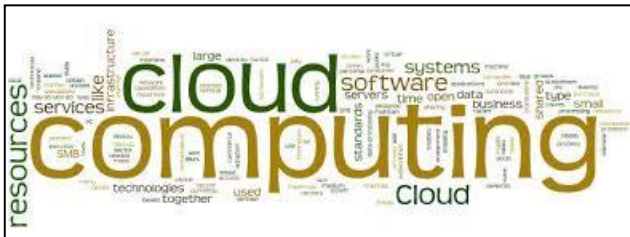


Part 1 – Introduction and Design Patterns

Gianluca Reali

Università degli Studi di Perugia

Slide objectives

When people talk about to “the cloud” they are in fact referring to data stored in massive datacenters around the world, which are each filled with many thousands of servers.

Main messaging

The cloud is not an abstract concept, it is a physical space with physical boundaries, requirements and limitations

These physical resources dedicated to the Azure cloud are commonly referred to as the “Azure Fabric” on which all the Azure cloud services run

Characteristics of cloud computing

- Network access to cloud services
- Pay only what you need from a measured service
- Multi-tenancy – many customers in same space
- On-demand self-service to scalable resources
- High bandwidth links to and between datacenters

Why the Cloud? Why not the Cloud?

If you use a (public) cloud:

- If you don't have a huge quantity of users, data, or processing, then your **costs** are probably lower:
  - you won't have to pay for as many system administrators
  - you won't have to buy big servers
  - you don't have to manage big networks
- You can have **scalability**, that is, if you have a sudden increase in users then the cloud infrastructure can handle it (of course, at a price).
  - to be able to do this yourself you would have to pay for and maintain extra infrastructure in terms of servers, networks, and system administrators
- Your data may be backed up at a different location, such that it can **survive** a large natural disaster. Or possibly even a war.

Why the Cloud? Why not the Cloud? (cont'd)

There are some security advantages:

- The cloud provider has a big incentive to keep all its **software up to date** with security patches
  - whereas you would have to have a good staff of system administrators, as well as regular update procedures, to make sure this is done in your own company
- The cloud most likely provides some sort of **physical security** for its servers
  - if you do the same in your company, this requires locked rooms with access control, and additional procedures that must be followed. For example, who gets a key? If someone is fired, do you change all the locks? Etc.
- The cloud provider has a big incentive to hire **good system administrators** who have been background-checked

Why the Cloud? Why not the Cloud? (cont'd)

What are some **negatives** for having your company's software on a cloud?

- Your particular application may not benefit from added scalability, particularly if it is a traditional **monolithic application** architecture for which a stable demand is expected
- It may be **difficult to organize** or re-write your application such that it would run well when located on a cloud.
  - There is some literature that says that an application must have a Service Oriented Architecture in order to be able to be hosted on a cloud
- Even if it is possible to move your application to the cloud such that its performance is good, there will probably be some cost involved in making this happen. Perhaps significant cost.

Why the Cloud? Why not the Cloud? (cont'd)

There may be security disadvantages associated with being on a cloud:

- The cloud provider has a big incentive to have good security, however, if you onboard your software to a cloud, **you give up control** of its security

There can be **data policy or privacy disadvantages** associated with being on a cloud:

- your data would be managed by the cloud provider, not by you.

7

there are cases where by law or by privacy agreement, an individual wishes to delete his/her private information. How can you ensure the cloud provider actually does this?

What is your cloud provider's policy on notifying you if there has been a data breach? How long will this take?—see Winkler (2012)

What is your responsibility for notifying the cloud provider if your data has been breached? Do you have any liability?

Why the Cloud? Why not the Cloud? (cont'd)

- in large cloud companies with data centers located in other countries (maybe not that friendly to your own country), then **physical location of the data** can be a big issue
- how do you know the cloud provider is not mining your private data for its own purposes and perhaps selling this information to a third party?
  - What is your company's **liability** if the cloud provider does this wrong?

General Features

This cloud model includes some essential **Characteristics, Service Models**, and **Deployment models**.

Characteristics

- On-demand, self-service
- Broad network access
- Resource pooling
  - Location independence
- Rapid elasticity
- Measured service

Service Models

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)
- Function as a Service (FaaS) - Serverless Computing
- … a lot of further proposals

Deployment Models

- *Private Cloud*
- *Public Cloud*
- *Community Cloud*
- *Hybrid Cloud*

9

At one end of the spectrum is infrastructure as a service (IaaS). With IaaS, you run the provisioning of the required virtual machines, as well as associated network and storage components. Next, you deploy the desired software and applications to virtual machines. This model is most similar to a traditional on-premises environment, except that the infrastructure is managed from the cloud providers. You can still manage individual virtual machines. The Platform-as-a-Service (PaaS) model offers a hosting environment managed, where you can deploy your application without needing to manage VMs or resources net. For example, instead of creating individual virtual machines, you specify a number of instances and the service will provision, configure and manage the necessary resources. The Functions-as-a-Service (FaaS) model goes even further than that eliminating the need to manage your hosting environment. Instead create compute instances and deploy code to your instances, you simply deploy the code and the service will run it automatically. It is not need to manage computing resources. These services use serverless architecture and augment or automatically reduce performance to the level needed to handle the traffic. IaaS offers the ultimate in control, flexibility, and portability. FaaS offers simplicity, elastic scalability and potentially cost savings, as you only pay based on code execution times. PaaS is a intermediate model. In general, the more flexibility a service offers, the more accountable you are at first resource management and configuration person. FaaS services automatically handle almost all of them aspects of running an application, while IaaS solutions involve provisioning, the autonomous configuration and management of created virtual machines and network components.

Software as a Service (SaaS)

Cloud-native applications are those software developed as software-as-a-service (Saas), or that application distribution model where a software manufacturer develops and makes available, therefore also managing the service, a Web App that can be granted free of charge or through a service subscription.

The **capability provided to the consumer is to use the provider's applications** running on a cloud infrastructure and accessible from various client devices through a thin client interface such as a Web browser (e.g., web-based email). The consumer does not manage or control the underlying cloud infrastructure, network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

Platform as a Service (PaaS)

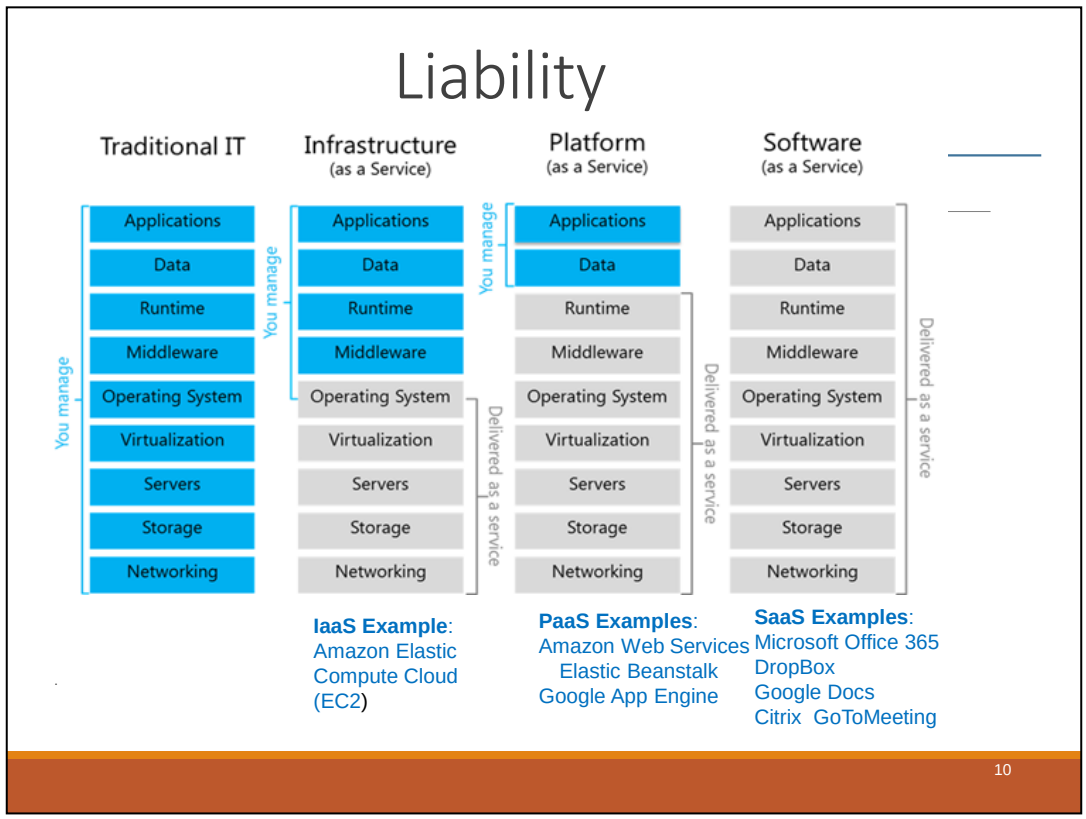
The **capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created applications using programming languages and tools supported by the provider** (e.g., Java, Python, .Net). The consumer does not manage or control the underlying cloud infrastructure, network, servers, operating systems, or storage, but the consumer has control over the deployed applications and possibly application hosting environment configurations.

Infrastructure as a Service (IaaS)

The **capability provided to the consumer is to rent processing, storage, networks, and other fundamental computing resources** where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly select networking components (e.g., firewalls, load balancers).

Function as a Service (FaaS)

FaaS is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities **without the complexity of building and maintaining the infrastructure** typically associated with developing and launching an app. Building an application following this model is one way of achieving a "serverless" architecture, and is typically used when building microservices applications.



Runtime describes software/instructions that are executed *while* your program is running, especially those instructions that you did not write explicitly, but are necessary for the proper execution of your code.

Low-level languages like C have very small (if any) runtime. More complex languages like Objective-C, which allows for dynamic message passing, have a much more extensive runtime.

You are correct that runtime code is library code, but library code is a more general term, describing the code produced by *any* library. Runtime code is specifically the code required to implement the features of the language itself.

Most languages have some form of runtime system that provides an environment in which programs run. This environment may address a number of issues including the layout of application memory, how the program accesses variables, mechanisms for passing parameters between procedures, interfacing with the operating system, and otherwise. The compiler makes assumptions depending on the specific runtime system to generate correct code.

IaaSdelivers computer infrastructure

PaaS deliver a computing platform where the developers can develop their own applications.

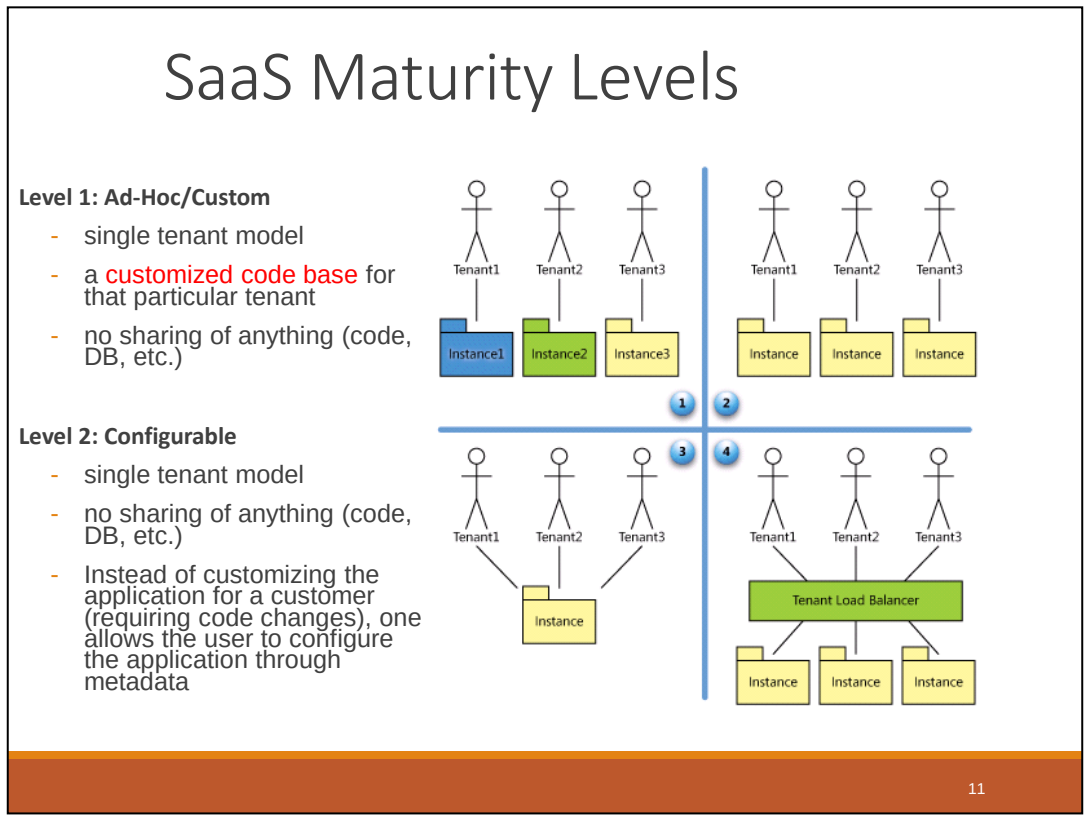
SaaS is a model of software deployment where the software applications are provided to the customers as a service.

Sometimes a SaaS offering can be run on top of some other cloud provider's IaaS or PaaS offering

> For example, DropBox originally ran its offerings on top of the Amazon cloud –see Metz (2016)

Microsoft Azure allows both IaaS and PaaS operation

Salesforce.com offers Force.com, which allows external developers to create add on applications that integrate with the main salesforce.com offerings



Single Tenant

In a single tenant model each of the tenants will get their own respective instances. There is absolutely no sharing of anything (code, DB, etc.). Each time a new customer (tenant) is added a new (logical) hardware is provisioned and new instance of the product is setup in the allocated environment. This could also be a logical separation of the hardware in the form of separate web sites within the same IIS server. Single tenant model is also commonly referred as Hosted model.

Maturity Level 1 and 2 falls under single tenant model. In both the levels the tenant gets their own instance of the software. However, the primary difference between these levels is that in Level 1 you may even have a customized code base for that particular tenant. Therefore, you may even end up having different code bases for each tenant (as depicted with multiple colors in the above picture). However, from a Tenant's perspective it's immaterial whether it's Level 1 or 2 as long as the software performs the expected functions.

Advantages

Takes less time to roll-out in to SaaS model, since the product does not require going through any changes to support SaaS model.

Overall SaaS transition complexity and cost is going to be less.

Does not require any SaaS Architecture/Engineering expertise. All that is required would be expertise on the hardware front, which is any addressed by IaaS vendors.

Level 1 offers a unique advantage of the ability to provide customized versions to your client. However, this could also be viewed as a disadvantage as you will have to keep maintaining several versions of your product.

Supports non-web native applications. For example, you can use windows citrix to deliver a desktop or client/server application (ex: applications developed in VB) in a SaaS model.

Certain market/customer base does not like the idea of multi-tenancy. For example: a banking customer will not like the idea of sharing their data/environment with other banking organizations. In some cases the security compliance also does not allow sharing. In these instances Single Tenant wins over multi-tenant.

Disadvantages

Maintenance efforts are going to be huge as you will have to maintain multiple code base/environments. For example, if you are going to make a fix then you will have to roll-it out 'n' number of times, where n is the number of tenants supported.

Operational costs are going to be extremely high, particularly over the long run.

Multi-Tenant

Multi-Tenancy is an architecture capability that allows an application/product to recognize tenants (tenants could be users, group of users or organizations) and exhibit functionalities as per the configuration set for the tenants. An on-premise application is typically designed to work for a single organization. Therefore, the very concept of making an on-premise application realize/operate in the context of a particular tenant is a big change. Every single functionality in the product has to be tweaked in order to achieve multi-tenancy. Segregation at the data layer is another big challenge. There are various degrees of multi-tenancy that can be supported. The complexity/effort also depends on the multi-tenancy degree that you are choosing to go with.

Maturity Level 3 and 4 fall under the multi-tenant model. Level 3 and 4 differ only in the level of scalability they can offer. Level 3 is more of a scale-up solution where you can upgrade the hardware to increase the number of tenants that can be supported. Level 4 supports scale-out solution where multiple hardwares can be added so that a load balanced environment can be created. As you can imagine, the decision between Level 3 and Level 4 lies with the amount of tenants that you are expecting to have on the system.

Advantages

Facilitates a cost effective way of delivering SaaS solution.

Huge savings in operational cost, particularly over the long run. This in turn will allow the ISVs to offer an attractive pricing to their customers, therefore creating a snow ball effect.

Enables to achieve higher levels of customer satisfaction.

Roll-out of upgrades/fixes is much easier.

Easier to adopt/implement best practices in the product features as you will be concentrating on only one version of the product.

The design principles of a multi-tenant system offer a high level of maintainability. For example, if a customer requests for few additional fields in one of the pages you can easily add them through custom field’s module.

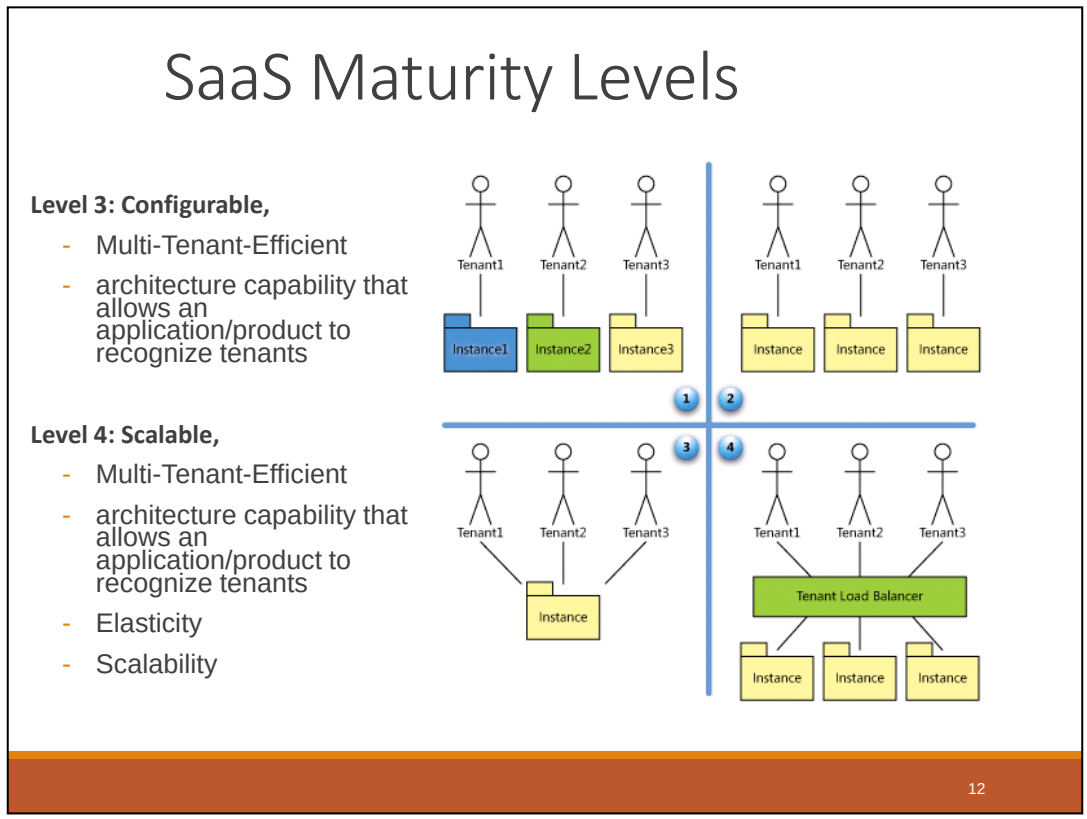
Scalability to expand the number of tenants. For example: Salesforce.com has more than 100,000 tenants running on their multi-tenant system. Can you even imagine how this will work in a single tenant model?

Disadvantages

Due to the amount of changes that has to happen in the product it takes a while to roll-out the solution.

Initial investment to enable multi-tenancy is high. While you will recover this cost in the form of savings in operational expenses it still requires that initial investment and as well the risk involved in achieving the expected business growth.

Requires SaaS architecture expertise, which may not be within the company. Will have to find a strong SaaS partner to guide you in the transition process.



Single Tenant

In a single tenant model each of the tenants will get their own respective instances. There is absolutely no sharing of anything (code, DB, etc.). Each time a new customer (tenant) is added a new (logical) hardware is provisioned and new instance of the product is setup in the allocated environment. This could also be a logical separation of the hardware in the form of separate web sites within the same IIS server. Single tenant model is also commonly referred as Hosted model.

Maturity Level 1 and 2 falls under single tenant model. In both the levels the tenant gets their own instance of the software. However, the primary difference between these levels is that in Level 1 you may even have a customized code base for that particular tenant. Therefore, you may even end up having different code bases for each tenant (as depicted with multiple colors in the above picture). However, from a Tenant's perspective it's immaterial whether it's Level 1 or 2 as long as the software performs the expected functions.

Advantages

Takes less time to roll-out in to SaaS model, since the product does not require going through any changes to support SaaS model.

Overall SaaS transition complexity and cost is going to be less.

Does not require any SaaS Architecture/Engineering expertise. All that is required would be expertise on the hardware front, which is any addressed by IaaS vendors.

Level 1 offers a unique advantage of the ability to provide customized versions to your client. However, this could also be viewed as a disadvantage as you will have to keep maintaining several versions of your product.

Supports non-web native applications. For example, you can use windows citrix to deliver a desktop or client/server application (ex: applications developed in VB) in a SaaS model.

Certain market/customer base does not like the idea of multi-tenancy. For example: a banking customer will not like the idea of sharing their data/environment with other banking organizations. In some cases the security compliance also does not allow sharing. In these instances Single Tenant wins over multi-tenant.

Disadvantages

Maintenance efforts are going to be huge as you will have to maintain multiple code base/environments. For example, if you are going to make a fix then you will have to roll-it out 'n' number of times, where n is the number of tenants supported.

Operational costs are going to be extremely high, particularly over the long run.

Multi-Tenant

Multi-Tenancy is an architecture capability that allows an application/product to recognize tenants (tenants could be users, group of users or organizations) and exhibit functionalities as per the configuration set for the tenants. An on-premise application is typically designed to work for a single organization. Therefore, the very concept of making an on-premise application realize/operate in the context of a particular tenant is a big change. Every single functionality in the product has to be tweaked in order to achieve multi-tenancy. Segregation at the data layer is another big challenge. There are various degrees of multi-tenancy that can be supported. The complexity/effort also depends on the multi-tenancy degree that you are choosing to go with.

Maturity Level 3 and 4 fall under the multi-tenant model. Level 3 and 4 differ only in the level of scalability they can offer. Level 3 is more of a scale-up solution where you can upgrade the hardware to increase the number of tenants that can be supported. Level 4 supports scale-out solution where multiple hardwares can be added so that a load balanced environment can be created. As you can imagine, the decision between Level 3 and Level 4 lies with the amount of tenants that you are expecting to have on the system.

Advantages

Facilitates a cost effective way of delivering SaaS solution.

Huge savings in operational cost, particularly over the long run. This in turn will allow the ISVs to offer an attractive pricing to their customers, therefore creating a snow ball effect.

Enables to achieve higher levels of customer satisfaction.

Roll-out of upgrades/fixes is much easier.

Easier to adopt/implement best practices in the product features as you will be concentrating on only one version of the product.

The design principles of a multi-tenant system offer a high level of maintainability. For example, if a customer requests for few additional fields in one of the pages you can easily add them through custom field's module.

Scalability to expand the number of tenants. For example: Salesforce.com has more than 100,000 tenants running on their multi-tenant system. Can you even imagine how this will work in a single tenant model?

Disadvantages

Due to the amount of changes that has to happen in the product it takes a while to roll-out the solution.

Initial investment to enable multi-tenancy is high. While you will recover this cost in the form of savings in operational expenses it still requires that initial investment and as well the risk involved in achieving the expected business growth.

Requires SaaS architecture expertise, which may not be within the company. Will have to find a strong SaaS partner to guide you in the transition process.

Deployment Models

There are four primary cloud deployment models:

- Public Cloud
- Private Cloud
- Community Cloud
- Hybrid Cloud

Their differences lie primarily in the scope and access of published cloud services, as they are made available to service consumers.

Private cloud: The cloud infrastructure is operated solely for an organization. It may exist on premise or off premise.

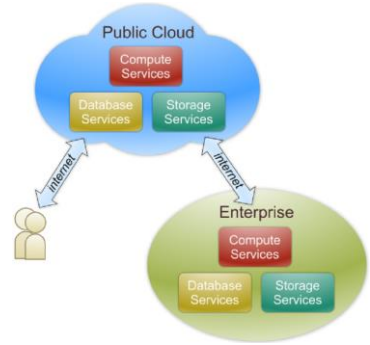
Public cloud: Mega-scale cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

Hybrid cloud: Composition of two or more clouds (private or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability.

Public Cloud

Public cloud definition

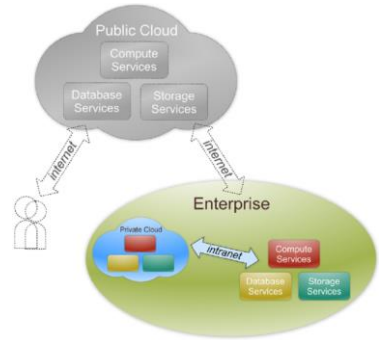
- The cloud infrastructure is made available to the **general public** or a **large industry group** and is owned by an organization selling cloud services.
- Also known as external cloud or multi-tenant cloud, this model essentially represents a cloud environment that is openly accessible.
- Basic characteristics:
  - Homogeneous infrastructure
  - Common policies
  - Shared resources and multi-tenant
  - Leased or rented infrastructure
  - Economies of scale



Private Cloud

Private cloud definition

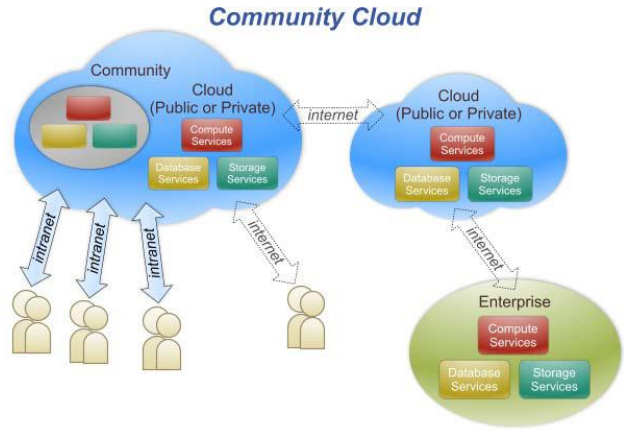
- The cloud infrastructure is **operated solely for single organization**. It may be managed by the organization or a third party and may exist on premise or off premise.
- Also referred to as internal cloud or on-premise cloud, a private cloud intentionally limits access to its resources to service consumers that belong to the same organization that owns the cloud.
- Basic characteristics :
  - Heterogeneous infrastructure
  - Customized and tailored policies
  - Dedicated resources
  - In-house infrastructure
  - End-to-end control



Community Cloud

Community cloud definition

- The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations).



17

They are multi-tenant platforms that enable different organizations to work on a shared platform.

A community cloud usually involves similar industries and complimentary businesses with shared goals all using the same hardware. By sharing the infrastructure between multiple companies, community cloud installations are able to save their members money. Data is still segmented and kept private, except in areas where shared access is agreed upon and configured.

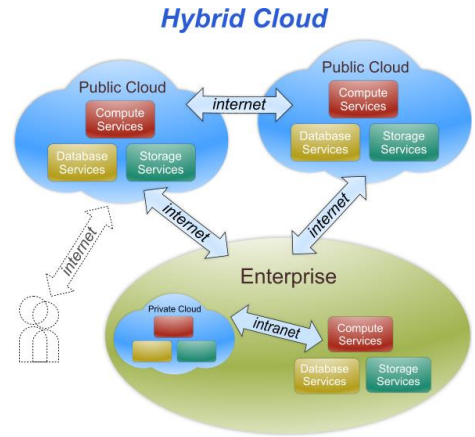
The main benefits are the shared costs and the increase in opportunities to collaborate in real-time across the same infrastructure. Uniformity of best practices will help to increase the overall security and efficiency of these setups, so they rely quite heavily on effective cooperation between tenants.

A good example is the **U.S.-based dedicated IBM SoftLayer cloud for federal agencies**.

Hybrid Cloud

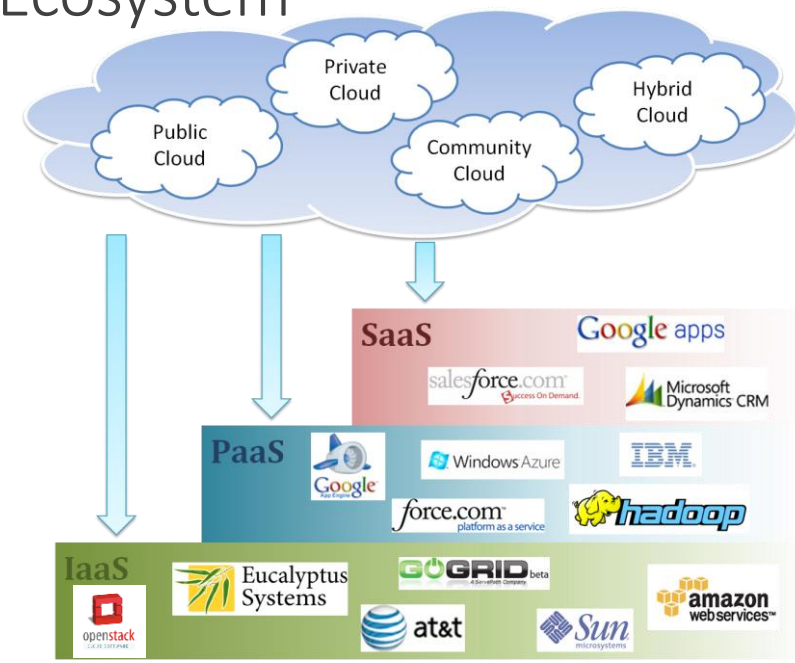
Hybrid cloud definition

- The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds).



meh

Cloud Ecosystem



Open Source Platforms Supporting IaaS

Cloud	History	License	Language Written in	Installaiton Difficulty
OpenStack	Founded by Rackspace and NASA in 2010	Apache v2.0	Python	difficult
CloudStack	Released by Citrix into Apache in 2011	Apache v2.0	Java,C	Medium
Eucalyptus	Began at Rice University and UCSB, now sponsored by HP. Has formal compatibility agreement with AWS	GPL v 3.0	Java,C	Medium
OpenNebula	Sponsored by OpenNebula Systems	Apache v2.0	C++, C, Ruby, Java, shell script, lex, yacc	Medium

Security Issues

Evergreen…

- Data Loss
- Downtimes
- Phishing
- Password Cracking
- Botnets and Other Malware

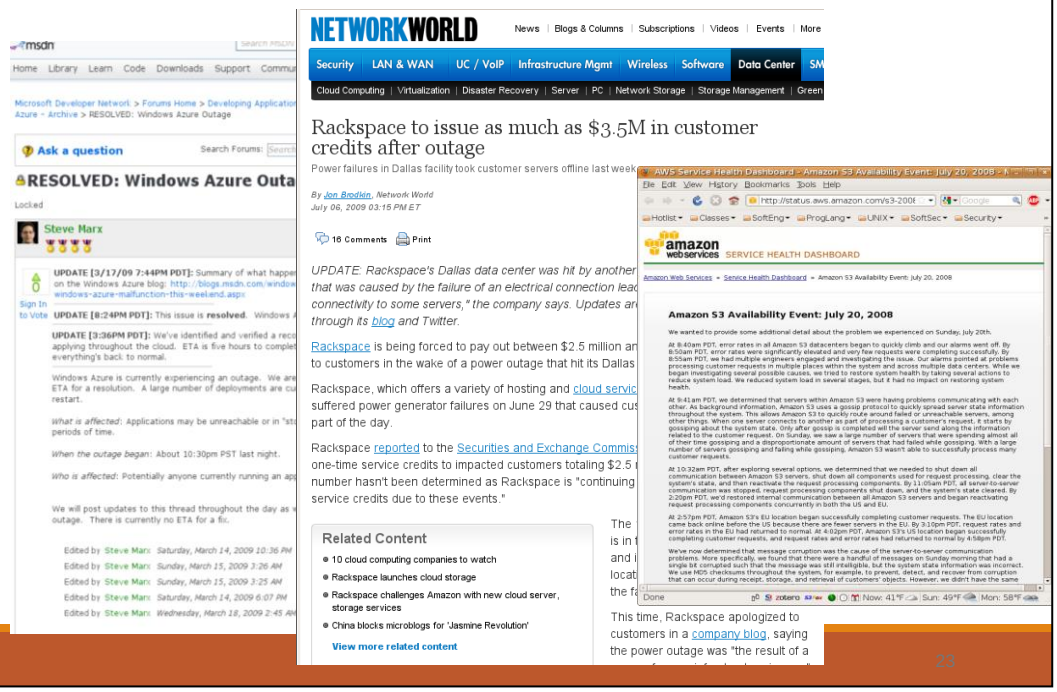
Data Loss

"Regrettably, based on Microsoft/Danger's latest recovery assessment of their systems, we must now inform you that personal information stored on your device—such as contacts, calendar entries, to-do lists or photos—that is no longer on your Sidekick almost certainly has been lost as a result of a server failure at Microsoft/Danger."

http://arstechnica.com/business/news/2009/10/t-mobile-microsoftdanger-data-loss-is-bad-for-the-cloud.ars

http://arstechnica.com/business/news/2009/10/t-mobile-microsoftdanger-data-loss-is-bad-for-the-cloud.ars

Downtimes


msdn
Home Library Learn Code Downloads Support
Ask a question
RESOLVED: Windows Azure Outa
Steve Marx
NETWORKWORLD
News | Blogs & Columns | Subscriptions | Videos | Events | More
Security LAN & WAN UC / VoIP Infrastructure Mgmt Wireless Software Data Center
Cloud Computing | Virtualization | Disaster Recovery | Server | PC | Network Storage | Storage Management | Green
Rackspace to issue as much as $3.5M in customer credits after outage
Power failures in Dallas facility took customer servers offline last week
By Jon Brodkin, Network World
July 06, 2009 03:15 PM ET
16 Comments Print
Related Content
10 cloud computing companies to watch
Rackspace launches cloud storage
Rackspace challenges Amazon with new cloud server, storage services
China blocks microblogs for 'Jasmine Revolution'
View more related content
AWS Service Health Dashboard - Amazon S3 Availability Event: July 20, 2008
amazon webservices SERVICE HEALTH DASHBOARD
Amazon S3 Availability Event: July 20, 2008

Phishing

"hey! check out this funny blog about you..."

http://news.cnet.com/twitter-phishing-scam-may-be-spreading/

Password Cracking

PCWorld | Business Center

Software & Services | Office Hardware | Security | Servers & Storage | Cell Phones & Mobile

Security Alert

BUSINESS CENTER

Cloud Computing Used to Hack Wireless Passwords

German security researcher Thomas Roth has found an innovative use for cloud computing: cracking wireless networks that rely on pre-shared key passphrases, such as those found in homes and smaller businesses.

SIMILAR ARTICLES:

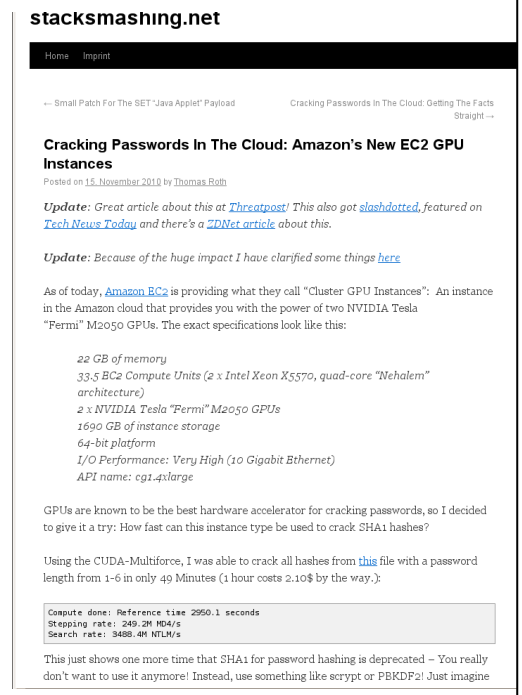
- Gawker Hack Exposes Ridiculous Password Habits
- What Cloud Computing Means For the Real World
- Can 'Encrypted Blobs' Help With Secure Cloud Computing?
- Virtualization is Key to Cloud Security
- 7 Ways to Avoid Getting Hacked by Anonymous
- Password Reuse Is All Too Common, Research Shows

Roth has created a program that runs on Amazon's Elastic Cloud Computing (EC2) system. It uses the massive computing power of EC2 to run through 400,000 possible passwords per second, a staggering amount, hitherto unheard of outside supercomputing circles--and very likely made possible because EC2 now allows graphics processing units (GPUs) to be used for computational tasks. Among other things, these are particularly suited to password cracking tasks.

In other words, this isn't a clever or elegant hack, and it doesn't rely on a flaw in wireless networking technology. Roth's software merely generates millions of passphrases, encrypts them, and sees if they allow access to the network.

However, employing the theoretically infinite resources of cloud computing to brute force a password is the clever part.

Purchasing the computers to run such a crack would cost tens of thousands of dollars, but Roth claims that a typical wireless password can be guessed by EC2 and his software in about six minutes. He proved this by hacking networks in the area where he lives. The type of EC2 computers used in the attack costs 28 cents per minute, so $1.68 is all it could take to lay open a wireless network.



stacksmashing.net

Home | Imprint

← Small Patch For The SET "Java Applet" Payload

Cracking Passwords In The Cloud: Getting The Facts Straight →

Cracking Passwords In The Cloud: Amazon's New EC2 GPU Instances

Posted on 15. November 2010 by Thomas Roth

***Update**: Great article about this at Threatpost! This also got slashdotted, featured on Tech News Today and there's a ZDNet article about this.*

***Update**: Because of the huge impact I have clarified some things here*

As of today, Amazon EC2 is providing what they call "Cluster GPU Instances": An instance in the Amazon cloud that provides you with the power of two NVIDIA Tesla "Fermi" M2050 GPUs. The exact specifications look like this:

> *22 GB of memory*
> *33.5 EC2 Compute Units (2 x Intel Xeon X5570, quad-core "Nehalem" architecture)*
> *2 x NVIDIA Tesla "Fermi" M2050 GPUs*
> *1690 GB of instance storage*
> *64-bit platform*
> *I/O Performance: Very High (10 Gigabit Ethernet)*
> *API name: cg1.4xlarge*

GPUs are known to be the best hardware accelerator for cracking passwords, so I decided to give it a try: How fast can this instance type be used to crack SHA1 hashes?

Using the CUDA-Multiforce, I was able to crack all hashes from this file with a password length from 1-6 in only 49 Minutes (1 hour costs 2.10$ by the way.):

```
Compute done: Reference time 2950.1 seconds
Stepping rate: 249.2M MD4/s
Search rate: 3488.4M NTLM/s
```

This just shows one more time that SHA1 for password hashing is deprecated – You really don't want to use it anymore! Instead, use something like scrypt or PBKDF2! Just imagine

Botnets and Malware

http://community.ca.com/blogs/securityadvisor/archive/2009/12/09/zeus-in-the-cloud.aspx

Virtualization Security

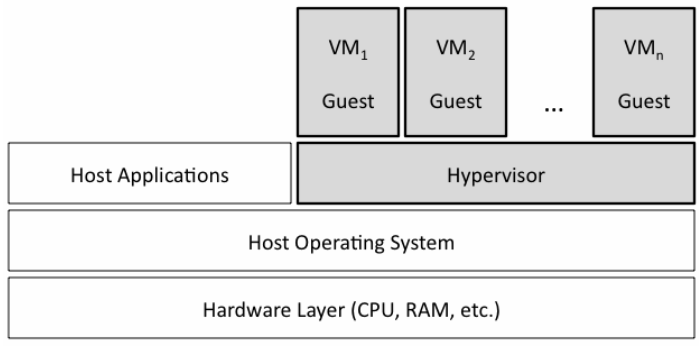
Features
- Isolation
- Snapshots

Issues
- State Restore
- Scaling
- Transience
- Data Lifetime

Features: Isolation

Using a VM for each application provides isolation

- Better isolation than running 2 apps on same server.
- Worse isolation than running on 2 physical servers



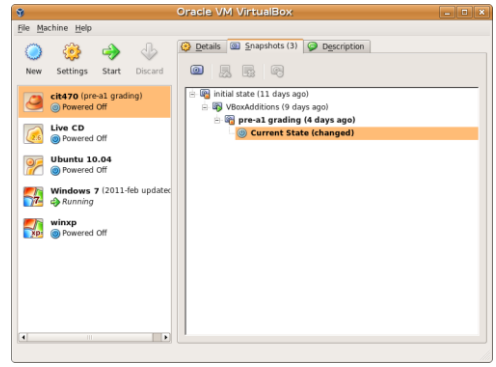
A Survey on the Security of Virtual Machines
http://www1.cse.wustl.edu/~jain/cse571-09/ftp/vmsec/index.html

http://www1.cse.wustl.edu/~jain/cse571-09/ftp/vmsec/index.html

Remember Spectre and Meltdown

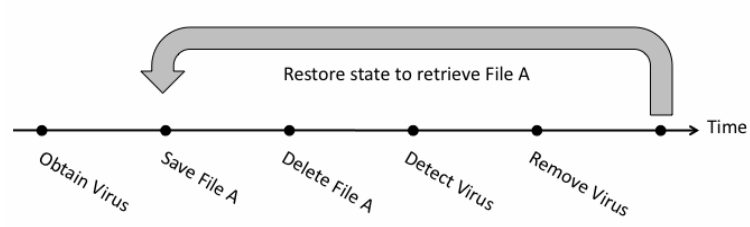
Features: Snapshot

- VMs can record state.
- In event of security incident, revert VM back to an uncompromised state.
- Must be sure to patch VM to avoid recurrence of compromise.



State Restore

- VMs can be restored to an infected or vulnerable state using snapshots.
- Patching becomes undone.
- Worms persist at low level forever due to reappearance of infected and vulnerable VMs.

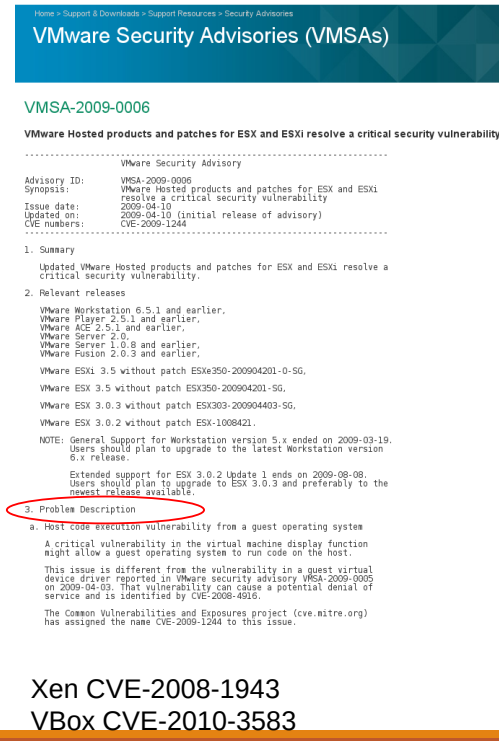


http://www1.cse.wustl.edu/~jain/cse571-09/ftp/vmsec/index.html

Hypervisor Security

Vulnerability consequences

- Guest code execution with privilege
- VM Escape (Host code execution)



Home > Support & Downloads > Support Resources > Security Advisories

VMware Security Advisories (VMSAs)

VMSA-2009-0006

VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability

VMware Security Advisory

Advisory ID: VMSA-2009-0006
Synopsis: VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability
Issue date: 2009-04-10
Updated on: 2009-04-10 (initial release of advisory)
CVE numbers: CVE-2009-1244

1. Summary

Updated VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability.

2. Relevant releases

VMware Workstation 6.5.1 and earlier,
VMware Player 2.5.1 and earlier,
VMware ACE 2.5.1 and earlier,
VMware Server 2.0,
VMware Server 1.0.8 and earlier,
VMware Fusion 2.0.3 and earlier,

VMware ESXi 3.5 without patch ESXe350-200904201-O-SG,

VMware ESX 3.5 without patch ESX350-200904201-SG,

VMware ESX 3.0.3 without patch ESX303-200904403-SG,

VMware ESX 3.0.2 without patch ESX-1008421.

NOTE: General Support for Workstation version 5.x ended on 2009-03-19. Users should plan to upgrade to the latest Workstation version 6.x release.

Extended support for ESX 3.0.2 Update 1 ends on 2009-08-08. Users should plan to upgrade to ESX 3.0.3 and preferably to the newest release available.

3. Problem Description

a. Host code execution vulnerability from a guest operating system

A critical vulnerability in the virtual machine display function might allow a guest operating system to run code on the host.

This issue is different from the vulnerability in a guest virtual device driver reported in VMware security advisory VMSA-2009-0005 on 2009-04-03. That vulnerability can cause a potential denial of service and is identified by CVE-2008-4916.

The Common Vulnerabilities and Exposures project (cve.mitre.org) has assigned the name CVE-2009-1244 to this issue.

Xen CVE-2008-1943
VBox CVE-2010-3583

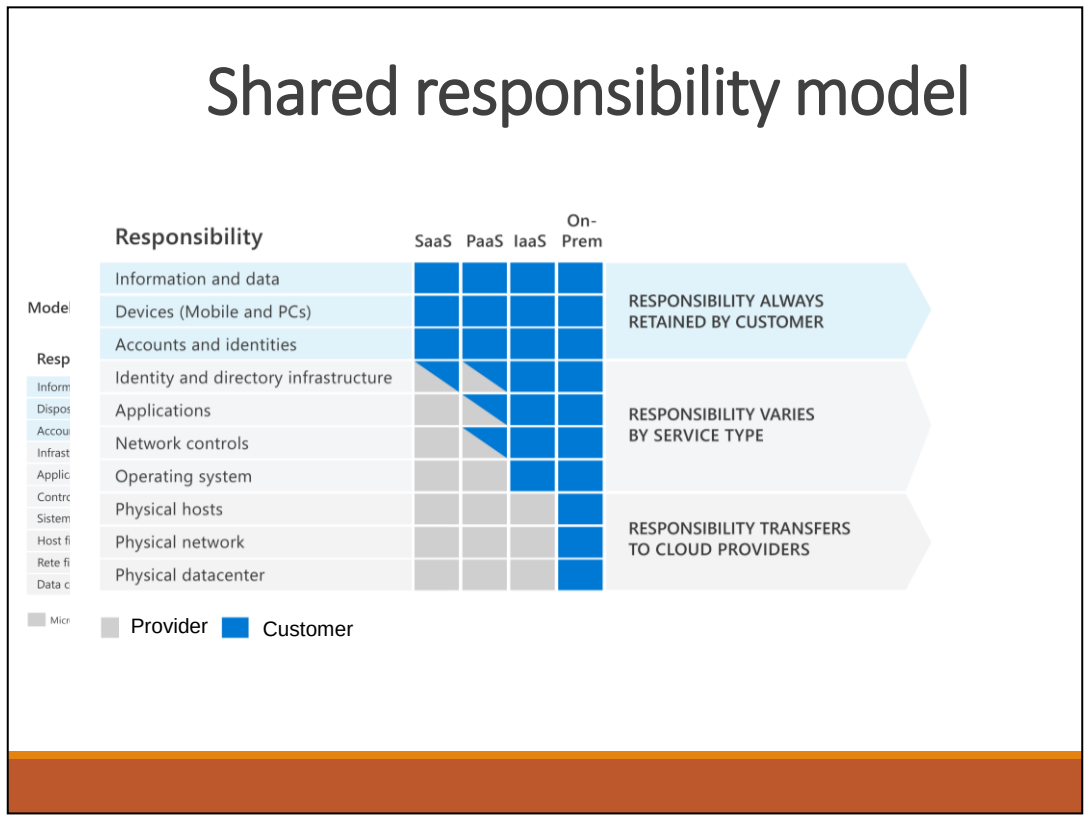
CVE: Common Vulnerabilities and Exposures

Scaling of VMs

- Growth in physical machines limited by budget and setup time.
- Adding a VM is easy as copying a file, leading to explosive growth in VMs.
- Rapid scaling can exceed capacity of organization's security systems.

New Security Issues

- Accountability
- No Security Perimeter
- Larger Attack Surface
- New Side Channels
- Lack of Auditability
- Regulatory Compliance



The *shared responsibility model* identifies which security tasks are handled by the cloud provider, and which security tasks are handled by you, the customer.

In organizations running only on-premises hardware and software, the organization is 100 percent responsible for implementing security and compliance. With cloud-based services, that responsibility is shared between the customer and the cloud provider.

The responsibilities vary depending on where the workload is hosted:

•Software as a Service (SaaS)

•Platform as a Service (PaaS)

•Infrastructure as a Service (IaaS)

•On-premises datacenter (On-prem)

The shared responsibility model makes responsibilities clear. When organizations move data to the cloud, some responsibilities transfer to the cloud provider and some to the customer organization.

The following diagram illustrates the areas of responsibility between the customer and the cloud provider, according to where data is held.

On-premises datacenters

In an on-premises datacenter, you have responsibility for everything from physical security to encrypting sensitive data.

Infrastructure as a Service (IaaS)

Of all cloud services, IaaS requires the most management by the cloud customer.

With IaaS, you're using the cloud provider’s computing infrastructure. The cloud customer isn't responsible for the physical components, such as computers and the network, or the physical security of the datacenter. However, the cloud customer still has responsibility for software components such as operating systems, network controls, applications, and protecting data.

Platform as a Service (PaaS)

PaaS provides an environment for building, testing, and deploying software applications. The goal of PaaS is to help you create an application quickly without managing the underlying infrastructure. With PaaS, the cloud provider manages the hardware and operating systems, and the customer is responsible for applications and data.

Software as a Service (SaaS)

SaaS is hosted and managed by the cloud provider, for the customer. It's usually licensed through a monthly or annual subscription. Microsoft 365, Skype, and Dynamics CRM Online are all examples of SaaS software. SaaS requires the least amount of management by the cloud customer. The cloud provider is responsible for managing everything except data, devices, accounts, and identities.

For all cloud deployment types you, the cloud customer, own your data and identities. You're responsible for protecting the security of your data and identities, and on-premises resources.

In summary, responsibilities always retained by the customer organization include:

•Information and data

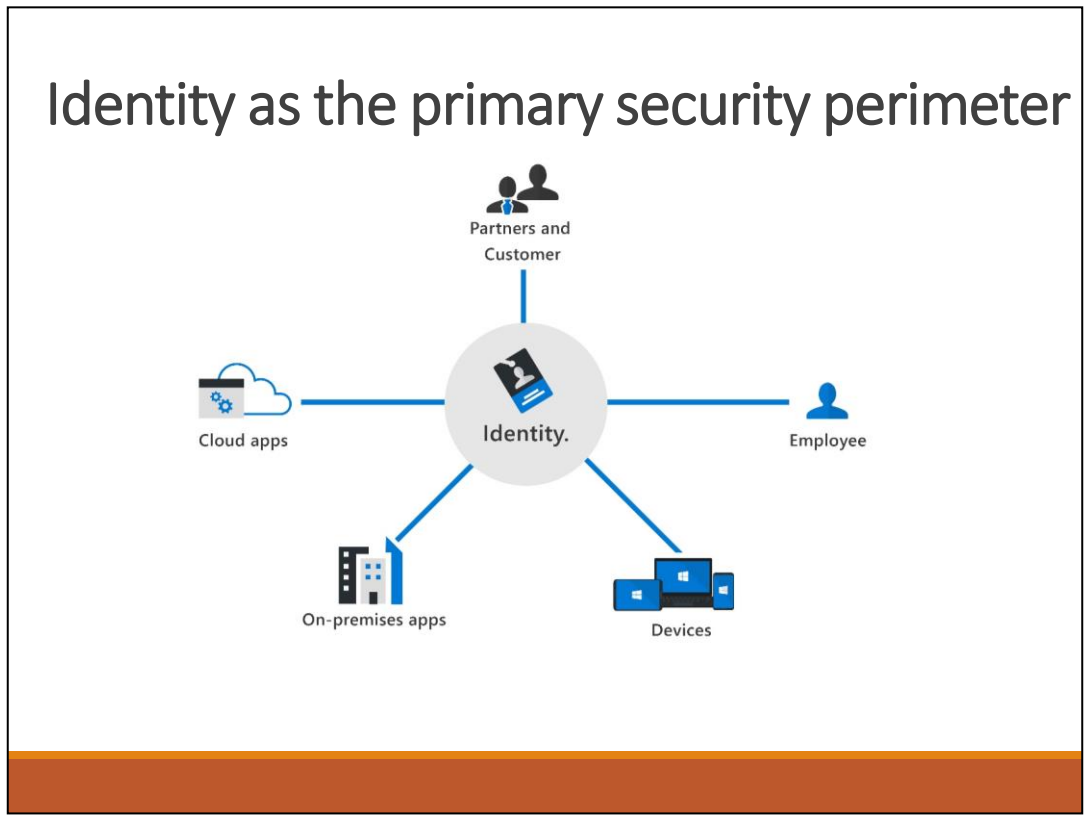
•Devices (mobile and PCs)

•Accounts and identities

The benefit of the shared responsibility model is that organizations are clear about their responsibilities, and those of the cloud provider.

No Security Perimeter

- Little control over physical or network location of cloud instance VMs
- Network access must be controlled on a host by host basis.



Digital collaboration has changed. Your employees and partners now need to collaborate and access organizational resources from anywhere, on any device, and without affecting their productivity. There has also been an acceleration in the number of people working from home.

Enterprise security needs to adapt to this new reality. The security perimeter can no longer be viewed as the on-premises network. It now extends to:

•SaaS applications for business-critical workloads that might be hosted outside the corporate network.

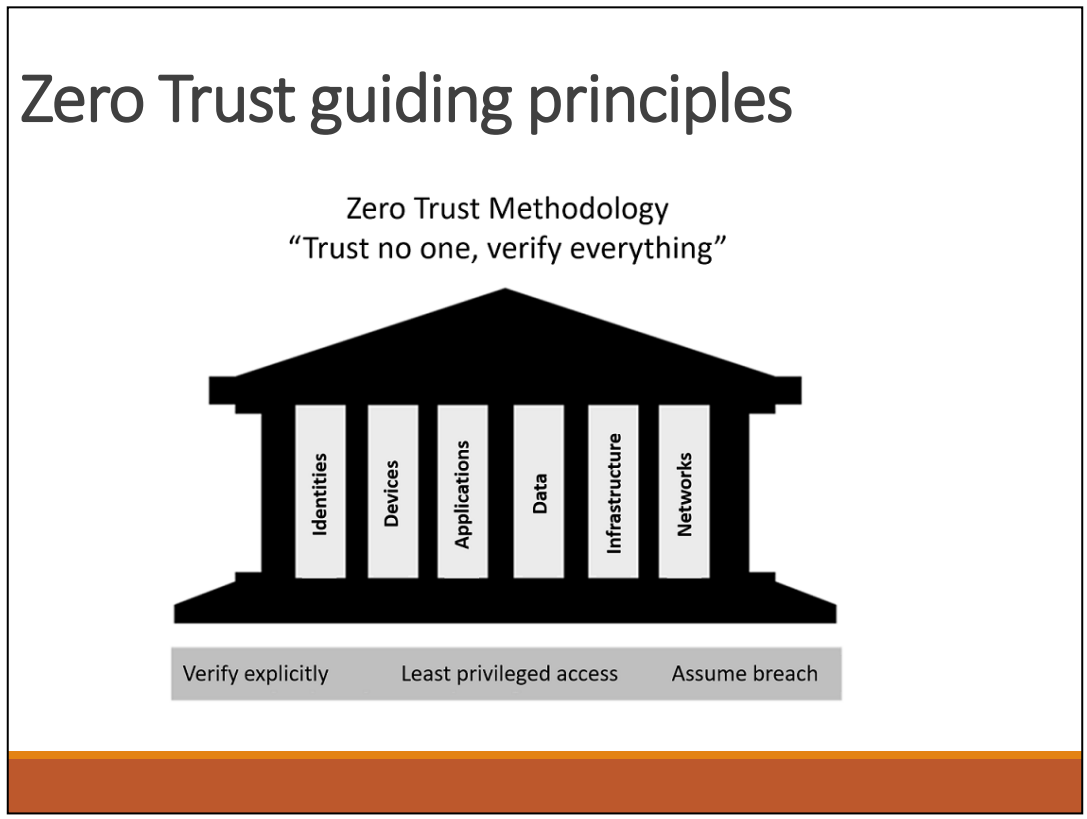
•The personal devices that employees are using to access corporate resources (BYOD, or bring your own device) while working from home.

•The unmanaged devices used by partners or customers when interacting with corporate data or collaborating with employees

•IoT devices installed throughout your corporate network and inside customer locations.

The traditional perimeter-based security model is no longer enough. Identity has become the new security perimeter that enables organizations to secure their assets.

But what do we mean by an identity? An identity is how someone or something can be verified and authenticated to be who they say they are. An identity may be associated with a user, an application, a device, or something else.



https://docs.microsoft.com/it-it/learn/modules/describe-security-concepts-methodologies/2-describe-zero-trust-methodology

Zero Trust assumes everything is on an open and untrusted network, even resources behind the firewalls of the corporate network. The Zero Trust model operates on the principle of "**trust no one, verify everything.**"

Attackers' ability to bypass conventional access controls is ending any illusion that traditional security strategies are sufficient. By no longer trusting the integrity of the corporate network, security is strengthened.

In practice, this means that we no longer assume that a password is sufficient to validate a user but add multi-factor authentication to provide additional checks. Instead of granting access to all devices on the corporate network, users are allowed access only to the specific applications or data that they need.

The Zero Trust model has three principles which guide and underpin how security is implemented. These are: verify explicitly, least privilege access, and assume breach.

•**Verify explicitly**. Always authenticate and authorize based on the available data points, including user identity, location, device, service or workload, data classification, and anomalies.

•**Least privileged access**. Limit user access with just-in-time and just-enough access (JIT/JEA), risk-based adaptive policies, and data protection to protect

both data and productivity.

•**Assume breach**. Segment access by network, user, devices, and application. Use encryption to protect data, and use analytics to get visibility, detect threats, and improve your security.

Six foundational pillars

In the Zero Trust model, all elements work together to provide end-to-end security. These six elements are the foundational pillars of the Zero Trust model:

•**Identities** may be users, services, or devices. When an identity attempts to access a resource, it must be verified with strong authentication, and follow least privilege access principles.

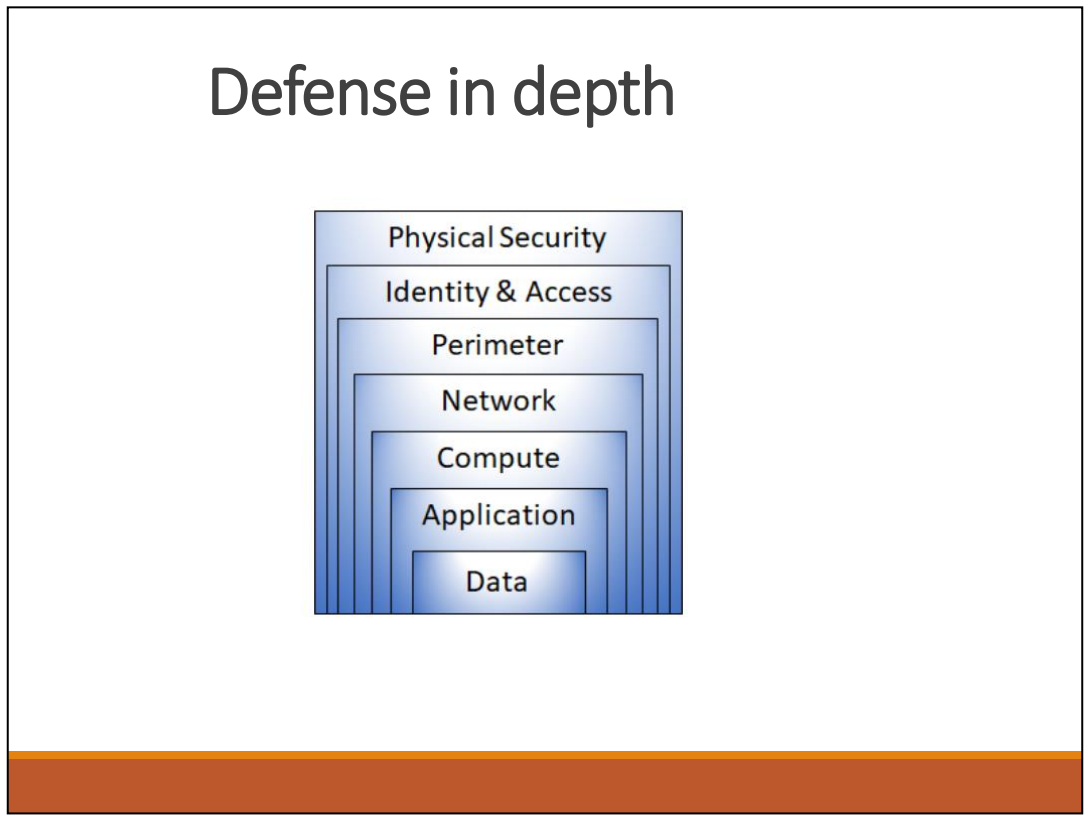
•**Devices** create a large attack surface as data flows from devices to on-premises workloads and the cloud. Monitoring devices for health and compliance is an important aspect of security.

•**Applications** are the way that data is consumed. This includes discovering all applications being used, sometimes called Shadow IT because not all applications are managed centrally. This pillar also includes managing permissions and access.

•**Data** should be classified, labeled, and encrypted based on its attributes. Security efforts are ultimately about protecting data, and ensuring it remains safe when it leaves devices, applications, infrastructure, and networks that the organization controls.

•**Infrastructure**, whether on-premises or cloud based, represents a threat vector. To improve security, you assess for version, configuration, and JIT access, and use telemetry to detect attacks and anomalies. This allows you to automatically block or flag risky behavior and take protective actions.

•**Networks** should be segmented, including deeper in-network micro segmentation. Also, real-time threat protection, end-to-end encryption, monitoring, and analytics should be employed.



Defense in depth uses a layered approach to security, rather than relying on a single perimeter. A defense in-depth strategy uses a series of mechanisms to slow the advance of an attack. Each layer provides protection so that, if one layer is breached, a subsequent layer will prevent an attacker getting unauthorized access to data.

Example layers of security might include:

•**Physical** security such as limiting access to a datacenter to only authorized personnel.

•**Identity and access** security controls, such as multi-factor authentication or condition-based access, to control access to infrastructure and change control.

•**Perimeter** security including distributed denial of service (DDoS) protection to filter large-scale attacks before they can cause a denial of service for users.

•**Network** security, such as network segmentation and network access controls, to limit communication between resources.

•**Compute** layer security such as securing access to virtual machines either on-premises or in the cloud by closing certain ports.

•**Application** layer security to ensure applications are secure and free of security vulnerabilities.

•**Data** layer security including controls to manage access to business and customer data and encryption to protect data.

Examples of Cloud Services

- Amazon EC2
- Amazon S3
- Amazon RDS
- MS Azure
- OpenStack

Amazon Cloud: EC2

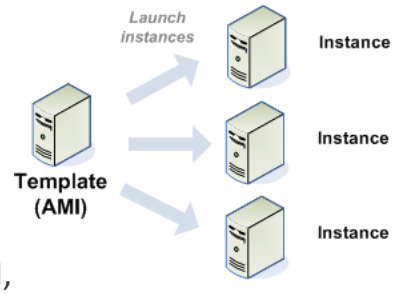
Amazon Elastic Compute Cloud (Amazon EC2) is a **web service** that provides **resizeable computing capacity**—literally, servers in Amazon's data centers—that you use to build and host your software systems. You can access the components and features that EC2 provides using a web-based GUI, command line tools, and APIs.

With EC2, you use and **pay for only the capacity that you need**. This eliminates the need to make large and expensive hardware purchases, reduces the need to forecast traffic, and enables you to automatically **scale your IT resources** to deal with changes in requirements or spikes in popularity related to your application or service.

Components of EC2: Amazon Machine Images and Instances, Regions and Availability Zones, Storage, Databases, Networking and Security, Monitoring, Auto-Scaling and Load Balancing, AWS Identity and Access Management.

Amazon Cloud EC2: AMI

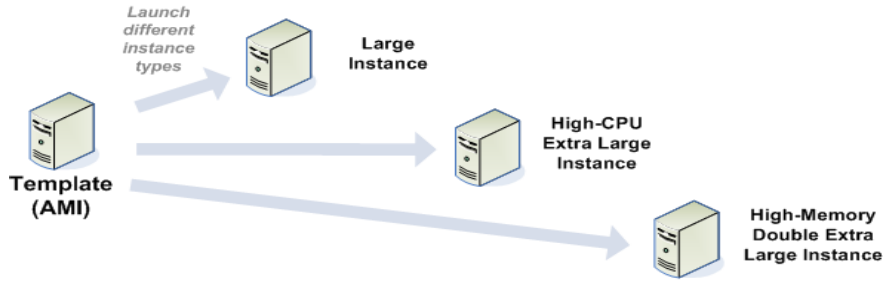
An *Amazon Machine Image (AMI)* is a template that contains a software configuration (operating system, application server, and applications). From an AMI, you launch *instances*, which are running copies of the AMI. You can launch multiple instances of an AMI, as shown in the figure.



Your instances keep running until you stop or you terminate them, or until they fail. If an instance fails, you can launch a new one from the AMI.
You can use a single AMI or multiple AMIs depending on your needs. From a single AMI, you can launch different *types of instances*.

Amazon Cloud EC2: AMI

- An *instance type* is essentially a hardware archetype. As illustrated in the following figure, you select a particular instance type based on the amount of memory and computing power you need for the application or software that you plan to run on the instance.
- Amazon publishes many AMIs that contain common software configurations for public use. In addition, membersFor of the AWS developer community have published their own custom AMIs.



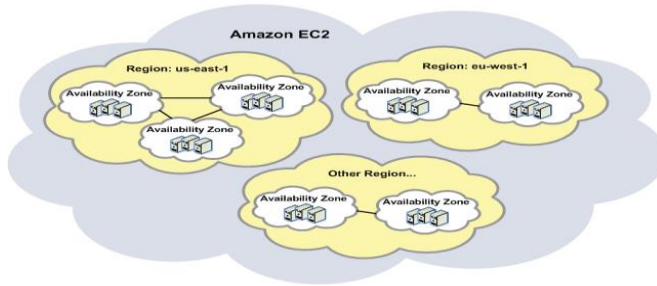
42

example, if your application is a web site or web service, your AMI could be preconfigured with a web server, the associated static content, and the code for all dynamic pages. Alternatively, you could configure your AMI to install all required software components and content itself by running a bootstrap script as soon as the instance starts. As a result, after launching the AMI, your web server will start and your application can begin accepting requests.

Amazon EC2: Regions and Availability Zones

Amazon has data centers in different areas of the world (for example, North America, Europe, and Asia). Correspondingly, Amazon EC2 is available to use in different *Regions*. By launching instances in separate Regions, you can design your application to be closer to specific customers or to meet legal or other requirements.

Each Region contains multiple distinct locations called *Availability Zones* . Each Availability Zone is **engineered to be isolated from failures** in other Availability zones and to provide **inexpensive, low-latency network connectivity to other zones in the same Region**. By launching instances in separate Availability Zones, you can protect your applications from the failure of a single location.



Amazon Storage Services

- Instance Store
- Amazon EC2 Storage
- Amazon S3 Storage

Instance Store

All instance types, with the exception of Micro instances, offer *instance store*. This is storage that doesn't persist if the instance is stopped or terminated. Instance store is an option for inexpensive temporary storage. You can use instance store volumes if you don't require data persistence.

Amazon Cloud EC2: Storage

Amazon EBS

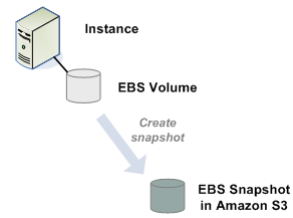
Amazon EBS volumes are the recommended storage option for the majority of use cases. Amazon EBS provides the instances with persistent, **block-level storage**. Amazon EBS volumes are essentially hard disks that you can attach to a running instance.

Amazon EBS is particularly suited for applications that require a database, file system, or access to raw block-level storage.

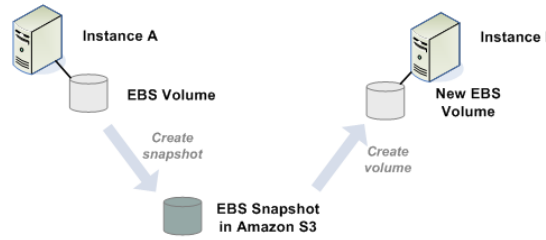
EBS: Elastic Block Store. You can attach multiple volumes to an instance.

Amazon Cloud EC2: Storage

To keep a back-up copy, you can create a *snapshot* of the volume. As illustrated in the following figure, snapshots are stored in Amazon S3.



You can create a new Amazon EBS volume from a snapshot, and attach it to another instance, as illustrated in the following figure.



Amazon Cloud EC2: Storage

You can also detach a volume from an instance and attach it to a different one, as illustrated in the following figure.

Amazon Cloud S3

Amazon S3

Amazon S3 is storage for the Internet. It provides a simple web service interface that enables you to store and retrieve any amount of data from anywhere on the web. Type of Object Storage.

48

Amazon S3 Functionality

1. **Write, read, and delete** objects containing from 1 byte to 5 terabytes of data each.
2. The number of objects you can store is **unlimited**.
3. Each object is stored in a bucket and retrieved via a **unique, developer-assigned key**.
4. A bucket can be stored in one of several Regions. You can choose a Region to **optimize for latency, minimize costs**, or address regulatory requirements.
5. Objects stored in a Region **never leave the Region** unless you transfer them out. For example, objects stored in the EU (Ireland) Region never leave the EU.
6. **Authentication** mechanisms are provided to ensure that data is kept secure from unauthorized access. Objects can be made private or public, and rights can be granted to specific users.
7. Options for **secure data upload/download and encryption** of data at rest are provided for additional data protection.
8. Uses standards-based **REST and SOAP interfaces** designed to work with any Internet-development toolkit.

Amazon Cloud S3: Use Cases

Content Storage, **Distribution** and **Sharing**

- Amazon S3 can store a variety of content ranging from web applications to media files. A user can offload an entire storage infrastructure onto the cloud.

Big Data Storage for Data Analysis

- Whether a user is storing pharmaceutical data for analysis, financial data for computation and pricing, or photo images for resizing, Amazon S3 can be used to store the original content. The user can then send this content to Amazon EC2 for computation, resizing, or other large scale analytics – without incurring any data transfer charges for moving the data between the services.

Backup, Archiving and Disaster Recovery

- The Amazon S3 solution offers a scalable and secure solution for backing up and archiving critical data.

49

Using Amazon S3 is easy. To get started you:

Create a Bucket to store your data. You can choose a Region where your bucket and object(s) reside to optimize latency, minimize costs, or address regulatory requirements.

Upload Objects to your Bucket. Your data is durably stored and backed by the Amazon S3 Service Level Agreement.

Using Amazon S3 is easy. To get started you:

Create a Bucket to store your data. You can choose a Region where your bucket and object(s) reside to optimize latency, minimize costs, or address regulatory requirements.

Upload Objects to your Bucket. Your data is durably stored and backed by the Amazon S3 Service Level Agreement.

Optionally, set access controls. You can grants others access to your data from anywhere in the world.

Amazon Cloud: Databases

If the application running on EC2 needs a database, the common ways to implement a database for the application are:

- Use Amazon Relational Database Service (Amazon RDS) to get a managed relational database in the cloud
- Launch an instance of a database AMI, and use that EC2 instance as the database

Amazon RDS offers the advantage of handling database management tasks, such as patching the software, backing up and storing the backups

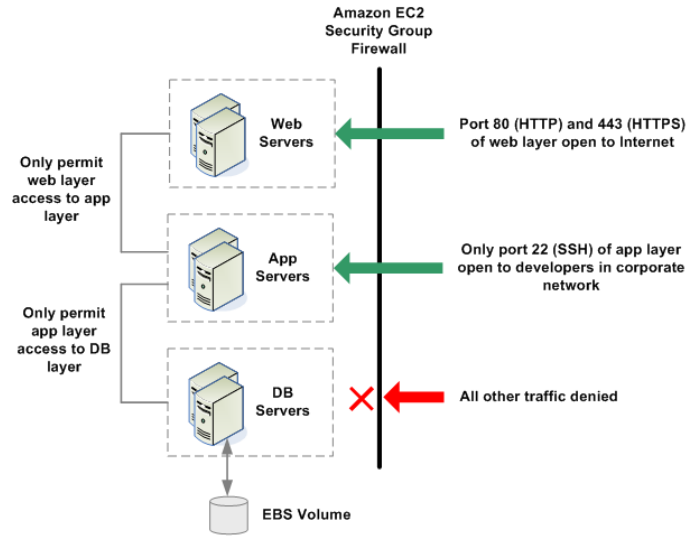
Amazon Cloud: Networking and Security

- **Each instance** is launched into the Amazon EC2 network space and assigned a **public IP address**. If an instance fails and a replacement instance is launched, the **replacement will have a different public IP address** than the original.
- *Security groups* are used to **control access to user instances**. These are analogous to an inbound network **firewall** that allows a user to specify the protocols, ports, and source IP ranges that are allowed to reach user instances.
- A user can create **multiple security groups** and assign different rules to each group. Each instance can be assigned to one or more security groups, and the rules determine which traffic is allowed in to the instance. A security group can be configured so that only specific IP addresses or specific security groups have access to the instance.

51

However, your application might need a static IP address. Amazon EC2 offers *elastic IP addresses* for those situations. For more information, see Instance IP Addresses.

Amazon Cloud: Networking and Security



52

The figure shows a basic three-tier web-hosting architecture running on Amazon EC2 instances. Each layer has a different security group (indicated by the dotted line around each set of instances). The security group for the web servers only allows access from hosts over TCP on ports 80 and 443 (HTTP and HTTPS) and from instances in the *App Servers* security group on port 22 (SSH) for direct host management.

The security group for the app servers allows access from the *Web Servers* security group for web requests, and from the corporate subnet over TCP on port 22 (SSH) for direct host management. The user's support engineers could log directly into the application servers from the corporate network, and then access

the other instances from the application server boxes.

The *DB Servers* security group permits only the App Servers security group to access the database servers.

Microsoft Azure

- Cloud-computing platform
  - Platform-as-a-Service running custom applications on pre-configured virtual machines
- Benefits
  - Virtualization
  - Scalability
  - Utility Computing
- Azure includes compute, storage, and application services
- It enables customers to store, process and analyze all types of data - from structured, semi-structured, to unstructured. SQL Data Services (SDS), the first service available within SQL Services, is the relational database.

53

Azure Services Platform

The Azure Services Platform is an internet-scale service-based application platform hosted in Microsoft data centers. Azure includes compute, storage, and application services, all available as elastic resources, and all accessible via standard communication protocols and models. Azure works with the tools, skills, data, and applications customers already have, including .NET, Windows, SQL Server, and more. With Azure, customers will build new connected applications or extend existing ones, quickly and cost-effectively. When launched, Azure will offer guaranteed reliable hosted infrastructure under a pay-as-you-go plan. Suitable for use in a wide variety of application types, and by all types and sizes of organizations and companies, Azure will be able to cut capital and operating expenses and reduce risk.

Components of the Azure Services Platform include:

Windows Azure: Our cloud services operating system that provides flexible and scalable service hosting, data storage, and automated service management.

Windows Azure is Microsoft's platform for building and deploying cloud

based applications, which provides customers with a service hosting and management environment for the Azure Services Platform.

Microsoft .NET Services: A set of higher level developer services like Access Control, Workflow, and Messaging that empower developers to build richer, better connected applications with less code.

.NET Services make developing loosely-coupled on-premises and cloud-based applications easier. .NET Services include a hosted service bus for connecting applications and services across network boundaries, access control for securing applications, and message orchestration. These hosted services allow customers to easily create federated applications that span from on-premises to the cloud.

Microsoft SQL Services delivers on the vision of extending the SQL Server Data Platform capabilities to cloud web-based services. It enables customers to store, process and analyze all types of data - from structured, semi-structured, to unstructured. SQL Data Services (SDS), the first service available within SQL Services, is the relational database. In the future, SQL Services will deliver a comprehensive set of integrated services such as search, reporting, analytics, data integration and data synchronization in the cloud.

Microsoft SQL Services: A set of cloud-based SQL Server capabilities. The first of these capabilities is SQL Data Services, which offers an Internet-facing database service with query capabilities and support for standard protocols.

Microsoft SharePoint Services: Applications available as services today that offer the power of choice to businesses of all sizes and are supported by a set of capabilities unique to the world of hybrid scenarios, such as the ability to federate user identities across the server-service continuum, so a user can authenticate to a portfolio of applications, some on-premises and some consumed as a service.

Live Services provides an easy on ramp for developers to connect with over 460 Million Windows Live users and build web apps with rich social sharing embedded experiences. The Live Framework provides the one uniform way to consume and program Live Services and over time will provide the ability to connect the power and scale of web apps to rich client experiences across a world of digital devices.

Live Services: A set of foundational services and APIs that allows third parties to tap into the massive Windows Live consumer network and its core infrastructure to build rich, engaging applications.

On top of the Azure Services Platform, we offer finished consumer services such as Windows Live, MSN, and Office Live, as well as enterprise-class applications such as Microsoft Exchange Online, Sharepoint Online, and CRM Online. These

solutions provide the power of choice for businesses of all sizes and vertical segments and present a set of exciting new opportunities for Microsoft

Azure Overview

Runs on Microsoft data centers
- Commodity hardware

Wide range of app dev technology:
- .NET Framework, Unmanaged code, others..
- C#, VB, C++, Java, ASP.NET, WCF, PHP

Several Storage Options
- BLOBs and simple data structures
  - RESTful approach to Windows Azure storage
- Traditional Relational, SQL Azure Database

Connectivity with other distributed applications

54

-Windows Azure is designed to run on cheap commodity hardware running in Microsoft-managed data centers.

- There is a growing list of application development technology that it supports.

  - From Microsoft centric technologies like .NET and C# to Java and other scripting languages.

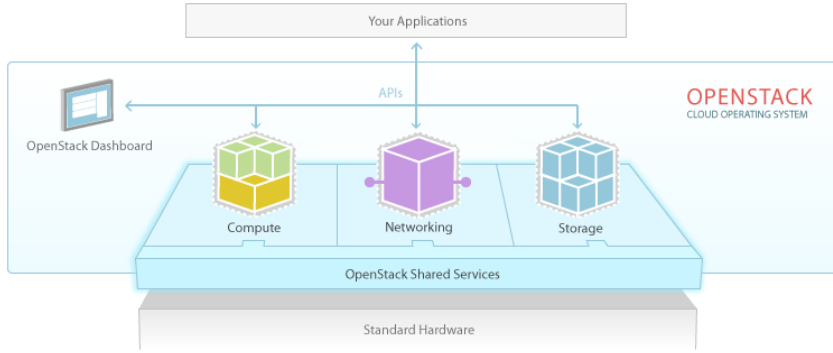
-It also has several storage options so you can choose what's most appropriate for your application

  - For example, for large images and videos, there's support for storing BLOBs on the cloud

  - If you need more traditional database functionality, there's SQL Azure Database

- The Azure Platform also provides mechanisms for connecting distributed applications on and off the cloud.

OpenStack



OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

Designing Cloud-Native Applications

- Designing cloud applications means taking vision and awareness of all the aspects involved, of the opportunities they enable and the issues they raise;
- Consider alternative solutions, carry out choices and approach decisions in a conscious manner in order to achieve what is required by the business in the best possible way, also considering all those maintenance and evolution activities that the high complexity of these areas requires.
- Rather than being conceived as monoliths, applications are **decomposed into decentralized and smaller si**ze. These services communicate through APIs or using event management or asynchronous messaging . Applications scale out by adding new instances as needed.

https://learn.microsoft.com/en-us/azure/architecture/patterns/

no

Designing Cloud-Native Applications

- The cloud solutions differ in terms of the calculation service provided (IaaS, PaaS, SaaS, FaaS) and in data storage system used (relational databases, key/value archives, document databases, graph databases, columnar databases, search engine databases, time series databases, files shared). For the latter there are a number of criteria to be taken into consideration in order to take the right one decision.

Designing Cloud-Native Applications

In order for a cloud application to be scalable, resilient and manageable, a series of principles must be followed:

- Design for **self-healing**. In a distributed system they can errors occur. Design the application to be able to correct automatically correct your mistakes.
- Make everything **redundant**. Apply redundancy in the for application avoid single points of failure.
- **Minimize coordination** between application services to achieve scalability.
- Design for **horizontal scaling**. Design the application to scale horizontally, adding or removing new instances as needed.

Designing Cloud-Native Applications

- Use **managed services**. Whenever possible, use Platform as a Service (PaaS) instead infrastructure as a service (IaaS).
- Use the **best data archive** for the process. Choosing the suitable storage technology for data and the methods of envisaged use.
- Designing from an **evolutionary** point of view. All efficient applications change over time. A Evolving design is critical to continuous innovation.
- Every **design decision** must be justified by a business requirement.

Managed cloud services are **services that offer partial or complete management of a client's cloud resources or infrastructure**. Management responsibilities can include migration, configuration, optimization, security, and maintenance.

Every design decision must be justified by a business requirement. This design principle might seem obvious, but is crucial to keep in mind when designing applications. Must your application support millions of users, or a few thousand? Are there large traffic bursts, or a steady workload? What level of application outage is acceptable? Ultimately, business requirements drive these design considerations.

Cloud Design Patterns

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

Challenges in cloud development

- Data management

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

Additionally, data should be **protected at rest, in transit, and via authorized access mechanisms** to maintain security assurances of confidentiality, integrity, and availability.

https://learn.microsoft.com/en-us/azure/architecture/patterns/

Cloud Design Patterns

Data Management Patterns

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.

https://learn.microsoft.com/en-us/azure/architecture/patterns/

Cache-Aside pattern

Load data on demand into a cache from a data store. This can improve performance and also helps to maintain consistency between data held in the cache and data in the underlying data store.

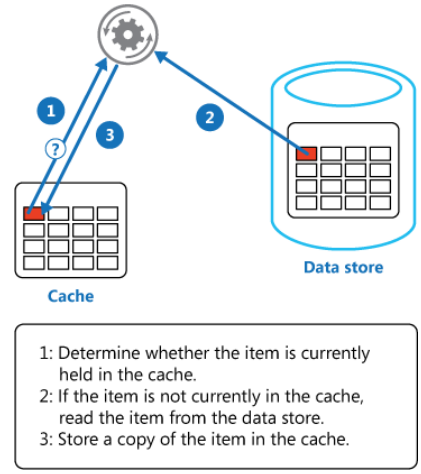
Context and problem
Applications use a cache to improve repeated access to information held in a data store. However, it's impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is as up-to-date as possible, but can also detect and handle situations that arise when the data in the cache has become stale.

https://learn.microsoft.com/en-us/azure/architecture/patterns/

Cache-Aside pattern

Solution
Many commercial caching systems provide **read-through and write-through/write-behind** operations. In these systems, an application retrieves data by referencing the cache. If the data isn't in the cache, it's retrieved from the data store and added to the cache.

For caches that don't provide this functionality, it's the responsibility of the applications that use the cache to maintain the data. **An application can emulate the functionality of read-through caching by implementing the cache-aside strategy. This strategy loads data into the cache on demand. The figure illustrates using the Cache-Aside pattern to store data in the cache.**



•**Cache-aside:** This is where the application is responsible for reading and writing from the database and the cache doesn't interact with the database at all. The cache is "kept aside" as a faster and more scalable in-memory data store. The application checks the cache before reading anything from the database. And, the application updates the cache after making any updates to the database. This way, the application ensures that the cache is kept synchronized with the database.

•**Read-through/Write-through (RT/WT):** This is where the application treats cache as the main data store and reads data from it and writes data to it. The cache is responsible for reading and writing this data to the database, thereby relieving the application of this responsibility.

If an application updates information, it can follow the write-through strategy by making the modification to the data store, and by invalidating the corresponding item in the cache.

When the item is next required, using the cache-aside strategy will cause the updated data to be retrieved from the data store and added back into the cache.

e.g. Documentation of Oracle Coherence:
https://docs.oracle.com/cd/E15357_01/coh.360/e15723/cache_rtwtwbra.htm#CO

When an application asks the cache for an entry, for example the key X, and X is not already in the cache, Coherence will automatically delegate to the CacheStore and ask it to load X from the underlying data source. If X exists in the data source, the CacheStore will load it, return it to Coherence, then Coherence will place it in the cache for future use and finally will return X to the application code that requested it. This is called **Read-Through caching**.

Write-Through: In this case, when the application updates a piece of data in the cache (that is, calls put(...) to change a cache entry,) the operation will not complete (that is, the put will not return) until Coherence has gone through the CacheStore and successfully stored the data to the underlying data source. This does not improve write performance at all, since you are still dealing with the latency of the write to the data source. Improving the write performance is the purpose for the Write-Behind Cache functionality.

In the **Write-Behind** scenario, modified cache entries are asynchronously written to the data source after a configured delay, whether after 10 seconds, 20 minutes, a day, a week or even longer. Note that this only applies to cache inserts and updates - cache entries are removed synchronously from the data source. For Write-Behind caching, Coherence maintains a write-behind queue of the data that must be updated in the data source. When the application updates X in the cache, X is added to the write-behind queue (if it isn't there already; otherwise, it is replaced), and after the specified write-behind delay Coherence will call the CacheStore to update the underlying data source with the latest state of X. Note that the write-behind delay is relative to the first of a series of modifications—in other words, the data in the data source will never lag behind the cache by more than the write-behind delay.

Cache-Aside pattern

Use this pattern when:

- A cache doesn't provide native read-through and write-through operations.
- Resource demand is unpredictable. This pattern enables applications to load data on demand. It makes no assumptions about which data an application will require in advance.

This pattern might not be suitable:

- When the cached data set is static. If the data will fit into the available cache space, prime the cache with the data on startup and apply a policy that prevents the data from expiring.
- For caching session state information in a web application hosted in a web farm. In this environment, you should avoid introducing dependencies based on client-server affinity.

https://learn.microsoft.com/en-us/azure/architecture/patterns/

See https://learn.microsoft.com/en-us/azure/azure-cache-for-redis/cache-aspnet-session-state-provider for session state information.

CQRS

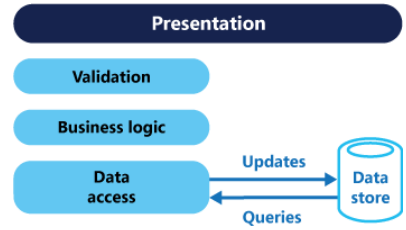
CQRS stands for **Command and Query Responsibility Segregation**, a pattern that separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security.

Context and problem
In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations. In more complex applications, however, this approach can become unwieldy.
Read and write workloads are often asymmetrical, with very different performance and scale requirements

https://learn.microsoft.com/en-us/azure/architecture/patterns/

CQRS

- There is often a **mismatch between the read and write representations of the data**, such as additional columns or properties that must be updated correctly even though they aren't required as part of an operation.
- **Data contention** can occur when operations are performed in parallel on the same set of data.
- The traditional approach can have a negative effect on **performance** due to load on the data store and data access layer, and the complexity of queries required to retrieve information.
- **Managing security and permissions** can become complex, because each entity is subject to both read and write operations, which might expose data in the wrong context.

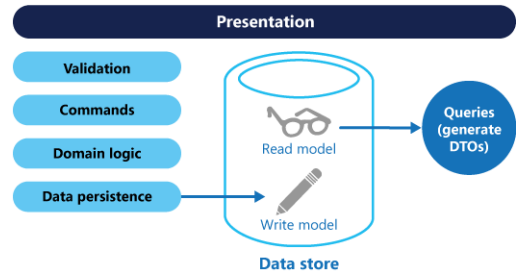


https://learn.microsoft.com/en-us/azure/architecture/patterns/

CQRS

CQRS separates reads and writes into different models, using **commands to update data**, and **queries to read data**.

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



Having separate query and update models simplifies the design and implementation. However, one disadvantage is that CQRS code can't automatically be generated from a database schema using scaffolding mechanisms such as O/RM tools (However, you will be able to build your customization on top of the generated code).

A command is an instruction, a directive to perform a specific task. It is an intention to change something. A command (procedure) does something but does not return a result.

A command should convey the intent of the user. Handling a command should result in one transaction on one aggregate; basically, each command should clearly state one well-defined change.

The commands make up the write model, and the write model should be as close as possible to the business processes.

A query is a request for information. A query (function or attribute) returns a result but does not change the state.It is an intention to get data, or the status of data, from a specific place. Nothing in the data should be changed by the request. As queries do not change anything, they don't need to involve the Domain Model.

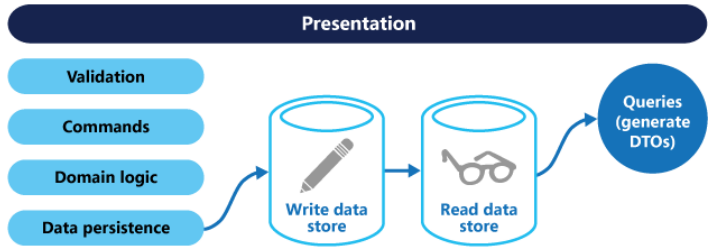
The queries make up the read model. The read model should be derived from the write model. It also doesn't have to be permanent: new read models can be

introduced into the system without impacting the existing ones. Read models can be deleted and recreated without losing the business logic or information, as this is stored in the write model.

A data transfer object (DTO) is an object that carries data between processes. The motivation for its use is that communication between processes is usually done resorting to remote interfaces (e.g., web services), where each call is an expensive operation. Because the majority of the cost of each call is related to the round-trip time between the client and the server, one way of reducing the number of calls is to use an object (the DTO) that aggregates the data that would have been transferred by the several calls, but that is served by one call only.

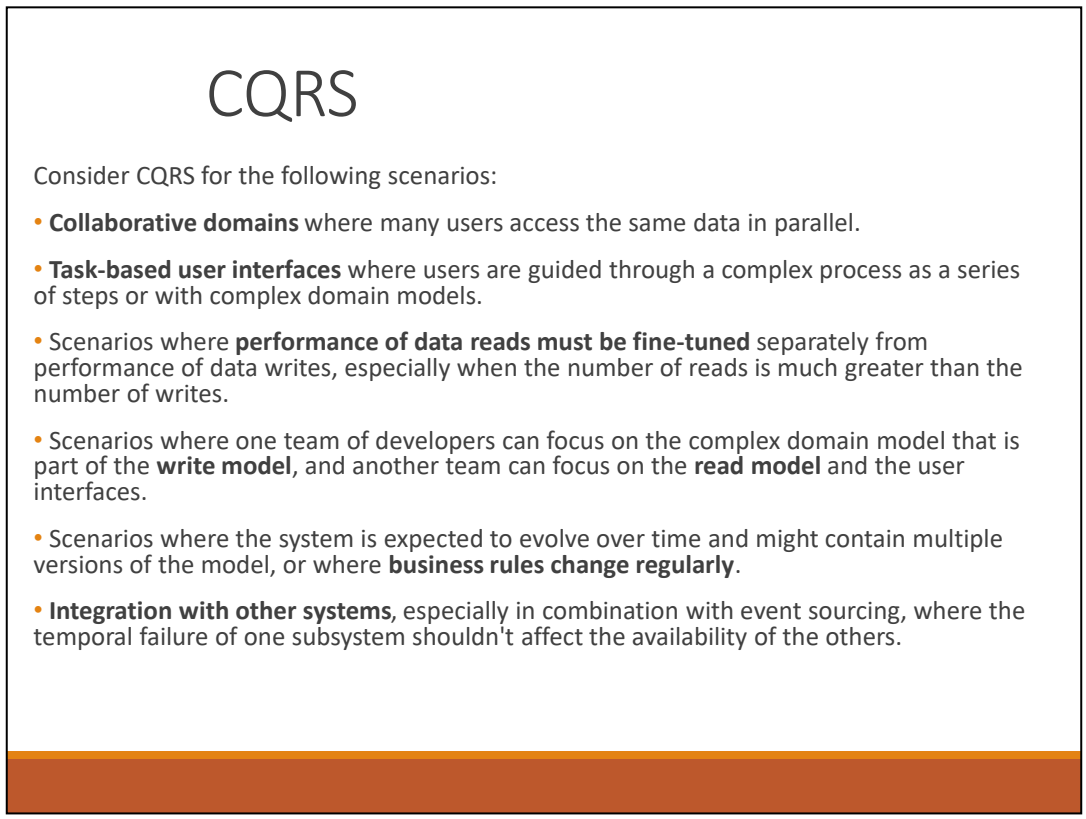
CQRS

For greater isolation, you can physically separate the read data from the write data. In that case, the read database can use its own data schema that is optimized for queries. For example, it can store a materialized view of the data, in order to avoid complex joins or complex O/RM mappings. It might even use a different type of data store. For example, the write database might be relational, while the read database is a document database.



If separate read and write databases are used, they must be kept in sync. Typically this is accomplished by having the write model publish an event whenever it updates the database. For more information about using events, see Event-driven architecture style. Since message brokers and databases usually cannot be enlisted into a single distributed transaction, there can be challenges in guaranteeing consistency when updating the database and publishing events. For more information, see the guidance on idempotent message processing.

Object–relational mapping (ORM, O/RM, and O/R mapping tool) is a programming technique for converting data between a relational database and the heap of an object-oriented programming language. This creates, in effect, a virtual object database that can be used from within the programming language.



Consider CQRS for the following scenarios:

•Collaborative domains where many users access the same data in parallel. CQRS allows you to define commands with enough granularity to minimize merge conflicts at the domain level, and conflicts that do arise can be merged by the command.

•Task-based user interfaces where users are guided through a complex process as a series of steps or with complex domain models. The write model has a full command-processing stack with business logic, input validation, and business validation. The write model may treat a set of associated objects as a single unit for data changes (an aggregate, in DDD terminology) and ensure that these objects are always in a consistent state. The read model has no business logic or validation stack, and just returns a DTO for use in a view model. The read model is eventually consistent with the write model.

•Scenarios where performance of data reads must be fine-tuned separately from performance of data writes, especially when the number of reads is much greater than the number of writes. In this scenario, you can scale out the read model, but run the write model on just a few instances. A small number of write model instances also helps to minimize the occurrence of merge conflicts.

•Scenarios where one team of developers can focus on the complex domain model that is part of the write model, and another team can focus on the read model and the user interfaces.

•Scenarios where the system is expected to evolve over time and might contain multiple versions of the model, or where business rules change regularly.

•Integration with other systems, especially in combination with event sourcing, where the temporal failure of one subsystem shouldn't affect the availability of the others.

CQRS

- This pattern isn't recommended when:
  - The domain or the business rules are simple.
  - A simple CRUD-style user interface and data access operations are sufficient.
- Consider applying CQRS to limited sections of your system where it will be most valuable.

Event Sourcing pattern

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with it, as in the traditional create, read, update, and delete (CRUD) model.

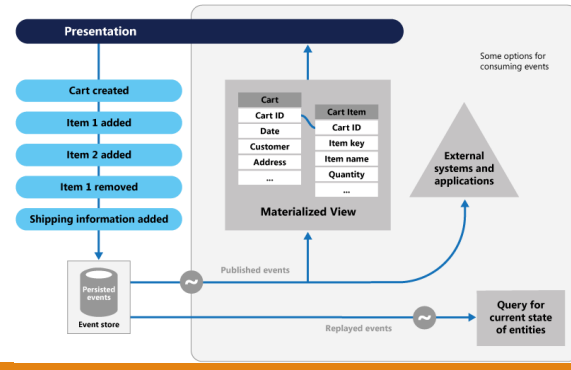
The CRUD approach has some limitations:

- CRUD systems perform **update operations directly against a data store**. These operations can **slow down performance and responsiveness** and can limit scalability, due to the processing overhead it requires.
- In a **collaborative domain** with many concurrent users, data **update conflicts are more likely** because the update operations take place on a single item of data.
- Unless there's another auditing mechanism that records the details of each operation in a separate log, **history is lost**.

https://learn.microsoft.com/en-us/azure/architecture/patterns/

Event Sourcing pattern

The Event Sourcing pattern defines an approach to handling operations on data that is driven by a sequence of events, each of which is **recorded in an append-only store**. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they are persisted. Each event represents a set of changes to the data (such as AddedItemToOrder).



The events are persisted in an event store that acts as the system of record (the authoritative data source) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.

Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems. For example, a system can maintain a materialized view of all customer orders that's used to populate parts of the UI. The application adds new orders, adds or removes items on the order, and adds shipping information. **The events that describe these changes can be handled and used to update the materialized view.**

At any point, it's possible for applications to **read the history of events**. You can then use it to materialize the current state of an entity by playing back and consuming all the events that are related to that entity. This process can occur on demand to materialize a domain object when handling a request. Or, the process occurs through a scheduled task so that the state of the entity can be stored as a materialized view, to support the presentation layer.

Event Sourcing pattern

Use this pattern in the following scenarios:

- When you want to **capture intent, purpose, or reason** in the data.
- When it is vital to **minimize or completely avoid** the occurrence of **conflicting updates** to data.
- When you want to record events that occur, to replay them to **restore the state of a system, to roll back changes, or to keep a history and audit log**.
- When you use **events**. It is a natural feature of the operation of the application, and it requires little extra development or implementation effort.
- When you need to **decouple** the process of inputting, or updating data from the tasks required to apply these actions.
- When you want **flexibility** to be able to **change the format of materialized models** and entity data if requirements change, or—when used with CQRS—you need to adapt a read model or the views that expose the data.
- When used with CQRS, and **eventual consistency is acceptable while a read model is updated**, or the performance impact of rehydrating entities and data from an event stream is acceptable.

Event Sourcing pattern

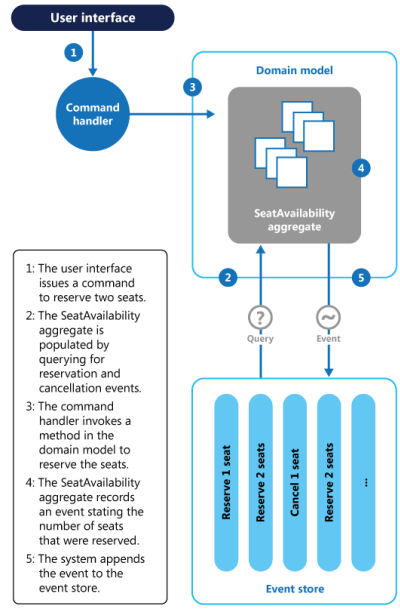
This pattern might not be useful in the following situations:

- Small or simple domains, systems that have little or no business logic, or nondomain systems that naturally work well with traditional CRUD data management mechanisms.
- Systems where consistency and real-time updates to the views of the data are required.
- Systems where audit trails, history, and capabilities to roll back and replay actions aren't required.
- Systems where there is only a low occurrence of conflicting updates to the underlying data. For example, systems that predominantly add data rather than updating it.

Event Sourcing pattern

Example: Conference management system

- A conference management system needs to track the number of completed bookings for a conference. This way it can check whether there are seats still available, when a potential attendee tries to make a booking.



The system could store the total number of bookings for a conference in at least two ways:

•The system could store the information about the total number of bookings as a separate entity in a database that holds booking information. As bookings are made or canceled, the system could increment or decrement this number as appropriate. This approach is simple in theory, but can cause scalability issues if a large number of attendees are attempting to book seats during a short period of time. For example, in the last day or so prior to the booking period closing.

•The system could store information about bookings and cancellations as events held in an event store. It could then calculate the number of seats available by replaying these events. This approach can be more scalable due to the immutability of events. The system only needs to be able to read data from the event store, or append data to the event store. Event information about bookings and cancellations is never modified.

The sequence of actions for reserving two seats is as follows:

1.The user interface issues a command to reserve seats for two attendees. The command is handled by a separate command handler. A piece of logic that is decoupled from the user interface and is responsible for handling requests posted as commands.

2.An aggregate containing information about all reservations for the conference is

constructed by querying the events that describe bookings and cancellations. This aggregate is called SeatAvailability, and is contained within a domain model that exposes methods for querying and modifying the data in the aggregate.

3. Some optimizations to consider are using snapshots (so that you don't need to query and replay the full list of events to obtain the current state of the aggregate), and maintaining a cached copy of the aggregate in memory.

4.The command handler invokes a method exposed by the domain model to make the reservations.

5.The SeatAvailability aggregate records an event containing the number of seats that were reserved. The next time the aggregate applies events, all the reservations will be used to compute how many seats remain.

6.The system appends the new event to the list of events in the event store.

If a user cancels a seat, the system follows a similar process except the command handler issues a command that generates a seat cancellation event and appends it to the event store.

In addition to providing more scope for scalability, using an event store also provides a complete history, or audit trail, of the bookings and cancellations for a conference. The events in the event store are the accurate record. There's no need to persist aggregates in any other way because the system can easily replay the events and restore the state to any point in time.

Index Table pattern

Create indexes over the fields in data stores that are frequently referenced by queries. This pattern can improve query performance by allowing applications to more quickly locate the data to retrieve from a data store.

Context and problem

- Many data stores organize the data for a collection of entities using the **primary key**. An application can use this key to locate and retrieve data.
- While the primary key is valuable for queries that fetch data based on the value of this key, **an application might not be able to use the primary key if it needs to retrieve data based on some other field**.
  - For example, if Customer ID is the primary key to retrieve customers, if an application queries data solely by referencing the value of some other attribute, such as the town in which the customer is located the application might have to fetch and examine every customer record, which could be a slow process.
- Many relational database management systems support **secondary indexes**, but most NoSQL data stores used by cloud applications don't provide an equivalent feature

Many relational database management systems support secondary indexes. A secondary index is a separate data structure that's organized by one or more nonprimary (secondary) key fields, and it indicates where the data for each indexed value is stored. The items in a secondary index are typically sorted by the value of the secondary keys to enable fast lookup of data. These indexes are usually maintained automatically by the database management system.

You can create as many secondary indexes as you need to support the different queries that your application performs. For example, in a Customers table in a relational database where the Customer ID is the primary key, it's beneficial to add a secondary index over the town field if the application frequently looks up customers by the town where they reside.

However, although secondary indexes are common in relational systems, most NoSQL data stores used by cloud applications don't provide an equivalent feature.

Index Table pattern

If the data store does not support secondary indexes, you can **emulate them manually** by creating your own index tables.

- **Three strategies** are commonly used for structuring an index table, depending on the number of secondary indexes that are required and the nature of the queries that an application performs.
- The **first strategy** is to **duplicate the data** in each index table but organize it by different keys (complete denormalization).

Secondary Key (Town)	Customer Data
Chicago	ID: 5, LastName: Smith, Town: Chicago, ...
Chicago	ID: 9, LastName: Jones, Town: Chicago, ...
Chicago	ID: 1000, LastName: Clarke, Town: Chicago, ...
...	...
Portland	ID: 3, LastName: Robinson, Town: Portland, ...
Portland	ID: 7, LastName: Clarke, Town: Portland, ...
Redmond	ID: 1, LastName: Smith, Town: Redmond, ...
Redmond	ID: 4, LastName: Brown, Town: Redmond, ...
Redmond	ID: 6, LastName: Green, Town: Redmond, ...
Redmond	ID: 8, LastName: Smith, Town: Redmond, ...
Seattle	ID: 2, LastName: Jones, Town: Seattle, ...
...	...

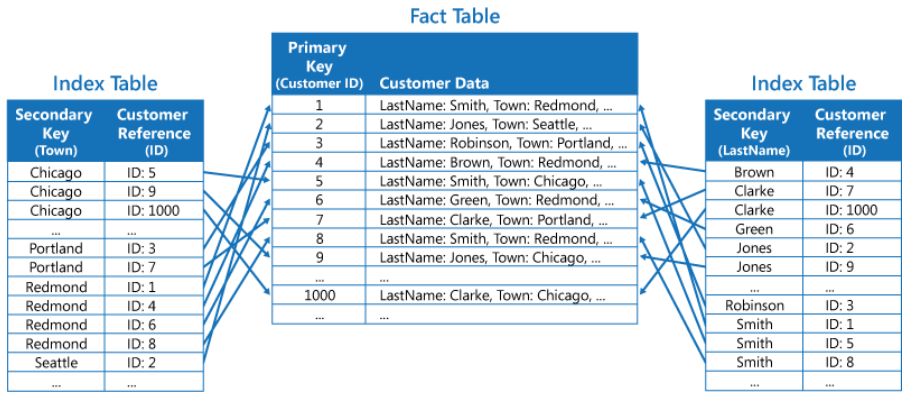
Secondary Key (LastName)	Customer Data
Brown	ID: 4, LastName: Brown, Town: Redmond, ...
Clarke	ID: 7, LastName: Clarke, Town: Portland, ...
Clarke	ID: 1000, LastName: Clarke, Town: Chicago, ...
Green	ID: 6, LastName: Green, Town: Redmond, ...
Jones	ID: 2, LastName: Jones, Town: Seattle, ...
Jones	ID: 9, LastName: Jones, Town: Chicago, ...
...	...
Robinson	ID: 3, LastName: Robinson, Town: Portland, ...
Smith	ID: 1, LastName: Smith, Town: Redmond, ...
Smith	ID: 5, LastName: Smith, Town: Chicago, ...
Smith	ID: 8, LastName: Smith, Town: Redmond, ...
...	...

If the data store doesn't support secondary indexes, you can emulate them manually by creating your own index tables. An index table organizes the data by a specified key. Three strategies are commonly used for structuring an index table, depending on the number of secondary indexes that are required and the nature of the queries that an application performs.

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

Index Table pattern

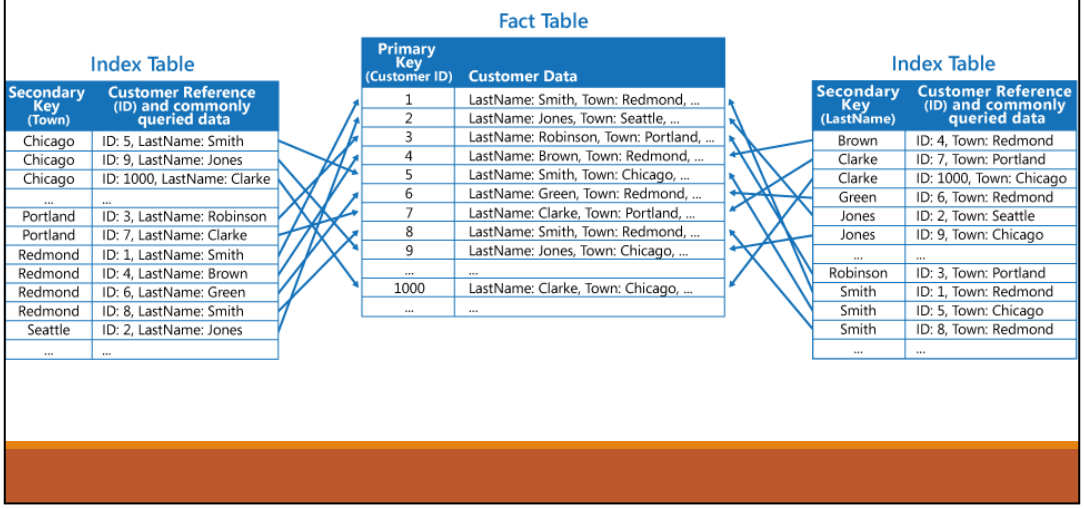
The **second strategy** is to **create normalized index tables organized by different keys** and reference the original data by using the primary key rather than duplicating it, as shown in the following figure. The original data is called a fact table.



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

Index Table pattern

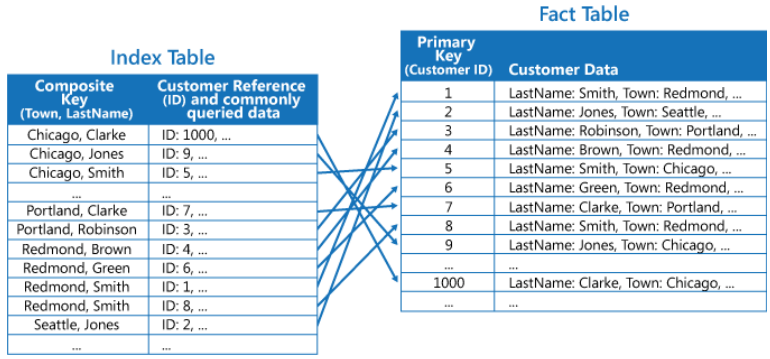
The **third strategy** is to create **partially normalized index tables organized by different keys that duplicate frequently retrieved fields**. Reference the fact table to access less frequently accessed fields. The next figure shows how commonly accessed data is duplicated in each index table.



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

Index Table pattern

If an application frequently queries data by specifying a combination of values (for example, "Find all customers that live in Redmond and that have a last name of Smith"), you could implement the keys to the items in the index table as a concatenation of the Town attribute and the LastName attribute. The next figure shows an index table based on composite keys. The keys are sorted by Town, and then by LastName for records that have the same value for Town.



Index tables can speed up query operations over sharded data, and are especially useful where the shard key is hashed. T

Index Table pattern

Issues and considerations

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

- The **overhead of maintaining secondary indexes** can be significant. You must analyze and understand the queries that your application uses. Only create index tables when they're likely to be used regularly. Do not create speculative index tables to support queries that an application does not perform, or performs only occasionally.
- **Duplicating data in an index table can add significant overhead** in storage costs and the effort required to maintain multiple copies of data.
- Implementing an index table as a normalized structure that references the original data requires **an application to perform two lookup operations** to find data. The first operation searches the index table to retrieve the primary key, and the second uses the primary key to fetch the data.
- If a system incorporates a number of index tables over very large data sets, it can be **difficult to maintain consistency** between index tables and the original data.

Index Table pattern

When to use this pattern

Use this pattern to **improve query performance** when an application frequently needs to retrieve data by using a key other than the primary (or shard) key.

This pattern might not be useful when:

- **Data is volatile**. An index table can become out of date very quickly, making it ineffective or making the overhead of maintaining the index table greater than any savings made by using it.
- A field selected as the **secondary key** for an index table **is nondiscriminating** and can only have a small set of values (for example, gender).
- The balance of the **data values** for a field selected as the **secondary key** for an index table are **highly skewed**. For example, if 90% of the records contain the same value in a field, then creating and maintaining an index table to look up data based on this field might create more overhead than scanning sequentially through the data. However, if queries very frequently target values that lie in the remaining 10%, this index can be useful. You should understand the queries that your application is performing, and how frequently they're performed.

Materialized View pattern

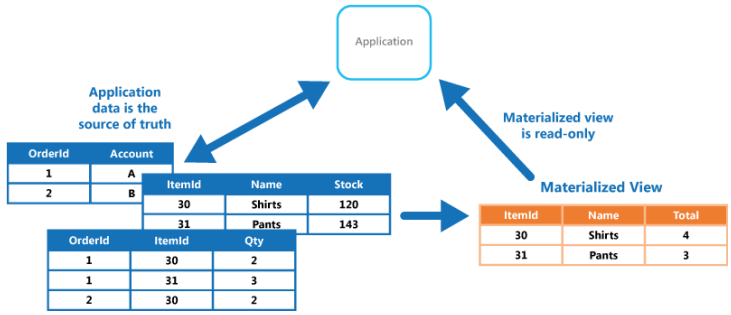
Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations. This can help support efficient querying and data extraction, and improve application performance.

Context and problem

- When storing data, the priority for developers and data administrators is often focused on how the data is stored, as opposed to how it's read. The chosen storage format is usually closely related to the format of the data, requirements for managing data size and data integrity, and the kind of store in use. For example, when using NoSQL document store, the data is often represented as a series of aggregates, each containing all of the information for that entity.
- However, this can have a negative effect on queries. When a query only needs a subset of the data from some entities, such as a summary of orders for several customers without all of the order details, it must extract all of the data for the relevant entities in order to obtain the required information.

Materialized View pattern

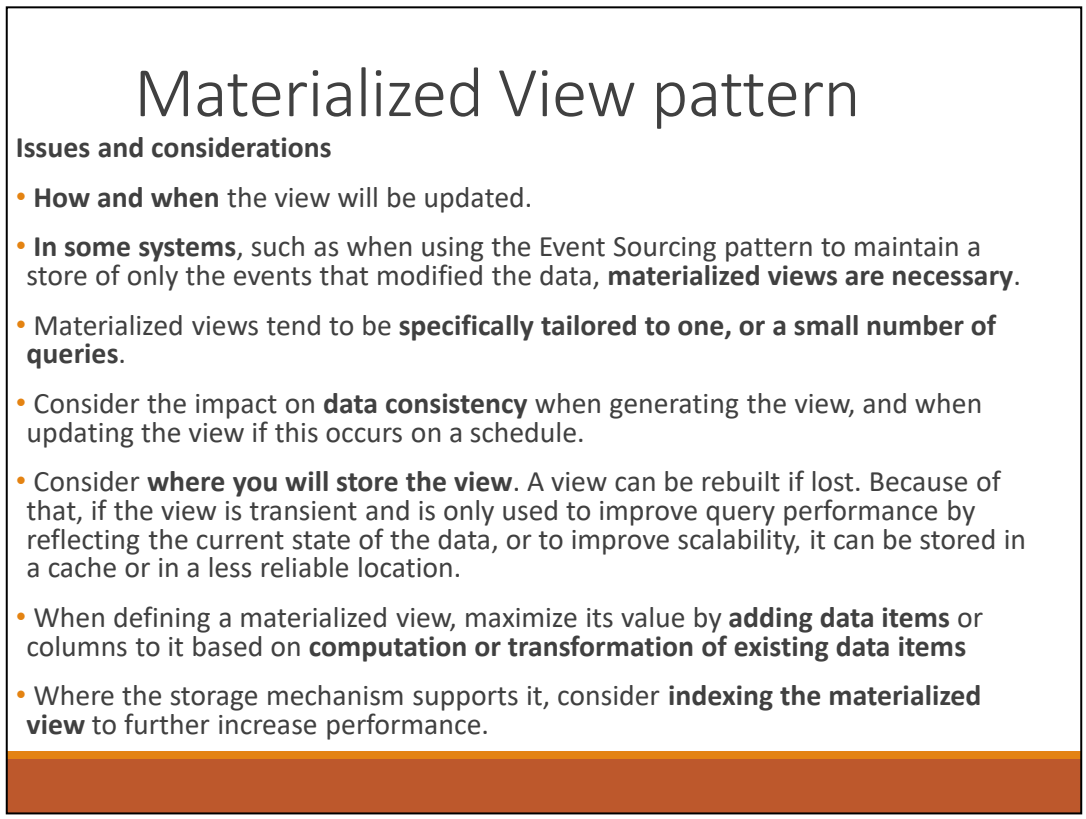
- To support efficient querying, a common solution is to generate, in advance, a view that materializes the data in a format suited to the required results set. The Materialized View pattern describes generating prepopulated views of data in environments where the source data isn't in a suitable format for querying, where generating a suitable query is difficult, or where query performance is poor due to the nature of the data or the data store.
- These materialized views, which only contain data required by a query, allow applications to quickly obtain the information they need.



In addition to joining tables or combining data entities, materialized views can include the current values of calculated columns or data items, the results of combining values or executing transformations on the data items, and values specified as part of the query. **A materialized view can even be optimized for just a single query.**

A key point is that a materialized view and the data it contains is completely disposable because it can be entirely rebuilt from the source data stores. **A materialized view is never updated directly by an application, and so it's a specialized cache**.

When the source data for the view changes, the view must be updated to include the new information. You can schedule this to **happen automatically**, or when the system detects a change to the original data. In some cases it might be necessary to regenerate the view manually. The figure shows an example of how the Materialized View pattern might be used.



Issues and considerations

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

How and when the view will be updated. Ideally it'll regenerate in response to an event indicating a change to the source data, although this can lead to excessive overhead if the source data changes rapidly. Alternatively, consider using a scheduled task, an external trigger, or a manual action to regenerate the view.

In some systems, such as when using the Event Sourcing pattern to maintain a store of only the events that modified the data, materialized views are necessary. Prepopulating views by examining all events to determine the current state might be the only way to obtain information from the event store. If you're not using Event Sourcing, you need to consider whether a materialized view is helpful or not. Materialized views tend to be specifically tailored to one, or a small number of queries. If many queries are used, materialized views can result in unacceptable storage capacity requirements and storage cost.

Consider the impact on data consistency when generating the view, and when updating the view if this occurs on a schedule. If the source data is changing at the point when the view is generated, the copy of the data in the view won't be fully consistent with the original data.

Consider where you'll store the view. The view doesn't have to be located in the same store or partition as the original data. It can be a subset from a few different partitions combined.

A view can be rebuilt if lost. Because of that, if the view is transient and is only

used to improve query performance by reflecting the current state of the data, or to improve scalability, it can be stored in a cache or in a less reliable location.

When defining a materialized view, maximize its value by adding data items or columns to it based on computation or transformation of existing data items, on values passed in the query, or on combinations of these values when appropriate.

Where the storage mechanism supports it, consider indexing the materialized view to further increase performance. Most relational databases support indexing for views, as do big data solutions based on Apache Hadoop.

Materialized View pattern

This pattern is useful when:

- Creating materialized views over data that's **difficult to query** directly
- Creating temporary views that can dramatically **improve query performance**
- Supporting **occasionally connected or disconnected scenarios** where connection to the data store isn't always available.
- **Simplifying queries** and exposing data for experimentation in a way that doesn't require knowledge of the source data format.
- Providing access to specific subsets of the source data that, for **security or privacy reasons**, shouldn't be generally accessible.
- **Bridging different data stores**, to take advantage of their individual capabilities.
- When using **microservices**, you are recommended to keep them **loosely coupled, including their data storage**. Therefore, materialized views can help you consolidate data from your services.

This pattern is useful when:

•Creating materialized views over data that's difficult to query directly, or where queries must be very complex to extract data that's stored in a normalized, semi-structured, or unstructured way.

•Creating temporary views that can dramatically improve query performance, or can act directly as source views or data transfer objects for the UI, for reporting, or for display.

•Supporting occasionally connected or disconnected scenarios where connection to the data store isn't always available. The view can be cached locally in this case.

•Simplifying queries and exposing data for experimentation in a way that doesn't require knowledge of the source data format. For example, by joining different tables in one or more databases, or one or more domains in NoSQL stores, and then formatting the data to fit its eventual use.

•Providing access to specific subsets of the source data that, for security or privacy reasons, shouldn't be generally accessible, open to modification, or fully exposed to users.

•Bridging different data stores, to take advantage of their individual capabilities. For example, using a cloud store that's efficient for writing as the reference data store, and a relational database that offers good query and read performance to hold the materialized views.

•When using microservices, you are recommended to keep them loosely coupled,

including their data storage. Therefore, materialized views can help you consolidate data from your services. If materialized views are not appropriate in your microservices architecture or specific scenario, please consider having well-defined boundaries that align to domain driven design (DDD) and aggregate their data when requested.

Materialized View pattern

This pattern isn't useful in the following situations:

- The source data is simple and easy to query.
- The source data changes very quickly, or can be accessed without using a view. In these cases, you should avoid the processing overhead of creating views.
- Consistency is a high priority. The views might not always be fully consistent with the original data.

Sharding pattern

Divide a data store into a set of **horizontal partitions or shards**. This can improve scalability when storing and accessing large volumes of data.

Context and problem

- **Storage space**. A data store for a large-scale cloud application is expected to contain a huge volume of data that could increase significantly over time.
- **Computing resources**. A cloud application is required to support a large number of concurrent users, each of which run queries that retrieve information from the data store.
- **Network bandwidth**. It's possible that the volume of network traffic might exceed the capacity of the network used to connect to the server, resulting in failed requests.
- **Geography**. It might be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons, or to reduce latency of data access.

A data store hosted by a single server might be subject to the following limitations:

•**Storage space**. A data store for a large-scale cloud application is expected to contain a huge volume of data that could increase significantly over time. A server typically provides only a finite amount of disk storage, but you can replace existing disks with larger ones, or add further disks to a machine as data volumes grow. However, the system will eventually reach a limit where it isn't possible to easily increase the storage capacity on a given server.

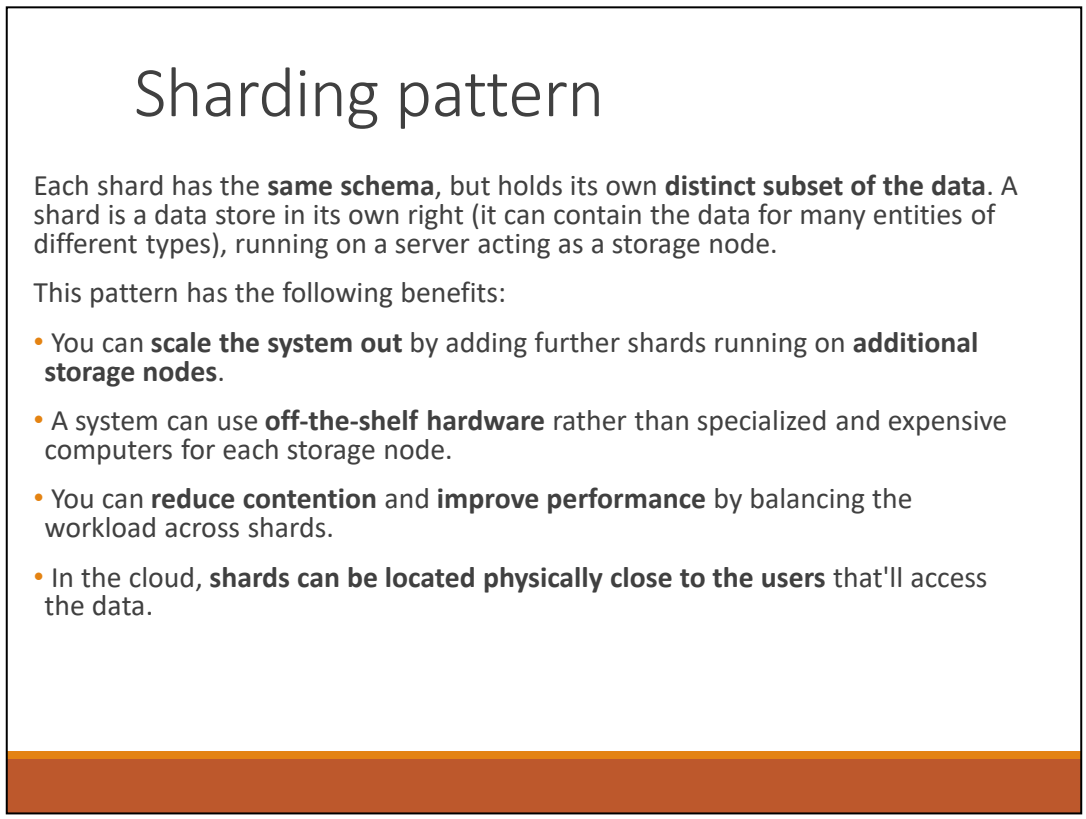
•**Computing resources**. A cloud application is required to support a large number of concurrent users, each of which run queries that retrieve information from the data store. A single server hosting the data store might not be able to provide the necessary computing power to support this load, resulting in extended response times for users and frequent failures as applications attempting to store and retrieve data time out. It might be possible to add memory or upgrade processors, but the system will reach a limit when it isn't possible to increase the compute resources any further.

•**Network bandwidth**. Ultimately, the performance of a data store running on a single server is governed by the rate the server can receive requests and send replies. It's possible that the volume of network traffic might exceed the capacity of the network used to connect to the server, resulting in failed requests.

•**Geography**. It might be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons, or to reduce latency of data access. If the users are dispersed across different

countries or regions, it might not be possible to store the entire data for the application in a single data store.

Scaling vertically by adding more disk capacity, processing power, memory, and network connections can postpone the effects of some of these limitations, but it's likely to only be a temporary solution. A commercial cloud application capable of supporting large numbers of users and high volumes of data must be able to scale almost indefinitely, so vertical scaling isn't necessarily the best solution.



When dividing a data store up into shards, decide which data should be placed in each shard. A shard typically contains items that fall within a specified range determined by one or more attributes of the data. These attributes form the shard key (sometimes referred to as the partition key). The shard key should be static. It shouldn't be based on data that might change.

Sharding physically organizes the data. When an application stores and retrieves data, the sharding logic directs the application to the appropriate shard. This sharding logic can be implemented as part of the data access code in the application, or it could be implemented by the data storage system if it transparently supports sharding.

Abstracting the physical location of the data in the sharding logic provides a high level of control over which shards contain which data. It also enables data to migrate between shards without reworking the business logic of an application if the data in the shards need to be redistributed later (for example, if the shards become unbalanced). The tradeoff is the additional data access overhead required in determining the location of each data item as it's retrieved.

To ensure optimal performance and scalability, it's important to split the data in a way that's appropriate for the types of queries that the application performs. In many cases, it's unlikely that the sharding scheme will exactly match the requirements of every query. For example, in a multi-tenant system an application might need to retrieve tenant data using the tenant ID, but it might also need to look up this data based on some other attribute such as the tenant's name or location. To handle these situations, implement a sharding strategy with a shard

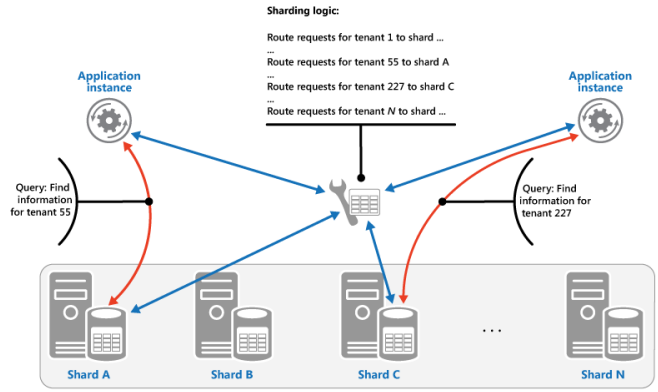
key that supports the most commonly performed queries.

Sharding pattern

Sharding strategies

Three strategies are commonly used when selecting the shard key and deciding how to distribute data across shards.

The Lookup strategy. In this strategy the **sharding logic** implements a **map** that **routes** a **request** for data to the shard that contains that data using the **shard key**.



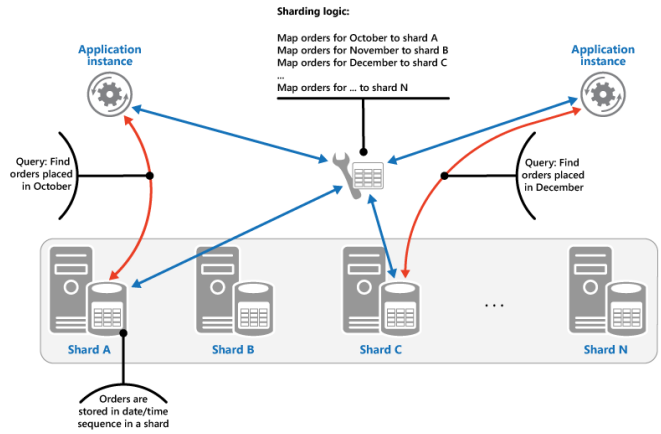
Note that there doesn't have to be a one-to-one correspondence between shards and the servers that host them—a single server can host multiple shards.

In this strategy the sharding logic implements a map that routes a request for data to the shard that contains that data using the shard key. In a multi-tenant application all the data for a tenant might be stored together in a shard using the tenant ID as the shard key. Multiple tenants might share the same shard, but the data for a single tenant won't be spread across multiple shards. The figure illustrates sharding tenant data based on tenant IDs.

The mapping between the shard key and the physical storage can be based on physical shards where each shard key maps to a physical partition. Alternatively, a more flexible technique for rebalancing shards is virtual partitioning, where shard keys map to the same number of virtual shards, which in turn map to fewer physical partitions. In this approach, an application locates data using a shard key that refers to a virtual shard, and the system transparently maps virtual shards to physical partitions. The mapping between a virtual shard and a physical partition can change without requiring the application code be modified to use a different set of shard keys.

Sharding pattern

The Range strategy. This strategy groups related items together in the same shard, and orders them by shard key—**the shard keys are sequential**. It's useful for applications that frequently retrieve sets of items using range queries (queries that return a set of data items for a shard key that falls within a given range).

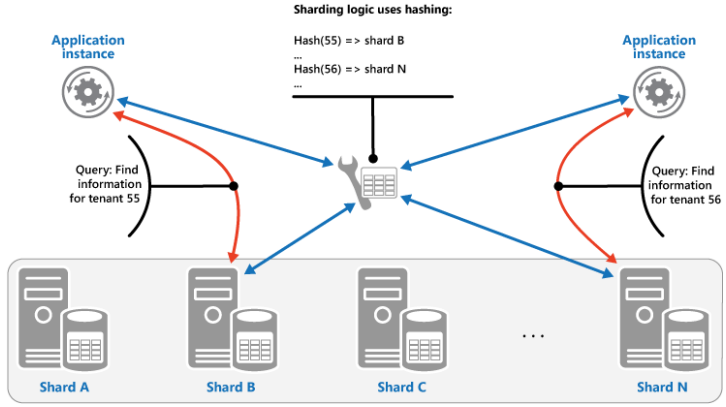


For example, if an application regularly needs to find all orders placed in a given month, this data can be retrieved more quickly if all orders for a month are stored in date and time order in the same shard. If each order was stored in a different shard, they'd have to be fetched individually by performing a large number of point queries (queries that return a single data item). The next figure illustrates storing sequential sets (ranges) of data in shard.

In this example, the shard key is a composite key containing the order month as the most significant element, followed by the order day and the time. The data for orders is naturally sorted when new orders are created and added to a shard. Some data stores support two-part shard keys containing a partition key element that identifies the shard and a row key that uniquely identifies an item in the shard. Data is usually held in row key order in the shard. Items that are subject to range queries and need to be grouped together can use a shard key that has the same value for the partition key but a unique value for the row key.

Sharding pattern

The Hash strategy. The purpose of this strategy is to reduce the chance of hotspots (shards that receive a disproportionate amount of load). It distributes the data across the shards in a way that achieves a balance between the size of each shard and the average load that each shard will encounter. The sharding logic computes the shard to store an item in based on a hash of one or more attributes of the data.



The chosen hashing function should distribute data evenly across the shards, possibly by introducing some random element into the computation. The next figure illustrates sharding tenant data based on a hash of tenant IDs.

To understand the advantage of the Hash strategy over other sharding strategies, consider how a multi-tenant application that enrolls new tenants sequentially might assign the tenants to shards in the data store. When using the Range strategy, the data for tenants 1 to n will all be stored in shard A, the data for tenants n+1 to m will all be stored in shard B, and so on. If the most recently registered tenants are also the most active, most data activity will occur in a small number of shards, which could cause hotspots. In contrast, the Hash strategy allocates tenants to shards based on a hash of their tenant ID. This means that sequential tenants are most likely to be allocated to different shards, which will distribute the load across them. The previous figure shows this for tenants 55 and 56.

Sharding pattern

The three sharding strategies have the following advantages and considerations:

- **Lookup**. This offers **more control over the way that shards are configured** and used. Using virtual shards **reduces the impact when rebalancing data because new physical partitions can be added** to even out the workload. The mapping between a virtual shard and the physical partitions that implement the shard can be modified without affecting application code that uses a shard key to store and retrieve data. Looking up shard locations can impose an additional overhead.
- **Range**. This is **easy to implement** and works well with range queries because they can often fetch multiple data items from a single shard in a single operation. This strategy offers **easier data management**. For example, if users in the same region are in the same shard, updates can be scheduled in each time zone based on the local load and demand pattern. However, this strategy **doesn't provide optimal balancing between shards**. **Rebalancing shards is difficult** and might not resolve the problem of uneven load if the majority of activity is for adjacent shard keys.
- **Hash**. This strategy offers a **better chance of more even data and load distribution**. Request routing can be accomplished directly by using the hash function. There's no need to maintain a map. Note that computing the hash might impose an additional overhead. Also, **rebalancing shards is difficult**.

Most common sharding systems implement one of the approaches described above, but you should also consider the business requirements of your applications and their patterns of data usage. For example, in a multi-tenant application:

You can shard data based on workload. You could segregate the data for highly volatile tenants in separate shards. The speed of data access for other tenants might be improved as a result.

You can shard data based on the location of tenants. You can take the data for tenants in a specific geographic region offline for backup and maintenance during off-peak hours in that region, while the data for tenants in other regions remains online and accessible during their business hours.

High-value tenants could be assigned their own private, high performing, lightly loaded shards, whereas lower-value tenants might be expected to share more densely-packed, busy shards.

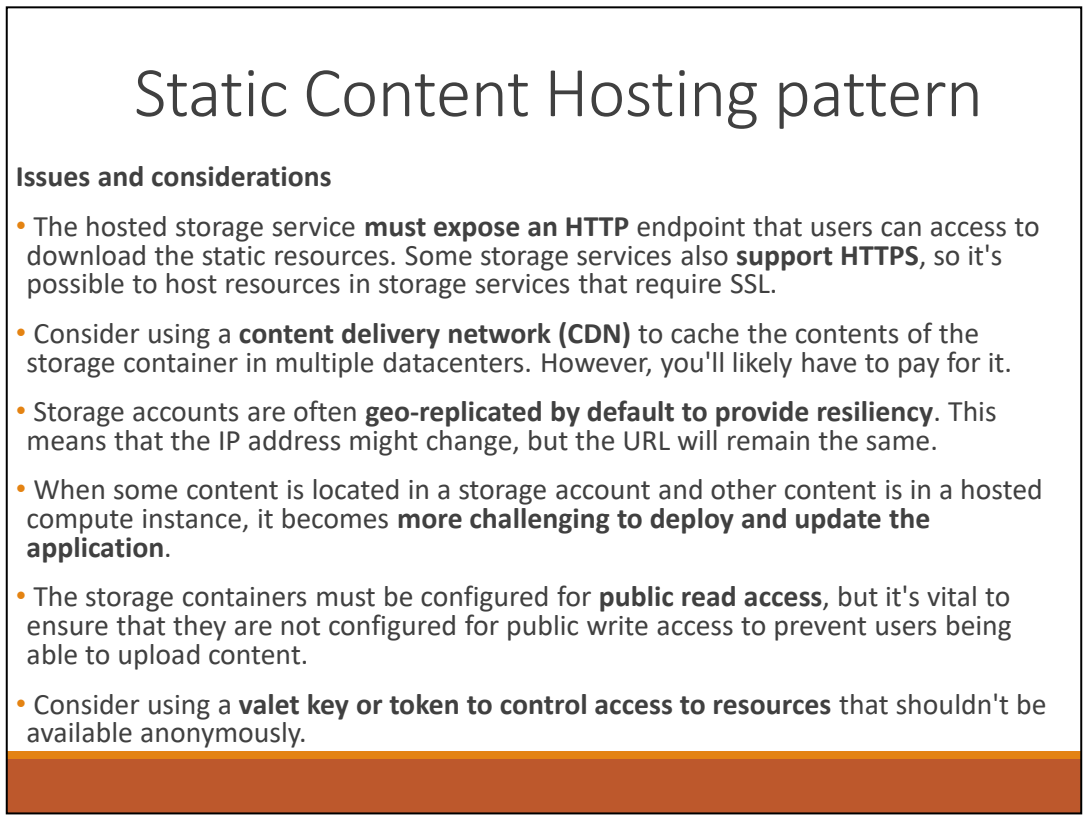
The data for tenants that need a high degree of data isolation and privacy can be stored on a completely separate server.

Static Content Hosting pattern

Deploy static content to a cloud-based storage service that can deliver them directly to the client. This can reduce the need for potentially expensive compute instances.

Context and problem

- Web applications typically include some elements of static content. This static content might include HTML pages and other resources such as images and documents that are available to the client, either as part of an HTML page (such as inline images, style sheets, and client-side JavaScript files) or as separate downloads (such as PDF documents).
- Although web servers are optimized for dynamic rendering and output caching, they still have to handle requests to download static content. This consumes processing cycles that could often be put to better use.



Issues and considerations

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

•The hosted storage service must expose an HTTP endpoint that users can access to download the static resources. Some storage services also support HTTPS, so it's possible to host resources in storage services that require SSL.

•For maximum performance and availability, consider using a content delivery network (CDN) to cache the contents of the storage container in multiple datacenters around the world. However, you'll likely have to pay for using the CDN.

•Storage accounts are often geo-replicated by default to provide resiliency against events that might affect a datacenter. This means that the IP address might change, but the URL will remain the same.

•When some content is located in a storage account and other content is in a hosted compute instance, it becomes more challenging to deploy and update the application. You might have to perform separate deployments, and version the application and content to manage it more easily—especially when the static content includes script files or UI components. However, if only static resources have to be updated, they can simply be uploaded to the storage account without needing to redeploy the application package.

•Storage services might not support the use of custom domain names. In this case it's necessary to specify the full URL of the resources in links because they'll be in a different domain from the dynamically-generated content containing the

links.

•The storage containers must be configured for public read access, but it's vital to ensure that they aren't configured for public write access to prevent users being able to upload content.

•Consider using a valet key or token to control access to resources that shouldn't be available anonymously. See the Valet Key pattern for more information.

Static Content Hosting pattern

This pattern is useful for:

- Minimizing the **hosting cost** for websites and applications that contain some **static resources**.
- Minimizing the hosting cost for websites that consist of only static content and resources. Depending on the capabilities of the hosting provider's storage system, it might be possible to entirely host a **fully static website in a storage account**.
- Exposing static resources and content for applications running in other hosting environments or on-premises servers.
- Locating content in more than one geographical area using a **content delivery network** that caches the contents of the storage account in multiple datacenters around the world.
- **Monitoring costs and bandwidth usage**. Using a separate storage account for some or all of the static content allows the costs to be more easily separated from hosting and runtime costs.

Static Content Hosting pattern

This pattern might not be useful in the following situations:

- The application needs to **perform some processing** on the static content before delivering it to the client. For example, it might be necessary to add a timestamp to a document.
- The **volume of static content is very small**. The overhead of retrieving this content from separate storage can outweigh the cost benefit of separating it out from the compute resource.

Valet Key pattern

Use a **token that provides clients with restricted direct access to a specific resource**, in order to offload data transfer from the application. This is particularly useful in applications that use cloud-hosted storage systems or queues, and can minimize cost and maximize scalability and performance.

Context and problem

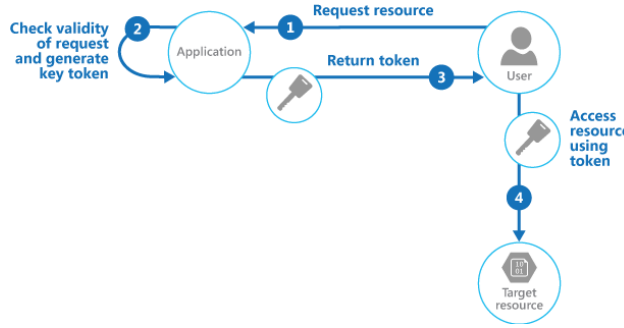
- Client programs and web browsers often need to read and write files or data streams to and from an application's storage. Typically, the application will handle the movement of the data — either by fetching it from storage and streaming it to the client, or by reading the uploaded stream from the client and storing it in the data store. However, this approach absorbs valuable resources such as compute, memory, and bandwidth.

Data stores have the ability to handle upload and download of data directly, without requiring that the application perform any processing to move this data. But, this typically requires the client to have access to the security credentials for the store. This can be a useful technique to minimize data transfer costs and the requirement to scale out the application, and to maximize performance. It means, though, that the application is no longer able to manage the security of the data. After the client has a connection to the data store for direct access, the application can't act as the gatekeeper. It's no longer in control of the process and can't prevent subsequent uploads or downloads from the data store.

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

Valet Key pattern

- You need to resolve the problem of controlling access to a data store where the store can't manage authentication and authorization of clients. One typical solution is to restrict access to the data store's public connection and provide the client with a key or token that the data store can validate.
- This key or token is usually referred to as a valet key. It provides **time-limited access to specific resources and allows only predefined operations** such as reading and writing to storage or queues, or uploading and downloading in a web browser.



Applications can create and issue valet keys to client devices and web browsers quickly and easily, allowing clients to perform the required operations without requiring the application to directly handle the data transfer. This removes the processing overhead, and the impact on performance and scalability, from the application and the server.

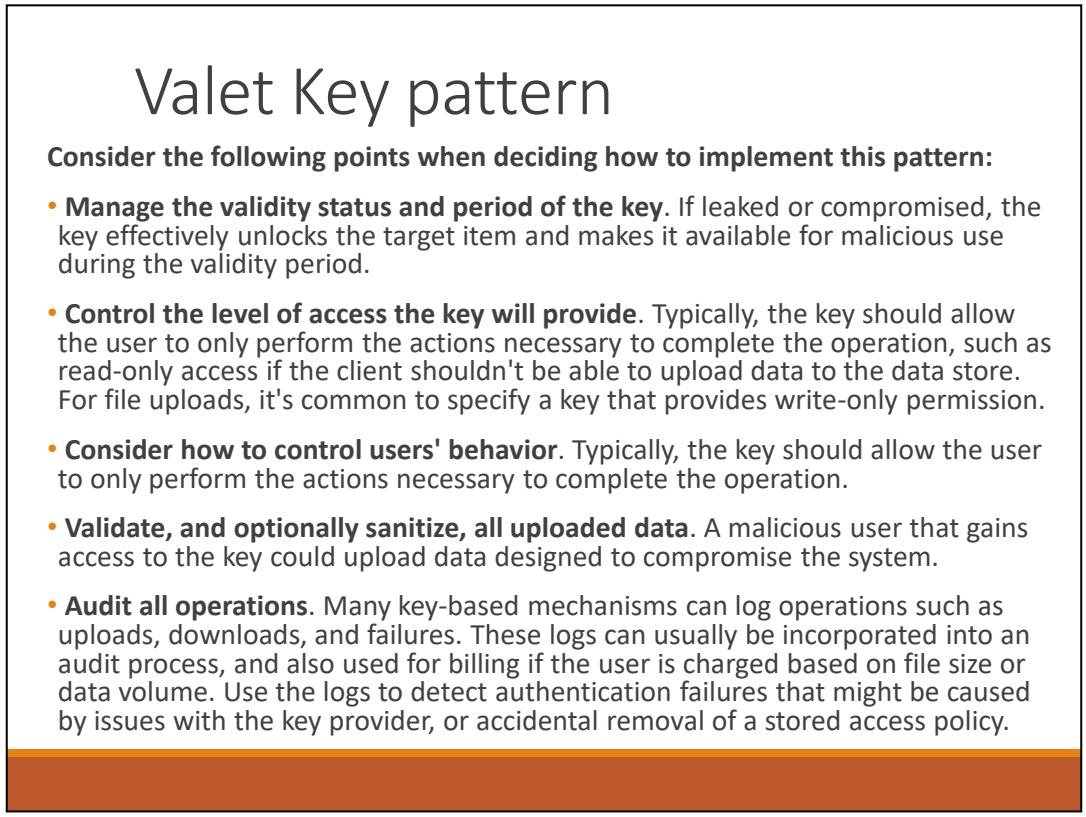
The client uses this token to access a specific resource in the data store for only a specific period, and with specific restrictions on access permissions, as shown in the figure. After the specified period, the key becomes invalid and won't allow access to the resource.

It's also possible to configure a key that has other dependencies, such as the scope of the data. For example, depending on the data store capabilities, the key can specify a complete table in a data store, or only specific rows in a table. In cloud storage systems the key can specify a container, or just a specific item within a container.

The key can also be invalidated by the application. This is a useful approach if the client notifies the server that the data transfer operation is complete. The server can then invalidate that key to prevent further access.

Using this pattern can simplify managing access to resources because there's no requirement to create and authenticate a user, grant permissions, and then remove the user again. It also makes it easy to limit the location, the permission, and the validity period—all by simply generating a key at runtime. The important factors are to limit the validity period, and especially the location of the resource,

as tightly as possible so that the recipient can only use it for the intended purpose.



Manage the validity status and period of the key. If leaked or compromised, the key effectively unlocks the target item and makes it available for malicious use during the validity period. A key can usually be revoked or disabled, depending on how it was issued. Server-side policies can be changed or, the server key it was signed with can be invalidated. Specify a short validity period to minimize the risk of allowing unauthorized operations to take place against the data store. However, if the validity period is too short, the client might not be able to complete the operation before the key expires. Allow authorized users to renew the key before the validity period expires if multiple accesses to the protected resource are required.

Control the level of access the key will provide. Typically, the key should allow the user to only perform the actions necessary to complete the operation, such as read-only access if the client shouldn't be able to upload data to the data store. For file uploads, it's common to specify a key that provides write-only permission, as well as the location and the validity period. It's critical to accurately specify the resource or the set of resources to which the key applies.

Consider how to control users' behavior. Implementing this pattern means some loss of control over the resources users are granted access to. The level of control that can be exerted is limited by the capabilities of the policies and permissions available for the service or the target data store. For example, it's usually not possible to create a key that limits the size of the data to be written to storage, or the number of times the key can be used to access a file. This can result in huge unexpected costs for data transfer, even when used by the intended client, and might be caused by an error in the code that causes

repeated upload or download. To limit the number of times a file can be uploaded, where possible, force the client to notify the application when one operation has completed. For example, some data stores raise events the application code can use to monitor operations and control user behavior. However, it's hard to enforce quotas for individual users in a multi-tenant scenario where the same key is used by all the users from one tenant.

Validate, and optionally sanitize, all uploaded data. A malicious user that gains access to the key could upload data designed to compromise the system. Alternatively, authorized users might upload data that's invalid and, when processed, could result in an error or system failure. To protect against this, ensure that all uploaded data is validated and checked for malicious content before use.

Audit all operations. Many key-based mechanisms can log operations such as uploads, downloads, and failures. These logs can usually be incorporated into an audit process, and also used for billing if the user is charged based on file size or data volume. Use the logs to detect authentication failures that might be caused by issues with the key provider, or accidental removal of a stored access policy.

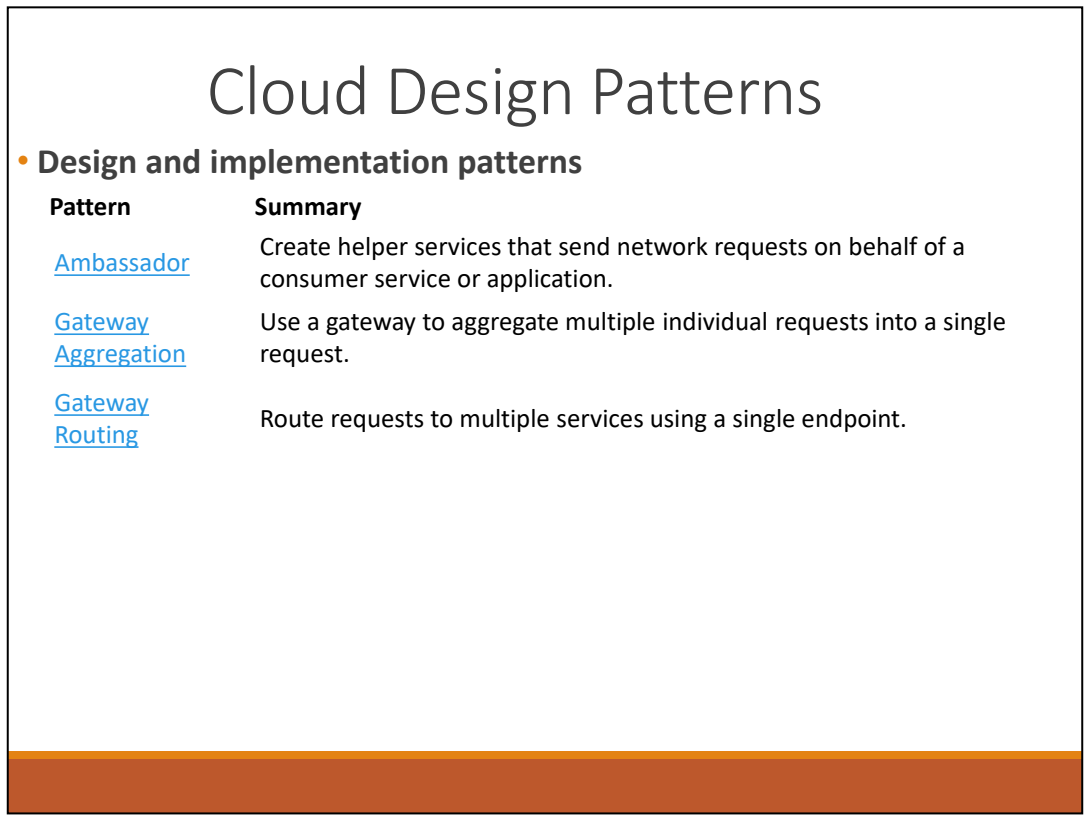
Deliver the key securely. It can be embedded in a URL that the user activates in a web page, or it can be used in a server redirection operation so that the download occurs automatically. Always use HTTPS to deliver the key over a secure channel.

Protect sensitive data in transit. Sensitive data delivered through the application will usually take place using TLS, and this should be enforced for clients accessing the data store directly.

Valet Key pattern

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

- **Deliver the key securely**. It can be embedded in a URL that the user activates in a web page, or it can be used in a server redirection operation so that the download occurs automatically. Always use HTTPS to deliver the key over a secure channel.
- **Protect sensitive data in transit**. Sensitive data delivered through the application will usually take place using TLS, and this should be enforced for clients accessing the data store directly.



https://learn.microsoft.com/en-us/azure/architecture/patterns/

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

Ambassador pattern

Create **helper services** that send network requests on behalf of a consumer service or application. An ambassador service can be thought of as an out-of-process proxy that is co-located with the client.

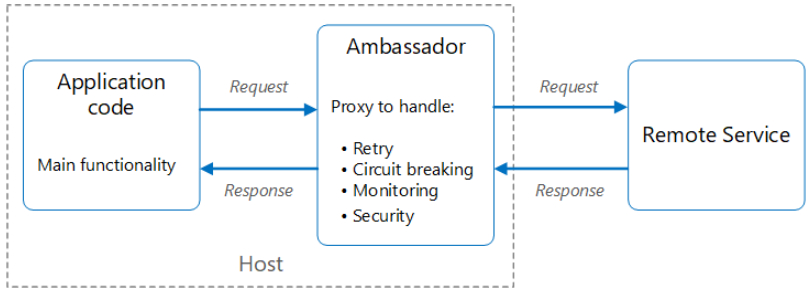
Context and problem

- Resilient cloud-based applications require features such as **circuit breaking**, **routing**, **metering and monitoring**, and the ability to make **network-related configuration updates**. It may be difficult or impossible to update legacy applications or existing code libraries to add these features, because the code is no longer maintained or can't be easily modified by the development team.
- Network calls may also require substantial configuration for **connection, authentication, and authorization**. If these calls are used across multiple applications, built using multiple languages and frameworks, the calls must be configured for each of these instances. In addition, network and security functionality may need to be managed by a central team within your organization. With a large code base, it can be risky for that team to update application code they aren't familiar with.

This pattern can be useful for offloading common client connectivity tasks such as monitoring, logging, routing, security (such as TLS), and resiliency patterns in a language agnostic way. It is often used with legacy applications, or other applications that are difficult to modify, in order to extend their networking capabilities. It can also enable a specialized team to implement those features.

Ambassador pattern

Put client frameworks and libraries into an external process that acts as a proxy between your application and external services. Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions.



ou can also use the ambassador pattern to standardize and extend instrumentation. The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application.

Features that are offloaded to the ambassador can be managed independently of the application. You can update and modify the ambassador without disturbing the application's legacy functionality. It also allows for separate, specialized teams to implement and maintain security, networking, or authentication features that have been moved to the ambassador.

Ambassador services can be deployed as a sidecar to accompany the lifecycle of a consuming application or service. Alternatively, if an ambassador is shared by multiple separate processes on a common host, it can be deployed as a daemon or Windows service. If the consuming service is containerized, the ambassador should be created as a separate container on the same host, with the appropriate links configured for communication.

Ambassador pattern

Issues and considerations

- The proxy adds some **latency overhead**. Consider whether a client library, invoked directly by the application, is a better approach.
- Consider the possible impact of including generalized features in the proxy. For example, **the ambassador could handle retries**, but that might **not be safe unless all operations are idempotent**.
- Consider a mechanism to **allow the client to pass some context to the proxy**, as well as back to the client. For example, include HTTP request headers to opt out of retry or specify the maximum number of times to retry.
- Consider how you will **package and deploy** the proxy.
- Consider whether to use a single shared instance for all clients or an instance for each client.

An idempotent operation is one that has no additional effect if it is called more than once with the same input parameters. For example, removing an item from a set can be considered an idempotent operation on the set.

Ambassador pattern

Use this pattern when you:

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

This pattern may not be suitable:

- When network request latency is critical. A proxy will introduce some overhead, although minimal, and in some cases this may affect the application.
- When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
- When connectivity features cannot be generalized and require deeper integration with the client application.

Ambassador pattern

Example

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

This pattern may not be suitable:

- When network request latency is critical. A proxy will introduce some overhead, although minimal, and in some cases this may affect the application.
- When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
- When connectivity features cannot be generalized and require deeper integration with the client application.

Gateway Aggregation pattern

Use a gateway to **aggregate multiple individual requests into a single request**. This pattern is useful when **a client must make multiple calls to different backend systems to perform an operation**.

Context and problem

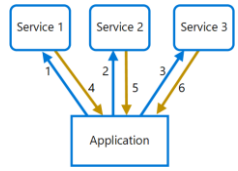
- To perform a single task, a client may have to make multiple calls to various backend services. An application that relies on many services to perform a task must expend resources on each request. When any new feature or service is added to the application, additional requests are needed, further increasing resource requirements and network calls. This chattiness between a client and a backend can adversely impact the performance and scale of the application.
- Microservice architectures have made this problem more common, as applications built around many smaller services naturally have a higher amount of cross-service calls.

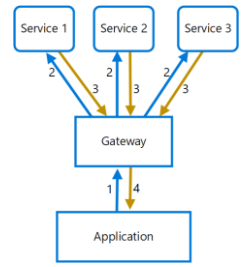
Gateway Aggregation pattern

- Use a gateway to reduce chattiness between the client and the services. The gateway receives client requests, dispatches requests to the various backend systems, and then aggregates the results and sends them back to the requesting client.
- This pattern can **reduce the number of requests that the application makes to backend services**, and improve application performance over high-latency networks.

Gateway Aggregation pattern

In the following diagram, the application sends a request to the gateway (1). The request contains a package of additional requests. The gateway decomposes these and processes each request by sending it to the relevant service (2). Each service returns a response to the gateway (3). The gateway combines the responses from each service and sends the response to the application (4). The application makes a single request and receives only a single response from the gateway.





Gateway Aggregation pattern

Issues and considerations

- The gateway should not introduce service coupling across the backend services.
- The gateway should be located near the backend services to reduce latency as much as possible.
- The gateway service may introduce a single point of failure. Ensure the gateway is properly designed to meet your application's availability requirements.
- The gateway may introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can be scaled to meet your anticipated growth.
- Perform load testing against the gateway to ensure you don't introduce cascading failures for services.
- If one or more service calls takes too long, it may be acceptable to timeout and return a partial set of data. Consider how your application will handle this scenario.

•Use asynchronous I/O to ensure that a delay at the backend doesn't cause performance issues in the application.

•Implement a resilient design, using techniques such as bulkheads, circuit breaking, retry, and timeouts.

•ement distributed tracing using correlation IDs to track each individual call.

•Monitor request metrics and response sizes.

•Consider returning cached data as a failover strategy to handle failures.

•Instead of building aggregation into the gateway, consider placing an aggregation service behind the gateway. Request aggregation will likely have different resource requirements than other services in the gateway and may impact the gateway's routing and offloading functionality

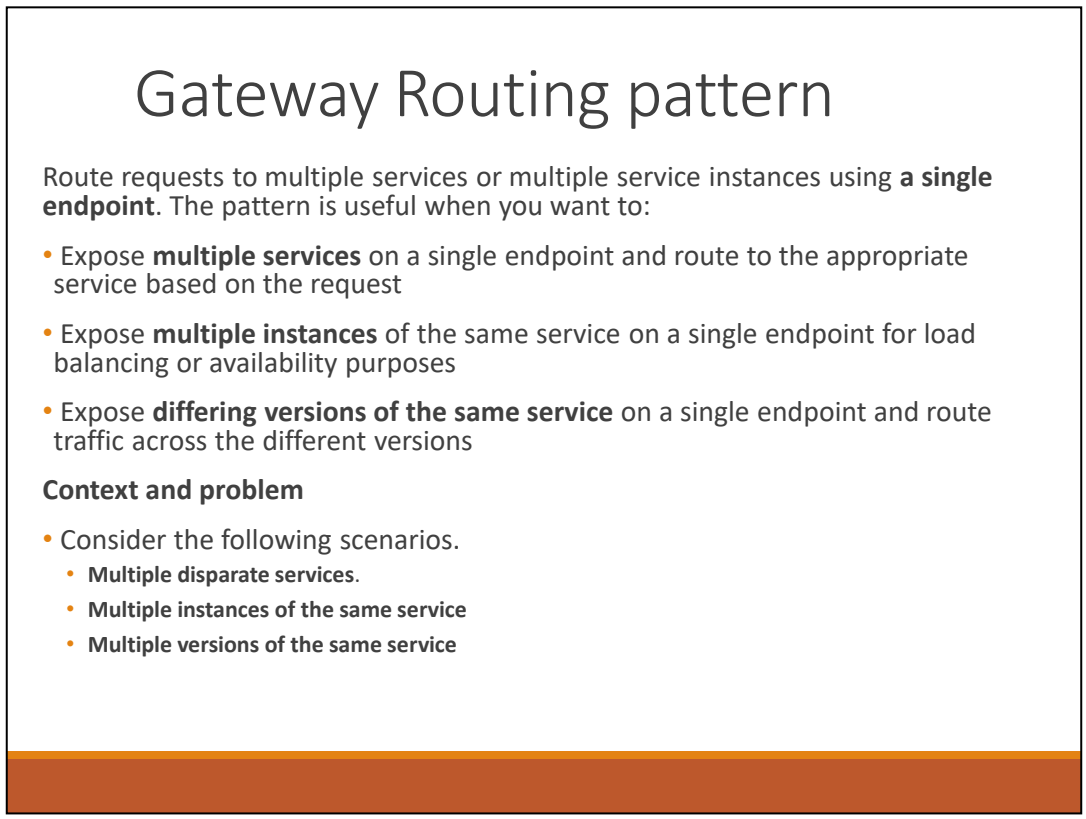
Gateway Aggregation pattern

Use this pattern when:

- A client needs to communicate with multiple backend services to perform an operation.
- The client may use networks with significant latency, such as cellular networks.

This pattern may not be suitable when:

- You want to reduce the number of calls between a client and a single service across multiple operations. In that scenario, it may be better to add a batch operation to the service.
- The client or application is located near the backend services and latency is not a significant factor.

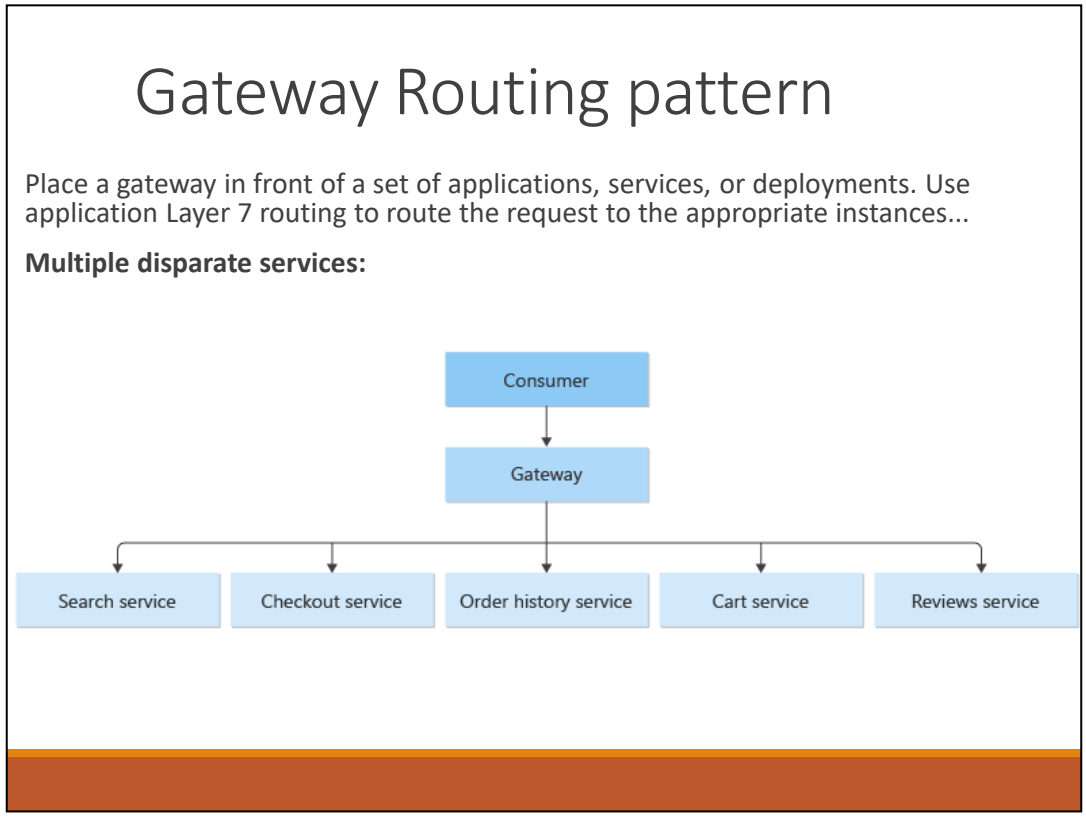


When a client needs to consume multiple services, multiple service instances or a combination of both, the client must be updated when services are added or removed. Consider the following scenarios.

•**Multiple disparate services** - An e-commerce application might provide services such as search, reviews, cart, checkout, and order history. Each service has a different API that the client must interact with, and the client must know about each endpoint in order to connect to the services. If an API changes, the client must be updated as well. If you refactor a service into two or more separate services, the code must change in both the service and the client.

•**Multiple instances of the same service** - The system can require running multiple instances of the same service in the same or different regions. Running multiple instances can be done for load balancing purposes or to meet availability requirements. Each time an instance is spun up or down to match demand, the client must be updated.

•**Multiple versions of the same service** - As part of the deployment strategy, new versions of a service can be deployed along side existing versions. This is known as blue green deployments. In these scenarios, the client must be updated each time there are changes to the percentage of traffic being routed to the new version and existing endpoint.



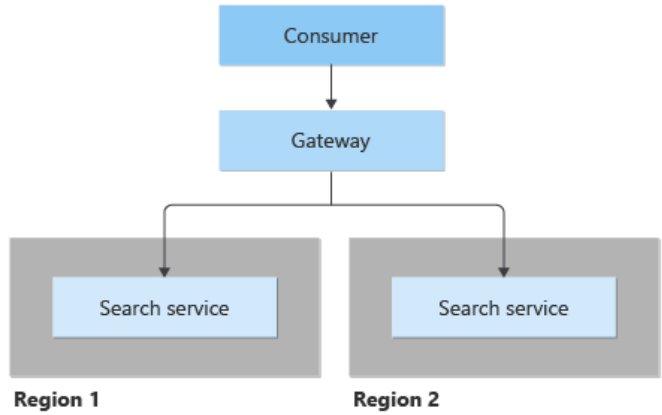
With this pattern, the client application only needs to know about a single endpoint and communicate with a single endpoint. The following illustrate how the Gateway Routing pattern addresses the three scenarios outlined in the context and problem section.

The gateway routing pattern is useful in this scenario where a client is consuming multiple services. If a service is consolidated, decomposed or replaced, the client doesn't necessarily require updating. It can continue making requests to the gateway, and only the routing changes.

A gateway also lets you abstract backend services from the clients, allowing you to keep client calls simple while enabling changes in the backend services behind the gateway. Client calls can be routed to whatever service or services need to handle the expected client behavior, allowing you to add, split, and reorganize services behind the gateway without changing the client.

Gateway Routing pattern

Multiple instances of the same service:

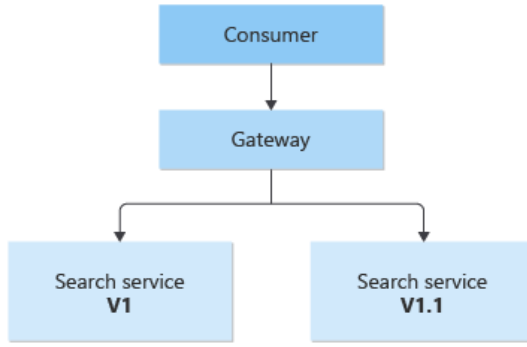


Elasticity is key in cloud computing. Services can be spun up to meet increasing demand or spun down when demand is low to save money. The complexity of registering and unregistering service instances is encapsulated in the gateway. The client is unaware of an increase or decreases in the number of services.

Service instances can be deployed in a single or multiple regions. The Geode pattern details how a multi-region, active-active deployment can improve latency and increase availability of a service.

Gateway Routing pattern

- **Multiple versions of the same service**



This pattern can be used for deployments, by allowing you to manage how updates are rolled out to users. When a new version of your service is deployed, it can be deployed in parallel with the existing version. Routing lets you control what version of the service is presented to the clients, giving you the flexibility to use various release strategies, whether incremental, parallel, or complete rollouts of updates. Any issues discovered after the new service is deployed can be quickly reverted by making a configuration change at the gateway, without affecting clients.

Gateway Routing pattern

Issues and considerations

- The gateway service can introduce **a single point of failure**.
- The gateway service can introduce **a bottleneck**.
- Perform **load testing against the gateway** to ensure you don't introduce cascading failures for services.
- Gateway routing is **level 7**. It can be based on IP, port, header, or URL.
- Gateway services can be **global or regional**.
- **The gateway service is the public endpoint for services it sits in front of**. Consider limiting public network access to the backend services, by making the services only accessible via the gateway or via a private virtual network**.**

•The gateway service can introduce a single point of failure. Ensure it's properly designed to meet your availability requirements. Consider resiliency and fault tolerance capabilities in the implementation.

•The gateway service can introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can easily scale in line with your growth expectations.

•Perform load testing against the gateway to ensure you don't introduce cascading failures for services.

•Gateway routing is level 7. It can be based on IP, port, header, or URL.

•Gateway services can be global or regional. Use a global gateway if your solution requires multi-region deployments of services. Consider using Application Gateway if you have a regional workload that requires granular control how traffic is balanced. For example, you want to balance traffic between virtual machines.

•The gateway service is the public endpoint for services it sits in front of. Consider limiting public network access to the backend services, by making the services only accessible via the gateway or via a private virtual network.

Gateway Routing pattern

Use this pattern when:

- A client needs to consume multiple services that can be accessed behind a gateway.
- You want to simplify client applications by using a single endpoint.
- You need to route requests from externally addressable endpoints to internal virtual endpoints, such as exposing ports on a VM to cluster virtual IP addresses.
- A client needs to consume services running in multiple regions for latency or availability benefits.
- A client needs to consume a variable number of service instances.
- You want to implement a deployment strategy where clients access multiple versions of the service at the same time.

•The gateway service can introduce a single point of failure. Ensure it's properly designed to meet your availability requirements. Consider resiliency and fault tolerance capabilities in the implementation.

•The gateway service can introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can easily scale in line with your growth expectations.

•Perform load testing against the gateway to ensure you don't introduce cascading failures for services.

•Gateway routing is level 7. It can be based on IP, port, header, or URL.

•Gateway services can be global or regional. Use a global gateway if your solution requires multi-region deployments of services. Consider using Application Gateway if you have a regional workload that requires granular control how traffic is balanced. For example, you want to balance traffic between virtual machines.

•The gateway service is the public endpoint for services it sits in front of. Consider limiting public network access to the backend services, by making the services only accessible via the gateway or via a private virtual network.

Cloud Design Patterns

Reliability

Pattern	Summary
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes, to smooth intermittent heavy loads.
Throttling	Control the consumption of resources by an instance of an application, an individual tenant, or an entire service.

Queue-Based Load Leveling pattern

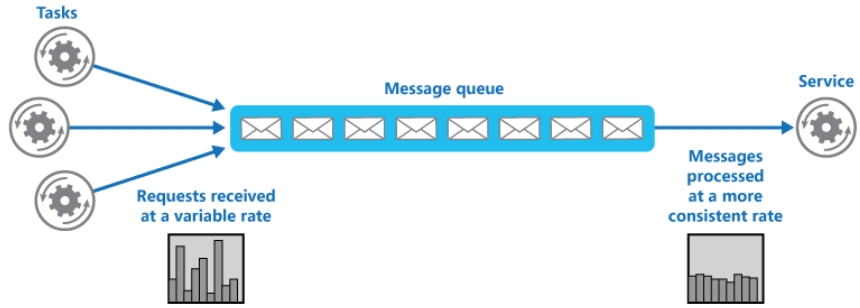
Use a **queue** that acts as a buffer between a task and a service it invokes in order to **smooth intermittent heavy loads** that can cause the service to fail or the task to time out. This can help to minimize the impact of peaks in demand on availability and responsiveness for both the task and the service.

Context and problem

- Many solutions in the cloud involve running tasks that invoke services. In this environment, if a service is subjected to intermittent heavy loads, it can cause **performance or reliability issues**.
- A service could be part of the same solution as the tasks that use it, or it could be a third-party service providing access to frequently used resources such as a cache or a storage service. If the same service is used by a number of tasks running concurrently, it can be **difficult to predict the volume of requests to the service at any time**.
- A service might experience peaks in demand that cause it to overload and be **unable to respond to requests in a timely manner**. Flooding a service with a large number of concurrent requests can also result in the service failing if it's unable to handle the contention these requests cause.

Queue-Based Load Leveling pattern

Refactor the solution and introduce a queue between the task and the service. The task and the service run asynchronously. The task posts a message containing the data required by the service to a queue. The queue acts as a buffer, storing the message until it's retrieved by the service. The service retrieves the messages from the queue and processes them. Requests from a number of tasks, which can be generated at a highly variable rate, can be passed to the service through the same message queue. This figure shows using a queue to level the load on a service.



The queue decouples the tasks from the service, and the service can handle the messages at its own pace regardless of the volume of requests from concurrent tasks. Additionally, there's no delay to a task if the service isn't available at the time it posts a message to the queue.

This pattern provides the following benefits:

•It can help to maximize availability because delays arising in services won't have an immediate and direct impact on the application, which can continue to post messages to the queue even when the service isn't available or isn't currently processing messages.

•It can help to maximize scalability because both the number of queues and the number of services can be varied to meet demand.

•It can help to control costs because the number of service instances deployed only have to be adequate to meet average load rather than the peak load.

•Some services implement throttling when demand reaches a threshold beyond which the system could fail. Throttling can reduce the functionality available. You can implement load leveling with these services to ensure that this threshold isn't reached.

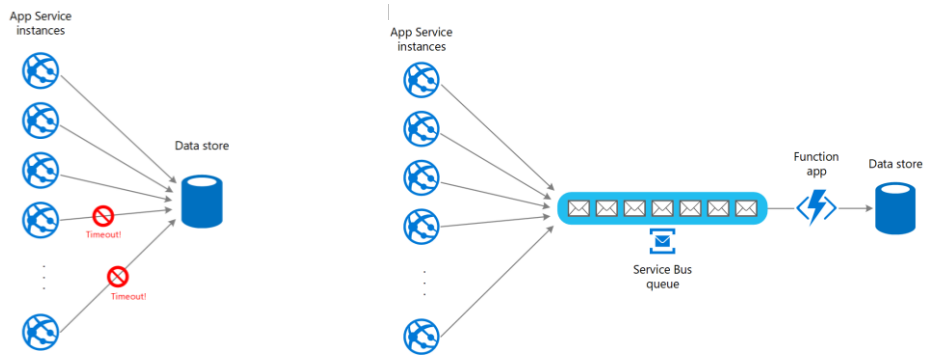
Queue-Based Load Leveling pattern

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

- It's necessary to implement application logic that controls the rate at which services handle messages to avoid overwhelming the target resource. Avoid passing spikes in demand to the next stage of the system. Test the system under load to ensure that it provides the required leveling, and adjust the number of queues and the number of service instances that handle messages to achieve this.
- Message queues are a one-way communication mechanism. If a task expects a reply from a service, it might be necessary to implement a mechanism that the service can use to send a response.
- Be careful if you apply autoscaling to services that are listening for requests on the queue. This can result in increased contention for any resources that these services share and diminish the effectiveness of using the queue to level the load.
- The pattern can lose information depending on the persistence of the Queue. If your queue crashes or drops information (due to system limits) there's a possibility that you don't have a guaranteed delivery. The behavior of your queue and system limits needs to be taken into consideration based on the needs of your solution.

Queue-Based Load Leveling pattern

- This pattern is useful to any application that uses services that are subject to overloading.
- This pattern is **not** useful if the application expects a response from the service with minimal latency.
- **Example:** A web app writes data to an external data store. If a large number of instances of the web app run concurrently, the data store might be unable to respond to requests quickly enough, causing requests to time out, be throttled, or otherwise fail. T



To resolve this, you can use a queue to level the load between the application instances and the data store. An Azure Functions app reads the messages from the queue and performs the read/write requests to the data store. The application logic in the function app can control the rate at which it passes requests to the data store, to prevent the store from being overwhelmed. (Otherwise the function app will just re-introduce the same problem at the back end.)

Throttling pattern

Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service. This can **allow the system to continue to function and meet service level agreements**, even when an increase in demand places an extreme load on resources.

Context and problem

- The, load on a cloud application typically varies over time based on the number of active users or the types of activities they are performing. For example, more users are likely to be active during **business hours** or the system might be required to perform **computationally expensive analytics at the end of each month**. There might also be sudden and unanticipated bursts in activity. If the processing requirements of the system exceed the capacity of the resources that are available, it will suffer from poor performance and can even fail. If the system has to meet an agreed level of service, such failure could be unacceptable.

There're many strategies available for handling varying load in the cloud, depending on the business goals for the application. One strategy is to use autoscaling to match the provisioned resources to the user needs at any given time. This has the potential to consistently meet user demand, while optimizing running costs. However, while autoscaling can trigger the provisioning of additional resources, this provisioning isn't immediate. If demand grows quickly, there can be a window of time where there's a resource deficit.

Throttling pattern

An **alternative strategy to autoscaling** is to allow applications to **use resources only up to a limit, and then throttle them when this limit is reached**. The system should monitor how it's using resources so that, when usage exceeds the threshold, it can **throttle requests from one or more users**. This will enable the system to continue functioning and meet any service level agreements (SLAs) that are in place. For more information on monitoring resource usage, see the Instrumentation and Telemetry Guidance..

The system could implement several throttling strategies, including:

- Rejecting requests from an individual user who's already accessed system APIs more than n times per second over a given period of time.
- Disabling or degrading the functionality of selected nonessential services.

The system could implement several throttling strategies, including:

•Rejecting requests from an individual user who's already accessed system APIs more than n times per second over a given period of time. This requires the system to meter the use of resources for each tenant or user running an application. For more information, see the Service Metering Guidance.

•Disabling or degrading the functionality of selected nonessential services so that essential services can run unimpeded with sufficient resources. For example, if the application is streaming video output, it could switch to a lower resolution.

Throttling pattern

The system could implement several throttling strategies, including:

- Using **load leveling** to smooth the volume of activity (this approach is covered in more detail by the Queue-based Load Leveling pattern). In a multi-tenant environment, this approach will reduce the performance for every tenant. Requests for other tenants can be held back, and handled when the backlog has eased. The Priority Queue pattern could be used to help implement this approach.
- **Deferring operations** being performed on behalf of lower priority applications or tenants.
  - You should **be careful when integrating with some 3rd-party services that might become unavailable or return errors**. Reduce the number of concurrent requests being processed so that the logs do not unnecessarily fill up with errors. You also avoid the costs that are associated with needlessly retrying the processing of requests that would fail because of that 3rd-party service. Then, when requests are processed successfully, go back to regular unthrottled request processing. One library that implements this functionality is NServiceBus.

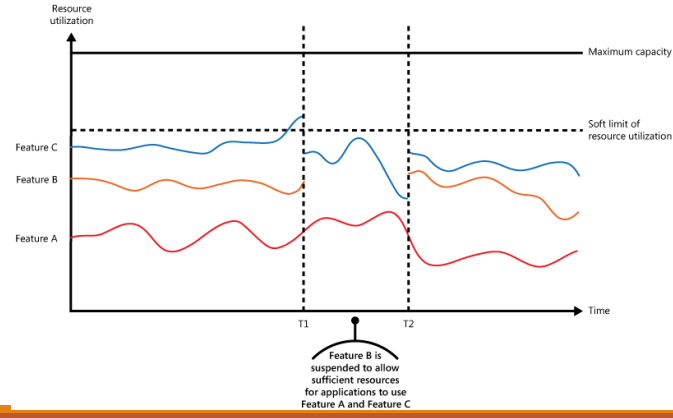
• Using load leveling to smooth the volume of activity (this approach is covered in more detail by the Queue-based Load Leveling pattern). In a multi-tenant environment, this approach will reduce the performance for every tenant. If the system must support a mix of tenants with different SLAs, the work for high-value tenants might be performed immediately. Requests for other tenants can be held back, and handled when the backlog has eased. The Priority Queue pattern could be used to help implement this approach.

•Deferring operations being performed on behalf of lower priority applications or tenants. These operations can be suspended or limited, with an exception generated to inform the tenant that the system is busy and that the operation should be retried later.

•You should be careful when integrating with some 3rd-party services that might become unavailable or return errors. Reduce the number of concurrent requests being processed so that the logs do not unnecessarily fill up with errors. You also avoid the costs that are associated with needlessly retrying the processing of requests that would fail because of that 3rd-party service. Then, when requests are processed successfully, go back to regular unthrottled request processing. One library that implements this functionality is NServiceBus.

Throttling pattern

The figure shows an area graph for resource use (a combination of memory, CPU, bandwidth, and other factors) against time for applications that are making use of three features. A feature is an area of functionality, such as a component that performs a specific set of tasks, a piece of code that performs a complex calculation, or an element that provides a service such as an in-memory cache. These features are labeled A, B, and C.

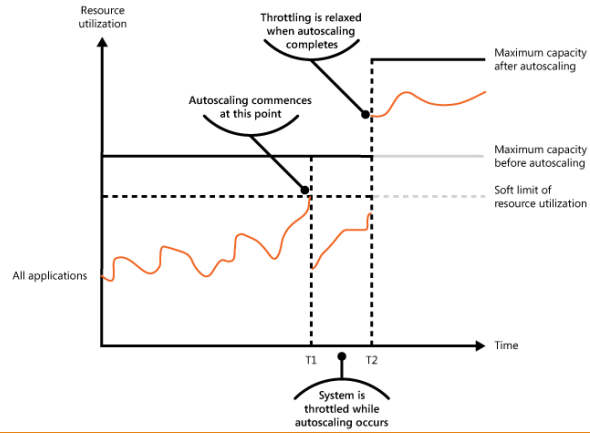


The area immediately below the line for a feature indicates the resources that are used by applications when they invoke this feature. For example, the area below the line for Feature A shows the resources used by applications that are making use of Feature A, and the area between the lines for Feature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

The previous figure illustrates the effects of deferring operations. Just prior to time T1, the total resources allocated to all applications using these features reach a threshold (the limit of resource use). At this point, the applications are in danger of exhausting the resources available. In this system, Feature B is less critical than Feature A or Feature C, so it's temporarily disabled and the resources that it was using are released. Between times T1 and T2, the applications using Feature A and Feature C continue running as normal. Eventually, the resource use of these two features diminishes to the point when, at time T2, there is sufficient capacity to enable Feature B again.

Throttling pattern

- The autoscaling and throttling approaches can also be combined to help keep the applications responsive and within SLAs. If the demand is expected to remain high, throttling provides a temporary solution while the system scales out. At this point, the full functionality of the system can be restored.
- The next figure shows an area graph of the overall resource use by all applications running in a system against time, and illustrates how throttling can be combined with autoscaling.



At time T1, the threshold specifying the soft limit of resource use is reached. At this point, the system can start to scale out. However, if the new resources don't become available quickly enough, then the existing resources might be exhausted and the system could fail. To prevent this from occurring, the system is temporarily throttled, as described earlier. When autoscaling has completed and the additional resources are available, throttling can be relaxed.

Throttling pattern

Issues and considerations

- Throttling an application, and the strategy to use, is an architectural decision that **impacts the entire design of a system**. Throttling should be considered **early** in the application design process because it isn't easy to add once a system has been implemented.
- **Throttling must be performed quickly**. The system must be capable of detecting an increase in activity and react accordingly. The system must also be able to **revert to its original state quickly** after the load has eased. This requires that the appropriate performance data is continually captured and monitored.
- If a service needs to temporarily deny a user request, it should return a specific **error code so the client application understands that the reason for the refusal** to perform an operation is due to throttling. The client application can wait for a period before retrying the request.
- Throttling can be used as a **temporary measure while a system autoscales**. In some cases it is better to simply throttle, rather than to scale, if a burst in activity is sudden and is not expected to be long lived because scaling can add considerably to running costs.

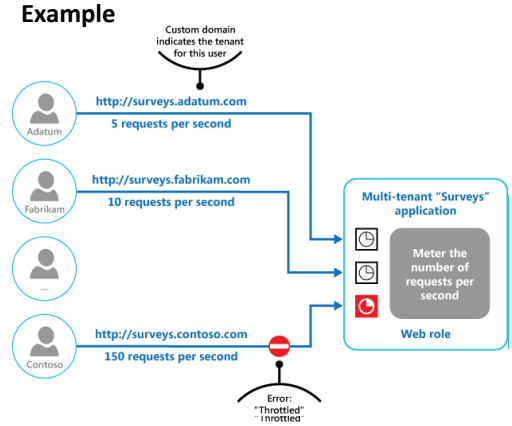
• If throttling is being used as a temporary measure while a system autoscales, and if resource demands grow very quickly, the system might not be able to continue functioning—even when operating in a throttled mode. If this isn't acceptable, consider maintaining larger capacity reserves and configuring more aggressive autoscaling.

Throttling pattern

Use this pattern:

- To ensure that a system continues to meet service level agreements.
- To prevent a single tenant from monopolizing the resources provided by an application.
- To handle bursts in activity.
- To help cost-optimize a system by limiting the maximum resource levels needed to keep it functioning.

Example



• If throttling is being used as a temporary measure while a system autoscales, and if resource demands grow very quickly, the system might not be able to continue functioning—even when operating in a throttled mode. If this isn't acceptable, consider maintaining larger capacity reserves and configuring more aggressive autoscaling.

Cloud Design Patterns

High Availability

Pattern	Summary
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
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.

Bulkhead pattern

The Bulkhead pattern is a type of application design that is tolerant of failure. In a bulkhead architecture, **elements of an application are isolated into pools so that if one fails, the others will continue to function**. It's named after the sectioned partitions (bulkheads) of a ship's hull. If the hull of a ship is compromised, only the damaged section fills with water, which prevents the ship from sinking.

Context and problem

- A cloud-based application may include multiple services, with each service having one or more consumers. Excessive load or failure in a service will impact all consumers of the service.
- Moreover, a consumer may send requests to multiple services simultaneously, using resources for each request. When the consumer sends a request to a service that is misconfigured or not responding, the resources used by the client's request may not be freed in a timely manner. As requests to the service continue, those resources may be exhausted. For example, the client's connection pool may be exhausted. At that point, requests by the consumer to other services are affected. Eventually the consumer can no longer send requests to other services, not just the original unresponsive service.

The same issue of resource exhaustion affects services with multiple consumers. A large number of requests originating from one client may exhaust available resources in the service. Other consumers are no longer able to consume the service, causing a cascading failure effect.

Bulkhead pattern

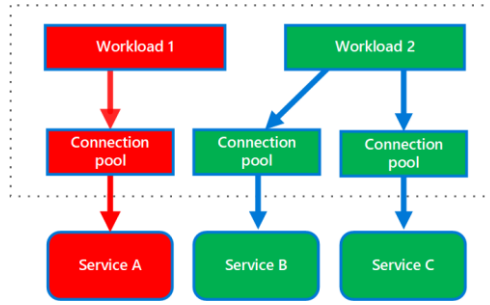
- **Partition service instances into different groups**, based on consumer load and availability requirements. This design helps to **isolate failures**, and allows you to sustain service functionality for some consumers, even during a failure.
- A consumer can also **partition resources**, to ensure that resources used to call one service don't affect the resources used to call another service. For example, a consumer that calls multiple services may be assigned a connection pool for each service. If a service begins to fail, it only affects the connection pool assigned for that service, allowing the consumer to continue using the other services.

The benefits of this pattern include:

- **Isolates consumers and services from cascading failures**. An issue affecting a consumer or service can be isolated within its own bulkhead, preventing the entire solution from failing.
- Allows you to **preserve some functionality** in the event of a service failure. Other services and features of the application will continue to work.
- Allows you to deploy services that offer a **different quality of service for consuming applications**. A high-priority consumer pool can be configured to use high-priority services.

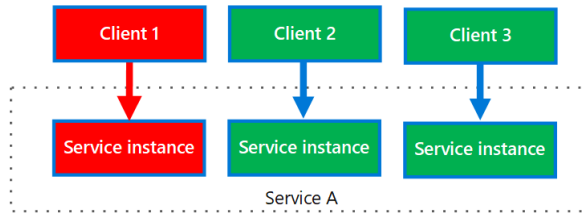
Bulkhead pattern

- The following diagram shows bulkheads structured around connection pools that call individual services. If Service A fails or causes some other issue, the connection pool is isolated, so only workloads using the thread pool assigned to Service A are affected. Workloads that use Service B and C are not affected and can continue working without interruption.



Bulkhead pattern

- The next diagram shows multiple clients calling a single service. Each client is assigned a separate service instance. Client 1 has made too many requests and overwhelmed its instance. Because each service instance is isolated from the others, the other clients can continue making calls.



Bulkhead pattern

Issues and considerations

- Define partitions around the **business and technical requirements** of the application.
- When partitioning services or consumers into bulkheads, consider the **level of isolation offered by the technology** as well as the overhead in terms of cost, performance and manageability.
- Consider **combining bulkheads with retry, circuit breaker, and throttling patterns** to provide more sophisticated fault handling.
- When partitioning services into bulkheads, consider deploying them into **separate virtual machines, containers, or processes**. Containers offer a good balance of resource isolation with fairly low overhead.
- **Services** that communicate using asynchronous messages **can be isolated through different sets of queues**. Each queue can have a dedicated set of instances processing messages on the queue, or a single group of instances using an algorithm to dequeue and dispatch processing.

When partitioning consumers into bulkheads, consider using processes, thread pools, and semaphores. Projects like resilience4j and Polly offer a framework for creating consumer bulkheads.

Determine the level of granularity for the bulkheads. For example, if you want to distribute tenants across partitions, you could place each tenant into a separate partition, or put several tenants into one partition.

Bulkhead pattern

Use this pattern to:

- Isolate resources used to consume a set of backend services, especially if the application can provide some level of functionality even when one of the services is not responding.
- Isolate critical consumers from standard consumers.
- Protect the application from cascading failures.

This pattern may not be suitable when:

- **Less efficient use** of resources may not be acceptable in the project.
- The **added complexity** is not necessary

Bulkhead pattern

Example

The following Kubernetes configuration file creates an isolated container to run a single service, with its own CPU and memory resources and limits.

```
apiVersion: v1
kind: Pod
metadata:
  name: drone-management
spec:
  containers:
  - name: drone-management-container
    image: drone-service
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "1"
```

Circuit Breaker pattern

Handle faults that might take a variable amount of time to recover from, when connecting to a remote service or resource. This can improve the stability and resiliency of an application.

Context and problem

- There can be situations where faults are due to unanticipated events, and that might **take long to fix**. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.
- If a service is very busy, failure in one part of the system might lead to cascading failures. An operation that invokes a service could be configured to implement a timeout, and reply with a failure message if the service fails to respond within this period. However, this strategy could cause many concurrent requests to the same operation to be blocked until the timeout period expires. **These blocked requests might hold critical system resources such as memory, threads, database connections, and so on**. Consequently, these resources could become exhausted. It would be preferable for the operation to **fail immediately**, and only attempt to invoke the service if it is likely to succeed.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them by using a strategy such as the Retry pattern.

However, there can also be situations where faults are due to unanticipated events, and that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure in one part of the system might lead to cascading failures. For example, an operation that invokes a service could be configured to implement a timeout, and reply with a failure message if the service fails to respond within this period. However, this strategy could cause many concurrent requests to the same operation to be blocked until the timeout period expires. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on. Consequently, these resources could become exhausted, causing failure of other possibly unrelated parts of the system that need to use the same resources. In these situations, it would be preferable for the operation to fail immediately, and only attempt to invoke the service if it is likely to succeed. Note that setting a shorter timeout might help to resolve this problem, but the timeout shouldn't be so short that the operation fails most of the time, even if the request to the service would

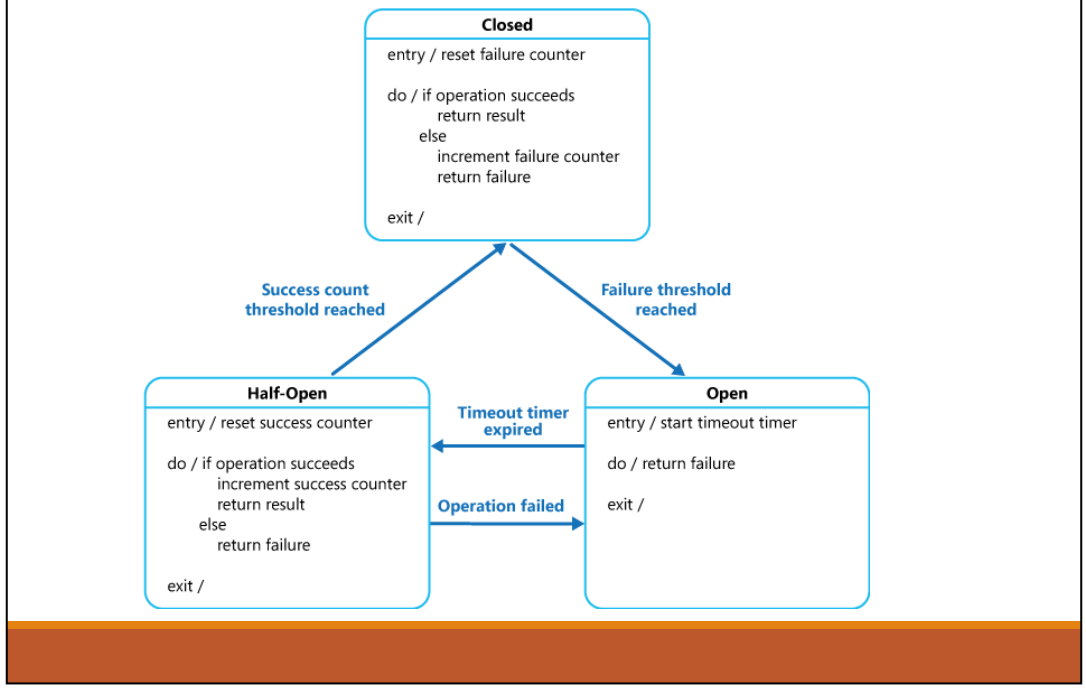
eventually succeed.

Circuit Breaker pattern

- The Circuit Breaker pattern can prevent an application from repeatedly trying to execute an operation that's likely to fail. Allowing it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long lasting. The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.
- A circuit breaker acts as a **proxy for operations that might fail**. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.
- The proxy can be implemented as a **state machine** with the following states that mimic the functionality of an electrical circuit breaker.

The purpose of the Circuit Breaker pattern is different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it'll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.

Circuit Breaker pattern



•**Closed**: In this state, the request from the application is routed to the operation. The proxy maintains a count of the number of recent failures, and if the call to the operation is unsuccessful the proxy increments this count. If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the **Open** state. At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the **Half-Open** state.

•The purpose of the timeout timer is to give the system time to fix the problem that caused the failure before allowing the application to try to perform the operation again.

•**Open**: The request from the application fails immediately and an exception is returned to the application.

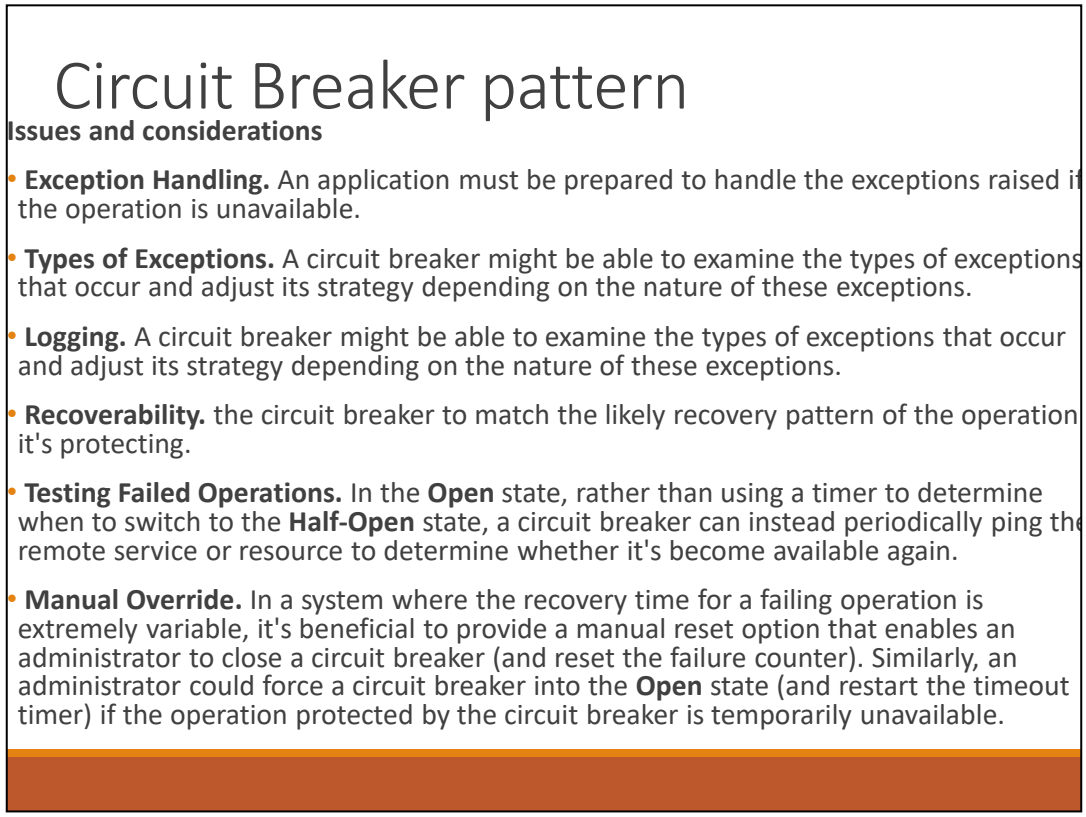
•**Half-Open**: A limited number of requests from the application are allowed to pass through and invoke the operation. If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state (the failure counter is reset). If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure.

•The **Half-Open** state is useful to prevent a recovering service from suddenly being flooded with requests. As a service recovers, it might be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work can cause the service to time out or fail again.

The Circuit Breaker pattern provides stability while the system recovers from a

failure and minimizes the impact on performance. It can help to maintain the response time of the system by quickly rejecting a request for an operation that's likely to fail, rather than waiting for the operation to time out, or never return. If the circuit breaker raises an event each time it changes state, this information can be used to monitor the health of the part of the system protected by the circuit breaker, or to alert an administrator when a circuit breaker trips to the **Open** state.

The pattern is customizable and can be adapted according to the type of the possible failure. For example, you can apply an increasing timeout timer to a circuit breaker. You could place the circuit breaker in the **Open** state for a few seconds initially, and then if the failure hasn't been resolved increase the timeout to a few minutes, and so on. In some cases, rather than the **Open** state returning failure and raising an exception, it could be useful to return a default value that is meaningful to the application.



Issues and considerations

You should consider the following points when deciding how to implement this pattern:

Exception Handling. An application invoking an operation through a circuit breaker must be prepared to handle the exceptions raised if the operation is unavailable. The way exceptions are handled will be application specific. For example, an application could temporarily degrade its functionality, invoke an alternative operation to try to perform the same task or obtain the same data, or report the exception to the user and ask them to try again later.

Types of Exceptions. A request might fail for many reasons, some of which might indicate a more severe type of failure than others. For example, a request might fail because a remote service has crashed and will take several minutes to recover, or because of a timeout due to the service being temporarily overloaded. A circuit breaker might be able to examine the types of exceptions that occur and adjust its strategy depending on the nature of these exceptions. For example, it might require a larger number of timeout exceptions to trip the circuit breaker to the **Open** state compared to the number of failures due to the service being completely unavailable.

Logging. A circuit breaker should log all failed requests (and possibly successful requests) to enable an administrator to monitor the health of the operation.

Recoverability. You should configure the circuit breaker to match the likely recovery pattern of the operation it's protecting. For example, if the circuit breaker remains in the **Open** state for a long period, it could raise exceptions even if the

reason for the failure has been resolved. Similarly, a circuit breaker could fluctuate and reduce the response times of applications if it switches from the **Open** state to the **Half-Open** state too quickly.

Testing Failed Operations. In the **Open** state, rather than using a timer to determine when to switch to the **Half-Open** state, a circuit breaker can instead periodically ping the remote service or resource to determine whether it's become available again. This ping could take the form of an attempt to invoke an operation that had previously failed, or it could use a special operation provided by the remote service specifically for testing the health of the service, as described by the Health Endpoint Monitoring pattern.

Manual Override. In a system where the recovery time for a failing operation is extremely variable, it's beneficial to provide a manual reset option that enables an administrator to close a circuit breaker (and reset the failure counter). Similarly, an administrator could force a circuit breaker into the **Open** state (and restart the timeout timer) if the operation protected by the circuit breaker is temporarily unavailable.

Concurrency. The same circuit breaker could be accessed by a large number of concurrent instances of an application. The implementation shouldn't block concurrent requests or add excessive overhead to each call to an operation.

Resource Differentiation. Be careful when using a single circuit breaker for one type of resource if there might be multiple underlying independent providers. For example, in a data store that contains multiple shards, one shard might be fully accessible while another is experiencing a temporary issue. If the error responses in these scenarios are merged, an application might try to access some shards even when failure is highly likely, while access to other shards might be blocked even though it's likely to succeed.

Accelerated Circuit Breaking. Sometimes a failure response can contain enough information for the circuit breaker to trip immediately and stay tripped for a minimum amount of time. For example, the error response from a shared resource that's overloaded could indicate that an immediate retry isn't recommended and that the application should instead try again in a few minutes.

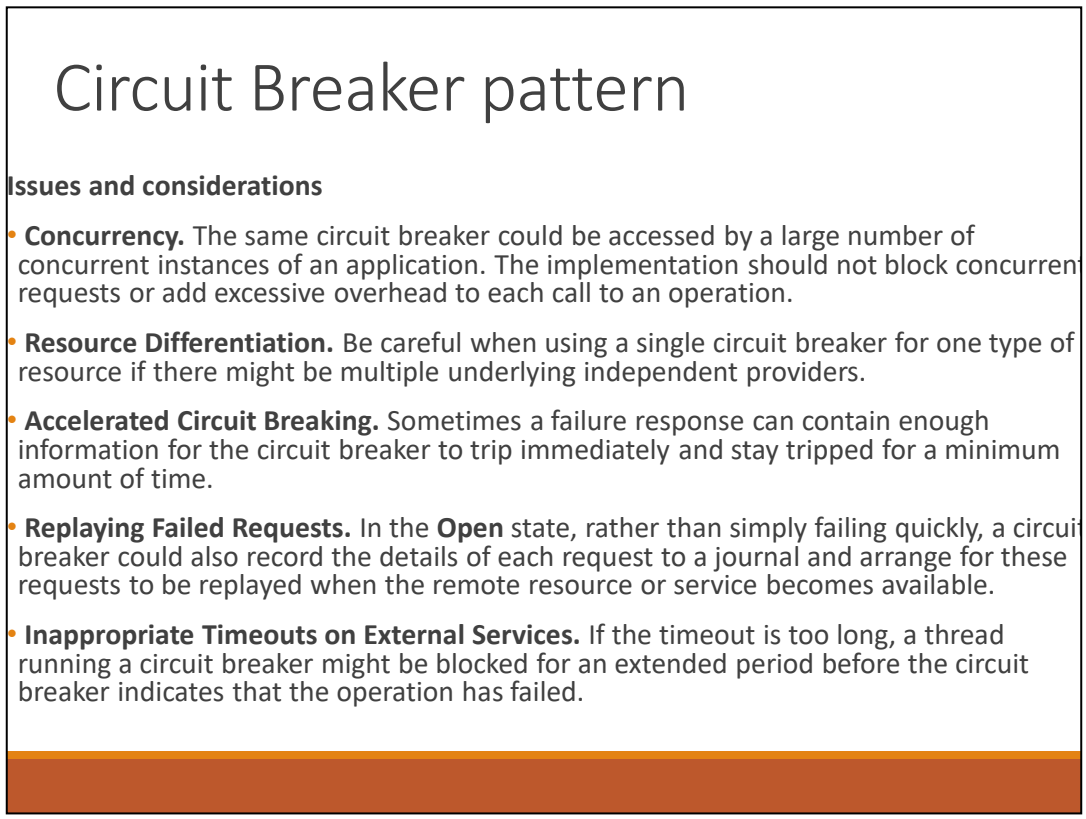
Note

A service can return HTTP 429 (Too Many Requests) if it is throttling the client, or HTTP 503 (Service Unavailable) if the service is not currently available. The response can include additional information, such as the anticipated duration of the delay.

Replaying Failed Requests. In the **Open** state, rather than simply failing quickly, a circuit breaker could also record the details of each request to a journal and arrange for these requests to be replayed when the remote resource or service becomes available.

Inappropriate Timeouts on External Services. A circuit breaker might not be able to fully protect applications from operations that fail in external services that are configured with a lengthy timeout period. If the timeout is too long, a thread running a circuit breaker might be blocked for an extended period before the circuit breaker indicates that the operation has failed. In this time, many other

application instances might also try to invoke the service through the circuit breaker and tie up a significant number of threads before they all fail.



Issues and considerations

You should consider the following points when deciding how to implement this pattern:

Exception Handling. An application invoking an operation through a circuit breaker must be prepared to handle the exceptions raised if the operation is unavailable. The way exceptions are handled will be application specific. For example, an application could temporarily degrade its functionality, invoke an alternative operation to try to perform the same task or obtain the same data, or report the exception to the user and ask them to try again later.

Types of Exceptions. A request might fail for many reasons, some of which might indicate a more severe type of failure than others. For example, a request might fail because a remote service has crashed and will take several minutes to recover, or because of a timeout due to the service being temporarily overloaded. A circuit breaker might be able to examine the types of exceptions that occur and adjust its strategy depending on the nature of these exceptions. For example, it might require a larger number of timeout exceptions to trip the circuit breaker to the **Open** state compared to the number of failures due to the service being completely unavailable.

Logging. A circuit breaker should log all failed requests (and possibly successful requests) to enable an administrator to monitor the health of the operation.

Recoverability. You should configure the circuit breaker to match the likely recovery pattern of the operation it's protecting. For example, if the circuit breaker remains in the **Open** state for a long period, it could raise exceptions even if the

reason for the failure has been resolved. Similarly, a circuit breaker could fluctuate and reduce the response times of applications if it switches from the **Open** state to the **Half-Open** state too quickly.

Testing Failed Operations. In the **Open** state, rather than using a timer to determine when to switch to the **Half-Open** state, a circuit breaker can instead periodically ping the remote service or resource to determine whether it's become available again. This ping could take the form of an attempt to invoke an operation that had previously failed, or it could use a special operation provided by the remote service specifically for testing the health of the service, as described by the Health Endpoint Monitoring pattern.

Manual Override. In a system where the recovery time for a failing operation is extremely variable, it's beneficial to provide a manual reset option that enables an administrator to close a circuit breaker (and reset the failure counter). Similarly, an administrator could force a circuit breaker into the **Open** state (and restart the timeout timer) if the operation protected by the circuit breaker is temporarily unavailable.

Concurrency. The same circuit breaker could be accessed by a large number of concurrent instances of an application. The implementation should not block concurrent requests or add excessive overhead to each call to an operation.

Resource Differentiation. Be careful when using a single circuit breaker for one type of resource if there might be multiple underlying independent providers. For example, in a data store that contains multiple shards, one shard might be fully accessible while another is experiencing a temporary issue. If the error responses in these scenarios are merged, an application might try to access some shards even when failure is highly likely, while access to other shards might be blocked even though it's likely to succeed.

Accelerated Circuit Breaking. Sometimes a failure response can contain enough information for the circuit breaker to trip immediately and stay tripped for a minimum amount of time. For example, the error response from a shared resource that's overloaded could indicate that an immediate retry isn't recommended and that the application should instead try again in a few minutes.

Note

A service can return HTTP 429 (Too Many Requests) if it is throttling the client, or HTTP 503 (Service Unavailable) if the service is not currently available. The response can include additional information, such as the anticipated duration of the delay.

Replaying Failed Requests. In the **Open** state, rather than simply failing quickly, a circuit breaker could also record the details of each request to a journal and arrange for these requests to be replayed when the remote resource or service becomes available.

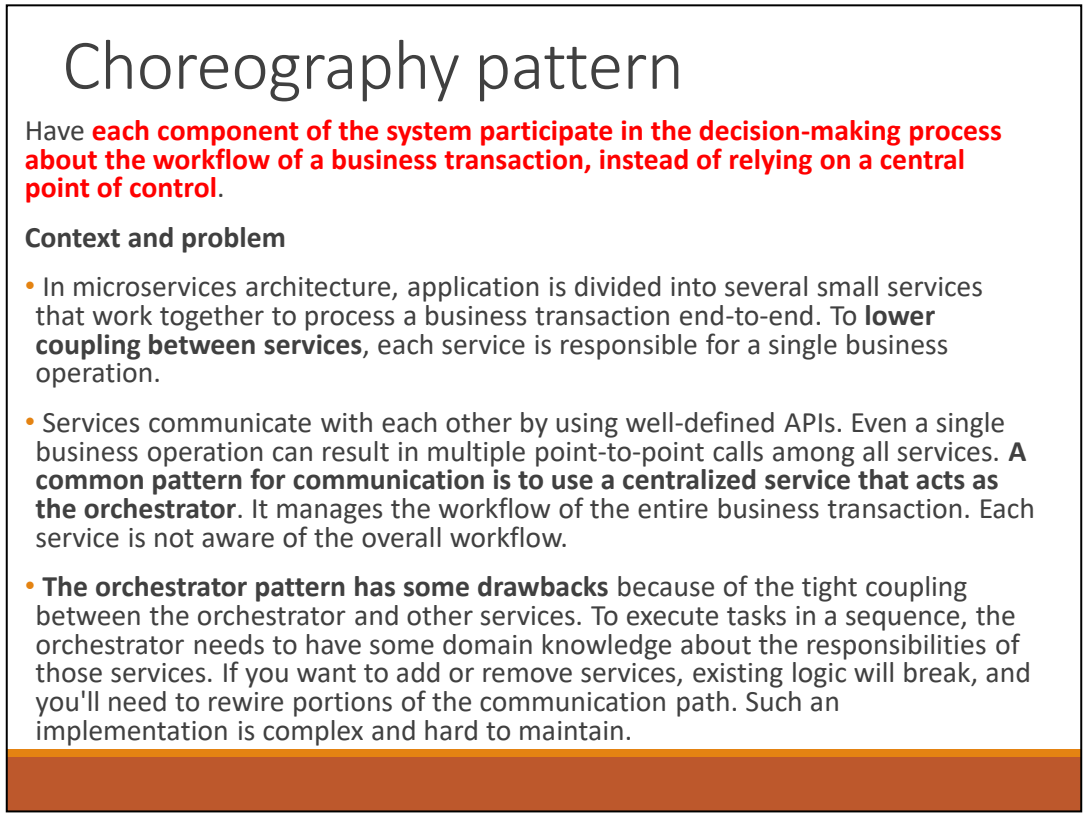
Inappropriate Timeouts on External Services. A circuit breaker might not be able to fully protect applications from operations that fail in external services that are configured with a lengthy timeout period. If the timeout is too long, a thread running a circuit breaker might be blocked for an extended period before the circuit breaker indicates that the operation has failed. In this time, many other

application instances might also try to invoke the service through the circuit breaker and tie up a significant number of threads before they all fail.

Cloud Design Patterns

Messaging

Pattern	Summary
Choreography	Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Publisher-Subscriber	Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.



Context and problem

In microservices architecture, it's often the case that a cloud-based application is divided into several small services that work together to process a business transaction end-to-end. To lower coupling between services, each service is responsible for a single business operation. Some benefits include faster development, smaller code base, and scalability. However, designing an efficient and scalable workflow is a challenge and often requires complex interservice communication.

The services communicate with each other by using well-defined APIs. Even a single business operation can result in multiple point-to-point calls among all services. A common pattern for communication is to use a centralized service that acts as the orchestrator. It acknowledges all incoming requests and delegates operations to the respective services. In doing so, it also manages the workflow of the entire business transaction. Each service just completes an operation and is not aware of the overall workflow.

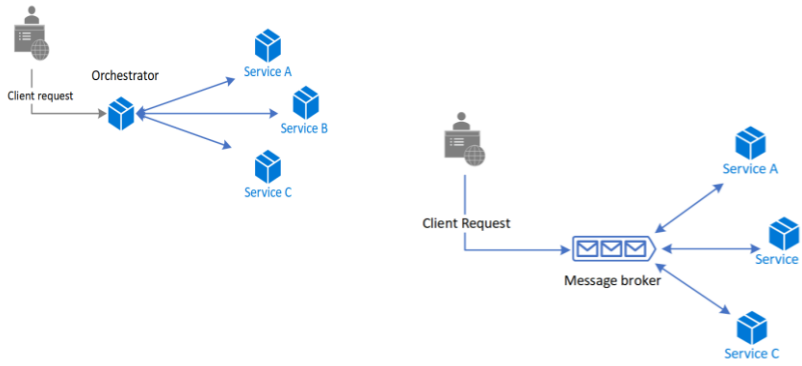
The orchestrator pattern reduces point-to-point communication between services but has some drawbacks because of the tight coupling between the orchestrator and other services that participate in processing of the business transaction. To execute tasks in a sequence, the orchestrator needs to have some domain knowledge about the responsibilities of those services. If you want to add or remove services, existing logic will break, and you'll need to rewire portions of the communication path. While you can configure the workflow, add or remove services easily with a well-designed orchestrator, such an implementation is

complex and hard to maintain.

Choreography pattern

Let each service decide when and how a business operation is processed, instead of depending on a central orchestrator.

One way to implement choreography is to use the asynchronous messaging pattern to coordinate the business operations.



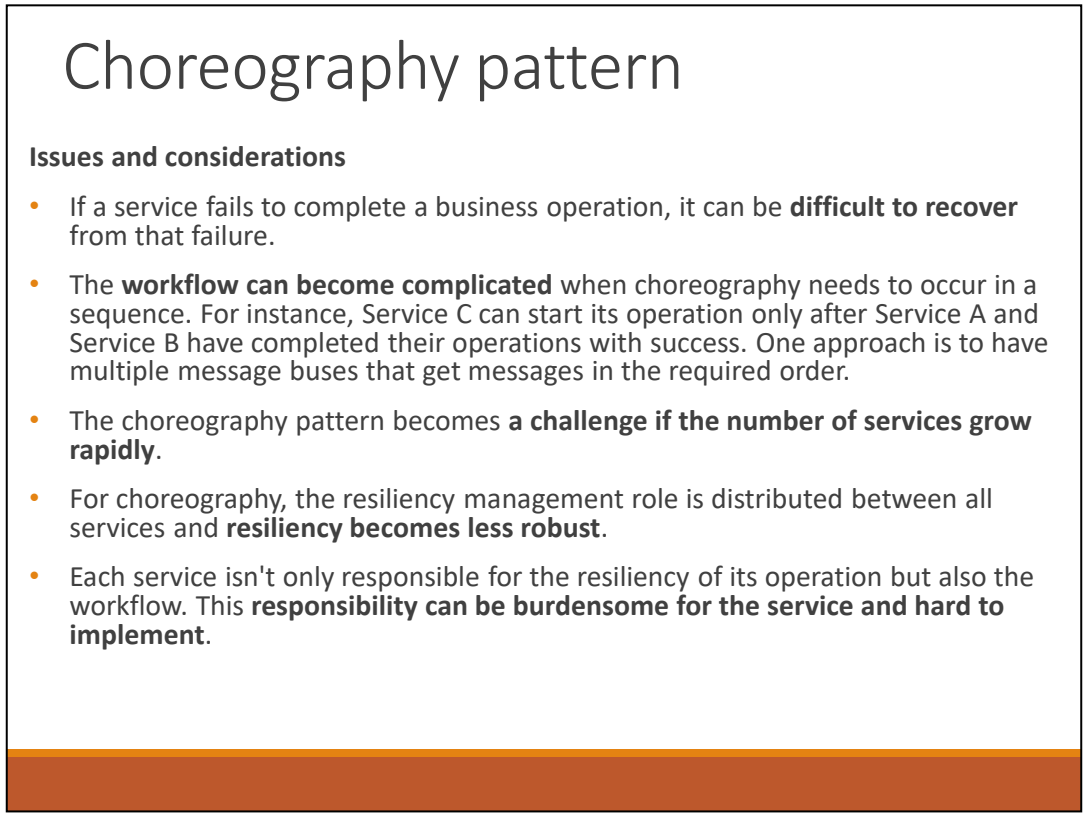
Choreography pattern

- A client request publishes messages to a message queue. As messages arrive, they are pushed to subscribers, or services, interested in that message. Each subscribed service does their operation as indicated by the message and responds to the message queue with success or failure of the operation. In case of success, the service can push a message back to the same queue or a different message queue so that another service can continue the workflow if needed. If an operation fails, the message bus can retry that operation.
- This way, services choreograph the workflow among themselves without depending on an orchestrator or having direct communication between them.
- Because there is not point-to-point communication, this pattern **helps reduce coupling between services**. Also, it can remove the performance bottleneck caused by the orchestrator when it has to deal with all transactions.

Choreography pattern

When to use this pattern

- Use the choreography pattern if you expect to update, remove, or add new services frequently. The entire app can be modified with lesser effort and minimal disruption to existing services.
- Consider this pattern if you experience performance bottlenecks in the central orchestrator.
- This pattern is a natural model for the **serverless architecture** where all services can be short lived, or event driven. Services can spin up because of an event, do their task, and are removed when the task is finished.



Issues and considerations

Decentralizing the orchestrator can cause issues while managing the workflow.

If a service fails to complete a business operation, it can be difficult to recover from that failure. One way is to have the service indicate failure by firing an event. Another service subscribes to those failed events takes necessary actions such as applying compensating transactions to undo successful operations in a request. The failed service might also fail to fire an event for the failure. In that case, consider using a retry and, or time out mechanism to recognize that operation as a failure. For an example, see the Example section.

It's simple to implement a workflow when you want to process independent business operations in parallel. You can use a single message bus. However, the workflow can become complicated when choreography needs to occur in a sequence. For instance, Service C can start its operation only after Service A and Service B have completed their operations with success. One approach is to have multiple message buses that get messages in the required order. For more information, see the Example section.

The choreography pattern becomes a challenge if the number of services grow rapidly. Given the high number of independent moving parts, the workflow between services tends to get complex. Also, distributed tracing becomes difficult.

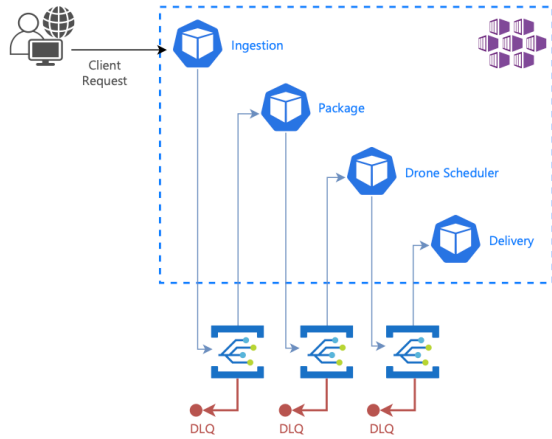
The orchestrator centrally manages the resiliency of the workflow and it can become a single point of failure. On the other hand, for choreography, the role is distributed between all services and resiliency becomes less robust.

Each service isn't only responsible for the resiliency of its operation but also the workflow. This responsibility can be burdensome for the service and hard to implement. Each service must retry transient, nontransient, and time-out failures, so that the request terminates gracefully, if needed. Also, the service must be diligent about communicating the success or failure of the operation so that other services can act accordingly.

Choreography pattern

Example

This example shows the choreography pattern with the Drone Delivery app. When a client requests a pickup, the app assigns a drone and notifies the client.



A single client business transaction requires three distinct business operations: creating or updating a package, assigning a drone to deliver the package, and checking the delivery status. Those operations are performed by three microservices: Package, Drone Scheduler, and Delivery services. Instead of a central orchestrator, the services use messaging to collaborate and coordinate the request among themselves.

The dead-letter queue (or undelivered-message queue) is the queue to which messages are sent if they cannot be routed to their correct destination. Each queue manager typically has a dead-letter queue.

A dead-letter queue (DLQ), sometimes referred to as an undelivered-message queue, is a holding queue for messages that cannot be delivered to their destination queues, for example because the queue does not exist, or because it is full. Dead-letter queues are also used at the sending end of a channel, for data-conversion errors.. Every queue manager in a network typically has a local queue to be used as a dead-letter queue so that messages that cannot be delivered to their correct destination can be stored for later retrieval.

Publisher-Subscriber pattern

Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.

Also called: Pub/sub messaging

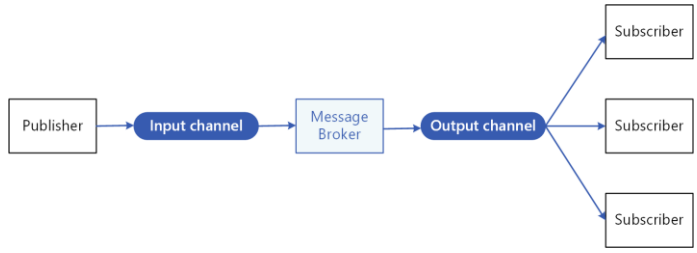
Context and problem

- In cloud-based and distributed applications, components of the system often need to provide information to other components as events happen.
- Asynchronous messaging is an effective way to decouple senders from consumers, and avoid blocking the sender to wait for a response. However, using a dedicated message queue for each consumer does not effectively scale to many consumers. Also, some of the consumers might be interested in only a subset of the information. How can the sender announce events to all interested consumers without knowing their identities.

Publisher-Subscriber pattern

Introduce an asynchronous messaging subsystem that includes the following:

- An input messaging channel used by the sender. The sender packages events into messages, using a known message format, and sends these messages via the input channel. The sender in this pattern is also called the *publisher*.
- One output messaging channel per consumer. The consumers are known as *subscribers*.
- A mechanism for copying each message from the input channel to the output channels for all subscribers interested in that message. This operation is typically handled by an intermediary such as a message broker or event bus.



Publisher-Subscriber pattern

pub/sub messaging has the following benefits:

- It decouples subsystems that still need to communicate.
- It increases scalability and improves responsiveness of the sender.
- It improves reliability.
- It allows for deferred or scheduled processing.
- It enables simpler integration between systems using different platforms, programming languages, or communication protocols, as well as between on-premises systems and applications running in the cloud.
- It facilitates asynchronous workflows across an enterprise.
- It improves testability. Channels can be monitored and messages can be inspected or logged as part of an overall integration test strategy.
- It provides separation of concerns for your applications.

ub/sub messaging has the following benefits:

•It decouples subsystems that still need to communicate. Subsystems can be managed independently, and messages can be properly managed even if one or more receivers are offline.

•It increases scalability and improves responsiveness of the sender. The sender can quickly send a single message to the input channel, then return to its core processing responsibilities. The messaging infrastructure is responsible for ensuring messages are delivered to interested subscribers.

•It improves reliability. Asynchronous messaging helps applications continue to run smoothly under increased loads and handle intermittent failures more effectively.

•It allows for deferred or scheduled processing. Subscribers can wait to pick up messages until off-peak hours, or messages can be routed or processed according to a specific schedule.

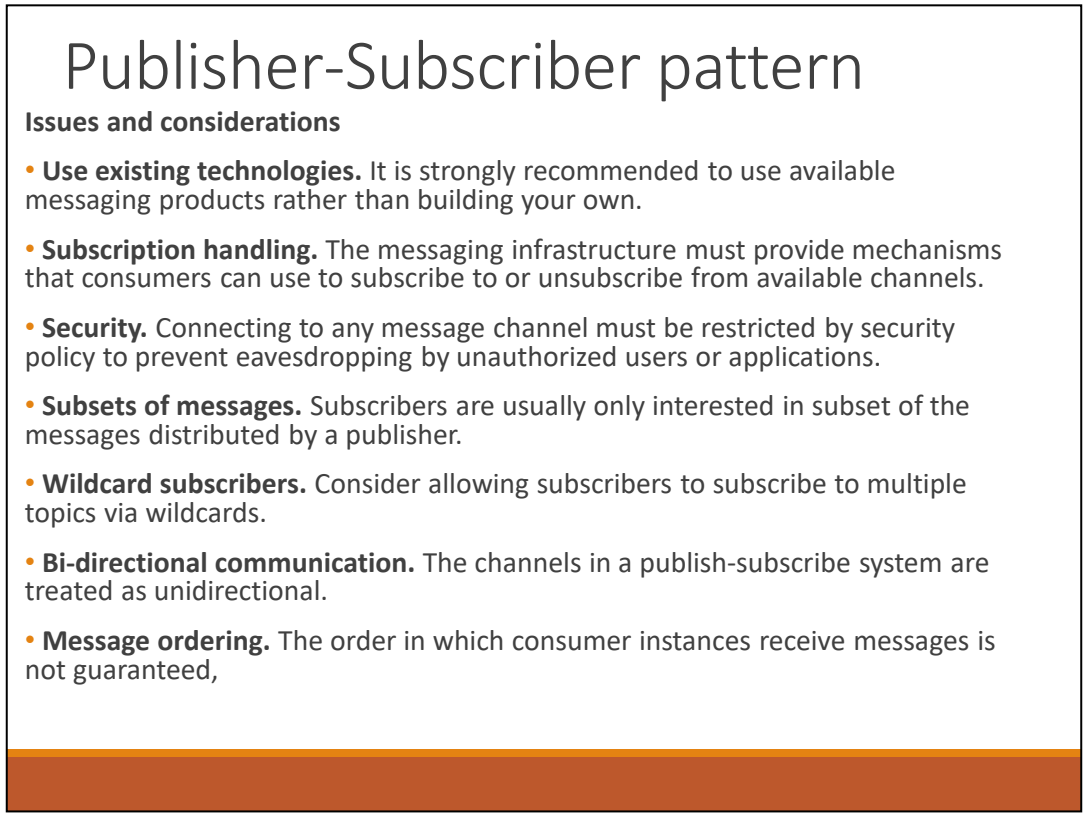
•It enables simpler integration between systems using different platforms, programming languages, or communication protocols, as well as between on-premises systems and applications running in the cloud.

•It facilitates asynchronous workflows across an enterprise.

•It improves testability. Channels can be monitored and messages can be inspected or logged as part of an overall integration test strategy.

•It provides separation of concerns for your applications. Each application can focus on its core capabilities, while the messaging infrastructure handles

everything required to reliably route messages to multiple consumers.



Issues and considerations

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

•**Existing technologies.** It is strongly recommended to use available messaging products and services that support a publish-subscribe model, rather than building your own. In Azure, consider using Service Bus, Event Hubs or Event Grid. Other technologies that can be used for pub/sub messaging include Redis, RabbitMQ, and Apache Kafka.

•**Subscription handling.** The messaging infrastructure must provide mechanisms that consumers can use to subscribe to or unsubscribe from available channels.

•**Security.** Connecting to any message channel must be restricted by security policy to prevent eavesdropping by unauthorized users or applications.

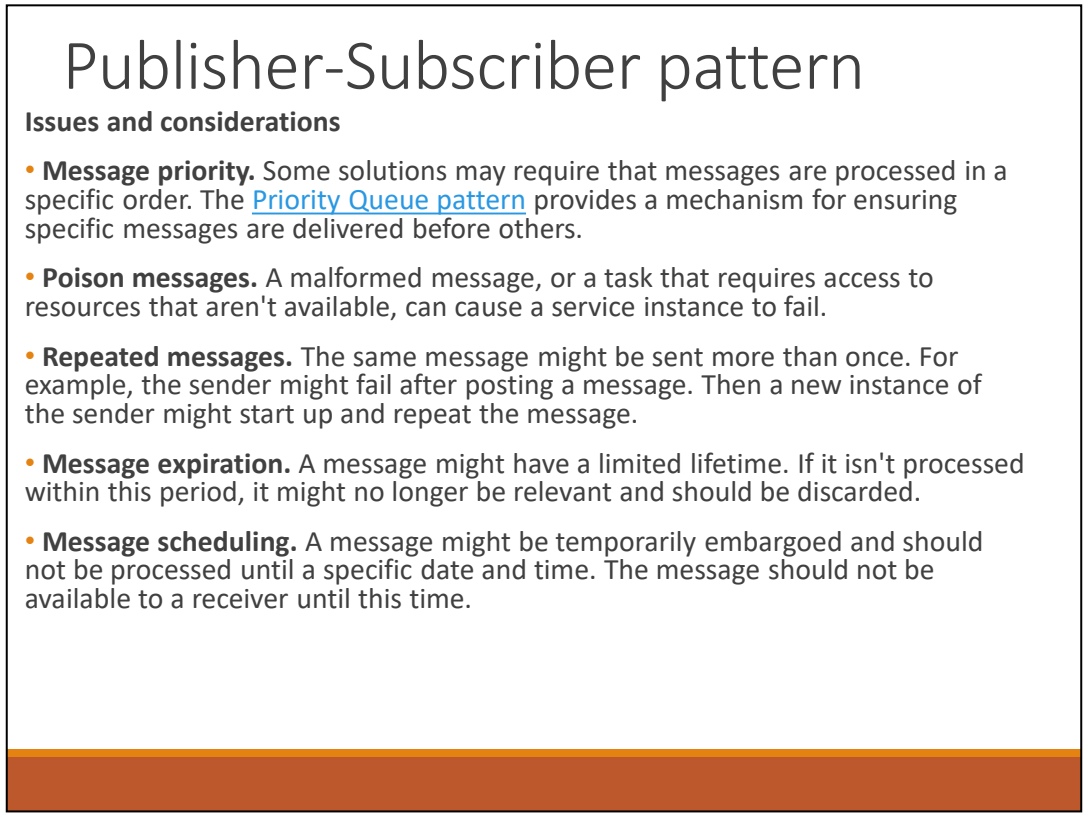
•**Subsets of messages.** Subscribers are usually only interested in subset of the messages distributed by a publisher. Messaging services often allow subscribers to narrow the set of messages received by:

- **Topics.** Each topic has a dedicated output channel, and each consumer can subscribe to all relevant topics.
- **Content filtering.** Messages are inspected and distributed based on the content of each message. Each subscriber can specify the content it is interested in.

•**Wildcard subscribers.** Consider allowing subscribers to subscribe to multiple topics via wildcards.

•**Bi-directional communication.** The channels in a publish-subscribe system are treated as unidirectional. If a specific subscriber needs to send acknowledgment or communicate status back to the publisher, consider using the Request/Reply Pattern. This pattern uses one channel to send a message to the subscriber, and a separate reply channel for communicating back to the publisher.

•**Message ordering.** The order in which consumer instances receive messages isn't guaranteed, and doesn't necessarily reflect the order in which the messages were created. Design the system to ensure that message processing is idempotent to help eliminate any dependency on the order of message handling.



•**Message priority.** Some solutions may require that messages are processed in a specific order. The Priority Queue pattern provides a mechanism for ensuring specific messages are delivered before others.

•**Poison messages.** A malformed message, or a task that requires access to resources that aren't available, can cause a service instance to fail. The system should prevent such messages being returned to the queue. Instead, capture and store the details of these messages elsewhere so that they can be analyzed if necessary. Some message brokers, like Azure Service Bus, support this via their dead-letter queue functionality.

•**Repeated messages.** The same message might be sent more than once. For example, the sender might fail after posting a message. Then a new instance of the sender might start up and repeat the message. The messaging infrastructure should implement duplicate message detection and removal (also known as de-duping) based on message IDs in order to provide at-most-once delivery of messages. Alternatively, if using messaging infrastructure which doesn't de-duplicate messages, make sure the message processing logic is idempotent.

•**Message expiration.** A message might have a limited lifetime. If it isn't processed within this period, it might no longer be relevant and should be discarded. A sender can specify an expiration time as part of the data in the message. A receiver can examine this information before deciding whether to perform the business logic associated with the message.

•**Message scheduling.** A message might be temporarily embargoed and should not be processed until a specific date and time. The message should not be

available to a receiver until this time.

Publisher-Subscriber pattern

Use this pattern when:

- An application needs to broadcast information to a significant number of consumers.
- An application needs to communicate with one or more independently-developed applications or services, which may use different platforms, programming languages, and communication protocols.
- An application can send information to consumers without requiring real-time responses from the consumers.
- The systems being integrated are designed to support an eventual consistency model for their data.
- An application needs to communicate information to multiple consumers, which may have different availability requirements or uptime schedules than the sender.

This pattern might not be useful when:

- An application has only a few consumers who need significantly different information from the producing application.
- An application requires near real-time interaction with consumers.

Publisher-Subscriber pattern

Example

- An application needs to broadcast information to a significant number of consumers.
- An application needs to communicate with one or more independently-developed applications or services, which may use different platforms, programming languages, and communication protocols.
- An application can send information to consumers without requiring real-time responses from the consumers.
- The systems being integrated are designed to support an eventual consistency model for their data.
- An application needs to communicate information to multiple consumers, which may have different availability requirements or uptime schedules than the sender.

This pattern might not be useful when:

- An application has only a few consumers who need significantly different information from the producing application.
- An application requires near real-time interaction with consumers.

Priority Queue pattern

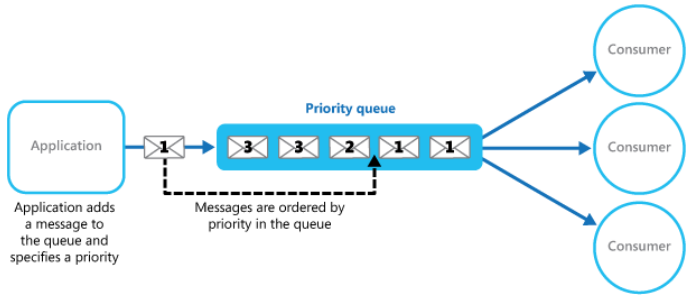
Prioritize requests sent to services so that **requests with a higher priority are received and processed more quickly than those with a lower priority**. This pattern is useful in applications that offer different service level guarantees to individual clients.

Context and problem

Applications can delegate specific tasks to other services, for example, to perform background processing or to integrate with other applications or services. In the cloud, a message queue is typically used to delegate tasks to background processing. In many cases, the order requests are received in by a service is not important. In some cases, though, it is necessary to prioritize specific requests. These requests should be processed earlier than lower priority requests that were sent previously by the application.

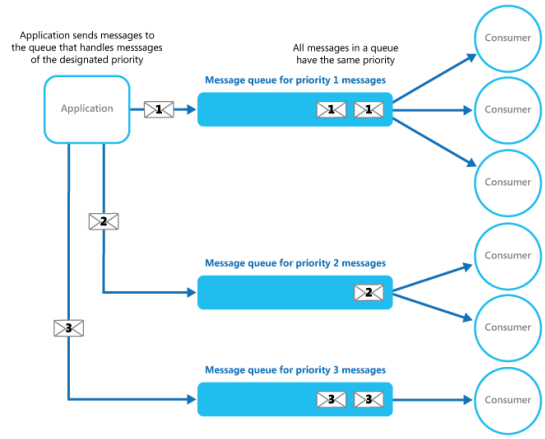
Priority Queue pattern

A queue is usually a first-in, first-out (FIFO) structure, and consumers typically receive messages in the same order that they were posted to the queue. However, some message queues support priority messaging. The application posting a message can assign a priority and the messages in the queue are automatically reordered so that those with a higher priority will be received before those with a lower priority. The figure illustrates a queue with priority messaging.



Priority Queue pattern

In systems that don't support priority-based message queues, an alternative solution is to maintain a separate queue for each priority. The application is responsible for posting messages to the appropriate queue. Each queue can have a separate pool of consumers. Higher priority queues can have a larger pool of consumers running on faster hardware than lower priority queues. The next figure illustrates using separate message queues for each priority.



A variation on this strategy is to have a single pool of consumers that check for messages on high priority queues first, and only then start to fetch messages from lower priority queues. There are some semantic differences between a solution that uses a single pool of consumer processes (either with a single queue that supports messages with different priorities or with multiple queues that each handle messages of a single priority), and a solution that uses multiple queues with a separate pool for each queue.

In the single pool approach, higher priority messages are always received and processed before lower priority messages. In theory, messages that have a very low priority could be continually superseded and might never be processed. In the multiple pool approach, lower priority messages will always be processed, just not as quickly as those of a higher priority (depending on the relative size of the pools and the resources that they have available).

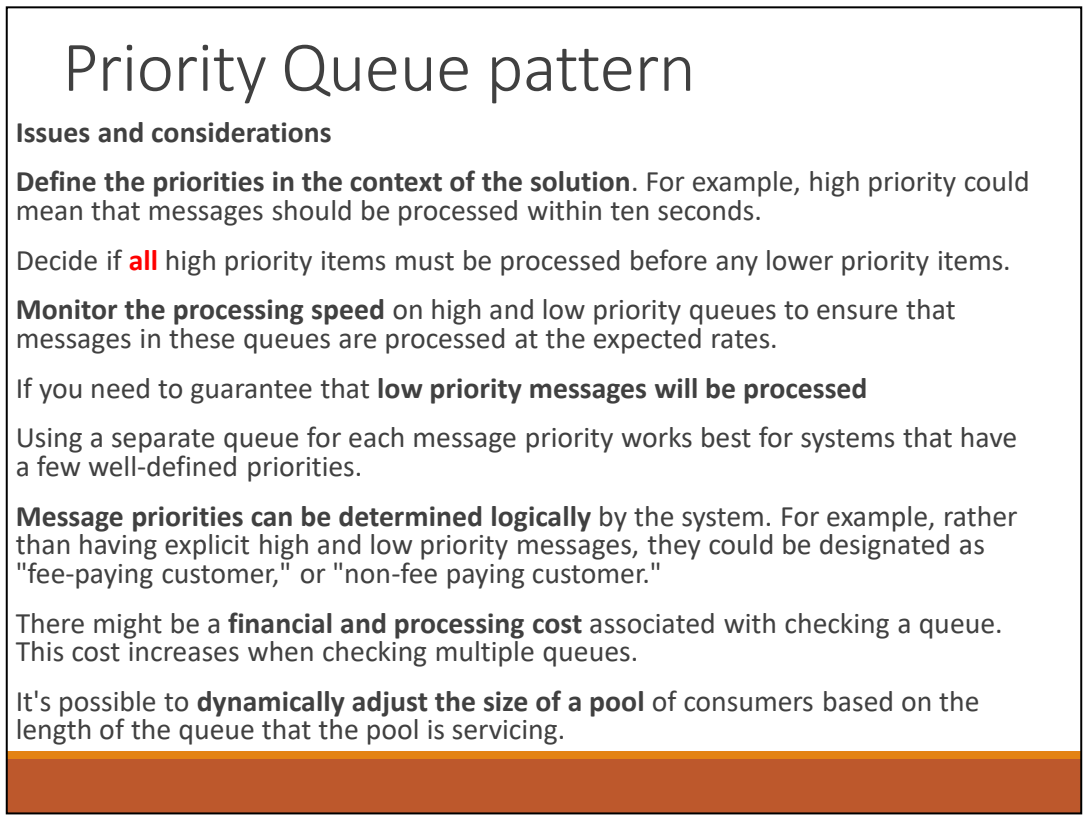
Priority Queue pattern

Using a priority-queuing mechanism can provide the following advantages:

- It allows applications to meet business requirements that require prioritization of availability or performance, such as offering different levels of service to specific groups of customers.

- It can help to **minimize operational costs**. In the single queue approach, you can scale back the number of consumers if necessary. High priority messages will still be processed first (although possibly more slowly), and lower priority messages might be delayed for longer. If you've implemented the multiple message queue approach with separate pools of consumers for each queue, you can reduce the pool of consumers for lower priority queues, or even suspend processing for some very low priority queues by stopping all the consumers that listen for messages on those queues.

- The **multiple message queue** approach can **help maximize application performance** and scalability by partitioning messages based on processing requirements. For example, vital tasks can be prioritized to be handled by receivers that run immediately while less important background tasks can be handled by receivers that are scheduled to run at less busy periods.



Issues and considerations

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

Define the priorities in the context of the solution. For example, high priority could mean that messages should be processed within ten seconds. Identify the requirements for handling high priority items, and the other resources that should be allocated to meet these criteria.

Decide if all high priority items must be processed before any lower priority items. If the messages are being processed by a single pool of consumers, you have to provide a mechanism that can preempt and suspend a task that's handling a low priority message if a higher priority message becomes available.

In the multiple queue approach, when using a single pool of consumer processes that listen on all queues rather than a dedicated consumer pool for each queue, the consumer must apply an algorithm that ensures it always services messages from higher priority queues before those from lower priority queues.

Monitor the processing speed on high and low priority queues to ensure that messages in these queues are processed at the expected rates.

If you need to guarantee that low priority messages will be processed, it's necessary to implement the multiple message queue approach with multiple pools of consumers. Alternatively, in a queue that supports message prioritization, it's possible to dynamically increase the priority of a queued message as it ages. However, this approach depends on the message queue providing this feature.

Using a separate queue for each message priority works best for systems that have a few well-defined priorities.

Message priorities can be determined logically by the system. For example, rather than having explicit high and low priority messages, they could be designated as "fee-paying customer," or "non-fee paying customer." Depending on your business model, your system can allocate more resources to processing messages from fee-paying customers than non-fee paying ones.

There might be a financial and processing cost associated with checking a queue for a message (some commercial messaging systems charge a small fee each time a message is posted or retrieved, and each time a queue is queried for messages). This cost increases when checking multiple queues.

It's possible to dynamically adjust the size of a pool of consumers based on the length of the queue that the pool is servicing. For more information, see the Autoscaling Guidance.