from the database server into other application tiers. Ideally, you should limit the database to performing data access operations, using only the capabilities that the database is optimized for, such as aggregation in an RDBMS.

For example, the previous Transact-SQL code can be replaced with a statement that simply retrieves the data to be processed.

```

SELECT
soh.[SalesOrderNumber] AS [OrderNumber],
soh.[Status] AS [Status],
soh.[OrderDate] AS [OrderDate],
soh.[DueDate] AS [DueDate],
soh.[ShipDate] AS [ShipDate],
soh.[SubTotal] AS [SubTotal],
soh.[TaxAmt] AS [TaxAmt],
soh.[TotalDue] AS [TotalDue],
c.[AccountNumber] AS [AccountNumber],
p.[Title] AS [CustomerTitle],
p.[FirstName] AS [CustomerFirstName],
p.[MiddleName] AS [CustomerMiddleName],
p.[LastName] AS [CustomerLastName],
p.[Suffix] AS [CustomerSuffix],
sod.[OrderQty] AS [Quantity],
sod.[UnitPrice] AS [UnitPrice],
sod.[LineTotal] AS [LineTotal],
sod.[ProductID] AS [ProductId]
FROM [Sales].[SalesOrderHeader] soh
INNER JOIN [Sales].[Customer] c ON soh.[CustomerID] = c.[CustomerID]
INNER JOIN [Person].[Person] p ON c.[PersonID] = p.[BusinessEntityID]
INNER JOIN [Sales].[SalesOrderDetail] sod ON soh.[SalesOrderID] = sod.[SalesOrderID]
WHERE soh.[TerritoryId] = @TerritoryId
AND soh.[SalesOrderId] IN (
    SELECT TOP 20 SalesOrderId
    FROM [Sales].[SalesOrderHeader] soh
    WHERE soh.[TerritoryId] = @TerritoryId
    ORDER BY soh.[TotalDue] DESC)
ORDER BY soh.[TotalDue] DESC, sod.[SalesOrderDetailID]

```

The application then uses the .NET Framework `System.Xml.Linq` APIs to format the results as XML.

```

// Create a new SqlCommand to run the Transact-SQL query
using (var command = new SqlCommand(...))
{
    command.Parameters.AddWithValue("@TerritoryId", id);

    // Run the query and create the initial XML document
    using (var reader = await command.ExecuteReaderAsync())
    {
        var lastOrderNumber = string.Empty;
        var doc = new XDocument();
        var orders = new XElement("Orders");
        doc.Add(orders);

        XElement lineItems = null;
        // Fetch each row in turn, format the results as XML, and add them to the XML document
        while (await reader.ReadAsync())
        {
            var orderNumber = reader["OrderNumber"].ToString();
            if (orderNumber != lastOrderNumber)
            {
                lastOrderNumber = orderNumber;

                var order = new XElement("Order");
                orders.Add(order);
                var customer = new XElement("Customer");
                lineItems = new XElement("OrderLineItems");
                order.Add(customer, lineItems);

                var orderDate = (DateTime)reader["OrderDate"];
                var totalDue = (Decimal)reader["TotalDue"];
                var reviewRequired = totalDue > 5000 ? 'Y' : 'N';

                order.Add(

```

```

        new XAttribute("OrderNumber", orderNumber),
        new XAttribute("Status", reader["Status"]),
        new XAttribute("ShipDate", reader["ShipDate"]),
        ... // More attributes, not shown.

        var fullName = string.Join(" ",
            reader["CustomerTitle"],
            reader["CustomerFirstName"],
            reader["CustomerMiddleName"],
            reader["CustomerLastName"],
            reader["CustomerSuffix"]
        )
        .Replace(" ", " ") //remove double spaces
        .Trim()
        .ToUpper();

        customer.Add(
            new XAttribute("AccountNumber", reader["AccountNumber"]),
            new XAttribute("FullName", fullName));
    }

    var productId = (int)reader["ProductID"];
    var quantity = (short)reader["Quantity"];
    var inventoryCheckRequired = (productId >= 710 && productId <= 720 && quantity >= 5) ? 'Y' : 'N';

    lineItems.Add(
        new XElement("LineItem",
            new XAttribute("Quantity", quantity),
            new XAttribute("UnitPrice", ((Decimal)reader["UnitPrice"]).ToString("C")),
            new XAttribute("LineTotal", RoundAndFormat(reader["LineTotal"])),
            new XAttribute("ProductId", productId),
            new XAttribute("InventoryCheckRequired", inventoryCheckRequired)
        ));
}
// Match the exact formatting of the XML returned from SQL
var xml = doc
    .ToString(SaveOptions.DisableFormatting)
    .Replace(">>", ">");
}
}

```

NOTE

This code is somewhat complex. For a new application, you might prefer to use a serialization library. However, the assumption here is that the development team is refactoring an existing application, so the method needs to return the exact same format as the original code.

Considerations

- Many database systems are highly optimized to perform certain types of data processing, such as calculating aggregate values over large datasets. Don't move those types of processing out of the database.
- Do not relocate processing if doing so causes the database to transfer far more data over the network. See the [Extraneous Fetching antipattern](#).
- If you move processing to an application tier, that tier may need to scale out to handle the additional work.

How to detect the problem

Symptoms of a busy database include a disproportionate decline in throughput and response times in operations that access the database.

You can perform the following steps to help identify this problem:

1. Use performance monitoring to identify how much time the production system spends performing database activity.
2. Examine the work performed by the database during these periods.
3. If you suspect that particular operations might cause too much database activity, perform load testing in a controlled environment. Each test should run a mixture of the suspect operations with a variable user load. Examine the telemetry from the load tests to observe how the database is used.
4. If the database activity reveals significant processing but little data traffic, review the source code to determine whether the processing can better be performed elsewhere.

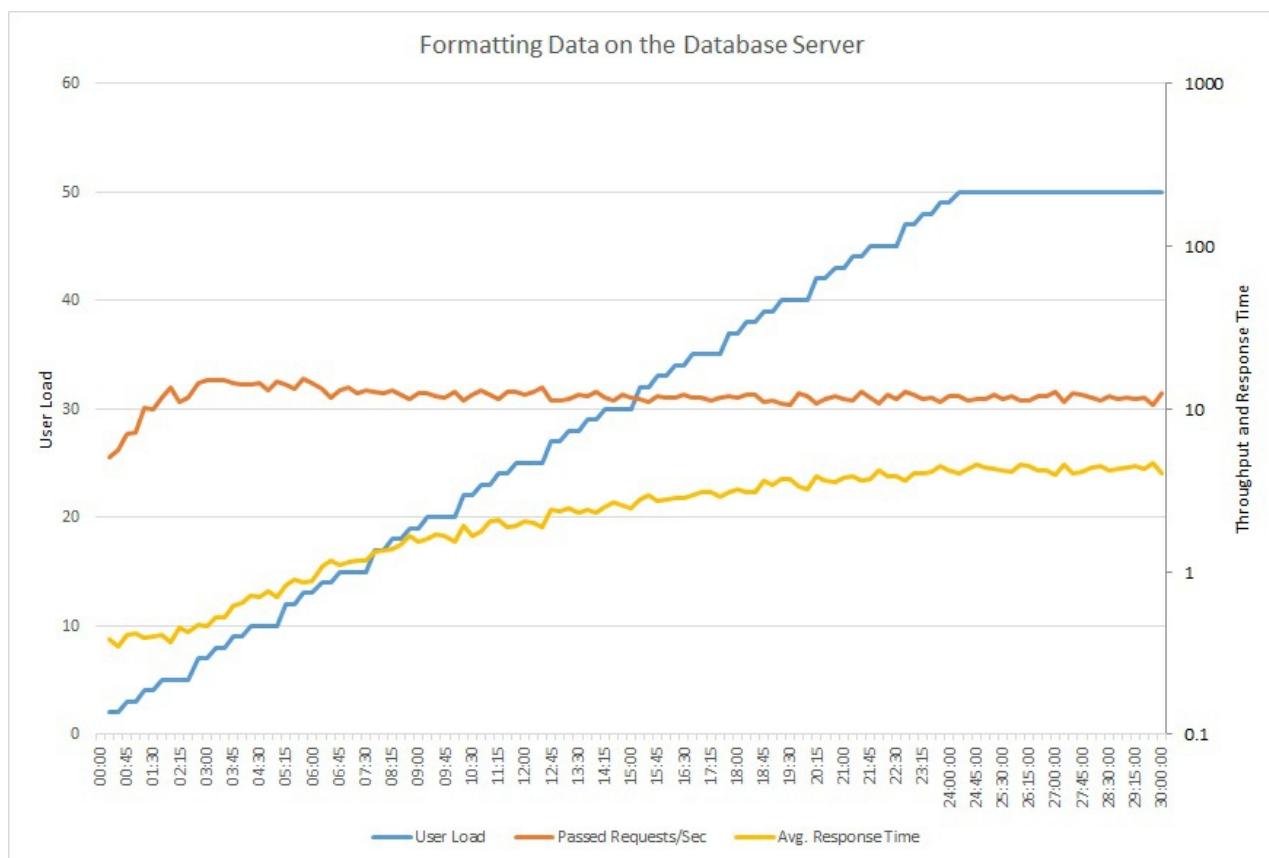
If the volume of database activity is low or response times are relatively fast, then a busy database is unlikely to be a performance problem.

Example diagnosis

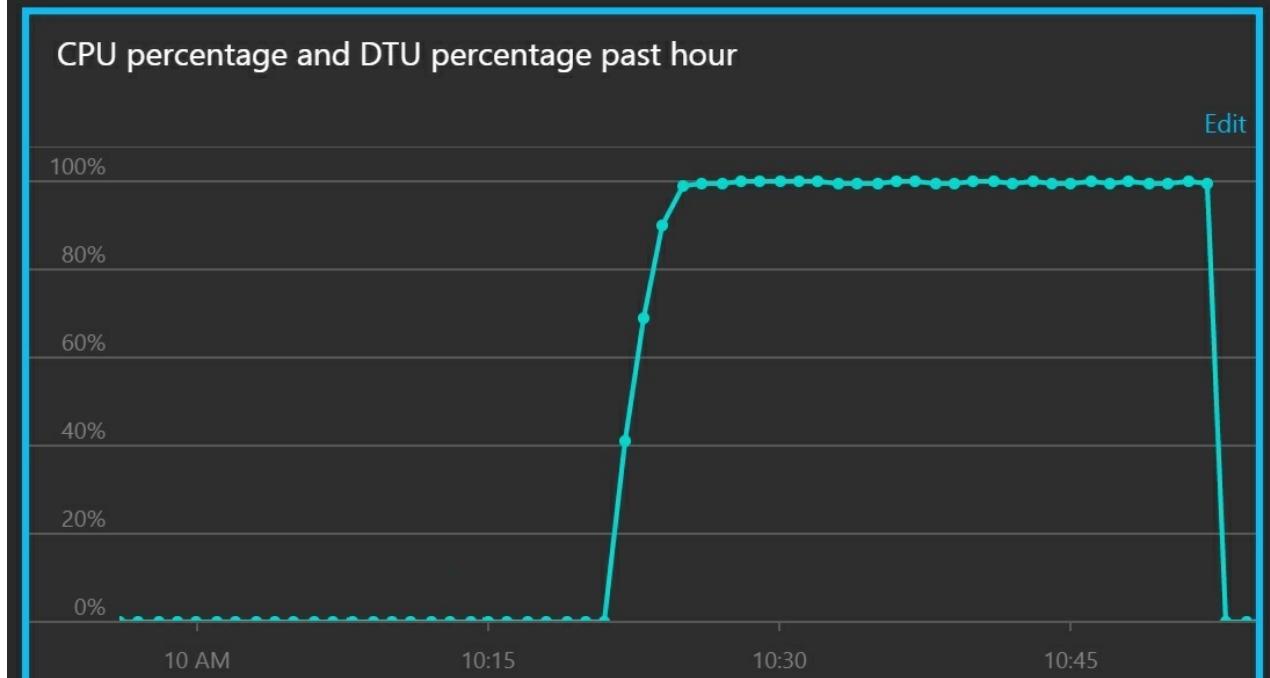
The following sections apply these steps to the sample application described earlier.

Monitor the volume of database activity

The following graph shows the results of running a load test against the sample application, using a step load of up to 50 concurrent users. The volume of requests quickly reaches a limit and stays at that level, while the average response time steadily increases. Note that a logarithmic scale is used for those two metrics.



The next graph shows CPU utilization and DTUs as a percentage of service quota. DTUs provides a measure of how much processing the database performs. The graph shows that CPU and DTU utilization both quickly reached 100%.



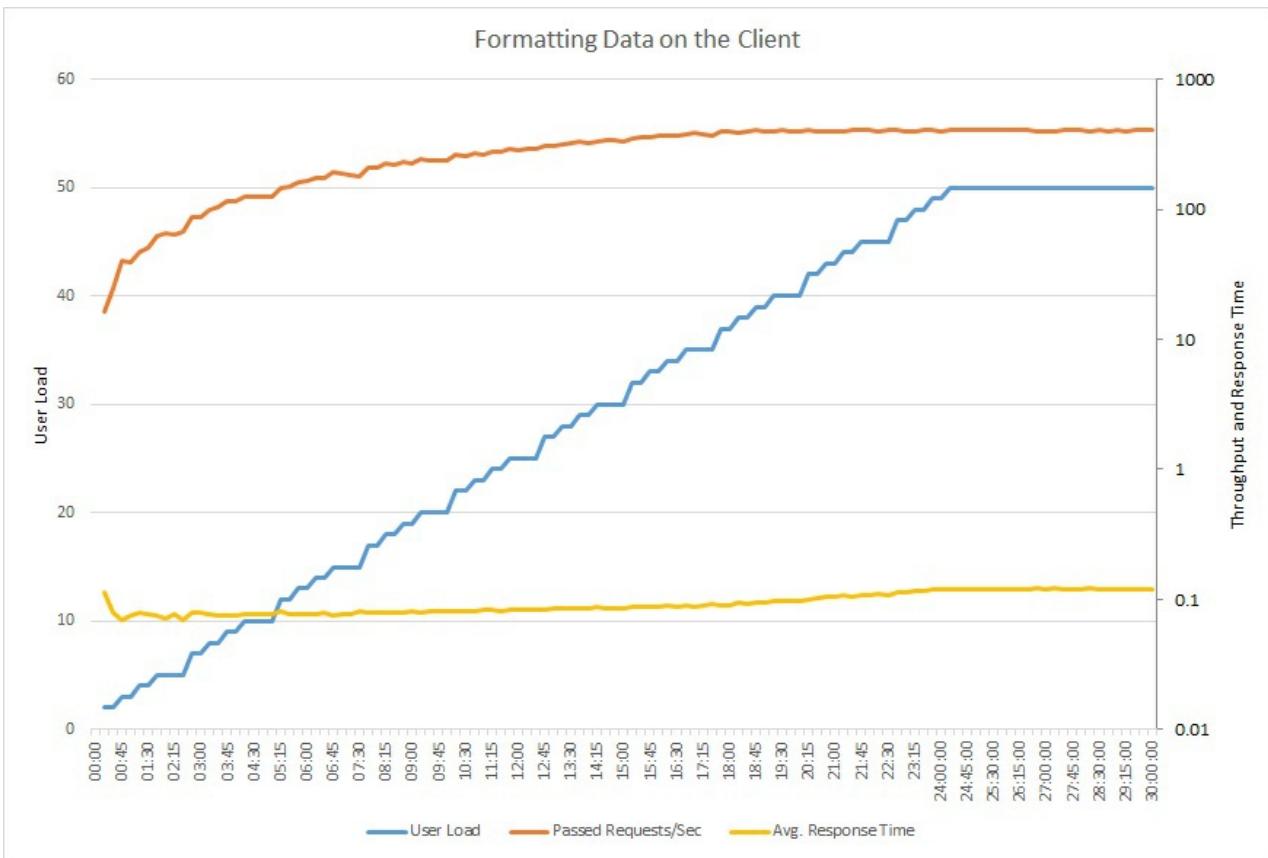
Examine the work performed by the database

It could be that the tasks performed by the database are genuine data access operations, rather than processing, so it is important to understand the SQL statements being run while the database is busy. Monitor the system to capture the SQL traffic and correlate the SQL operations with application requests.

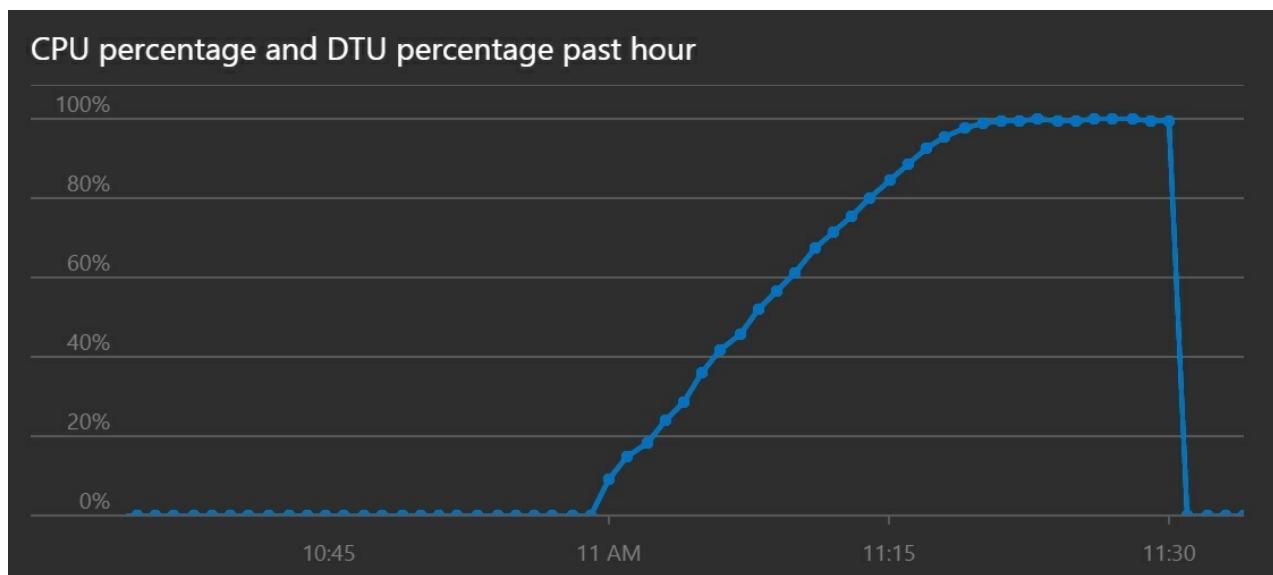
If the database operations are purely data access operations, without a lot of processing, then the problem might be [Extraneous Fetching](#).

Implement the solution and verify the result

The following graph shows a load test using the updated code. Throughput is significantly higher, over 400 requests per second versus 12 earlier. The average response time is also much lower, just above 0.1 seconds compared to over 4 seconds.



CPU and DTU utilization shows that the system took longer to reach saturation, despite the increased throughput.



Related resources

- [Extraneous Fetching antipattern](#)

Busy Front End antipattern

8/22/2017 • 8 minutes to read • [Edit Online](#)

Performing asynchronous work on a large number of background threads can starve other concurrent foreground tasks of resources, decreasing response times to unacceptable levels.

Problem description

Resource-intensive tasks can increase the response times for user requests and cause high latency. One way to improve response times is to offload a resource-intensive task to a separate thread. This approach lets the application stay responsive while processing happens in the background. However, tasks that run on a background thread still consume resources. If there are too many of them, they can starve the threads that are handling requests.

NOTE

The term *resource* can encompass many things, such as CPU utilization, memory occupancy, and network or disk I/O.

This problem typically occurs when an application is developed as monolithic piece of code, with all of the business logic combined into a single tier shared with the presentation layer.

Here's an example using ASP.NET that demonstrates the problem. You can find the complete sample [here](#).

```
public class WorkInFrontEndController : ApiController
{
    [HttpPost]
    [Route("api/workinfrontend")]
    public HttpResponseMessage Post()
    {
        new Thread(() =>
        {
            //Simulate processing
            Thread.Sleep(Int32.MaxValue / 100);
        }).Start();

        return Request.CreateResponse(HttpStatusCode.Accepted);
    }
}

public class UserProfileController : ApiController
{
    [HttpGet]
    [Route("api/userprofile/{id}")]
    public UserProfile Get(int id)
    {
        //Simulate processing
        return new UserProfile() { FirstName = "Alton", LastName = "Hudgens" };
    }
}
```

- The `Post` method in the `WorkInFrontEnd` controller implements an HTTP POST operation. This operation simulates a long-running, CPU-intensive task. The work is performed on a separate thread, in an attempt to enable the POST operation to complete quickly.
- The `Get` method in the `UserProfile` controller implements an HTTP GET operation. This method is much

less CPU intensive.

The primary concern is the resource requirements of the `Post` method. Although it puts the work onto a background thread, the work can still consume considerable CPU resources. These resources are shared with other operations being performed by other concurrent users. If a moderate number of users send this request at the same time, overall performance is likely to suffer, slowing down all operations. Users might experience significant latency in the `Get` method, for example.

How to fix the problem

Move processes that consume significant resources to a separate back end.

With this approach, the front end puts resource-intensive tasks onto a message queue. The back end picks up the tasks for asynchronous processing. The queue also acts as a load leveler, buffering requests for the back end. If the queue length becomes too long, you can configure autoscaling to scale out the back end.

Here is a revised version of the previous code. In this version, the `Post` method puts a message on a Service Bus queue.

```
public class WorkInBackgroundController : ApiController
{
    private static readonly QueueClient QueueClient;
    private static readonly string QueueName;
    private static readonly ServiceBusQueueHandler ServiceBusQueueHandler;

    public WorkInBackgroundController()
    {
        var serviceBusConnectionString = ...;
        QueueName = ...;
        ServiceBusQueueHandler = new ServiceBusQueueHandler(serviceBusConnectionString);
        QueueClient = ServiceBusQueueHandler.GetQueueClientAsync(QueueName).Result;
    }

    [HttpPost]
    [Route("api/workinbackground")]
    public async Task<long> Post()
    {
        return await ServiceBusQueueHandler.AddWorkLoadToQueueAsync(QueueClient, QueueName, 0);
    }
}
```

The back end pulls messages from the Service Bus queue and does the processing.

```

public async Task RunAsync(CancellationToken cancellationToken)
{
    this._queueClient.OnMessageAsync(
        // This lambda is invoked for each message received.
        async (receivedMessage) =>
    {
        try
        {
            // Simulate processing of message
            Thread.Sleep(Int32.MaxValue / 1000);

            await receivedMessage.CompleteAsync();
        }
        catch
        {
            receivedMessage.Abandon();
        }
    });
}

```

Considerations

- This approach adds some additional complexity to the application. You must handle queuing and dequeuing safely to avoid losing requests in the event of a failure.
- The application takes a dependency on an additional service for the message queue.
- The processing environment must be sufficiently scalable to handle the expected workload and meet the required throughput targets.
- While this approach should improve overall responsiveness, the tasks that are moved to the back end may take longer to complete.

How to detect the problem

Symptoms of a busy front end include high latency when resource-intensive tasks are being performed. End users are likely to report extended response times or failures caused by services timing out. These failures could also return HTTP 500 (Internal Server) errors or HTTP 503 (Service Unavailable) errors. Examine the event logs for the web server, which are likely to contain more detailed information about the causes and circumstances of the errors.

You can perform the following steps to help identify this problem:

1. Perform process monitoring of the production system, to identify points when response times slow down.
2. Examine the telemetry data captured at these points to determine the mix of operations being performed and the resources being used.
3. Find any correlations between poor response times and the volumes and combinations of operations that were happening at those times.
4. Load test each suspected operation to identify which operations are consuming resources and starving other operations.
5. Review the source code for those operations to determine why they might cause excessive resource consumption.

Example diagnosis

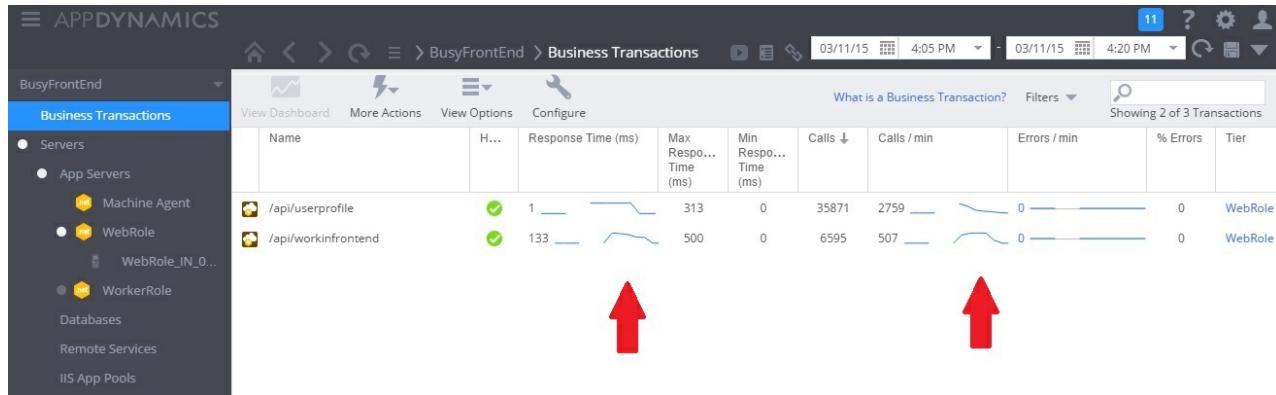
The following sections apply these steps to the sample application described earlier.

Identify points of slowdown

Instrument each method to track the duration and resources consumed by each request. Then monitor the application in production. This can provide an overall view of how requests compete with each other. During

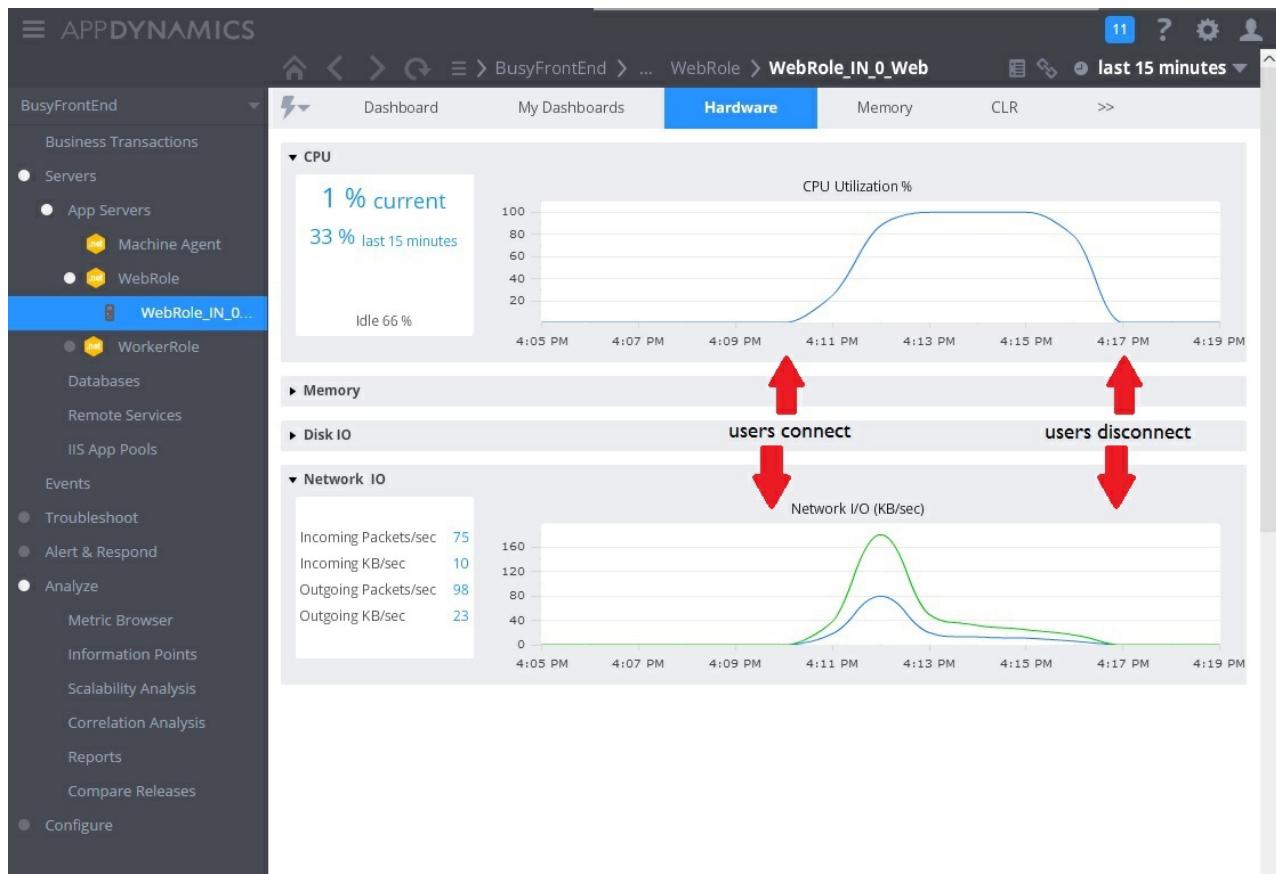
periods of stress, slow-running resource-hungry requests will likely impact other operations, and this behavior can be observed by monitoring the system and noting the drop off in performance.

The following image shows a monitoring dashboard. (We used [AppDynamics] for our tests.) Initially, the system has light load. Then users start requesting the `UserProfile` GET method. The performance is reasonably good until other users start issuing requests to the `WorkInFrontEnd` POST method. At that point, response times increase dramatically (first arrow). Response times only improve after the volume of requests to the `WorkInFrontEnd` controller diminishes (second arrow).



Examine telemetry data and find correlations

The next image shows some of the metrics gathered to monitor resource utilization during the same interval. At first, few users are accessing the system. As more users connect, CPU utilization becomes very high (100%). Also notice that the network I/O rate initially goes up as CPU usage rises. But once CPU usage peaks, network I/O actually goes down. That's because the system can only handle a relatively small number of requests once the CPU is at capacity. As users disconnect, the CPU load tails off.

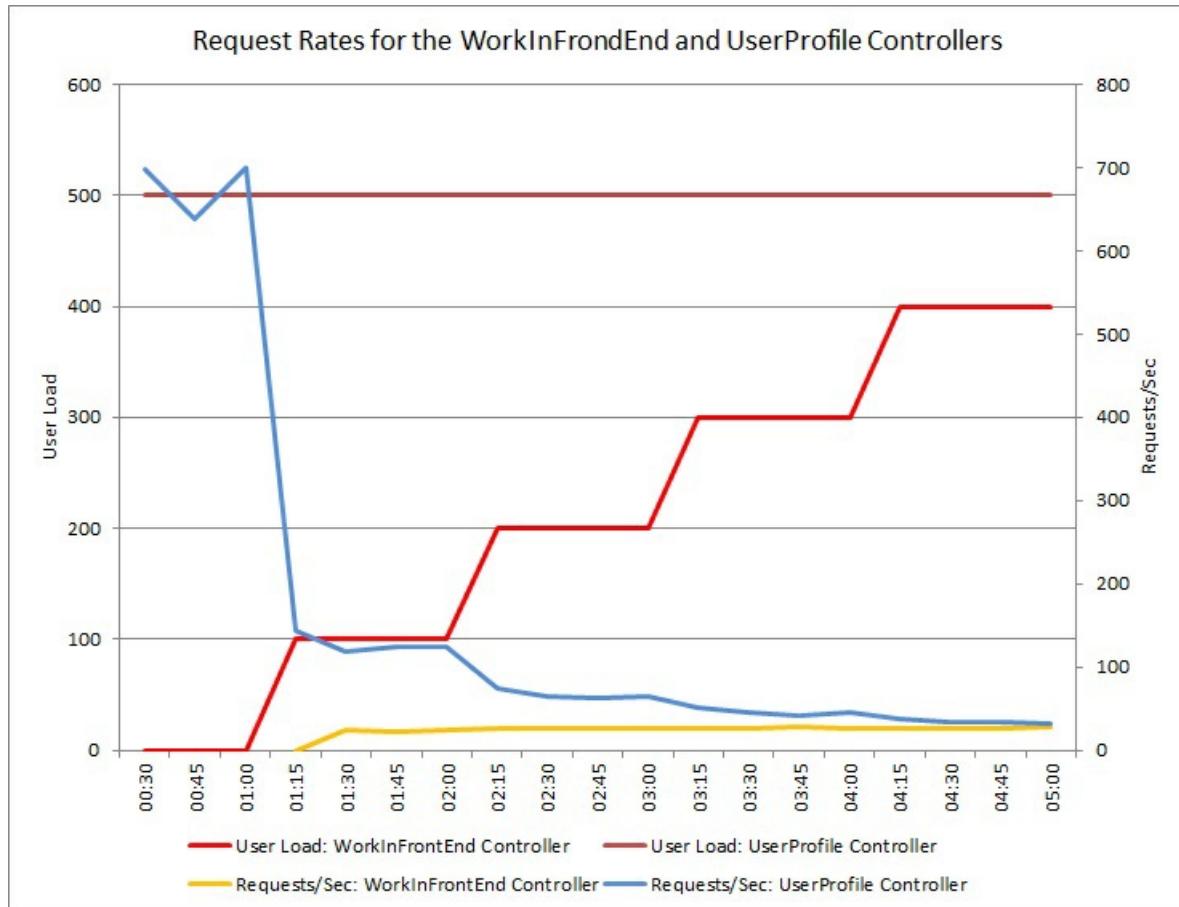


At this point, it appears the `Post` method in the `WorkInFrontEnd` controller is a prime candidate for closer examination. Further work in a controlled environment is needed to confirm the hypothesis.

Perform load testing

The next step is to perform tests in a controlled environment. For example, run a series of load tests that include and then omit each request in turn to see the effects.

The graph below shows the results of a load test performed against an identical deployment of the cloud service used in the previous tests. The test used a constant load of 500 users performing the `Get` operation in the `UserProfile` controller, along with a step load of users performing the `Post` operation in the `WorkInFrontEnd` controller.



Initially, the step load is 0, so the only active users are performing the `UserProfile` requests. The system is able to respond to approximately 500 requests per second. After 60 seconds, a load of 100 additional users starts sending POST requests to the `WorkInFrontEnd` controller. Almost immediately, the workload sent to the `UserProfile` controller drops to about 150 requests per second. This is due to the way the load-test runner functions. It waits for a response before sending the next request, so the longer it takes to receive a response, the lower the request rate.

As more users send POST requests to the `WorkInFrontEnd` controller, the response rate of the `UserProfile` controller continues to drop. But note that the volume of requests handled by the `WorkInFrontEnd` controller remains relatively constant. The saturation of the system becomes apparent as the overall rate of both requests tends towards a steady but low limit.

Review the source code

The final step is to look at the source code. The development team was aware that the `Post` method could take a considerable amount of time, which is why the original implementation used a separate thread. That solved the immediate problem, because the `Post` method did not block waiting for a long-running task to complete.

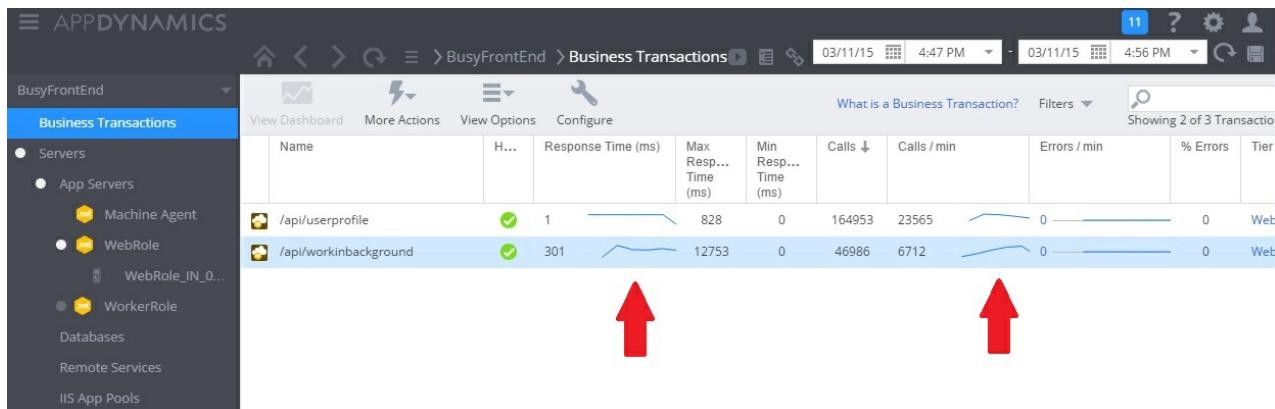
However, the work performed by this method still consumes CPU, memory, and other resources. Enabling this process to run asynchronously might actually damage performance, as users can trigger a large number of these operations simultaneously, in an uncontrolled manner. There is a limit to the number of threads that a server can run. Past this limit, the application is likely to get an exception when it tries to start a new thread.

NOTE

This doesn't mean you should avoid asynchronous operations. Performing an asynchronous await on a network call is a recommended practice. (See the [Synchronous I/O antipattern](#).) The problem here is that CPU-intensive work was spawned on another thread.

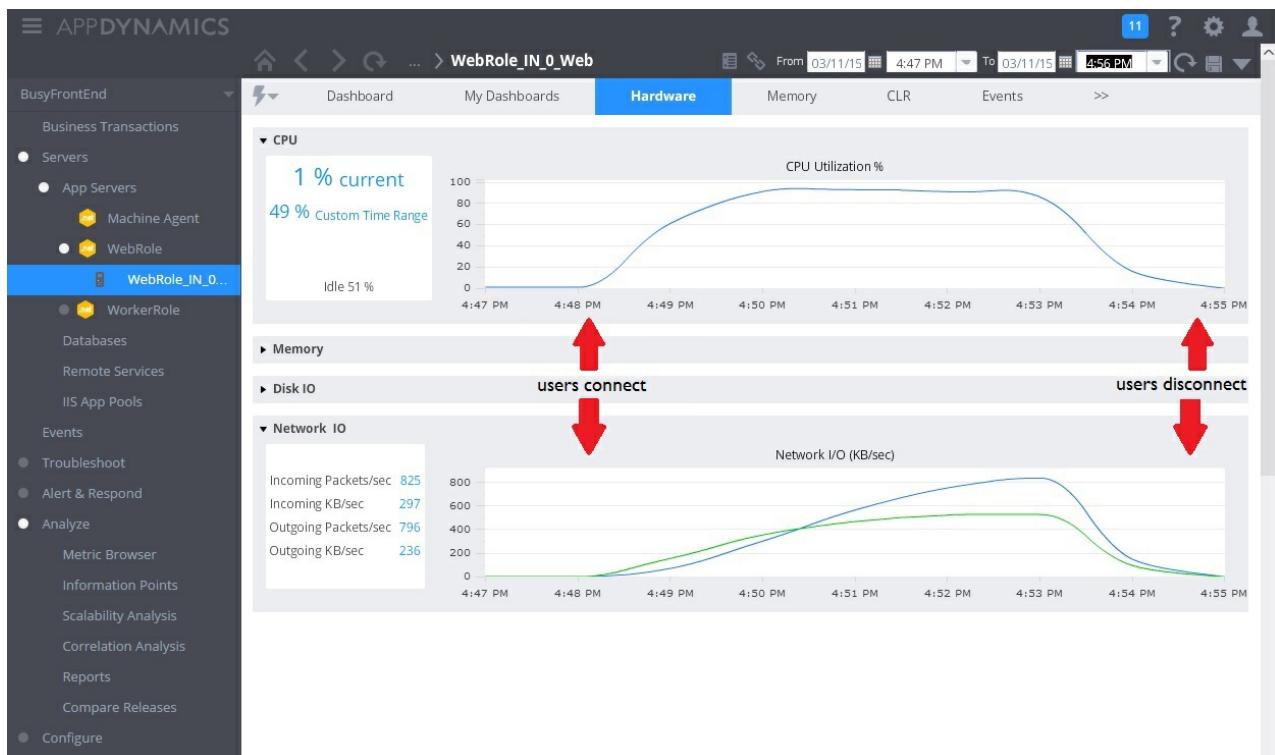
Implement the solution and verify the result

The following image shows performance monitoring after the solution was implemented. The load was similar to that shown earlier, but the response times for the `UserProfile` controller are now much faster. The volume of requests increased over the same duration, from 2,759 to 23,565.

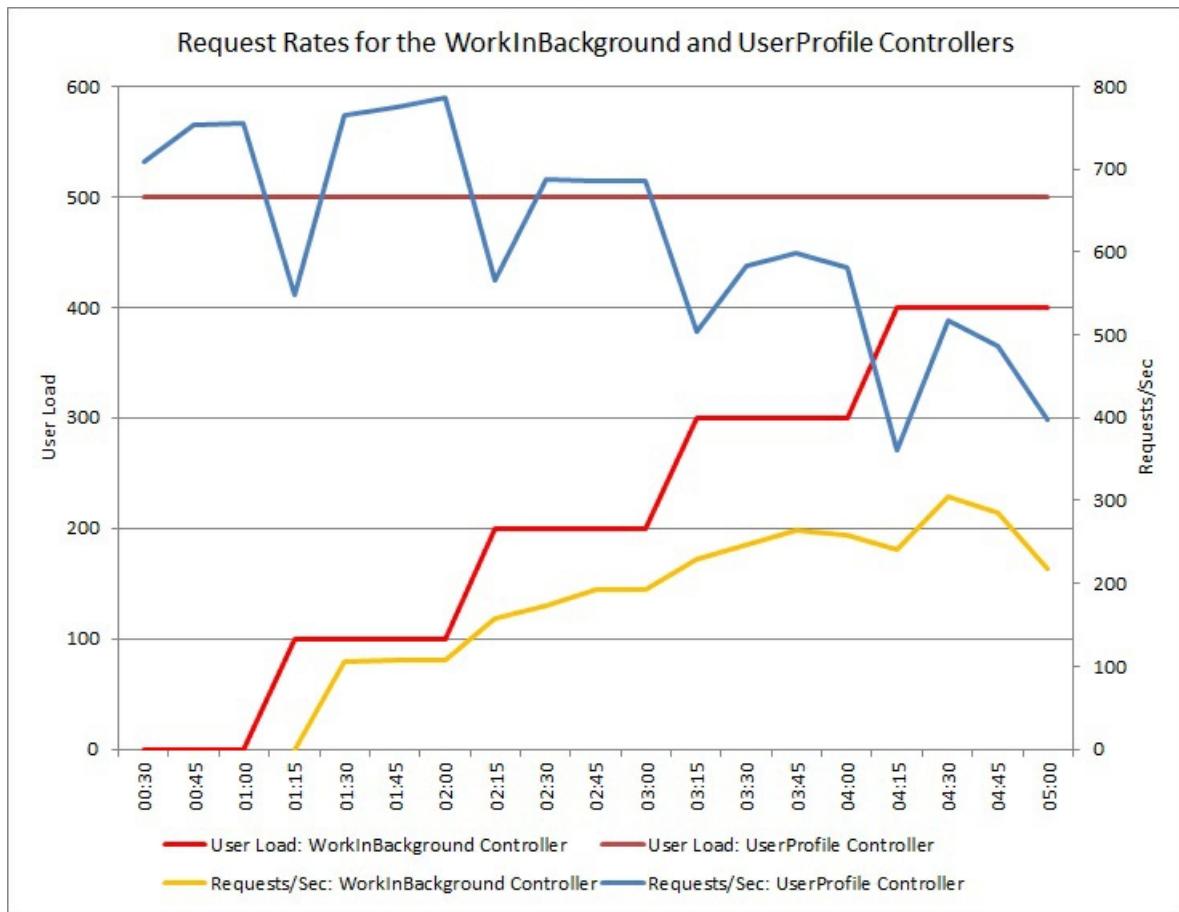


Note that the `WorkInBackground` controller also handled a much larger volume of requests. However, you can't make a direct comparison in this case, because the work being performed in this controller is very different from the original code. The new version simply queues a request, rather than performing a time consuming calculation. The main point is that this method no longer drags down the entire system under load.

CPU and network utilization also show the improved performance. The CPU utilization never reached 100%, and the volume of handled network requests was far greater than earlier, and did not tail off until the workload dropped.



The following graph shows the results of a load test. The overall volume of requests serviced is greatly improved compared to the the earlier tests.



Related guidance

- [Autoscaling best practices](#)
- [Background jobs best practices](#)
- [Queue-Based Load Leveling pattern](#)
- [Web Queue Worker architecture style](#)

Chatty I/O antipattern

2/22/2018 • 9 minutes to read • [Edit Online](#)

The cumulative effect of a large number of I/O requests can have a significant impact on performance and responsiveness.

Problem description

Network calls and other I/O operations are inherently slow compared to compute tasks. Each I/O request typically has significant overhead, and the cumulative effect of numerous I/O operations can slow down the system. Here are some common causes of chatty I/O.

Reading and writing individual records to a database as distinct requests

The following example reads from a database of products. There are three tables, `Product`, `ProductSubcategory`, and `ProductPriceListHistory`. The code retrieves all of the products in a subcategory, along with the pricing information, by executing a series of queries:

1. Query the subcategory from the `ProductSubcategory` table.
2. Find all products in that subcategory by querying the `Product` table.
3. For each product, query the pricing data from the `ProductPriceListHistory` table.

The application uses [Entity Framework](#) to query the database. You can find the complete sample [here](#).

```
public async Task<IHttpActionResult> GetProductsInSubCategoryAsync(int subcategoryId)
{
    using (var context = GetContext())
    {
        // Get product subcategory.
        var productSubcategory = await context.ProductSubcategories
            .Where(psc => psc.ProductSubcategoryId == subcategoryId)
            .FirstOrDefaultAsync();

        // Find products in that category.
        productSubcategory.Product = await context.Products
            .Where(p => subcategoryId == p.ProductSubcategoryId)
            .ToListAsync();

        // Find price history for each product.
        foreach (var prod in productSubcategory.Product)
        {
            int productId = prod.ProductId;
            var productListPriceHistory = await context.ProductListPriceHistory
                .Where(pl => pl.ProductId == productId)
                .ToListAsync();
            prod.ProductListPriceHistory = productListPriceHistory;
        }
        return Ok(productSubcategory);
    }
}
```

This example shows the problem explicitly, but sometimes an O/RM can mask the problem, if it implicitly fetches child records one at a time. This is known as the "N+1 problem".

Implementing a single logical operation as a series of HTTP requests

This often happens when developers try to follow an object-oriented paradigm, and treat remote objects as if they

were local objects in memory. This can result in too many network round trips. For example, the following web API exposes the individual properties of `User` objects through individual HTTP GET methods.

```
public class UserController : ApiController
{
    [HttpGet]
    [Route("users/{id:int}/username")]
    public HttpResponseMessage GetUserName(int id)
    {
        ...
    }

    [HttpGet]
    [Route("users/{id:int}/gender")]
    public HttpResponseMessage GetGender(int id)
    {
        ...
    }

    [HttpGet]
    [Route("users/{id:int}/dateofbirth")]
    public HttpResponseMessage GetDateOfBirth(int id)
    {
        ...
    }
}
```

While there's nothing technically wrong with this approach, most clients will probably need to get several properties for each `User`, resulting in client code like the following.

```
HttpResponseMessage response = await client.GetAsync("users/1/username");
response.EnsureSuccessStatusCode();
var userName = await response.Content.ReadAsStringAsync();

response = await client.GetAsync("users/1/gender");
response.EnsureSuccessStatusCode();
var gender = await response.Content.ReadAsStringAsync();

response = await client.GetAsync("users/1/dateofbirth");
response.EnsureSuccessStatusCode();
var dob = await response.Content.ReadAsStringAsync();
```

Reading and writing to a file on disk

File I/O involves opening a file and moving to the appropriate point before reading or writing data. When the operation is complete, the file might be closed to save operating system resources. An application that continually reads and writes small amounts of information to a file will generate significant I/O overhead. Small write requests can also lead to file fragmentation, slowing subsequent I/O operations still further.

The following example uses a `FileStream` to write a `Customer` object to a file. Creating the `FileStream` opens the file, and disposing it closes the file. (The `using` statement automatically disposes the `FileStream` object.) If the application calls this method repeatedly as new customers are added, the I/O overhead can accumulate quickly.

```

private async Task SaveCustomerToFileAsync(Customer cust)
{
    using (Stream fileStream = new FileStream(CustomersFileName, FileMode.Append))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        byte [] data = null;
        using (MemoryStream memStream = new MemoryStream())
        {
            formatter.Serialize(memStream, cust);
            data = memStream.ToArray();
        }
        await fileStream.WriteAsync(data, 0, data.Length);
    }
}

```

How to fix the problem

Reduce the number of I/O requests by packaging the data into larger, fewer requests.

Fetch data from a database as a single query, instead of several smaller queries. Here's a revised version of the code that retrieves product information.

```

public async Task<IHttpActionResult> GetProductCategoryDetailsAsync(int subCategoryId)
{
    using (var context = GetContext())
    {
        var subCategory = await context.ProductSubcategories
            .Where(psc => psc.ProductSubcategoryId == subCategoryId)
            .Include("Product.ProductListPriceHistory")
            .FirstOrDefaultAsync();

        if (subCategory == null)
            return NotFound();

        return Ok(subCategory);
    }
}

```

Follow REST design principles for web APIs. Here's a revised version of the web API from the earlier example. Instead of separate GET methods for each property, there is a single GET method that returns the `User`. This results in a larger response body per request, but each client is likely to make fewer API calls.

```

public class UserController : ApiController
{
    [HttpGet]
    [Route("users/{id:int}")]
    public HttpResponseMessage GetUser(int id)
    {
        ...
    }
}

// Client code
HttpResponseMessage response = await client.GetAsync("users/1");
response.EnsureSuccessStatusCode();
var user = await response.Content.ReadAsStringAsync();

```

For file I/O, consider buffering data in memory and then writing the buffered data to a file as a single operation. This approach reduces the overhead from repeatedly opening and closing the file, and helps to reduce fragmentation of the file on disk.

```

// Save a list of customer objects to a file
private async Task SaveCustomerListToFileAsync(List<Customer> customers)
{
    using (Stream fileStream = new FileStream(CustomersFileName, FileMode.Append))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        foreach (var cust in customers)
        {
            byte[] data = null;
            using (MemoryStream memStream = new MemoryStream())
            {
                formatter.Serialize(memStream, cust);
                data = memStream.ToArray();
            }
            await fileStream.WriteAsync(data, 0, data.Length);
        }
    }
}

// In-memory buffer for customers.
List<Customer> customers = new List<Customers>();

// Create a new customer and add it to the buffer
var cust = new Customer(...);
customers.Add(cust);

// Add more customers to the list as they are created
...

// Save the contents of the list, writing all customers in a single operation
await SaveCustomerListToFileAsync(customers);

```

Considerations

- The first two examples make *fewer* I/O calls, but each one retrieves *more* information. You must consider the tradeoff between these two factors. The right answer will depend on the actual usage patterns. For example, in the web API example, it might turn out that clients often need just the user name. In that case, it might make sense to expose it as a separate API call. For more information, see the [Extraneous Fetching](#) antipattern.
- When reading data, do not make your I/O requests too large. An application should only retrieve the information that it is likely to use.
- Sometimes it helps to partition the information for an object into two chunks, *frequently accessed data* that accounts for most requests, and *less frequently accessed data* that is used rarely. Often the most frequently accessed data is a relatively small portion of the total data for an object, so returning just that portion can save significant I/O overhead.
- When writing data, avoid locking resources for longer than necessary, to reduce the chances of contention during a lengthy operation. If a write operation spans multiple data stores, files, or services, then adopt an eventually consistent approach. See [Data Consistency guidance](#).
- If you buffer data in memory before writing it, the data is vulnerable if the process crashes. If the data rate typically has bursts or is relatively sparse, it may be safer to buffer the data in an external durable queue such as [Event Hubs](#).
- Consider caching data that you retrieve from a service or a database. This can help to reduce the volume of I/O by avoiding repeated requests for the same data. For more information, see [Caching best practices](#).

How to detect the problem

Symptoms of chatty I/O include high latency and low throughput. End users are likely to report extended response times or failures caused by services timing out, due to increased contention for I/O resources.

You can perform the following steps to help identify the causes of any problems:

1. Perform process monitoring of the production system to identify operations with poor response times.
2. Perform load testing of each operation identified in the previous step.
3. During the load tests, gather telemetry data about the data access requests made by each operation.
4. Gather detailed statistics for each request sent to a data store.
5. Profile the application in the test environment to establish where possible I/O bottlenecks might be occurring.

Look for any of these symptoms:

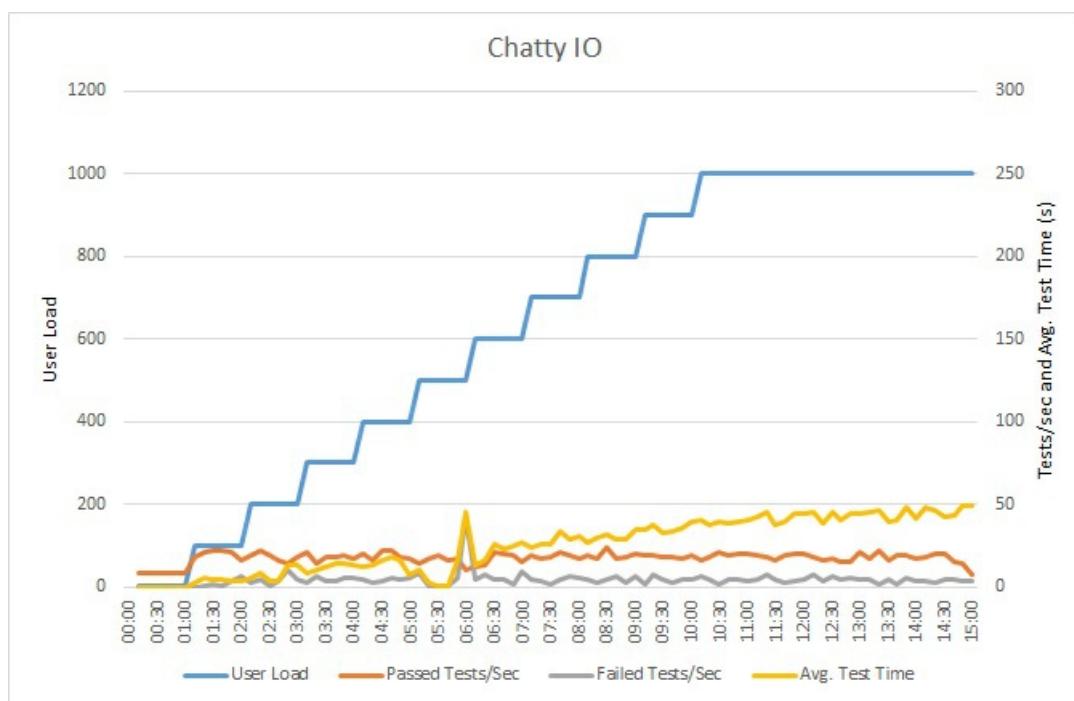
- A large number of small I/O requests made to the same file.
- A large number of small network requests made by an application instance to the same service.
- A large number of small requests made by an application instance to the same data store.
- Applications and services becoming I/O bound.

Example diagnosis

The following sections apply these steps to the example shown earlier that queries a database.

Load test the application

This graph shows the results of load testing. Median response time is measured in 10s of seconds per request. The graph shows very high latency. With a load of 1000 users, a user might have to wait for nearly a minute to see the results of a query.



NOTE

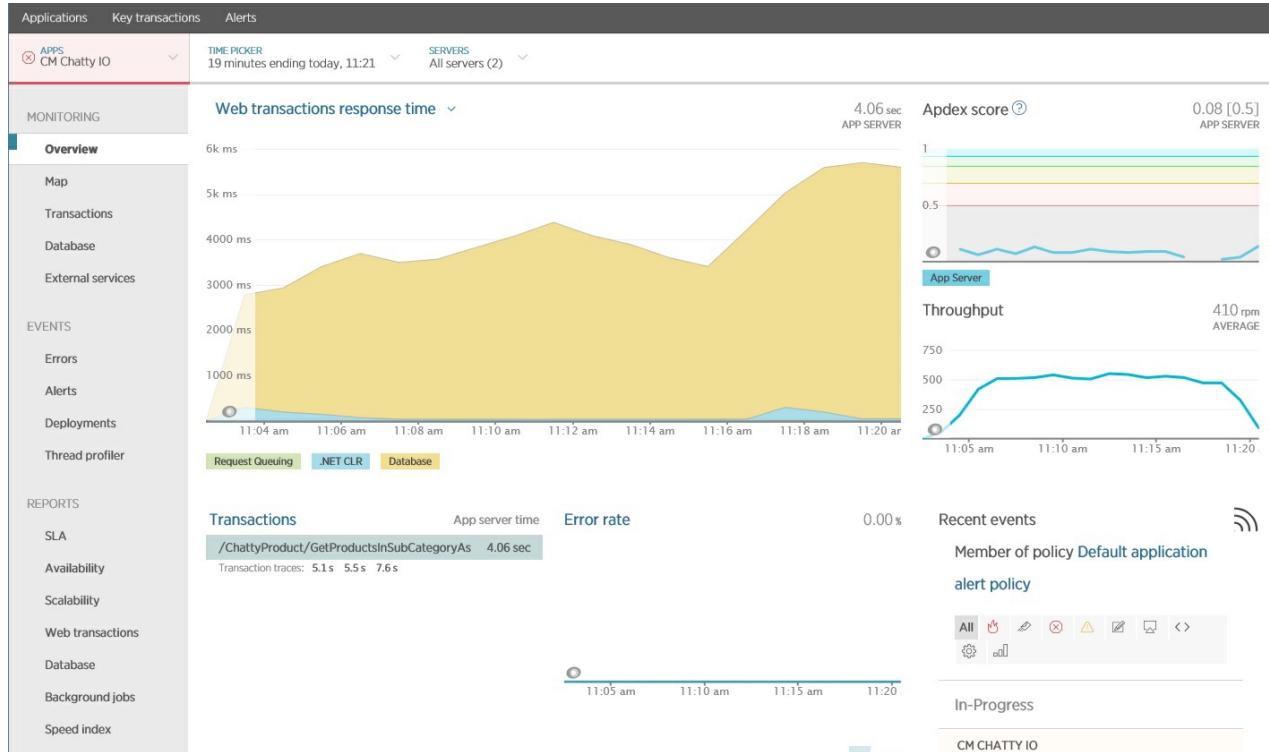
The application was deployed as an Azure App Service web app, using Azure SQL Database. The load test used a simulated step workload of up to 1000 concurrent users. The database was configured with a connection pool supporting up to 1000 concurrent connections, to reduce the chance that contention for connections would affect the results.

Monitor the application

You can use an application performance monitoring (APM) package to capture and analyze the key metrics that

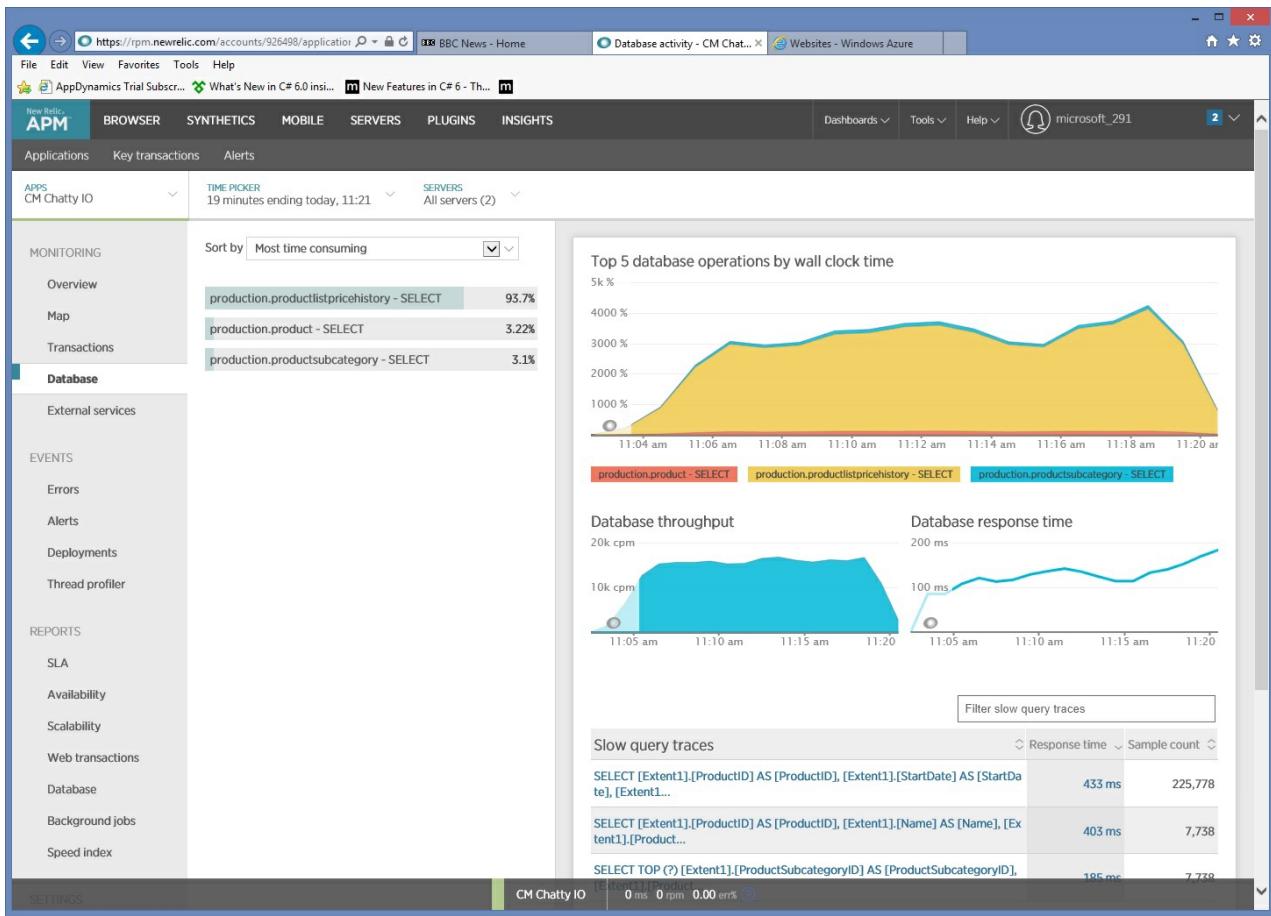
might identify chatty I/O. Which metrics are important will depend on the I/O workload. For this example, the interesting I/O requests were the database queries.

The following image shows results generated using [New Relic APM](#). The average database response time peaked at approximately 5.6 seconds per request during the maximum workload. The system was able to support an average of 410 requests per minute throughout the test.

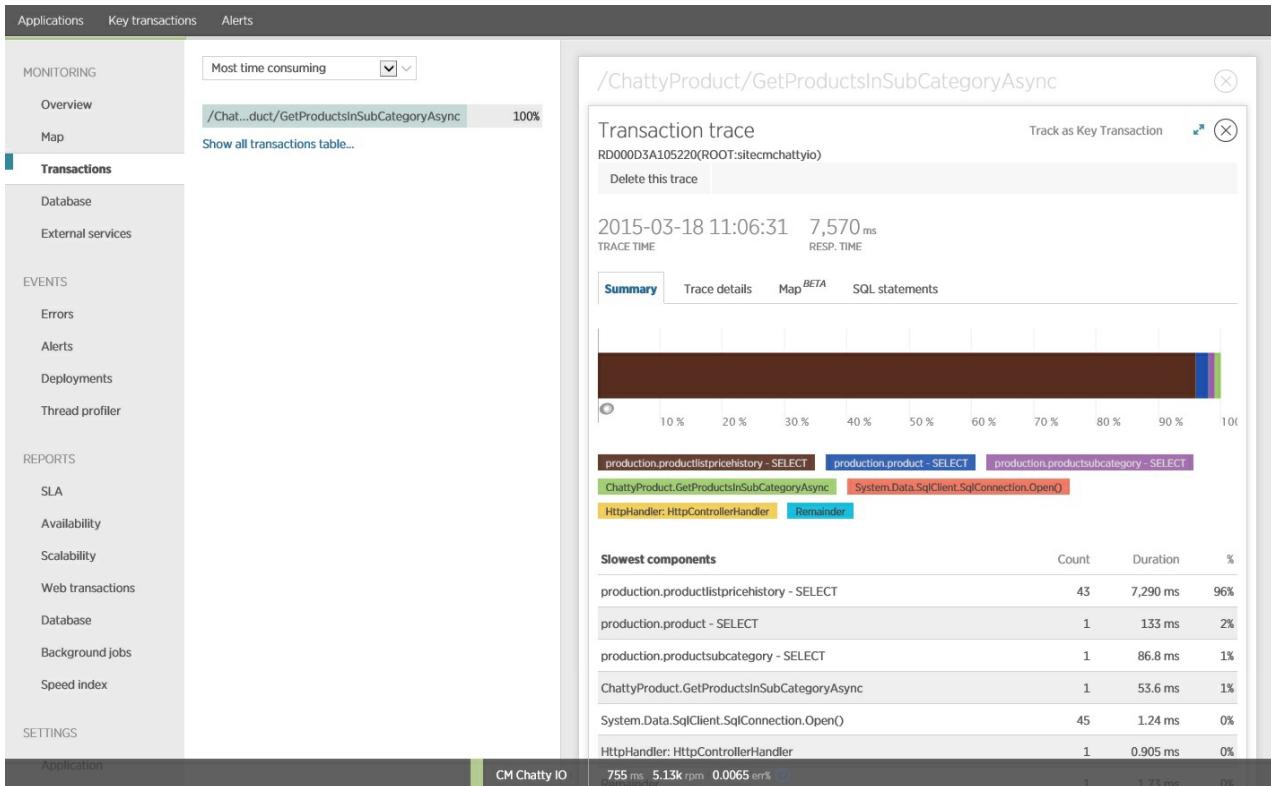


Gather detailed data access information

Digging deeper into the monitoring data shows the application executes three different SQL SELECT statements. These correspond to the requests generated by Entity Framework to fetch data from the `ProductListPriceHistory`, `Product`, and `ProductSubcategory` tables. Furthermore, the query that retrieves data from the `ProductListPriceHistory` table is by far the most frequently executed SELECT statement, by an order of magnitude.



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.

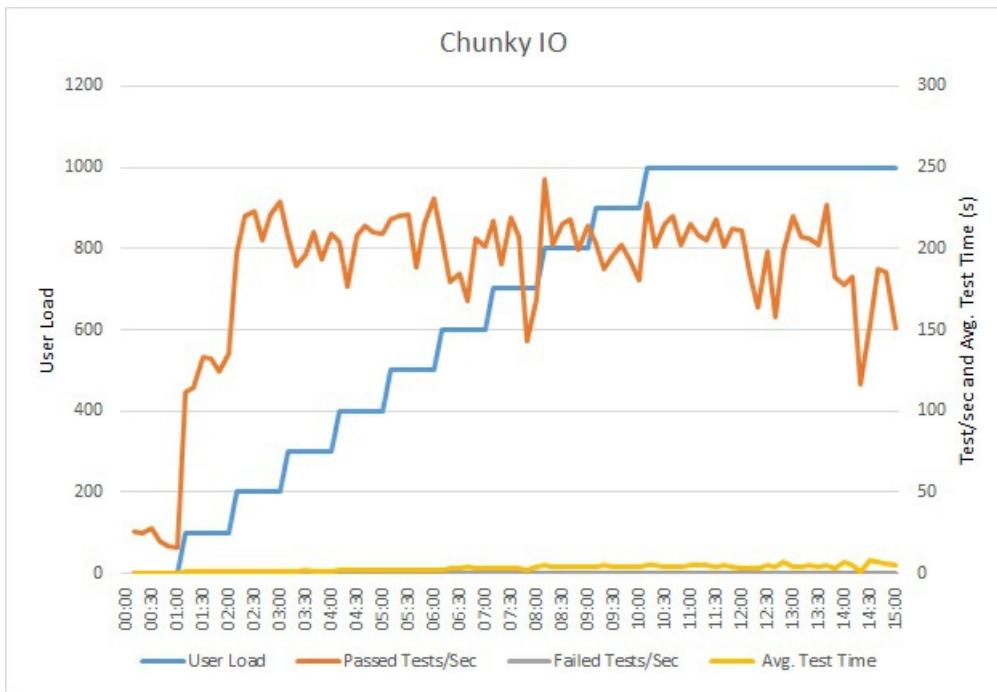
The screenshot shows the Application Insights interface. On the left, the navigation menu includes Applications, Key transactions, and Alerts. Under MONITORING, the 'Transactions' section is selected, showing a list of items like Database, External services, and a summary of most time consuming transactions. One transaction is highlighted: `/Chat...duct/GetProductsInSubCategoryAsync` with a duration of 100%. A link to 'Show all transactions table...' is also present. The main pane displays a 'Transaction trace' for the selected transaction. It shows the timestamp `2015-03-18 11:06:31`, duration `7,570 ms`, and trace ID `RD000D3A105220(ROOT:sitecmchattyio)`. There are tabs for Summary, Trace details, Map (Beta), and SQL statements, with 'SQL statements' currently selected. A checkbox 'Show fast queries (< 5ms)' is checked. Below is a table listing three SQL queries:

Total duration	Call count	SQL
7,290 ms	43	<code>SELECT [Extent1].[ProductID] AS [ProductID], [Extent1].[StartDate] AS [Start Date], [Extent1].[EndDate] AS [End Date], [Extent1].[ListPrice] AS [List Price] FROM [Production].[ProductListPriceHistory] AS [Extent1] WHERE [Extent1].[ProductID] = @p_linq_0</code>
86 ms	1	<code>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</code>
133 ms	1	<code>SELECT [Extent1].[ProductID] AS [ProductID], [Extent1].[Name] AS [Name], [Extent1].[ProductNumber] AS [ProductNumber], [Extent1].[ListPrice] AS [List Price], [Extent1].[ProductSubcategoryId] AS [ProductSubcategoryId] FROM [Production].[Product] AS [Extent1] WHERE @p_linq_0 = [Extent1].[ProductSubcategoryId]</code>

If you are using an O/RM, such as Entity Framework, tracing the SQL queries can provide insight into how the O/RM translates programmatic calls into SQL statements, and indicate areas where data access might be optimized.

Implement the solution and verify the result

Rewriting the call to Entity Framework produced the following results.



This load test was performed on the same deployment, using the same load profile. This time the graph shows much lower latency. The average request time at 1000 users is between 5 and 6 seconds, down from nearly a minute.

This time the system supported an average of 3,970 requests per minute, compared to 410 for the earlier test.

Tracing the SQL statement shows that all the data is fetched in a single SELECT statement. Although this query is considerably more complex, it is performed only once per operation. And while complex joins can become expensive, relational database systems are optimized for this type of query.

Applications Key transactions Alerts

External services

EVENTS

- Errors
- Alerts
- Deployments
- Thread profiler

REPORTS

- SLA
- Availability
- Scalability
- Web transactions
- Database
- Background jobs
- Speed index

SETTINGS

- Application
- Availability monitoring
- Environment

ALERTS >

2015-03-18 11:40:24 11,700 ms
TRACE TIME RESP. TIME

Summary Trace details Map BETA SQL statements

Show fast queries (< 5ms)

Total duration	Call count	SQL
11,700 ms	1	<pre> SELECT [Project1].[ProductSubcategoryId] AS [ProductSubcategoryId], [Project1].[ProductCategoryId] AS [ProductCategoryId], [Project1].[Name] AS [Name], [Project1].[C2] AS [C1], [Project1].[ProductId] AS [ProductId], [Project1].[Name1] AS [Name1], [Project1].[ProductNumber] AS [ProductNumber], [Project1].[ListPrice] AS [ListPrice], [Project1].[ProductSubcategoryID1] AS [ProductSubcategoryID1], [Project1].[C1] AS [C2], [Project1].[ProductId1] AS [ProductId1], [Project1].[StartDate] AS [StartDate], [Project1].[EndDate] AS [EndDate], [Project1].[ListPrice1] AS [ListPrice1] FROM (SELECT [Limit1].[ProductSubcategoryId] AS [ProductSubcategoryId], [Limit1].[Name] AS [Name], [Join1].[ProductId1] AS [ProductId1], [Join1].[Name] AS [Name1], [Join1].[ProductNumber] AS [ProductNumber], [Join1].[ListPrice1] AS [ListPrice1], [Join1].[ProductID2] AS [ProductID1], [Join1].[StartDate] AS [StartDate], [Join1].[EndDate] AS [EndDate], [Join1].[ListPrice2] AS [ListPrice1], CASE WHEN ([Join1].[ProductId1] IS NULL) THEN CAST(NULL AS int) WHEN ([Join1].[ProductId2] IS NULL) THEN CAST(NULL AS int) ELSE ? END AS [C1], CASE WHEN ([Join1].[ProductId1] IS NULL) THEN CAST(NULL AS int) ELSE ? END AS [C2] FROM (SELECT TOP (?) [Extent1].[ProductSubcategoryId] AS [ProductSubcategoryId], [Extent1].[Name] AS [Name], [Extent1].[ProductId] AS [ProductId], [Extent1].[Extent1] AS [Name] FROM [Production].[ProductSubcategory] AS [Extent1] WHERE [Extent1].[ProductSubcategoryId] = @p__l1nq_0) AS [Limit1] LEFT OUTER JOIN (SELECT [Extent2].[ProductId] AS [ProductId1], [Extent2].[Name] AS [Name], [Extent2].[ProductNumber] AS [ProductNumber], [Extent2].[ListPrice] AS [ListPrice1], [Extent2].[ProductSubcategoryId] AS [ProductSubcategoryId1], [Extent2].[ProductId2], [Extent3].[StartDate] AS [StartDate], [Extent3].[EndDate]... (more)))</pre>

Related resources

- [API Design best practices](#)
- [Caching best practices](#)
- [Data Consistency Primer](#)
- [Extraneous Fetching antipattern](#)
- [No Caching antipattern](#)

Extraneous Fetching antipattern

4/11/2018 • 10 minutes to read • [Edit Online](#)

Retrieving more data than needed for a business operation can result in unnecessary I/O overhead and reduce responsiveness.

Problem description

This antipattern can occur if the application tries to minimize I/O requests by retrieving all of the data that it *might* need. This is often a result of overcompensating for the [Chatty I/O](#) antipattern. For example, an application might fetch the details for every product in a database. But the user may need just a subset of the details (some may not be relevant to customers), and probably doesn't need to see *all* of the products at once. Even if the user is browsing the entire catalog, it would make sense to paginate the results — showing 20 at a time, for example.

Another source of this problem is following poor programming or design practices. For example, the following code uses Entity Framework to fetch the complete details for every product. Then it filters the results to return only a subset of the fields, discarding the rest. You can find the complete sample [here](#).

```
public async Task<IHttpActionResult> GetAllFieldsAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Execute the query. This happens at the database.
        var products = await context.Products.ToListAsync();

        // Project fields from the query results. This happens in application memory.
        var result = products.Select(p => new ProductInfo { Id = p.ProductId, Name = p.Name });
        return Ok(result);
    }
}
```

In the next example, the application retrieves data to perform an aggregation that could be done by the database instead. The application calculates total sales by getting every record for all orders sold, and then computing the sum over those records. You can find the complete sample [here](#).

```
public async Task<IHttpActionResult> AggregateOnClientAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Fetch all order totals from the database.
        var orderAmounts = await context.SalesOrderHeaders.Select(soh => soh.TotalDue).ToListAsync();

        // Sum the order totals in memory.
        var total = orderAmounts.Sum();
        return Ok(total);
    }
}
```

The next example shows a subtle problem caused by the way Entity Framework uses LINQ to Entities.

```

var query = from p in context.Products.AsEnumerable()
            where p.SellStartDate < DateTime.Now.AddDays(-7) // AddDays cannot be mapped by LINQ to Entities
            select ...;

List<Product> products = query.ToList();

```

The application is trying to find products with a `SellStartDate` more than a week old. In most cases, LINQ to Entities would translate a `where` clause to a SQL statement that is executed by the database. In this case, however, LINQ to Entities cannot map the `AddDays` method to SQL. Instead, every row from the `Product` table is returned, and the results are filtered in memory.

The call to `AsEnumerable` is a hint that there is a problem. This method converts the results to an `IEnumerable` interface. Although `IEnumerable` supports filtering, the filtering is done on the *client* side, not the database. By default, LINQ to Entities uses `IQueryable`, which passes the responsibility for filtering to the data source.

How to fix the problem

Avoid fetching large volumes of data that may quickly become outdated or might be discarded, and only fetch the data needed for the operation being performed.

Instead of getting every column from a table and then filtering them, select the columns that you need from the database.

```

public async Task<IHttpActionResult> GetRequiredFieldsAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Project fields as part of the query itself
        var result = await context.Products
            .Select(p => new ProductInfo {Id = p.ProductId, Name = p.Name})
            .ToListAsync();
        return Ok(result);
    }
}

```

Similarly, perform aggregation in the database and not in application memory.

```

public async Task<IHttpActionResult> AggregateOnDatabaseAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Sum the order totals as part of the database query.
        var total = await context.SalesOrderHeaders.SumAsync(soh => soh.TotalDue);
        return Ok(total);
    }
}

```

When using Entity Framework, ensure that LINQ queries are resolved using the `IQueryable` interface and not `IEnumerable`. You may need to adjust the query to use only functions that can be mapped to the data source. The earlier example can be refactored to remove the `AddDays` method from the query, allowing filtering to be done by the database.

```

DateTime dateSince = DateTime.Now.AddDays(-7); // AddDays has been factored out.
var query = from p in context.Products
            where p.SellStartDate < dateSince // This criterion can be passed to the database by LINQ to
Entities
            select ...;

List<Product> products = query.ToList();

```

Considerations

- In some cases, you can improve performance by partitioning data horizontally. If different operations access different attributes of the data, horizontal partitioning may reduce contention. Often, most operations are run against a small subset of the data, so spreading this load may improve performance. See [Data partitioning](#).
- For operations that have to support unbounded queries, implement pagination and only fetch a limited number of entities at a time. For example, if a customer is browsing a product catalog, you can show one page of results at a time.
- When possible, take advantage of features built into the data store. For example, SQL databases typically provide aggregate functions.
- If you're using a data store that doesn't support a particular function, such as aggregation, you could store the calculated result elsewhere, updating the value as records are added or updated, so the application doesn't have to recalculate the value each time it's needed.
- If you see that requests are retrieving a large number of fields, examine the source code to determine whether all of these fields are actually necessary. Sometimes these requests are the result of poorly designed `SELECT *` query.
- Similarly, requests that retrieve a large number of entities may be sign that the application is not filtering data correctly. Verify that all of these entities are actually needed. Use database-side filtering if possible, for example, by using `WHERE` clauses in SQL.
- Offloading processing to the database is not always the best option. Only use this strategy when the database is designed or optimized to do so. Most database systems are highly optimized for certain functions, but are not designed to act as general-purpose application engines. For more information, see the [Busy Database antipattern](#).

How to detect the problem

Symptoms of extraneous fetching include high latency and low throughput. If the data is retrieved from a data store, increased contention is also probable. End users are likely to report extended response times or failures caused by services timing out. These failures could return HTTP 500 (Internal Server) errors or HTTP 503 (Service Unavailable) errors. Examine the event logs for the web server, which are likely to contain more detailed information about the causes and circumstances of the errors.

The symptoms of this antipattern and some of the telemetry obtained might be very similar to those of the [Monolithic Persistence antipattern](#).

You can perform the following steps to help identify the cause:

1. Identify slow workloads or transactions by performing load-testing, process monitoring, or other methods of capturing instrumentation data.
2. Observe any behavioral patterns exhibited by the system. Are there particular limits in terms of transactions per second or volume of users?

3. Correlate the instances of slow workloads with behavioral patterns.
4. Identify the data stores being used. For each data source, run lower level telemetry to observe the behavior of operations.
5. Identify any slow-running queries that reference these data sources.
6. Perform a resource-specific analysis of the slow-running queries and ascertain how the data is used and consumed.

Look for any of these symptoms:

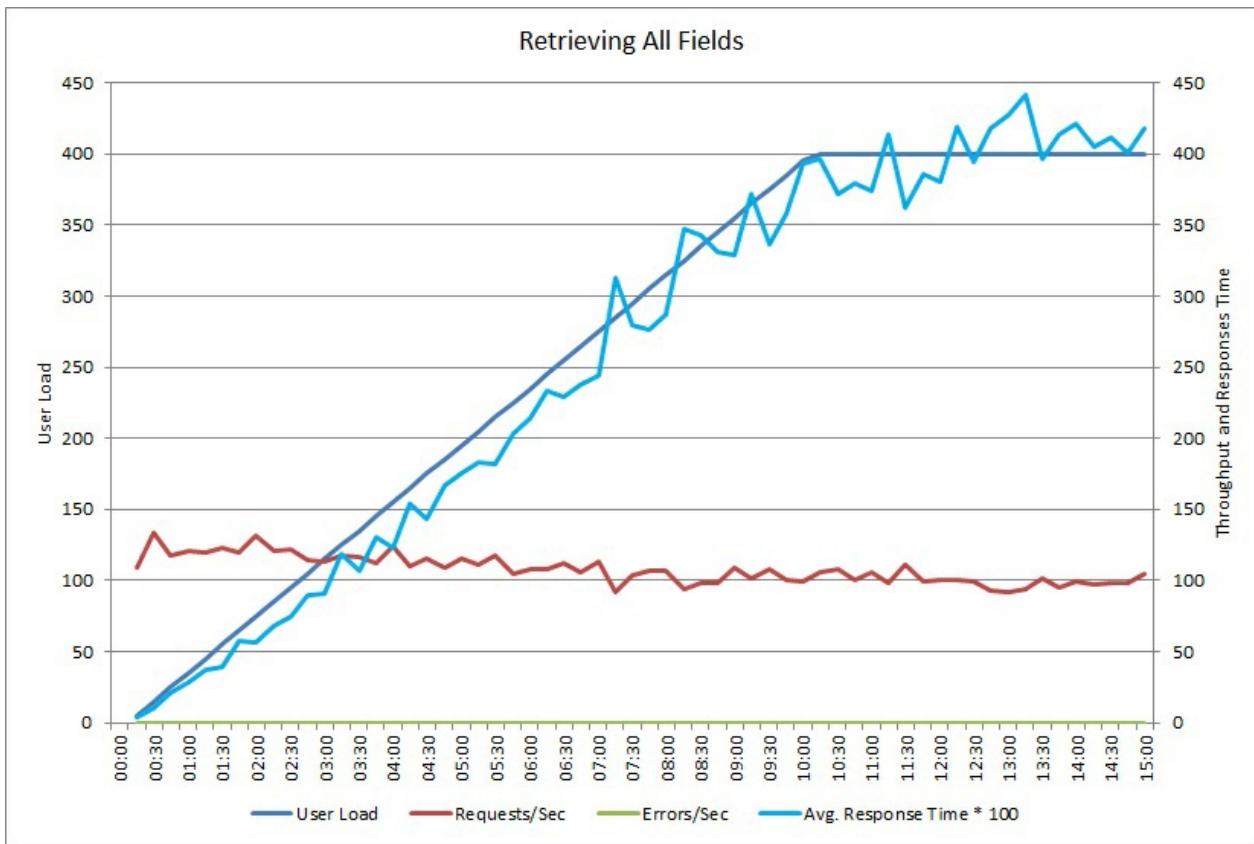
- Frequent, large I/O requests made to the same resource or data store.
- Contention in a shared resource or data store.
- An operation that frequently receives large volumes of data over the network.
- Applications and services spending significant time waiting for I/O to complete.

Example diagnosis

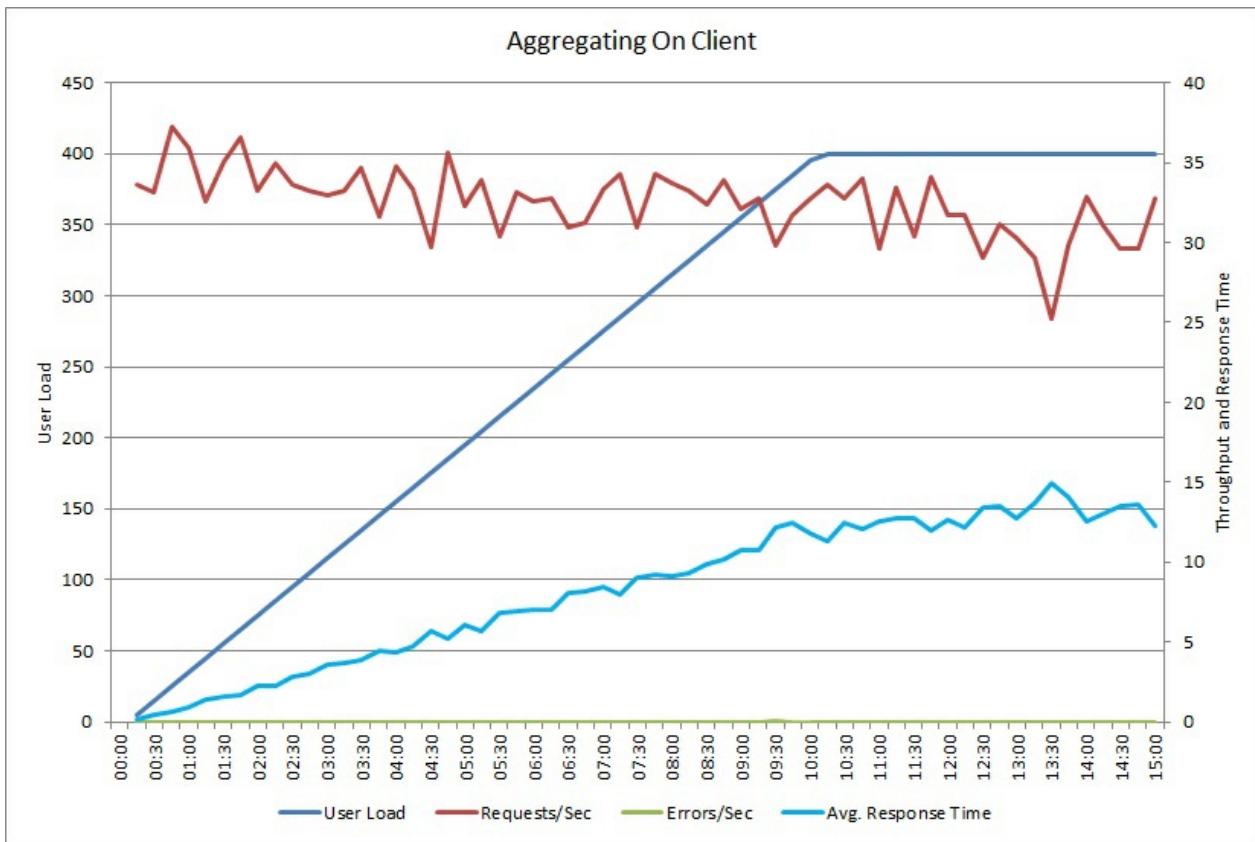
The following sections apply these steps to the previous examples.

Identify slow workloads

This graph shows performance results from a load test that simulated up to 400 concurrent users running the `GetAllFieldsAsync` method shown earlier. Throughput diminishes slowly as the load increases. Average response time goes up as the workload increases.



A load test for the `AggregateOnClientAsync` operation shows a similar pattern. The volume of requests is reasonably stable. The average response time increases with the workload, although more slowly than the previous graph.



Correlate slow workloads with behavioral patterns

Any correlation between regular periods of high usage and slowing performance can indicate areas of concern. Closely examine the performance profile of functionality that is suspected to be slow running, to determine whether it matches the load testing performed earlier.

Load test the same functionality using step-based user loads, to find the point where performance drops significantly or fails completely. If that point falls within the bounds of your expected real-world usage, examine how the functionality is implemented.

A slow operation is not necessarily a problem, if it is not being performed when the system is under stress, is not time critical, and does not negatively affect the performance of other important operations. For example, generating monthly operational statistics might be a long-running operation, but it can probably be performed as a batch process and run as a low priority job. On the other hand, customers querying the product catalog is a critical business operation. Focus on the telemetry generated by these critical operations to see how the performance varies during periods of high usage.

Identify data sources in slow workloads

If you suspect that a service is performing poorly because of the way it retrieves data, investigate how the application interacts with the repositories it uses. Monitor the live system to see which sources are accessed during periods of poor performance.

For each data source, instrument the system to capture the following:

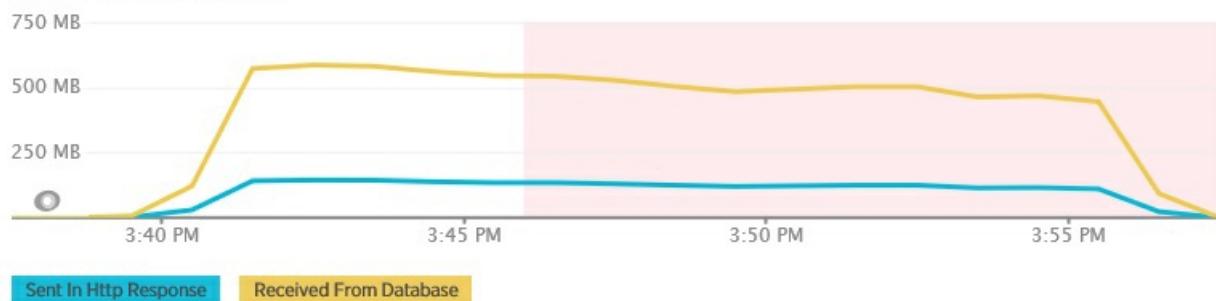
- The frequency that each data store is accessed.
- The volume of data entering and exiting the data store.
- The timing of these operations, especially the latency of requests.
- The nature and rate of any errors that occur while accessing each data store under typical load.

Compare this information against the volume of data being returned by the application to the client. Track the ratio of the volume of data returned by the data store against the volume of data returned to the client. If there is any large disparity, investigate to determine whether the application is fetching data that it doesn't need.

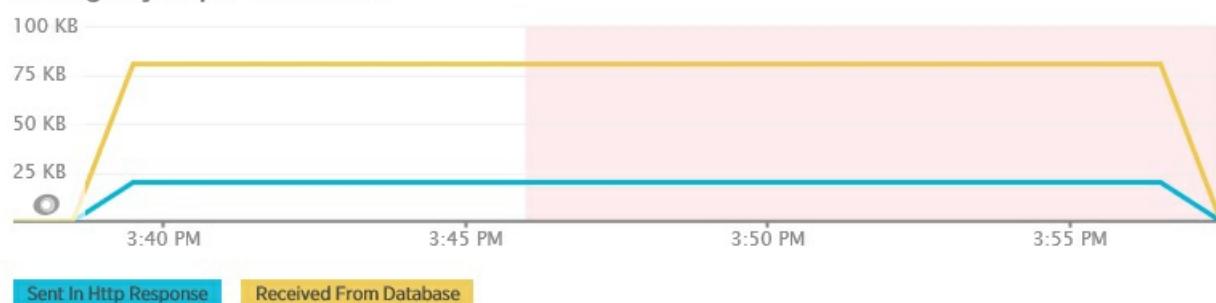
You may be able to capture this data by observing the live system and tracing the lifecycle of each user request, or you can model a series of synthetic workloads and run them against a test system.

The following graphs show telemetry captured using [New Relic APM](#) during a load test of the `GetAllFieldsAsync` method. Note the difference between the volumes of data received from the database and the corresponding HTTP responses.

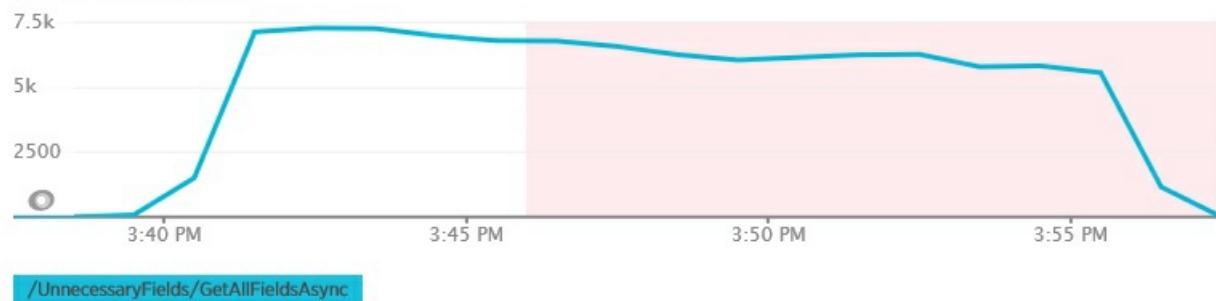
Total Bytes per Minute



Average Bytes per Transaction

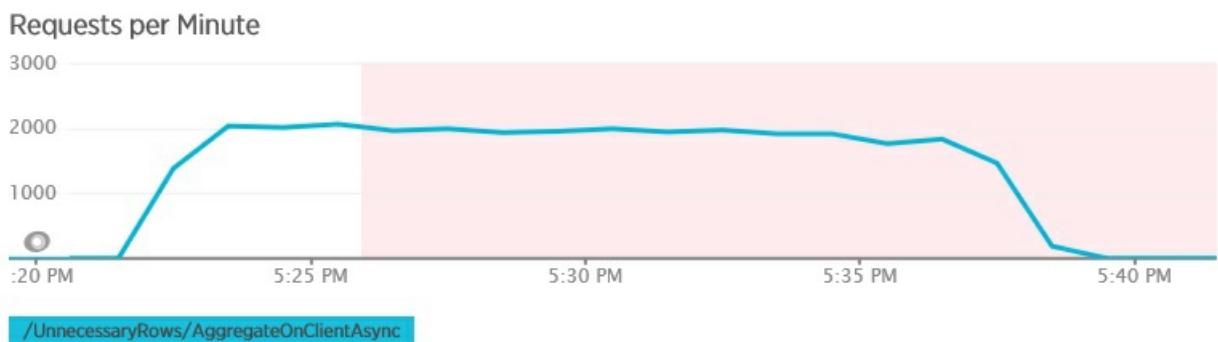
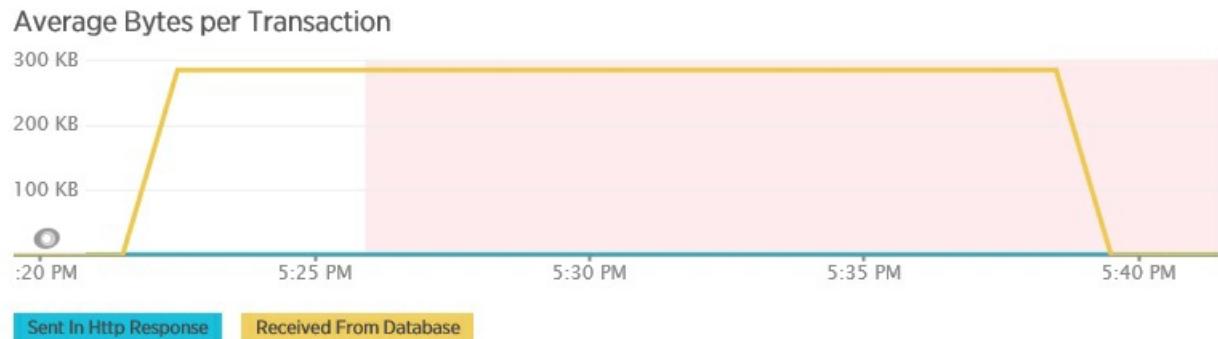
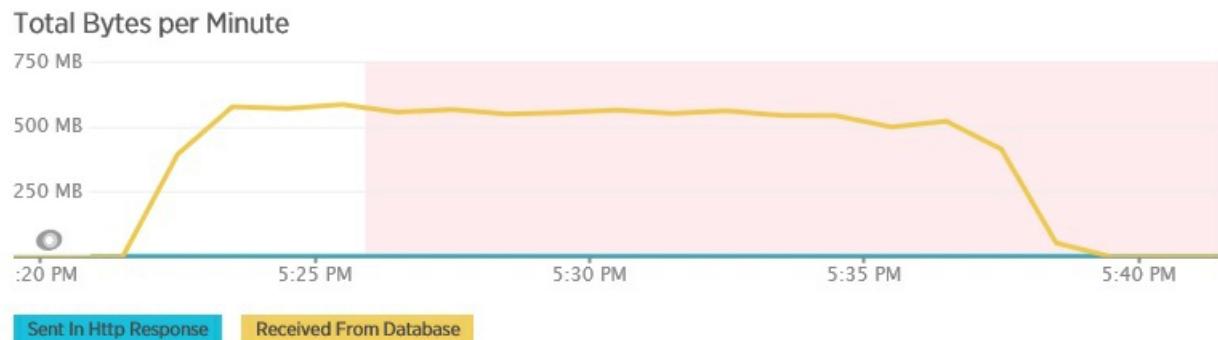


Requests per Minute



For each request, the database returned 80,503 bytes, but the response to the client only contained 19,855 bytes, about 25% of the size of the database response. The size of the data returned to the client can vary depending on the format. For this load test, the client requested JSON data. Separate testing using XML (not shown) had a response size of 35,655 bytes, or 44% of the size of the database response.

The load test for the `AggregateOnClientAsync` method shows more extreme results. In this case, each test performed a query that retrieved over 280Kb of data from the database, but the JSON response was a mere 14 bytes. The wide disparity is because the method calculates an aggregated result from a large volume of data.



Identify and analyze slow queries

Look for database queries that consume the most resources and take the most time to execute. You can add instrumentation to find the start and completion times for many database operations. Many data stores also provide in-depth information on how queries are performed and optimized. For example, the Query Performance pane in the Azure SQL Database management portal lets you select a query and view detailed runtime performance information. Here is the query generated by the `GetAllFieldsAsync` operation:

```

SELECT
    [Extent1].[ProductID] AS [ProductID],
    [Extent1].[Name] AS [Name],
    [Extent1].[ProductNumber] AS [ProductNumber],
    [Extent1].[MakeFlag] AS [MakeFlag],
    [Extent1].[FinishedGoodsFlag] AS [FinishedGoodsFlag],
    [Extent1].[Color] AS [Color],
    [Extent1].[SafetyStockLevel] AS [SafetyStockLevel],
    [Extent1].[ReorderPoint] AS [ReorderPoint],
    [Extent1].[StandardCost] AS [StandardCost],
    [Extent1].[ListPrice] AS [ListPrice],
    [Extent1].[Size] AS [Size],
    [Extent1].[SizeUnitMeasureCode] AS [SizeUnitMeasureCode],
    [Extent1].[WeightUnitMeasureCode] AS [WeightUnitMeasureCode],
    [Extent1].[Weight] AS [Weight],

```

Query Plan Details [Query Plan](#)

Resource Use

Resource	Total / sec	Total	Last Run	Minimum	Maximum
Duration (ms)	117	63450	40	0	3867
CPU (ms)	1	752	0	0	3
Logical Reads	25	13872	16	16	16
Physical Reads	0	0	0	0	0
Logical Writes	0	0	0	0	0

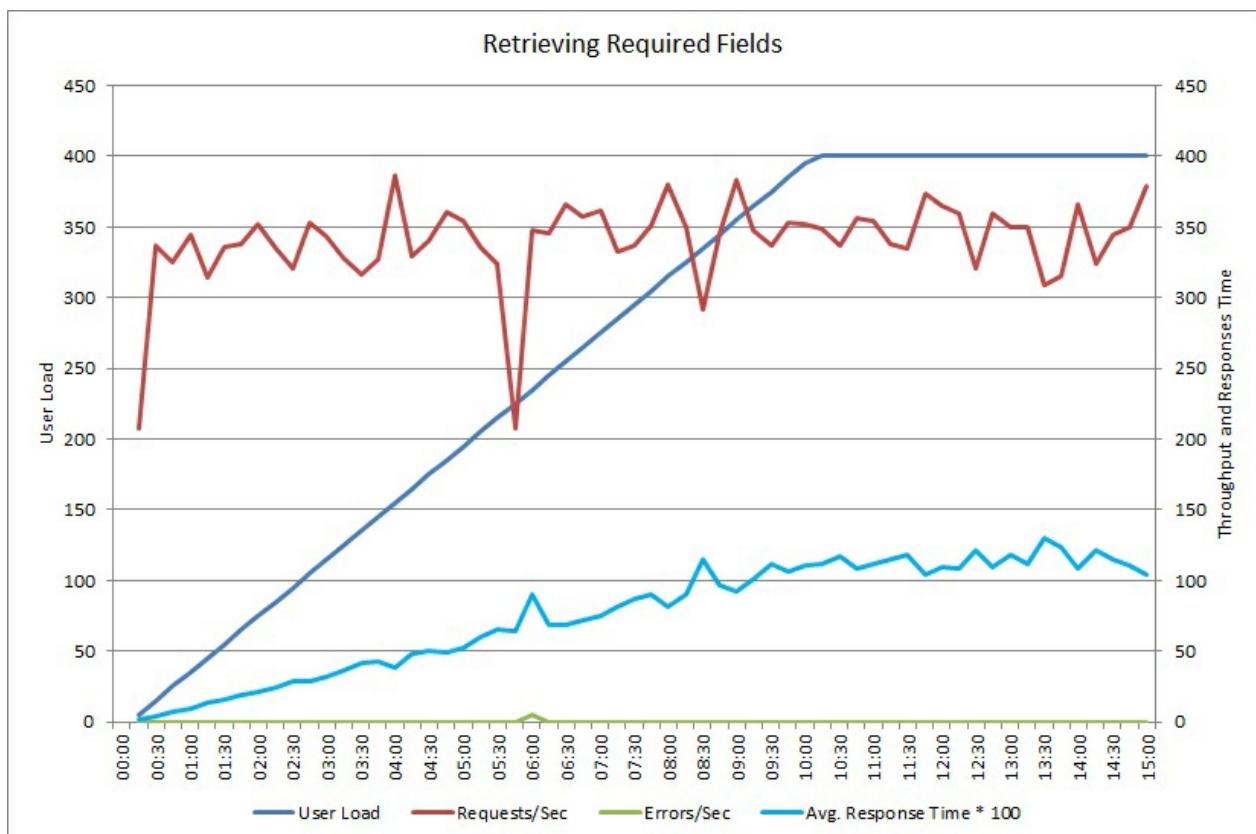
Plan Information

Advanced Information

Run Count	867	Plan Handle	0x0600E0004D00CD02A0EDD87D550000000100000000000000000000000000000000
Last Run Time	03/03/2015 15:32:25	SQL Handle	0x020000004D00CD02B85696A75DA7BEBF79E79259988F30C300000000
Plan Generation Count	1	Query Hash	0x05ACF4F9056ACADC
Time Plan Cached	03/03/2015 15:26:32	Query Plan Hash	0x0DA11AA10A268A7B

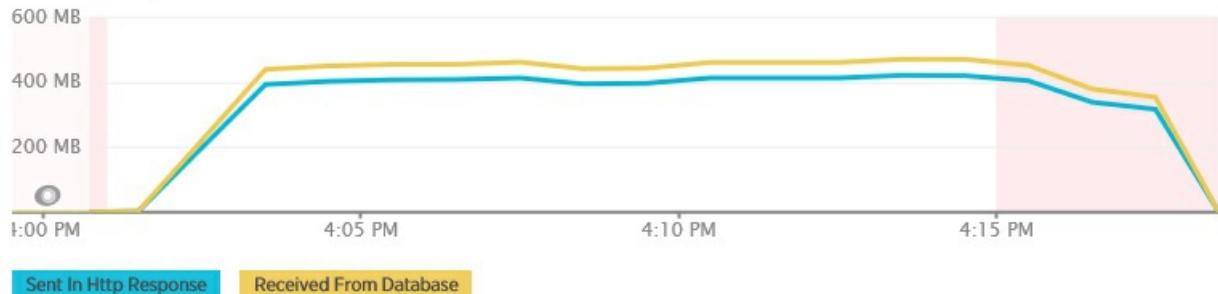
Implement the solution and verify the result

After changing the `GetRequiredFieldsAsync` method to use a SELECT statement on the database side, load testing showed the following results.

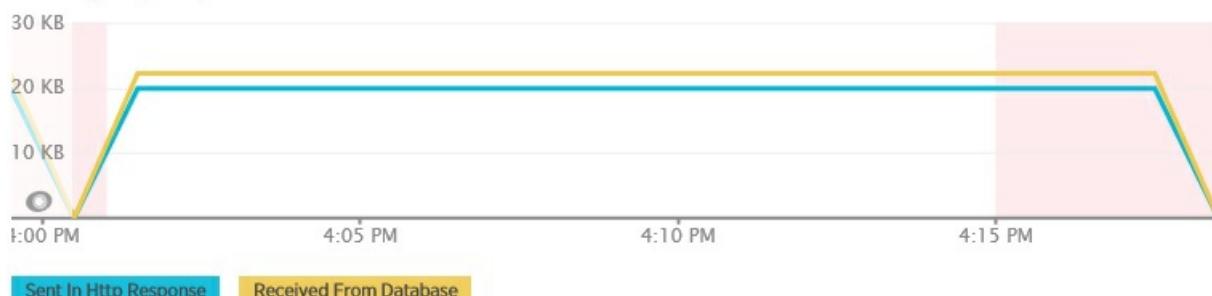


This load test used the same deployment and the same simulated workload of 400 concurrent users as before. The graph shows much lower latency. Response time rises with load to approximately 1.3 seconds, compared to 4 seconds in the previous case. The throughput is also higher at 350 requests per second compared to 100 earlier. The volume of data retrieved from the database now closely matches the size of the HTTP response messages.

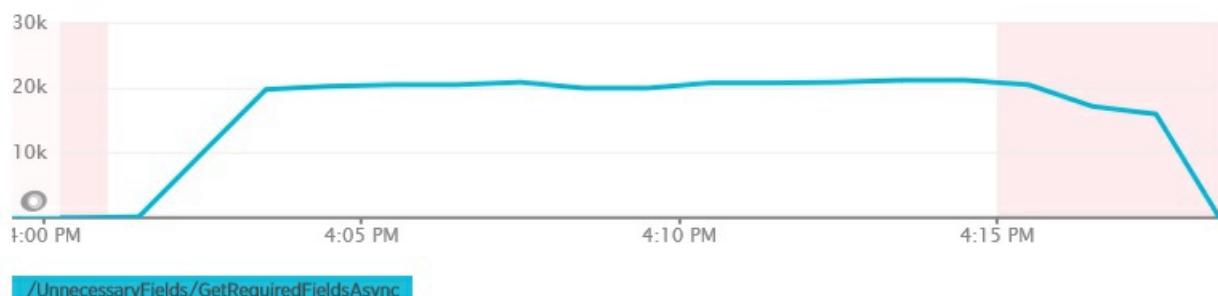
Total Bytes per Minute



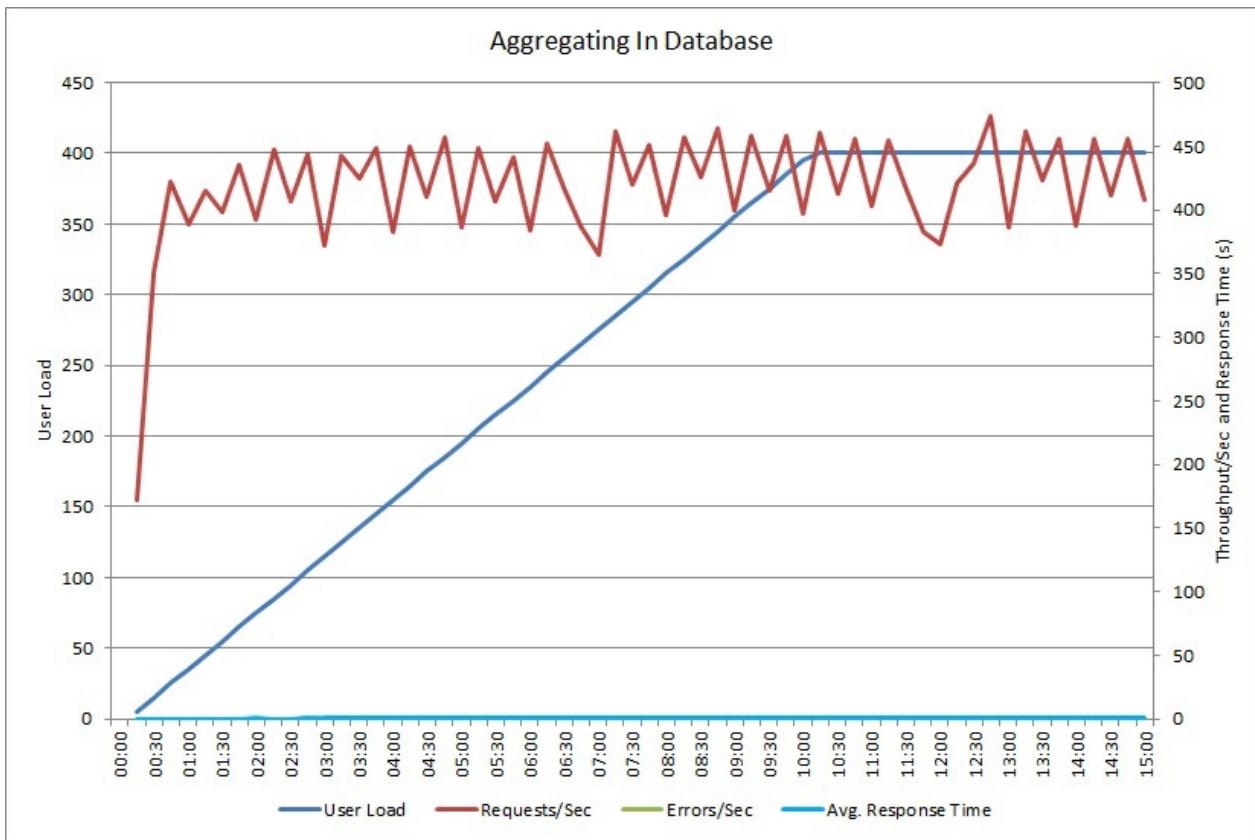
Average Bytes per Transaction



Requests per Minute



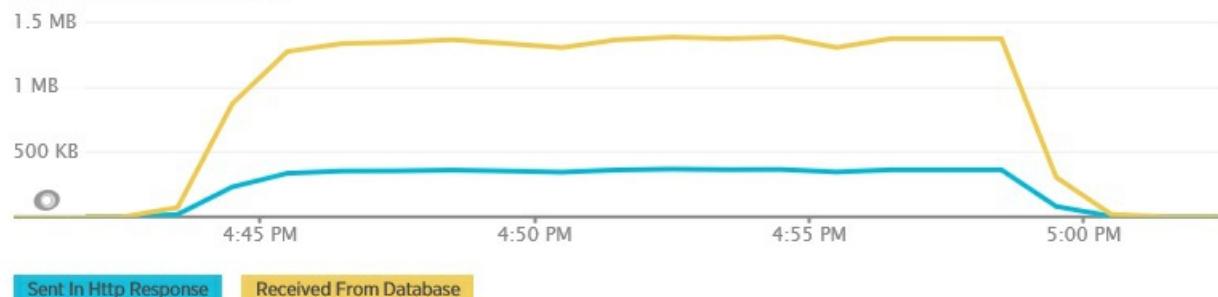
Load testing using the `AggregateOnDatabaseAsync` method generates the following results:



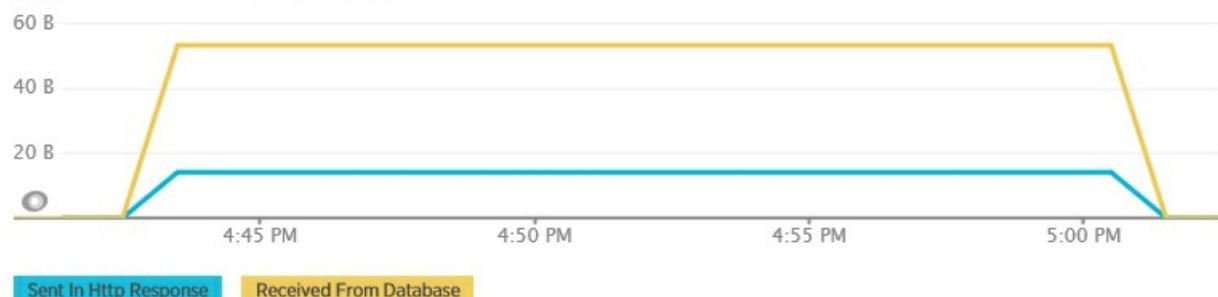
The average response time is now minimal. This is an order of magnitude improvement in performance, caused primarily by the large reduction in I/O from the database.

Here is the corresponding telemetry for the `AggregateOnDatabaseAsync` method. The amount of data retrieved from the database was vastly reduced, from over 280Kb per transaction to 53 bytes. As a result, the maximum sustained number of requests per minute was raised from around 2,000 to over 25,000.

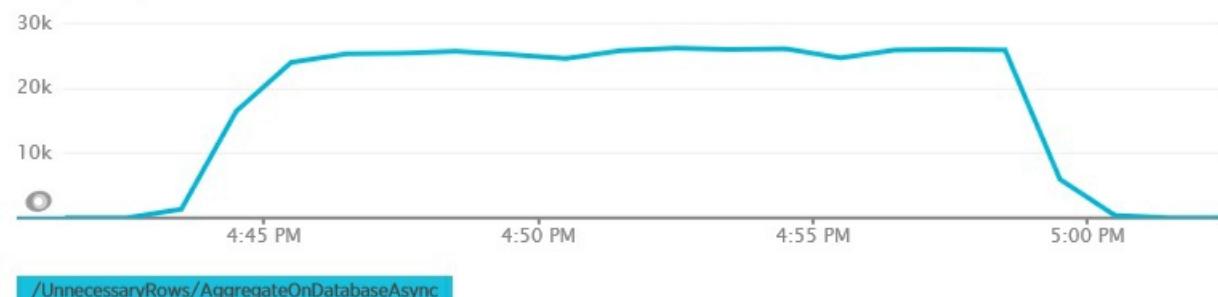
Total Bytes per Minute



Average Bytes per Transaction



Requests per Minute



Related resources

- [Busy Database antipattern](#)
- [Chatty I/O antipattern](#)
- [Data partitioning best practices](#)

Improper Instantiation antipattern

2/22/2018 • 6 minutes to read • [Edit Online](#)

It can hurt performance to continually create new instances of an object that is meant to be created once and then shared.

Problem description

Many libraries provide abstractions of external resources. Internally, these classes typically manage their own connections to the resource, acting as brokers that clients can use to access the resource. Here are some examples of broker classes that are relevant to Azure applications:

- `System.Net.Http.HttpClient`. Communicates with a web service using HTTP.
- `Microsoft.ServiceBus.Messaging.QueueClient`. Posts and receives messages to a Service Bus queue.
- `Microsoft.Azure.Documents.Client.DocumentClient`. Connects to a Cosmos DB instance
- `StackExchange.Redis.ConnectionMultiplexer`. Connects to Redis, including Azure Redis Cache.

These classes are intended to be instantiated once and reused throughout the lifetime of an application. However, it's a common misunderstanding that these classes should be acquired only as necessary and released quickly. (The ones listed here happen to be .NET libraries, but the pattern is not unique to .NET.) The following ASP.NET example creates an instance of `HttpClient` to communicate with a remote service. You can find the complete sample [here](#).

```
public class NewHttpClientInstancePerRequestController : ApiController
{
    // This method creates a new instance of HttpClient and disposes it for every call to GetProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        using (var httpClient = new HttpClient())
        {
            var hostName = HttpContext.Current.Request.Url.Host;
            var result = await httpClient.GetStringAsync(string.Format("http://{0}:8080/api...", hostName));
            return new Product { Name = result };
        }
    }
}
```

In a web application, this technique is not scalable. A new `HttpClient` object is created for each user request.

Under heavy load, the web server may exhaust the number of available sockets, resulting in `SocketException` errors.

This problem is not restricted to the `HttpClient` class. Other classes that wrap resources or are expensive to create might cause similar issues. The following example creates an instances of the `ExpensiveToCreateService` class. Here the issue is not necessarily socket exhaustion, but simply how long it takes to create each instance. Continually creating and destroying instances of this class might adversely affect the scalability of the system.

```

public class NewServiceInstancePerRequestController : ApiController
{
    public async Task<Product> GetProductAsync(string id)
    {
        var expensiveToCreateService = new ExpensiveToCreateService();
        return await expensiveToCreateService.GetProductByIdAsync(id);
    }
}

public class ExpensiveToCreateService
{
    public ExpensiveToCreateService()
    {
        // Simulate delay due to setup and configuration of ExpensiveToCreateService
        Thread.Sleep(Int32.MaxValue / 100);
    }
    ...
}

```

How to fix the problem

If the class that wraps the external resource is shareable and thread-safe, create a shared singleton instance or a pool of reusable instances of the class.

The following example uses a static `HttpClient` instance, thus sharing the connection across all requests.

```

public class SingleHttpClientInstanceController : ApiController
{
    private static readonly HttpClient httpClient;

    static SingleHttpClientInstanceController()
    {
        httpClient = new HttpClient();
    }

    // This method uses the shared instance of HttpClient for every call to GetProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        var hostName = HttpContext.Current.Request.Url.Host;
        var result = await httpClient.GetStringAsync(string.Format("http://{0}:8080/api/...", hostName));
        return new Product { Name = result };
    }
}

```

Considerations

- The key element of this antipattern is repeatedly creating and destroying instances of a *shareable* object. If a class is not shareable (not thread-safe), then this antipattern does not apply.
- The type of shared resource might dictate whether you should use a singleton or create a pool. The `HttpClient` class is designed to be shared rather than pooled. Other objects might support pooling, enabling the system to spread the workload across multiple instances.
- Objects that you share across multiple requests *must* be thread-safe. The `HttpClient` class is designed to be used in this manner, but other classes might not support concurrent requests, so check the available documentation.
- Some resource types are scarce and should not be held onto. Database connections are an example. Holding an open database connection that is not required may prevent other concurrent users from gaining access to the database.

- In the .NET Framework, many objects that establish connections to external resources are created by using static factory methods of other classes that manage these connections. These factories objects are intended to be saved and reused, rather than disposed and recreated. For example, in Azure Service Bus, the `QueueClient` object is created through a `MessagingFactory` object. Internally, the `MessagingFactory` manages connections. For more information, see [Best Practices for performance improvements using Service Bus Messaging](#).

How to detect the problem

Symptoms of this problem include a drop in throughput or an increased error rate, along with one or more of the following:

- An increase in exceptions that indicate exhaustion of resources such as sockets, database connections, file handles, and so on.
- Increased memory use and garbage collection.
- An increase in network, disk, or database activity.

You can perform the following steps to help identify this problem:

1. Performing process monitoring of the production system, to identify points when response times slow down or the system fails due to lack of resources.
2. Examine the telemetry data captured at these points to determine which operations might be creating and destroying resource-consuming objects.
3. Load test each suspected operation, in a controlled test environment rather than the production system.
4. Review the source code and examine the how broker objects are managed.

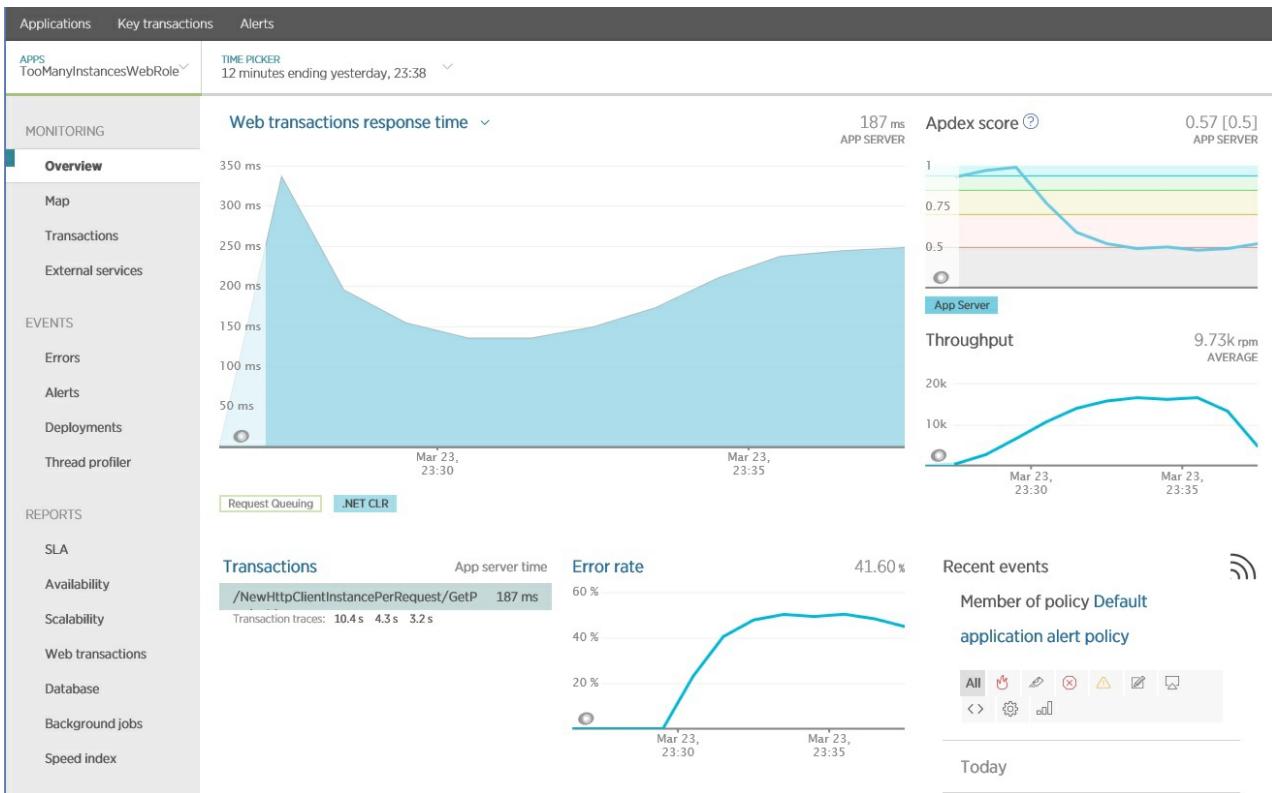
Look at stack traces for operations that are slow-running or that generate exceptions when the system is under load. This information can help to identify how these operations are utilizing resources. Exceptions can help to determine whether errors are caused by shared resources being exhausted.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

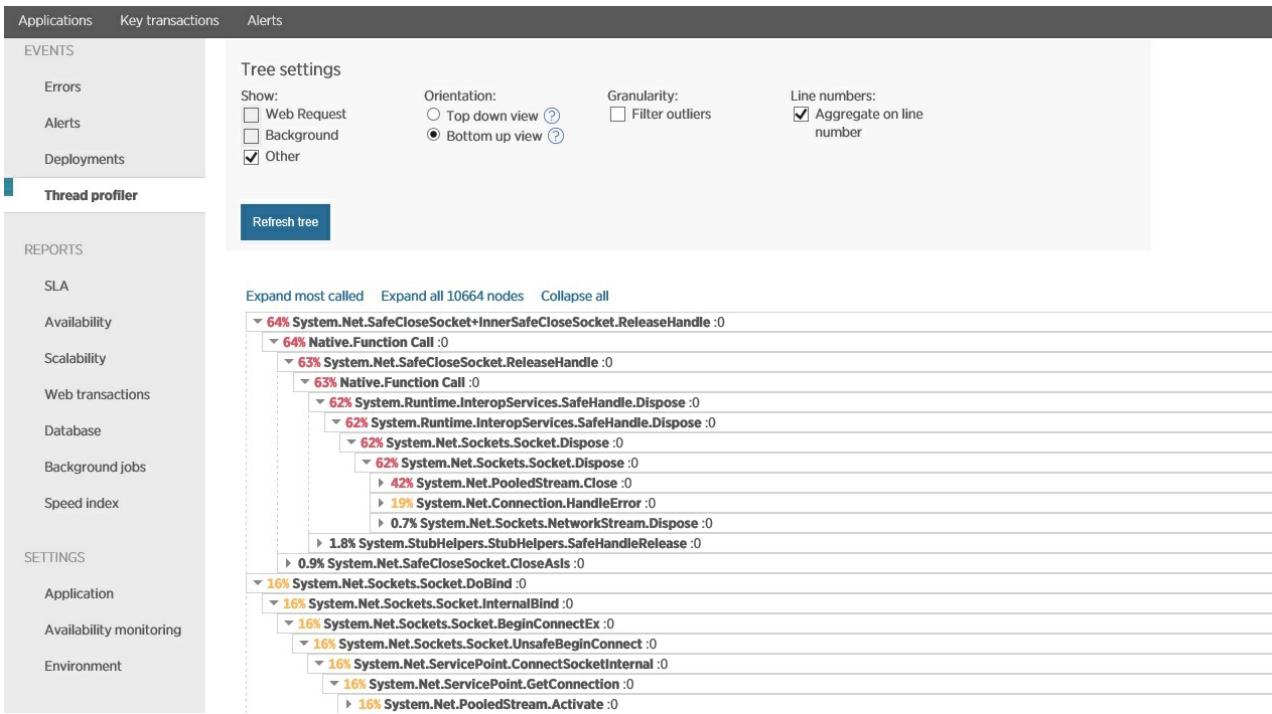
Identify points of slow down or failure

The following image shows results generated using [New Relic APM](#), showing operations that have a poor response time. In this case, the `GetProductAsync` method in the `NewHttpClientInstancePerRequest` controller is worth investigating further. Notice that the error rate also increases when these operations are running.



Examine telemetry data and find correlations

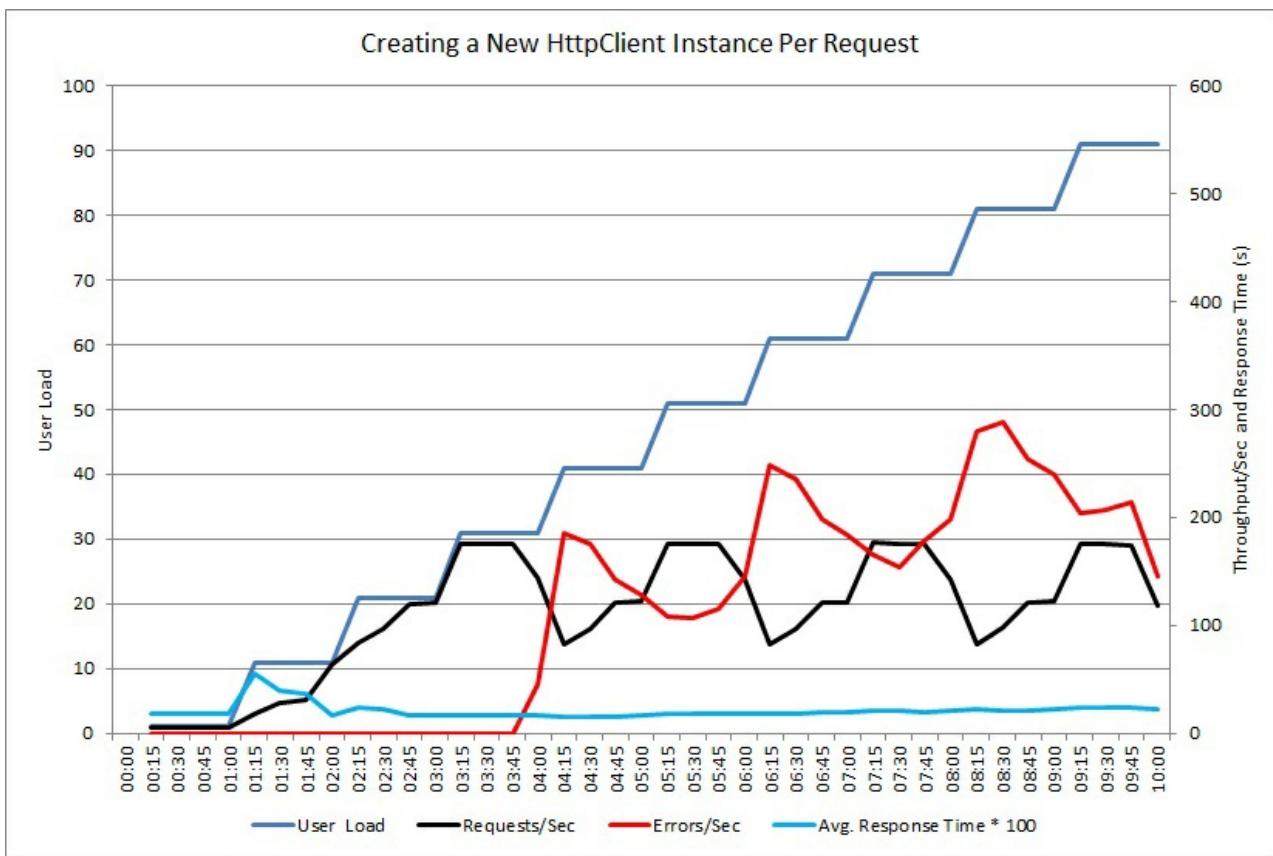
The next image shows data captured using thread profiling, over the same period corresponding as the previous image. The system spends a significant time opening socket connections, and even more time closing them and handling socket exceptions.



Performing load testing

Use load testing to simulate the typical operations that users might perform. This can help to identify which parts of a system suffer from resource exhaustion under varying loads. Perform these tests in a controlled environment rather than the production system. The following graph shows the throughput of requests handled by the

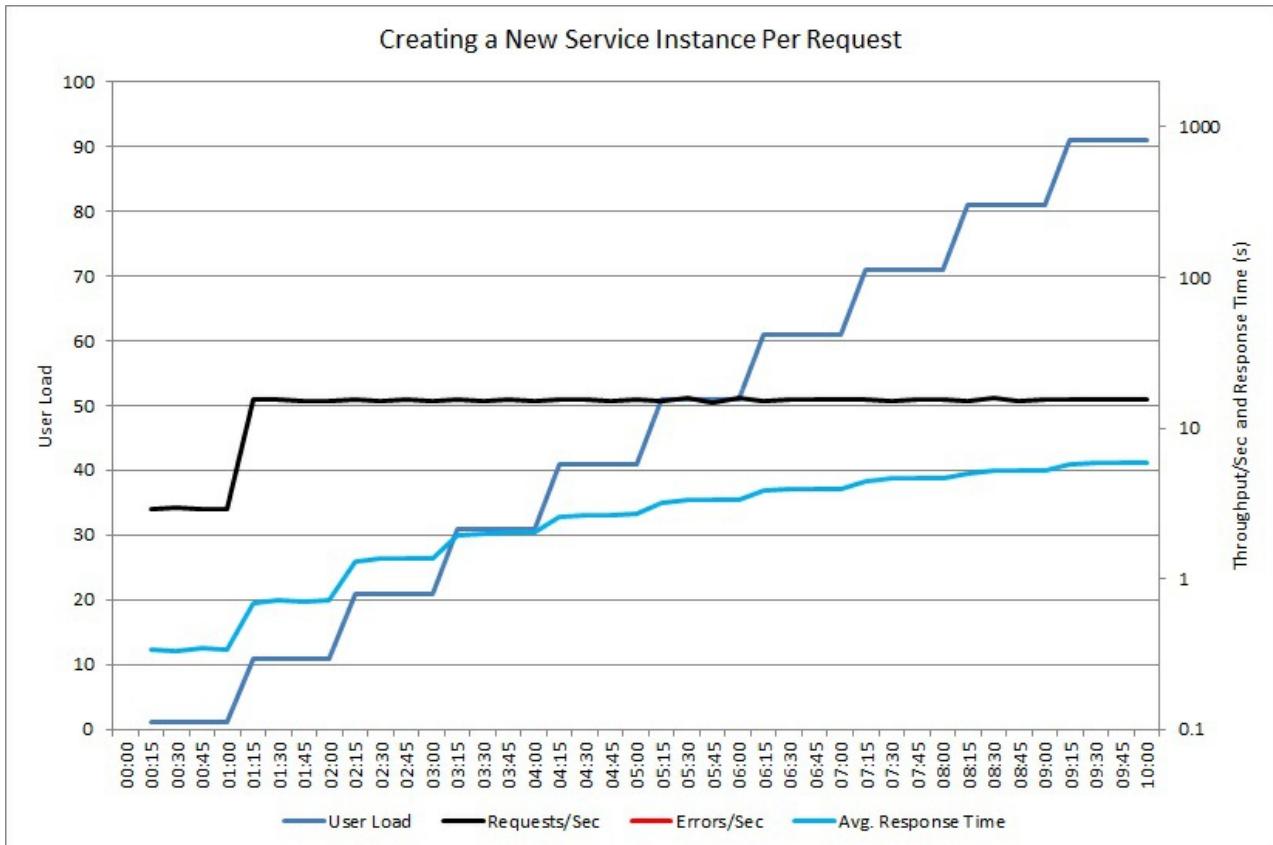
`NewHttpClientInstancePerRequest` controller as the user load increases to 100 concurrent users.



At first, the volume of requests handled per second increases as the workload increases. At about 30 users, however, the volume of successful requests reaches a limit, and the system starts to generate exceptions. From then on, the volume of exceptions gradually increases with the user load.

The load test reported these failures as HTTP 500 (Internal Server) errors. Reviewing the telemetry showed that these errors were caused by the system running out of socket resources, as more and more `HttpClient` objects were created.

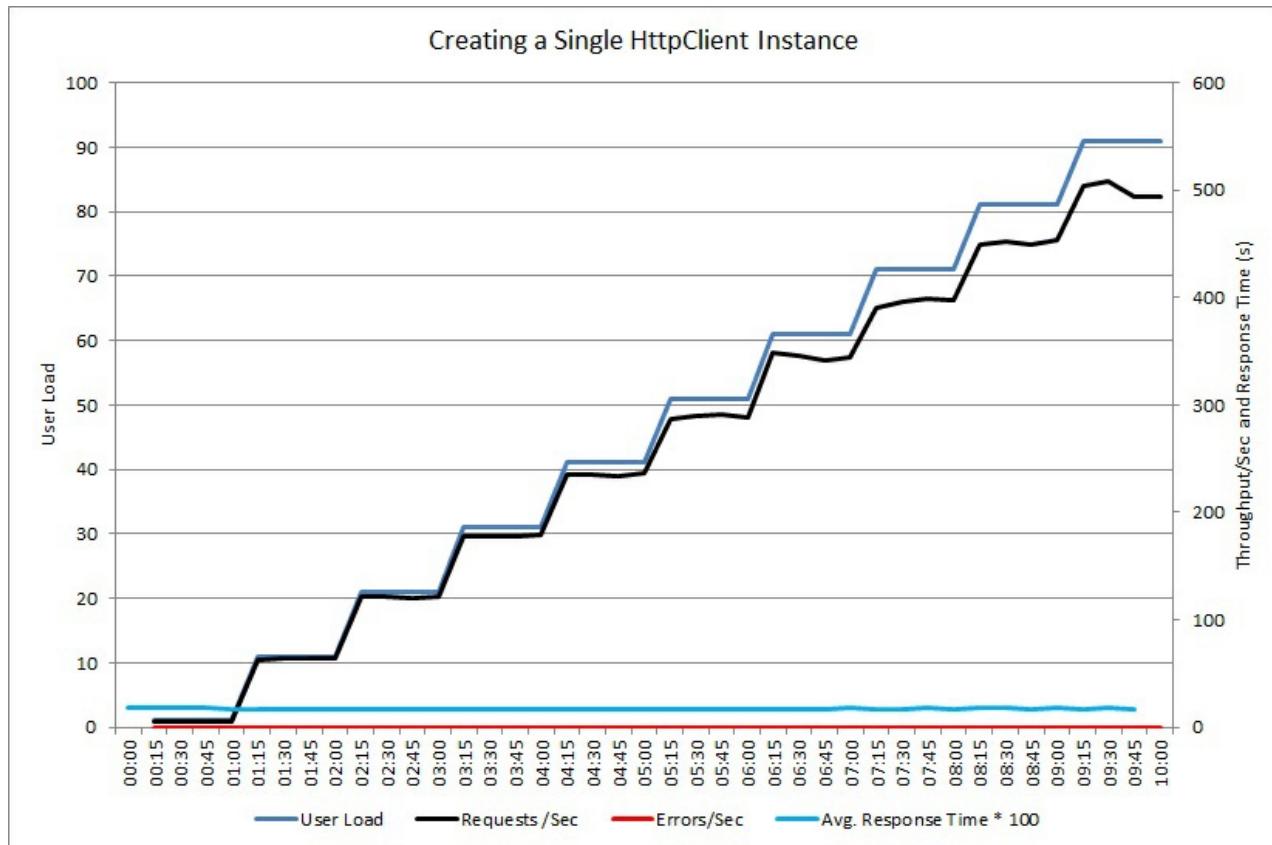
The next graph shows a similar test for a controller that creates the custom `ExpensiveToCreateService` object.



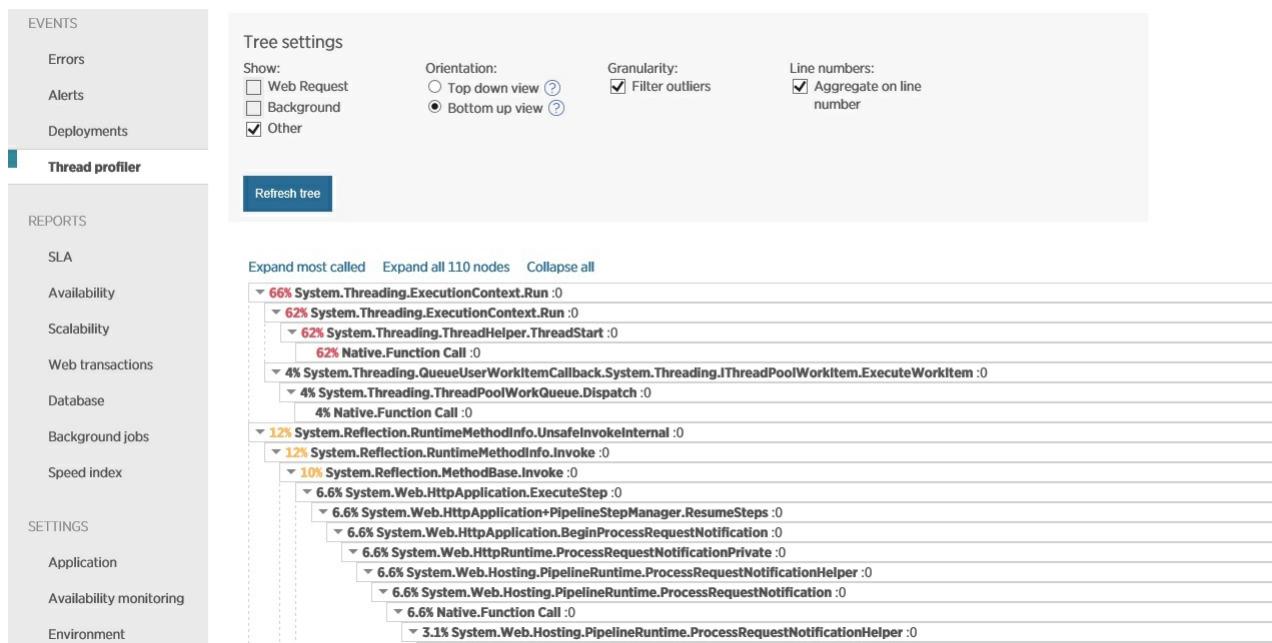
This time, the controller does not generate any exceptions, but throughput still reaches a plateau, while the average response time increases by a factor of 20. (The graph uses a logarithmic scale for response time and throughput.) Telemetry showed that creating new instances of the `ExpensiveToCreateService` was the main cause of the problem.

Implement the solution and verify the result

After switching the `GetProductAsync` method to share a single `HttpClient` instance, a second load test showed improved performance. No errors were reported, and the system was able to handle an increasing load of up to 500 requests per second. The average response time was cut in half, compared with the previous test.

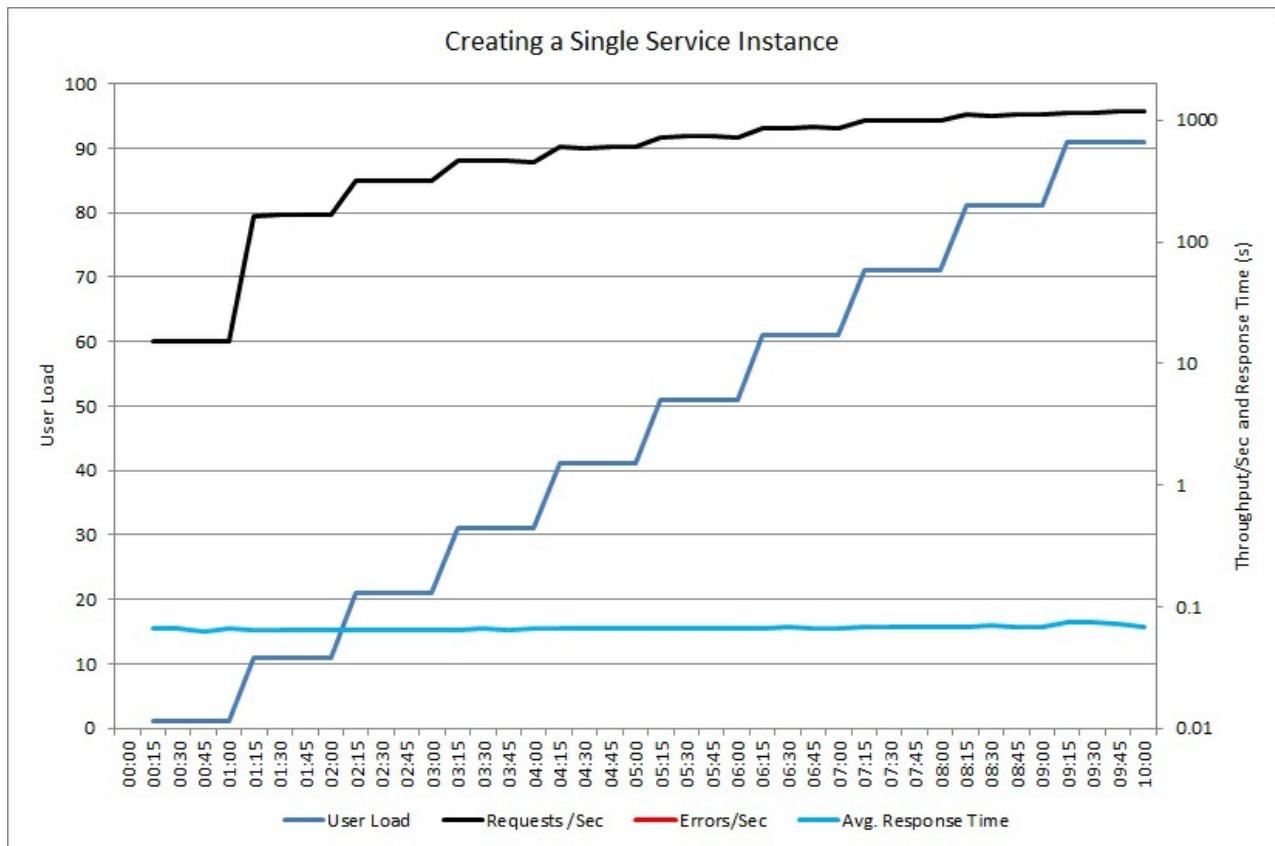


For comparison, the following image shows the stack trace telemetry. This time, the system spends most of its time performing real work, rather than opening and closing sockets.



The next graph shows a similar load test using a shared instance of the `ExpensiveToCreateService` object. Again, the

volume of handled requests increases in line with the user load, while the average response time remains low.



Monolithic Persistence antipattern

6/23/2017 • 6 minutes to read • [Edit Online](#)

Putting all of an application's data into a single data store can hurt performance, either because it leads to resource contention, or because the data store is not a good fit for some of the data.

Problem description

Historically, applications have often used a single data store, regardless of the different types of data that the application might need to store. Usually this was done to simplify the application design, or else to match the existing skill set of the development team.

Modern cloud-based systems often have additional functional and nonfunctional requirements, and need to store many heterogenous types of data, such as documents, images, cached data, queued messages, application logs, and telemetry. Following the traditional approach and putting all of this information into the same data store can hurt performance, for two main reasons:

- Storing and retrieving large amounts of unrelated data in the same data store can cause contention, which in turn leads to slow response times and connection failures.
- Whichever data store is chosen, it might not be the best fit for all of the different types of data, or it might not be optimized for the operations that the application performs.

The following example shows an ASP.NET Web API controller that adds a new record to a database and also records the result to a log. The log is held in the same database as the business data. You can find the complete sample [here](#).

```
public class MonoController : ApiController
{
    private static readonly string ProductionDb = ...;

    public async Task<IHttpActionResult> PostAsync([FromBody]string value)
    {
        await DataAccess.InsertPurchaseOrderHeaderAsync(ProductionDb);
        await DataAccess.LogAsync(ProductionDb, LogTableName);
        return Ok();
    }
}
```

The rate at which log records are generated will probably affect the performance of the business operations. And if another component, such as an application process monitor, regularly reads and processes the log data, that can also affect the business operations.

How to fix the problem

Separate data according to its use. For each data set, select a data store that best matches how that data set will be used. In the previous example, the application should be logging to a separate store from the database that holds business data:

```

public class PolyController : ApiController
{
    private static readonly string ProductionDb = ...;
    private static readonly string LogDb = ...;

    public async Task<IHttpActionResult> PostAsync([FromBody]string value)
    {
        await DataAccess.InsertPurchaseOrderHeaderAsync(ProductionDb);
        // Log to a different data store.
        await DataAccess.LogAsync(LogDb, LogTableName);
        return Ok();
    }
}

```

Considerations

- Separate data by the way it is used and how it is accessed. For example, don't store log information and business data in the same data store. These types of data have significantly different requirements and patterns of access. Log records are inherently sequential, while business data is more likely to require random access, and is often relational.
- Consider the data access pattern for each type of data. For example, store formatted reports and documents in a document database such as [Cosmos DB](#), but use [Azure Redis Cache](#) to cache temporary data.
- If you follow this guidance but still reach the limits of the database, you may need to scale up the database. Also consider scaling horizontally and partitioning the load across database servers. However, partitioning may require redesigning the application. For more information, see [Data partitioning](#).

How to detect the problem

The system will likely slow down dramatically and eventually fail, as the system runs out of resources such as database connections.

You can perform the following steps to help identify the cause.

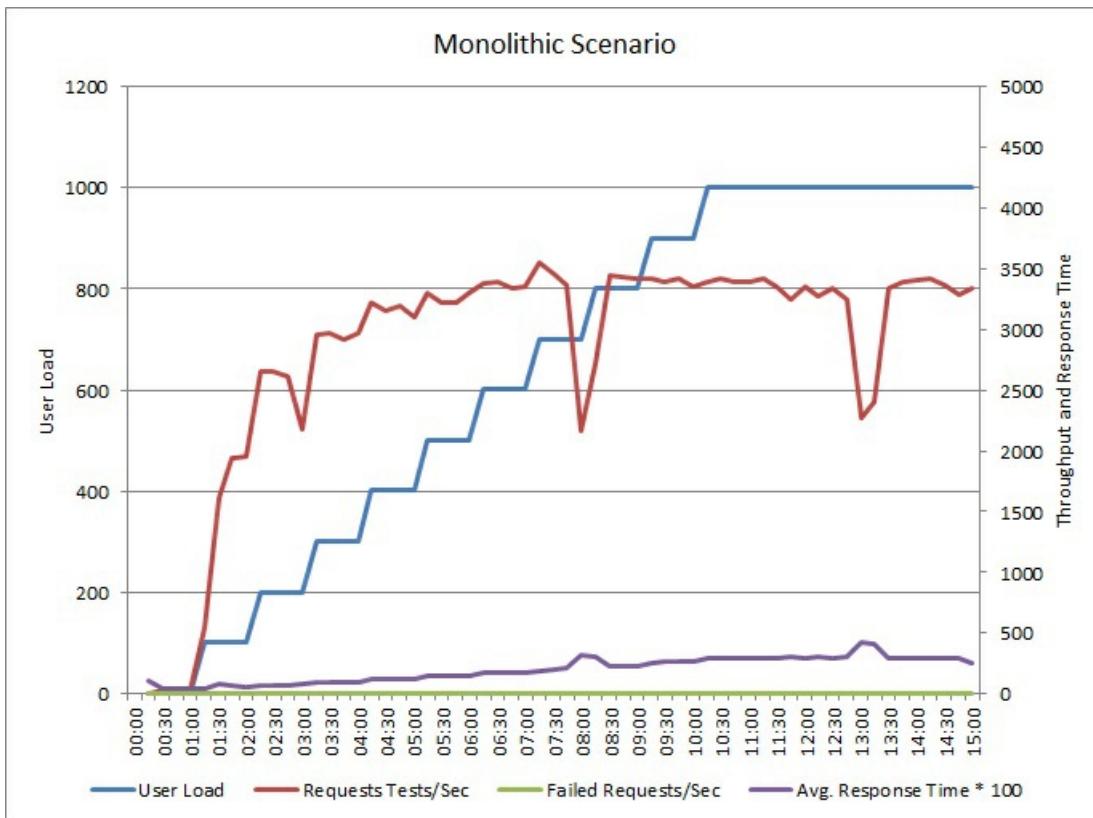
- Instrument the system to record the key performance statistics. Capture timing information for each operation, as well as the points where the application reads and writes data.
- If possible, monitor the system running for a few days in a production environment to get a real-world view of how the 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.
- Use the telemetry data to identify periods of poor performance.
- Identify which data stores were accessed during those periods.
- Identify data storage resources that might be experiencing contention.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Instrument and monitor the system

The following graph shows the results of load testing the sample application described earlier. The test used a step load of up to 1000 concurrent users.



As the load increases to 700 users, so does the throughput. But at that point, throughput levels off, and the system appears to be running at its maximum capacity. The average response gradually increases with user load, showing that the system can't keep up with demand.

Identify periods of poor performance

If you are monitoring the production system, you might notice patterns. For example, response times might drop off significantly at the same time each day. This could be caused by a regular workload or scheduled batch job, or just because the system has more users at certain times. You should focus on the telemetry data for these events.

Look for correlations between increased response times and increased database activity or I/O to shared resources. If there are correlations, it means the database might be a bottleneck.

Identify which data stores are accessed during those periods

The next graph shows the utilization of database throughput units (DTU) during the load test. (A DTU is a measure of available capacity, and is a combination of CPU utilization, memory allocation, I/O rate.) Utilization of DTUs quickly reached 100%. This is roughly the point where throughput peaked in the previous graph. Database utilization remained very high until the test finished. There is a slight drop toward the end, which could be caused by throttling, competition for database connections, or other factors.

Monitoring



Examine the telemetry for the data stores

Instrument the data stores to capture the low-level details of the activity. In the sample application, the data access statistics showed a high volume of insert operations performed against both the `PurchaseOrderHeader` table and the `MonoLog` table.

Query	Run Count	CPU ms/sec	Duration ms/sec
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 + 1 , ((CASE eq1.statement_end - statement_start_offset)*8)/2) AS [text]) AS XML) AS [text]	2	23	25
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 + 1 , ((CASE eq1.statement_end - statement_start_offset)*8)/2) AS [text]) AS XML) AS [text]	1	14	16
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 + 1 , ((CASE eq1.statement_end - statement_start_offset)*8)/2) AS [text]) AS XML) AS [text]	2	13	14
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 + 1 , ((CASE eq1.statement_end - statement_start_offset)*8)/2) AS [text]) AS XML) AS [text]	2	12	12
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 + 1 , ((CASE eq1.statement_end - statement_start_offset)*8)/2) AS [text]) AS XML) AS [text]	1	11	12
INSERT INTO Purchasing.PurchaseOrderHeader (RevisionNumber, Status, EmployeeID, VendorID, !)	21662	7	2155
select @UsedPageCount = ISNULL(SUM(reserved_page_count), 0) FROM sys.dm_db_partition_stats	2	6	14
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	13789	2	1094
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	10579	1	832
SELECT database_id FROM sys.databases WHERE name=N'AdventureWorks2012' and db_name()=	18	1	1
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	896	0	72
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	97	0	11
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	13	0	1
SELECT dtb.collation_name AS [Database_Collation], dtb.create_date AS [Database_CreationDate],	2	0	0
SELECT st.name AS [FederationSystemDataType_Name], st.user_type_id AS [FederationSystemData	2	0	0
select is_federation_member from sys.databases where name = N'AdventureWorks2012'	4	0	0
SELECT CAST(st.system_type_id AS int) AS [SystemDataType_BaseDataType], st.is_nullable AS [Syst	2	0	0
select is_federation_member from sys.databases where name = N'AdventureWorks2012'	1	0	0
SELECT CAST(CASE WHEN btst.name IN ('varchar', 'nvarchar', 'varbinary') AND st.max_length = -1 T	2	0	0
SELECT fed.name AS [Federation_Name], fed.federation_id AS [Federation_ID], DB_NAME() AS [Fec	2	0	0
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 + 1 , ((CASE eq1.statement_end - statement_start_offset)*8)/2) AS [text]) AS XML) AS [text]	1	0	0
INSERT INTO Purchasing.PurchaseOrderHeader (RevisionNumber, Status, EmployeeID, VendorID, !)	1	0	0

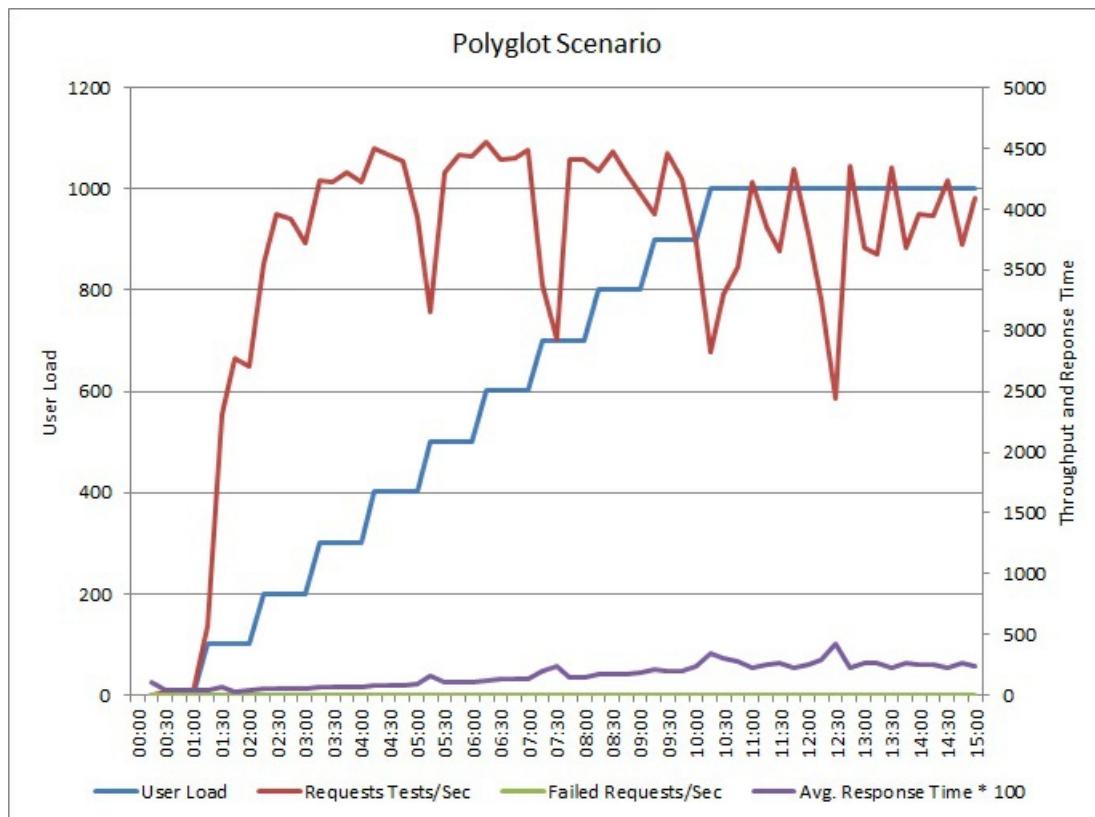
Identify resource contention

At this point, you can review the source code, focusing on the points where contended resources are accessed by the application. Look for situations such as:

- Data that is logically separate being written to the same store. Data such as logs, reports, and queued messages should not be held in the same database as business information.
- A mismatch between the choice of data store and the type of data, such as large blobs or XML documents in a relational database.
- Data with significantly different usage patterns that share the same store, such as high-write/low-read data being stored with low-write/high-read data.

Implement the solution and verify the result

The application was changed to write logs to a separate data store. Here are the load test results:



The pattern of throughput is similar to the earlier graph, but the point at which performance peaks is approximately 500 requests per second higher. The average response time is marginally lower. However, these statistics don't tell the full story. Telemetry for the business database shows that DTU utilization peaks at around 75%, rather than 100%.



Similarly, the maximum DTU utilization of the log database only reaches about 70%. The databases are no longer the limiting factor in the performance of the system.

Monitoring



Related resources

- [Choose the right data store](#)
- [Criteria for choosing a data store](#)
- [Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence](#)
- [Data partitioning](#)

No Caching antipattern

1/4/2018 • 8 minutes to read • [Edit Online](#)

In a cloud application that handles many concurrent requests, repeatedly fetching the same data can reduce performance and scalability.

Problem description

When data is not cached, it can cause a number of undesirable behaviors, including:

- Repeatedly fetching the same information from a resource that is expensive to access, in terms of I/O overhead or latency.
- Repeatedly constructing the same objects or data structures for multiple requests.
- Making excessive calls to a remote service that has a service quota and throttles clients past a certain limit.

In turn, these problems can lead to poor response times, increased contention in the data store, and poor scalability.

The following example uses Entity Framework to connect to a database. Every client request results in a call to the database, even if multiple requests are fetching exactly the same data. The cost of repeated requests, in terms of I/O overhead and data access charges, can accumulate quickly.

```
public class PersonRepository : IPersonRepository
{
    public async Task<Person> GetAsync(int id)
    {
        using (var context = new AdventureWorksContext())
        {
            return await context.People
                .Where(p => p.Id == id)
                .FirstOrDefaultAsync()
                .ConfigureAwait(false);
        }
    }
}
```

You can find the complete sample [here](#).

This antipattern typically occurs because:

- Not using a cache is simpler to implement, and it works fine under low loads. Caching makes the code more complicated.
- The benefits and drawbacks of using a cache are not clearly understood.
- There is concern about the overhead of maintaining the accuracy and freshness of cached data.
- An application was migrated from an on-premises system, where network latency was not an issue, and the system ran on expensive high-performance hardware, so caching wasn't considered in the original design.
- Developers aren't aware that caching is a possibility in a given scenario. For example, developers may not think of using ETags when implementing a web API.

How to fix the problem

The most popular caching strategy is the *on-demand* or *cache-aside* strategy.

- On read, the application tries to read the data from the cache. If the data isn't in the cache, the application retrieves it from the data source and adds it to the cache.
- On write, the application writes the change directly to the data source and removes the old value from the cache. It will be retrieved and added to the cache the next time it is required.

This approach is suitable for data that changes frequently. Here is the previous example updated to use the [Cache-Aside][cache-aside] pattern.

```
public class CachedPersonRepository : IPersonRepository
{
    private readonly PersonRepository _innerRepository;

    public CachedPersonRepository(PersonRepository innerRepository)
    {
        _innerRepository = innerRepository;
    }

    public async Task<Person> GetAsync(int id)
    {
        return await CacheService.GetAsync<Person>("p:" + id, () =>
_innerRepository.GetAsync(id)).ConfigureAwait(false);
    }
}

public class CacheService
{
    private static ConnectionMultiplexer _connection;

    public static async Task<T> GetAsync<T>(string key, Func<Task<T>> loadCache, double expirationTimeInMinutes)
    {
        IDatabase cache = Connection.GetDatabase();
        T value = await GetAsync<T>(cache, key).ConfigureAwait(false);
        if (value == null)
        {
            // Value was not found in the cache. Call the lambda to get the value from the database.
            value = await loadCache().ConfigureAwait(false);
            if (value != null)
            {
                // Add the value to the cache.
                await SetAsync(cache, key, value, expirationTimeInMinutes).ConfigureAwait(false);
            }
        }
        return value;
    }
}
```

Notice that the `GetAsync` method now calls the `CacheService` class, rather than calling the database directly. The `CacheService` class first tries to get the item from Azure Redis Cache. If the value isn't found in Redis Cache, the `CacheService` invokes a lambda function that was passed to it by the caller. The lambda function is responsible for fetching the data from the database. This implementation decouples the repository from the particular caching solution, and decouples the `CacheService` from the database.

Considerations

- If the cache is unavailable, perhaps because of a transient failure, don't return an error to the client. Instead, fetch the data from the original data source. However, be aware that while the cache is being recovered, the original data store could be swamped with requests, resulting in timeouts and failed connections. (After all, this is one of the motivations for using a cache in the first place.) Use a technique such as the [Circuit Breaker pattern](#) to avoid overwhelming the data source.

- Applications that cache nonstatic data should be designed to support eventual consistency.
- For web APIs, you can support client-side caching by including a Cache-Control header in request and response messages, and using ETags to identify versions of objects. For more information, see [API implementation](#).
- You don't have to cache entire entities. If most of an entity is static but only a small piece changes frequently, cache the static elements and retrieve the dynamic elements from the data source. This approach can help to reduce the volume of I/O being performed against the data source.
- In some cases, if volatile data is short-lived, it can be useful to cache it. For example, consider a device that continually sends status updates. It might make sense to cache this information as it arrives, and not write it to a persistent store at all.
- To prevent data from becoming stale, many caching solutions support configurable expiration periods, so that data is automatically removed from the cache after a specified interval. You may need to tune the expiration time for your scenario. Data that is highly static can stay in the cache for longer periods than volatile data that may become stale quickly.
- If the caching solution doesn't provide built-in expiration, you may need to implement a background process that occasionally sweeps the cache, to prevent it from growing without limits.
- Besides caching data from an external data source, you can use caching to save the results of complex computations. Before you do that, however, instrument the application to determine whether the application is really CPU bound.
- It might be useful to prime the cache when the application starts. Populate the cache with the data that is most likely to be used.
- Always include instrumentation that detects cache hits and cache misses. Use this information to tune caching policies, such what data to cache, and how long to hold data in the cache before it expires.
- If the lack of caching is a bottleneck, then adding caching may increase the volume of requests so much that the web front end becomes overloaded. Clients may start to receive HTTP 503 (Service Unavailable) errors. These are an indication that you should scale out the front end.

How to detect the problem

You can perform the following steps to help identify whether lack of caching is causing performance problems:

1. Review the application design. Take an inventory of all the data stores that the application uses. For each, determine whether the application is using a cache. If possible, determine how frequently the data changes. Good initial candidates for caching include data that changes slowly, and static reference data that is read frequently.
2. Instrument the application and monitor the live system to find out how frequently the application retrieves data or calculates information.
3. Profile the application in a test environment to capture low-level metrics about the overhead associated with data access operations or other frequently performed calculations.
4. Perform load testing in a test environment to identify how the system responds under a normal workload and under heavy load. Load testing should simulate the pattern of data access observed in the production environment using realistic workloads.
5. Examine the data access statistics for the underlying data stores and review how often the same data requests are repeated.

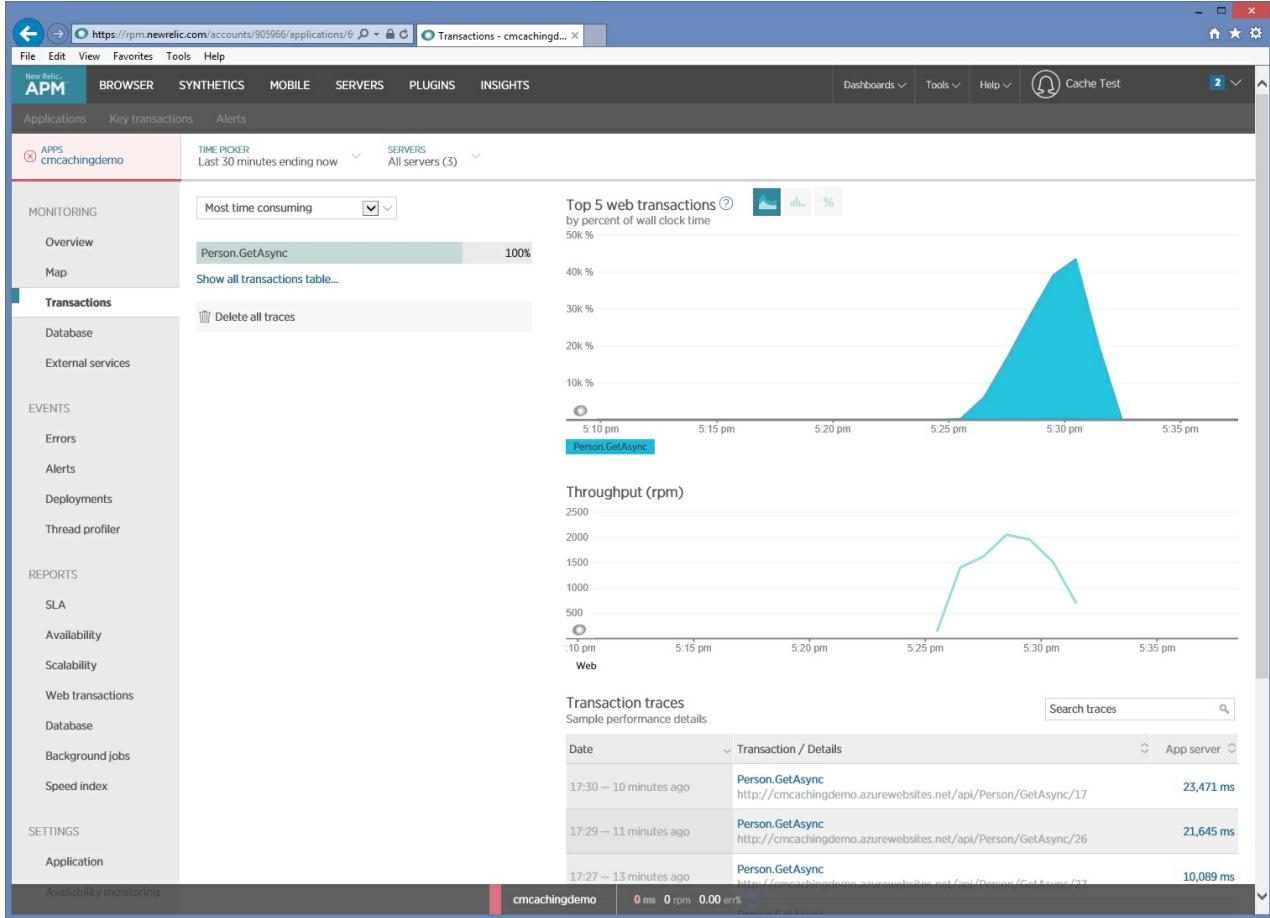
Example diagnosis

The following sections apply these steps to the sample application described earlier.

Instrument the application and monitor the live system

Instrument the application and monitor it to get information about the specific requests that users make while the application is in production.

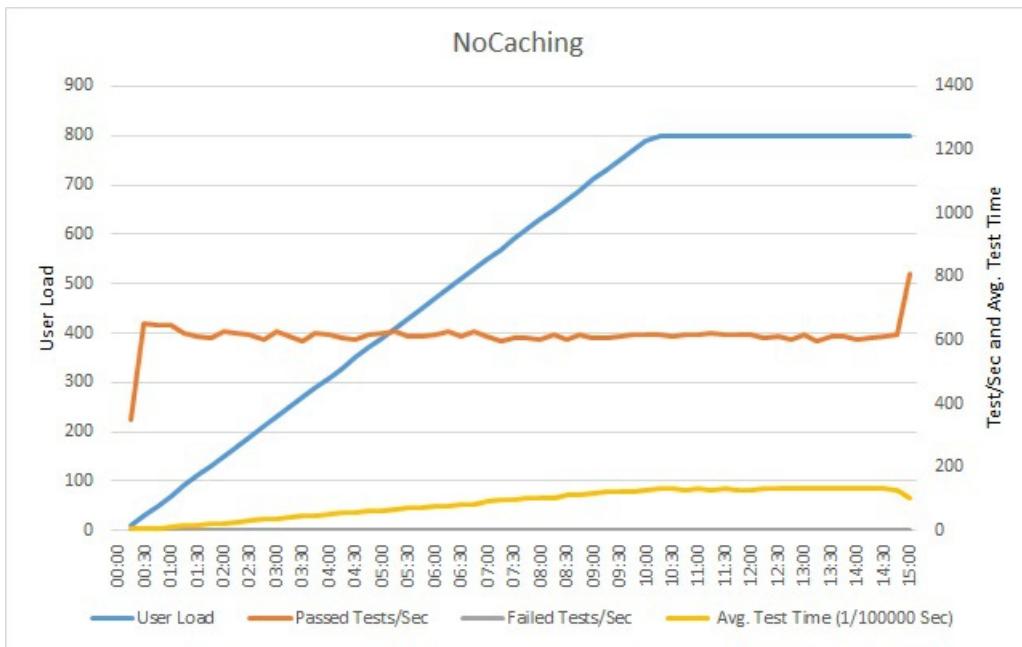
The following image shows monitoring data captured by [New Relic](#) during a load test. In this case, the only HTTP GET operation performed is `Person.GetAsync`. But in a live production environment, knowing the relative frequency that each request is performed can give you insight into which resources should be cached.



If you need a deeper analysis, you can use a profiler to capture low-level performance data in a test environment (not the production system). Look at metrics such as I/O request rates, memory usage, and CPU utilization. These metrics may show a large number of requests to a data store or service, or repeated processing that performs the same calculation.

Load test the application

The following graph shows the results of load testing the sample application. The load test simulates a step load of up to 800 users performing a typical series of operations.



The number of successful tests performed each second reaches a plateau, and additional requests are slowed as a result. The average test time steadily increases with the workload. The response time levels off once the user load peaks.

Examine data access statistics

Data access statistics and other information provided by a data store can give useful information, such as which queries are repeated most frequently. For example, in Microsoft SQL Server, the `sys.dm_exec_query_stats` management view has statistical information for recently executed queries. The text for each query is available in the `sys.dm_exec_query_plan` view. You can use a tool such as SQL Server Management Studio to run the following SQL query and determine how frequently queries are performed.

```
SELECT UseCounts, Text, Query_Plan
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)
```

The `UseCount` column in the results indicates how frequently each query is run. The following image shows that the third query was run more than 250,000 times, significantly more than any other query.

```

SELECT UseCounts, Text, Query_Plan
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)

```

100 % < Results Messages

UseCounts	Text	Query_Plan
1	SELECT UseCounts, Text, Query_Plan FROM sys.dm_exec_cached_plans WHERE query_plan IS NOT NULL	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	select is_federation_member from sys.databases where database_id = 1	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
256049	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1] WHERE [Extent1].[BusinessEntityId] = @p__linq_0	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
2	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
2	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
3	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
1	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
10	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
11	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
12	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
13	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
14	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
15	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
16	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
17	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
18	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
19	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
20	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
21	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
22	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
23	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">
24	(@p__linq_0 int)SELECT TOP (2) [Extent1].[BusinessEntityId], [Extent1].[FirstName], [Extent1].[LastName] FROM [Person].[Person] AS [Extent1]	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan">

Here is the SQL query that is causing so many database requests:

```

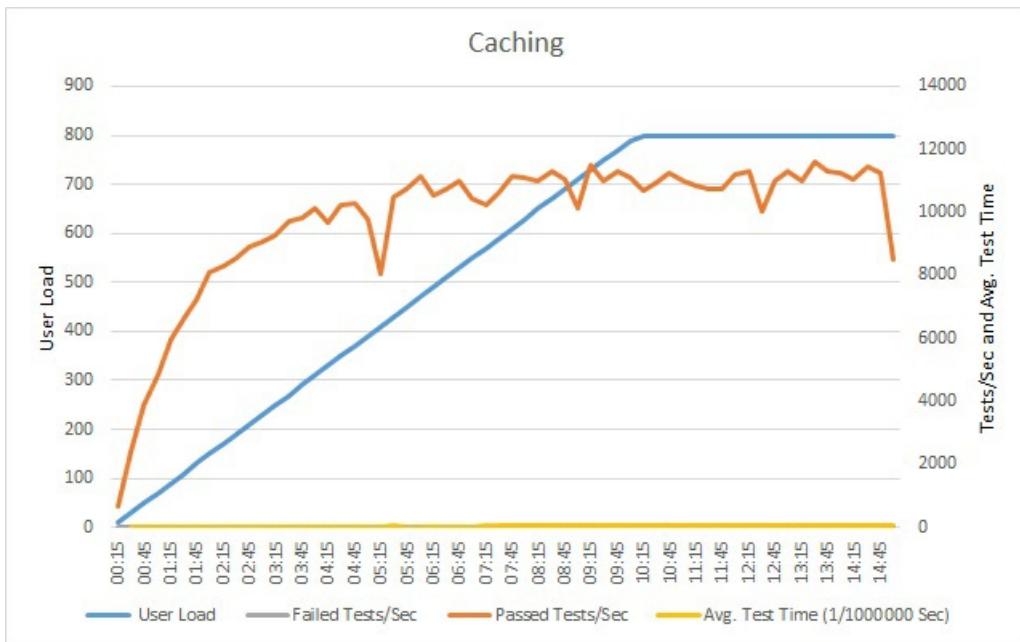
(@p__linq_0 int)SELECT TOP (2)
[Extent1].[BusinessEntityId] AS [BusinessEntityId],
[Extent1].[FirstName] AS [FirstName],
[Extent1].[LastName] AS [LastName]
FROM [Person].[Person] AS [Extent1]
WHERE [Extent1].[BusinessEntityId] = @p__linq_0

```

This is the query that Entity Framework generates in `GetByIdAsync` method shown earlier.

Implement the solution and verify the result

After you incorporate a cache, repeat the load tests and compare the results to the earlier load tests without a cache. Here are the load test results after adding a cache to the sample application.



The volume of successful tests still reaches a plateau, but at a higher user load. The request rate at this load is significantly higher than earlier. Average test time still increases with load, but the maximum response time is 0.05 ms, compared with 1ms earlier — a 20× improvement.

Related resources

- [API implementation best practices](#)
- [Cache-Aside Pattern](#)
- [Caching best practices](#)
- [Circuit Breaker pattern](#)

Synchronous I/O antipattern

6/23/2017 • 6 minutes to read • [Edit Online](#)

Blocking the calling thread while I/O completes can reduce performance and affect vertical scalability.

Problem description

A synchronous I/O operation blocks the calling thread while the I/O completes. The calling thread enters a wait state and is unable to perform useful work during this interval, wasting processing resources.

Common examples of I/O include:

- Retrieving or persisting data to a database or any type of persistent storage.
- Sending a request to a web service.
- Posting a message or retrieving a message from a queue.
- Writing to or reading from a local file.

This antipattern typically occurs because:

- It appears to be the most intuitive way to perform an operation.
- The application requires a response from a request.
- The application uses a library that only provides synchronous methods for I/O.
- An external library performs synchronous I/O operations internally. A single synchronous I/O call can block an entire call chain.

The following code uploads a file to Azure blob storage. There are two places where the code blocks waiting for synchronous I/O, the `CreateIfNotExists` method and the `UploadFromStream` method.

```
var blobClient = storageAccount.CreateCloudBlobClient();
var container = blobClient.GetContainerReference("uploadedfiles");

container.CreateIfNotExists();
var blockBlob = container.GetBlockBlobReference("myblob");

// Create or overwrite the "myblob" blob with contents from a local file.
using (var fileStream = File.OpenRead(HostingEnvironment.MapPath("~/FileToUpload.txt")))
{
    blockBlob.UploadFromStream(fileStream);
}
```

Here's an example of waiting for a response from an external service. The `GetUserProfile` method calls a remote service that returns a `UserProfile`.

```

public interface IUserProfileService
{
    UserProfile GetUserProfile();
}

public class SyncController : ApiController
{
    private readonly IUserProfileService _userProfileService;

    public SyncController()
    {
        _userProfileService = new FakeUserProfileService();
    }

    // This is a synchronous method that calls the synchronous GetUserProfile method.
    public UserProfile GetUserProfile()
    {
        return _userProfileService.GetUserProfile();
    }
}

```

You can find the complete code for both of these examples [here](#).

How to fix the problem

Replace synchronous I/O operations with asynchronous operations. This frees the current thread to continue performing meaningful work rather than blocking, and helps improve the utilization of compute resources. Performing I/O asynchronously is particularly efficient for handling an unexpected surge in requests from client applications.

Many libraries provide both synchronous and asynchronous versions of methods. Whenever possible, use the asynchronous versions. Here is the asynchronous version of the previous example that uploads a file to Azure blob storage.

```

var blobClient = storageAccount.CreateCloudBlobClient();
var container = blobClient.GetContainerReference("uploadedfiles");

await container.CreateIfNotExistsAsync();

var blockBlob = container.GetBlockBlobReference("myblob");

// Create or overwrite the "myblob" blob with contents from a local file.
using (var fileStream = File.OpenRead(HostingEnvironment.MapPath("~/FileToUpload.txt")))
{
    await blockBlob.UploadFromStreamAsync(fileStream);
}

```

The `await` operator returns control to the calling environment while the asynchronous operation is performed. The code after this statement acts as a continuation that runs when the asynchronous operation has completed.

A well designed service should also provide asynchronous operations. Here is an asynchronous version of the web service that returns user profiles. The `GetUserProfileAsync` method depends on having an asynchronous version of the User Profile service.

```

public interface IUserProfileService
{
    Task<UserProfile> GetUserProfileAsync();
}

public class AsyncController : ApiController
{
    private readonly IUserProfileService _userProfileService;

    public AsyncController()
    {
        _userProfileService = new FakeUserProfileService();
    }

    // This is an synchronous method that calls the Task based GetUserProfileAsync method.
    public Task<UserProfile> GetUserProfileAsync()
    {
        return _userProfileService.GetUserProfileAsync();
    }
}

```

For libraries that don't provide asynchronous versions of operations, it may be possible to create asynchronous wrappers around selected synchronous methods. Follow this approach with caution. While it may improve responsiveness on the thread that invokes the asynchronous wrapper, it actually consumes more resources. An extra thread may be created, and there is overhead associated with synchronizing the work done by this thread. Some tradeoffs are discussed in this blog post: [Should I expose asynchronous wrappers for synchronous methods?](#)

Here is an example of an asynchronous wrapper around a synchronous method.

```

// Asynchronous wrapper around synchronous library method
private async Task<int> LibraryIOOperationAsync()
{
    return await Task.Run(() => LibraryIOperation());
}

```

Now the calling code can await on the wrapper:

```

// Invoke the asynchronous wrapper using a task
await LibraryIOperationAsync();

```

Considerations

- I/O operations that are expected to be very short lived and are unlikely to cause contention might be more performant as synchronous operations. An example might be reading small files on an SSD drive. The overhead of dispatching a task to another thread, and synchronizing with that thread when the task completes, might outweigh the benefits of asynchronous I/O. However, these cases are relatively rare, and most I/O operations should be done asynchronously.
- Improving I/O performance may cause other parts of the system to become bottlenecks. For example, unblocking threads might result in a higher volume of concurrent requests to shared resources, leading in turn to resource starvation or throttling. If that becomes a problem, you might need to scale out the number of web servers or partition data stores to reduce contention.

How to detect the problem

For users, the application may seem unresponsive or appear to hang periodically. The application might fail with

timeout exceptions. These failures could also return HTTP 500 (Internal Server) errors. On the server, incoming client requests might be blocked until a thread becomes available, resulting in excessive request queue lengths, manifested as HTTP 503 (Service Unavailable) errors.

You can perform the following steps to help identify the problem:

1. Monitor the production system and determine whether blocked worker threads are constraining throughput.
2. If requests are being blocked due to lack of threads, review the application to determine which operations may be performing I/O synchronously.
3. Perform controlled load testing of each operation that is performing synchronous I/O, to find out whether those operations are affecting system performance.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Monitor web server performance

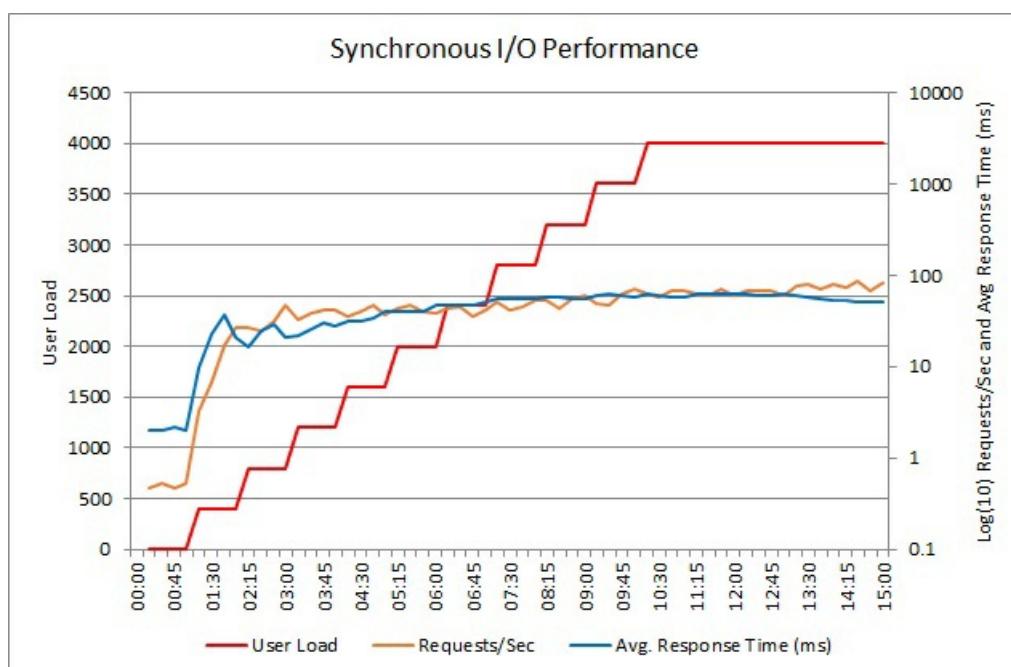
For Azure web applications and web roles, it's worth monitoring the performance of the IIS web server. In particular, pay attention to the request queue length to establish whether requests are being blocked waiting for available threads during periods of high activity. You can gather this information by enabling Azure diagnostics. For more information, see:

- [Monitor Apps in Azure App Service](#)
- [Create and use performance counters in an Azure application](#)

Instrument the application to see how requests are handled once they have been accepted. Tracing the flow of a request can help to identify whether it is performing slow-running calls and blocking the current thread. Thread profiling can also highlight requests that are being blocked.

Load test the application

The following graph shows the performance of the synchronous `GetUserProfile` method shown earlier, under varying loads of up to 4000 concurrent users. The application is an ASP.NET application running in an Azure Cloud Service web role.



The synchronous operation is hard-coded to sleep for 2 seconds, to simulate synchronous I/O, so the minimum

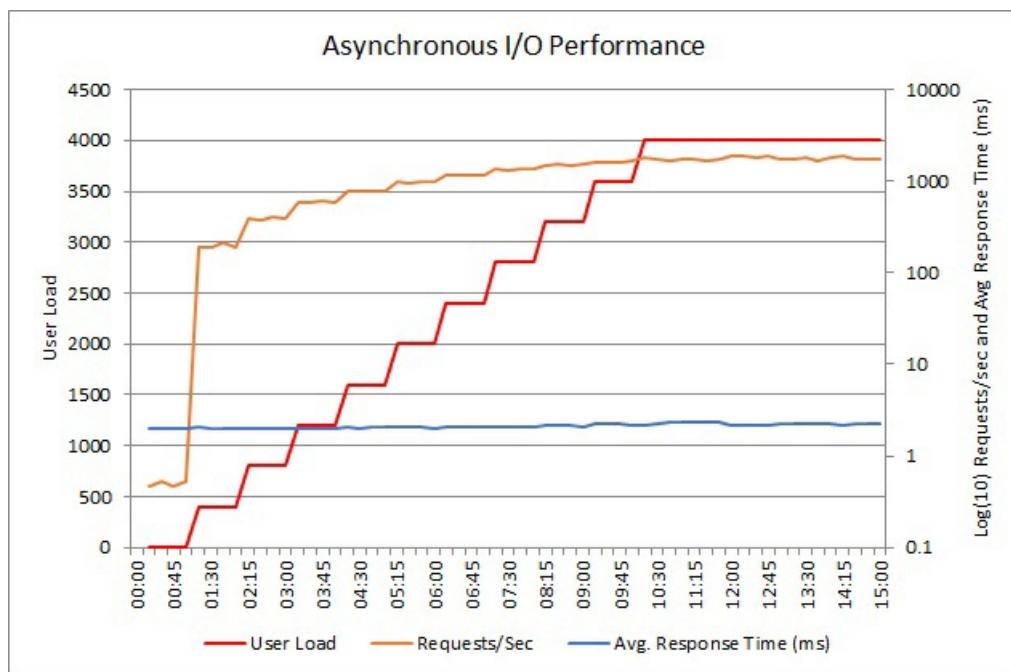
response time is slightly over 2 seconds. When the load reaches approximately 2500 concurrent users, the average response time reaches a plateau, although the volume of requests per second continues to increase. Note that the scale for these two measures is logarithmic. The number of requests per second doubles between this point and the end of the test.

In isolation, it's not necessarily clear from this test whether the synchronous I/O is a problem. Under heavier load, the application may reach a tipping point where the web server can no longer process requests in a timely manner, causing client applications to receive time-out exceptions.

Incoming requests are queued by the IIS web server and handed to a thread running in the ASP.NET thread pool. Because each operation performs I/O synchronously, the thread is blocked until the operation completes. As the workload increases, eventually all of the ASP.NET threads in the thread pool are allocated and blocked. At that point, any further incoming requests must wait in the queue for an available thread. As the queue length grows, requests start to time out.

Implement the solution and verify the result

The next graph shows the results from load testing the asynchronous version of the code.



Throughput is far higher. Over the same duration as the previous test, the system successfully handles a nearly tenfold increase in throughput, as measured in requests per second. Moreover, the average response time is relatively constant and remains approximately 25 times smaller than the previous test.

Availability checklist

1/19/2018 • 10 minutes to read • [Edit Online](#)

Availability is the proportion of time that a system is functional and working, and is one of the [pillars of software quality](#). Use this checklist to review your application architecture from an availability standpoint.

Application design

Avoid any single point of failure. All components, services, resources, and compute instances should be deployed as multiple instances to prevent a single point of failure from affecting availability. This includes authentication mechanisms. Design the application to be configurable to use multiple instances, and to automatically detect failures and redirect requests to non-failed instances where the platform does not do this automatically.

Decompose workloads by service-level objective. If a service is composed of critical and less-critical workloads, manage them differently and specify the service features and number of instances to meet their availability requirements.

Minimize and understand service dependencies. Minimize the number of different services used where possible, and ensure you understand all of the feature and service dependencies that exist in the system. This includes the nature of these dependencies, and the impact of failure or reduced performance in each one on the overall application. See [Defining your resiliency requirements](#).

Design tasks and messages to be idempotent where possible. An operation is idempotent if it can be repeated multiple times and produce the same result. Idempotency can ensure that duplicated requests don't cause problems. Message consumers and the operations they carry out should be idempotent so that repeating a previously executed operation does not render the results invalid. This may mean detecting duplicated messages, or ensuring consistency by using an optimistic approach to handling conflicts.

Use a message broker that implements high availability for critical transactions. Many cloud applications use messaging to initiate tasks that are performed asynchronously. To guarantee delivery of messages, the messaging system should provide high availability. [Azure Service Bus Messaging](#) implements *at least once* semantics. This means that a message posted to a queue will not be lost, although duplicate copies may be delivered under certain circumstances. If message processing is idempotent (see the previous item), repeated delivery should not be a problem.

Design applications to gracefully degrade. The load on an application may exceed the capacity of one or more parts, causing reduced availability and failed connections. Scaling can help to alleviate this, but it may reach a limit imposed by other factors, such as resource availability or cost. When an application reaches a resource limit, it should take appropriate action to minimize the impact for the user. For example, in an ecommerce system, if the order-processing subsystem is under strain or fails, it can be temporarily disabled while allowing other functionality, such as browsing the product catalog. It might be appropriate to postpone requests to a failing subsystem, for example still enabling customers to submit orders but saving them for later processing, when the orders subsystem is available again.

Gracefully handle rapid burst events. Most applications need to handle varying workloads over time. Auto-scaling can help to handle the load, but it may take some time for additional instances to come online and handle requests. Prevent sudden and unexpected bursts of activity from overwhelming the application: design it to queue requests to the services it uses and degrade gracefully when queues are near to full capacity. Ensure that there is sufficient performance and capacity available under non-burst conditions to drain the queues and handle outstanding requests. For more information, see the [Queue-Based Load Leveling Pattern](#).

Deployment and maintenance

Deploy multiple instances of services. If your application depends on a single instance of a service, it creates a single point of failure. Provisioning multiple instances improves both resiliency and scalability. For [Azure App Service](#), select an [App Service Plan](#) that offers multiple instances. For Azure Cloud Services, configure each of your roles to use [multiple instances](#). For [Azure Virtual Machines \(VMs\)](#), ensure that your VM architecture includes more than one VM and that each VM is included in an [availability set](#).

Consider deploying your application across multiple regions. If your application is deployed to a single region, in the rare event the entire region becomes unavailable, your application will also be unavailable. This may be unacceptable under the terms of your application's SLA. If so, consider deploying your application and its services across multiple regions.

Automate and test deployment and maintenance tasks. Distributed applications consist of multiple parts that must work together. Deployment should be automated, using tested and proven mechanisms such as scripts. These can update and validate configuration, and automate the deployment process. Use [Azure Resource Manager templates](#) to provision Azure resource. Also use automated techniques to perform application updates. It is vital to test all of these processes fully to ensure that errors do not cause additional downtime. All deployment tools must have suitable security restrictions to protect the deployed application; define and enforce deployment policies carefully and minimize the need for human intervention.

Use staging and production features of the platform.. For example, Azure App Service supports [deployment slots](#), which you can use to stage a deployment before swapping it to production. Azure Service Fabric supports [rolling upgrades](#) to application services.

Place virtual machines (VMs) in an availability set. To maximize availability, create multiple instances of each VM role and place these instances in the same availability set. If have multiple VMs that serve different roles, such as different application tiers, create an availability set for each VM role. For example, create an availability set for the web tier and another for the data tier.

Data management

Geo-replicate data in Azure Storage. Data in Azure Storage is automatically replicated within in a datacenter. For even higher availability, use Read-access geo-redundant storage (-RAGRS), which replicates your data to a secondary region and provides read-only access to the data in the secondary location. The data is durable even in the case of a complete regional outage or a disaster. For more information, see [Azure Storage replication](#).

Geo-replicate databases. Azure SQL Database and Cosmos DB both support geo-replication, which enables you to configure secondary database replicas in other regions. Secondary databases are available for querying and for failover in the case of a data center outage or the inability to connect to the primary database. For more information, see [Failover groups and active geo-replication \(SQL Database\)](#) and [How to distribute data globally with Azure Cosmos DB](#).

Use optimistic concurrency and eventual consistency. Transactions that block access to resources through locking (pessimistic concurrency) can cause poor performance and considerably reduce availability. These problems can become especially acute in distributed systems. In many cases, careful design and techniques such as partitioning can minimize the chances of conflicting updates occurring. Where data is replicated, or is read from a separately updated store, the data will only be eventually consistent. But the advantages usually far outweigh the impact on availability of using transactions to ensure immediate consistency.

Use periodic backup and point-in-time restore. Regularly and automatically back up data that is not preserved elsewhere, and verify you can reliably restore both the data and the application itself should a failure occur. Ensure that backups meet your Recovery Point Objective (RPO). Data replication is not a backup feature, because human error or malicious operations can corrupt data across all the replicas. The backup process must be secure to protect the data in transit and in storage. Databases or parts of a data store can usually be recovered to a previous point in

time by using transaction logs. For more information, see [Recover from data corruption or accidental deletion](#)

Errors and failures

Configure request timeouts. Services and resources may become unavailable, causing requests to fail. Ensure that the timeouts you apply are appropriate for each service or resource as well as the client that is accessing them. In some cases, you might allow a longer timeout for a particular instance of a client, depending on the context and other actions that the client is performing. Very short timeouts may cause excessive retry operations for services and resources that have considerable latency. Very long timeouts can cause blocking if a large number of requests are queued, waiting for a service or resource to respond.

Retry failed operations caused by transient faults. Design a retry strategy for access to all services and resources where they do not inherently support automatic connection retry. Use a strategy that includes an increasing delay between retries as the number of failures increases, to prevent overloading of the resource and to allow it to gracefully recover and handle queued requests. Continual retries with very short delays are likely to exacerbate the problem. For more information, see [Retry guidance for specific services](#).

Implement circuit breaking to avoid cascading failures. There may be situations in which transient or other faults, ranging in severity from a partial loss of connectivity to the complete failure of a service, take much longer than expected to return to normal. If a service is very busy, failure in one part of the system may lead to cascading failures, and result in many operations becoming blocked while holding onto critical system resources such as memory, threads, and database connections. Instead of continually retrying an operation that is unlikely to succeed, the application should quickly accept that the operation has failed, and gracefully handle this failure. Use the Circuit Breaker pattern to reject requests for specific operations for defined periods. For more information, see [Circuit Breaker Pattern](#).

Compose or fall back to multiple components. Design applications to use multiple instances without affecting operation and existing connections where possible. Use multiple instances and distribute requests between them, and detect and avoid sending requests to failed instances, in order to maximize availability.

Fall back to a different service or workflow. For example, if writing to SQL Database fails, temporarily store data in blob storage or Redis Cache. Provide a way to replay the writes to SQL Database when the service becomes available. In some cases, a failed operation may have an alternative action that allows the application to continue to work even when a component or service fails. If possible, detect failures and redirect requests to other services that can offer a suitable alternative functionality, or to back up or reduced functionality instances that can maintain core operations while the primary service is offline.

Monitoring and disaster recovery

Provide rich instrumentation for likely failures and failure events to report the situation to operations staff. For failures that are likely but have not yet occurred, provide sufficient data to enable operations staff to determine the cause, mitigate the situation, and ensure that the system remains available. For failures that have already occurred, the application should return an appropriate error message to the user but attempt to continue running, albeit with reduced functionality. In all cases, the monitoring system should capture comprehensive details to enable operations staff to effect a quick recovery, and if necessary, for designers and developers to modify the system to prevent the situation from arising again.

Monitor system health by implementing checking functions. The health and performance of an application can degrade over time, without being noticeable until it fails. Implement probes or check functions that are executed regularly from outside the application. These checks can be as simple as measuring response time for the application as a whole, for individual parts of the application, for individual services that the application uses, or for individual components. Check functions can execute processes to ensure they produce valid results, measure latency and check availability, and extract information from the system.

Regularly test all failover and fallback systems. Changes to systems and operations may affect failover and

fallback functions, but the impact may not be detected until the main system fails or becomes overloaded. Test it before it is required to compensate for a live problem at runtime.

Test the monitoring systems. Automated failover and fallback systems, and manual visualization of system health and performance by using dashboards, all depend on monitoring and instrumentation functioning correctly. If these elements fail, miss critical information, or report inaccurate data, an operator might not realize that the system is unhealthy or failing.

Track the progress of long-running workflows and retry on failure. Long-running workflows are often composed of multiple steps. Ensure that each step is independent and can be retried to minimize the chance that the entire workflow will need to be rolled back, or that multiple compensating transactions need to be executed. Monitor and manage the progress of long-running workflows by implementing a pattern such as [Scheduler Agent Supervisor Pattern](#).

Plan for disaster recovery. Create an accepted, fully-tested plan for recovery from any type of failure that may affect system availability. Choose a multi-site disaster recovery architecture for any mission-critical applications. Identify a specific owner of the disaster recovery plan, including automation and testing. Ensure the plan is well-documented, and automate the process as much as possible. Establish a backup strategy for all reference and transactional data, and test the restoration of these backups regularly. Train operations staff to execute the plan, and perform regular disaster simulations to validate and improve the plan.

DevOps Checklist

4/11/2018 • 14 minutes to read • [Edit Online](#)

DevOps is the integration of development, quality assurance, and IT operations into a unified culture and set of processes for delivering software. Use this checklist as a starting point to assess your DevOps culture and process.

Culture

Ensure business alignment across organizations and teams. Conflicts over resources, purpose, goals, and priorities within an organization can be a risk to successful operations. Ensure that the business, development, and operations teams are all aligned.

Ensure the entire team understands the software lifecycle. Your team needs to understand the overall lifecycle of the application, and which part of the lifecycle the application is currently in. This helps all team members know what they should be doing now, and what they should be planning and preparing for in the future.

Reduce cycle time. Aim to minimize the time it takes to move from ideas to usable developed software. Limit the size and scope of individual releases to keep the test burden low. Automate the build, test, configuration, and deployment processes whenever possible. Clear any obstacles to communication among developers, and between developers and operations.

Review and improve processes. Your processes and procedures, both automated and manual, are never final. Set up regular reviews of current workflows, procedures, and documentation, with a goal of continual improvement.

Do proactive planning. Proactively plan for failure. Have processes in place to quickly identify issues when they occur, escalate to the correct team members to fix, and confirm resolution.

Learn from failures. Failures are inevitable, but it's important to learn from failures to avoid repeating them. If an operational failure occurs, triage the issue, document the cause and solution, and share any lessons that were learned. Whenever possible, update your build processes to automatically detect that kind of failure in the future.

Optimize for speed and collect data. Every planned improvement is a hypothesis. Work in the smallest increments possible. Treat new ideas as experiments. Instrument the experiments so that you can collect production data to assess their effectiveness. Be prepared to fail fast if the hypothesis is wrong.

Allow time for learning. Both failures and successes provide good opportunities for learning. Before moving on to new projects, allow enough time to gather the important lessons, and make sure those lessons are absorbed by your team. Also give the team the time to build skills, experiment, and learn about new tools and techniques.

Document operations. Document all tools, processes, and automated tasks with the same level of quality as your product code. Document the current design and architecture of any systems you support, along with recovery processes and other maintenance procedures. Focus on the steps you actually perform, not theoretically optimal processes. Regularly review and update the documentation. For code, make sure that meaningful comments are included, especially in public APIs, and use tools to automatically generate code documentation whenever possible.

Share knowledge. Documentation is only useful if people know that it exists and can find it. Ensure the documentation is organized and easily discoverable. Be creative: Use brown bags (informal presentations), videos, or newsletters to share knowledge.

Development

Provide developers with production-like environments. If development and test environments don't match the production environment, it is hard to test and diagnose problems. Therefore, keep development and test

environments as close to the production environment as possible. Make sure that test data is consistent with the data used in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

Ensure that all authorized team members can provision infrastructure and deploy the application.

Setting up production-like resources and deploying the application should not involve complicated manual tasks or detailed technical knowledge of the system. Anyone with the right permissions should be able to create or deploy production-like resources without going to the operations team.

This recommendation doesn't imply that anyone can push live updates to the production deployment. It's about reducing friction for the development and QA teams to create production-like environments.

Instrument the application for insight. To understand the health of your application, you need to know how it's performing and whether it's experiencing any errors or problems. Always include instrumentation as a design requirement, and build the instrumentation into the application from the start. Instrumentation must include event logging for root cause analysis, but also telemetry and metrics to monitor the overall health and usage of the application.

Track your technical debt. In many projects, release schedules can get prioritized over code quality to one degree or another. Always keep track when this occurs. Document any shortcuts or other nonoptimal implementations, and schedule time in the future to revisit these issues.

Consider pushing updates directly to production. To reduce the overall release cycle time, consider pushing properly tested code commits directly to production. Use [feature toggles](#) to control which features are enabled. This allows you to move from development to release quickly, using the toggles to enable or disable features. Toggles are also useful when performing tests such as [canary releases](#), where a particular feature is deployed to a subset of the production environment.

Testing

Automate testing. Manually testing software is tedious and susceptible to error. Automate common testing tasks and integrate the tests into your build processes. Automated testing ensures consistent test coverage and reproducibility. Integrated UI tests should also be performed by an automated tool. Azure offers development and test resources that can help you configure and execute testing. For more information, see [Development and test](#).

Test for failures. If a system can't connect to a service, how does it respond? Can it recover once the service is available again? Make fault injection testing a standard part of review on test and staging environments. When your test process and practices are mature, consider running these tests in production.

Test in production. The release process doesn't end with deployment to production. Have tests in place to ensure that deployed code works as expected. For deployments that are infrequently updated, schedule production testing as a regular part of maintenance.

Automate performance testing to identify performance issues early. The impact of a serious performance issue can be just as severe as a bug in the code. While automated functional tests can prevent application bugs, they might not detect performance problems. Define acceptable performance goals for metrics like latency, load times, and resource usage. Include automated performance tests in your release pipeline, to make sure the application meets those goals.

Perform capacity testing. An application might work fine under test conditions, and then have problems in production due to scale or resource limitations. Always define the maximum expected capacity and usage limits. Test to make sure the application can handle those limits, but also test what happens when those limits are exceeded. Capacity testing should be performed at regular intervals.

After the initial release, you should run performance and capacity tests whenever updates are made to production code. Use historical data to fine tune tests and to determine what types of tests need to be performed.

Perform automated security penetration testing. Ensuring your application is secure is as important as testing any other functionality. Make automated penetration testing a standard part of the build and deployment process. Schedule regular security tests and vulnerability scanning on deployed applications, monitoring for open ports, endpoints, and attacks. Automated testing does not remove the need for in-depth security reviews at regular intervals.

Perform automated business continuity testing. Develop tests for large scale business continuity, including backup recovery and failover. Set up automated processes to perform these tests regularly.

Release

Automate deployments. Automate deploying the application to test, staging, and production environments. Automation enables faster and more reliable deployments, and ensures consistent deployments to any supported environment. It removes the risk of human error caused by manual deployments. It also makes it easy to schedule releases for convenient times, to minimize any effects of potential downtime.

Use continuous integration. Continuous integration (CI) is the practice of merging all developer code into a central codebase on a regular schedule, and then automatically performing standard build and test processes. CI ensures that an entire team can work on a codebase at the same time without having conflicts. It also ensures that code defects are found as early as possible. Preferably, the CI process should run every time that code is committed or checked in. At the very least, it should run once per day.

Consider adopting a [trunk based development model](#). In this model, developers commit to a single branch (the trunk). There is a requirement that commits never break the build. This model facilitates CI, because all feature work is done in the trunk, and any merge conflicts are resolved when the commit happens.

Consider using continuous delivery. Continuous delivery (CD) is the practice of ensuring that code is always ready to deploy, by automatically building, testing, and deploying code to production-like environments. Adding continuous delivery to create a full CI/CD pipeline will help you detect code defects as soon as possible, and ensures that properly tested updates can be released in a very short time.

Continuous deployment is an additional process that automatically takes any updates that have passed through the CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning, and may not be appropriate for all teams.

Make small incremental changes. Large code changes have a greater potential to introduce bugs. Whenever possible, keep changes small. This limits the potential effects of each change, and makes it easier to understand and debug any issues.

Control exposure to changes. Make sure you're in control of when updates are visible to your end users. Consider using feature toggles to control when features are enabled for end users.

Implement release management strategies to reduce deployment risk. Deploying an application update to production always entails some risk. To minimize this risk, use strategies such as [canary releases](#) or [blue-green deployments](#) to deploy updates to a subset of users. Confirm the update works as expected, and then roll the update out to the rest of the system.

Document all changes. Minor updates and configuration changes can be a source of confusion and versioning conflict. Always keep a clear record of any changes, no matter how small. Log everything that changes, including patches applied, policy changes, and configuration changes. (Don't include sensitive data in these logs. For example, log that a credential was updated, and who made the change, but don't record the updated credentials.) The record of the changes should be visible to the entire team.

Automate Deployments. Automate all deployments, and have systems in place to detect any problems during rollout. Have a mitigation process for preserving the existing code and data in production, before the update

replaces them in all production instances. Have an automated way to roll forward fixes or roll back changes.

Consider making infrastructure immutable. Immutable infrastructure is the principle that you shouldn't modify infrastructure after it's deployed to production. Otherwise, you can get into a state where ad hoc changes have been applied, making it hard to know exactly what changed. Immutable infrastructure works by replacing entire servers as part of any new deployment. This allows the code and the hosting environment to be tested and deployed as a block. Once deployed, infrastructure components aren't modified until the next build and deploy cycle.

Monitoring

Make systems observable. The operations team should always have clear visibility into the health and status of a system or service. Set up external health endpoints to monitor status, and ensure that applications are coded to instrument the operations metrics. Use a common and consistent schema that lets you correlate events across systems. [Azure Diagnostics](#) and [Application Insights](#) are the standard method of tracking the health and status of Azure resources. Microsoft [Operation Management Suite](#) also provides centralized monitoring and management for cloud or hybrid solutions.

Aggregate and correlate logs and metrics. A properly instrumented telemetry system will provide a large amount of raw performance data and event logs. Make sure that telemetry and log data is processed and correlated in a short period of time, so that operations staff always have an up-to-date picture of system health. Organize and display data in ways that give a cohesive view of any issues, so that whenever possible it's clear when events are related to one another.

Consult your corporate retention policy for requirements on how data is processed and how long it should be stored.

Implement automated alerts and notifications. Set up monitoring tools like [Azure Monitor](#) to detect patterns or conditions that indicate potential or current issues, and send alerts to the team members who can address the issues. Tune the alerts to avoid false positives.

Monitor assets and resources for expirations. Some resources and assets, such as certificates, expire after a given amount of time. Make sure to track which assets expire, when they expire, and what services or features depend on them. Use automated processes to monitor these assets. Notify the operations team before an asset expires, and escalate if expiration threatens to disrupt the application.

Management

Automate operations tasks. Manually handling repetitive operations processes is error-prone. Automate these tasks whenever possible to ensure consistent execution and quality. Code that implements the automation should be versioned in source control. As with any other code, automation tools must be tested.

Take an infrastructure-as-code approach to provisioning. Minimize the amount of manual configuration needed to provision resources. Instead, use scripts and [Azure Resource Manager](#) templates. Keep the scripts and templates in source control, like any other code you maintain.

Consider using containers. Containers provide a standard package-based interface for deploying applications. Using containers, an application is deployed using self-contained packages that include any software, dependencies, and files needed to run the application, which greatly simplifies the deployment process.

Containers also create an abstraction layer between the application and the underlying operating system, which provides consistency across environments. This abstraction can also isolate a container from other processes or applications running on a host.

Implement resiliency and self-healing. Resiliency is the ability of an application to recover from failures.

Strategies for resiliency include retrying transient failures, and failing over to a secondary instance or even another region. For more information, see [Designing resilient applications for Azure](#). Instrument your applications so that issues are reported immediately and you can manage outages or other system failures.

Have an operations manual. An operations manual or *runbook* documents the procedures and management information needed for operations staff to maintain a system. Also document any operations scenarios and mitigation plans that might come into play during a failure or other disruption to your service. Create this documentation during the development process, and keep it up to date afterwards. This is a living document, and should be reviewed, tested, and improved regularly.

Shared documentation is critical. Encourage team members to contribute and share knowledge. The entire team should have access to documents. Make it easy for anyone on the team to help keep documents updated.

Document on-call procedures. Make sure on-call duties, schedules, and procedures are documented and shared to all team members. Keep this information up-to-date at all times.

Document escalation procedures for third-party dependencies. If your application depends on external third-party services that you don't directly control, you must have a plan to deal with outages. Create documentation for your planned mitigation processes. Include support contacts and escalation paths.

Use configuration management. Configuration changes should be planned, visible to operations, and recorded. This could take the form of a configuration management database, or a configuration-as-code approach. Configuration should be audited regularly to ensure that what's expected is actually in place.

Get an Azure support plan and understand the process. Azure offers a number of [support plans](#). Determine the right plan for your needs, and make sure the entire team knows how to use it. Team members should understand the details of the plan, how the support process works, and how to open a support ticket with Azure. If you are anticipating a high-scale event, Azure support can assist you with increasing your service limits. For more information, see the [Azure Support FAQs](#).

Follow least-privilege principles when granting access to resources. Carefully manage access to resources. Access should be denied by default, unless a user is explicitly given access to a resource. Only grant a user access to what they need to complete their tasks. Track user permissions and perform regular security audits.

Use role-based access control. Assigning user accounts and access to resources should not be a manual process. Use [Role-Based Access Control](#) (RBAC) grant access based on [Azure Active Directory](#) identities and groups.

Use a bug tracking system to track issues. Without a good way to track issues, it's easy to miss items, duplicate work, or introduce additional problems. Don't rely on informal person-to-person communication to track the status of bugs. Use a bug tracking tool to record details about problems, assign resources to address them, and provide an audit trail of progress and status.

Manage all resources in a change management system. All aspects of your DevOps process should be included in a management and versioning system, so that changes can be easily tracked and audited. This includes code, infrastructure, configuration, documentation, and scripts. Treat all these types of resources as code throughout the test/build/review process.

Use checklists. Create operations checklists to ensure processes are followed. It's common to miss something in a large manual, and following a checklist can force attention to details that might otherwise be overlooked. Maintain the checklists, and continually look for ways to automate tasks and streamline processes.

For more about DevOps, see [What is DevOps?](#) on the Visual Studio site.

Resiliency checklist

3/6/2018 • 17 minutes to read • [Edit Online](#)

Resiliency is the ability of a system to recover from failures and continue to function, and is one of the [pillars of software quality](#). Designing your application for resiliency requires planning for and mitigating a variety of failure modes that could occur. Use this checklist to review your application architecture from a resiliency standpoint. Also review the [Resiliency checklist for specific Azure services](#).

Requirements

Define your customer's availability requirements. Your customer will have availability requirements for the components in your application and this will affect your application's design. Get agreement from your customer for the availability targets of each piece of your application, otherwise your design may not meet the customer's expectations. For more information, see [Defining your resiliency requirements](#).

Application Design

Perform a failure mode analysis (FMA) for your application. FMA is a process for building resiliency into an application early in the design stage. For more information, see [Failure mode analysis](#). The goals of an FMA include:

- Identify what types of failures an application might experience.
- Capture the potential effects and impact of each type of failure on the application.
- Identify recovery strategies.

Deploy multiple instances of services. If your application depends on a single instance of a service, it creates a single point of failure. Provisioning multiple instances improves both resiliency and scalability. For [Azure App Service](#), select an [App Service Plan](#) that offers multiple instances. For Azure Cloud Services, configure each of your roles to use [multiple instances](#). For [Azure Virtual Machines \(VMs\)](#), ensure that your VM architecture includes more than one VM and that each VM is included in an [availability set](#).

Use autoscaling to respond to increases in load. If your application is not configured to scale out automatically as load increases, it's possible that your application's services will fail if they become saturated with user requests. For more details, see the following:

- General: [Scalability checklist](#)
- Azure App Service: [Scale instance count manually or automatically](#)
- Cloud Services: [How to auto scale a cloud service](#)
- Virtual Machines: [Automatic scaling and virtual machine scale sets](#)

Use load balancing to distribute requests. Load balancing distributes your application's requests to healthy service instances by removing unhealthy instances from rotation. If your service uses Azure App Service or Azure Cloud Services, it is already load balanced for you. However, if your application uses Azure VMs, you will need to provision a load balancer. See the [Azure Load Balancer](#) overview for more details.

Configure Azure Application Gateways to use multiple instances. Depending on your application's requirements, an [Azure Application Gateway](#) may be better suited to distributing requests to your application's services. However, single instances of the Application Gateway service are not guaranteed by an SLA so it's possible that your application could fail if the Application Gateway instance fails. Provision more than one medium or larger Application Gateway instance to guarantee availability of the service under the terms of the [SLA](#).

Use Availability Sets for each application tier. Placing your instances in an [availability set](#) provides a higher

SLA.

Consider deploying your application across multiple regions. If your application is deployed to a single region, in the rare event the entire region becomes unavailable, your application will also be unavailable. This may be unacceptable under the terms of your application's SLA. If so, consider deploying your application and its services across multiple regions. A multi-region deployment can use an active-active pattern (distributing requests across multiple active instances) or an active-passive pattern (keeping a "warm" instance in reserve, in case the primary instance fails). We recommend that you deploy multiple instances of your application's services across regional pairs. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

Use Azure Traffic Manager to route your application's traffic to different regions. Azure Traffic Manager performs load balancing at the DNS level and will route traffic to different regions based on the [traffic routing](#) method you specify and the health of your application's endpoints. Without Traffic Manager, you are limited to a single region for your deployment, which limits scale, increases latency for some users, and causes application downtime in the case of a region-wide service disruption.

Configure and test health probes for your load balancers and traffic managers. Ensure that your health logic checks the critical parts of the system and responds appropriately to health probes.

- The health probes for [Azure Traffic Manager](#) and [Azure Load Balancer](#) serve a specific function. For Traffic Manager, the health probe determines whether to fail over to another region. For a load balancer, it determines whether to remove a VM from rotation.
- For a Traffic Manager probe, your health endpoint should check any critical dependencies that are deployed within the same region, and whose failure should trigger a failover to another region.
- For a load balancer, the health endpoint should report the health of the VM. Don't include other tiers or external services. Otherwise, a failure that occurs outside the VM will cause the load balancer to remove the VM from rotation.
- For guidance on implementing health monitoring in your application, see [Health Endpoint Monitoring Pattern](#).

Monitor third-party services. If your application has dependencies on third-party services, identify where and how these third-party services can fail and what effect those failures will have on your application. A third-party service may not include monitoring and diagnostics, so it's important to log your invocations of them and correlate them with your application's health and diagnostic logging using a unique identifier. For more information on proven practices for monitoring and diagnostics, see [Monitoring and Diagnostics guidance](#).

Ensure that any third-party service you consume provides an SLA. If your application depends on a third-party service, but the third party provides no guarantee of availability in the form of an SLA, your application's availability also cannot be guaranteed. Your SLA is only as good as the least available component of your application.

Implement resiliency patterns for remote operations where appropriate. If your application depends on communication between remote services, follow design patterns for dealing with transient failures, such as [Retry Pattern](#), and [Circuit Breaker Pattern](#). For more information, see [Resiliency strategies](#).

Implement asynchronous operations whenever possible. Synchronous operations can monopolize resources and block other operations while the caller waits for the process to complete. Design each part of your application to allow for asynchronous operations whenever possible. For more information on how to implement asynchronous programming in C#, see [Asynchronous Programming with async and await](#).

Data management

Understand the replication methods for your application's data sources. Your application data will be stored in different data sources and have different availability requirements. Evaluate the replication methods for each type of data storage in Azure, including [Azure Storage Replication](#) and [SQL Database Active Geo-Replication](#) to ensure that your application's data requirements are satisfied.

Ensure that no single user account has access to both production and backup data. Your data backups are compromised if one single user account has permission to write to both production and backup sources. A malicious user could purposely delete all your data, while a regular user could accidentally delete it. Design your application to limit the permissions of each user account so that only the users that require write access have write access and it's only to either production or backup, but not both.

Document your data source fail over and fail back process and test it. In the case where your data source fails catastrophically, a human operator will have to follow a set of documented instructions to fail over to a new data source. If the documented steps have errors, an operator will not be able to successfully follow them and fail over the resource. Regularly test the instruction steps to verify that an operator following them is able to successfully fail over and fail back the data source.

Validate your data backups. Regularly verify that your backup data is what you expect by running a script to validate data integrity, schema, and queries. There's no point having a backup if it's not useful to restore your data sources. Log and report any inconsistencies so the backup service can be repaired.

Consider using a storage account type that is geo-redundant. Data stored in an Azure Storage account is always replicated locally. However, there are multiple replication strategies to choose from when a Storage Account is provisioned. Select [Azure Read-Access Geo Redundant Storage \(RA-GRS\)](#) to protect your application data against the rare case when an entire region becomes unavailable.

NOTE

For VMs, do not rely on RA-GRS replication to restore the VM disks (VHD files). Instead, use [Azure Backup](#).

Security

Implement application-level protection against distributed denial of service (DDoS) attacks. Azure services are protected against DDoS attacks at the network layer. However, Azure cannot protect against application-layer attacks, because it is difficult to distinguish between true user requests from malicious user requests. For more information on how to protect against application-layer DDoS attacks, see the "Protecting against DDoS" section of [Microsoft Azure Network Security](#) (PDF download).

Implement the principle of least privilege for access to the application's resources. The default for access to the application's resources should be as restrictive as possible. Grant higher level permissions on an approval basis. Granting overly permissive access to your application's resources by default can result in someone purposely or accidentally deleting resources. Azure provides [role-based access control](#) to manage user privileges, but it's important to verify least privilege permissions for other resources that have their own permissions systems such as SQL Server.

Testing

Perform failover and fallback testing for your application. If you haven't fully tested failover and fallback, you can't be certain that the dependent services in your application come back up in a synchronized manner during disaster recovery. Ensure that your application's dependent services failover and fail back in the correct order.

Perform fault-injection testing for your application. Your application can fail for many different reasons, such as certificate expiration, exhaustion of system resources in a VM, or storage failures. Test your application in an environment as close as possible to production, by simulating or triggering real failures. For example, delete certificates, artificially consume system resources, or delete a storage source. Verify your application's ability to recover from all types of faults, alone and in combination. Check that failures are not propagating or cascading through your system.

Run tests in production using both synthetic and real user data. Test and production are rarely identical, so it's important to use blue/green or a canary deployment and test your application in production. This allows you to

test your application in production under real load and ensure it will function as expected when fully deployed.

Deployment

Document the release process for your application. Without detailed release process documentation, an operator might deploy a bad update or improperly configure settings for your application. Clearly define and document your release process, and ensure that it's available to the entire operations team.

Automate your application's deployment process. If your operations staff is required to manually deploy your application, human error can cause the deployment to fail.

Design your release process to maximize application availability. If your release process requires services to go offline during deployment, your application will be unavailable until they come back online. Use the [blue/green](#) or [canary release](#) deployment technique to deploy your application to production. Both of these techniques involve deploying your release code alongside production code so users of release code can be redirected to production code in the event of a failure.

Log and audit your application's deployments. If you use staged deployment techniques such as blue/green or canary releases there will be more than one version of your application running in production. If a problem should occur, it's critical to determine which version of your application is causing a problem. Implement a robust logging strategy to capture as much version-specific information as possible.

Have a rollback plan for deployment. It's possible that your application deployment could fail and cause your application to become unavailable. Design a rollback process to go back to a last known good version and minimize downtime.

Operations

Implement best practices for monitoring and alerting in your application. Without proper monitoring, diagnostics, and alerting, there is no way to detect failures in your application and alert an operator to fix them. For more information, see [Monitoring and Diagnostics guidance](#).

Measure remote call statistics and make the information available to the application team. If you don't track and report remote call statistics in real time and provide an easy way to review this information, the operations team will not have an instantaneous view into the health of your application. And if you only measure average remote call time, you will not have enough information to reveal issues in the services. Summarize remote call metrics such as latency, throughput, and errors in the 99 and 95 percentiles. Perform statistical analysis on the metrics to uncover errors that occur within each percentile.

Track the number of transient exceptions and retries over an appropriate timeframe. If you don't track and monitor transient exceptions and retry attempts over time, it's possible that an issue or failure could be hidden by your application's retry logic. That is, if your monitoring and logging only shows success or failure of an operation, the fact that the operation had to be retried multiple times due to exceptions will be hidden. A trend of increasing exceptions over time indicates that the service is having an issue and may fail. For more information, see [Retry service specific guidance](#).

Implement an early warning system that alerts an operator. Identify the key performance indicators of your application's health, such as transient exceptions and remote call latency, and set appropriate threshold values for each of them. Send an alert to operations when the threshold value is reached. Set these thresholds at levels that identify issues before they become critical and require a recovery response.

Ensure that more than one person on the team is trained to monitor the application and perform any manual recovery steps. If you only have a single operator on the team who can monitor the application and kick off recovery steps, that person becomes a single point of failure. Train multiple individuals on detection and recovery and make sure there is always at least one active at any time.

Ensure that your application does not run up against Azure subscription limits. Azure subscriptions have limits on certain resource types, such as number of resource groups, number of cores, and number of storage accounts. If your application requirements exceed Azure subscription limits, create another Azure subscription and provision sufficient resources there.

Ensure that your application does not run up against per-service limits. Individual Azure services have consumption limits — for example, limits on storage, throughput, number of connections, requests per second, and other metrics. Your application will fail if it attempts to use resources beyond these limits. This will result in service throttling and possible downtime for affected users. Depending on the specific service and your application requirements, you can often avoid these limits by scaling up (for example, choosing another pricing tier) or scaling out (adding new instances).

Design your application's storage requirements to fall within Azure storage scalability and performance targets. Azure storage is designed to function within predefined scalability and performance targets, so design your application to utilize storage within those targets. If you exceed these targets your application will experience storage throttling. To fix this, provision additional Storage Accounts. If you run up against the Storage Account limit, provision additional Azure Subscriptions and then provision additional Storage Accounts there. For more information, see [Azure Storage Scalability and Performance Targets](#).

Select the right VM size for your application. Measure the actual CPU, memory, disk, and I/O of your VMs in production and verify that the VM size you've selected is sufficient. If not, your application may experience capacity issues as the VMs approach their limits. VM sizes are described in detail in [Sizes for virtual machines in Azure](#).

Determine if your application's workload is stable or fluctuating over time. If your workload fluctuates over time, use Azure VM scale sets to automatically scale the number of VM instances. Otherwise, you will have to manually increase or decrease the number of VMs. For more information, see the [Virtual Machine Scale Sets Overview](#).

Select the right service tier for Azure SQL Database. If your application uses Azure SQL Database, ensure that you have selected the appropriate service tier. If you select a tier that is not able to handle your application's database transaction unit (DTU) requirements, your data use will be throttled. For more information on selecting the correct service plan, see [SQL Database options and performance: Understand what's available in each service tier](#).

Create a process for interacting with Azure support. If the process for contacting [Azure support](#) is not set before the need to contact support arises, downtime will be prolonged as the support process is navigated for the first time. Include the process for contacting support and escalating issues as part of your application's resiliency from the outset.

Ensure that your application doesn't use more than the maximum number of storage accounts per subscription. Azure allows a maximum of 200 storage accounts per subscription. If your application requires more storage accounts than are currently available in your subscription, you will have to create a new subscription and create additional storage accounts there. For more information, see [Azure subscription and service limits, quotas, and constraints](#).

Ensure that your application doesn't exceed the scalability targets for virtual machine disks. An Azure IaaS VM supports attaching a number of data disks depending on several factors, including the VM size and type of storage account. If your application exceeds the scalability targets for virtual machine disks, provision additional storage accounts and create the virtual machine disks there. For more information, see [Azure Storage Scalability and Performance Targets](#)

Telemetry

Log telemetry data while the application is running in the production environment. Capture robust telemetry information while the application is running in the production environment or you will not have sufficient information to diagnose the cause of issues while it's actively serving users. For more information, see [Monitoring](#)

and Diagnostics.

Implement logging using an asynchronous pattern. If logging operations are synchronous, they might block your application code. Ensure that your logging operations are implemented as asynchronous operations.

Correlate log data across service boundaries. In a typical n-tier application, a user request may traverse several service boundaries. For example, a user request typically originates in the web tier and is passed to the business tier and finally persisted in the data tier. In more complex scenarios, a user request may be distributed to many different services and data stores. Ensure that your logging system correlates calls across service boundaries so you can track the request throughout your application.

Azure Resources

Use Azure Resource Manager templates to provision resources. Resource Manager templates make it easier to automate deployments via PowerShell or the Azure CLI, which leads to a more reliable deployment process. For more information, see [Azure Resource Manager overview](#).

Give resources meaningful names. Giving resources meaningful names makes it easier to locate a specific resource and understand its role. For more information, see [Naming conventions for Azure resources](#)

Use role-based access control (RBAC). Use RBAC to control access to the Azure resources that you deploy. RBAC lets you assign authorization roles to members of your DevOps team, to prevent accidental deletion or changes to deployed resources. For more information, see [Get started with access management in the Azure portal](#)

Use resource locks for critical resources, such as VMs. Resource locks prevent an operator from accidentally deleting a resource. For more information, see [Lock resources with Azure Resource Manager](#)

Choose regional pairs. When deploying to two regions, choose regions from the same regional pair. In the event of a broad outage, recovery of one region is prioritized out of every pair. Some services such as Geo-Redundant Storage provide automatic replication to the paired region. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#)

Organize resource groups by function and lifecycle. In general, a resource group should contain resources that share the same lifecycle. This makes it easier to manage deployments, delete test deployments, and assign access rights, reducing the chance that a production deployment is accidentally deleted or modified. Create separate resource groups for production, development, and test environments. In a multi-region deployment, put resources for each region into separate resource groups. This makes it easier to redeploy one region without affecting the other region(s).

Next steps

- [Resiliency checklist for specific Azure services](#)
- [Failure mode analysis](#)

Resiliency checklist for specific Azure services

3/6/2018 • 12 minutes to read • [Edit Online](#)

Resiliency is the ability of a system to recover from failures and continue to function, and is one of the [pillars of software quality](#). Every technology has its own particular failure modes, which you must consider when designing and implementing your application. Use this checklist to review the resiliency considerations for specific Azure services. Also review the [general resiliency checklist](#).

App Service

Use Standard or Premium tier. These tiers support staging slots and automated backups. For more information, see [Azure App Service plans in-depth overview](#)

Avoid scaling up or down. Instead, select a tier and instance size that meet your performance requirements under typical load, and then [scale out](#) the instances to handle changes in traffic volume. Scaling up and down may trigger an application restart.

Store configuration as app settings. Use app settings to hold configuration settings as app settings. Define the settings in your Resource Manager templates, or using PowerShell, so that you can apply them as part of an automated deployment / update process, which is more reliable. For more information, see [Configure web apps in Azure App Service](#).

Create separate App Service plans for production and test. Don't use slots on your production deployment for testing. All apps within the same App Service plan share the same VM instances. If you put production and test deployments in the same plan, it can negatively affect the production deployment. For example, load tests might degrade the live production site. By putting test deployments into a separate plan, you isolate them from the production version.

Separate web apps from web APIs. If your solution has both a web front-end and a web API, consider decomposing them into separate App Service apps. This design makes it easier to decompose the solution by workload. You can run the web app and the API in separate App Service plans, so they can be scaled independently. If you don't need that level of scalability at first, you can deploy the apps into the same plan, and move them into separate plans later, if needed.

Avoid using the App Service backup feature to back up Azure SQL databases. Instead, use [SQL Database automated backups](#). App Service backup exports the database to a SQL .bacpac file, which costs DTUs.

Deploy to a staging slot. Create a deployment slot for staging. Deploy application updates to the staging slot, and verify the deployment before swapping it into production. This reduces the chance of a bad update in production. It also ensures that all instances are warmed up before being swapped into production. Many applications have a significant warmup and cold-start time. For more information, see [Set up staging environments for web apps in Azure App Service](#).

Create a deployment slot to hold the last-known-good (LKG) deployment. When you deploy an update to production, move the previous production deployment into the LKG slot. This makes it easier to roll back a bad deployment. If you discover a problem later, you can quickly revert to the LKG version. For more information, see [Basic web application](#).

Enable diagnostics logging, including application logging and web server logging. Logging is important for monitoring and diagnostics. See [Enable diagnostics logging for web apps in Azure App Service](#)

Log to blob storage. This makes it easier to collect and analyze the data.

Create a separate storage account for logs. Don't use the same storage account for logs and application data. This helps to prevent logging from reducing application performance.

Monitor performance. Use a performance monitoring service such as [New Relic](#) or [Application Insights](#) to monitor application performance and behavior under load. Performance monitoring gives you real-time insight into the application. It enables you to diagnose issues and perform root-cause analysis of failures.

Application Gateway

Provision at least two instances. Deploy Application Gateway with at least two instances. A single instance is a single point of failure. Use two or more instances for redundancy and scalability. In order to qualify for the [SLA](#), you must provision two or more medium or larger instances.

Cosmos DB

Replicate the database across regions. Cosmos DB allows you to associate any number of Azure regions with a Cosmos DB database account. A Cosmos DB database can have one write region and multiple read regions. If there is a failure in the write region, you can read from another replica. The Client SDK handles this automatically. You can also fail over the write region to another region. For more information, see [How to distribute data globally with Azure Cosmos DB](#).

Event Hubs

Use checkpoints. An event consumer should write its current position to persistent storage at some predefined interval. That way, if the consumer experiences a fault (for example, the consumer crashes, or the host fails), then a new instance can resume reading the stream from the last recorded position. For more information, see [Event consumers](#).

Handle duplicate messages. If an event consumer fails, message processing is resumed from the last recorded checkpoint. Any messages that were already processed after the last checkpoint will be processed again. Therefore, your message processing logic must be idempotent, or the application must be able to deduplicate messages.

Handle exceptions. An event consumer typically processes a batch of messages in a loop. You should handle exceptions within this processing loop to avoid losing an entire batch of messages if a single message causes an exception.

Use a dead-letter queue. If processing a message results in a non-transient failure, put the message onto a dead-letter queue, so that you can track the status. Depending on the scenario, you might retry the message later, apply a compensating transaction, or take some other action. Note that Event Hubs does not have any built-in dead-letter queue functionality. You can use Azure Queue Storage or Service Bus to implement a dead-letter queue, or use Azure Functions or some other eventing mechanism.

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

Redis Cache

Configure Geo-replication. Geo-replication provides a mechanism for linking two Premium tier Azure Redis Cache instances. Data written to the primary cache is replicated to a secondary read-only cache. For more information, see [How to configure Geo-replication for Azure Redis Cache](#)

Configure data persistence. Redis persistence allows you to persist data stored in Redis. You can also take snapshots and back up the data, which you can load in case of a hardware failure. For more information, see [How to configure data persistence for a Premium Azure Redis Cache](#)

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

Search

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

Configure indexers for multi-region deployments. If you have a multi-region deployment, consider your options for continuity in indexing.

- If the data source is geo-replicated, you should generally point each indexer of each regional Azure Search service to its local data source replica. However, that approach is not recommended for large datasets stored in Azure SQL Database. The reason is that Azure Search cannot perform incremental indexing from secondary SQL Database replicas, only from primary replicas. Instead, point all indexers to the primary replica. After a failover, point the Azure Search indexers at the new primary replica.
- If the data source is not geo-replicated, point multiple indexers at the same data source, so that Azure Search services in multiple regions continuously and independently index from the data source. For more information, see [Azure Search performance and optimization considerations](#).

Storage

For application data, use read-access geo-redundant storage (RA-GRS). RA-GRS storage replicates the data to a secondary region, and provides read-only access from the secondary region. If there is a storage outage in the primary region, the application can read the data from the secondary region. For more information, see [Azure Storage replication](#).

For VM disks, use Managed Disks. [Managed Disks](#) provide better reliability for VMs in an availability set, because the disks are sufficiently isolated from each other to avoid single points of failure. Also, Managed Disks aren't subject to the IOPS limits of VHDs created in a storage account. For more information, see [Manage the availability of Windows virtual machines in Azure](#).

For Queue storage, create a backup queue in another region. For Queue storage, a read-only replica has limited use, because you can't queue or dequeue items. Instead, create a backup queue in a storage account in another region. If there is a storage outage, the application can use the backup queue, until the primary region becomes available again. That way, the application can still process new requests.

SQL Database

Use Standard or Premium tier. These tiers provide a longer point-in-time restore period (35 days). For more information, see [SQL Database options and performance](#).

Enable SQL Database auditing. Auditing can be used to diagnose malicious attacks or human error. For more information, see [Get started with SQL database auditing](#).

Use Active Geo-Replication Use Active Geo-Replication to create a readable secondary in a different region. If your primary database fails, or simply needs to be taken offline, perform a manual failover to the secondary database. Until you fail over, the secondary database remains read-only. For more information, see [SQL Database Active Geo-Replication](#).

Use sharding. Consider using sharding to partition the database horizontally. Sharding can provide fault isolation. For more information, see [Scaling out with Azure SQL Database](#).

Use point-in-time restore to recover from human error. Point-in-time restore returns your database to an earlier point in time. For more information, see [Recover an Azure SQL database using automated database backups](#).

Use geo-restore to recover from a service outage. Geo-restore restores a database from a geo-redundant backup. For more information, see [Recover an Azure SQL database using automated database backups](#).

SQL Server running in a VM

Replicate the database. Use SQL Server Always On Availability Groups to replicate the database. Provides high availability if one SQL Server instance fails. For more information, see [Run Windows VMs for an N-tier application](#)

Back up the database. If you are already using [Azure Backup](#) to back up your VMs, consider using [Azure Backup for SQL Server workloads using DPM](#). With this approach, there is one backup administrator role for the organization and a unified recovery procedure for VMs and SQL Server. Otherwise, use [SQL Server Managed Backup to Microsoft Azure](#).

Traffic Manager

Perform manual failback. After a Traffic Manager failover, perform manual failback, rather than automatically failing back. Before failing back, verify that all application subsystems are healthy. Otherwise, you can create a situation where the application flips back and forth between data centers. For more information, see [Run VMs in multiple regions for high availability](#).

Create a health probe endpoint. Create a custom endpoint that reports on the overall health of the application. This enables Traffic Manager to fail over if any critical path fails, not just the front end. The endpoint should return an HTTP error code if any critical dependency is unhealthy or unreachable. Don't report errors for non-critical services, however. Otherwise, the health probe might trigger failover when it's not needed, creating false positives. For more information, see [Traffic Manager endpoint monitoring and failover](#).

Virtual Machines

Avoid running a production workload on a single VM. A single VM deployment is not resilient to planned or unplanned maintenance. Instead, put multiple VMs in an availability set or [VM scale set](#), with a load balancer in front.

Specify an availability set when you provision the VM. Currently, there is no way to add a VM to an availability set after the VM is provisioned. When you add a new VM to an existing availability set, make sure to create a NIC for the VM, and add the NIC to the back-end address pool on the load balancer. Otherwise, the load balancer won't route network traffic to that VM.

Put each application tier into a separate Availability Set. In an N-tier application, don't put VMs from different tiers into the same availability set. VMs in an availability set are placed across fault domains (FDs) and update domains (UD). However, to get the redundancy benefit of FDs and UD, every VM in the availability set must be able to handle the same client requests.

Choose the right VM size based on performance requirements. When moving an existing workload to Azure, start with the VM size that's the closest match to your on-premises servers. Then measure the performance of your actual workload with respect to CPU, memory, and disk IOPS, and adjust the size if needed. This helps to ensure the application behaves as expected in a cloud environment. Also, if you need multiple NICs, be aware of the NIC limit for each size.

Use Managed Disks for VHDS. [Managed Disks](#) provide better reliability for VMs in an availability set, because the disks are sufficiently isolated from each other to avoid single points of failure. Also, Managed Disks aren't subject to the IOPS limits of VHDS created in a storage account. For more information, see [Manage the availability of Windows virtual machines in Azure](#).

Install applications on a data disk, not the OS disk. Otherwise, you may reach the disk size limit.

Use Azure Backup to back up VMs. Backups protect against accidental data loss. For more information, see [Protect Azure VMs with a recovery services vault](#).

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

Use the AzureLogCollector extension. (Windows VMs only.) This extension aggregates Azure platform logs and uploads them to Azure storage, without the operator remotely logging into the VM. For more information, see [AzureLogCollector Extension](#).

Virtual Network

To whitelist or block public IP addresses, add an NSG to the subnet. Block access from malicious users, or allow access only from users who have privilege to access the application.

Create a custom health probe. Load Balancer Health Probes can test either HTTP or TCP. If a VM runs an HTTP server, the HTTP probe is a better indicator of health status than a TCP probe. For an HTTP probe, use a custom endpoint that reports the overall health of the application, including all critical dependencies. For more information, see [Azure Load Balancer overview](#).

Don't block the health probe. The Load Balancer Health probe is sent from a known IP address, 168.63.129.16. Don't block traffic to or from this IP in any firewall policies or network security group (NSG) rules. Blocking the health probe would cause the load balancer to remove the VM from rotation.

Enable Load Balancer logging. The logs show how many VMs on the back-end are not receiving network traffic due to failed probe responses. For more information, see [Log analytics for Azure Load Balancer](#).

Scalability checklist

1/10/2018 • 14 minutes to read • [Edit Online](#)

Scalability is the ability of a system to handle increased load, and is one of the [pillars of software quality](#). Use this checklist to review your application architecture from a scalability standpoint.

Application design

Partition the workload. Design parts of the process to be discrete and decomposable. Minimize the size of each part, while following the usual rules for separation of concerns and the single responsibility principle. This allows the component parts to be distributed in a way that maximizes use of each compute unit (such as a role or database server). It also makes it easier to scale the application by adding instances of specific resources. For complex domains, consider adopting a [microservices architecture](#).

Design for scaling. Scaling allows applications to react to variable load by increasing and decreasing the number of instances of roles, queues, and other services they use. However, the application must be designed with this in mind. For example, the application and the services it uses must be stateless, to allow requests to be routed to any instance. This also prevents the addition or removal of specific instances from adversely impacting current users. You should also implement configuration or auto-detection of instances as they are added and removed, so that code in the application can perform the necessary routing. For example, a web application might use a set of queues in a round-robin approach to route requests to background services running in worker roles. The web application must be able to detect changes in the number of queues, to successfully route requests and balance the load on the application.

Scale as a unit. Plan for additional resources to accommodate growth. For each resource, know the upper scaling limits, and use sharding or decomposition to go beyond these limits. Determine the scale units for the system in terms of well-defined sets of resources. This makes applying scale-out operations easier, and less prone to negative impact on the application through limitations imposed by lack of resources in some part of the overall system. For example, adding x number of web and worker roles might require y number of additional queues and z number of storage accounts to handle the additional workload generated by the roles. So a scale unit could consist of x web and worker roles, y queues, and z storage accounts. Design the application so that it's easily scaled by adding one or more scale units.

Avoid client affinity. Where possible, ensure that the application does not require affinity. Requests can thus be routed to any instance, and the number of instances is irrelevant. This also avoids the overhead of storing, retrieving, and maintaining state information for each user.

Take advantage of platform autoscaling features. Where the hosting platform supports an autoscaling capability, such as Azure Autoscale, prefer it to custom or third-party mechanisms unless the built-in mechanism can't fulfill your requirements. Use scheduled scaling rules where possible to ensure resources are available without a start-up delay, but add reactive autoscaling to the rules where appropriate to cope with unexpected changes in demand. You can use the autoscaling operations in the Service Management API to adjust autoscaling, and to add custom counters to rules. For more information, see [Auto-scaling guidance](#).

Offload intensive CPU/IO tasks as background tasks. If a request to a service is expected to take a long time to run or absorb considerable resources, offload the processing for this request to a separate task. Use worker roles or background jobs (depending on the hosting platform) to execute these tasks. This strategy enables the service to continue receiving further requests and remain responsive. For more information, see [Background jobs guidance](#).

Distribute the workload for background tasks. Where there are many background tasks, or the tasks require

considerable time or resources, spread the work across multiple compute units (such as worker roles or background jobs). For one possible solution, see the [Competing Consumers Pattern](#).

Consider moving towards a *shared-nothing* architecture. A shared-nothing architecture uses independent, self-sufficient nodes that have no single point of contention (such as shared services or storage). In theory, such a system can scale almost indefinitely. While a fully shared-nothing approach is generally not practical for most applications, it may provide opportunities to design for better scalability. For example, avoiding the use of server-side session state, client affinity, and data partitioning are good examples of moving towards a shared-nothing architecture.

Data management

Use data partitioning. Divide the data across multiple databases and database servers, or design the application to use data storage services that can provide this partitioning transparently (examples include Azure SQL Database Elastic Database, and Azure Table storage). This approach can help to maximize performance and allow easier scaling. There are different partitioning techniques, such as horizontal, vertical, and functional. You can use a combination of these to achieve maximum benefit from increased query performance, simpler scalability, more flexible management, better availability, and to match the type of store to the data it will hold. Also, consider using different types of data store for different types of data, choosing the types based on how well they are optimized for the specific type of data. This may include using table storage, a document database, or a column-family data store, instead of, or as well as, a relational database. For more information, see [Data partitioning guidance](#).

Design for eventual consistency. Eventual consistency improves scalability by reducing or removing the time needed to synchronize related data partitioned across multiple stores. The cost is that data is not always consistent when it is read, and some write operations may cause conflicts. Eventual consistency is ideal for situations where the same data is read frequently but written infrequently. For more information, see the [Data Consistency Primer](#).

Reduce chatty interactions between components and services. Avoid designing interactions in which an application is required to make multiple calls to a service (each of which returns a small amount of data), rather than a single call that can return all of the data. Where possible, combine several related operations into a single request when the call is to a service or component that has noticeable latency. This makes it easier to monitor performance and optimize complex operations. For example, use stored procedures in databases to encapsulate complex logic, and reduce the number of round trips and resource locking.

Use queues to level the load for high velocity data writes. Surges in demand for a service can overwhelm that service and cause escalating failures. To prevent this, consider implementing the [Queue-Based Load Leveling Pattern](#). Use a queue that acts as a buffer between a task and a service that it invokes. This can smooth intermittent heavy loads that may otherwise cause the service to fail or the task to time out.

Minimize the load on the data store. The data store is commonly a processing bottleneck, a costly resource, and often not easy to scale out. Where possible, remove logic (such as processing XML documents or JSON objects) from the data store, and perform processing within the application. For example, instead of passing XML to the database (other than as an opaque string for storage), serialize or deserialize the XML within the application layer and pass it in a form that is native to the data store. It's typically much easier to scale out the application than the data store, so you should attempt to do as much of the compute-intensive processing as possible within the application.

Minimize the volume of data retrieved. Retrieve only the data you require by specifying columns and using criteria to select rows. Make use of table value parameters and the appropriate isolation level. Use mechanisms like entity tags to avoid retrieving data unnecessarily.

Aggressively use caching. Use caching wherever possible to reduce the load on resources and services that generate or deliver data. Caching is typically suited to data that is relatively static, or that requires considerable processing to obtain. Caching should occur at all levels where appropriate in each layer of the application, including data access and user interface generation. For more information, see the [Caching Guidance](#).

Handle data growth and retention. The amount of data stored by an application grows over time. This growth increases storage costs, and increases latency when accessing the data — which affects application throughput and performance. It may be possible to periodically archive some of the old data that is no longer accessed, or move data that is rarely accessed into long-term storage that is more cost efficient, even if the access latency is higher.

Optimize Data Transfer Objects (DTOs) using an efficient binary format. DTOs are passed between the layers of an application many times. Minimizing the size reduces the load on resources and the network. However, balance the savings with the overhead of converting the data to the required format in each location where it is used. Adopt a format that has the maximum interoperability to enable easy reuse of a component.

Set cache control. Design and configure the application to use output caching or fragment caching where possible, to minimize processing load.

Enable client side caching. Web applications should enable cache settings on the content that can be cached. This is commonly disabled by default. Configure the server to deliver the appropriate cache control headers to enable caching of content on proxy servers and clients.

Use Azure blob storage and the Azure Content Delivery Network to reduce the load on the application. Consider storing static or relatively static public content, such as images, resources, scripts, and style sheets, in blob storage. This approach relieves the application of the load caused by dynamically generating this content for each request. Additionally, consider using the Content Delivery Network to cache this content and deliver it to clients. Using the Content Delivery Network can improve performance at the client because the content is delivered from the geographically closest datacenter that contains a Content Delivery Network cache. For more information, see the [Content Delivery Network Guidance](#).

Optimize and tune SQL queries and indexes. Some T-SQL statements or constructs may have an impact on performance that can be reduced by optimizing the code in a stored procedure. For example, avoid converting **datetime** types to a **varchar** before comparing with a **datetime** literal value. Use date/time comparison functions instead. Lack of appropriate indexes can also slow query execution. If you use an object/relational mapping framework, understand how it works and how it may affect performance of the data access layer. For more information, see [Query Tuning](#).

Consider de-normalizing data. Data normalization helps to avoid duplication and inconsistency. However, maintaining multiple indexes, checking for referential integrity, performing multiple accesses to small chunks of data, and joining tables to reassemble the data imposes an overhead that can affect performance. Consider if some additional storage volume and duplication is acceptable in order to reduce the load on the data store. Also, consider if the application itself (which is typically easier to scale) can be relied upon to take over tasks such as managing referential integrity in order to reduce the load on the data store. For more information, see [Data partitioning guidance](#).

Implementation

Review the performance antipatterns. See [Performance antipatterns for cloud applications](#) for common practices that are likely to cause scalability problems when an application is under pressure.

Use asynchronous calls. Use asynchronous code wherever possible when accessing resources or services that may be limited by I/O or network bandwidth, or that have a noticeable latency, in order to avoid locking the calling thread.

Avoid locking resources, and use an optimistic approach instead. Never lock access to resources such as storage or other services that have noticeable latency, because this is a primary cause of poor performance. Always use optimistic approaches to managing concurrent operations, such as writing to storage. Use features of the storage layer to manage conflicts. In distributed applications, data may be only eventually consistent.

Compress highly compressible data over high latency, low bandwidth networks. In the majority of cases in a web application, the largest volume of data generated by the application and passed over the network is HTTP

responses to client requests. HTTP compression can reduce this considerably, especially for static content. This can reduce cost as well as reducing the load on the network, though compressing dynamic content does apply a fractionally higher load on the server. In other, more generalized environments, data compression can reduce the volume of data transmitted and minimize transfer time and costs, but the compression and decompression processes incur overhead. As such, compression should only be used when there is a demonstrable gain in performance. Other serialization methods, such as JSON or binary encodings, may reduce the payload size while having less impact on performance, whereas XML is likely to increase it.

Minimize the time that connections and resources are in use. Maintain connections and resources only for as long as you need to use them. For example, open connections as late as possible, and allow them to be returned to the connection pool as soon as possible. Acquire resources as late as possible, and dispose of them as soon as possible.

Minimize the number of connections required. Service connections absorb resources. Limit the number that are required and ensure that existing connections are reused whenever possible. For example, after performing authentication, use impersonation where appropriate to run code as a specific identity. This can help to make best use of the connection pool by reusing connections.

NOTE

APIs for some services automatically reuse connections, provided service-specific guidelines are followed. It's important that you understand the conditions that enable connection reuse for each service that your application uses.

Send requests in batches to optimize network use. For example, send and read messages in batches when accessing a queue, and perform multiple reads or writes as a batch when accessing storage or a cache. This can help to maximize efficiency of the services and data stores by reducing the number of calls across the network.

Avoid a requirement to store server-side session state where possible. Server-side session state management typically requires client affinity (that is, routing each request to the same server instance), which affects the ability of the system to scale. Ideally, you should design clients to be stateless with respect to the servers that they use. However, if the application must maintain session state, store sensitive data or large volumes of per-client data in a distributed server-side cache that all instances of the application can access.

Optimize table storage schemas. When using table stores that require the table and column names to be passed and processed with every query, such as Azure table storage, consider using shorter names to reduce this overhead. However, do not sacrifice readability or manageability by using overly compact names.

Create resource dependencies during deployment or at application startup. Avoid repeated calls to methods that test the existence of a resource and then create the resource if it does not exist. Methods such as *CloudTable.CreateIfNotExists* and *CloudQueue.CreateIfNotExists* in the Azure Storage Client Library follow this pattern. These methods can impose considerable overhead if they are invoked before each access to a storage table or storage queue. Instead:

- Create the required resources when the application is deployed, or when it first starts (a single call to *CreateIfNotExists* for each resource in the startup code for a web or worker role is acceptable). However, be sure to handle exceptions that may arise if your code attempts to access a resource that doesn't exist. In these situations, you should log the exception, and possibly alert an operator that a resource is missing.
- Under some circumstances, it may be appropriate to create the missing resource as part of the exception handling code. But you should adopt this approach with caution as the non-existence of the resource might be indicative of a programming error (a misspelled resource name for example), or some other infrastructure-level issue.

Use lightweight frameworks. Carefully choose the APIs and frameworks you use to minimize resource usage, execution time, and overall load on the application. For example, using Web API to handle service requests can reduce the application footprint and increase execution speed, but it may not be suitable for advanced scenarios

where the additional capabilities of Windows Communication Foundation are required.

Consider minimizing the number of service accounts. For example, use a specific account to access resources or services that impose a limit on connections, or perform better where fewer connections are maintained. This approach is common for services such as databases, but it can affect the ability to accurately audit operations due to the impersonation of the original user.

Carry out performance profiling and load testing during development, as part of test routines, and before final release to ensure the application performs and scales as required. This testing should occur on the same type of hardware as the production platform, and with the same types and quantities of data and user load as it will encounter in production. For more information, see [Testing the performance of a cloud service](#).

Designing resilient applications for Azure

4/11/2018 • 26 minutes to read • [Edit Online](#)

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

Building a reliable application in the cloud is different than building a reliable application in an enterprise setting. While historically you may have purchased higher-end hardware to scale up, in a cloud environment you must scale out instead of scaling up. Costs for cloud environments are kept low through the use of commodity hardware. Instead of focusing on preventing failures and optimizing "mean time between failures," in this new environment the focus shifts to "mean time to restore." The goal is to minimize the effect of a failure.

This article provides an overview of how to build resilient applications in Microsoft Azure. It starts with a definition of the term *resiliency* and related concepts. Then it describes a process for achieving resiliency, using a structured approach over the lifetime of an application, from design and implementation to deployment and operations.

What is resiliency?

Resiliency is the ability of a system to recover from failures and continue to function. It's not about *avoiding* failures, but *responding* to failures in a way that avoids downtime or data loss. The goal of resiliency is to return the application to a fully functioning state following a failure.

Two important aspects of resiliency are high availability and disaster recovery.

- **High availability** (HA) is the ability of the application to continue running in a healthy state, without significant downtime. By "healthy state," we mean the application is responsive, and users can connect to the application and interact with it.
- **Disaster recovery** (DR) is the ability to recover from rare but major incidents: non-transient, wide-scale failures, such as service disruption that affects an entire region. Disaster recovery includes data backup and archiving, and may include manual intervention, such as restoring a database from backup.

One way to think about HA versus DR is that DR starts when the impact of a fault exceeds the ability of the HA design to handle it.

When you design resiliency, you must understand your availability requirements. How much downtime is acceptable? This is partly a function of cost. How much will potential downtime cost your business? How much should you invest in making the application highly available? You also have to define what it means for the application to be available. For example, is the application "down" if a customer can submit an order but the system cannot process it within the normal timeframe? Also consider the probability of a particular type of outage occurring, and whether a mitigation strategy is cost-effective.

Another common term is **business continuity** (BC), which is the ability to perform essential business functions during and after adverse conditions, such as a natural disaster or a downed service. BC covers the entire operation of the business, including physical facilities, people, communications, transportation, and IT. This article focuses on cloud applications, but resilience planning must be done in the context of overall BC requirements.

Data backup is a critical part of DR. If the stateless components of an application fail, you can always redeploy them. But if data is lost, the system can't return to a stable state. Data must be backed up, ideally in a different region in case of a region-wide disaster.

Backup is distinct from **data replication**. Data replication involves copying data in near-real-time, so that the system can fail over quickly to a replica. Many databases systems support replication; for example, SQL Server

supports SQL Server Always On Availability Groups. Data replication can reduce how long it takes to recover from an outage, by ensuring that a replica of the data is always standing by. However, data replication won't protect against human error. If data gets corrupted because of human error, the corrupted data just gets copied to the replicas. Therefore, you still need to include long-term backup in your DR strategy.

Process to achieve resiliency

Resiliency is not an add-on. It must be designed into the system and put into operational practice. Here is a general model to follow:

1. **Define** your availability requirements, based on business needs.
2. **Design** the application for resiliency. Start with an architecture that follows proven practices, and then identify the possible failure points in that architecture.
3. **Implement** strategies to detect and recover from failures.
4. **Test** the implementation by simulating faults and triggering forced failovers.
5. **Deploy** the application into production using a reliable, repeatable process.
6. **Monitor** the application to detect failures. By monitoring the system, you can gauge the health of the application and respond to incidents if necessary.
7. **Respond** if there are incidents that require manual interventions.

In the remainder of this article, we discuss each of these steps in more detail.

Defining your resiliency requirements

Resiliency planning starts with business requirements. Here are some approaches for thinking about resiliency in those terms.

Decompose by workload

Many cloud solutions consist of multiple application workloads. The term "workload" in this context means a discrete capability or computing task, which can be logically separated from other tasks, in terms of business logic and data storage requirements. For example, an e-commerce app might include the following workloads:

- Browse and search a product catalog.
- Create and track orders.
- View recommendations.

These workloads might have different requirements for availability, scalability, data consistency, disaster recovery, and so forth. Again, these are business decisions.

Also consider usage patterns. Are there certain critical periods when the system must be available? For example, a tax-filing service can't go down right before the filing deadline, a video streaming service must stay up during a big sports event, and so on. During the critical periods, you might have redundant deployments across several regions, so the application could fail over if one region failed. However, a multi-region deployment is more expensive, so during less critical times, you might run the application in a single region.

RTO and RPO

Two important metrics to consider are the recovery time objective and recovery point objective.

- **Recovery time objective** (RTO) is the maximum acceptable time that an application can be unavailable after an incident. If your RTO is 90 minutes, you must be able to restore the application to a running state within 90 minutes from the start of a disaster. If you have a very low RTO, you might keep a second deployment continually running on standby, to protect against a regional outage.
- **Recovery point objective** (RPO) is the maximum duration of data loss that is acceptable during a disaster. For example, if you store data in a single database, with no replication to other databases, and perform

hourly backups, you could lose up to an hour of data.

RTO and RPO are business requirements. Conducting a risk assessment can help you define the application's RTO and RPO. Another common metric is **mean time to recover** (MTTR), which is the average time that it takes to restore the application after a failure. MTTR is an empirical fact about a system. If MTTR exceeds the RTO, then a failure in the system will cause an unacceptable business disruption, because it won't be possible to restore the system within the defined RTO.

SLAs

In Azure, the [Service Level Agreement](#) (SLA) describes Microsoft's commitments for uptime and connectivity. If the SLA for a particular service is 99.9%, it means you should expect the service to be available 99.9% of the time.

NOTE

The Azure SLA also includes provisions for obtaining a service credit if the SLA is not met, along with specific definitions of "availability" for each service. That aspect of the SLA acts as an enforcement policy.

You should define your own target SLAs for each workload in your solution. An SLA makes it possible to evaluate whether the architecture meets the business requirements. For example, if a workload requires 99.99% uptime, but depends on a service with a 99.9% SLA, that service cannot be a single-point of failure in the system. One remedy is to have a fallback path in case the service fails, or take other measures to recover from a failure in that service.

The following table shows the potential cumulative downtime for various SLA levels.

SLA	DOWNTIME PER WEEK	DOWNTIME PER MONTH	DOWNTIME PER YEAR
99%	1.68 hours	7.2 hours	3.65 days
99.9%	10.1 minutes	43.2 minutes	8.76 hours
99.95%	5 minutes	21.6 minutes	4.38 hours
99.99%	1.01 minutes	4.32 minutes	52.56 minutes
99.999%	6 seconds	25.9 seconds	5.26 minutes

Of course, higher availability is better, everything else being equal. But as you strive for more 9s, the cost and complexity to achieve that level of availability grows. An uptime of 99.99% translates to about 5 minutes of total downtime per month. Is it worth the additional complexity and cost to reach five 9s? The answer depends on the business requirements.

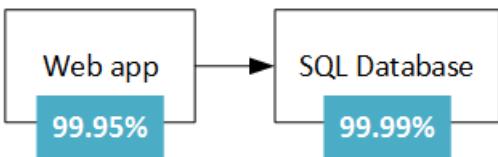
Here are some other considerations when defining an SLA:

- To achieve four 9's (99.99%), you probably can't rely on manual intervention to recover from failures. The application must be self-diagnosing and self-healing.
- Beyond four 9's, it is challenging to detect outages quickly enough to meet the SLA.
- Think about the time window that your SLA is measured against. The smaller the window, the tighter the tolerances. It probably doesn't make sense to define your SLA in terms of hourly or daily uptime.

Composite SLAs

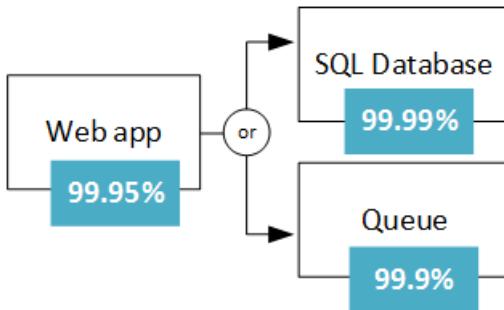
Consider an App Service web app that writes to Azure SQL Database. At the time of this writing, these Azure services have the following SLAs:

- App Service Web Apps = 99.95%
- SQL Database = 99.99%



What is the maximum downtime you would expect for this application? If either service fails, the whole application fails. In general, the probability of each service failing is independent, so the composite SLA for this application is $99.95\% \times 99.99\% = 99.94\%$. That's lower than the individual SLAs, which isn't surprising, because an application that relies on multiple services has more potential failure points.

On the other hand, you can improve the composite SLA by creating independent fallback paths. For example, if SQL Database is unavailable, put transactions into a queue, to be processed later.



With this design, the application is still available even if it can't connect to the database. However, it fails if the database and the queue both fail at the same time. The expected percentage of time for a simultaneous failure is 0.0001×0.001 , so the composite SLA for this combined path is:

- Database OR queue = $1.0 - (0.0001 \times 0.001) = 99.99999\%$

The total composite SLA is:

- Web app AND (database OR queue) = $99.95\% \times 99.99999\% = \sim 99.95\%$

But there are tradeoffs to this approach. The application logic is more complex, you are paying for the queue, and there may be data consistency issues to consider.

SLA for multi-region deployments. Another HA technique is to deploy the application in more than one region, and use Azure Traffic Manager to fail over if the application fails in one region. For a two-region deployment, the composite SLA is calculated as follows.

Let N be the composite SLA for the application deployed in one region. The expected chance that the application will fail in both regions at the same time is $(1 - N) \times (1 - N)$. Therefore,

- Combined SLA for both regions = $1 - (1 - N)(1 - N) = N + (1 - N)N$

Finally, you must factor in the [SLA for Traffic Manager](#). At the time of this writing, the SLA for Traffic Manager SLA is 99.99%.

- Composite SLA = $99.99\% \times (\text{combined SLA for both regions})$

Also, failing over is not instantaneous and can result in some downtime during a failover. See [Traffic Manager endpoint monitoring and failover](#).

The calculated SLA number is a useful baseline, but it doesn't tell the whole story about availability. Often, an application can degrade gracefully when a non-critical path fails. Consider an application that shows a catalog of books. If the application can't retrieve the thumbnail image for the cover, it might show a placeholder image. In that case, failing to get the image does not reduce the application's uptime, although it affects the user experience.

Redundancy and designing for failure

Failures can vary in the scope of their impact. Some hardware failures, such as a failed disk, may affect a single host machine. A failed network switch could affect a whole server rack. Less common are failures that disrupt a whole data center, such as loss of power in a data center. Rarely, an entire region could become unavailable.

One of the main ways to make an application resilient is through redundancy. But you need to plan for this redundancy when you design the application. Also, the level of redundancy that you need depends on your business requirements — not every application needs redundancy across regions to guard against a regional outage. In general, there is a tradeoff between greater redundancy and reliability versus higher cost and complexity.

Azure has a number of features to make an application redundant at every level of failure, from an individual VM to an entire region.

Single VM. Azure provides an uptime SLA for single VMs. Although you can get a higher SLA by running two or more VMs, a single VM may be reliable enough for some workloads. For production workloads, we recommend using two or more VMs for redundancy.

Availability sets. To protect against localized hardware failures, such as a disk or network switch failing, deploy two or more VMs in an availability set. An availability set consists of two or more *fault domains* that share a common power source and network switch. VMs in an availability set are distributed across the fault domains, so if a hardware failure affects one fault domain, network traffic can still be routed the VMs in the other fault domains. For more information about Availability Sets, see [Manage the availability of Windows virtual machines in Azure](#).

Availability zones. An Availability Zone is a physically separate zone within an Azure region. Each Availability Zone has a distinct power source, network, and cooling. Deploying VMs across availability zones helps to protect an application against datacenter-wide failures.

Paired regions. To protect an application against a regional outage, you can deploy the application across multiple regions, using Azure Traffic Manager to distribute internet traffic to the different regions. Each Azure region is paired with another region. Together, these form a [regional pair](#). With the exception of Brazil South, regional pairs are located within the same geography in order to meet data residency requirements for tax and law enforcement jurisdiction purposes.

When you design a multi-region application, take into account that network latency across regions is higher than within a region. For example, if you are replicating a database to enable failover, use synchronous data replication within a region, but asynchronous data replication across regions.

	AVAILABILITY SET	AVAILABILITY ZONE	PAIRED REGION
Scope of failure	Rack	Datacenter	Region
Request routing	Load Balancer	Cross-zone Load Balancer	Traffic Manager
Network latency	Very low	Low	Mid to high
Virtual network	VNet	VNet	Cross-region VNet peering

Designing for resiliency

During the design phase, you should perform a failure mode analysis (FMA). The goal of an FMA is to identify possible points of failure, and define how the application will respond to those failures.

- How will the application detect this type of failure?

- How will the application respond to this type of failure?
- How will you log and monitor this type of failure?

For more information about the FMA process, with specific recommendations for Azure, see [Azure resiliency guidance: Failure mode analysis](#).

Example of identifying failure modes and detection strategy

Failure point: Call to an external web service / API.

FAILURE MODE	DETECTION STRATEGY
Service is unavailable	HTTP 5xx
Throttling	HTTP 429 (Too Many Requests)
Authentication	HTTP 401 (Unauthorized)
Slow response	Request times out

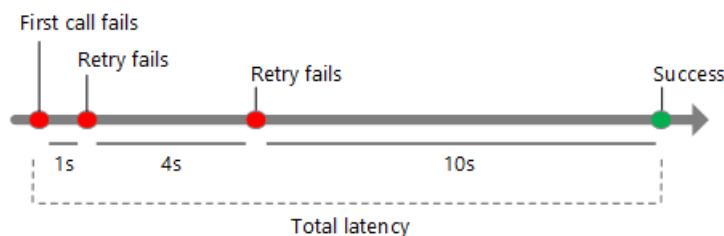
Resiliency strategies

This section provides a survey of some common resiliency strategies. Most of these are not limited to a particular technology. The descriptions in this section summarize the general idea behind each technique, with links to further reading.

Retry transient failures

Transient failures can be caused by momentary loss of network connectivity, a dropped database connection, or a timeout when a service is busy. Often, a transient failure can be resolved simply by retrying the request. For many Azure services, the client SDK implements automatic retries, in a way that is transparent to the caller; see [Retry service specific guidance](#).

Each retry attempt adds to the total latency. Also, too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. To avoid this, increase the delay between each retry attempt, and limit the total number of failed requests.



For more information, see [Retry Pattern](#).

Load balance across instances

For scalability, a cloud application should be able to scale out by adding more instances. This approach also improves resiliency, because unhealthy instances can be removed from rotation.

For example:

- Put two or more VMs behind a load balancer. The load balancer distributes traffic to all the VMs. See [Run load-balanced VMs for scalability and availability](#).
- Scale out an Azure App Service app to multiple instances. App Service automatically balances load across

instances. See [Basic web application](#).

- Use [Azure Traffic Manager](#) to distribute traffic across a set of endpoints.

Replicate data

Replicating data is a general strategy for handling non-transient failures in a data store. Many storage technologies provide built-in replication, including Azure SQL Database, Cosmos DB, and Apache Cassandra.

It's important to consider both the read and write paths. Depending on the storage technology, you might have multiple writable replicas, or a single writable replica and multiple read-only replicas.

To maximize availability, replicas can be placed in multiple regions. However, this increases the latency when replicating the data. Typically, replicating across regions is done asynchronously, which implies an eventual consistency model and potential data loss if a replica fails.

Degrade gracefully

If a service fails and there is no failover path, the application may be able to degrade gracefully while still providing an acceptable user experience. For example:

- Put a work item on a queue, to be handled later.
- Return an estimated value.
- Use locally cached data.
- Show the user an error message. (This option is better than having the application stop responding to requests.)

Throttle high-volume users

Sometimes a small number of users create excessive load. That can have an impact on other users, reducing the overall availability of your application.

When a single client makes an excessive number of requests, the application might throttle the client for a certain period of time. During the throttling period, the application refuses some or all of the requests from that client (depending on the exact throttling strategy). The threshold for throttling might depend on the customer's service tier.

Throttling does not imply the client was necessarily acting maliciously, only that it exceeded its service quota. In some cases, a consumer might consistently exceed their quota or otherwise behave badly. In that case, you might go further and block the user. Typically, this is done by blocking an API key or an IP address range.

For more information, see [Throttling Pattern](#).

Use a circuit breaker

The Circuit Breaker pattern can prevent an application from repeatedly trying an operation that is likely to fail. This is similar to a physical circuit breaker, a switch that interrupts the flow of current when a circuit is overloaded.

The circuit breaker wraps calls to a service. It has three states:

- **Closed.** This is the normal state. The circuit breaker sends requests to the service, and a counter tracks the number of recent failures. If the failure count exceeds a threshold within a given time period, the circuit breaker switches to the Open state.
- **Open.** In this state, the circuit breaker immediately fails all requests, without calling the service. The application should use a mitigation path, such as reading data from a replica or simply returning an error to the user. When the circuit breaker switches to Open, it starts a timer. When the timer expires, the circuit breaker switches to the Half-open state.
- **Half-open.** In this state, the circuit breaker lets a limited number of requests go through to the service. If they succeed, the service is assumed to be recovered, and the circuit breaker switches back to the Closed state. Otherwise, it reverts to the Open state. The Half-Open state prevents a recovering service from suddenly being inundated with requests.

For more information, see [Circuit Breaker Pattern](#).

Use load leveling to smooth out spikes in traffic

Applications may experience sudden spikes in traffic, which can overwhelm services on the backend. If a backend service cannot respond to requests quickly enough, it may cause requests to queue (back up), or cause the service to throttle the application.

To avoid this, you can use a queue as a buffer. When there is a new work item, instead of calling the backend service immediately, the application queues a work item to run asynchronously. The queue acts as a buffer that smooths out peaks in the load.

For more information, see [Queue-Based Load Leveling Pattern](#).

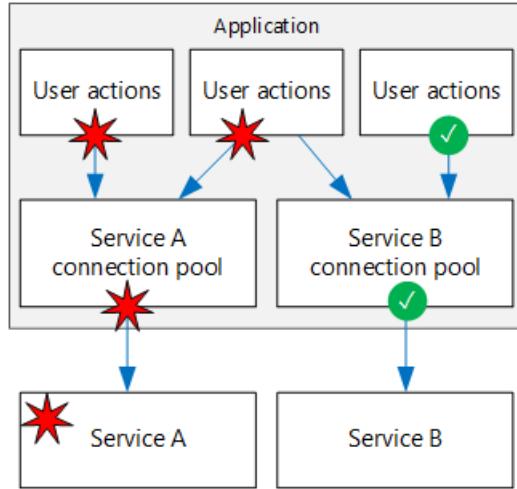
Isolate critical resources

Failures in one subsystem can sometimes cascade, causing failures in other parts of the application. This can happen if a failure causes some resources, such as threads or sockets, not to get freed in a timely manner, leading to resource exhaustion.

To avoid this, you can partition a system into isolated groups, so that a failure in one partition does not bring down the entire system. This technique is sometimes called the Bulkhead pattern.

Examples:

- Partition a database (for example, by tenant) and assign a separate pool of web server instances for each partition.
- Use separate thread pools to isolate calls to different services. This helps to prevent cascading failures if one of the services fails. For an example, see the Netflix [Hystrix library](#).
- Use [containers](#) to limit the resources available to a particular subsystem.



Apply compensating transactions

A compensating transaction is a transaction that undoes the effects of another completed transaction.

In a distributed system, it can be very difficult to achieve strong transactional consistency. Compensating transactions are a way to achieve consistency by using a series of smaller, individual transactions that can be undone at each step.

For example, to book a trip, a customer might reserve a car, a hotel room, and a flight. If any of these steps fails, the entire operation fails. Instead of trying to use a single distributed transaction for the entire operation, you can define a compensating transaction for each step. For example, to undo a car reservation, you cancel the reservation. In order to complete the whole operation, a coordinator executes each step. If any step fails, the coordinator applies compensating transactions to undo any steps that were completed.

For more information, see [Compensating Transaction Pattern](#).

Testing for resiliency

Generally, you can't test resiliency in the same way that you test application functionality (by running unit tests and so on). Instead, you must test how the end-to-end workload performs under failure conditions which only occur intermittently.

Testing is an iterative process. Test the application, measure the outcome, analyze and address any failures that result, and repeat the process.

Fault injection testing. Test the resiliency of the system during failures, either by triggering actual failures or by simulating them. Here are some common failure scenarios to test:

- Shut down VM instances.
- Crash processes.
- Expire certificates.
- Change access keys.
- Shut down the DNS service on domain controllers.
- Limit available system resources, such as RAM or number of threads.
- Unmount disks.
- Redeploy a VM.

Measure the recovery times and verify that your business requirements are met. Test combinations of failure modes as well. Make sure that failures don't cascade, and are handled in an isolated way.

This is another reason why it's important to analyze possible failure points during the design phase. The results of that analysis should be inputs into your test plan.

Load testing. Load test the application using a tool such as [Visual Studio Team Services](#) or [Apache JMeter](#). Load testing is crucial for identifying failures that only happen under load, such as the backend database being overwhelmed or service throttling. Test for peak load, using production data or synthetic data that is as close to production data as possible. The goal is to see how the application behaves under real-world conditions.

Resilient deployment

Once an application is deployed to production, updates are a possible source of errors. In the worst case, a bad update can cause downtime. To avoid this, the deployment process must be predictable and repeatable.

Deployment includes provisioning Azure resources, deploying application code, and applying configuration settings. An update may involve all three, or a subset.

The crucial point is that manual deployments are prone to error. Therefore, it's recommended to have an automated, idempotent process that you can run on demand, and re-run if something fails.

- Use Resource Manager templates to automate provisioning of Azure resources.
- Use [Azure Automation Desired State Configuration](#) (DSC) to configure VMs.
- Use an automated deployment process for application code.

Two concepts related to resilient deployment are *infrastructure as code* and *immutable infrastructure*.

- **Infrastructure as code** is the practice of using code to provision and configure infrastructure. Infrastructure as code may use a declarative approach or an imperative approach (or a combination of both). Resource Manager templates are an example of a declarative approach. PowerShell scripts are an example of an imperative approach.
- **Immutable infrastructure** is the principle that you shouldn't modify infrastructure after it's deployed to production. Otherwise, you can get into a state where ad hoc changes have been applied, so it's hard to know exactly what changed, and hard to reason about the system.

Another question is how to roll out an application update. We recommend techniques such as blue-green deployment or canary releases, which push updates in highly controlled way to minimize possible impacts from a bad deployment.

- **Blue-green deployment** is a technique where an update is deployed into a production environment separate from the live application. After you validate the deployment, switch the traffic routing to the updated version. For example, Azure App Service Web Apps enables this with staging slots.
- **Canary releases** are similar to blue-green deployments. Instead of switching all traffic to the updated version, you roll out the update to a small percentage of users, by routing a portion of the traffic to the new deployment. If there is a problem, back off and revert to the old deployment. Otherwise, route more of the traffic to the new version, until it gets 100% of the traffic.

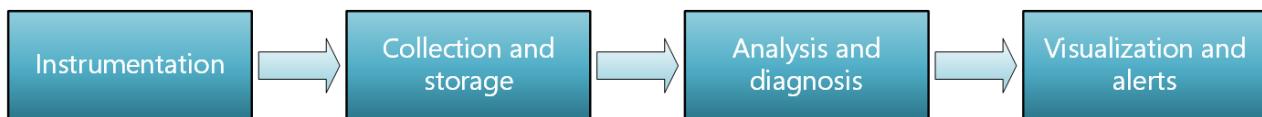
Whatever approach you take, make sure that you can roll back to the last-known-good deployment, in case the new version is not functioning. Also, if errors occur, the application logs must indicate which version caused the error.

Monitoring and diagnostics

Monitoring and diagnostics are crucial for resiliency. If something fails, you need to know that it failed, and you need insights into the cause of the failure.

Monitoring a large-scale distributed system poses a significant challenge. Think about an application that runs on a few dozen VMs — it's not practical to log into each VM, one at a time, and look through log files, trying to troubleshoot a problem. Moreover, the number of VM instances is probably not static. VMs get added and removed as the application scales in and out, and occasionally an instance may fail and need to be reprovisioned. In addition, a typical cloud application might use multiple data stores (Azure storage, SQL Database, Cosmos DB, Redis cache), and a single user action may span multiple subsystems.

You can think of the monitoring and diagnostics process as a pipeline with several distinct stages:



- **Instrumentation.** The raw data for monitoring and diagnostics comes from a variety of sources, including application logs, web server logs, OS performance counters, database logs, and diagnostics built into the Azure platform. Most Azure services have a diagnostics feature that you can use to determine the cause of problems.
- **Collection and storage.** Raw instrumentation data can be held in various locations and with various formats (e.g., application trace logs, IIS logs, performance counters). These disparate sources are collected, consolidated, and put into reliable storage.
- **Analysis and diagnosis.** After the data is consolidated, it can be analyzed to troubleshoot issues and provide an overall view of application health.
- **Visualization and alerts.** In this stage, telemetry data is presented in such a way that an operator can quickly notice problems or trends. Examples include dashboards or email alerts.

Monitoring is not the same as failure detection. For example, your application might detect a transient error and retry, resulting in no downtime. But it should also log the retry operation, so that you can monitor the error rate, in order to get an overall picture of application health.

Application logs are an important source of diagnostics data. Best practices for application logging include:

- Log in production. Otherwise, you lose insight where you need it most.
- Log events at service boundaries. Include a correlation ID that flows across service boundaries. If a transaction flows through multiple services and one of them fails, the correlation ID will help you pinpoint why the transaction failed.
- Use semantic logging, also known as structured logging. Unstructured logs make it hard to automate the

consumption and analysis of the log data, which is needed at cloud scale.

- Use asynchronous logging. Otherwise, the logging system itself can cause the application to fail by causing requests to back up, as they block while waiting to write a logging event.
- Application logging is not the same as auditing. Auditing may be done for compliance or regulatory reasons. As such, audit records must be complete, and it's not acceptable to drop any while processing transactions. If an application requires auditing, this should be kept separate from diagnostics logging.

For more information about monitoring and diagnostics, see [Monitoring and diagnostics guidance](#).

Manual failure responses

Previous sections have focused on automated recovery strategies, which are critical for high availability. However, sometimes manual intervention is needed.

- **Alerts.** Monitor your application for warning signs that may require proactive intervention. For example, if you see that SQL Database or Cosmos DB consistently throttles your application, you might need to increase your database capacity or optimize your queries. In this example, even though the application might handle the throttling errors transparently, your telemetry should still raise an alert so that you can follow up.
- **Manual failover.** Some systems cannot fail over automatically and require a manual failover.
- **Operational readiness testing.** If your application fails over to a secondary region, you should perform an operational readiness test before you fail back to the primary region. The test should verify that the primary region is healthy and ready to receive traffic again.
- **Data consistency check.** If a failure happens in a data store, there may be data inconsistencies when the store becomes available again, especially if the data was replicated.
- **Restoring from backup.** For example, if SQL Database experiences a regional outage, you can geo-restore the database from the latest backup.

Document and test your disaster recovery plan. Evaluate the business impact of application failures. Automate the process as much as possible, and document any manual steps, such as manual failover or data restoration from backups. Regularly test your disaster recovery process to validate and improve the plan.

Summary

This article discussed resiliency from a holistic perspective, emphasizing some of the unique challenges of the cloud. These include the distributed nature of cloud computing, the use of commodity hardware, and the presence of transient network faults.

Here are the major points to take away from this article:

- Resiliency leads to higher availability, and lower mean time to recover from failures.
- Achieving resiliency in the cloud requires a different set of techniques from traditional on-premises solutions.
- Resiliency does not happen by accident. It must be designed and built in from the start.
- Resiliency touches every part of the application lifecycle, from planning and coding to operations.
- Test and monitor!

Resiliency checklist

3/6/2018 • 17 minutes to read • [Edit Online](#)

Resiliency is the ability of a system to recover from failures and continue to function, and is one of the [pillars of software quality](#). Designing your application for resiliency requires planning for and mitigating a variety of failure modes that could occur. Use this checklist to review your application architecture from a resiliency standpoint. Also review the [Resiliency checklist for specific Azure services](#).

Requirements

Define your customer's availability requirements. Your customer will have availability requirements for the components in your application and this will affect your application's design. Get agreement from your customer for the availability targets of each piece of your application, otherwise your design may not meet the customer's expectations. For more information, see [Defining your resiliency requirements](#).

Application Design

Perform a failure mode analysis (FMA) for your application. FMA is a process for building resiliency into an application early in the design stage. For more information, see [Failure mode analysis](#). The goals of an FMA include:

- Identify what types of failures an application might experience.
- Capture the potential effects and impact of each type of failure on the application.
- Identify recovery strategies.

Deploy multiple instances of services. If your application depends on a single instance of a service, it creates a single point of failure. Provisioning multiple instances improves both resiliency and scalability. For [Azure App Service](#), select an [App Service Plan](#) that offers multiple instances. For Azure Cloud Services, configure each of your roles to use [multiple instances](#). For [Azure Virtual Machines \(VMs\)](#), ensure that your VM architecture includes more than one VM and that each VM is included in an [availability set](#).

Use autoscaling to respond to increases in load. If your application is not configured to scale out automatically as load increases, it's possible that your application's services will fail if they become saturated with user requests. For more details, see the following:

- General: [Scalability checklist](#)
- Azure App Service: [Scale instance count manually or automatically](#)
- Cloud Services: [How to auto scale a cloud service](#)
- Virtual Machines: [Automatic scaling and virtual machine scale sets](#)

Use load balancing to distribute requests. Load balancing distributes your application's requests to healthy service instances by removing unhealthy instances from rotation. If your service uses Azure App Service or Azure Cloud Services, it is already load balanced for you. However, if your application uses Azure VMs, you will need to provision a load balancer. See the [Azure Load Balancer](#) overview for more details.

Configure Azure Application Gateways to use multiple instances. Depending on your application's requirements, an [Azure Application Gateway](#) may be better suited to distributing requests to your application's services. However, single instances of the Application Gateway service are not guaranteed by an SLA so it's possible that your application could fail if the Application Gateway instance fails. Provision more than one medium or larger Application Gateway instance to guarantee availability of the service under the terms of the [SLA](#).

Use Availability Sets for each application tier. Placing your instances in an [availability set](#) provides a higher SLA.

Consider deploying your application across multiple regions. If your application is deployed to a single region, in the rare event the entire region becomes unavailable, your application will also be unavailable. This may be unacceptable under the terms of your application's SLA. If so, consider deploying your application and its services across multiple regions. A multi-region deployment can use an active-active pattern (distributing requests across multiple active instances) or an active-passive pattern (keeping a "warm" instance in reserve, in case the primary instance fails). We recommend that you deploy multiple instances of your application's services across regional pairs. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

Use Azure Traffic Manager to route your application's traffic to different regions. Azure Traffic Manager performs load balancing at the DNS level and will route traffic to different regions based on the [traffic routing](#) method you specify and the health of your application's endpoints. Without Traffic Manager, you are limited to a single region for your deployment, which limits scale, increases latency for some users, and causes application downtime in the case of a region-wide service disruption.

Configure and test health probes for your load balancers and traffic managers. Ensure that your health logic checks the critical parts of the system and responds appropriately to health probes.

- The health probes for [Azure Traffic Manager](#) and [Azure Load Balancer](#) serve a specific function. For Traffic Manager, the health probe determines whether to fail over to another region. For a load balancer, it determines whether to remove a VM from rotation.
- For a Traffic Manager probe, your health endpoint should check any critical dependencies that are deployed within the same region, and whose failure should trigger a failover to another region.
- For a load balancer, the health endpoint should report the health of the VM. Don't include other tiers or external services. Otherwise, a failure that occurs outside the VM will cause the load balancer to remove the VM from rotation.
- For guidance on implementing health monitoring in your application, see [Health Endpoint Monitoring Pattern](#).

Monitor third-party services. If your application has dependencies on third-party services, identify where and how these third-party services can fail and what effect those failures will have on your application. A third-party service may not include monitoring and diagnostics, so it's important to log your invocations of them and correlate them with your application's health and diagnostic logging using a unique identifier. For more information on proven practices for monitoring and diagnostics, see [Monitoring and Diagnostics guidance](#).

Ensure that any third-party service you consume provides an SLA. If your application depends on a third-party service, but the third party provides no guarantee of availability in the form of an SLA, your application's availability also cannot be guaranteed. Your SLA is only as good as the least available component of your application.

Implement resiliency patterns for remote operations where appropriate. If your application depends on communication between remote services, follow design patterns for dealing with transient failures, such as [Retry Pattern](#), and [Circuit Breaker Pattern](#). For more information, see [Resiliency strategies](#).

Implement asynchronous operations whenever possible. Synchronous operations can monopolize resources and block other operations while the caller waits for the process to complete. Design each part of your application to allow for asynchronous operations whenever possible. For more information on how to implement asynchronous programming in C#, see [Asynchronous Programming with async and await](#).

Data management

Understand the replication methods for your application's data sources. Your application data will be stored in different data sources and have different availability requirements. Evaluate the replication methods for each type of data storage in Azure, including [Azure Storage Replication](#) and [SQL Database Active Geo-Replication](#) to

ensure that your application's data requirements are satisfied.

Ensure that no single user account has access to both production and backup data. Your data backups are compromised if one single user account has permission to write to both production and backup sources. A malicious user could purposely delete all your data, while a regular user could accidentally delete it. Design your application to limit the permissions of each user account so that only the users that require write access have write access and it's only to either production or backup, but not both.

Document your data source fail over and fail back process and test it. In the case where your data source fails catastrophically, a human operator will have to follow a set of documented instructions to fail over to a new data source. If the documented steps have errors, an operator will not be able to successfully follow them and fail over the resource. Regularly test the instruction steps to verify that an operator following them is able to successfully fail over and fail back the data source.

Validate your data backups. Regularly verify that your backup data is what you expect by running a script to validate data integrity, schema, and queries. There's no point having a backup if it's not useful to restore your data sources. Log and report any inconsistencies so the backup service can be repaired.

Consider using a storage account type that is geo-redundant. Data stored in an Azure Storage account is always replicated locally. However, there are multiple replication strategies to choose from when a Storage Account is provisioned. Select [Azure Read-Access Geo Redundant Storage \(RA-GRS\)](#) to protect your application data against the rare case when an entire region becomes unavailable.

NOTE

For VMs, do not rely on RA-GRS replication to restore the VM disks (VHD files). Instead, use [Azure Backup](#).

Security

Implement application-level protection against distributed denial of service (DDoS) attacks. Azure services are protected against DDoS attacks at the network layer. However, Azure cannot protect against application-layer attacks, because it is difficult to distinguish between true user requests from malicious user requests. For more information on how to protect against application-layer DDoS attacks, see the "Protecting against DDoS" section of [Microsoft Azure Network Security](#) (PDF download).

Implement the principle of least privilege for access to the application's resources. The default for access to the application's resources should be as restrictive as possible. Grant higher level permissions on an approval basis. Granting overly permissive access to your application's resources by default can result in someone purposely or accidentally deleting resources. Azure provides [role-based access control](#) to manage user privileges, but it's important to verify least privilege permissions for other resources that have their own permissions systems such as SQL Server.

Testing

Perform failover and failback testing for your application. If you haven't fully tested failover and failback, you can't be certain that the dependent services in your application come back up in a synchronized manner during disaster recovery. Ensure that your application's dependent services failover and fail back in the correct order.

Perform fault-injection testing for your application. Your application can fail for many different reasons, such as certificate expiration, exhaustion of system resources in a VM, or storage failures. Test your application in an environment as close as possible to production, by simulating or triggering real failures. For example, delete certificates, artificially consume system resources, or delete a storage source. Verify your application's ability to recover from all types of faults, alone and in combination. Check that failures are not propagating or cascading through your system.

Run tests in production using both synthetic and real user data. Test and production are rarely identical, so it's important to use blue/green or a canary deployment and test your application in production. This allows you to test your application in production under real load and ensure it will function as expected when fully deployed.

Deployment

Document the release process for your application. Without detailed release process documentation, an operator might deploy a bad update or improperly configure settings for your application. Clearly define and document your release process, and ensure that it's available to the entire operations team.

Automate your application's deployment process. If your operations staff is required to manually deploy your application, human error can cause the deployment to fail.

Design your release process to maximize application availability. If your release process requires services to go offline during deployment, your application will be unavailable until they come back online. Use the [blue/green](#) or [canary release](#) deployment technique to deploy your application to production. Both of these techniques involve deploying your release code alongside production code so users of release code can be redirected to production code in the event of a failure.

Log and audit your application's deployments. If you use staged deployment techniques such as blue/green or canary releases there will be more than one version of your application running in production. If a problem should occur, it's critical to determine which version of your application is causing a problem. Implement a robust logging strategy to capture as much version-specific information as possible.

Have a rollback plan for deployment. It's possible that your application deployment could fail and cause your application to become unavailable. Design a rollback process to go back to a last known good version and minimize downtime.

Operations

Implement best practices for monitoring and alerting in your application. Without proper monitoring, diagnostics, and alerting, there is no way to detect failures in your application and alert an operator to fix them. For more information, see [Monitoring and Diagnostics guidance](#).

Measure remote call statistics and make the information available to the application team. If you don't track and report remote call statistics in real time and provide an easy way to review this information, the operations team will not have an instantaneous view into the health of your application. And if you only measure average remote call time, you will not have enough information to reveal issues in the services. Summarize remote call metrics such as latency, throughput, and errors in the 99 and 95 percentiles. Perform statistical analysis on the metrics to uncover errors that occur within each percentile.

Track the number of transient exceptions and retries over an appropriate timeframe. If you don't track and monitor transient exceptions and retry attempts over time, it's possible that an issue or failure could be hidden by your application's retry logic. That is, if your monitoring and logging only shows success or failure of an operation, the fact that the operation had to be retried multiple times due to exceptions will be hidden. A trend of increasing exceptions over time indicates that the service is having an issue and may fail. For more information, see [Retry service specific guidance](#).

Implement an early warning system that alerts an operator. Identify the key performance indicators of your application's health, such as transient exceptions and remote call latency, and set appropriate threshold values for each of them. Send an alert to operations when the threshold value is reached. Set these thresholds at levels that identify issues before they become critical and require a recovery response.

Ensure that more than one person on the team is trained to monitor the application and perform any manual recovery steps. If you only have a single operator on the team who can monitor the application and kick

off recovery steps, that person becomes a single point of failure. Train multiple individuals on detection and recovery and make sure there is always at least one active at any time.

Ensure that your application does not run up against Azure subscription limits. Azure subscriptions have limits on certain resource types, such as number of resource groups, number of cores, and number of storage accounts. If your application requirements exceed Azure subscription limits, create another Azure subscription and provision sufficient resources there.

Ensure that your application does not run up against per-service limits. Individual Azure services have consumption limits — for example, limits on storage, throughput, number of connections, requests per second, and other metrics. Your application will fail if it attempts to use resources beyond these limits. This will result in service throttling and possible downtime for affected users. Depending on the specific service and your application requirements, you can often avoid these limits by scaling up (for example, choosing another pricing tier) or scaling out (adding new instances).

Design your application's storage requirements to fall within Azure storage scalability and performance targets. Azure storage is designed to function within predefined scalability and performance targets, so design your application to utilize storage within those targets. If you exceed these targets your application will experience storage throttling. To fix this, provision additional Storage Accounts. If you run up against the Storage Account limit, provision additional Azure Subscriptions and then provision additional Storage Accounts there. For more information, see [Azure Storage Scalability and Performance Targets](#).

Select the right VM size for your application. Measure the actual CPU, memory, disk, and I/O of your VMs in production and verify that the VM size you've selected is sufficient. If not, your application may experience capacity issues as the VMs approach their limits. VM sizes are described in detail in [Sizes for virtual machines in Azure](#).

Determine if your application's workload is stable or fluctuating over time. If your workload fluctuates over time, use Azure VM scale sets to automatically scale the number of VM instances. Otherwise, you will have to manually increase or decrease the number of VMs. For more information, see the [Virtual Machine Scale Sets Overview](#).

Select the right service tier for Azure SQL Database. If your application uses Azure SQL Database, ensure that you have selected the appropriate service tier. If you select a tier that is not able to handle your application's database transaction unit (DTU) requirements, your data use will be throttled. For more information on selecting the correct service plan, see [SQL Database options and performance: Understand what's available in each service tier](#).

Create a process for interacting with Azure support. If the process for contacting [Azure support](#) is not set before the need to contact support arises, downtime will be prolonged as the support process is navigated for the first time. Include the process for contacting support and escalating issues as part of your application's resiliency from the outset.

Ensure that your application doesn't use more than the maximum number of storage accounts per subscription. Azure allows a maximum of 200 storage accounts per subscription. If your application requires more storage accounts than are currently available in your subscription, you will have to create a new subscription and create additional storage accounts there. For more information, see [Azure subscription and service limits, quotas, and constraints](#).

Ensure that your application doesn't exceed the scalability targets for virtual machine disks. An Azure IaaS VM supports attaching a number of data disks depending on several factors, including the VM size and type of storage account. If your application exceeds the scalability targets for virtual machine disks, provision additional storage accounts and create the virtual machine disks there. For more information, see [Azure Storage Scalability and Performance Targets](#)

Telemetry

Log telemetry data while the application is running in the production environment. Capture robust telemetry information while the application is running in the production environment or you will not have sufficient information to diagnose the cause of issues while it's actively serving users. For more information, see [Monitoring and Diagnostics](#).

Implement logging using an asynchronous pattern. If logging operations are synchronous, they might block your application code. Ensure that your logging operations are implemented as asynchronous operations.

Correlate log data across service boundaries. In a typical n-tier application, a user request may traverse several service boundaries. For example, a user request typically originates in the web tier and is passed to the business tier and finally persisted in the data tier. In more complex scenarios, a user request may be distributed to many different services and data stores. Ensure that your logging system correlates calls across service boundaries so you can track the request throughout your application.

Azure Resources

Use Azure Resource Manager templates to provision resources. Resource Manager templates make it easier to automate deployments via PowerShell or the Azure CLI, which leads to a more reliable deployment process. For more information, see [Azure Resource Manager overview](#).

Give resources meaningful names. Giving resources meaningful names makes it easier to locate a specific resource and understand its role. For more information, see [Naming conventions for Azure resources](#)

Use role-based access control (RBAC). Use RBAC to control access to the Azure resources that you deploy. RBAC lets you assign authorization roles to members of your DevOps team, to prevent accidental deletion or changes to deployed resources. For more information, see [Get started with access management in the Azure portal](#)

Use resource locks for critical resources, such as VMs. Resource locks prevent an operator from accidentally deleting a resource. For more information, see [Lock resources with Azure Resource Manager](#)

Choose regional pairs. When deploying to two regions, choose regions from the same regional pair. In the event of a broad outage, recovery of one region is prioritized out of every pair. Some services such as Geo-Redundant Storage provide automatic replication to the paired region. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#)

Organize resource groups by function and lifecycle. In general, a resource group should contain resources that share the same lifecycle. This makes it easier to manage deployments, delete test deployments, and assign access rights, reducing the chance that a production deployment is accidentally deleted or modified. Create separate resource groups for production, development, and test environments. In a multi-region deployment, put resources for each region into separate resource groups. This makes it easier to redeploy one region without affecting the other region(s).

Next steps

- [Resiliency checklist for specific Azure services](#)
- [Failure mode analysis](#)

Failure mode analysis

5/7/2018 • 17 minutes to read • [Edit Online](#)

Failure mode analysis (FMA) is a process for building resiliency into a system, by identifying possible failure points in the system. The FMA should be part of the architecture and design phases, so that you can build failure recovery into the system from the beginning.

Here is the general process to conduct an FMA:

1. Identify all of the components in the system. Include external dependencies, such as identity providers, third-party services, and so on.
2. For each component, identify potential failures that could occur. A single component may have more than one failure mode. For example, you should consider read failures and write failures separately, because the impact and possible mitigations will be different.
3. Rate each failure mode according to its overall risk. Consider these factors:
 - What is the likelihood of the failure. Is it relatively common? Extremely rare? You don't need exact numbers; the purpose is to help rank the priority.
 - What is the impact on the application, in terms of availability, data loss, monetary cost, and business disruption?
4. For each failure mode, determine how the application will respond and recover. Consider tradeoffs in cost and application complexity.

As a starting point for your FMA process, this article contains a catalog of potential failure modes and their mitigations. The catalog is organized by technology or Azure service, plus a general category for application-level design. The catalog is not exhaustive, but covers many of the core Azure services.

App Service

App Service app shuts down.

Detection. Possible causes:

- Expected shutdown
 - An operator shuts down the application; for example, using the Azure portal.
 - The app was unloaded because it was idle. (Only if the `Always On` setting is disabled.)
- Unexpected shutdown
 - The app crashes.
 - An App Service VM instance becomes unavailable.

Application_End logging will catch the app domain shutdown (soft process crash) and is the only way to catch the application domain shutdowns.

Recovery

- If the shutdown was expected, use the application's shutdown event to shut down gracefully. For example, in ASP.NET, use the `Application_End` method.
- If the application was unloaded while idle, it is automatically restarted on the next request. However, you will incur the "cold start" cost.

- To prevent the application from being unloaded while idle, enable the `Always On` setting in the web app. See [Configure web apps in Azure App Service](#).
- To prevent an operator from shutting down the app, set a resource lock with `ReadOnly` level. See [Lock resources with Azure Resource Manager](#).
- If the app crashes or an App Service VM becomes unavailable, App Service automatically restarts the app.

Diagnostics. Application logs and web server logs. See [Enable diagnostics logging for web apps in Azure App Service](#).

A particular user repeatedly makes bad requests or overloads the system.

Detection. Authenticate users and include user ID in application logs.

Recovery

- Use [Azure API Management](#) to throttle requests from the user. See [Advanced request throttling with Azure API Management](#)
- Block the user.

Diagnostics. Log all authentication requests.

A bad update was deployed.

Detection. Monitor the application health through the Azure Portal (see [Monitor Azure web app performance](#)) or implement the [health endpoint monitoring pattern](#).

Recovery. Use multiple [deployment slots](#) and roll back to the last-known-good deployment. For more information, see [Basic web application](#).

Azure Active Directory

OpenID Connect (OIDC) authentication fails.

Detection. Possible failure modes include:

1. Azure AD is not available, or cannot be reached due to a network problem. Redirection to the authentication endpoint fails, and the OIDC middleware throws an exception.
2. Azure AD tenant does not exist. Redirection to the authentication endpoint returns an HTTP error code, and the OIDC middleware throws an exception.
3. User cannot authenticate. No detection strategy is necessary; Azure AD handles login failures.

Recovery

1. Catch unhandled exceptions from the middleware.
2. Handle `AuthenticationFailed` events.
3. Redirect the user to an error page.
4. User retries.

Azure Search

Writing data to Azure Search fails.

Detection. Catch `Microsoft.Rest.Azure.CloudException` errors.

Recovery

The [Search .NET SDK](#) automatically retries after transient failures. Any exceptions thrown by the client SDK should be treated as non-transient errors.

The default retry policy uses exponential back-off. To use a different retry policy, call `SetRetryPolicy` on the

`SearchIndexClient` or `SearchServiceClient` class. For more information, see [Automatic Retries](#).

Diagnostics. Use [Search Traffic Analytics](#).

Reading data from Azure Search fails.

Detection. Catch `Microsoft.Rest.Azure.CloudException` errors.

Recovery

The [Search .NET SDK](#) automatically retries after transient failures. Any exceptions thrown by the client SDK should be treated as non-transient errors.

The default retry policy uses exponential back-off. To use a different retry policy, call `SetRetryPolicy` on the `SearchIndexClient` or `SearchServiceClient` class. For more information, see [Automatic Retries](#).

Diagnostics. Use [Search Traffic Analytics](#).

Cassandra

Reading or writing to a node fails.

Detection. Catch the exception. For .NET clients, this will typically be `System.Web.HttpException`. Other client may have other exception types. For more information, see [Cassandra error handling done right](#).

Recovery

- Each [Cassandra client](#) has its own retry policies and capabilities. For more information, see [Cassandra error handling done right](#).
- Use a rack-aware deployment, with data nodes distributed across the fault domains.
- Deploy to multiple regions with local quorum consistency. If a non-transient failure occurs, fail over to another region.

Diagnostics. Application logs

Cloud Service

Web or worker roles are unexpectedly being shut down.

Detection. The `RoleEnvironmentStopping` event is fired.

Recovery. Override the `RoleEntryPoint.OnStop` method to gracefully clean up. For more information, see [The Right Way to Handle Azure OnStop Events](#) (blog).

Cosmos DB

Reading data fails.

Detection. Catch `System.Net.Http.HttpRequestException` Or `Microsoft.Azure.Documents.DocumentClientException`.

Recovery

- The SDK automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. Exceptions that the client raises are either beyond the retry policy or are not transient errors.
- If Cosmos DB throttles the client, it returns an HTTP 429 error. Check the status code in the `DocumentClientException`. If you are getting error 429 consistently, consider increasing the throughput value of the collection.
 - If you are using the MongoDB API, the service returns error code 16500 when throttling.
- Replicate the Cosmos DB database across two or more regions. All replicas are readable. Using the client SDKs,

specify the `PreferredLocations` parameter. This is an ordered list of Azure regions. All reads will be sent to the first available region in the list. If the request fails, the client will try the other regions in the list, in order. For more information, see [How to setup Azure Cosmos DB global distribution using the SQL API](#).

Diagnostics. Log all errors on the client side.

Writing data fails.

Detection. Catch `System.Net.Http.HttpRequestException` or `Microsoft.Azure.Documents.DocumentClientException`.

Recovery

- The SDK automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. Exceptions that the client raises are either beyond the retry policy or are not transient errors.
- If Cosmos DB throttles the client, it returns an HTTP 429 error. Check the status code in the `DocumentClientException`. If you are getting error 429 consistently, consider increasing the throughput value of the collection.
- Replicate the Cosmos DB database across two or more regions. If the primary region fails, another region will be promoted to write. You can also trigger a failover manually. The SDK does automatic discovery and routing, so application code continues to work after a failover. During the failover period (typically minutes), write operations will have higher latency, as the SDK finds the new write region. For more information, see [How to setup Azure Cosmos DB global distribution using the SQL API](#).
- As a fallback, persist the document to a backup queue, and process the queue later.

Diagnostics. Log all errors on the client side.

Elasticsearch

Reading data from Elasticsearch fails.

Detection. Catch the appropriate exception for the particular [Elasticsearch client](#) being used.

Recovery

- Use a retry mechanism. Each client has its own retry policies.
- Deploy multiple Elasticsearch nodes and use replication for high availability.

For more information, see [Running Elasticsearch on Azure](#).

Diagnostics. You can use monitoring tools for Elasticsearch, or log all errors on the client side with the payload. See the 'Monitoring' section in [Running Elasticsearch on Azure](#).

Writing data to Elasticsearch fails.

Detection. Catch the appropriate exception for the particular [Elasticsearch client](#) being used.

Recovery

- Use a retry mechanism. Each client has its own retry policies.
- If the application can tolerate a reduced consistency level, consider writing with `write_consistency` setting of `quorum`.

For more information, see [Running Elasticsearch on Azure](#).

Diagnostics. You can use monitoring tools for Elasticsearch, or log all errors on the client side with the payload. See the 'Monitoring' section in [Running Elasticsearch on Azure](#).

Queue storage

Writing a message to Azure Queue storage fails consistently.

Detection. After N retry attempts, the write operation still fails.

Recovery

- Store the data in a local cache, and forward the writes to storage later, when the service becomes available.
- Create a secondary queue, and write to that queue if the primary queue is unavailable.

Diagnostics. Use [storage metrics](#).

The application cannot process a particular message from the queue.

Detection. Application specific. For example, the message contains invalid data, or the business logic fails for some reason.

Recovery

Move the message to a separate queue. Run a separate process to examine the messages in that queue.

Consider using Azure Service Bus Messaging queues, which provides a [dead-letter queue](#) functionality for this purpose.

NOTE

If you are using Storage queues with WebJobs, the WebJobs SDK provides built-in poison message handling. See [How to use Azure queue storage with the WebJobs SDK](#).

Diagnostics. Use application logging.

Redis Cache

Reading from the cache fails.

Detection. Catch `StackExchange.Redis.RedisConnectionException`.

Recovery

1. Retry on transient failures. Azure Redis cache supports built-in retry through See [Redis Cache retry guidelines](#).
2. Treat non-transient failures as a cache miss, and fall back to the original data source.

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

Writing to the cache fails.

Detection. Catch `StackExchange.Redis.RedisConnectionException`.

Recovery

1. Retry on transient failures. Azure Redis cache supports built-in retry through See [Redis Cache retry guidelines](#).
2. If the error is non-transient, ignore it and let other transactions write to the cache later.

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

SQL Database

Cannot connect to the database in the primary region.

Detection. Connection fails.

Recovery

Prerequisite: The database must be configured for active geo-replication. See [SQL Database Active Geo-Replication](#).

- For queries, read from a secondary replica.
- For inserts and updates, manually fail over to a secondary replica. See [Initiate a planned or unplanned failover for Azure SQL Database](#).

The replica uses a different connection string, so you will need to update the connection string in your application.

Client runs out of connections in the connection pool.

Detection. Catch `System.InvalidOperationException` errors.

Recovery

- Retry the operation.
- As a mitigation plan, isolate the connection pools for each use case, so that one use case can't dominate all the connections.
- Increase the maximum connection pools.

Diagnostics. Application logs.

Database connection limit is reached.

Detection. Azure SQL Database limits the number of concurrent workers, logins, and sessions. The limits depend on the service tier. For more information, see [Azure SQL Database resource limits](#).

To detect these errors, catch `System.Data.SqlClient.SqlException` and check the value of `SqlException.Number` for the SQL error code. For a list of relevant error codes, see [SQL error codes for SQL Database client applications: Database connection error and other issues](#).

Recovery. These errors are considered transient, so retrying may resolve the issue. If you consistently hit these errors, consider scaling the database.

Diagnostics. - The `sys.event_log` query returns successful database connections, connection failures, and deadlocks.

- Create an [alert rule](#) for failed connections.
- Enable [SQL Database auditing](#) and check for failed logins.

Service Bus Messaging

Reading a message from a Service Bus queue fails.

Detection. Catch exceptions from the client SDK. The base class for Service Bus exceptions is `MessagingException`. If the error is transient, the `IsTransient` property is true.

For more information, see [Service Bus messaging exceptions](#).

Recovery

1. Retry on transient failures. See [Service Bus retry guidelines](#).
2. Messages that cannot be delivered to any receiver are placed in a *dead-letter queue*. Use this queue to see which messages could not be received. There is no automatic cleanup of the dead-letter queue. Messages remain there until you explicitly retrieve them. See [Overview of Service Bus dead-letter queues](#).

Writing a message to a Service Bus queue fails.

Detection. Catch exceptions from the client SDK. The base class for Service Bus exceptions is `MessagingException`. If the error is transient, the `IsTransient` property is true.

For more information, see [Service Bus messaging exceptions](#).

Recovery

1. The Service Bus client automatically retries after transient errors. By default, it uses exponential back-off. After the maximum retry count or maximum timeout period, the client throws an exception. For more information, see [Service Bus retry guidelines](#).
2. If the queue quota is exceeded, the client throws [QuotaExceededException](#). The exception message gives more details. Drain some messages from the queue before retrying, and consider using the Circuit Breaker pattern to avoid continued retries while the quota is exceeded. Also, make sure the [BrokeredMessage.TimeToLive](#) property is not set too high.
3. Within a region, resiliency can be improved by using [partitioned queues or topics](#). A non-partitioned queue or topic is assigned to one messaging store. If this messaging store is unavailable, all operations on that queue or topic will fail. A partitioned queue or topic is partitioned across multiple messaging stores.
4. For additional resiliency, create two Service Bus namespaces in different regions, and replicate the messages. You can use either active replication or passive replication.
 - Active replication: The client sends every message to both queues. The receiver listens on both queues. Tag messages with a unique identifier, so the client can discard duplicate messages.
 - Passive replication: The client sends the message to one queue. If there is an error, the client falls back to the other queue. The receiver listens on both queues. This approach reduces the number of duplicate messages that are sent. However, the receiver must still handle duplicate messages.

For more information, see [GeoReplication sample](#) and [Best practices for insulating applications against Service Bus outages and disasters](#).

Duplicate message.

Detection. Examine the `MessageId` and `DeliveryCount` properties of the message.

Recovery

- If possible, design your message processing operations to be idempotent. Otherwise, store message IDs of messages that are already processed, and check the ID before processing a message.
- Enable duplicate detection, by creating the queue with `RequiresDuplicateDetection` set to true. With this setting, Service Bus automatically deletes any message that is sent with the same `MessageId` as a previous message. Note the following:
 - This setting prevents duplicate messages from being put into the queue. It doesn't prevent a receiver from processing the same message more than once.
 - Duplicate detection has a time window. If a duplicate is sent beyond this window, it won't be detected.

Diagnostics. Log duplicated messages.

The application cannot process a particular message from the queue.

Detection. Application specific. For example, the message contains invalid data, or the business logic fails for some reason.

Recovery

There are two failure modes to consider.

- The receiver detects the failure. In this case, move the message to the dead-letter queue. Later, run a separate process to examine the messages in the dead-letter queue.
- The receiver fails in the middle of processing the message — for example, due to an unhandled exception. To

handle this case, use `PeekLock` mode. In this mode, if the lock expires, the message becomes available to other receivers. If the message exceeds the maximum delivery count or the time-to-live, the message is automatically moved to the dead-letter queue.

For more information, see [Overview of Service Bus dead-letter queues](#).

Diagnostics. Whenever the application moves a message to the dead-letter queue, write an event to the application logs.

Service Fabric

A request to a service fails.

Detection. The service returns an error.

Recovery

- Locate a proxy again (`ServiceProxy` or `ActorProxy`) and call the service/actor method again.
- **Stateful service.** Wrap operations on reliable collections in a transaction. If there is an error, the transaction will be rolled back. The request, if pulled from a queue, will be processed again.
- **Stateless service.** If the service persists data to an external store, all operations need to be idempotent.

Diagnostics. Application log

Service Fabric node is shut down.

Detection. A cancellation token is passed to the service's `RunAsync` method. Service Fabric cancels the task before shutting down the node.

Recovery. Use the cancellation token to detect shutdown. When Service Fabric requests cancellation, finish any work and exit `RunAsync` as quickly as possible.

Diagnostics. Application logs

Storage

Writing data to Azure Storage fails

Detection. The client receives errors when writing.

Recovery

1. Retry the operation, to recover from transient failures. The [retry policy](#) in the client SDK handles this automatically.
2. Implement the Circuit Breaker pattern to avoid overwhelming storage.
3. If N retry attempts fail, perform a graceful fallback. For example:
 - Store the data in a local cache, and forward the writes to storage later, when the service becomes available.
 - If the write action was in a transactional scope, compensate the transaction.

Diagnostics. Use [storage metrics](#).

Reading data from Azure Storage fails.

Detection. The client receives errors when reading.

Recovery

1. Retry the operation, to recover from transient failures. The [retry policy](#) in the client SDK handles this

- automatically.
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 non-transient errors, implement the [Circuit Breaker](#) pattern.
- If the calling VM exceeds its network egress limit, the outbound queue will fill up. If the outbound queue is consistently full, consider scaling out.

Diagnostics. Log events at service boundaries.

VM instance becomes unavailable or unhealthy.

Detection. Configure a Load Balancer [health probe](#) that signals whether the VM instance is healthy. The probe should check whether critical functions are responding correctly.

Recovery. For each application tier, put multiple VM instances into the same availability set, and place a load balancer in front of the VMs. If the health probe fails, the Load Balancer stops sending new connections to the unhealthy instance.

Diagnostics. - Use Load Balancer [log analytics](#).

- Configure your monitoring system to monitor all of the health monitoring endpoints.

Operator accidentally shuts down a VM.

Detection. N/A

Recovery. Set a resource lock with `ReadOnly` level. See [Lock resources with Azure Resource Manager](#).

Diagnostics. Use [Azure Activity Logs](#).

WebJobs

Continuous job stops running when the SCM host is idle.

Detection. Pass a cancellation token to the WebJob function. For more information, see [Graceful shutdown](#).

Recovery. Enable the `Always On` setting in the web app. For more information, see [Run Background tasks with WebJobs](#).

Application design

Application can't handle a spike in incoming requests.

Detection. Depends on the application. Typical symptoms:

- The website starts returning HTTP 5xx error codes.
- Dependent services, such as database or storage, start to throttle requests. Look for HTTP errors such as HTTP 429 (Too Many Requests), depending on the service.
- HTTP queue length grows.

Recovery

- Scale out to handle increased load.
- Mitigate failures to avoid having cascading failures disrupt the entire application. Mitigation strategies include:
 - Implement the [Throttling Pattern](#) to avoid overwhelming backend systems.
 - Use [queue-based load leveling](#) to buffer requests and process them at an appropriate pace.
 - Prioritize certain clients. For example, if the application has free and paid tiers, throttle customers on the free tier, but not paid customers. See [Priority queue pattern](#).

Diagnostics. Use [App Service diagnostic logging](#). Use a service such as [Azure Log Analytics](#), [Application Insights](#), or [New Relic](#) to help understand the diagnostic logs.

One of the operations in a workflow or distributed transaction fails.

Detection. After N retry attempts, it still fails.

Recovery

- As a mitigation plan, implement the [Scheduler Agent Supervisor](#) pattern to manage the entire workflow.
- Don't retry on timeouts. There is a low success rate for this error.
- Queue work, in order to retry later.

Diagnostics. Log all operations (successful and failed), including compensating actions. Use correlation IDs, so that you can track all operations within the same transaction.

A call to a remote service fails.

Detection. HTTP error code.

Recovery

1. Retry on transient failures.
2. If the call fails after N attempts, take a fallback action. (Application specific.)
3. Implement the [Circuit Breaker pattern](#) to avoid cascading failures.

Diagnostics. Log all remote call failures.

Next steps

For more information about the FMA process, see [Resilience by design for cloud services](#) (PDF download).

Azure for AWS Professionals

4/11/2018 • 15 minutes to read • [Edit Online](#)

This article helps Amazon Web Services (AWS) experts understand the basics of Microsoft Azure accounts, platform, and services. It also covers key similarities and differences between the AWS and Azure platforms.

You'll learn:

- How accounts and resources are organized in Azure.
- How available solutions are structured in Azure.
- How the major Azure services differ from AWS services.

Azure and AWS built their capabilities independently over time so that each has important implementation and design differences.

Overview

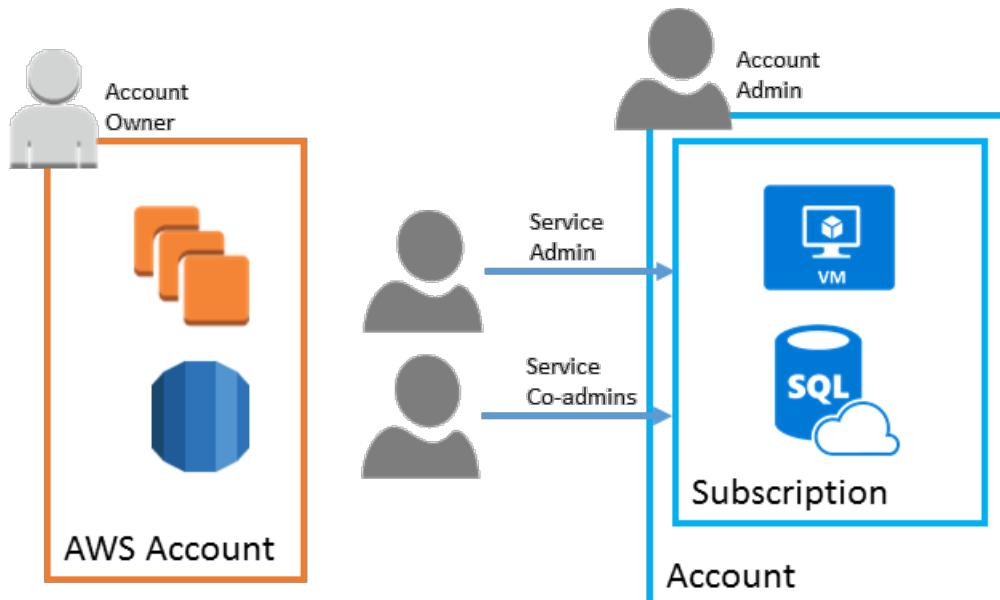
Like AWS, Microsoft Azure is built around a core set of compute, storage, database, and networking services. In many cases, both platforms offer a basic equivalence between the products and services they offer. Both AWS and Azure allow you to build highly available solutions based on Windows or Linux hosts. So, if you're used to development using Linux and OSS technology, both platforms can do the job.

While the capabilities of both platforms are similar, the resources that provide those capabilities are often organized differently. Exact one-to-one relationships between the services required to build a solution are not always clear. There are also cases where a particular service might be offered on one platform, but not the other. See [charts of comparable Azure and AWS services](#).

Accounts and subscriptions

Azure services can be purchased using several pricing options, depending on your organization's size and needs. See the [pricing overview](#) page for details.

[Azure subscriptions](#) are a grouping of resources with an assigned owner responsible for billing and permissions management. Unlike AWS, where any resources created under the AWS account are tied to that account, subscriptions exist independently of their owner accounts, and can be reassigned to new owners as needed.



Comparison of structure and ownership of AWS accounts and Azure subscriptions

Subscriptions are assigned three types of administrator accounts:

- **Account Administrator** - The subscription owner and the account billed for the resources used in the subscription. The account administrator can only be changed by transferring ownership of the subscription.
- **Service Administrator** - This account has rights to create and manage resources in the subscription, but is not responsible for billing. By default, the account administrator and service administrator are assigned to the same account. The account administrator can assign a separate user to the service administrator account for managing the technical and operational aspects of a subscription. There is only one service administrator per subscription.
- **Co-administrator** - There can be multiple co-administrator accounts assigned to a subscription. Co-administrators cannot change the service administrator, but otherwise have full control over subscription resources and users.

Below the subscription level user roles and individual permissions can also be assigned to specific resources, similarly to how permissions are granted to IAM users and groups in AWS. In Azure all user accounts are associated with either a Microsoft Account or Organizational Account (an account managed through an Azure Active Directory).

Like AWS accounts, subscriptions have default service quotas and limits. For a full list of these limits, see [Azure subscription and service limits, quotas, and constraints](#). These limits can be increased up to the maximum by [filing a support request in the management portal](#).

See also

- [How to add or change Azure administrator roles](#)
- [How to download your Azure billing invoice and daily usage data](#)

Resource management

The term "resource" in Azure is used in the same way as in AWS, meaning any compute instance, storage object, networking device, or other entity you can create or configure within the platform.

Azure resources are deployed and managed using one of two models: [Azure Resource Manager](#), or the older [Azure classic deployment model](#). Any new resources are created using the Resource Manager model.

Resource groups

Both Azure and AWS have entities called "resource groups" that organize resources such as VMs, storage, and virtual networking devices. However, [Azure resource groups](#) are not directly comparable to AWS resource groups.

While AWS allows a resource to be tagged into multiple resource groups, an Azure resource is always associated with one resource group. A resource created in one resource group can be moved to another group, but can only be in one resource group at a time. Resource groups are the fundamental grouping used by Azure Resource Manager.

Resources can also be organized using [tags](#). Tags are key-value pairs that allow you to group resources across your subscription irrespective of resource group membership.

Management interfaces

Azure offers several ways to manage your resources:

- [Web interface](#). Like the AWS Dashboard, the Azure portal provides a full web-based management interface for Azure resources.

- [REST API](#). The Azure Resource Manager REST API provides programmatic access to most of the features available in the Azure portal.
- [Command Line](#). The Azure CLI 2.0 tool provides a command-line interface capable of creating and managing Azure resources. Azure CLI is available for [Windows, Linux, and Mac OS](#).
- [PowerShell](#). The Azure modules for PowerShell allow you to execute automated management tasks using a script. PowerShell is available for [Windows, Linux, and Mac OS](#).
- [Templates](#). Azure Resource Manager templates provide similar JSON template-based resource management capabilities to the AWS CloudFormation service.

In each of these interfaces, the resource group is central to how Azure resources get created, deployed, or modified. This is similar to the role a "stack" plays in grouping AWS resources during CloudFormation deployments.

The syntax and structure of these interfaces are different from their AWS equivalents, but they provide comparable capabilities. In addition, many third party management tools used on AWS, like [Hashicorp's Terraform](#) and [Netflix Spinnaker](#), are also available on Azure.

See also

- [Azure resource group guidelines](#)

Regions and zones (high availability)

Failures can vary in the scope of their impact. Some hardware failures, such as a failed disk, may affect a single host machine. A failed network switch could affect a whole server rack. Less common are failures that disrupt a whole data center, such as loss of power in a data center. Rarely, an entire region could become unavailable.

One of the main ways to make an application resilient is through redundancy. But you need to plan for this redundancy when you design the application. Also, the level of redundancy that you need depends on your business requirements — not every application needs redundancy across regions to guard against a regional outage. In general, there is a tradeoff between greater redundancy and reliability versus higher cost and complexity.

In AWS, a region is divided into two or more Availability Zones. An Availability Zone corresponds with a physically isolated datacenter in the geographic region. Azure has a number of features to make an application redundant at every level of failure, including **availability sets**, **availability zones**, and **paired regions**.

The following table summarizes each option.

	AVAILABILITY SET	AVAILABILITY ZONE	PAIRED REGION
Scope of failure	Rack	Datacenter	Region
Request routing	Load Balancer	Cross-zone Load Balancer	Traffic Manager
Network latency	Very low	Low	Mid to high
Virtual networking	VNet	VNet	Cross-region VNet peering

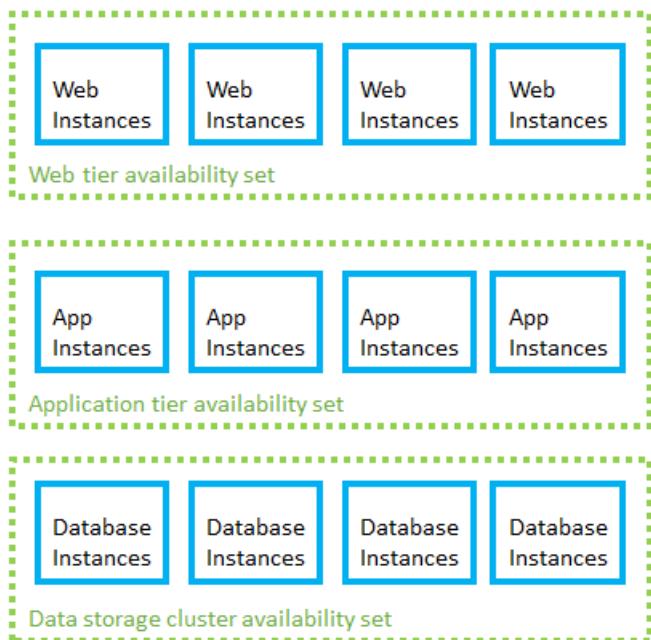
Availability sets

To protect against localized hardware failures, such as a disk or network switch failing, deploy two or more VMs in an availability set. An availability set consists of two or more *fault domains* that share a common power source and network switch. VMs in an availability set are distributed across the fault domains, so if a hardware failure affects one fault domain, network traffic can still be routed the VMs in the other fault domains. For more information

about Availability Sets, see [Manage the availability of Windows virtual machines in Azure](#).

When VM instances are added to availability sets, they are also assigned an [update domain](#). An update domain is a group of VMs that are set for planned maintenance events at the same time. Distributing VMs across multiple update domains ensures that planned update and patching events affect only a subset of these VMs at any given time.

Availability sets should be organized by the instance's role in your application to ensure one instance in each role is operational. For example, in a three-tier web application, create separate availability sets for the front-end, application, and data tiers.



Availability zones

An [Availability Zone](#) is a physically separate zone within an Azure region. Each Availability Zone has a distinct power source, network, and cooling. Deploying VMs across availability zones helps to protect an application against datacenter-wide failures.

Paired regions

To protect an application against a regional outage, you can deploy the application across multiple regions, using [Azure Traffic Manager](#) to distribute internet traffic to the different regions. Each Azure region is paired with another region. Together, these form a [regional pair](#). With the exception of Brazil South, regional pairs are located within the same geography in order to meet data residency requirements for tax and law enforcement jurisdiction purposes.

Unlike Availability Zones, which are physically separate datacenters but may be in relatively nearby geographic areas, paired regions are usually separated by at least 300 miles. This is intended to ensure larger scale disasters only impact one of the regions in the pair. Neighboring pairs can be set to sync database and storage service data, and are configured so that platform updates are rolled out to only one region in the pair at a time.

Azure [geo-redundant storage](#) is automatically backed up to the appropriate paired region. For all other resources, creating a fully redundant solution using paired regions means creating a full copy of your solution in both regions.

See also

- [Regions and availability for virtual machines in Azure](#)
- [High availability for Azure applications](#)
- [Disaster recovery for Azure applications](#)
- [Planned maintenance for Linux virtual machines in Azure](#)

Services

Consult the [complete AWS and Azure service comparison matrix](#) for a full listing of how all services map between platforms.

Not all Azure products and services are available in all regions. Consult the [Products by Region](#) page for details. You can find the uptime guarantees and downtime credit policies for each Azure product or service on the [Service Level Agreements](#) page.

The following sections provide a brief explanation of how commonly used features and services differ between the AWS and Azure platforms.

Compute services

EC2 Instances and Azure virtual machines

Although AWS instance types and Azure virtual machine sizes breakdown in a similar way, there are differences in the RAM, CPU, and storage capabilities.

- [Amazon EC2 Instance Types](#)
- [Sizes for virtual machines in Azure \(Windows\)](#)
- [Sizes for virtual machines in Azure \(Linux\)](#)

Unlike AWS' per second billing, Azure on-demand VMs are billed by the minute.

Azure has no equivalent to EC2 Spot Instances or Dedicated Hosts.

EBS and Azure Storage for VM disks

Durable data storage for Azure VMs is provided by [data disks](#) residing in blob storage. This is similar to how EC2 instances store disk volumes on Elastic Block Store (EBS). [Azure temporary storage](#) also provides VMs the same low-latency temporary read-write storage as EC2 Instance Storage (also called ephemeral storage).

Higher performance disk IO is supported using [Azure premium storage](#). This is similar to the Provisioned IOPS storage options provided by AWS.

Lambda, Azure Functions, Azure Web-Jobs, and Azure Logic Apps

[Azure Functions](#) is the primary equivalent of AWS Lambda in providing serverless, on-demand code. However, Lambda functionality also overlaps with other Azure services:

- [WebJobs](#) - allow you to create scheduled or continuously running background tasks.
- [Logic Apps](#) - provides communications, integration, and business rule management services.

Autoscaling, Azure VM scaling, and Azure App Service Autoscale

Autoscaling in Azure is handled by two services:

- [VM scale sets](#) - allow you to deploy and manage an identical set of VMs. The number of instances can autoscale based on performance needs.
- [App Service Autoscale](#) - provides the capability to autoscale Azure App Service solutions.

Container Service

The [Azure Container Service](#) supports Docker containers managed through Docker Swarm, Kubernetes, or DC/OS.

Other compute services

Azure offers several compute services that do not have direct equivalents in AWS:

- [Azure Batch](#) - allows you to manage compute-intensive work across a scalable collection of virtual machines.

- Service Fabric - platform for developing and hosting scalable [microservice](#) solutions.

See also

- [Create a Linux VM on Azure using the Portal](#)
- [Azure Reference Architecture: Running a Linux VM on Azure](#)
- [Get started with Node.js web apps in Azure App Service](#)
- [Azure Reference Architecture: Basic web application](#)
- [Create your first Azure Function](#)

Storage

S3/EBS/EFS and Azure Storage

In the AWS platform, cloud storage is primarily broken down into three services:

- **Simple Storage Service (S3)** - basic object storage. Makes data available through an Internet accessible API.
- **Elastic Block Storage (EBS)** - block level storage, intended for access by a single VM.
- **Elastic File System (EFS)** - file storage meant for use as shared storage for up to thousands of EC2 instances.

In Azure Storage, subscription-bound [storage accounts](#) allow you to create and manage the following storage services:

- [Blob storage](#) - stores any type of text or binary data, such as a document, media file, or application installer. You can set Blob storage for private access or share contents publicly to the Internet. Blob storage serves the same purpose as both AWS S3 and EBS.
- [Table storage](#) - stores structured datasets. Table storage is a NoSQL key-attribute data store that allows for rapid development and fast access to large quantities of data. Similar to AWS' SimpleDB and DynamoDB services.
- [Queue storage](#) - provides messaging for workflow processing and for communication between components of cloud services.
- [File storage](#) - offers shared storage for legacy applications using the standard server message block (SMB) protocol. File storage is used in a similar manner to EFS in the AWS platform.

Glacier and Azure Storage

[Azure Archive Blob Storage](#) is comparable to AWS Glacier storage service. It is intended for rarely accessed data that is stored for at least 180 days and can tolerate several hours of retrieval latency.

For data that is infrequently accessed but must be available immediately when accessed, [Azure Cool Blob Storage tier](#) provides cheaper storage than standard blob storage. This storage tier is comparable to AWS S3 - Infrequent Access storage service.

See also

- [Microsoft Azure Storage Performance and Scalability Checklist](#)
- [Azure Storage security guide](#)
- [Patterns & Practices: Content Delivery Network \(CDN\) guidance](#)

Networking

Elastic Load Balancing, Azure Load Balancer, and Azure Application Gateway

The Azure equivalents of the two Elastic Load Balancing services are:

- [Load Balancer](#) - provides the same capabilities as the AWS Classic Load Balancer, allowing you to distribute traffic for multiple VMs at the network level. It also provides failover capability.
- [Application Gateway](#) - offers application-level rule-based routing comparable to the AWS Application Load Balancer.

Route 53, Azure DNS, and Azure Traffic Manager

In AWS, Route 53 provides both DNS name management and DNS-level traffic routing and failover services. In Azure this is handled through two services:

- [Azure DNS](#) provides domain and DNS management.
- [Traffic Manager](#) provides DNS level traffic routing, load balancing, and failover capabilities.

Direct Connect and Azure ExpressRoute

Azure provides similar site-to-site dedicated connections through its [ExpressRoute](#) service. ExpressRoute allows you to connect your local network directly to Azure resources using a dedicated private network connection. Azure also offers more conventional [site-to-site VPN connections](#) at a lower cost.

See also

- [Create a virtual network using the Azure portal](#)
- [Plan and design Azure Virtual Networks](#)
- [Azure Network Security Best Practices](#)

Database services

RDS and Azure relational database services

Azure provides several different relational database services that are the equivalent of AWS' Relational Database Service (RDS).

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

Other database engines such as [SQL Server](#), [Oracle](#), and [MySQL](#) can be deployed using Azure VM Instances.

Costs for AWS RDS are determined by the amount of hardware resources that your instance uses, like CPU, RAM, storage, and network bandwidth. In the Azure database services, cost depends on your database size, concurrent connections, and throughput levels.

See also

- [Azure SQL Database Tutorials](#)
- [Configure geo-replication for Azure SQL Database with the Azure portal](#)
- [Introduction to Cosmos DB: A NoSQL JSON Database](#)
- [How to use Azure Table storage from Node.js](#)

Security and identity

Directory service and Azure Active Directory

Azure splits up directory services into the following offerings:

- [Azure Active Directory](#) - cloud based directory and identity management service.
- [Azure Active Directory B2B](#) - enables access to your corporate applications from partner-managed identities.
- [Azure Active Directory B2C](#) - service offering support for single sign-on and user management for consumer facing applications.

- [Azure Active Directory Domain Services](#) - hosted domain controller service, allowing Active Directory compatible domain join and user management functionality.

Web application firewall

In addition to the [Application Gateway Web Application Firewall](#), you can also [use web application firewalls](#) from third-party vendors like [Barracuda Networks](#).

See also

- [Getting started with Microsoft Azure security](#)
- [Azure Identity Management and access control security best practices](#)

Application and messaging services

Simple Email Service

AWS provides the Simple Email Service (SES) for sending notification, transactional, or marketing emails. In Azure, third-party solutions like [Sendgrid](#) provide email services.

Simple Queueing Service

AWS Simple Queueing Service (SQS) provides a messaging system for connecting applications, services, and devices within the AWS platform. Azure has two services that provide similar functionality:

- [Queue storage](#) - a cloud messaging service that allows communication between application components within the Azure platform.
- [Service Bus](#) - a more robust messaging system for connecting applications, services, and devices. Using the related [Service Bus relay](#), Service Bus can also connect to remotely hosted applications and services.

Device Farm

The AWS Device Farm provides cross-device testing services. In Azure, [Xamarin Test Cloud](#) provides similar cross-device front-end testing for mobile devices.

In addition to front-end testing, the [Azure DevTest Labs](#) provides back end testing resources for Linux and Windows environments.

See also

- [How to use Queue storage from Node.js](#)
- [How to use Service Bus queues](#)

Analytics and big data

The [Cortana Intelligence Suite](#) is Azure's package of products and services designed to capture, organize, analyze, and visualize large amounts of data. The Cortana suite consists of the following services:

- [HDInsight](#) - managed Apache distribution that includes Hadoop, Spark, Storm, or HBase.
- [Data Factory](#) - provides data orchestration and data pipeline functionality.
- [SQL Data Warehouse](#) - large-scale relational data storage.
- [Data Lake Store](#) - large-scale storage optimized for big data analytics workloads.
- [Machine Learning](#) - used to build and apply predictive analytics on data.
- [Stream Analytics](#) - real-time data analysis.
- [Data Lake Analytics](#) - large-scale analytics service optimized to work with Data Lake Store
- [PowerBI](#) - used to power data visualization.

See also

- [Cortana Intelligence Gallery](#)

- [Understanding Microsoft big data solutions](#)
- [Azure Data Lake & Azure HDInsight Blog](#)

Internet of Things

See also

- [Get started with Azure IoT Hub](#)
- [Comparison of IoT Hub and Event Hubs](#)

Mobile services

Notifications

Notification Hubs do not support sending SMS or email messages, so third-party services are needed for those delivery types.

See also

- [Create an Android app](#)
- [Authentication and Authorization in Azure Mobile Apps](#)
- [Sending push notifications with Azure Notification Hubs](#)

Management and monitoring

See also

- [Monitoring and diagnostics guidance](#)
- [Best practices for creating Azure Resource Manager templates](#)
- [Azure Resource Manager Quickstart templates](#)

Next steps

- [Complete AWS and Azure service comparison matrix](#)
- [Interactive Azure Platform Big Picture](#)
- [Get started with Azure](#)
- [Azure solution architectures](#)
- [Azure Reference Architectures](#)
- [Patterns & Practices: Azure Guidance](#)
- [Free Online Course: Microsoft Azure for AWS Experts](#)

Designing, building, and operating microservices on Azure

1/8/2018 • 6 minutes to read • [Edit Online](#)

Microservices have become a popular architectural style for building cloud applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. To be more than just a buzzword, however, microservices require a different approach to designing and building applications.

In this set of articles, we explore how to build and run a microservices architecture on Azure. Topics include:

- Using Domain Driven Design (DDD) to design a microservices architecture.
- Choosing the right Azure technologies for compute, storage, messaging, and other elements of the design.
- Understanding microservices design patterns.
- Designing for resiliency, scalability, and performance.
- Building a CI/CD pipeline.

Throughout, we focus on an end-to-end scenario: A drone delivery service that lets customers schedule packages to be picked up and delivered via drone. You can find the code for our reference implementation on GitHub



[Reference implementation](#)

But first, let's start with fundamentals. What are microservices, and what are the advantages of adopting a microservices architecture?

Why build microservices?

In a microservices architecture, the application is composed of small, independent services. Here are some of the defining characteristics of microservices:

- Each microservice implements a single business capability.
- A microservice is small enough that a single small team of developers can write and maintain it.
- Microservices run in separate processes, communicating through well-defined APIs or messaging patterns.
- Microservices do not share data stores or data schemas. Each microservice is responsible for managing its own data.
- Microservices have separate code bases, and do not share source code. They may use common utility libraries, however.
- Each microservice can be deployed and updated independently of other services.

Done correctly, microservices can provide a number of useful benefits:

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process; as a result, new features may be held up waiting for a bug fix to be integrated, tested, and published.
- **Small code, small teams.** A microservice should be small enough that a single feature team can build, test, and deploy it. Small code bases are easier to understand. In a large monolithic application, there is a

tendency over time for code dependencies to become tangled, so that adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features. Small team sizes also promote greater agility. The "two-pizza rule" says that a team should be small enough that two pizzas can feed the team. Obviously that's not an exact metric and depends on team appetites! But the point is that large groups tend to be less productive, because communication is slower, management overhead goes up, and agility diminishes.

- **Mix of technologies.** Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Resiliency.** If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).
- **Scalability.** A microservices architecture allows each microservice to be scaled independently of the others. That lets you scale out subsystems that require more resources, without scaling out the entire application. If you deploy services inside containers, you can also pack a higher density of microservices onto a single host, which allows for more efficient utilization of resources.
- **Data isolation.** It is much easier to perform schema updates, because only a single microservice is impacted. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

No free lunch

These benefits don't come for free. This series of articles is designed to address some of the challenges of building microservices that are resilient, scalable, and manageable.

- **Service boundaries.** When you build microservices, you need to think carefully about where to draw the boundaries between services. Once services are built and deployed in production, it can be hard to refactor across those boundaries. Choosing the right service boundaries is one of the biggest challenges when designing a microservices architecture. How big should each service be? When should functionality be factored across several services, and when should it be kept inside the same service? In this guide, we describe an approach that uses domain-driven design to find service boundaries. It starts with [Domain analysis](#) to find the bounded contexts, then applies a set of [tactical DDD patterns](#) based on functional and non-functional requirements.
- **Data consistency and integrity.** A basic principle of microservices is that each service manages its own data. This keeps services decoupled, but can lead to challenges with data integrity or redundancy. We explore some of these issues in the [Data considerations](#).
- **Network congestion and latency.** The use of many small, granular services can result in more interservice communication and longer end-to-end latency. The chapter [Interservice communication](#) describes considerations for messaging between services. Both synchronous and asynchronous communication have a place in microservices architectures. Good [API design](#) is important so that services remain loosely coupled, and can be independently deployed and updated.
- **Complexity.** A microservices application has more moving parts. Each service may be simple, but the services have to work together as a whole. A single user operation may involve multiple services. In the chapter [Ingestion and workflow](#), we examine some of the issues around ingesting requests at high throughput, coordinating a workflow, and handling failures.
- **Communication between clients and the application.** When you decompose an application into many small services, how should clients communicate with those services? Should a client call each individual service directly, or route requests through an [API Gateway](#)?
- **Monitoring.** Monitoring a distributed application can be a lot harder than a monolithic application, because

you must correlate telemetry from multiple services. The chapter [Logging and monitoring](#) addresses these concerns.

- **Continuous integration and delivery (CI/CD).** One of the main goals of microservices is agility. To achieve this, you must have automated and robust [CI/CD](#), so that you can quickly and reliably deploy individual services into test and production environments.

The Drone Delivery application

To explore these issues, and to illustrate some of the best practices for a microservices architecture, we created a reference implementation that we call the Drone Delivery application. You can find the reference implementation on [GitHub](#).

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

NOTE

For help in choosing between a microservices architecture and other architectural styles, see the [Azure Application Architecture Guide](#).

Our reference implementation uses Kubernetes with [Azure Container Service \(ACS\)](#). However, many of the high-level architectural decisions and challenges will apply to any container orchestrator, including [Azure Service Fabric](#).

[Domain analysis](#)

Manage Identity in Multitenant Applications

8/14/2017 • 3 minutes to read • [Edit Online](#)

This series of articles describes best practices for multitenancy, when using Azure AD for authentication and identity management.



[Sample code](#)

When you're building a multitenant application, one of the first challenges is managing user identities, because now every user belongs to a tenant. For example:

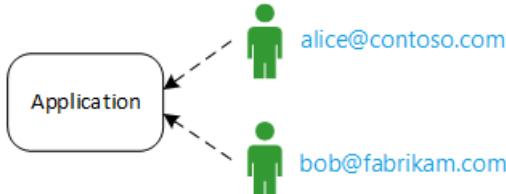
- Users sign in with their organizational credentials.
- Users should have access to their organization's data, but not data that belongs to other tenants.
- An organization can sign up for the application, and then assign application roles to its members.

Azure Active Directory (Azure AD) has some great features that support all of these scenarios.

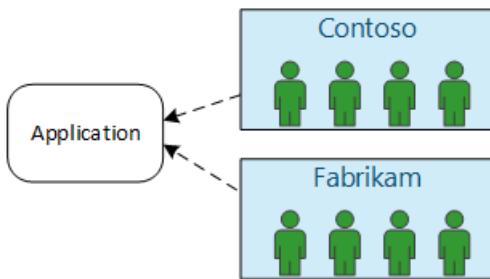
To accompany this series of articles, we created a complete [end-to-end implementation](#) of a multitenant application. The articles reflect what we learned in the process of building the application. To get started with the application, see [Run the Surveys application](#).

Introduction

Let's say you're writing an enterprise SaaS application to be hosted in the cloud. Of course, the application will have users:



But those users belong to organizations:



Example: Tailspin sells subscriptions to its SaaS application. Contoso and Fabrikam sign up for the app. When Alice (`alice@contoso`) signs in, the application should know that Alice is part of Contoso.

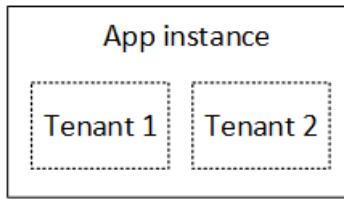
- Alice *should* have access to Contoso data.
- Alice *should not* have access to Fabrikam data.

This guidance will show you how to manage user identities in a multitenant application, using [Azure Active Directory](#) (Azure AD) to handle sign-in and authentication.

What is multitenancy?

A *tenant* is a group of users. In a SaaS application, the tenant is a subscriber or customer of the application. *Multitenancy* is an architecture where multiple tenants share the same physical instance of the app. Although tenants share physical resources (such as VMs or storage), each tenant gets its own logical instance of the app.

Typically, application data is shared among the users within a tenant, but not with other tenants.

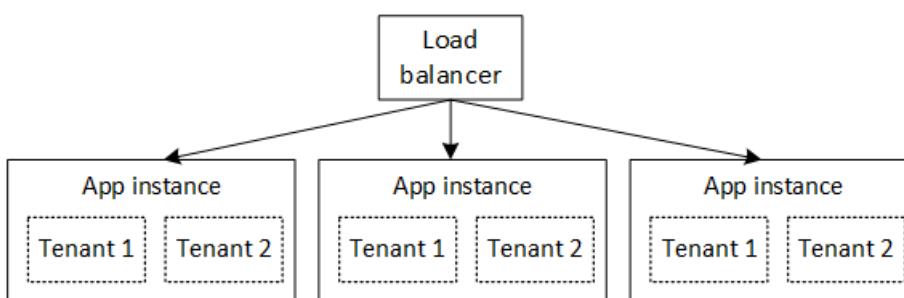


Compare this architecture with a single-tenant architecture, where each tenant has a dedicated physical instance. In a single-tenant architecture, you add tenants by spinning up new instances of the app.



Multitenancy and horizontal scaling

To achieve scale in the cloud, it's common to add more physical instances. This is known as *horizontal scaling* or *scaling out*. Consider a web app. To handle more traffic, you can add more server VMs and put them behind a load balancer. Each VM runs a separate physical instance of the web app.



Any request can be routed to any instance. Together, the system functions as a single logical instance. You can tear down a VM or spin up a new VM, without affecting users. In this architecture, each physical instance is multi-tenant, and you scale by adding more instances. If one instance goes down, it should not affect any tenant.

Identity in a multitenant app

In a multitenant app, you must consider users in the context of tenants.

Authentication

- Users sign into the app with their organization credentials. They don't have to create new user profiles for the app.
- Users within the same organization are part of the same tenant.
- When a user signs in, the application knows which tenant the user belongs to.

Authorization

- When authorizing a user's actions (say, viewing a resource), the app must take into account the user's tenant.
- Users might be assigned roles within the application, such as "Admin" or "Standard User". Role assignments should be managed by the customer, not by the SaaS provider.

Example. Alice, an employee at Contoso, navigates to the application in her browser and clicks the “Log in” button. She is redirected to a login screen where she enters her corporate credentials (username and password). At this point, she is logged into the app as `alice@contoso.com`. The application also knows that Alice is an admin user for this application. Because she is an admin, she can see a list of all the resources that belong to Contoso. However, she cannot view Fabrikam's resources, because she is an admin only within her tenant.

In this guidance, we'll look specifically at using Azure AD for identity management.

- We assume the customer stores their user profiles in Azure AD (including Office365 and Dynamics CRM tenants)
- Customers with on-premise Active Directory (AD) can use [Azure AD Connect](#) to sync their on-premise AD with Azure AD.

If a customer with on-premise AD cannot use Azure AD Connect (due to corporate IT policy or other reasons), the SaaS provider can federate with the customer's AD through Active Directory Federation Services (AD FS). This option is described in [Federating with a customer's AD FS](#).

This guidance does not consider other aspects of multitenancy such as data partitioning, per-tenant configuration, and so forth.

Next

Migrate an Azure Cloud Services application to Azure Service Fabric

4/11/2018 • 13 minutes to read • [Edit Online](#)



Sample code

This article describes migrating an application from Azure Cloud Services to Azure Service Fabric. It focuses on architectural decisions and recommended practices.

For this project, we started with a Cloud Services application called Surveys and ported it to Service Fabric. The goal was to migrate the application with as few changes as possible. In a later article, we will optimize the application for Service Fabric by adopting a microservices architecture.

Before reading this article, it will be useful to understand the basics of Service Fabric and microservices architectures in general. See the following articles:

- [Overview of Azure Service Fabric](#)
- [Why a microservices approach to building applications?](#)

About the Surveys application

In 2012, the patterns & practices group created an application called Surveys, for a book called [Developing Multi-tenant Applications for the Cloud](#). The book describes a fictitious company named Tailspin that designs and implements the Surveys application.

Surveys is a multitenant application that allows customers to create surveys. After a customer signs up for the application, members of the customer's organization can create and publish surveys, and collect the results for analysis. The application includes a public website where people can take a survey. Read more about the original Tailspin scenario [here](#).

Now Tailspin wants to move the Surveys application to a microservices architecture, using Service Fabric running on Azure. Because the application is already deployed as a Cloud Services application, Tailspin adopts a multi-phase approach:

1. Port the cloud services to Service Fabric, while minimizing changes to the application.
2. Optimize the application for Service Fabric, by moving to a microservices architecture.

This article describes the first phase. A later article will describe the second phase. In a real-world project, it's likely that both stages would overlap. While porting to Service Fabric, you would also start to re-architect the application into micro-services. Later you might refine the architecture further, perhaps dividing coarse-grained services into smaller services.

The application code is available on [GitHub](#). This repo contains both the Cloud Services application and the Service Fabric version.

The cloud service is an updated version of the original application from the *Developing Multi-tenant Applications* book.

Why Microservices?

An in-depth discussion of microservices is beyond scope of this article, but here are some of the benefits that

Tailspin hopes to get by moving to a microservices architecture:

- **Application upgrades.** Services can be deployed independently, so you can take an incremental approach to upgrading an application.
- **Resiliency and fault isolation.** If a service fails, other services continue to run.
- **Scalability.** Services can be scaled independently.
- **Flexibility.** Services are designed around business scenarios, not technology stacks, making it easier to migrate services to new technologies, frameworks, or data stores.
- **Agile development.** Individual services have less code than a monolithic application, making the code base easier to understand, reason about, and test.
- **Small, focused teams.** Because the application is broken down into many small services, each service can be built by a small focused team.

Why Service Fabric?

Service Fabric is a good fit for a microservices architecture, because most of the features needed in a distributed system are built into Service Fabric, including:

- **Cluster management.** Service Fabric automatically handles node failover, health monitoring, and other cluster management functions.
- **Horizontal scaling.** When you add nodes to a Service Fabric cluster, the application automatically scales, as services are distributed across the new nodes.
- **Service discovery.** Service Fabric provides a discovery service that can resolve the endpoint for a named service.
- **Stateless and stateful services.** Stateful services use [reliable collections](#), which can take the place of a cache or queue, and can be partitioned.
- **Application lifecycle management.** Services can be upgraded independently and without application downtime.
- **Service orchestration** across a cluster of machines.
- **Higher density** for optimizing resource consumption. A single node can host multiple services.

Service Fabric is used by various Microsoft services, including Azure SQL Database, Cosmos DB, Azure Event Hubs, and others, making it a proven platform for building distributed cloud applications.

Comparing Cloud Services with Service Fabric

The following table summarizes some of the important differences between Cloud Services and Service Fabric applications. For a more in-depth discussion, see [Learn about the differences between Cloud Services and Service Fabric before migrating applications](#).

	CLOUD SERVICES	SERVICE FABRIC
Application composition	Roles	Services
Density	One role instance per VM	Multiple services in a single node
Minimum number of nodes	2 per role	5 per cluster, for production deployments
State management	Stateless	Stateless or stateful*
Hosting	Azure	Cloud or on-premises

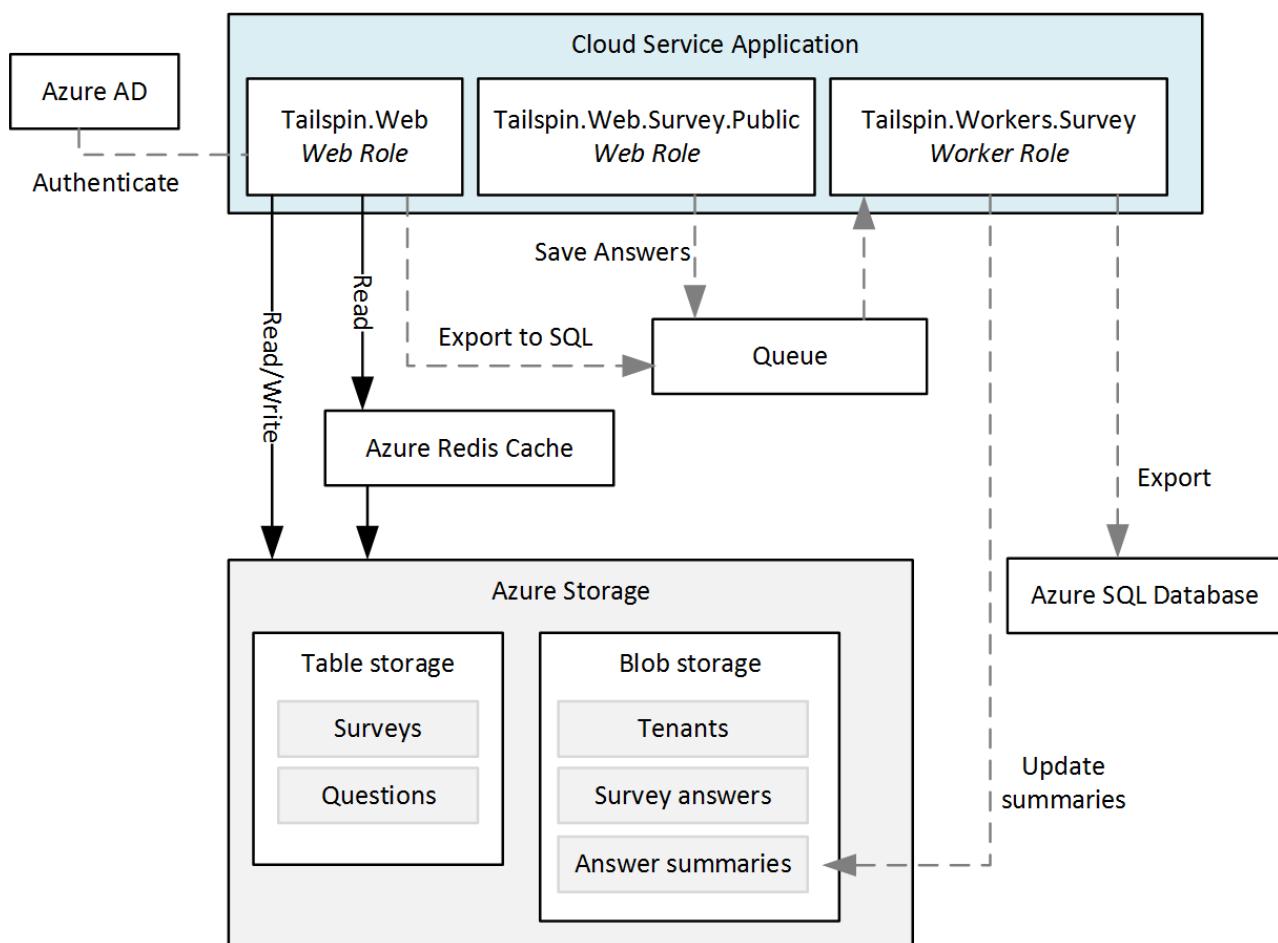
	CLOUD SERVICES	SERVICE FABRIC
Web hosting	IIS**	Self-hosting
Deployment model	Classic deployment model	Resource Manager
Packaging	Cloud service package files (.cspkg)	Application and service packages
Application update	VIP swap or rolling update	Rolling update
Auto-scaling	Built-in service	VM Scale Sets for auto scale out
Debugging	Local emulator	Local cluster

* Stateful services use [reliable collections](#) to store state across replicas, so that all reads are local to the nodes in the cluster. Writes are replicated across nodes for reliability. Stateless services can have external state, using a database or other external storage.

** Worker roles can also self-host ASP.NET Web API using OWIN.

The Surveys application on Cloud Services

The following diagram shows the architecture of the Surveys application running on Cloud Services.



The application consists of two web roles and a worker role.

- The **Tailspin.Web** web role hosts an ASP.NET website that Tailspin customers use to create and manage surveys. Customers also use this website to sign up for the application and manage their subscriptions. Finally, Tailspin administrators can use it to see the list of tenants and manage tenant data.

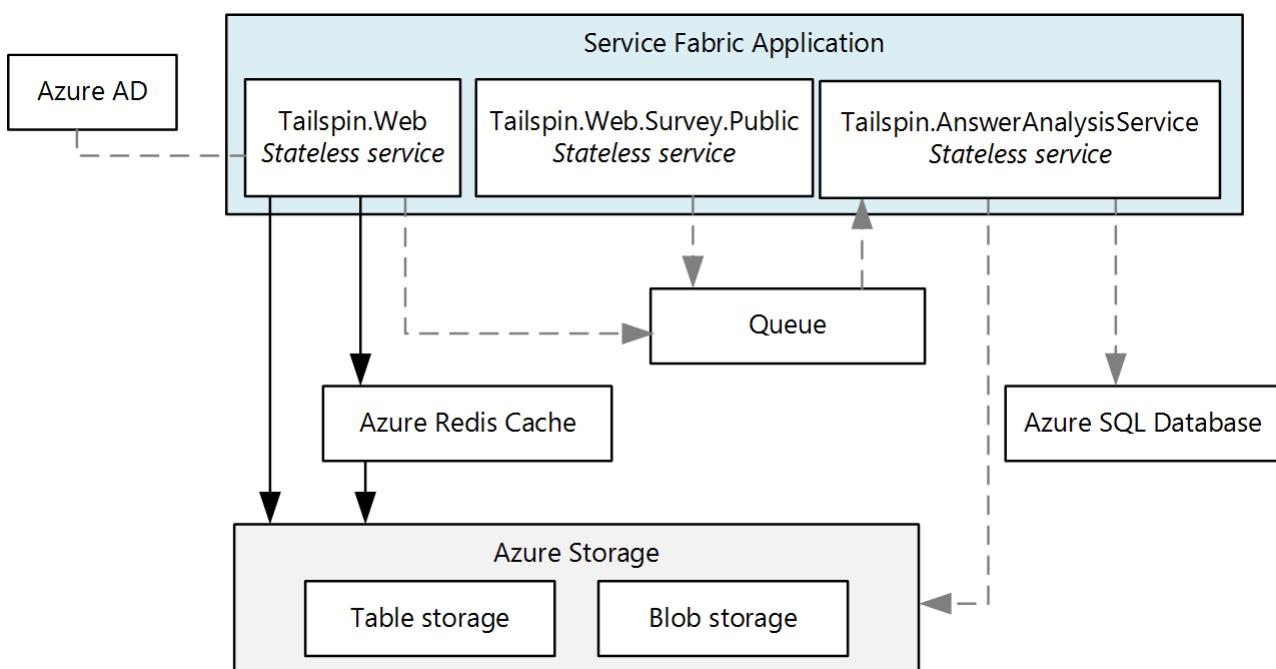
- The **Tailspin.Web.Survey.Public** web role hosts an ASP.NET website where people can take the surveys that Tailspin customers publish.
- The **Tailspin.Workers.Survey** worker role does background processing. The web roles put work items onto a queue, and the worker role processes the items. Two background tasks are defined: Exporting survey answers to Azure SQL Database, and calculating statistics for survey answers.

In addition to Cloud Services, the Surveys application uses some other Azure services:

- **Azure Storage** to store surveys, surveys answers, and tenant information.
- **Azure Redis Cache** to cache some of the data that is stored in Azure Storage, for faster read access.
- **Azure Active Directory** (Azure AD) to authenticate customers and Tailspin administrators.
- **Azure SQL Database** to store the survey answers for analysis.

Moving to Service Fabric

As mentioned, the goal of this phase was migrating to Service Fabric with the minimum necessary changes. To that end, we created stateless services corresponding to each cloud service role in the original application:



Intentionally, this architecture is very similar to the original application. However, the diagram hides some important differences. In the rest of this article, we'll explore those differences.

Converting the cloud service roles to services

As mentioned, we migrated each cloud service role to a Service Fabric service. Because cloud service roles are stateless, for this phase it made sense to create stateless services in Service Fabric.

For the migration, we followed the steps outlined in [Guide to converting Web and Worker Roles to Service Fabric stateless services](#).

Creating the web front-end services

In Service Fabric, a service runs inside a process created by the Service Fabric runtime. For a web front end, that means the service is not running inside IIS. Instead, the service must host a web server. This approach is called *self-hosting*, because the code that runs inside the process acts as the web server host.

The requirement to self-host means that a Service Fabric service can't use ASP.NET MVC or ASP.NET Web Forms,

because those frameworks require IIS and do not support self-hosting. Options for self-hosting include:

- [ASP.NET Core](#), self-hosted using the [Kestrel](#) web server.
- [ASP.NET Web API](#), self-hosted using [OWIN](#).
- Third-party frameworks such as [Nancy](#).

The original Surveys application uses ASP.NET MVC. Because ASP.NET MVC cannot be self-hosted in Service Fabric, we considered the following migration options:

- Port the web roles to ASP.NET Core, which can be self-hosted.
- Convert the web site into a single-page application (SPA) that calls a web API implemented using ASP.NET Web API. This would have required a complete redesign of the web front end.
- Keep the existing ASP.NET MVC code and deploy IIS in a Windows Server container to Service Fabric. This approach would require little or no code change.

The first option, porting to ASP.NET Core, allowed us to take advantage of the latest features in ASP.NET Core. To do the conversion, we followed the steps described in [Migrating From ASP.NET MVC to ASP.NET Core MVC](#).

NOTE

When using ASP.NET Core with Kestrel, you should place a reverse proxy in front of Kestrel to handle traffic from the Internet, for security reasons. For more information, see [Kestrel web server implementation in ASP.NET Core](#). The section [Deploying the application](#) describes a recommended Azure deployment.

HTTP listeners

In Cloud Services, a web or worker role exposes an HTTP endpoint by declaring it in the [service definition file](#). A web role must have at least one endpoint.

```
<!-- Cloud service endpoint -->
<Endpoints>
    <InputEndpoint name="HttpIn" protocol="http" port="80" />
</Endpoints>
```

Similarly, Service Fabric endpoints are declared in a service manifest:

```
<!-- Service Fabric endpoint -->
<Endpoints>
    <Endpoint Protocol="http" Name="ServiceEndpoint" Type="Input" Port="8002" />
</Endpoints>
```

Unlike a cloud service role, however, Service Fabric services can be co-located within the same node. Therefore, every service must listen on a distinct port. Later in this article, we'll discuss how client requests on port 80 or port 443 get routed to the correct port for the service.

A service must explicitly create listeners for each endpoint. The reason is that Service Fabric is agnostic about communication stacks. For more information, see [Build a web service front end for your application using ASP.NET Core](#).

Packaging and configuration

A cloud service contains the following configuration and package files:

FILE	DESCRIPTION
Service definition (.csdef)	Settings used by Azure to configure the cloud service. Defines the roles, endpoints, startup tasks, and the names of configuration settings.
Service configuration (.cscfg)	Per-deployment settings, including the number of role instances, endpoint port numbers, and the values of configuration settings.
Service package (.cspkg)	Contains the application code and configurations, and the service definition file.

There is one .csdef file for the entire application. You can have multiple .cscfg files for different environments, such as local, test, or production. When the service is running, you can update the .cscfg but not the .csdef. For more information, see [What is the Cloud Service model and how do I package it?](#)

Service Fabric has a similar division between a service *definition* and service *settings*, but the structure is more granular. To understand Service Fabric's configuration model, it helps to understand how a Service Fabric application is packaged. Here is the structure:

- ```

Application package
 - Service packages
 - Code package
 - Configuration package
 - Data package (optional)

```

The application package is what you deploy. It contains one or more service packages. A service package contains code, configuration, and data packages. The code package contains the binaries for the services, and the configuration package contains configuration settings. This model allows you to upgrade individual services without redeploying the entire application. It also lets you update just the configuration settings, without redeploying the code or restarting the service.

A Service Fabric application contains the following configuration files:

| FILE                    | LOCATION              | DESCRIPTION                                                                      |
|-------------------------|-----------------------|----------------------------------------------------------------------------------|
| ApplicationManifest.xml | Application package   | Defines the services that compose the application.                               |
| ServiceManifest.xml     | Service package       | Describes one or more services.                                                  |
| Settings.xml            | Configuration package | Contains configuration settings for the services defined in the service package. |

For more information, see [Model an application in Service Fabric](#).

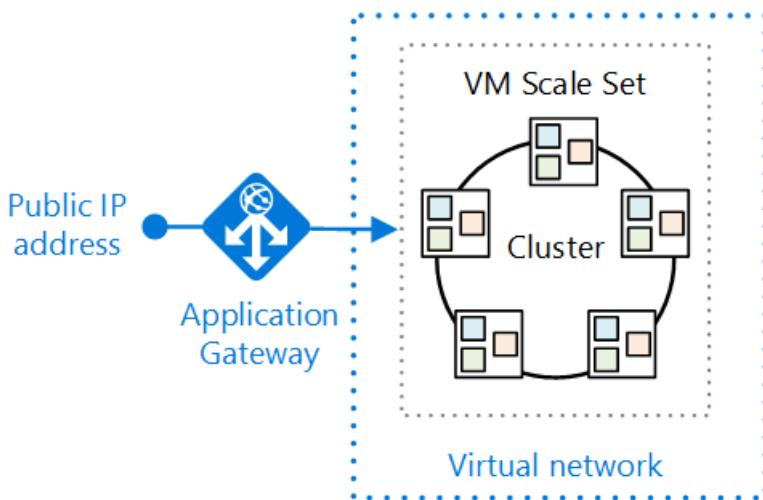
To support different configuration settings for multiple environments, use the following approach, described in [Manage application parameters for multiple environments](#):

1. Define the setting in the Setting.xml file for the service.
2. In the application manifest, define an override for the setting.
3. Put environment-specific settings into application parameter files.

## Deploying the application

Whereas Azure Cloud Services is a managed service, Service Fabric is a runtime. You can create Service Fabric clusters in many environments, including Azure and on premises. In this article, we focus on deploying to Azure.

The following diagram shows a recommended deployment:



The Service Fabric cluster is deployed to a [VM scale set](#). Scale sets are an Azure Compute resource that can be used to deploy and manage a set of identical VMs.

As mentioned, the Kestrel web server requires a reverse proxy for security reasons. This diagram shows [Azure Application Gateway](#), which is an Azure service that offers various layer 7 load balancing capabilities. It acts as a reverse-proxy service, terminating the client connection and forwarding requests to back-end endpoints. You might use a different reverse proxy solution, such as nginx.

### Layer 7 routing

In the [original Surveys application](#), one web role listened on port 80, and the other web role listened on port 443.

| PUBLIC SITE                               | SURVEY MANAGEMENT SITE                     |
|-------------------------------------------|--------------------------------------------|
| <code>http://tailspin.cloudapp.net</code> | <code>https://tailspin.cloudapp.net</code> |

Another option is to use layer 7 routing. In this approach, different URL paths get routed to different port numbers on the back end. For example, the public site might use URL paths starting with `/public/`.

Options for layer 7 routing include:

- Use Application Gateway.
- Use a network virtual appliance (NVA), such as nginx.
- Write a custom gateway as a stateless service.

Consider this approach if you have two or more services with public HTTP endpoints, but want them to appear as one site with a single domain name.

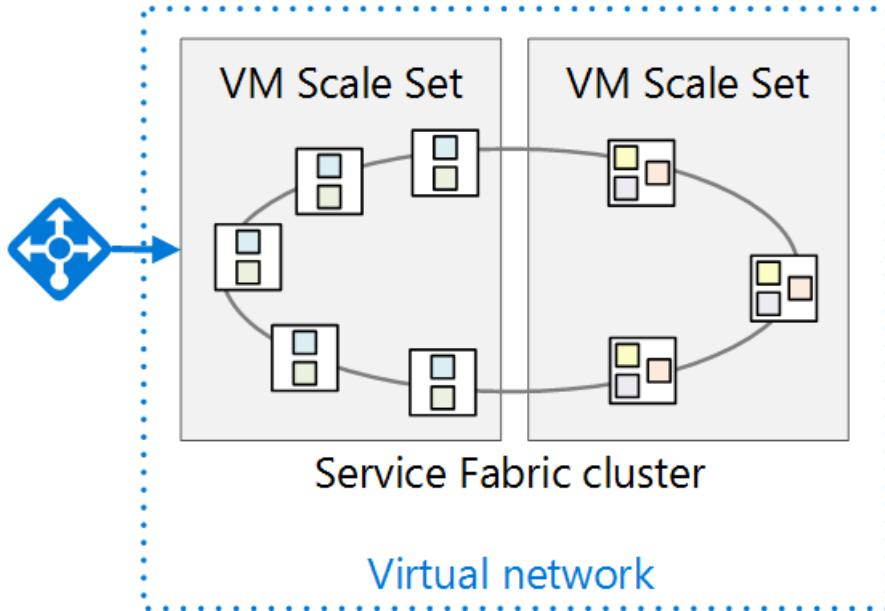
One approach that we *don't* recommend is allowing external clients to send requests through the Service Fabric [reverse proxy](#). Although this is possible, the reverse proxy is intended for service-to-service communication. Opening it to external clients exposes *any* service running in the cluster that has an HTTP endpoint.

### Node types and placement constraints

In the deployment shown above, all the services run on all the nodes. However, you can also group services, so that certain services run only on particular nodes within the cluster. Reasons to use this approach include:

- Run some services on different VM types. For example, some services might be compute-intensive or require GPUs. You can have a mix of VM types in your Service Fabric cluster.
- Isolate front-end services from back-end services, for security reasons. All the front-end services will run on one set of nodes, and the back-end services will run on different nodes in the same cluster.
- Different scale requirements. Some services might need to run on more nodes than other services. For example, if you define front-end nodes and back-end nodes, each set can be scaled independently.

The following diagram shows a cluster that separates front-end and back-end services:



To implement this approach:

- When you create the cluster, define two or more node types.
- For each service, use [placement constraints](#) to assign the service to a node type.

When you deploy to Azure, each node type is deployed to a separate VM scale set. The Service Fabric cluster spans all node types. For more information, see [The relationship between Service Fabric node types and Virtual Machine Scale Sets](#).

If a cluster has multiple node types, one node type is designated as the *primary* node type. Service Fabric runtime services, such as the Cluster Management Service, run on the primary node type. Provision at least 5 nodes for the primary node type in a production environment. The other node type should have at least 2 nodes.

## Configuring and managing the cluster

Clusters must be secured to prevent unauthorized users from connecting to your cluster. It is recommended to use Azure AD to authenticate clients, and X.509 certificates for node-to-node security. For more information, see [Service Fabric cluster security scenarios](#).

To configure a public HTTPS endpoint, see [Specify resources in a service manifest](#).

You can scale out the application by adding VMs to the cluster. VM scale sets support auto-scaling using auto-scale rules based on performance counters. For more information, see [Scale a Service Fabric cluster in or out using auto-scale rules](#).

While the cluster is running, you should collect logs from all the nodes in a central location. For more information, see [Collect logs by using Azure Diagnostics](#).

## Conclusion

Porting the Surveys application to Service Fabric was fairly straightforward. To summarize, we did the following:

- Converted the roles to stateless services.
- Converted the web front ends to ASP.NET Core.
- Changed the packaging and configuration files to the Service Fabric model.

In addition, the deployment changed from Cloud Services to a Service Fabric cluster running in a VM Scale Set.

## Next steps

Now that the Surveys application has been successfully ported, Tailspin wants to take advantage of Service Fabric features such as independent service deployment and versioning. Learn how Tailspin decomposed these services to a more granular architecture to take advantage of these Service Fabric features in [Refactor an Azure Service Fabric Application migrated from Azure Cloud Services](#)

# Refactor an Azure Service Fabric Application migrated from Azure Cloud Services

3/6/2018 • 9 minutes to read • [Edit Online](#)

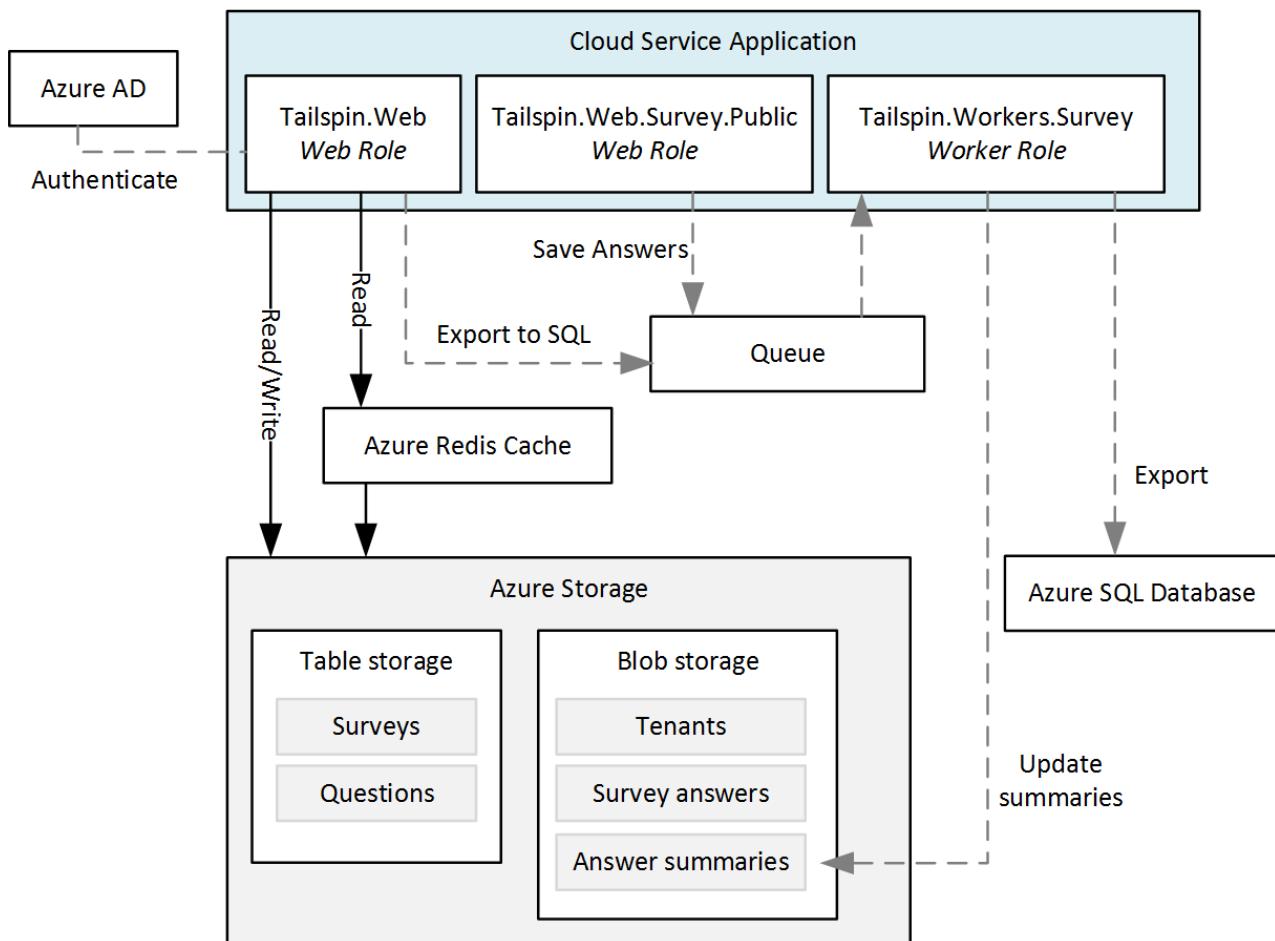


[Sample code](#)

This article describes refactoring an existing Azure Service Fabric application to a more granular architecture. This article focuses on the design, packaging, performance, and deployment considerations of the refactored Service Fabric application.

## Scenario

As discussed in the previous article, [Migrating an Azure Cloud Services application to Azure Service Fabric](#), the patterns & practices team authored a book in 2012 that documented the process for designing and implementing a Cloud Services application in Azure. The book describes a fictitious company named Tailspin that wants to create a Cloud Services application named **Surveys**. The Surveys application allows users to create and publish surveys that can be answered by the public. The following diagram shows the architecture of this version of the Surveys application:



The **Tailspin.Web** web role hosts an ASP.NET MVC site that Tailspin customers use to:

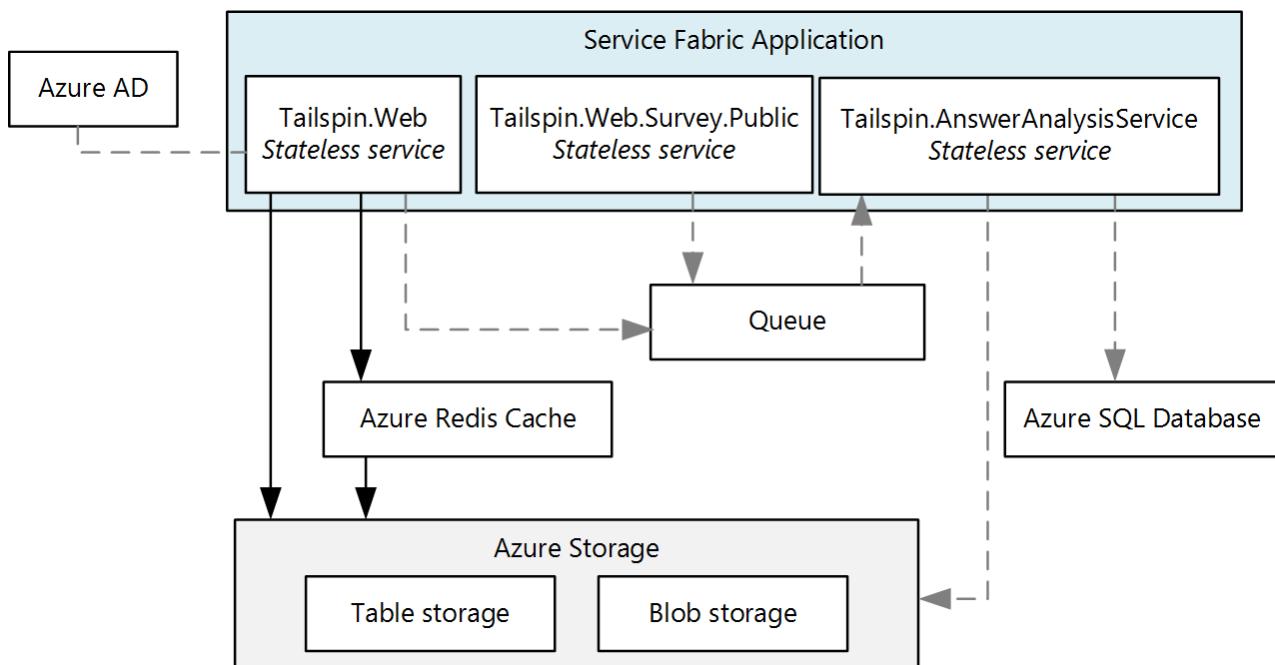
- sign up for the Surveys application,
- create or delete a single survey,
- view results for a single survey,

- request that survey results be exported to SQL, and
- view aggregated survey results and analysis.

The **Tailspin.Web.Survey.Public** web role also hosts an ASP.NET MVC site that the public visits to fill out the surveys. These responses are put in a queue to be saved.

The **Tailspin.Workers.Survey** worker role performs background processing by picking up requests from multiple queues.

The patterns & practices team then created a new project to port this application to Azure Service Fabric. The goal of this project was to make only the necessary code changes to get the application running in an Azure Service Fabric cluster. As a result, the original web and worker roles were not decomposed into a more granular architecture. The resulting architecture is very similar to the Cloud Service version of the application:



The **Tailspin.Web** service is ported from the original *Tailspin.Web* web role.

The **Tailspin.Web.Survey.Public** service is ported from the original *Tailspin.Web.Survey.Public* web role.

The **Tailspin.AnswerAnalysisService** service is ported from the original *Tailspin.Workers.Survey* worker role.

#### NOTE

While minimal code changes were made to each of the web and worker roles, **Tailspin.Web** and **Tailspin.Web.Survey.Public** were modified to self-host a [Kestrel](#) web server. The earlier Surveys application is an ASP.NET application that was hosted using Internet Information Services (IIS), but it is not possible to run IIS as a service in Service Fabric. Therefore, any web server must be capable of being self-hosted, such as [Kestrel](#). It is possible to run IIS in a container in Service Fabric in some situations. See [scenarios for using containers](#) for more information.

Now, Tailspin is refactoring the Surveys application to a more granular architecture. Tailspin's motivation for refactoring is to make it easier to develop, build, and deploy the Surveys application. By decomposing the existing web and worker roles to a more granular architecture, Tailspin wants to remove the existing tightly coupled communication and data dependencies between these roles.

Tailspin sees other benefits in moving the Surveys application to a more granular architecture:

- Each service can be packaged into independent projects with a scope small enough to be managed by a small team.
- Each service can be independently versioned and deployed.

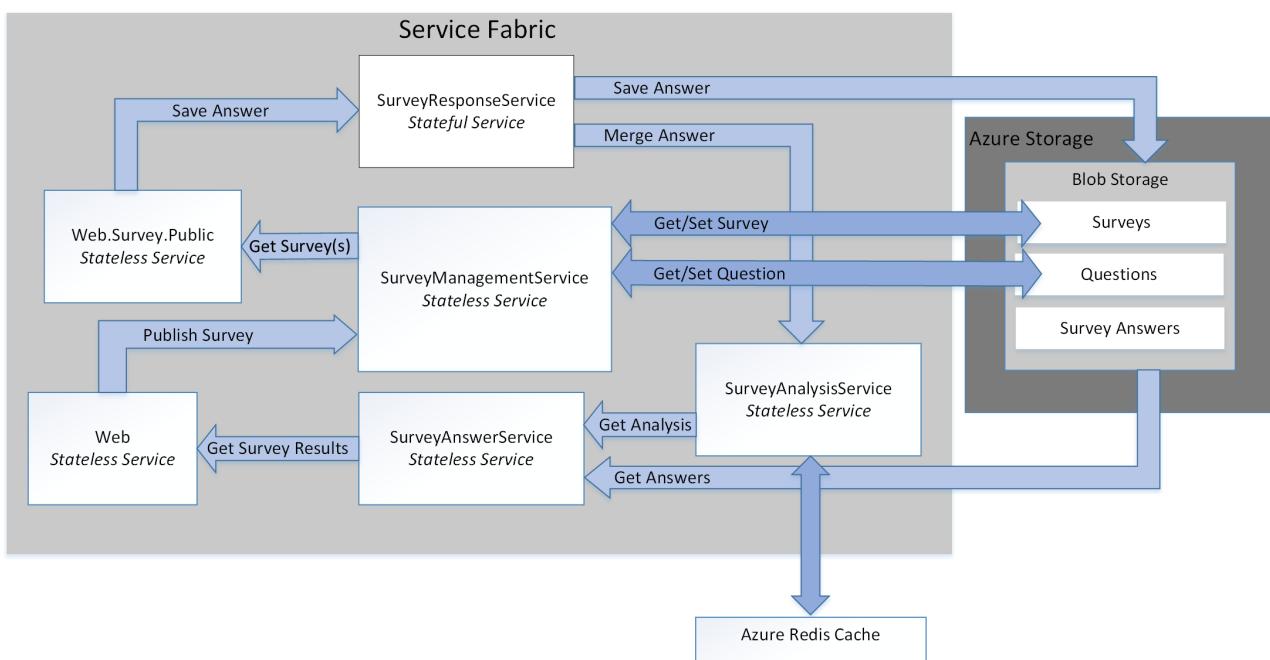
- Each service can be implemented using the best technology for that service. For example, a service fabric cluster can include services built using different versions of the .Net Frameworks, Java, or other languages such as C or C++.
- Each service can be independently scaled to respond to increases and decreases in load.

#### NOTE

Multitenancy is out of scope for the refactoring of this application. Tailspin has several options to support multitenancy and can make these design decisions later without affecting the initial design. For example, Tailspin can create separate instances of the services for each tenant within a cluster or create a separate cluster for each tenant.

## Design considerations

The following diagram shows the architecture of the Surveys application refactored to a more granular architecture:



**Tailspin.Web** is a stateless service self-hosting an ASP.NET MVC application that Tailspin customers visit to create surveys and view survey results. This service shares most of its code with the *Tailspin.Web* service from the ported Service Fabric application. As mentioned earlier, this service uses ASP.NET Core and switches from using Kestrel as web frontend to implementing a WebListener.

**Tailspin.Web.Survey.Public** is a stateless service also self-hosting an ASP.NET MVC site. Users visit this site to select surveys from a list and then fill them out. This service shares most of its code with the *Tailspin.Web.Survey.Public* service from the ported Service Fabric application. This service also uses ASP.NET Core and also switches from using Kestrel as web frontend to implementing a WebListener.

**Tailspin.SurveyResponseService** is a stateful service that stores survey answers in Azure Blob Storage. It also merges answers into the survey analysis data. The service is implemented as a stateful service because it uses a **ReliableConcurrentQueue** to process survey answers in batches. This functionality was originally implemented in the *Tailspin.AnswerAnalysisService* service in the ported Service Fabric application.

**Tailspin.SurveyManagementService** is a stateless service that stores and retrieves surveys and survey questions. The service uses Azure Blob storage. This functionality was also originally implemented in the data access components of the *Tailspin.Web* and *Tailspin.Web.Survey.Public* services in the ported Service Fabric application. Tailspin refactored the original functionality into this service to allow it to scale independently.

**Tailspin.SurveyAnswerService** is a stateless service that retrieves survey answers and survey analysis. The service also uses Azure Blob storage. This functionality was also originally implemented in the data access components of the *Tailspin.Web* service in the ported Service Fabric application. Tailspin refactored the original functionality into this service because it expects less load and wants to use fewer instances to conserve resources.

**Tailspin.SurveyAnalysisService** is a stateless service that persists survey answer summary data in a Redis cache for quick retrieval. This service is called by the *Tailspin.SurveyResponseService* each time a survey is answered and the new survey answer data is merged in the summary data. This service includes the functionality originally implemented in the *Tailspin.AnswerAnalysisService* service from the ported Service Fabric application.

## Stateless versus stateful services

Azure Service Fabric supports the following programming models:

- The guest executable model allows any executable to be packaged as a service and deployed to a Service Fabric cluster. Service Fabric orchestrates and manages execution of the guest executable.
- The container model allows for deployment of services in container images. Service Fabric supports creation and management of containers on top of Linux kernel containers as well as Windows Server containers.
- The reliable services programming model allows for the creation of stateless or stateful services that integrate with all Service Fabric platform features. Stateful services allow for replicated state to be stored in the Service Fabric cluster. Stateless services do not.
- The reliable actors programming model allows for the creation of services that implement the virtual actor pattern.

All the services in the Surveys application are stateless reliable services, except for the *Tailspin.SurveyResponseService* service. This service implements a [ReliableConcurrentQueue](#) to process survey answers when they are received. Responses in the ReliableConcurrentQueue are saved into Azure Blob Storage and passed to the *Tailspin.SurveyAnalysisService* for analysis. Tailspin chooses a ReliableConcurrentQueue because responses do not require strict first-in-first-out (FIFO) ordering provided by a queue such as Azure Service Bus. A ReliableConcurrentQueue is also designed to deliver high throughput and low latency for queue and dequeue operations.

Note that operations to persist dequeued items from a ReliableConcurrentQueue should ideally be idempotent. If an exception is thrown during the processing of an item from the queue, the same item may be processed more than once. In the Surveys application, the operation to merge survey answers to the *Tailspin.SurveyAnalysisService* is not idempotent because Tailspin decided that the survey analysis data is only a current snapshot of the analysis data and does not need to be consistent. The survey answers saved to Azure Blob Storage are eventually consistent, so the survey final analysis can always be recalculated correctly from this data.

## Communication framework

Each service in the Surveys application communicates using a RESTful web API. RESTful APIs offer the following benefits:

- Ease of use: each service is built using ASP.NET Core MVC, which natively supports the creation of Web APIs.
- Security: While each service does not require SSL, Tailspin could require each service to do so.
- Versioning: clients can be written and tested against a specific version of a web API.

Services in the Survey application make use of the [reverse proxy](#) implemented by Service Fabric. Reverse proxy is a service that runs on each node in the Service Fabric cluster and provides endpoint resolution, automatic retry, and handles other types of connection failures. To use the reverse proxy, each RESTful API call to a specific service is made using a predefined reverse proxy port. For example, if the reverse proxy port has been set to **19081**, a call to the *Tailspin.SurveyAnswerService* can be made as follows:

```
static SurveyAnswerService()
{
 httpClient = new HttpClient
 {
 BaseAddress = new Uri("http://localhost:19081/Tailspin/SurveyAnswerService/")
 };
}
```

To enable reverse proxy, specify a reverse proxy port during creation of the Service Fabric cluster. For more information, see [reverse proxy](#) in Azure Service Fabric.

## Performance considerations

Tailspin created the ASP.NET Core services for *Tailspin.Web* and *Tailspin.Web.Surveys.Public* using Visual Studio templates. By default, these templates include logging to the console. Logging to the console may be done during development and debugging, but all logging to the console should be removed when the application is deployed to production.

### NOTE

For more information about setting up monitoring and diagnostics for Service Fabric applications running in production, see [monitoring and diagnostics](#) for Azure Service Fabric.

For example, the following lines in *startup.cs* for each of the web front end services should be commented out:

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
 //loggerFactory.AddConsole(Configuration.GetSection("Logging"));
 //loggerFactory.AddDebug();

 app.UseMvc();
}
```

### NOTE

These lines may be conditionally excluded when Visual Studio is set to "release" when publishing.

Finally, when Tailspin deploys the Tailspin application to production, they switch Visual Studio to **release** mode.

## Deployment considerations

The refactored Surveys application is composed of five stateless services and one stateful service, so cluster planning is limited to determining the correct VM size and number of nodes. In the *applicationmanifest.xml* file that describes the cluster, Tailspin sets the *InstanceCount* attribute of the *StatelessService* tag to -1 for each of the services. A value of -1 directs Service Fabric to create an instance of the service on each node in the cluster.

### NOTE

Stateful services require the additional step of planning the correct number of partitions and replicas for their data.

Tailspin deploys the cluster using the Azure Portal. The Service Fabric Cluster resource type deploys all of the necessary infrastructure, including VM scale sets and a load balancer. The recommended VM sizes are displayed in

the Azure portal during the provisioning process for the Service Fabric cluster. Note that because the VMs are deployed in a VM scale set, they can be both scaled up and out as user load increases.

**NOTE**

As discussed earlier, in the migrated version of the Surveys application the two web front ends were self-hosted using ASP.NET Core and Kestrel as a web server. While the migrated version of the Survey application does not use a reverse proxy, it is strongly recommended to use a reverse proxy such as IIS, Nginx, or Apache. For more information see [introduction to Kestrel web server implementation in ASP.NET core](#). In the refactored Surveys application, the two web front ends are self-hosted using ASP.NET Core with [WebListener](#) as a web server so a reverse proxy is not necessary.

## Next steps

The Surveys application code is available on [GitHub](#).

If you are just getting started with [Azure Service Fabric](#), first set up your development environment then download the latest [Azure SDK](#) and the [Azure Service Fabric SDK](#). The SDK includes the OneBox cluster manager so you can deploy and test the Surveys application locally with full F5 debugging.

# Extend Azure Resource Manager template functionality

3/16/2018 • 2 minutes to read • [Edit Online](#)

In 2016, the Microsoft patterns & practices team created a set of Azure Resource Manager [template building blocks](#) with the goal of simplifying resource deployment. Each building block contains a set of pre-built templates that deploy sets of resources specified by separate parameter files.

The building block templates are designed to be combined together to create larger and more complex deployments. For example, deploying a virtual machine in Azure requires a virtual network, storage accounts, and other resources. The [virtual network building block template](#) deploys a virtual network and subnets. The [virtual machine building block template](#) deploys storage accounts, network interfaces, and the actual VMs. You can then create a script or template to call both building block templates with their corresponding parameter files to deploy a complete architecture with one operation.

While developing the building block templates, p&p designed several concepts to extend Azure Resource Manager template functionality. In this series, we will describe several of these concepts so you can use them in your own templates.

## NOTE

These articles assume you have an advanced understanding of Azure Resource Manager templates.

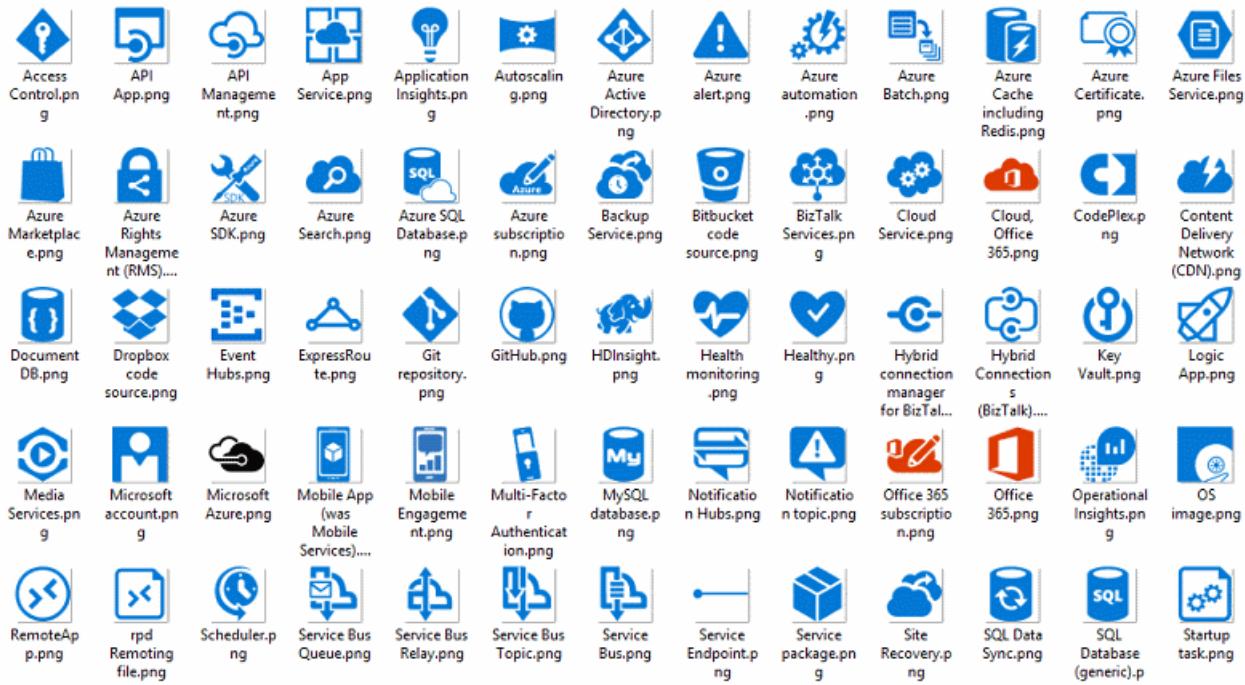
# Icons and diagrams

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

These resources include icons, Visio templates, PNG files, and SVG files that are useful for producing your own architecture diagrams.

## Microsoft Azure, Cloud and Enterprise Symbol/Icon Set

The symbol/icon set is a collection of Visio, PowerPoint, PNG, and SVG assets that you can use to produce custom technical content. [View the training video](#) and [download symbol/icon set](#).



Additional symbols for Microsoft Office and related technologies are available in the [Microsoft Office Visio stencil](#). They are not optimized for architectural diagrams.

### NOTE

These assets are not intended for use in user interfaces. Third-party symbols are not owned by Microsoft. Please contact the symbols team at [CnESymbols@microsoft.com](mailto:CnESymbols@microsoft.com) with comments, feedback, or questions about usage.

## Reference Architectures Visio template

A version of the diagrams used in the [Reference Architectures](#) is available for [download in Visio format](#).

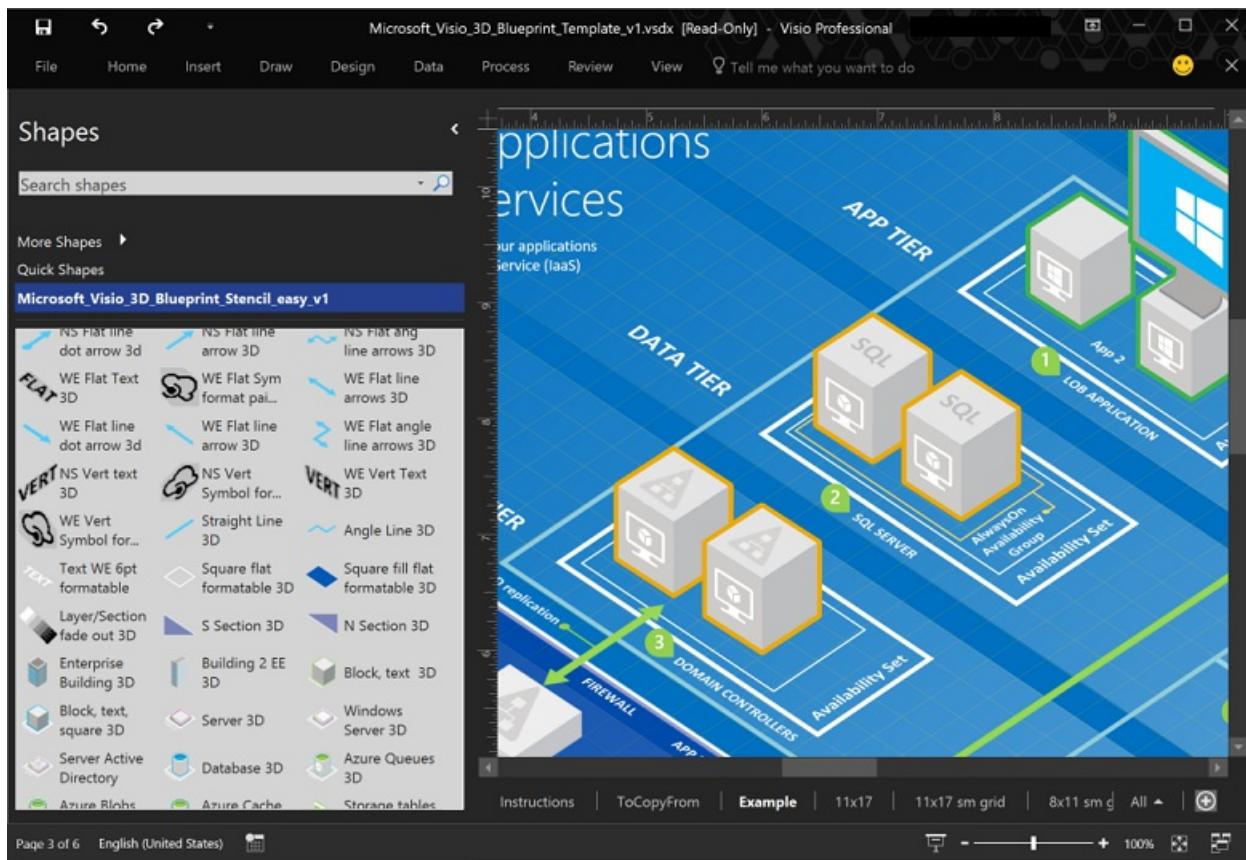
## Solution Architecture Diagrams

Microsoft publishes [solution architectures and accompanying diagrams](#). The diagrams are in downloadable in SVG format. The SVG can be opened and then modified by many tools, including Visio and PowerPoint. If you ungroup the diagram, you can select the individual icons.

## 3D Blueprint Visio template

A Visio template is available for producing 3D (isometric) architectural diagrams.

- [View the training video](#)
- [Download the Microsoft 3D Blueprint Visio Template](#)



#### NOTE

This template is no longer under active development.