Applicare patterns di sviluppo con Azure

Marco Parenzan
Solutions Sales Specialist @ Insight
Azure MVP

#CodeGen

#dotnetconf

@cloudgen_verona

Main sponsor

Community sponsors

Syncfusion®

Packt

MANNING

elmah.io

Udemy™

JET BRAINS

Local Event

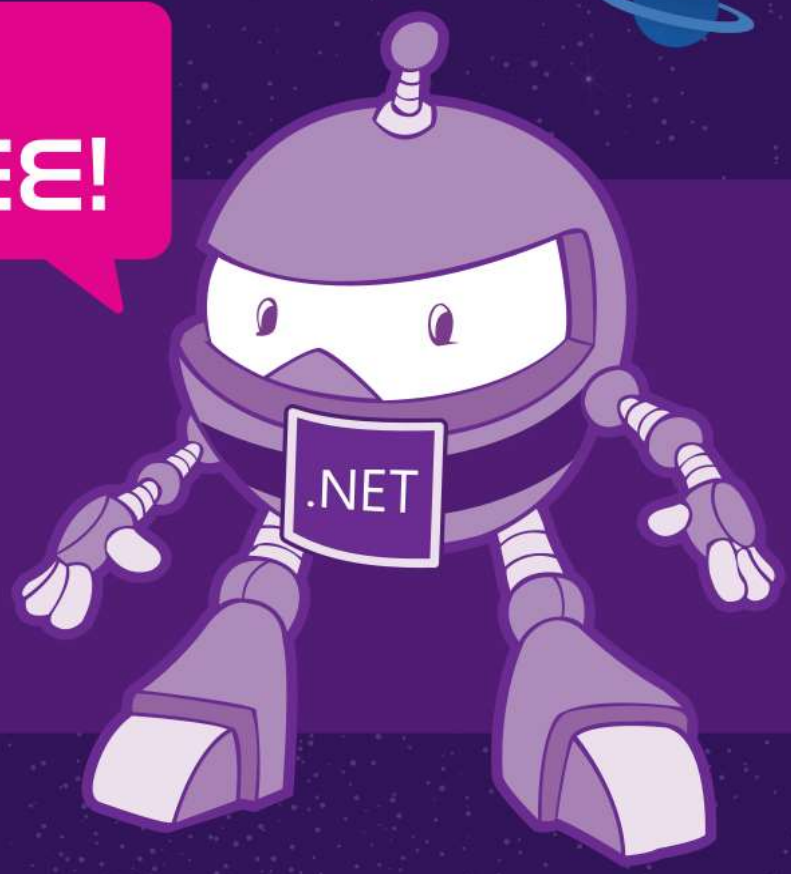
CloudGen

It's FREE!

.NET Conf

Discover the world of .NET

September 12-14, 2018



Tune in: www.dotnetconf.net

Marco Parenzan
Solutions Sales Specialist @ Insight
Azure MVP

marco_parenzan

marcoparenzan

marcoparenzan

Applicare patterns di sviluppo con Azure

Marco Parenzan
Solutions Sales Specialist @ Insight
Azure MVP

“

Everything works…

…until ~~it breaks~~ the damage!

“ I don’t have to convince you to move to the cloud…

“

…I want to convince you what moving to the cloud means…

“

…just to avoid wrong expectations…

“

…saving money…

“

…you can only increase business!

> You can have HA only with money and design (patterns)

“

Azure: the best place where applying your GOF book

“ Cloud is 66% Lift&Shift
Don’t rewrite

IaaS and Lift & Shift: an anti-pattern?

From Wikipedia [https://en.wikipedia.org/wiki/Anti-pattern]:

An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive

IaaS and Lift & Shift an anti-pattern: why?

Moving to the cloud is just a matter of costs

It costs

Then limits comes again

And opportunities can become big limits

You don’t avoid IaaS limits (infrastructural, hidden, unquoted costs)

IaaS is really evil?

NO....

...if, and only if,...

...a rewrite/evolution strategy is planned at day 1!

IaaS is not evil?

IaaS is the toolbox to hybridize company and cloud

Everything should be used with «care»

> PaaS if possible
> IaaS if (really really really) needed

[Software] Design pattern

From Wikipedia [https://en.wikipedia.org/wiki/Software_design_pattern]:

> In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.
>
> Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

> Lot of services are practice, not strictly «design»

“

Azure PaaS Services are Pattern implementations

“ Believe it or not….
This is the way

context

Agile

Alm

Devops

Ci/cd

Microservices

Elastic

Geo redundancy

Just an example: Azure Virtual Machine

It’s not a pattern

It’s a tool in your toolbox TO RUN your pattern implementation

Try to scale a VM in production….!

(every PaaS service run over VMs)

Just an example: Azure Sql Database

It’s just a Database

It’s a pattern!

Why?

«It’s ONLY 99.9% compatible with SQL Server»

It’s really very important?

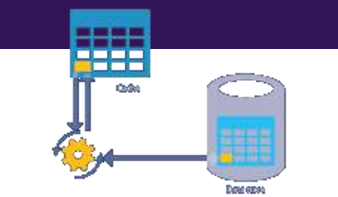
How many projects use SQL Server for its full power?

It’s a lot if it use Views or Stored procedures, even Triggers!

why a pattern? Because it’s just most developers do

Cache-aside

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

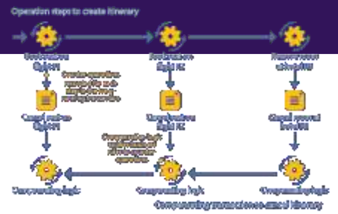


Circuit Breaker

Handle faults that may take a variable amount of time to rectify when connecting to a remote service or resource. This pattern can improve the stability and resiliency of an application.

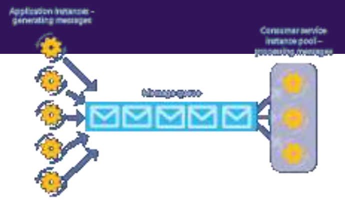
Compensating Transaction

Undo the work performed by a series of steps, which together define an eventually consistent operation, if one or more of the operations fails. Operations that follow the eventual consistency model are commonly found in cloud-hosted applications that implement complex business processes and workflows.



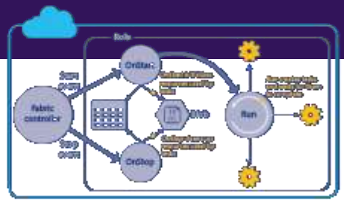
Competing Consumers

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



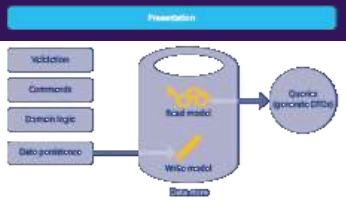
Compute Resource Consolidation

Consolidate multiple tasks or operations into a single computational unit. This pattern can increase compute resource utilization, and reduce the costs and management overhead associated with performing compute processing in cloud-hosted applications.

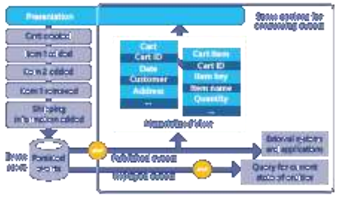


Command and Query Responsibility Segregation (CQRS)

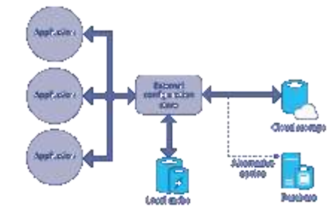
Segregate operations that read data from operations that update data by using separate interfaces. This pattern can maximize performance, scalability, and security; support evolution of the system over time through higher flexibility; and prevent update commands from causing merge conflicts at the domain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



Event Sourcing

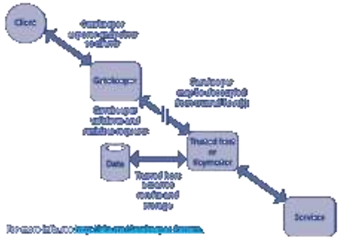


External Configuration Store



Federated Identity

Gatekeeper



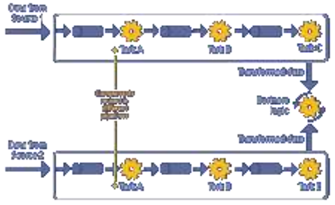
Health Endpoint Monitoring

Index Table

Leader Election

Materialized View

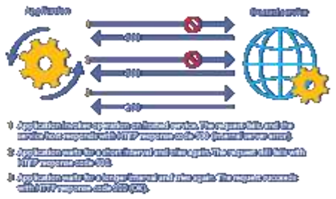
Pipes and Filters



Priority Queue

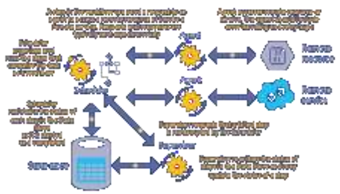
Queue-Based Load Leveling

Retry



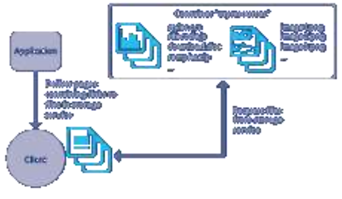
Runtime Reconfiguration

Scheduler Agent Supervisor

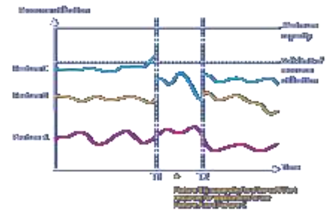


Sharding

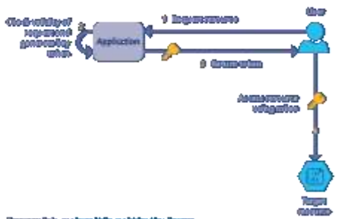
Static Content Hosting



Throttling



Valet Key



Some useful links

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

https://docs.microsoft.com/en-us/azure/architecture/

https://azureinteractives.azurewebsites.net/CloudDesignPatterns/default.html

“ Patterns

Categories

Availability

Data management

Design and implementation

Messaging

Management and monitoring

Performance and scalability

Resiliency

Security

Availability

Availability defines the proportion of time that the system is functional and working...

Health Endpoint Monitoring

Queue-Based Load Leveling

Throttling

Data Management

Data management is the key element of cloud applications, and influences most of the quality attributes…

- Cache-Aside
- Command and Query Reponsibility
- Event Sourcing
- Index Table
- Index Table
- Materialized View
- Sharding
- Static Content Hosting
- Valet Key

Design and Implementation

Good design encompasses factors…

…Decisions made during the design and implementation phase have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.

Ambassador
Anti Corruption Layer
Backends for Frontends
Command And Query Responsibility
Computer Resource Consolidation
External Configuration Store
Gateway Aggregation
Gateway Offloading
Gateway Routing
Leader Election
Pipes and Filters
Sidecar
Static Conten Hosting
Strangler

Messaging

The distributed nature of cloud applications requires a messaging infrastructure that connects the components and services, ideally in a loosely coupled manner in order to maximize scalability. …

Compensating Consumers

Pipes and Filters

Priority Queue

Queue-Based Load Leveling

Scheduler Agent Supervisor

Management and Monitoring

Cloud applications run in a remote datacenter where you do not have full control of the infrastructure or, in some cases, the operating system.
…

Ambassador
Anti Corruption Layer
External Configuration Store
Gateway Aggregation
Gateway Offloading
Gateway Routing
Healt Endpoint Monitoring
Sidecar
Strangler

Performance and Scalability

Performance is an indication of the responsiveness of a system, while scalability is the ability to gracefully handle increases in load, perhaps through an increase in available resources…

- Cache-aside
- Command and Query Responsibility
- Compensating Consumers
- Index Table
- Materialized View
- Priority Queue
- Sharding
- Static Content Hosting
- Throttling

Resiliency

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

- Bulkhead
- Circuit Breaker
- Compensating Transaction
- Leader Election
- Retry
- Scheduler Agent Supervisor

Security

Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. …

Federated Identity

Gatekeeper

Valet key

“ A scenario

Room reservation

Reserve a room from office

See a panel on the room to see the daily reservations

Classic approach

Client/server or three tier

Db/centric➔one db for all

Issues

- Coupling
- Integrity
- Performances
- Scalability
- Availability
- Single point of failure

Modern solution

four apps
Reservation
Worker
Handler
Panel
Four data stores
requests
Write
Events
read

Cons:

- Initial Higher cost

Pro:

- Long term lower cost
- Flexible
- Maintainable
- Scalable
- Available

“ Our solution

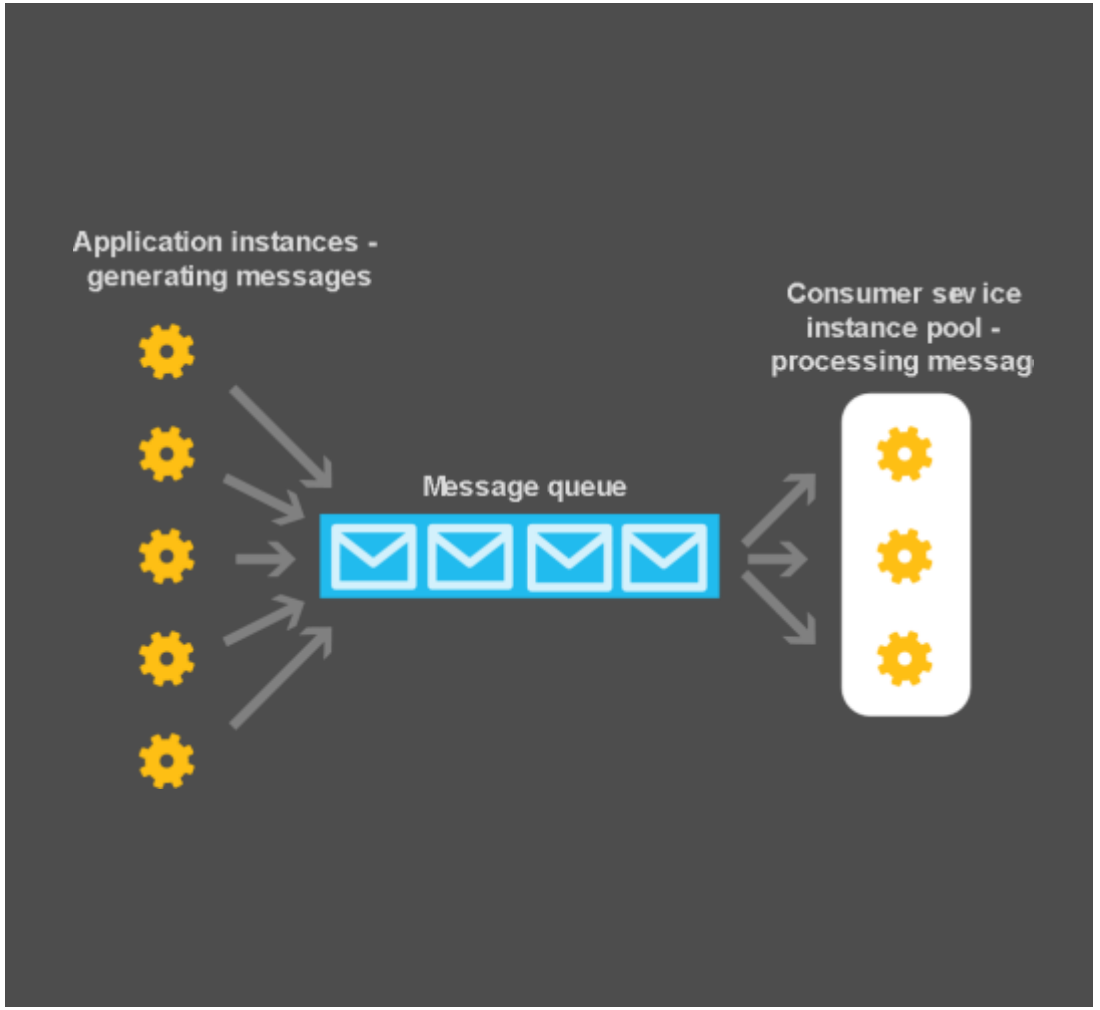
Queue-Based Load Leveling

From Availability and Performances

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

Compensating Consumers

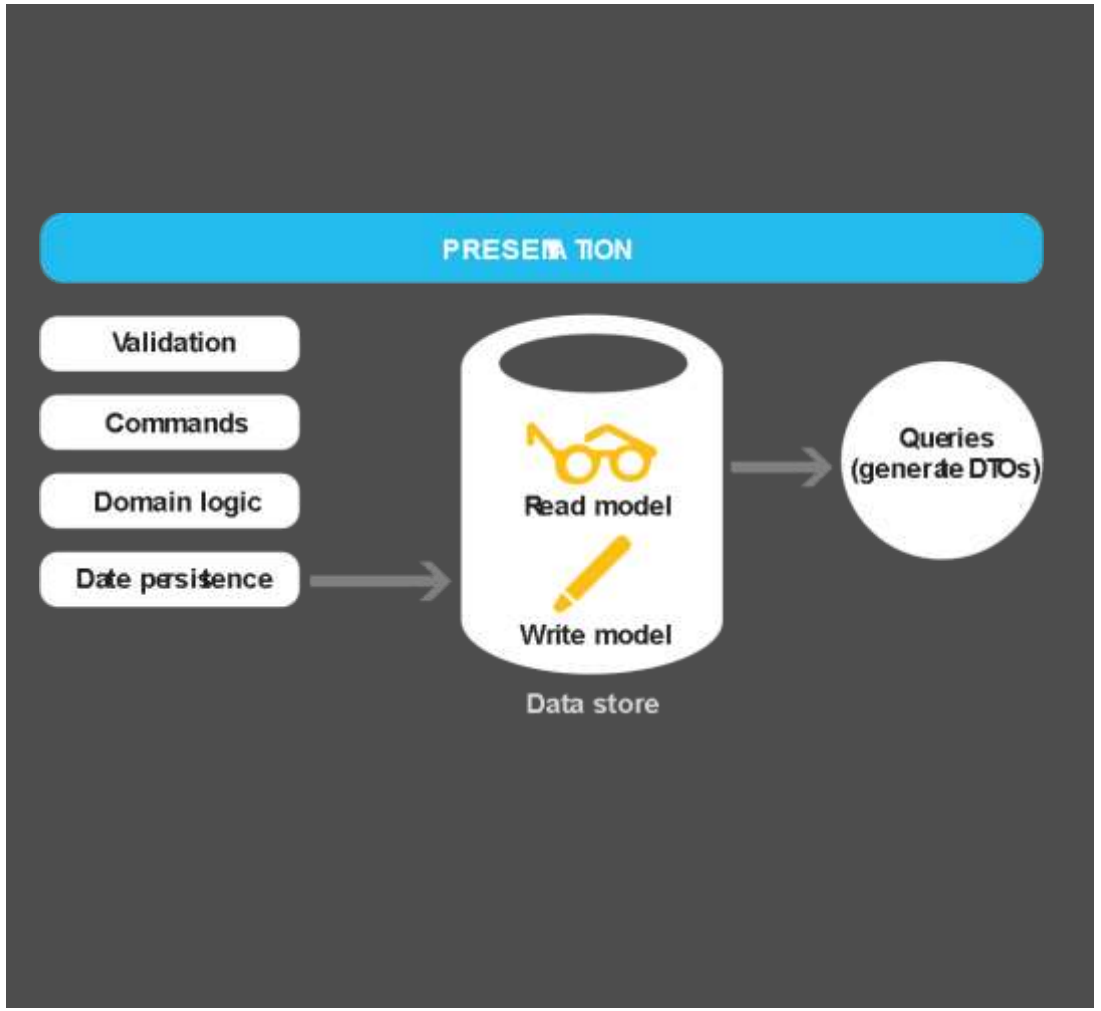
Enable multiple concurrent consumers to process messages received on the same messaging channel. This pattern enables a system to process multiple messages concurrently to optimize throughput, to improve scalability and availability, and to balance the workload.



“That’s why you have (lot of) worker roles, containers, actors and microservices…

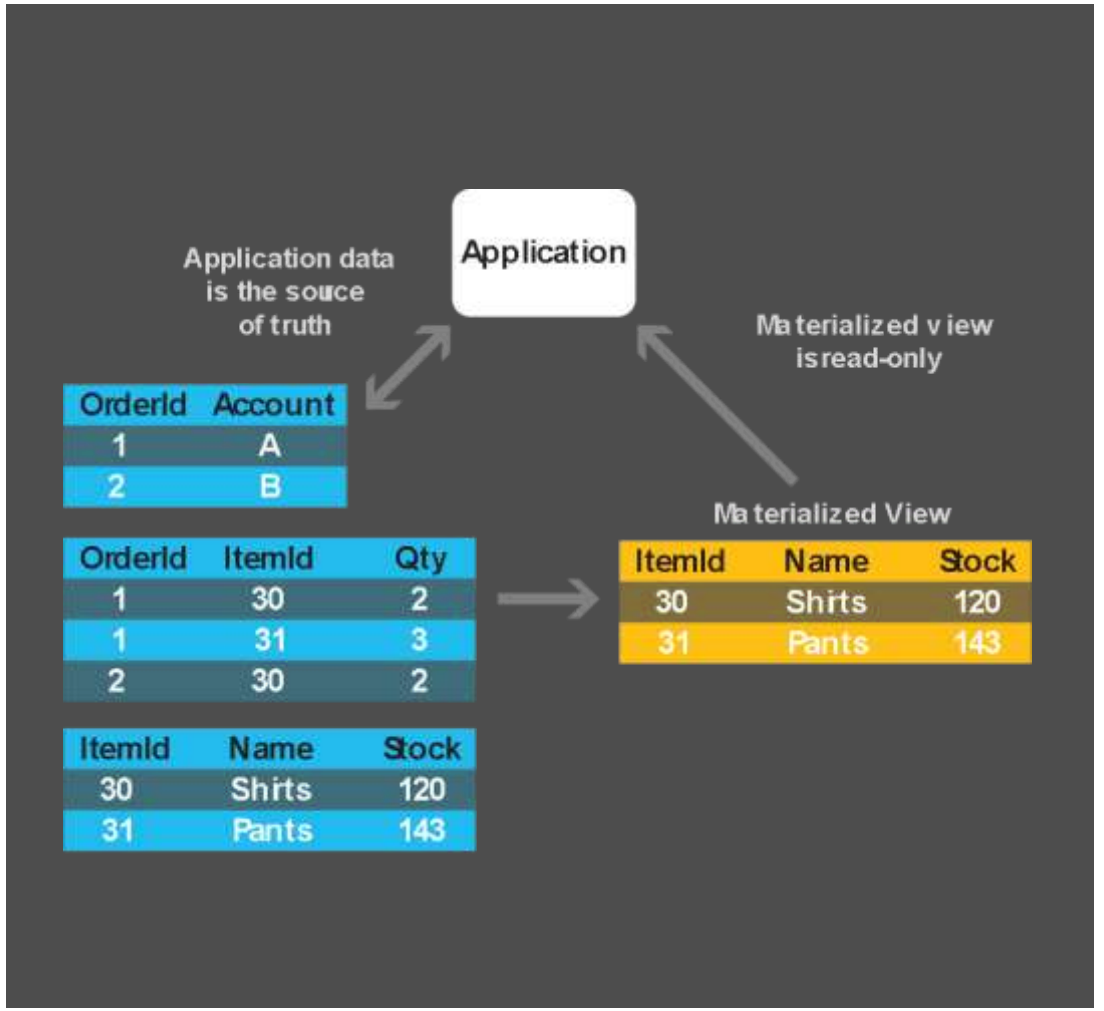
Command and Query Responsibility Segregation (CQRS)

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



Materialized View

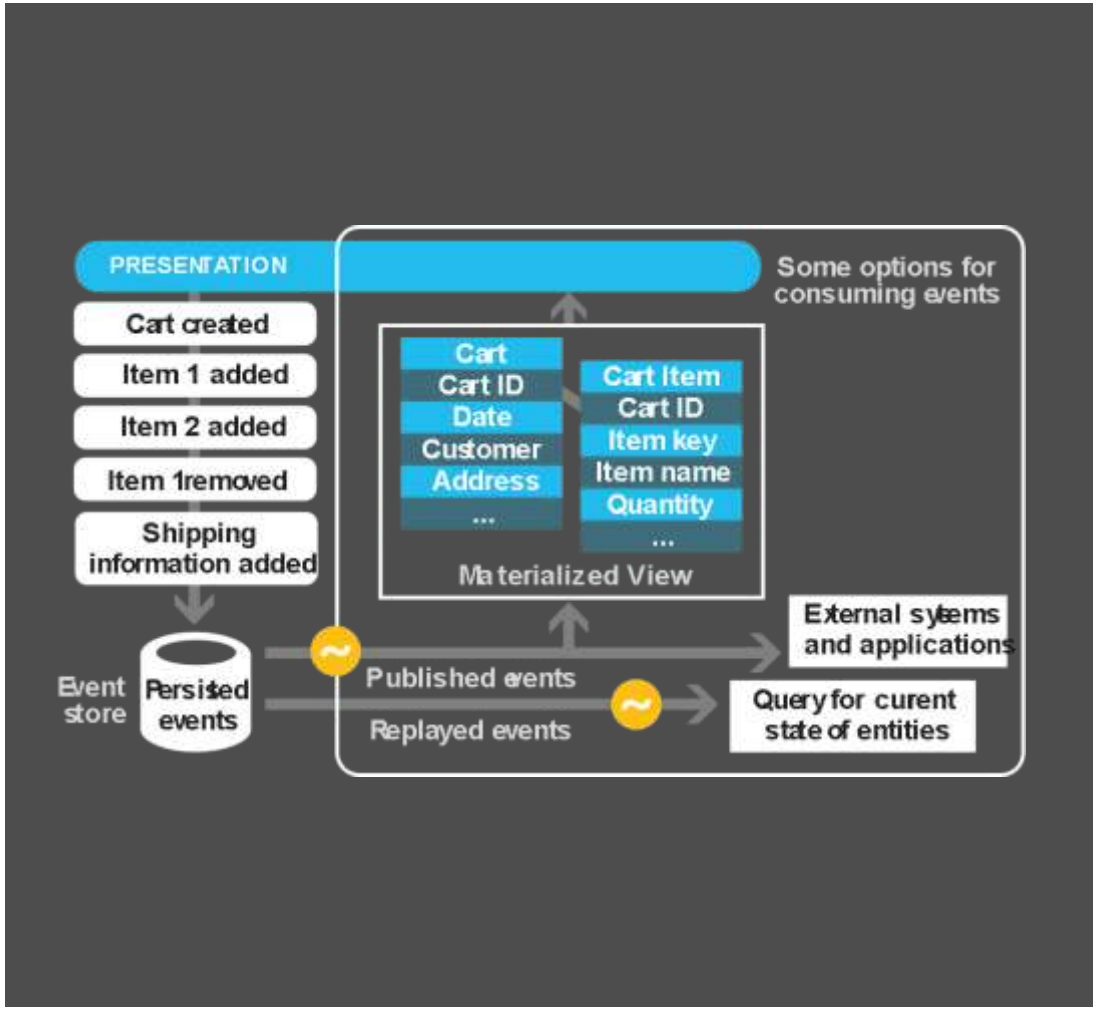
Generate pre-populated views over the data in one or more data stores when the data is formatted in a way that does not favor the required query operations. This pattern can help to support efficient querying and data extraction, and improve performance.



> That’s why there are a lot o data services (No SQL, in memory, graph, …)

Event Sourcing

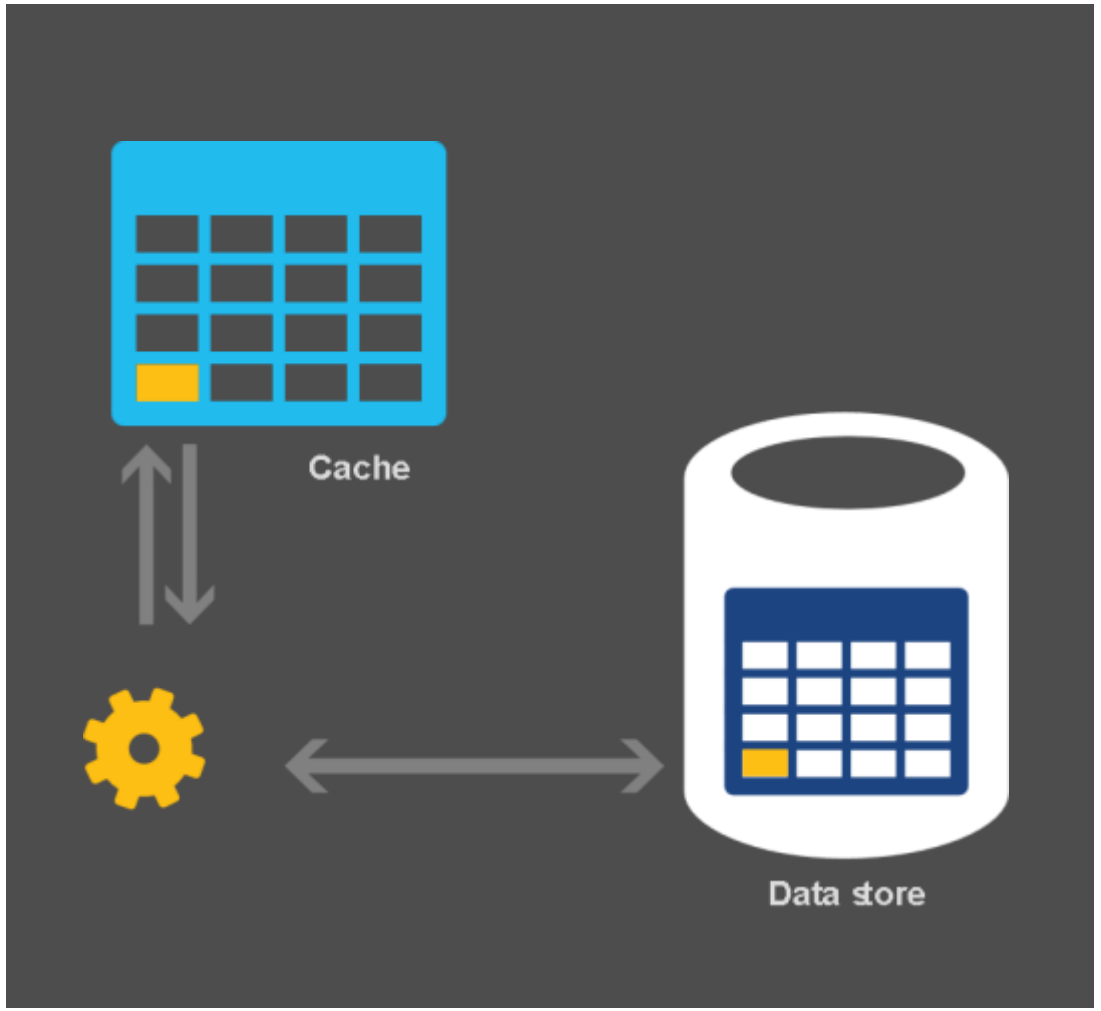
Use an append-only store to record actions taken on data, rather than the current state, and use the store to materialize the domain objects. In complex domains this can avoid synchronizing the data model and the business domain; improve performance, scalability, and responsiveness; provide consistency; and provide audit history to enable compensating actions.



“That’s why you have event-based services like Functions, Logic Apps, Event Grid and Serverless!!!

Cache-aside

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



“ Conclusions

> Investing in patterns means being natural over cloud

“ Find the patterns you are confortable

> Patterns change the way you think software development, not only cloud

Grazie

Domande?


marcoparenzan


marco_parenzan


marcoparenzan

“ Patterns

“Availability

Availability

Availability defines the proportion of time that the system is functional and working

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

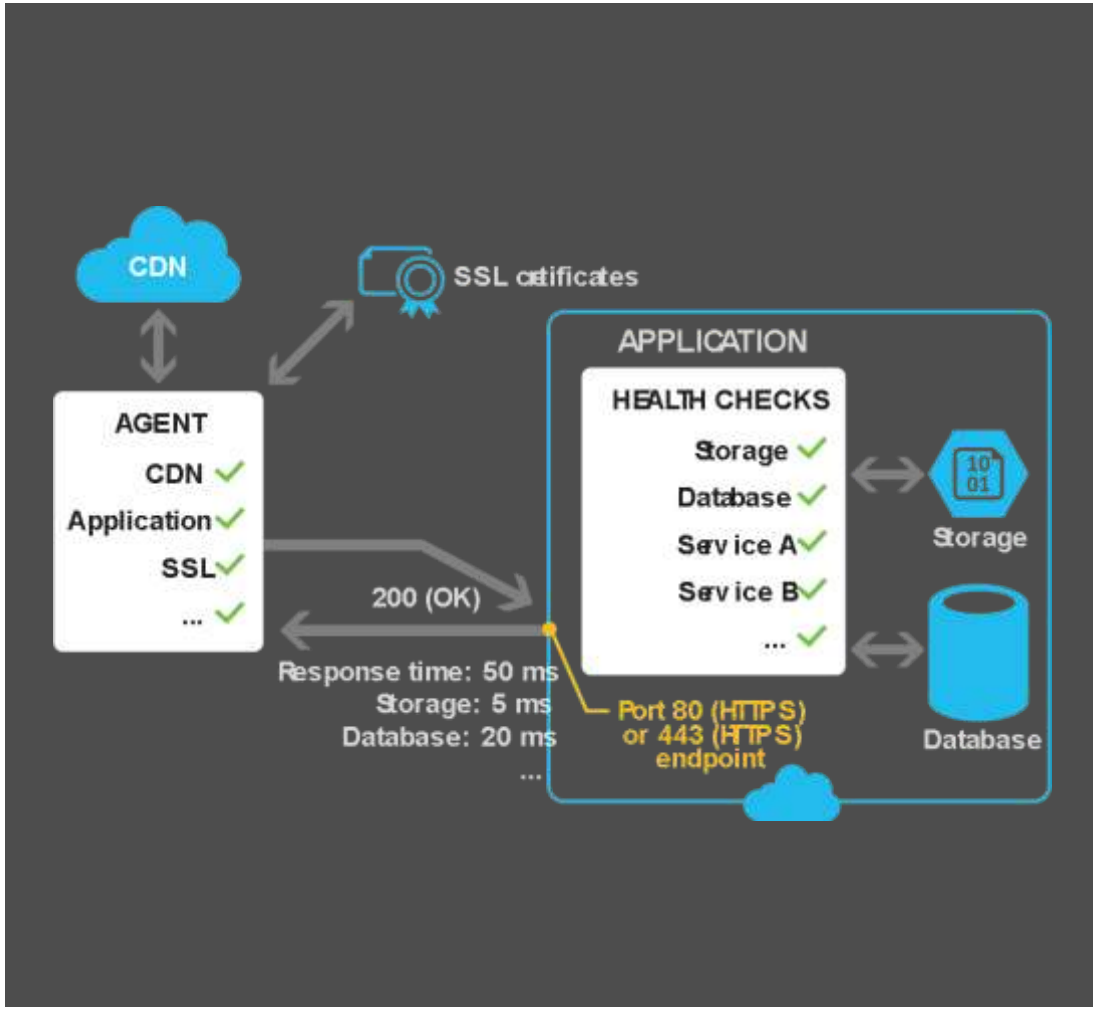
Health Endpoint Monitoring

Queue-Based Load Leveling

Throttling

Health Endpoint Monitoring

Implement functional checks within an application that external tools can access through exposed endpoints at regular intervals. This pattern can help to verify that applications and services are performing correctly.

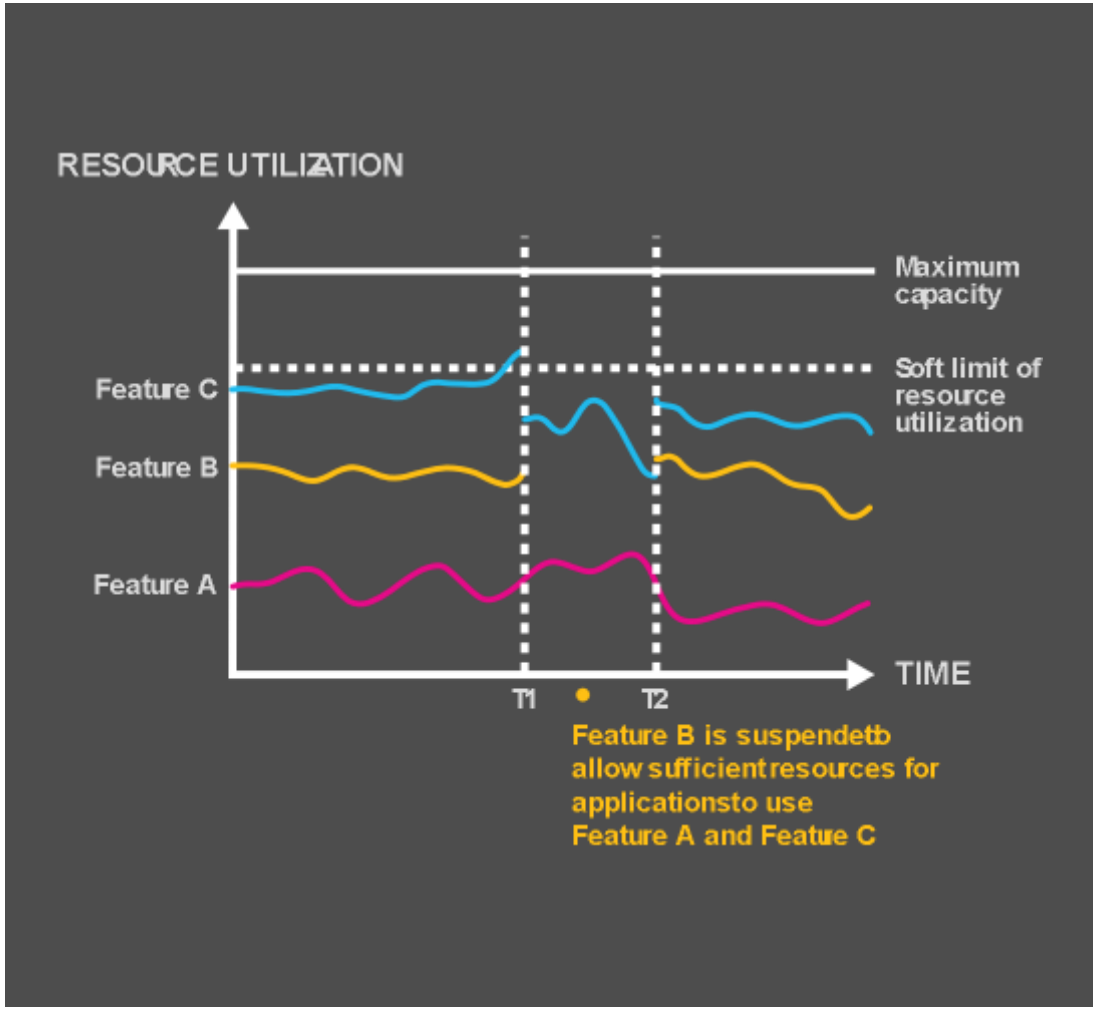


Queue-Based Load Leveling

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

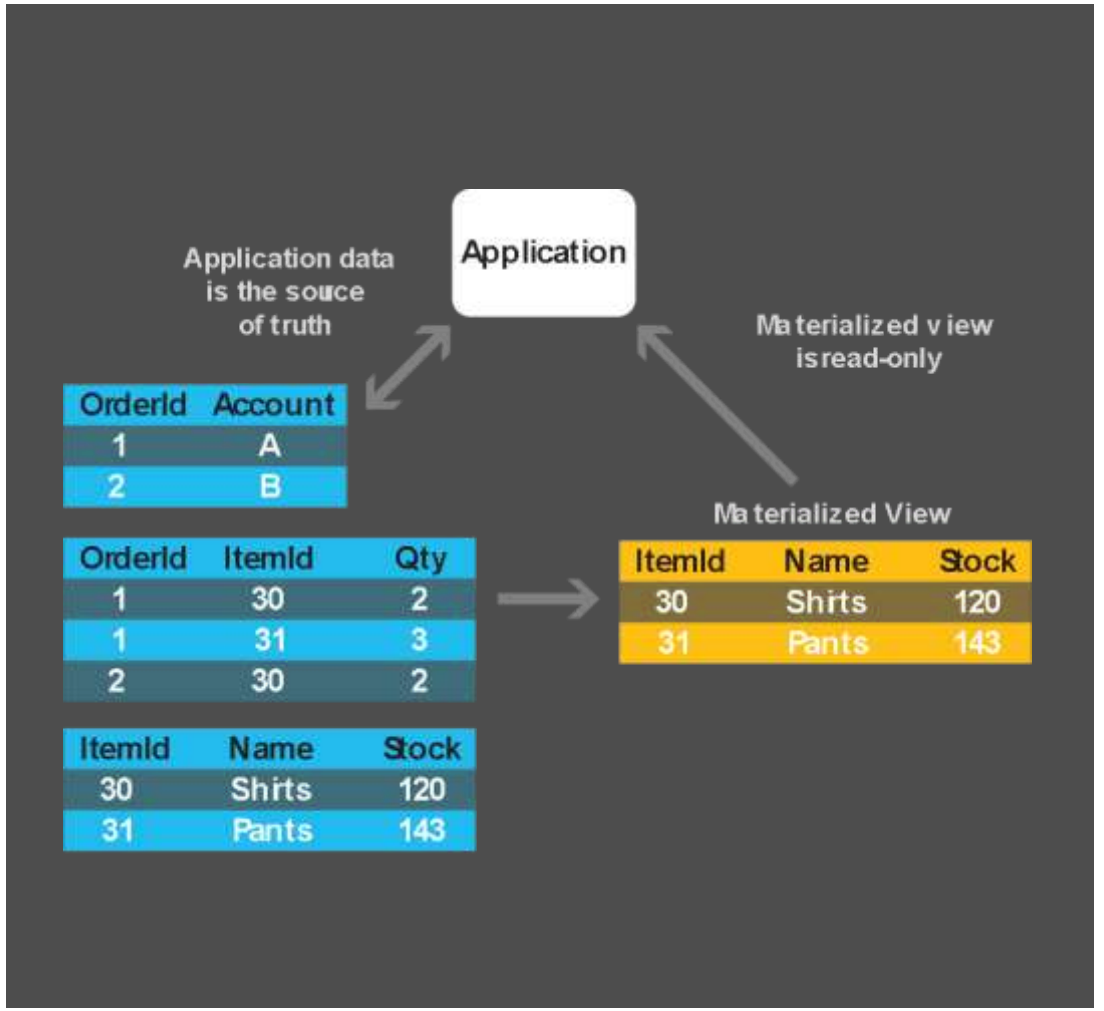
Throttling

Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service. This pattern 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.



Materialized View

Generate pre-populated views over the data in one or more data stores when the data is formatted in a way that does not favor the required query operations. This pattern can help to support efficient querying and data extraction, and improve performance.



“ Data Management

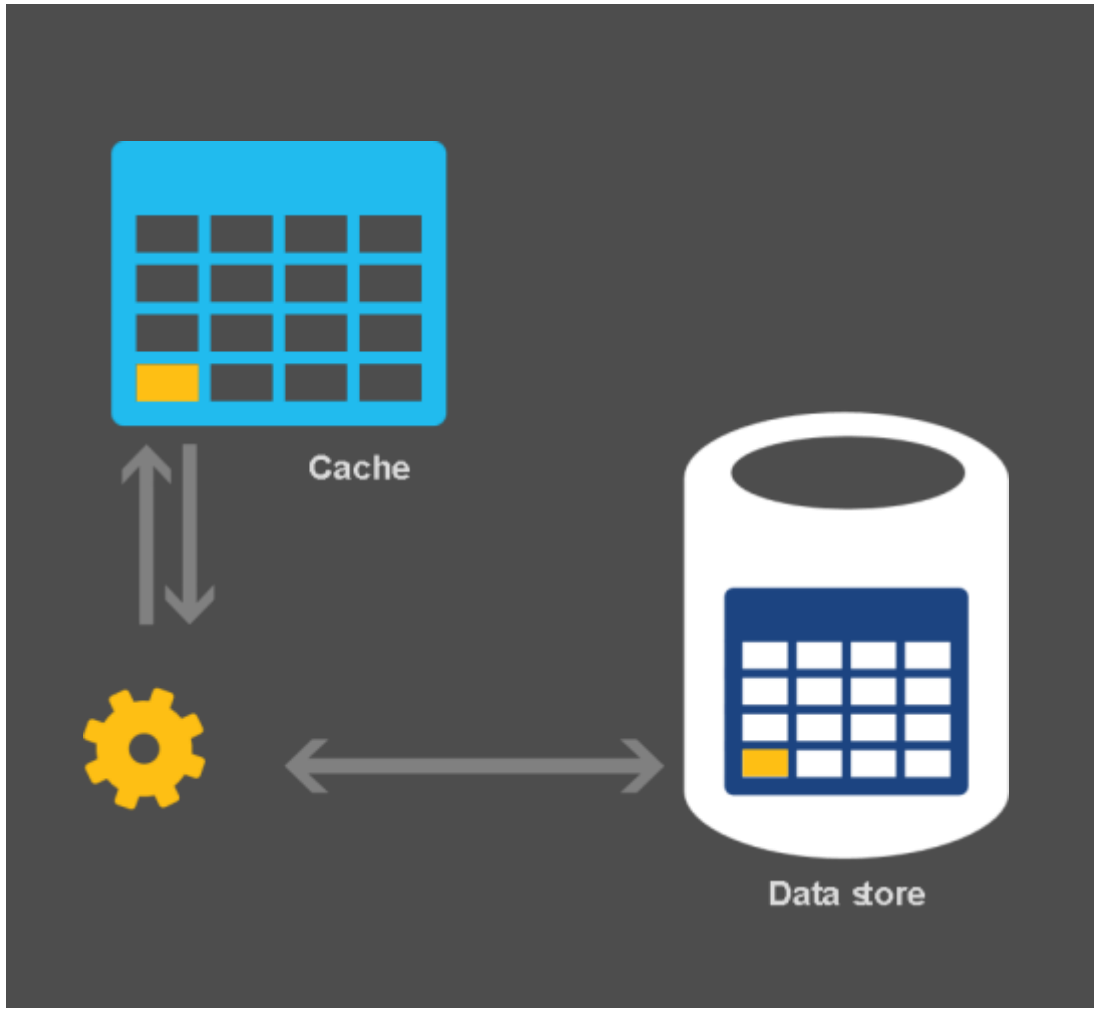
Data Management

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

- Cache-Aside
- Command and Query Reponsibility
- Event Sourcing
- Index Table
- Index Table
- Materialized View
- Sharding
- Static Content Hosting
- Valet Key

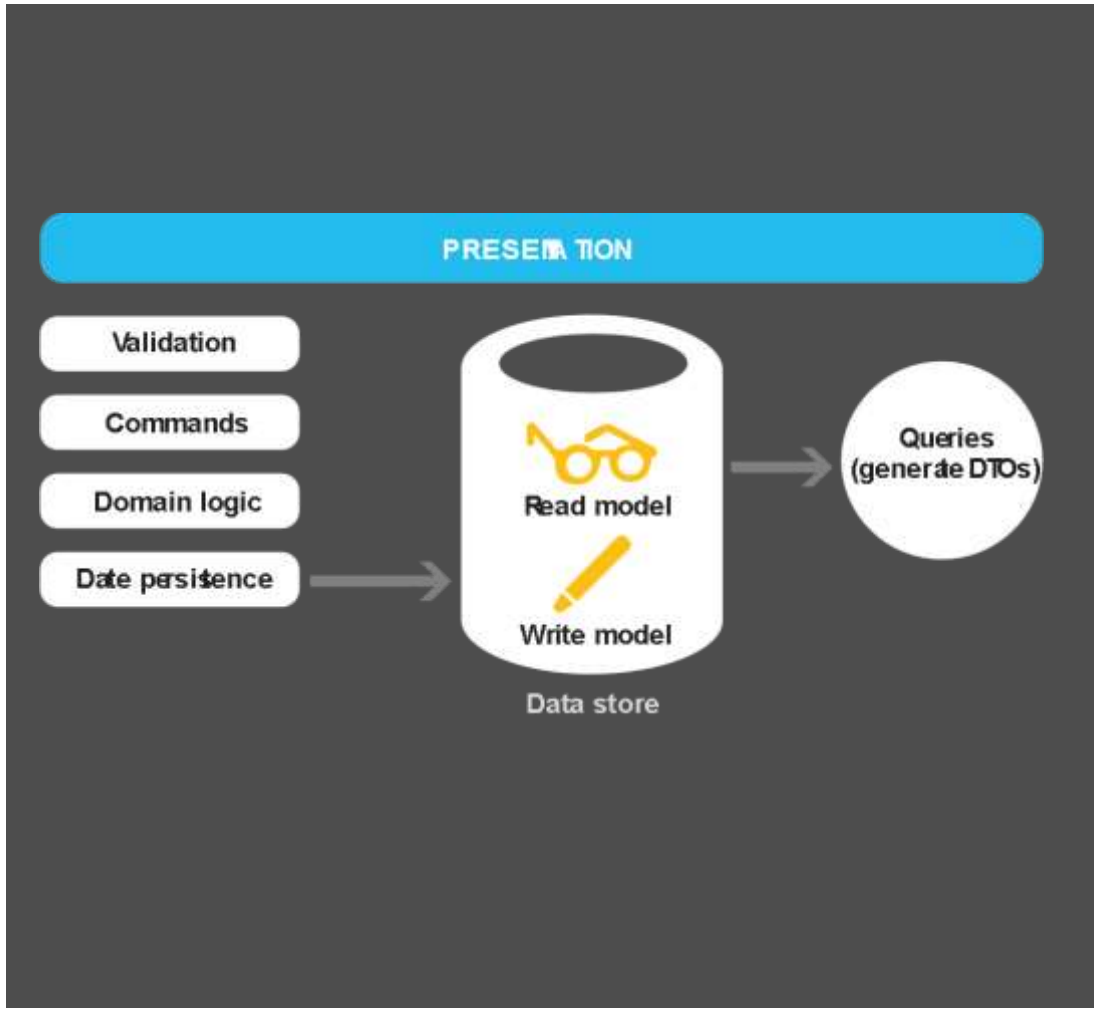
Cache-aside

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



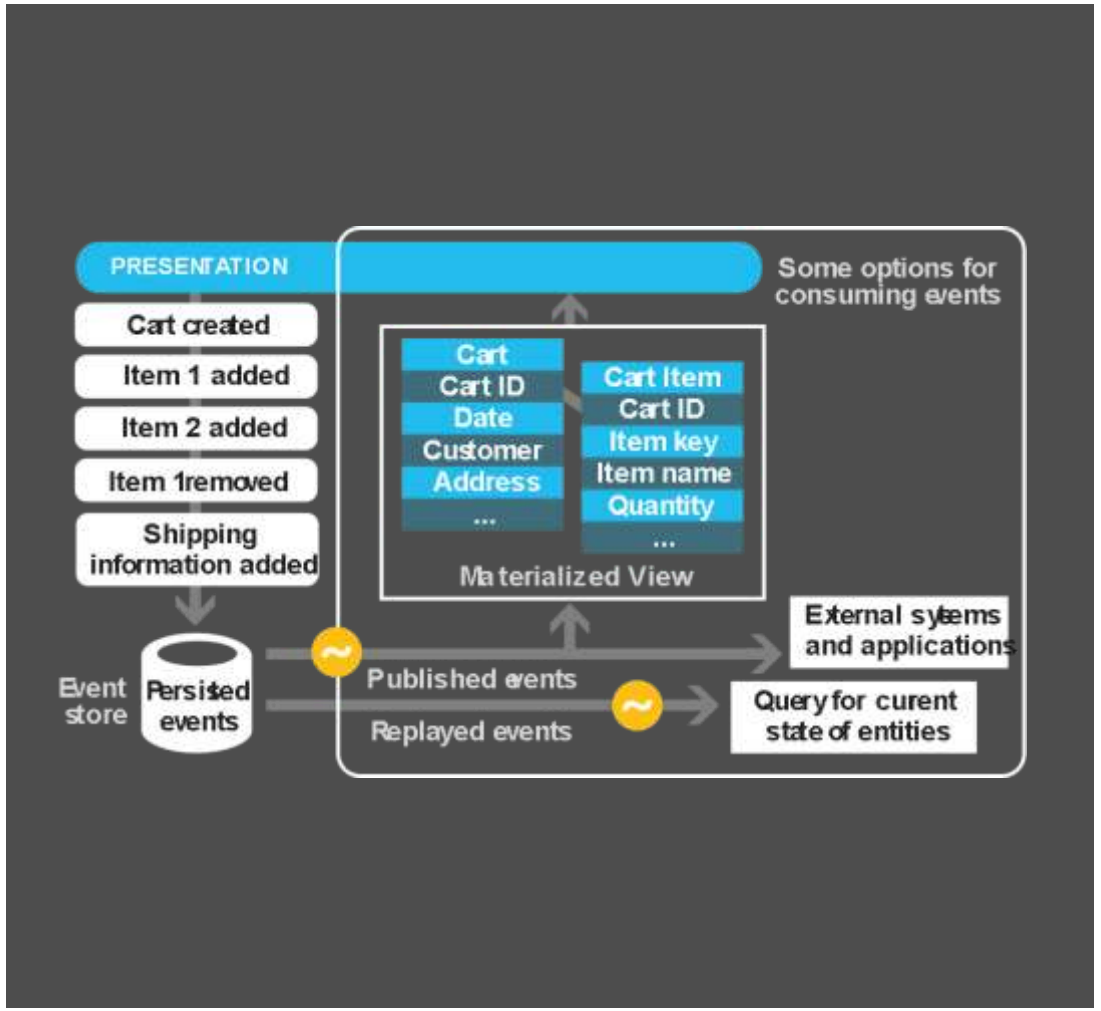
Command and Query Responsibility Segregation (CQRS)

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



Event Sourcing

Use an append-only store to record actions taken on data, rather than the current state, and use the store to materialize the domain objects. In complex domains this can avoid synchronizing the data model and the business domain; improve performance, scalability, and responsiveness; provide consistency; and provide audit history to enable compensating actions.

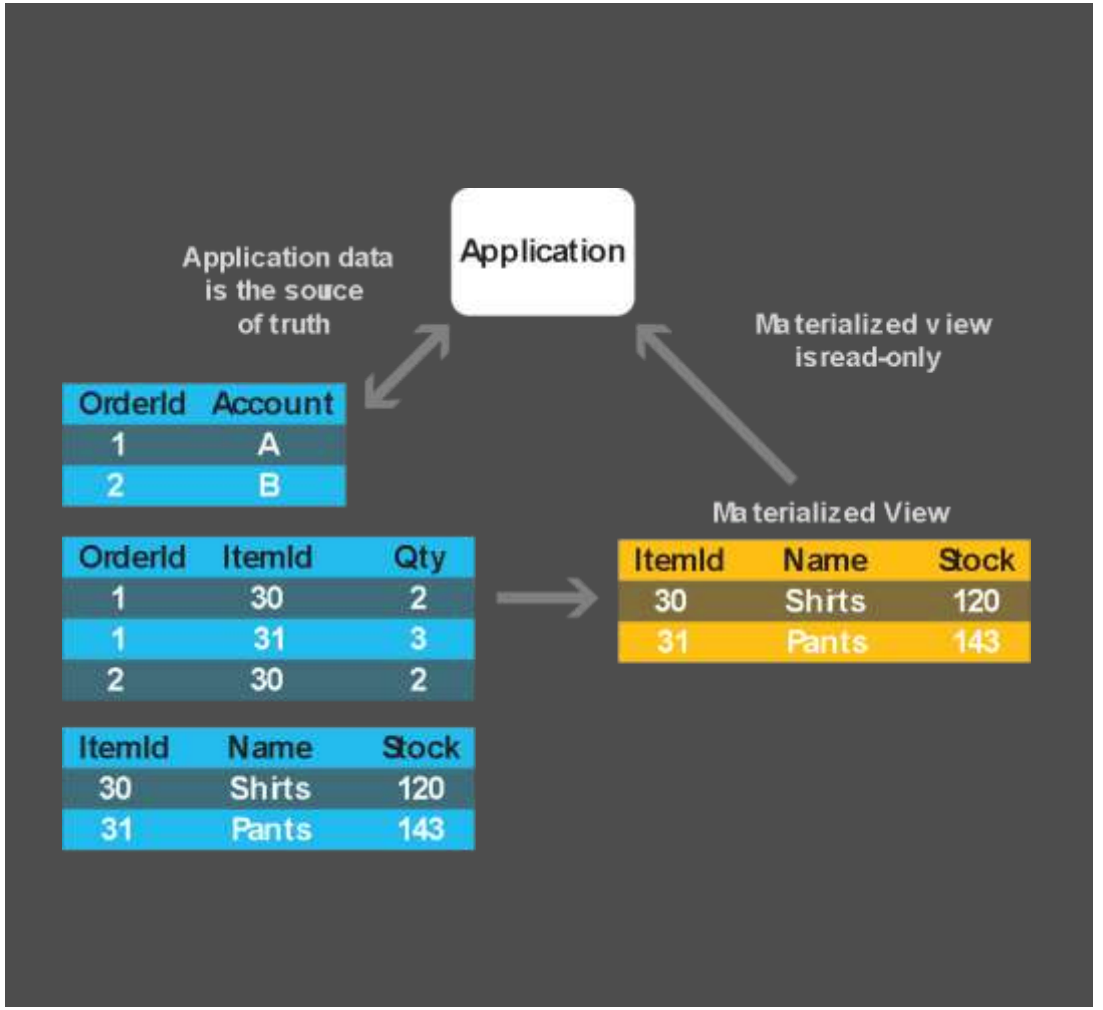


Index Table

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

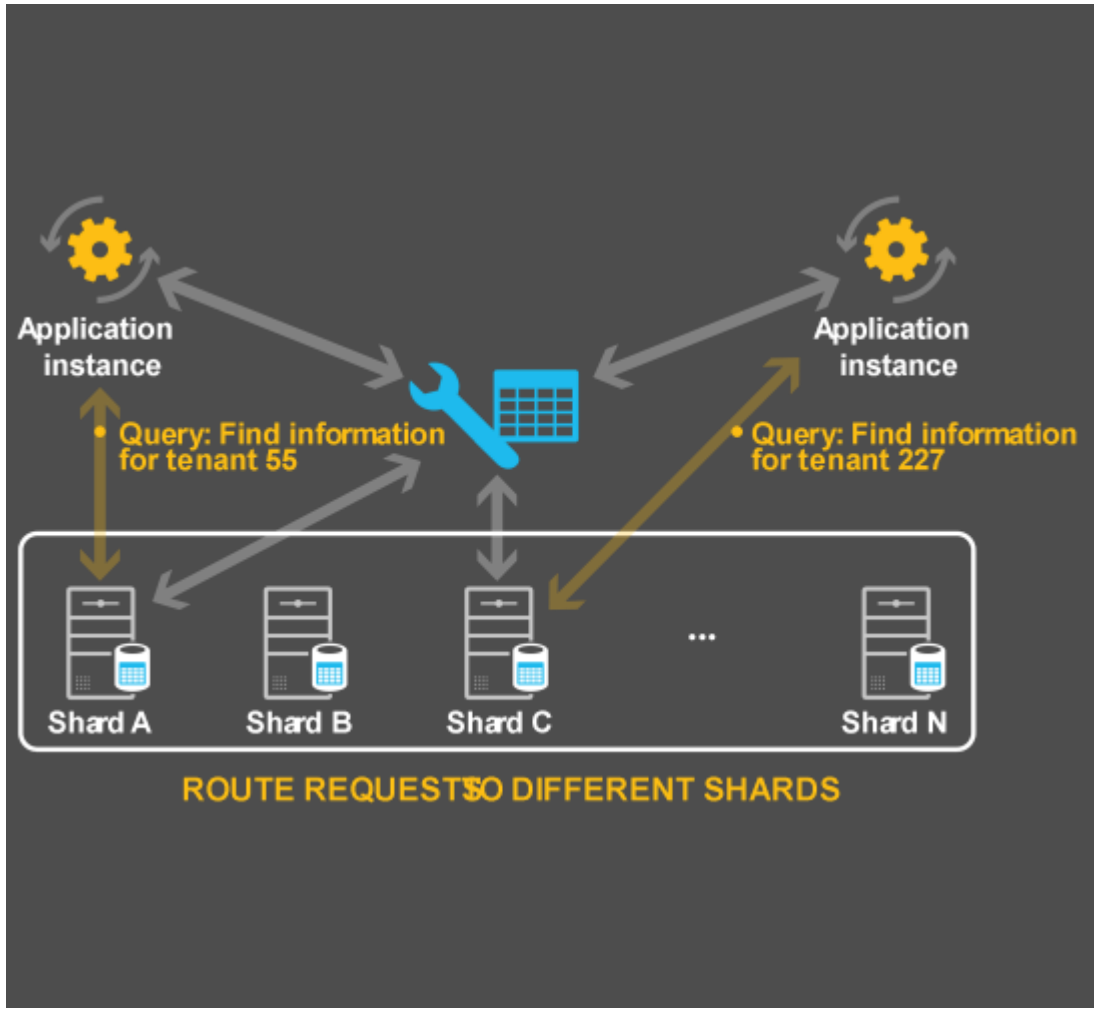
Materialized View

Generate pre-populated views over the data in one or more data stores when the data is formatted in a way that does not favor the required query operations. This pattern can help to support efficient querying and data extraction, and improve performance.

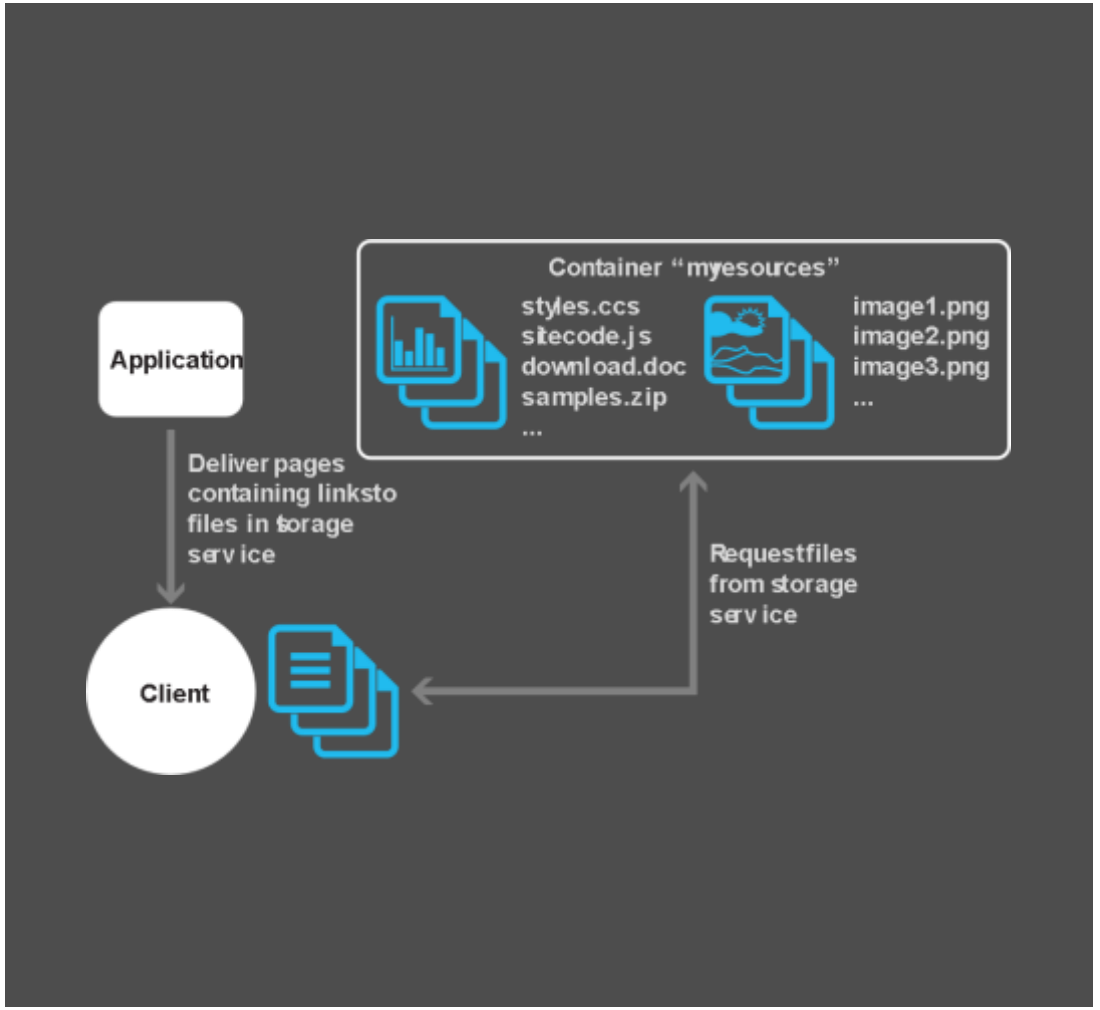


Sharding

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

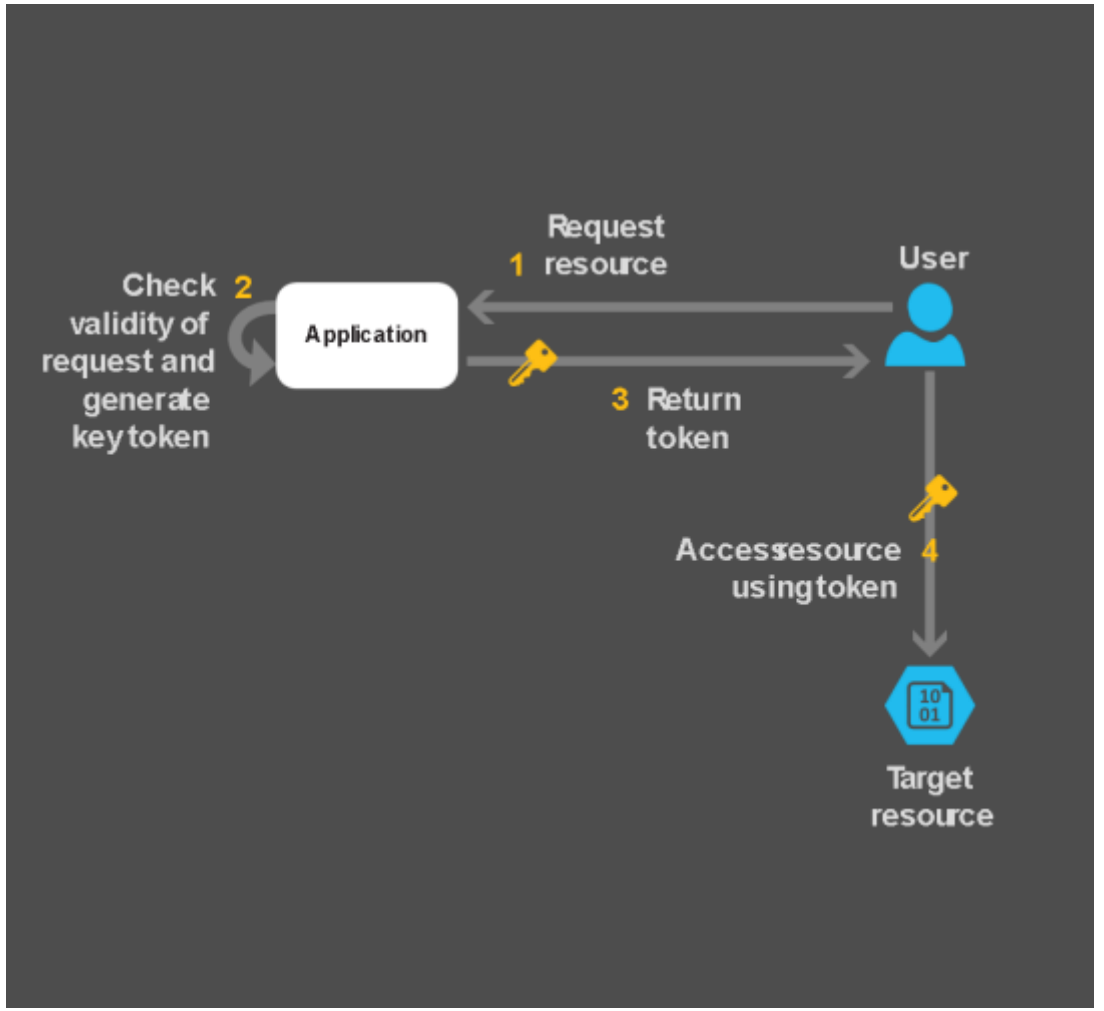


Static Content Hosting



Valet key

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



“ Design and Implementation

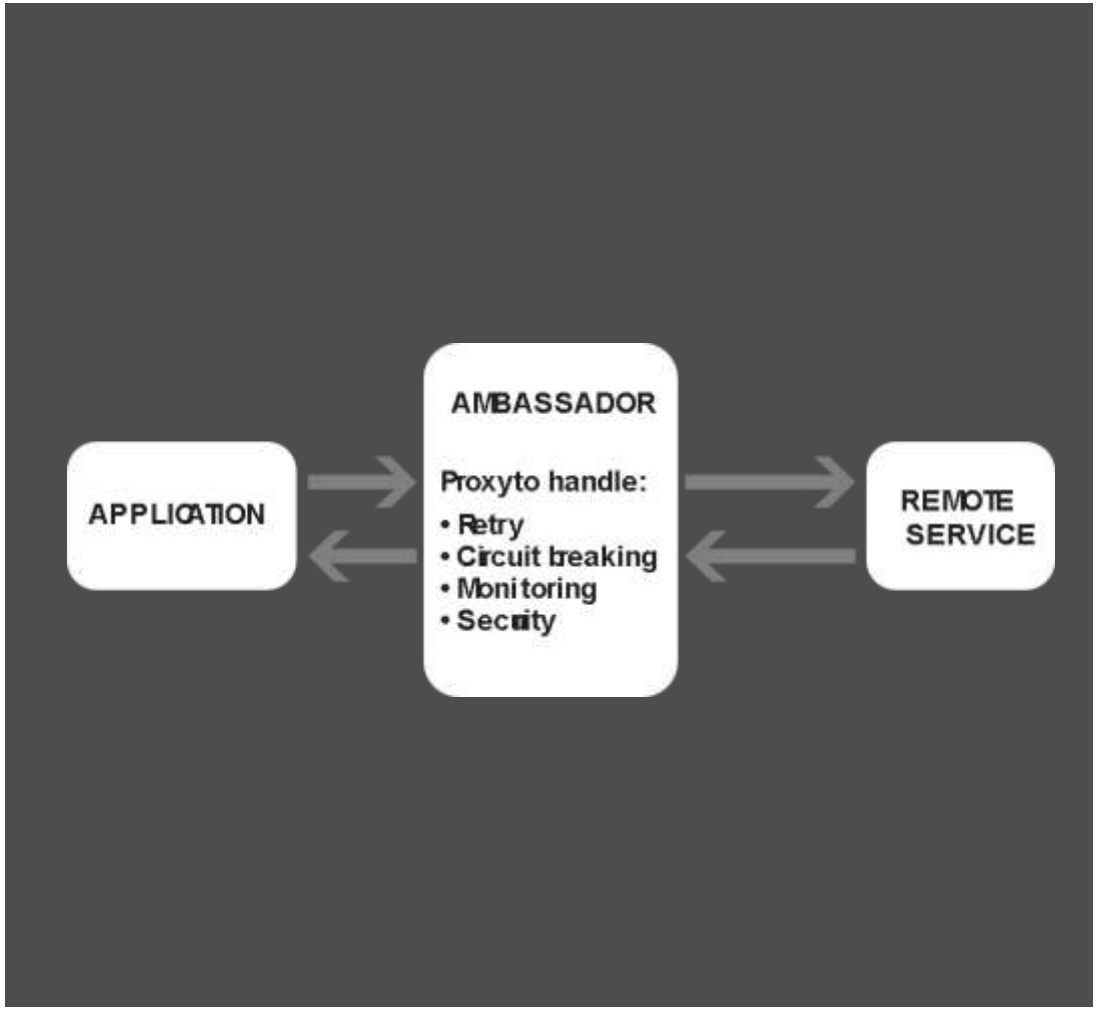
Design and Implementation

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

Ambassador
Anti Corruption Layer
Backends for Frontends
Command And Query Responsibility
Computer Resource Consolidation
External Configuration Store
Gateway Aggregation
Gateway Offloading
Gateway Routing
Leader Election
Pipes and Filters
Sidecar
Static Conten Hosting
Strangler

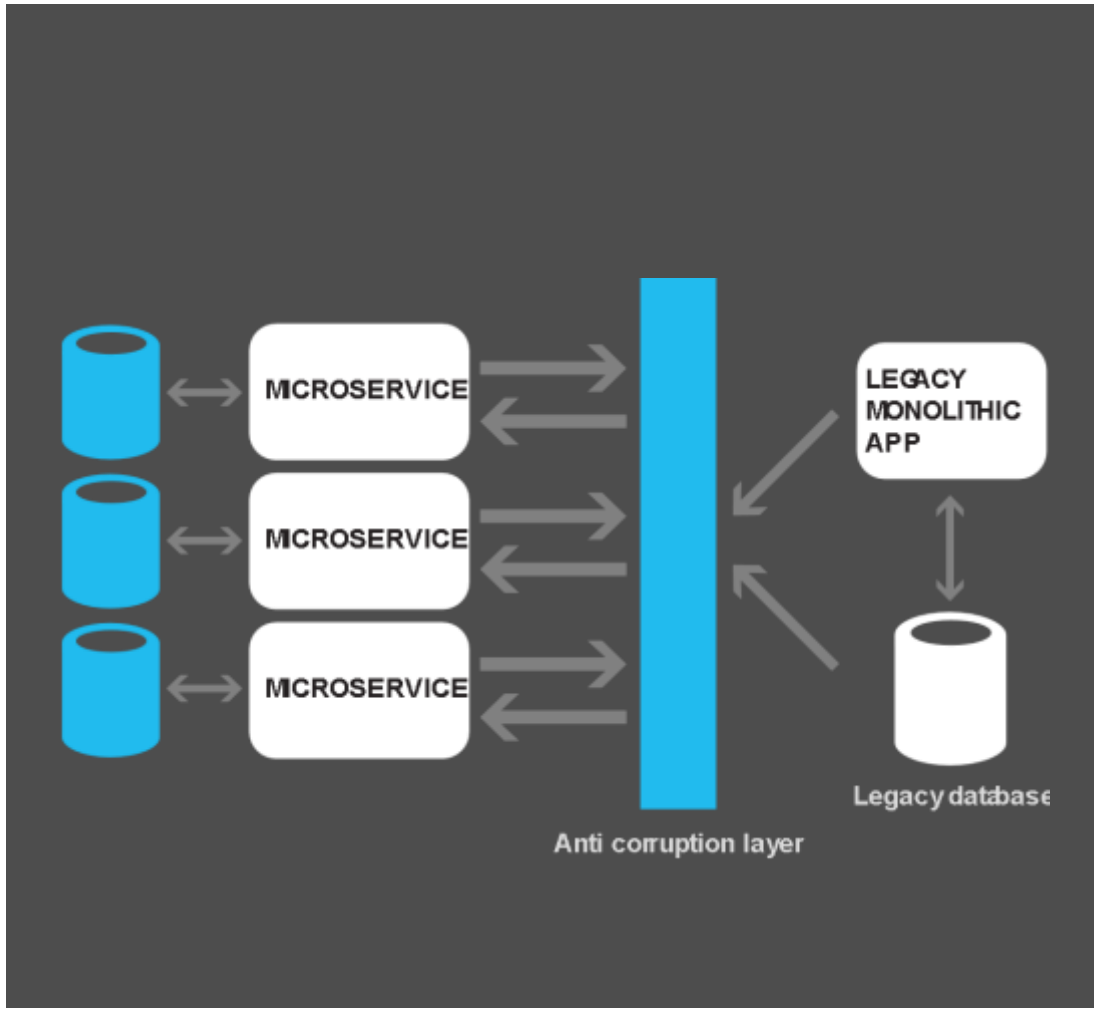
Ambassador

Create helper services for making networking 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.



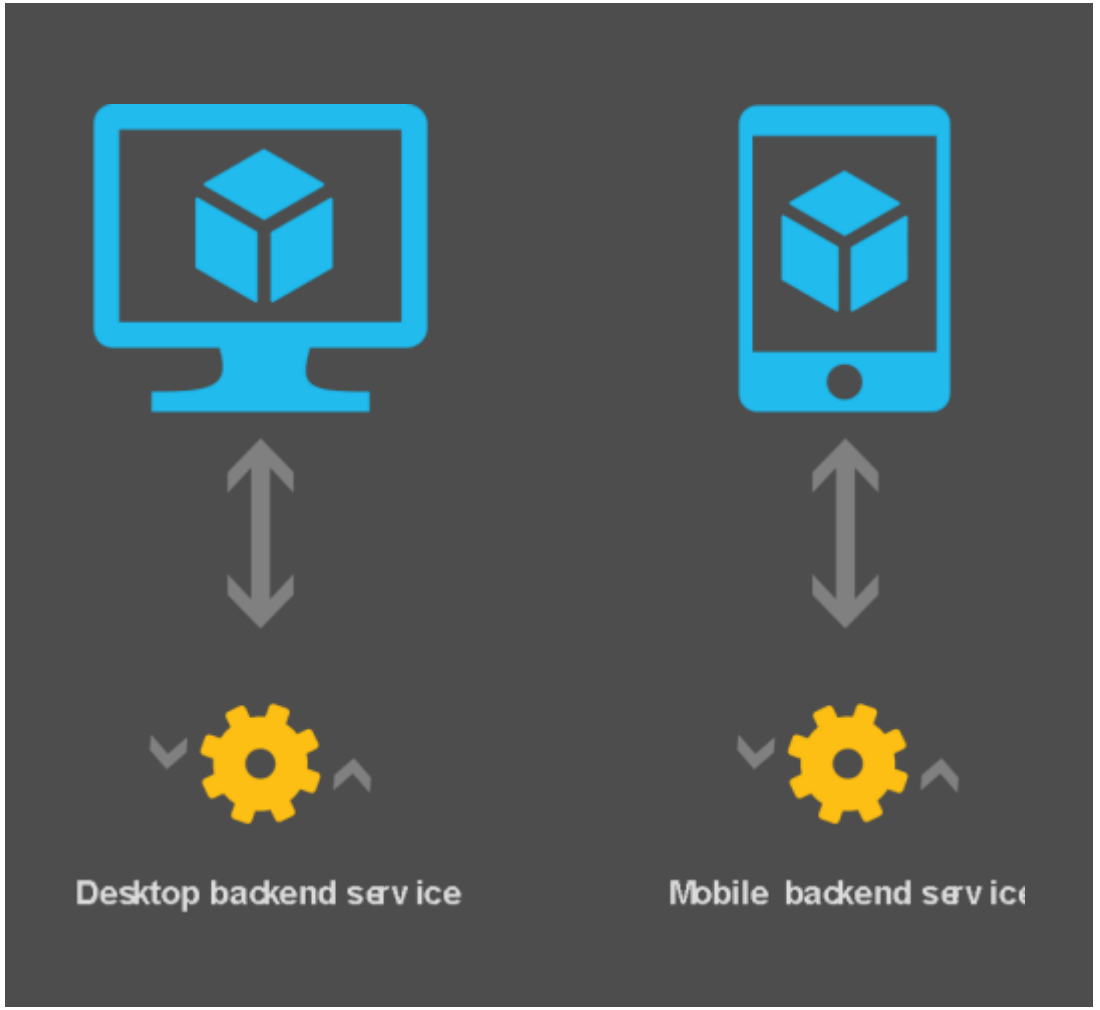
Anti Corruption Layer

Ensure the design and structure of new apps and services are not limited by legacy resources. Implement a façade or adapter layer in between legacy systems and modern systems where features and capabilities are being migrated to. This façade or layer translates requests between modern systems and legacy applications using methods appropriate to each.



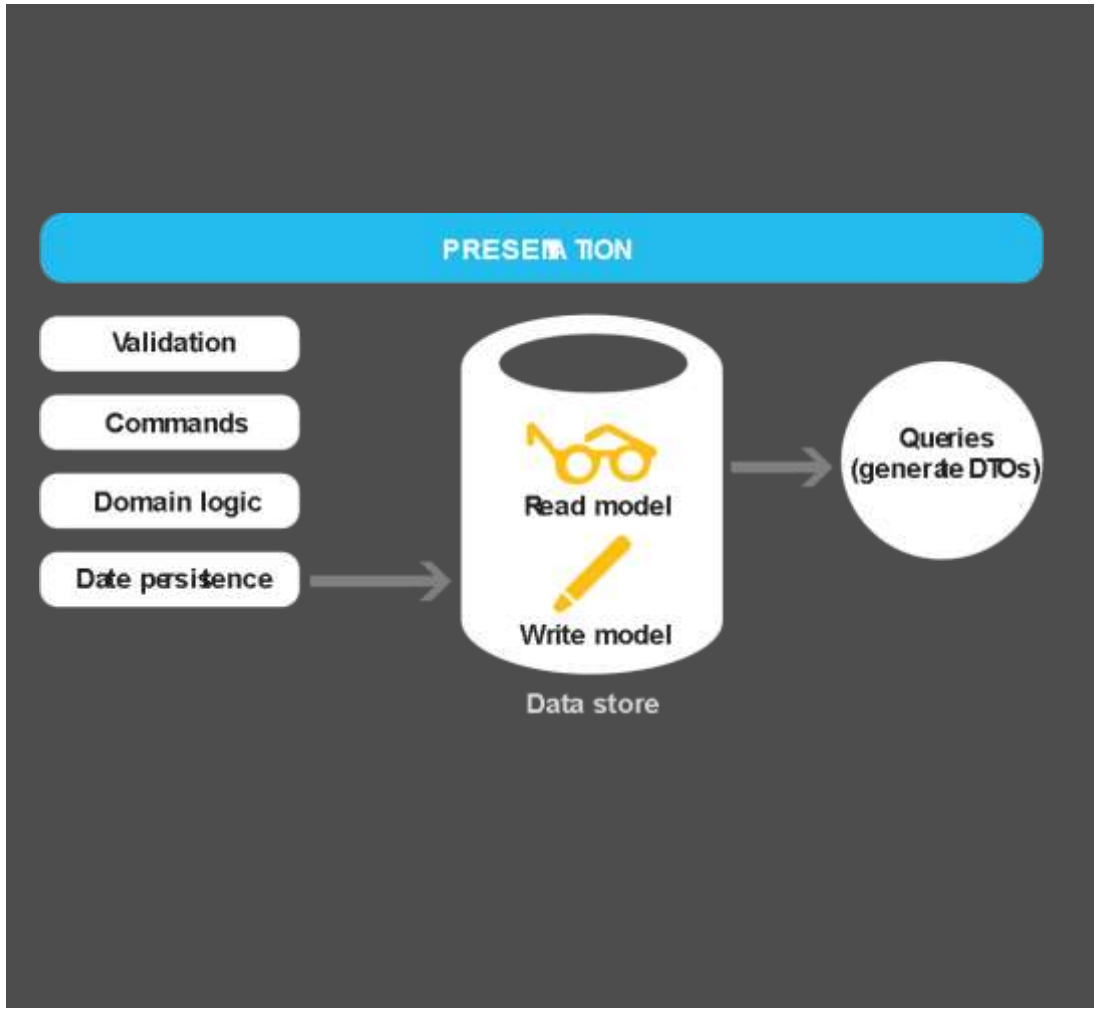
Backends for Frontends

Develop targeted backend services for use with specific frontend applications or interfaces. This pattern is useful when you wish to avoid implementing customizations for multiple interfaces in a single backend.



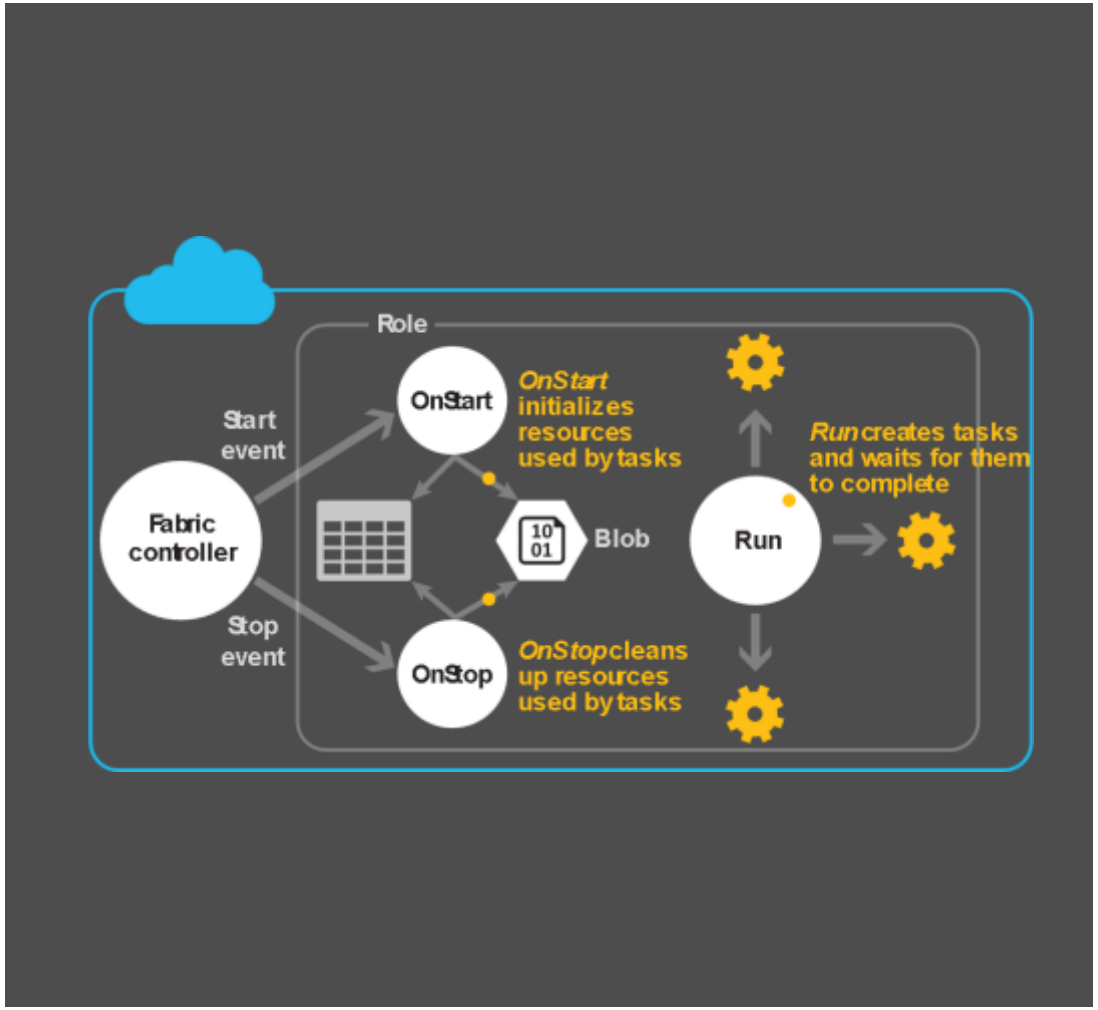
Command and Query Responsibility Segregation (CQRS)

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



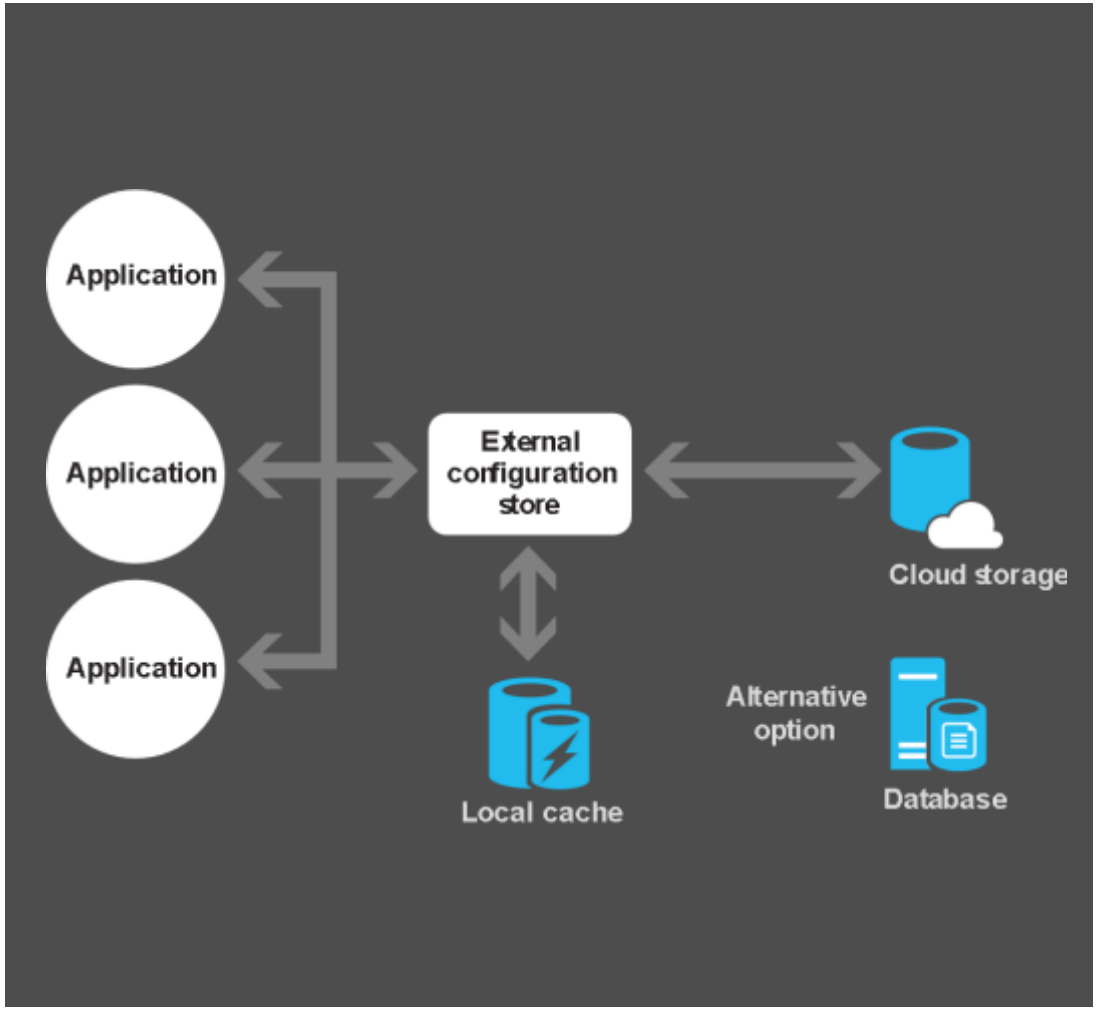
Compute Resource Consolidation

Consolidate multiple tasks or operations into a single computational unit. This pattern can increase compute resource utilization, and reduce the costs and management overhead associated with performing compute processing in cloud-hosted applications.



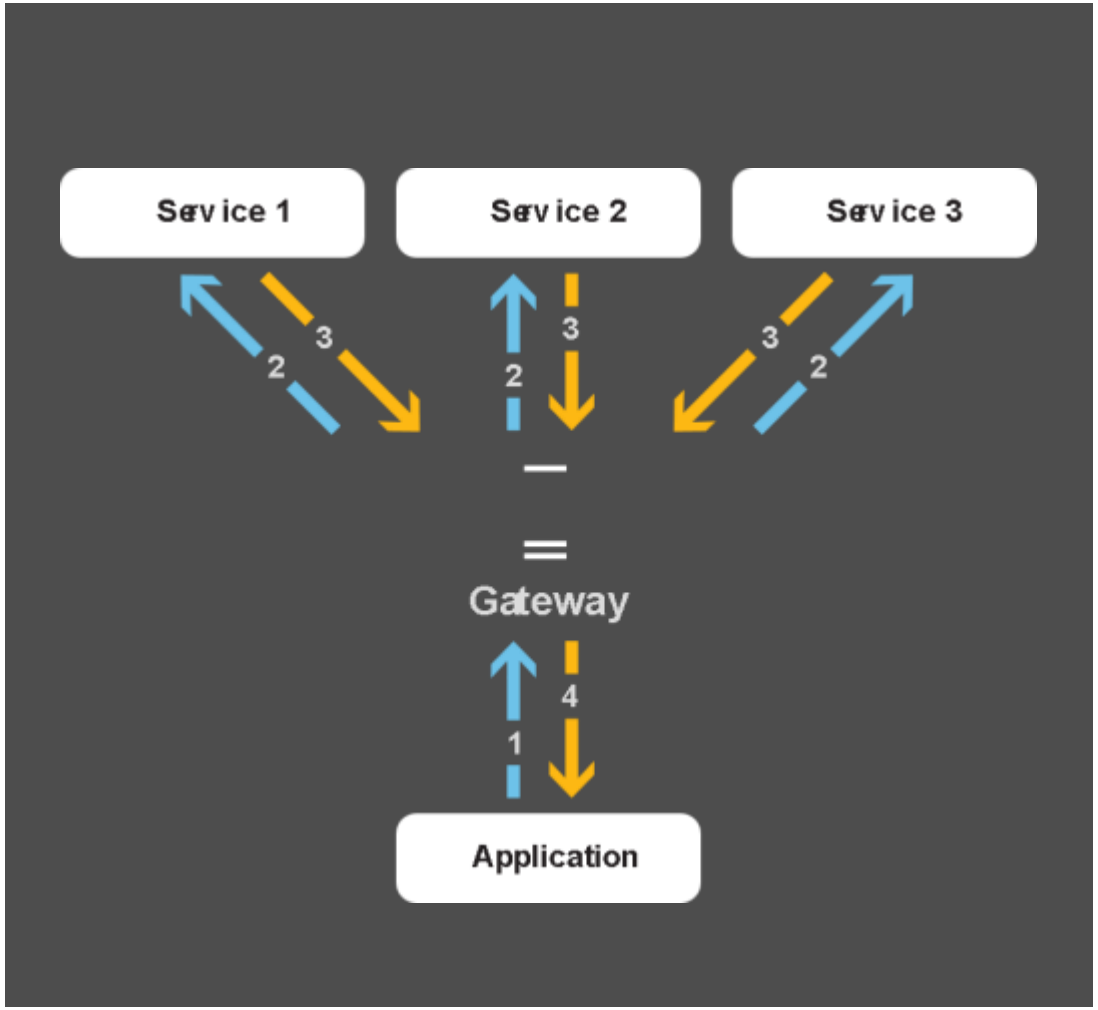
External Configuration Store

Move configuration information out of the application deployment package to a centralized location. This pattern can provide opportunities for easier management and control of configuration data, and for sharing configuration data across applications and application instances.



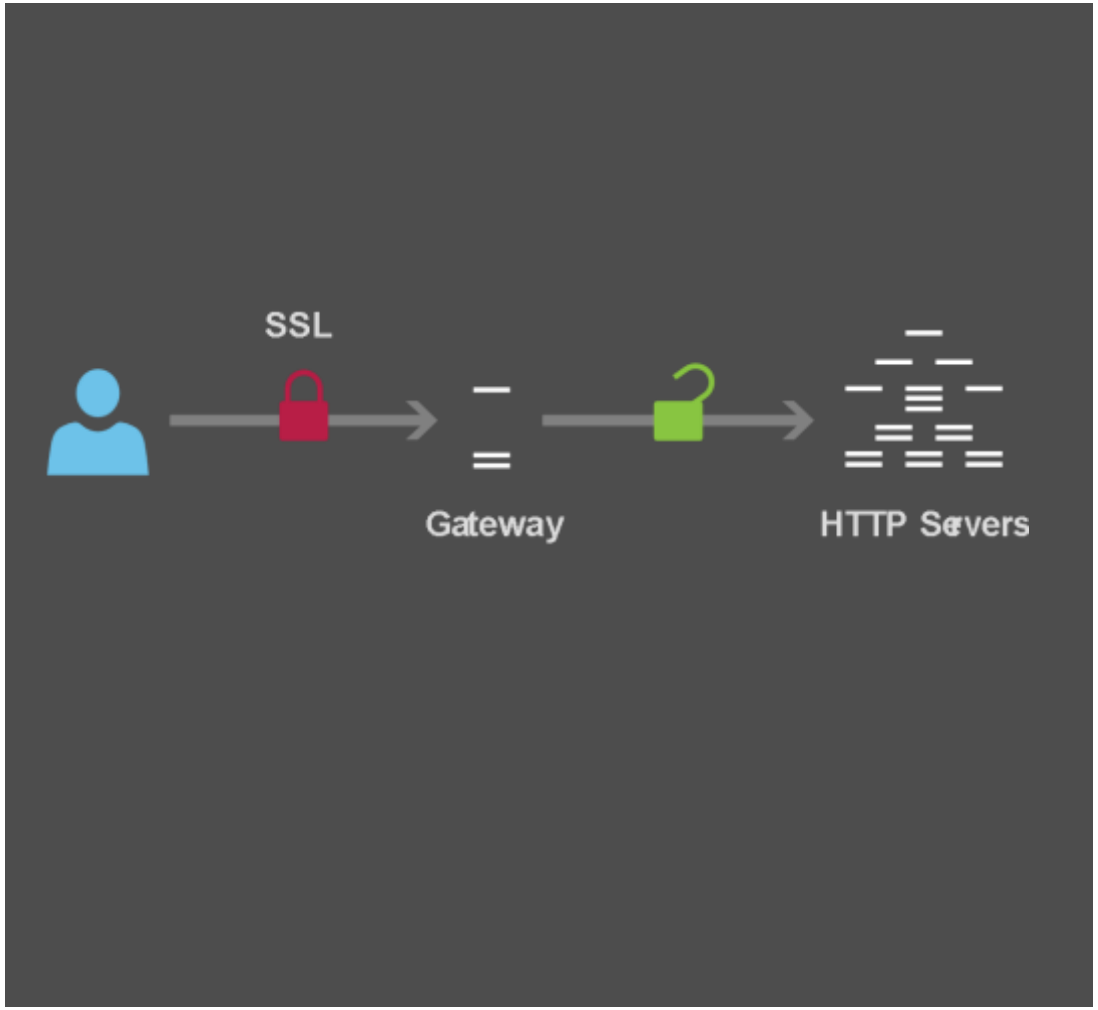
Gateway Aggregation

Aggregate multiple individual requests to a single request using the gateway aggregation pattern. This pattern is useful when a client must make multiple calls to different backend systems to perform an operation.



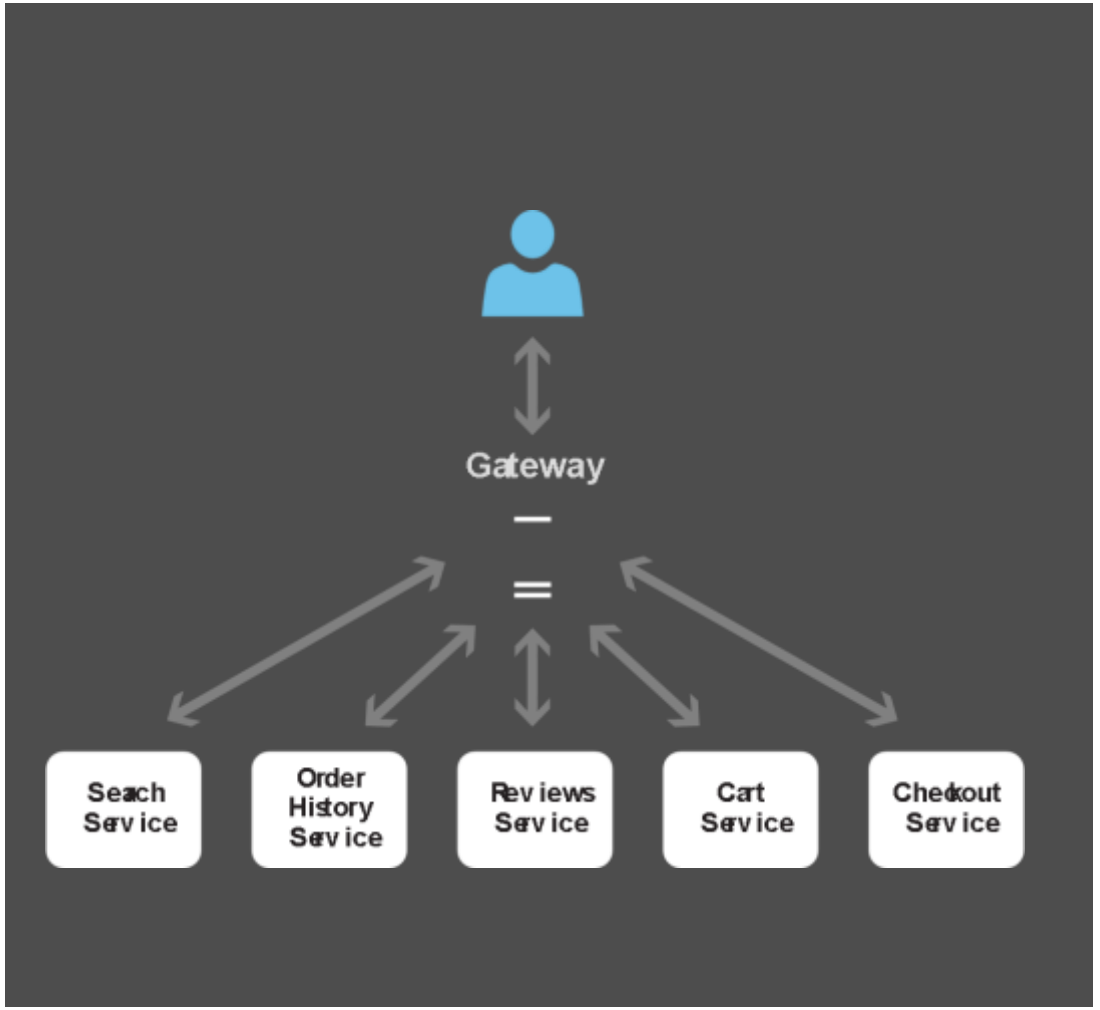
Gateway Offloading

Offload shared or specialized service functionality to a gateway proxy. This pattern can simplify application development by moving shared service functionality, such as the use of SSL certificates, to a gateway proxy, simplifying application management.



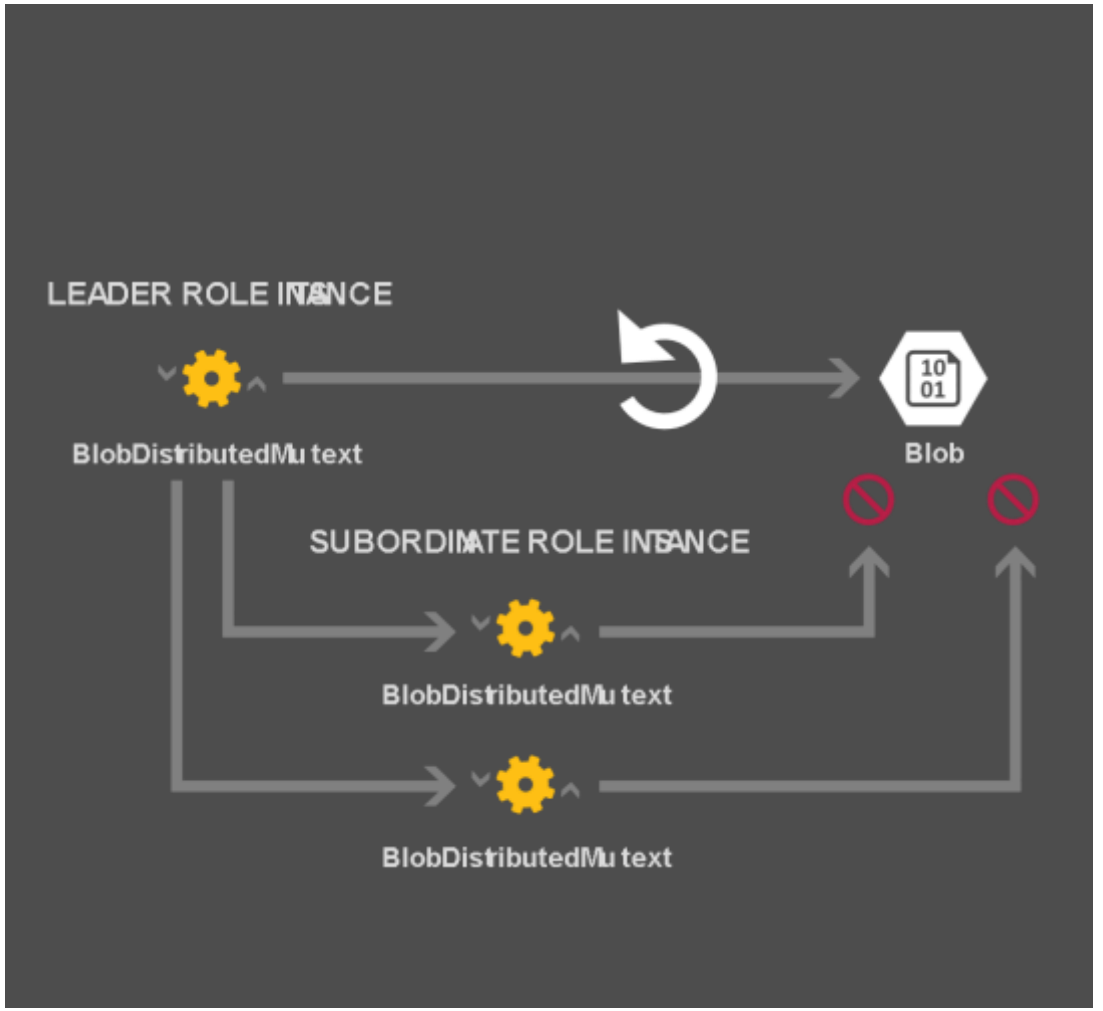
Gateway Routing

Route requests to multiple services using a single endpoint with the gateway routing pattern. This pattern is useful when you wish to expose multiple services on a single endpoint and route to the appropriate service based on the request.



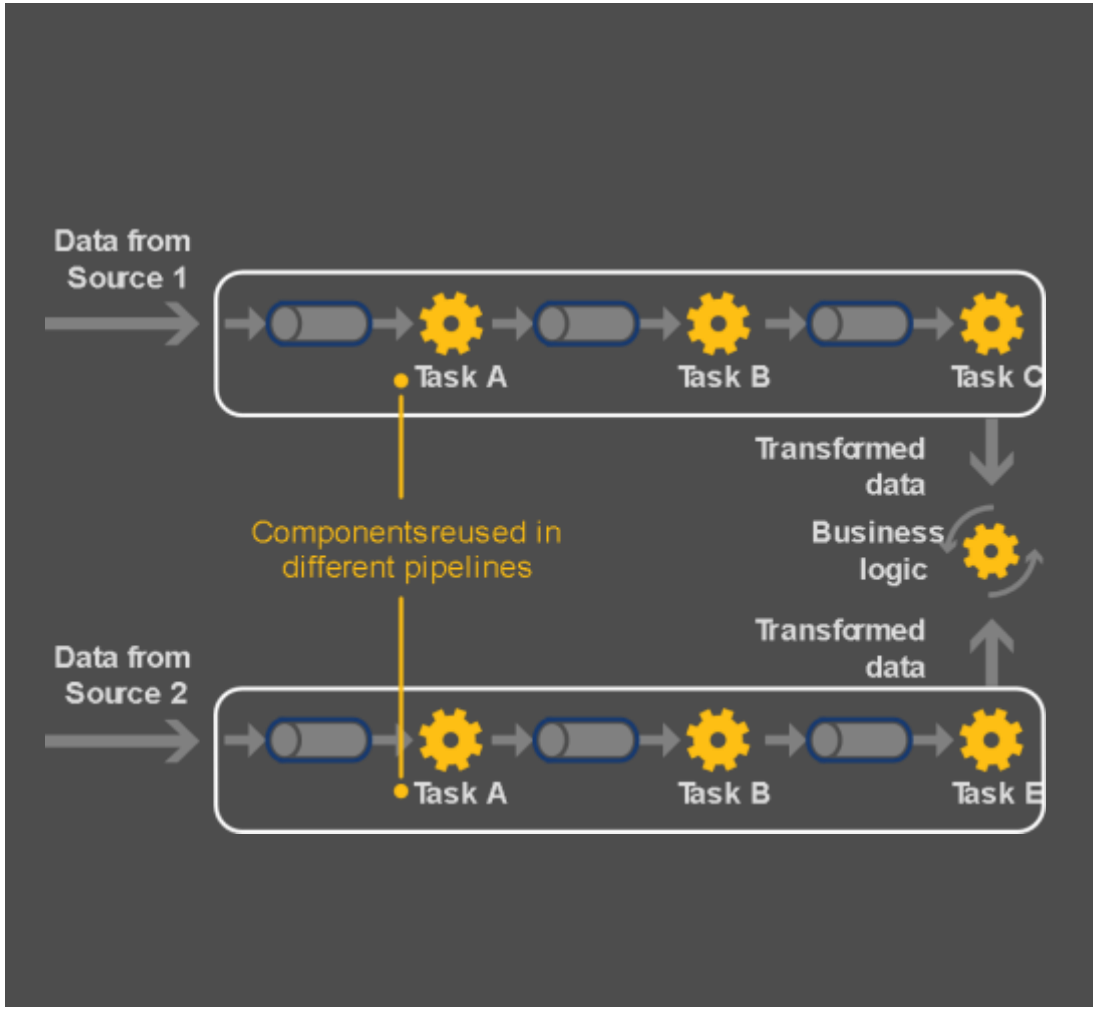
Leader Election

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. This pattern can help to ensure that task instances do not conflict with each other, cause contention for shared resources, or inadvertently interfere with the work that other task instances are performing.



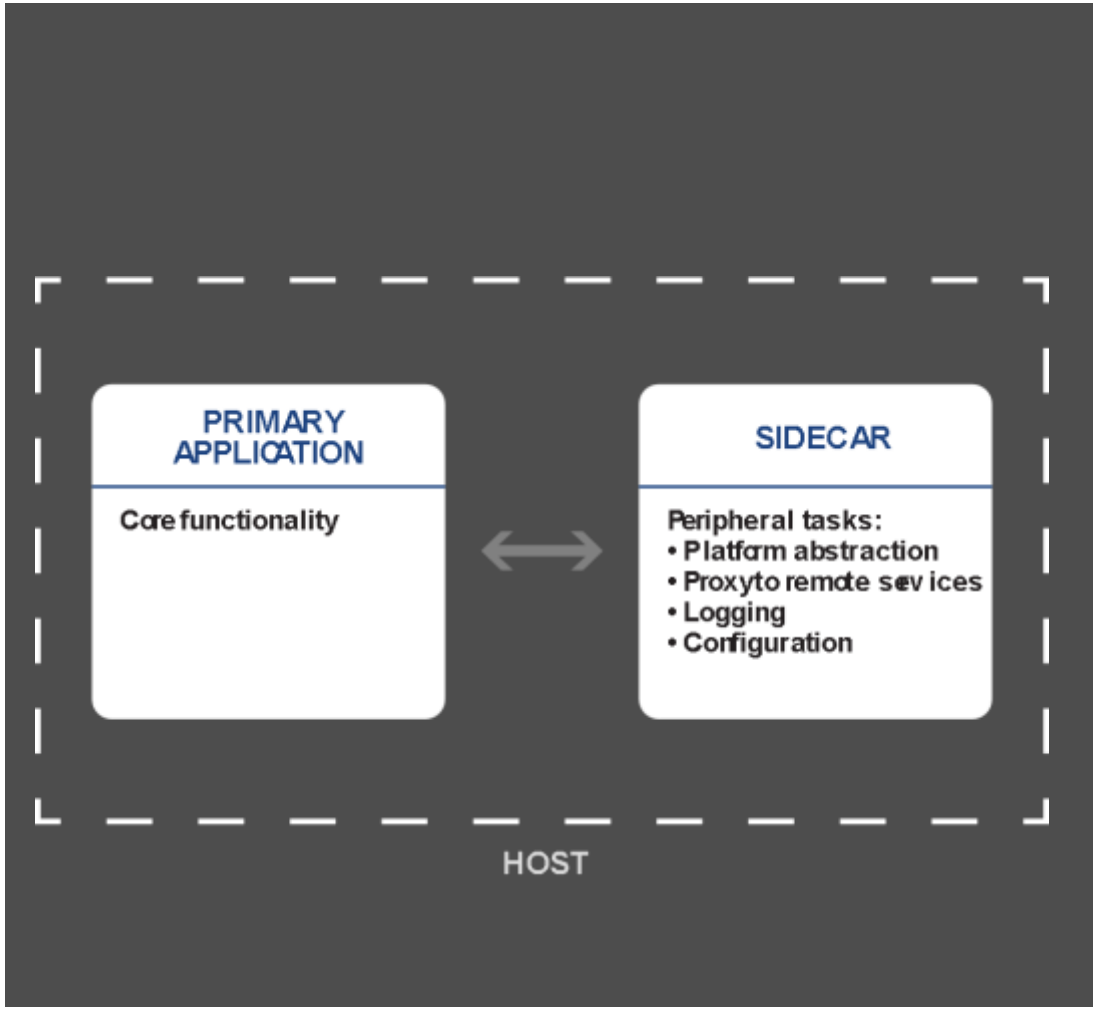
Pipes and Filters

Decompose a task that performs complex processing into a series of discrete elements that can be reused. This pattern can improve performance, scalability, and reusability by allowing task elements that perform the processing to be deployed and scaled independently.

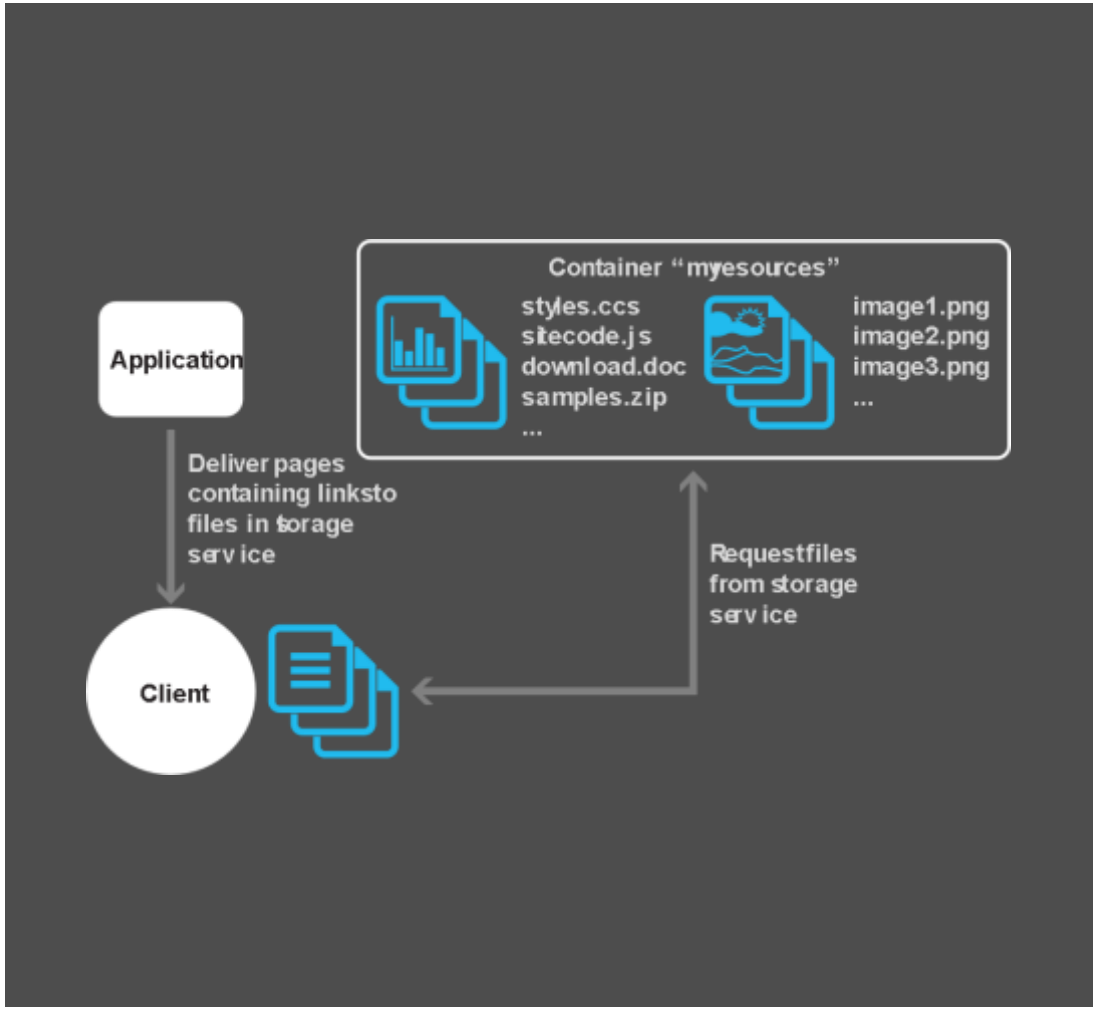


Sidecar

Deploy components of an application into a separate process or container using the sidecar pattern to provide isolation, encapsulation, and to enable applications composed of heterogeneous components and technologies.

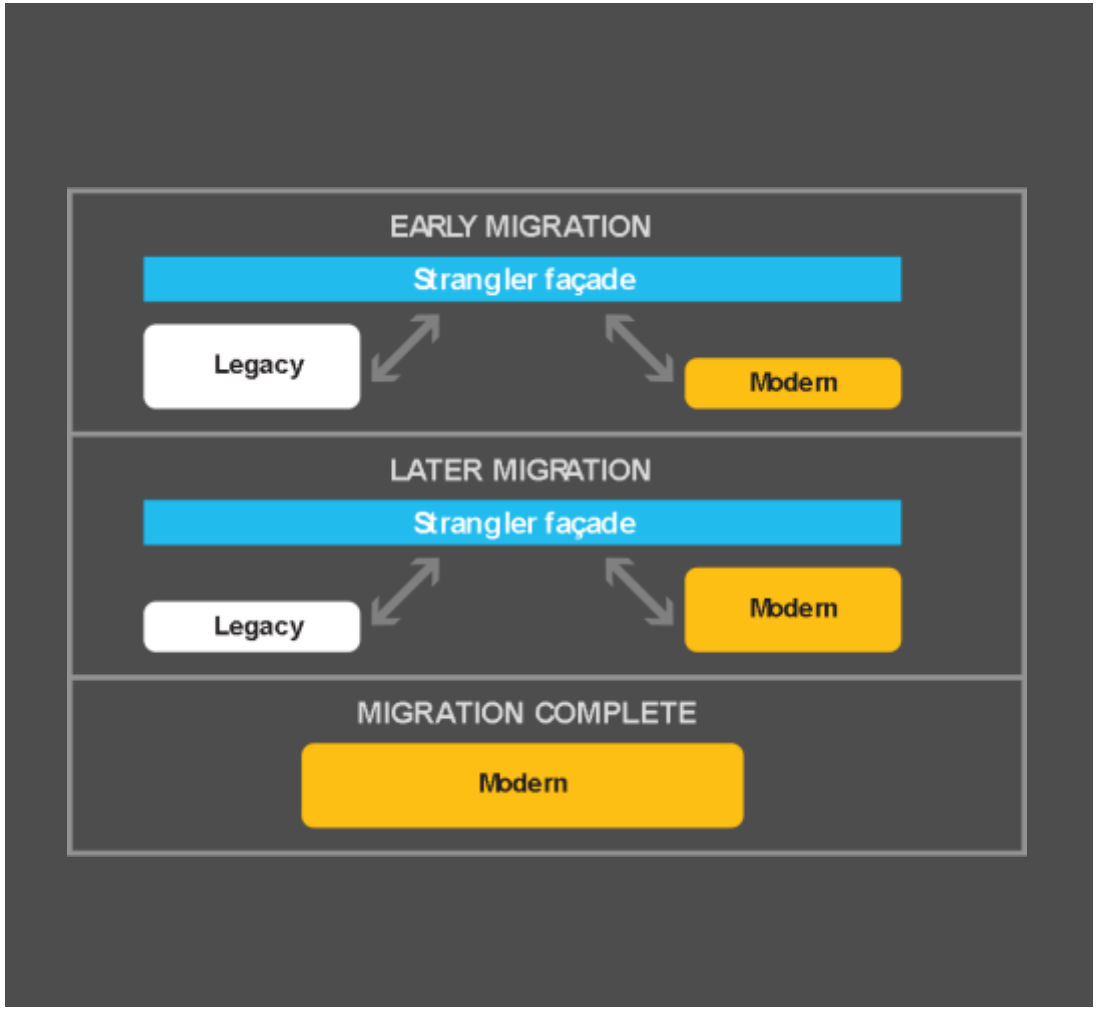


Static Content Hosting



Strangler

Incrementally move features from legacy systems to more modern systems while allowing the new system to add features in the meantime. As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it.



“ Messaging

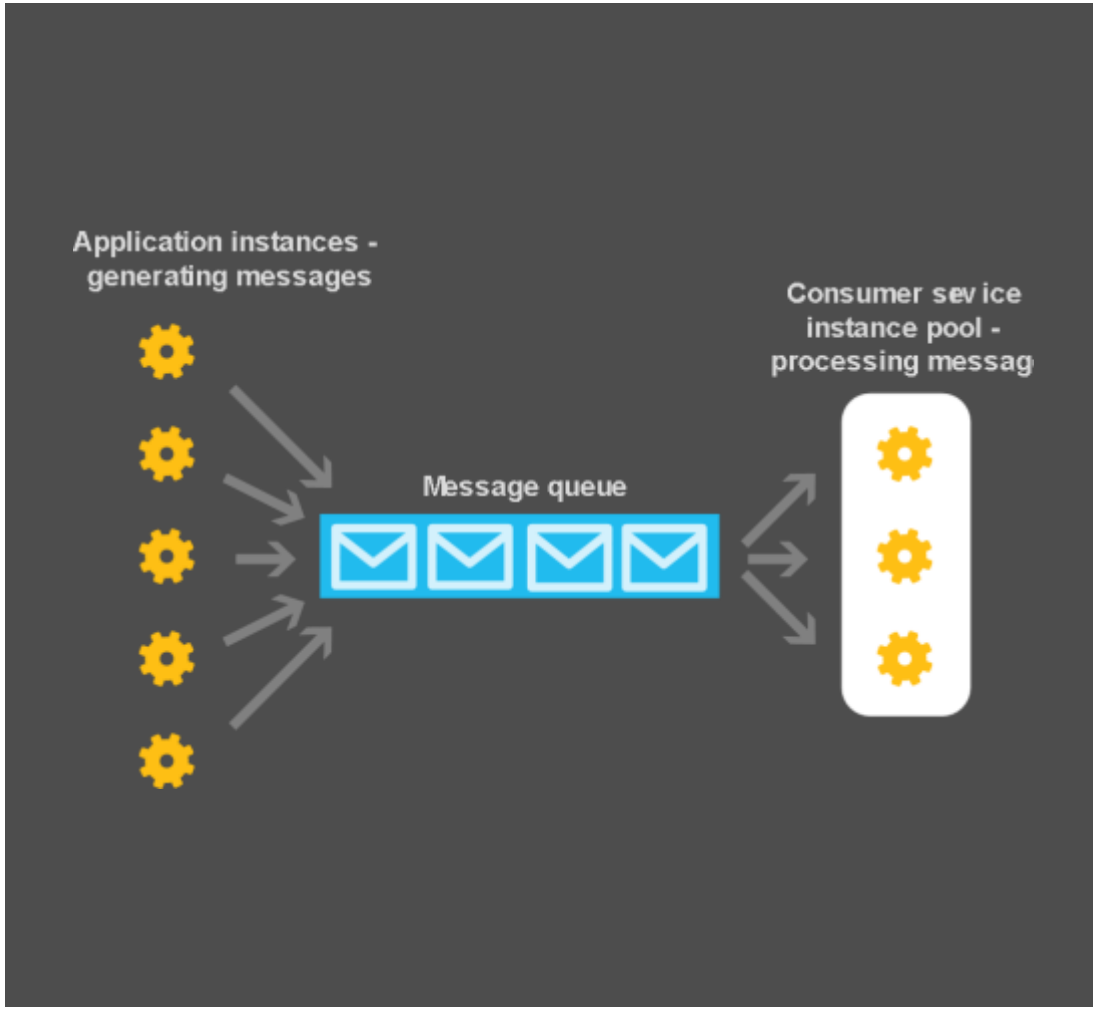
Messaging

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

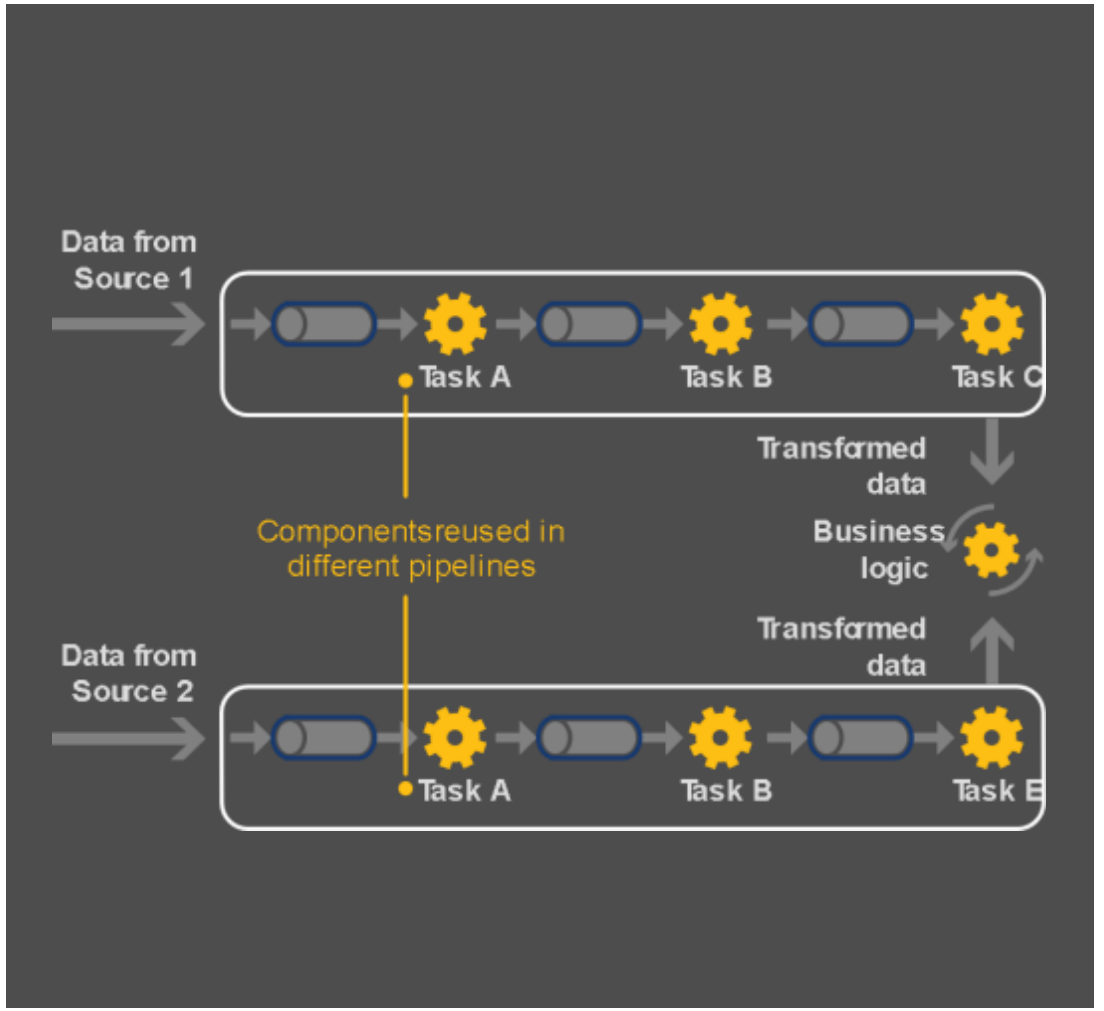
Compensating Consumers
Pipes and Filters
Priority Queue
Queue-Based Load Leveling
Scheduler Agent Supervisor

Compensating Consumers

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



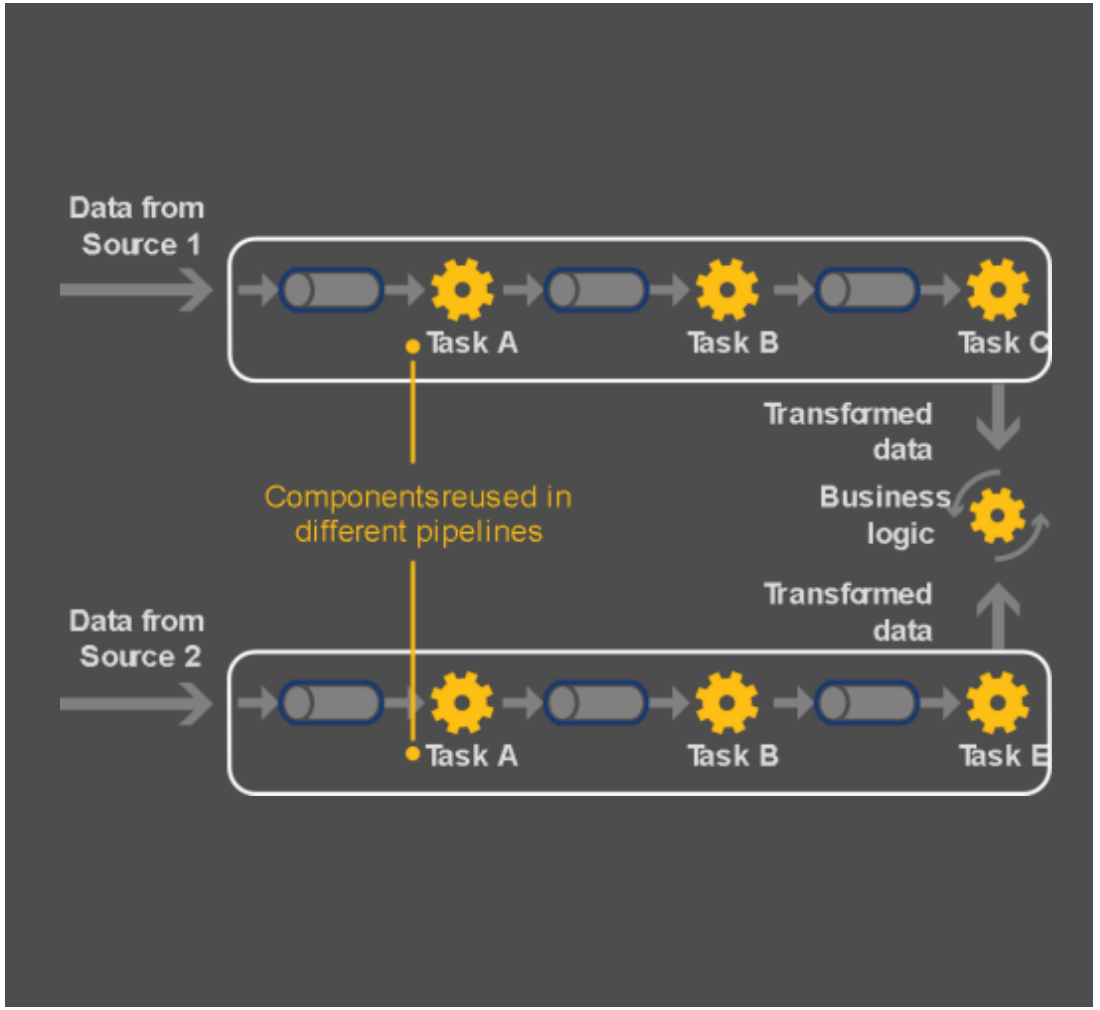
Pipes and Filters



Decompose a task that performs complex processing into a series of discrete elements that can be reused. This pattern can improve performance, scalability, and reusability by allowing task elements that perform the processing to be deployed and scaled independently.

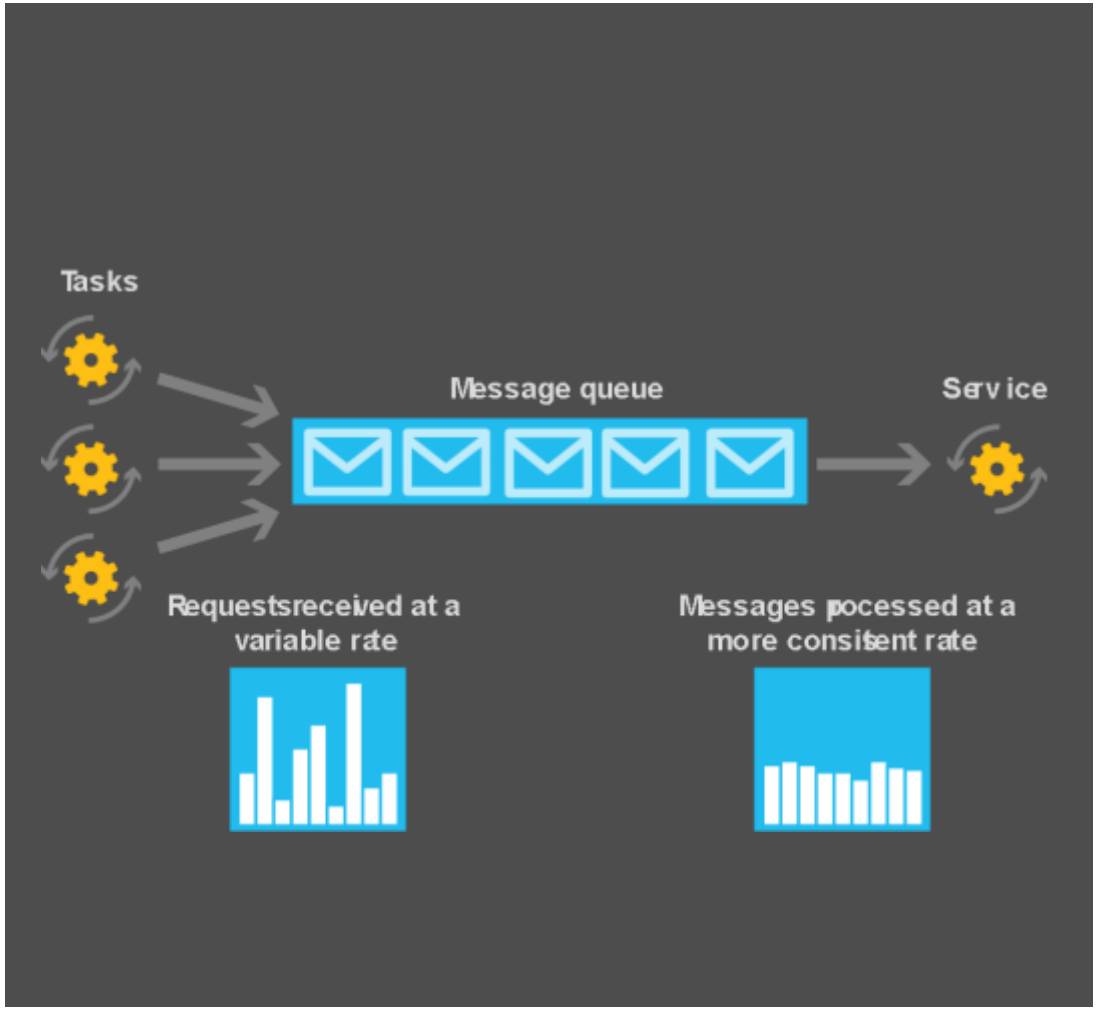
Priority Queue

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



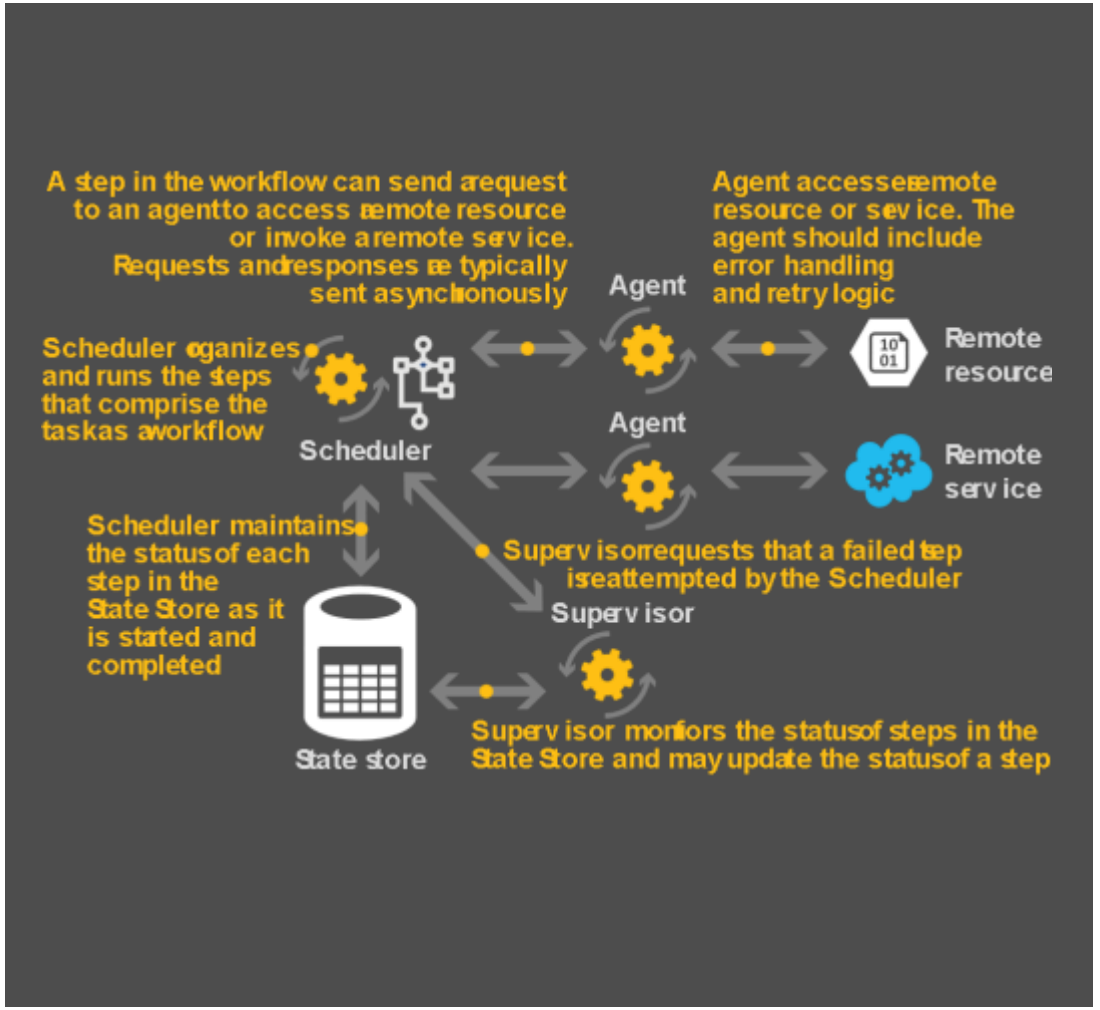
Queue-Based Load Leveling

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



Scheduler Agent Supervisor

Coordinate a set of actions across a distributed set of services and other remote resources, attempt to transparently handle faults if any of these actions fail, or undo the effects of the work performed if the system cannot recover from a fault. This pattern can add resiliency to a distributed system by enabling it to recover and retry actions that fail due to transient exceptions, long-lasting faults, and process failures.



“ Management and Monitoring

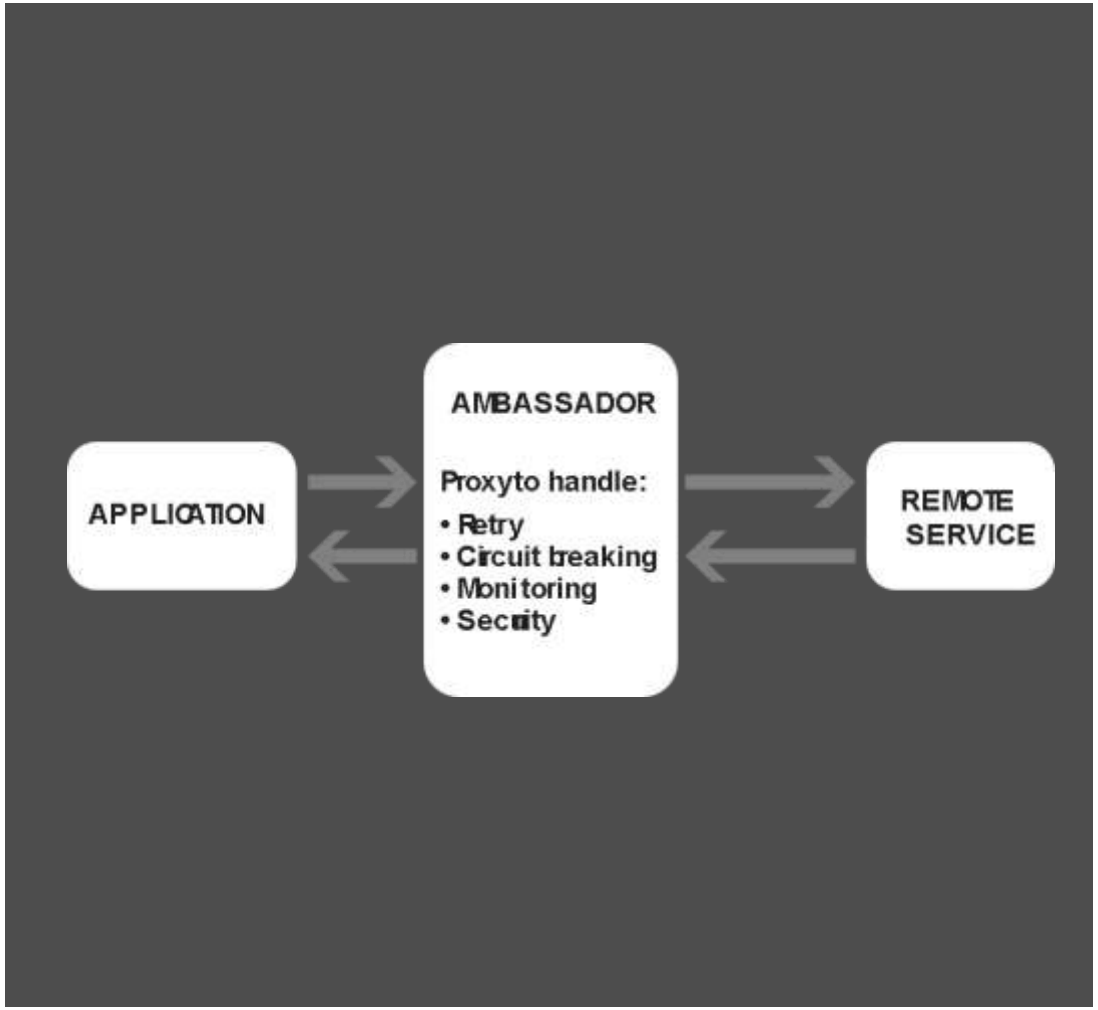
Management and Monitoring

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

Ambassador
Anti Corruption Layer
External Configuration Store
Gateway Aggregation
Gateway Offloading
Gateway Routing
Healt Endpoint Monitoring
Sidecar
Strangler

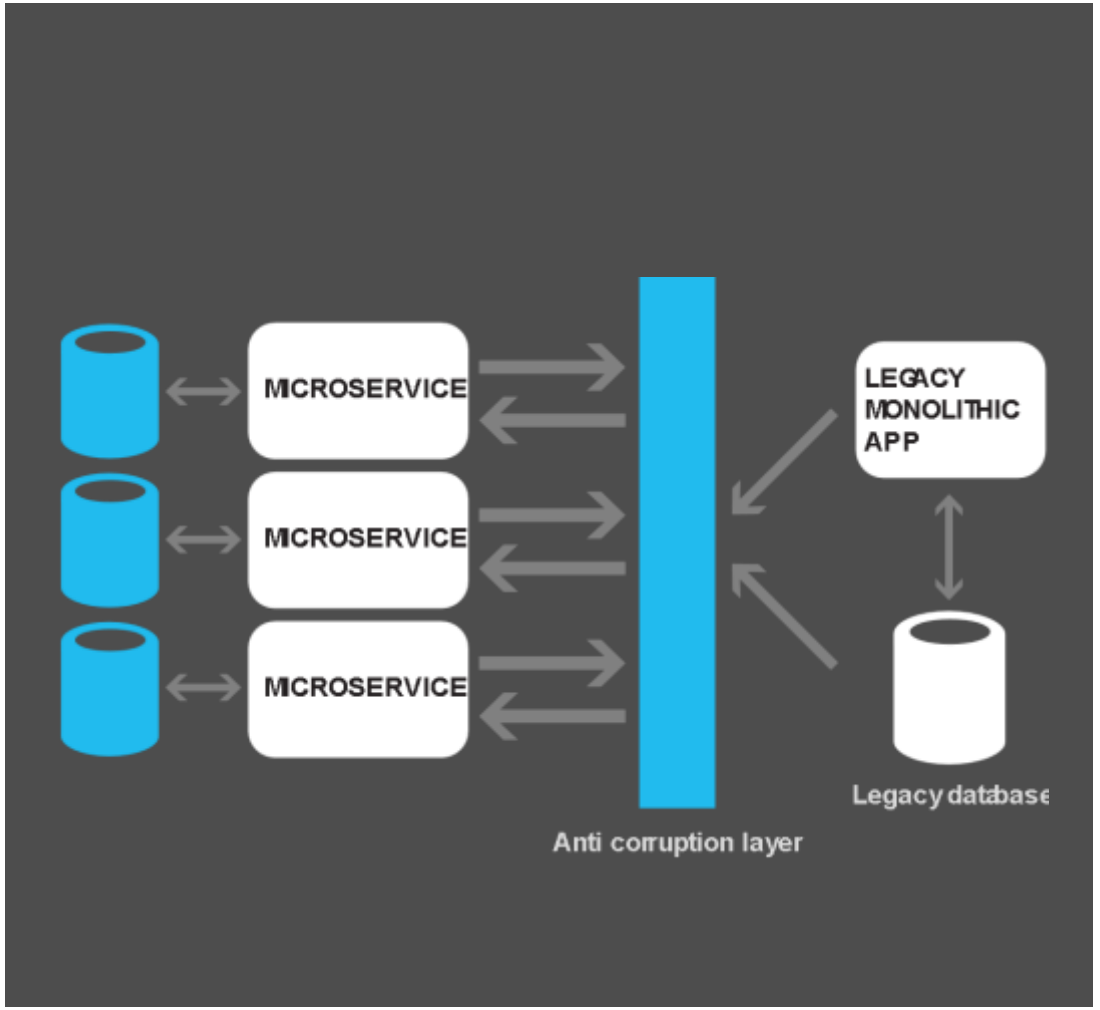
Ambassador

Create helper services for making networking 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.



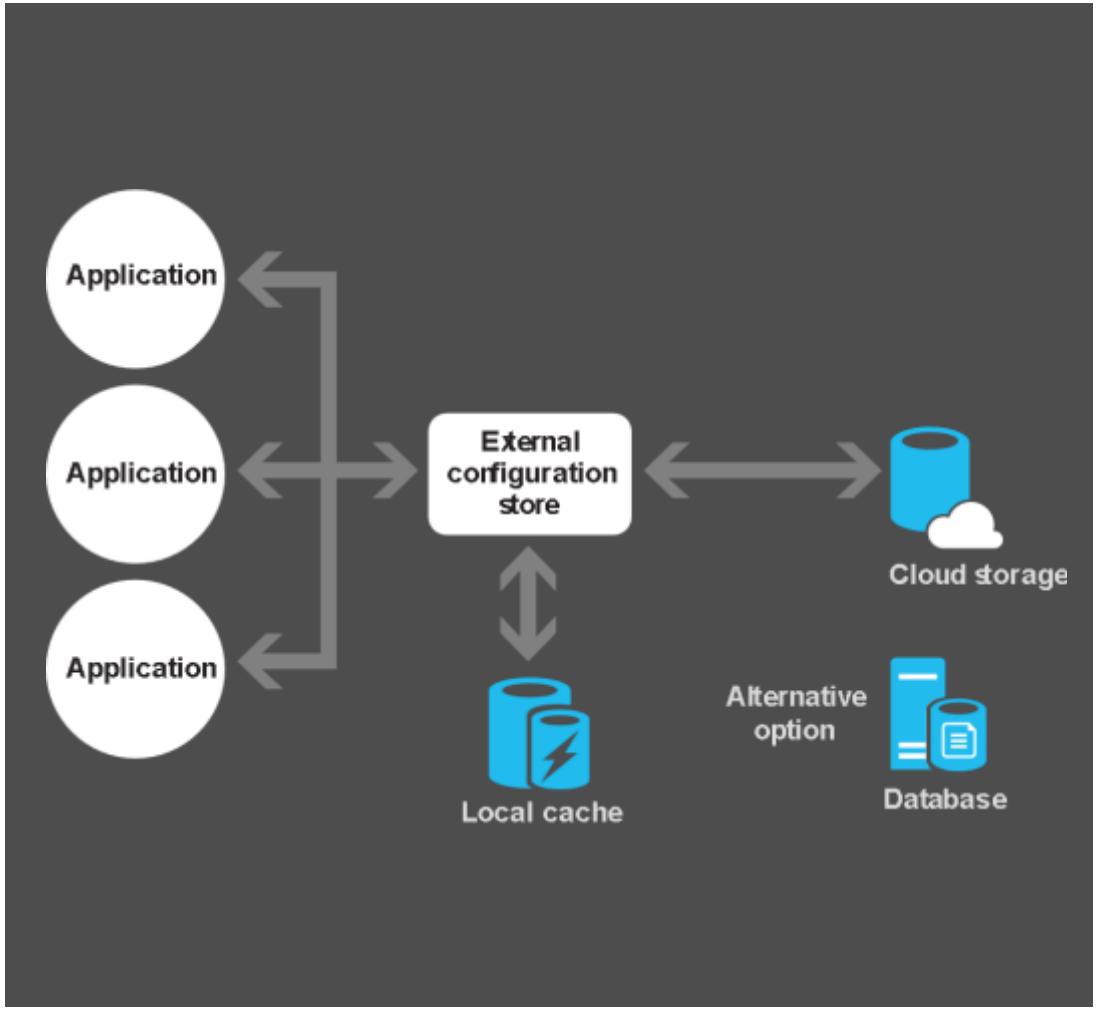
Anti Corruption Layer

Ensure the design and structure of new apps and services are not limited by legacy resources. Implement a façade or adapter layer in between legacy systems and modern systems where features and capabilities are being migrated to. This façade or layer translates requests between modern systems and legacy applications using methods appropriate to each.



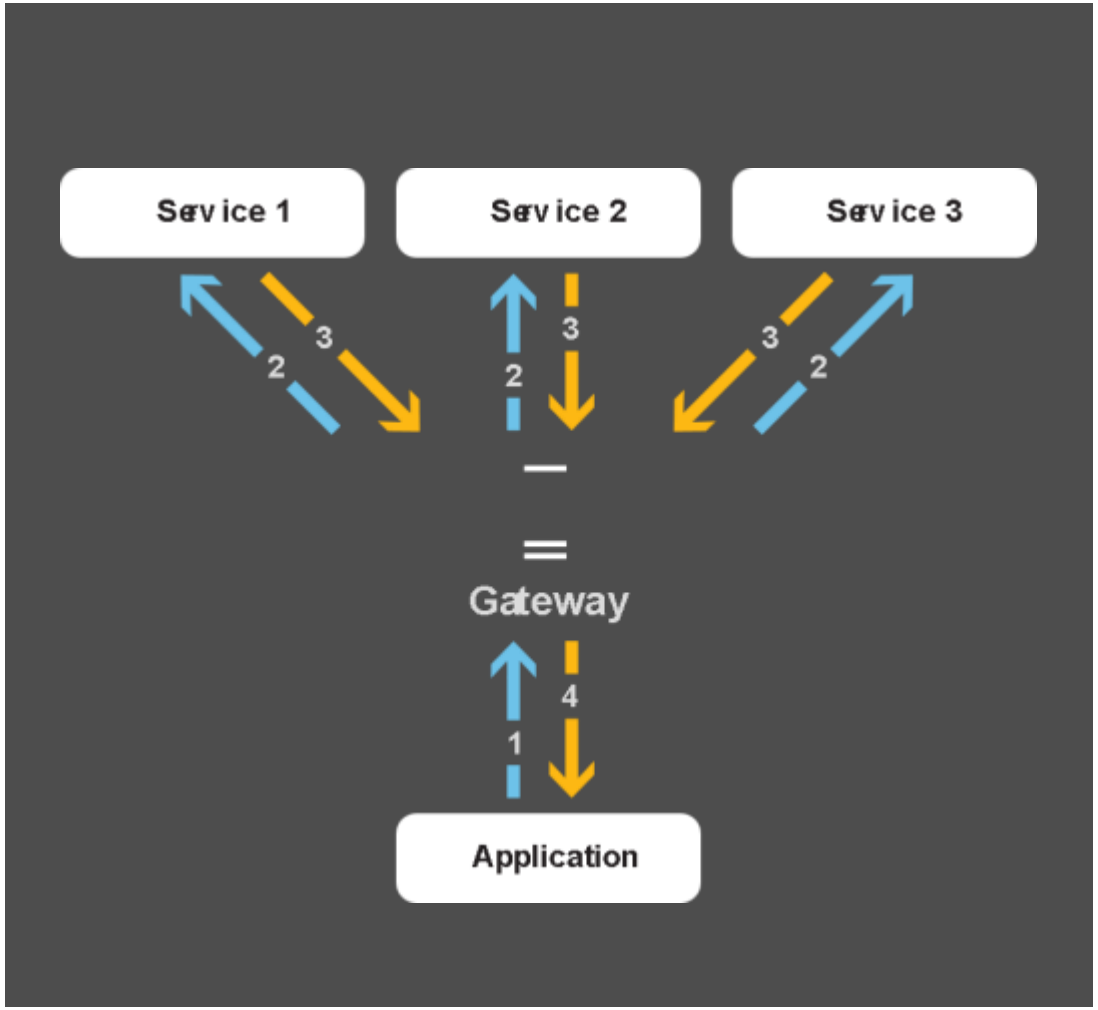
External Configuration Store

Move configuration information out of the application deployment package to a centralized location. This pattern can provide opportunities for easier management and control of configuration data, and for sharing configuration data across applications and application instances.



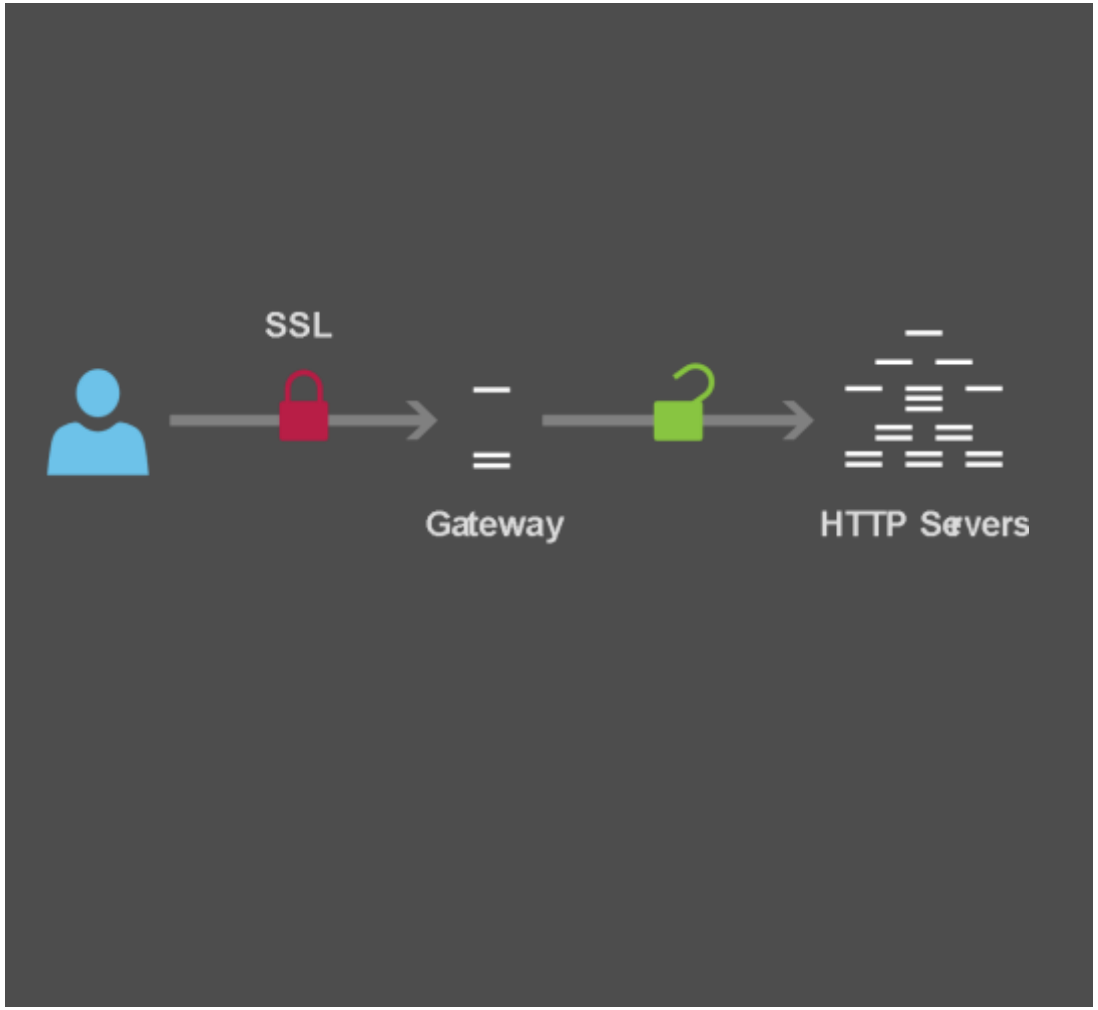
Gateway Aggregation

Aggregate multiple individual requests to a single request using the gateway aggregation pattern. This pattern is useful when a client must make multiple calls to different backend systems to perform an operation.



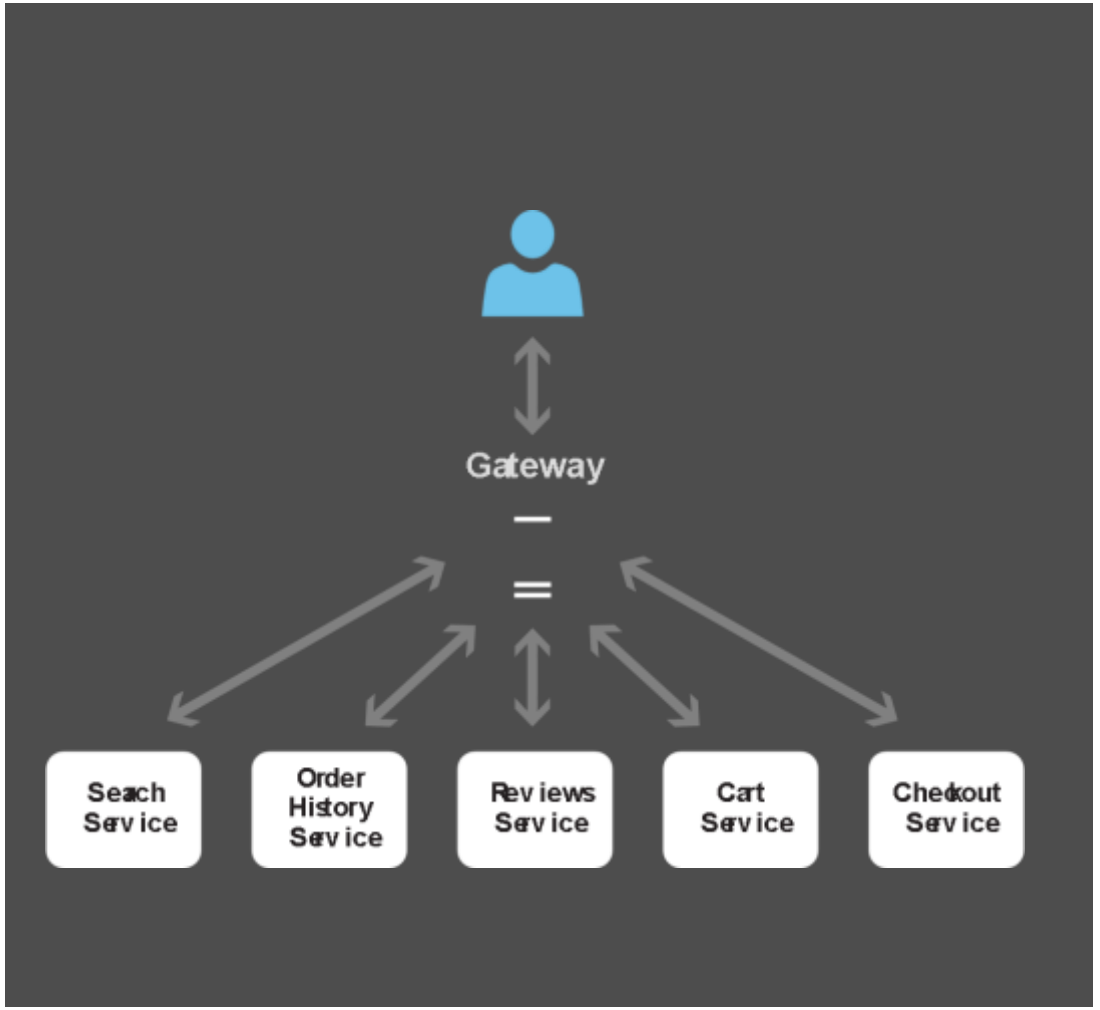
Gateway Offloading

Offload shared or specialized service functionality to a gateway proxy. This pattern can simplify application development by moving shared service functionality, such as the use of SSL certificates, to a gateway proxy, simplifying application management.



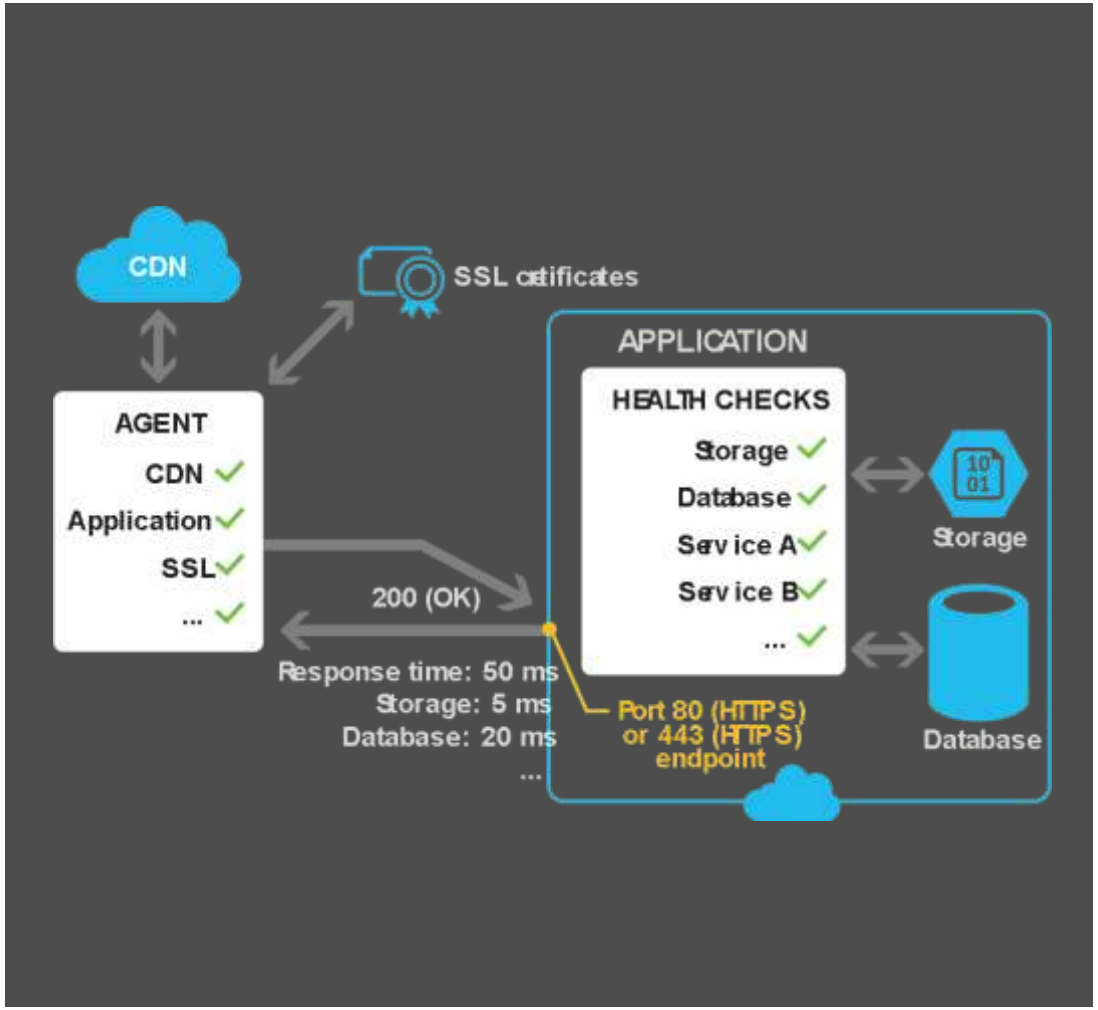
Gateway Routing

Route requests to multiple services using a single endpoint with the gateway routing pattern. This pattern is useful when you wish to expose multiple services on a single endpoint and route to the appropriate service based on the request.



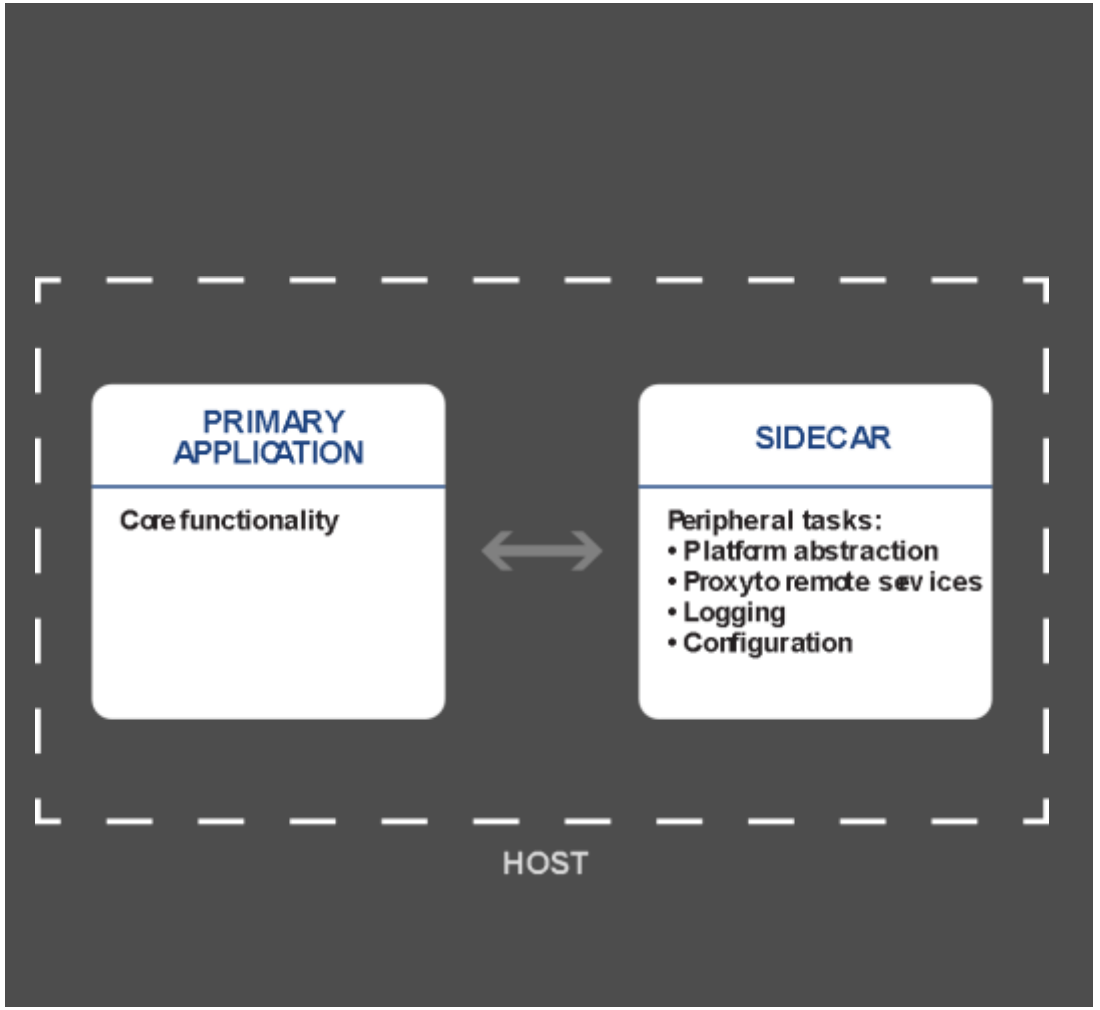
Health Endpoint Monitoring

Implement functional checks within an application that external tools can access through exposed endpoints at regular intervals. This pattern can help to verify that applications and services are performing correctly.



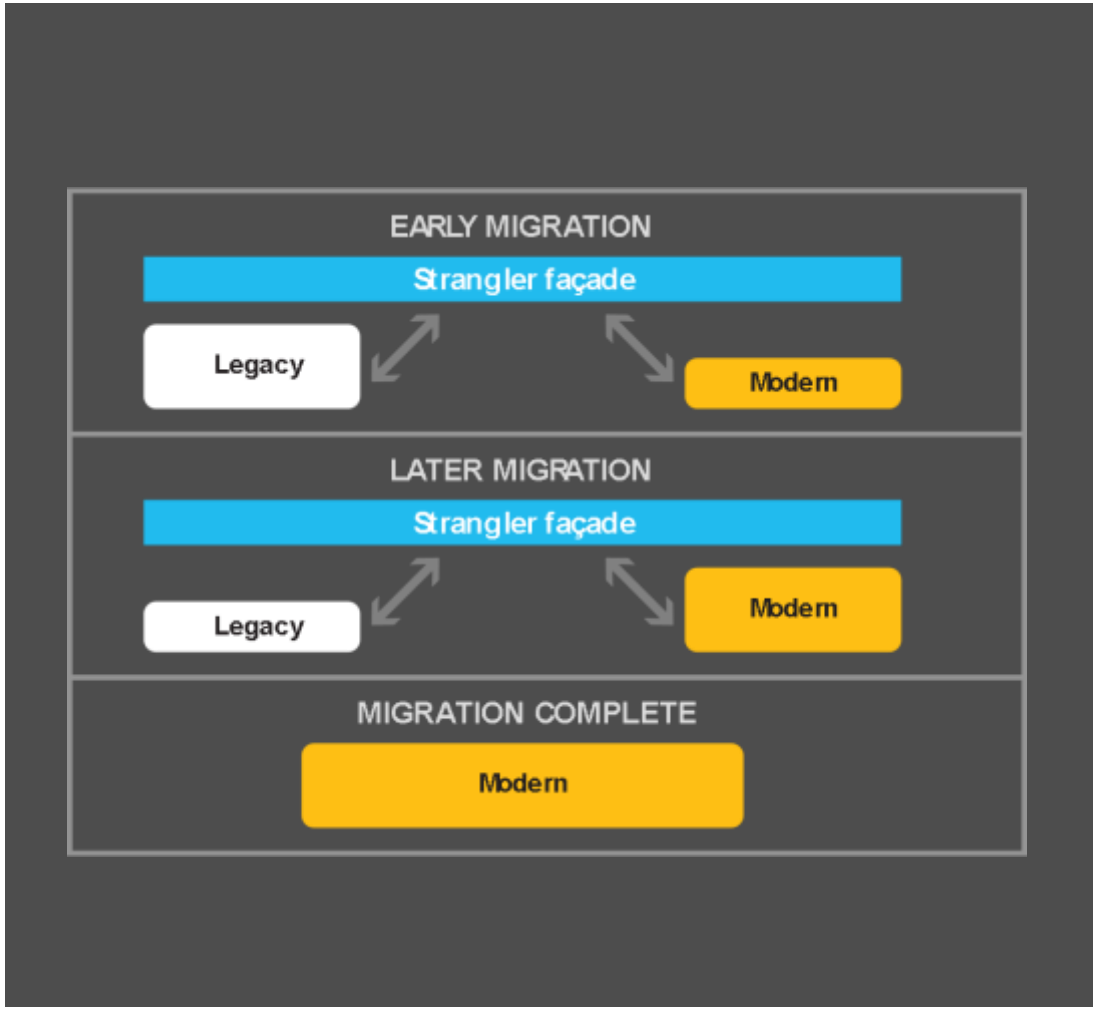
Sidecar

Deploy components of an application into a separate process or container using the sidecar pattern to provide isolation, encapsulation, and to enable applications composed of heterogeneous components and technologies.



Strangler

Incrementally move features from legacy systems to more modern systems while allowing the new system to add features in the meantime. As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it.



“ Performance and Scalability

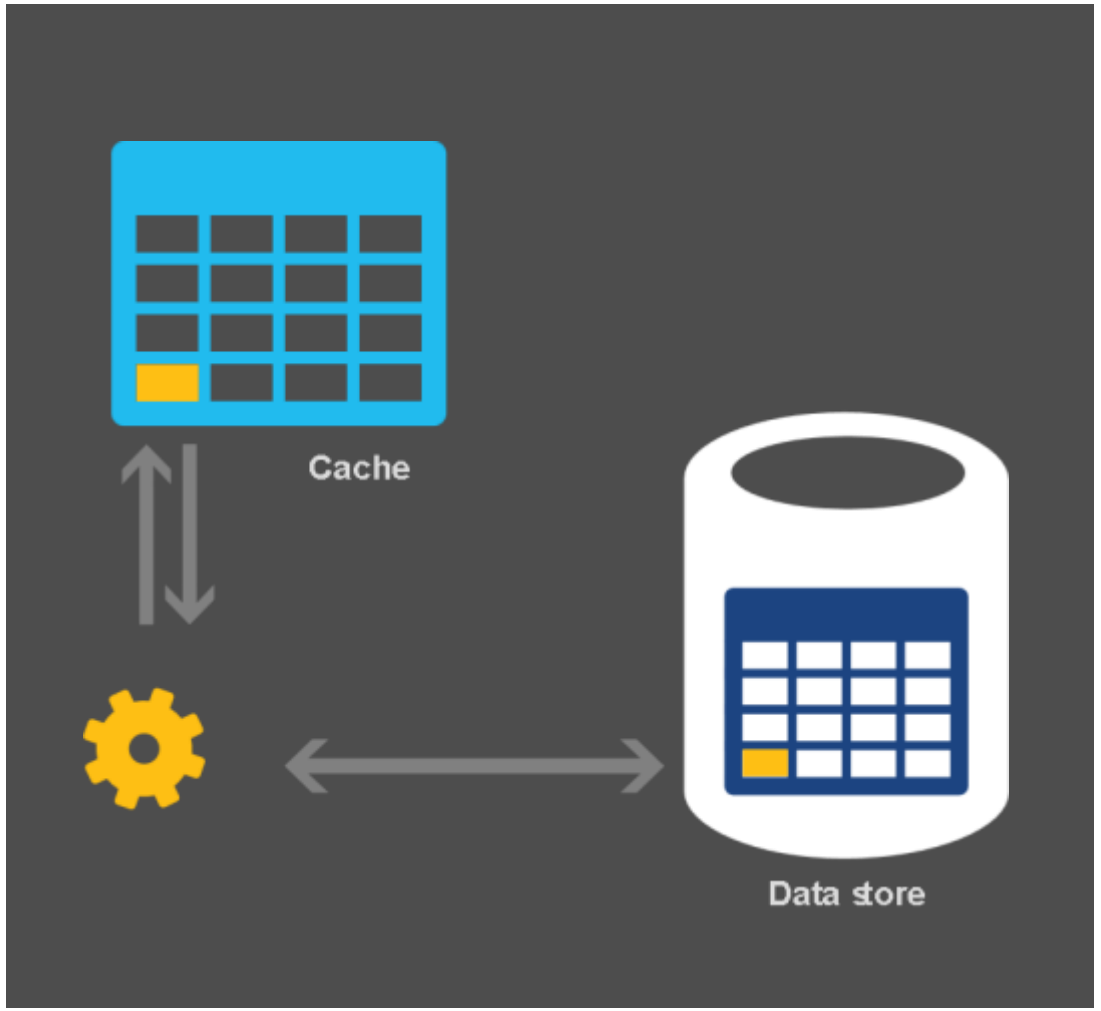
Performance and Scalability

Performance is an indication of the responsiveness of a system, while scalability is the ability to gracefully handle increases in load, perhaps through an increase in available resources. Cloud applications, especially in multi-tenant scenarios, typically encounter variable workloads and unpredictable activity peaks and should be able to scale out within limits to meet demand, and scale in when demand decreases. Scalability concerns not just compute instances, but other items such as data storage, messaging infrastructure, and more.

- Cache-aside
- Command and Query Responsibility
- Compensating Consumers
- Index Table
- Materialized View
- Priority Queue
- Sharding
- Static Content Hosting
- Throttling

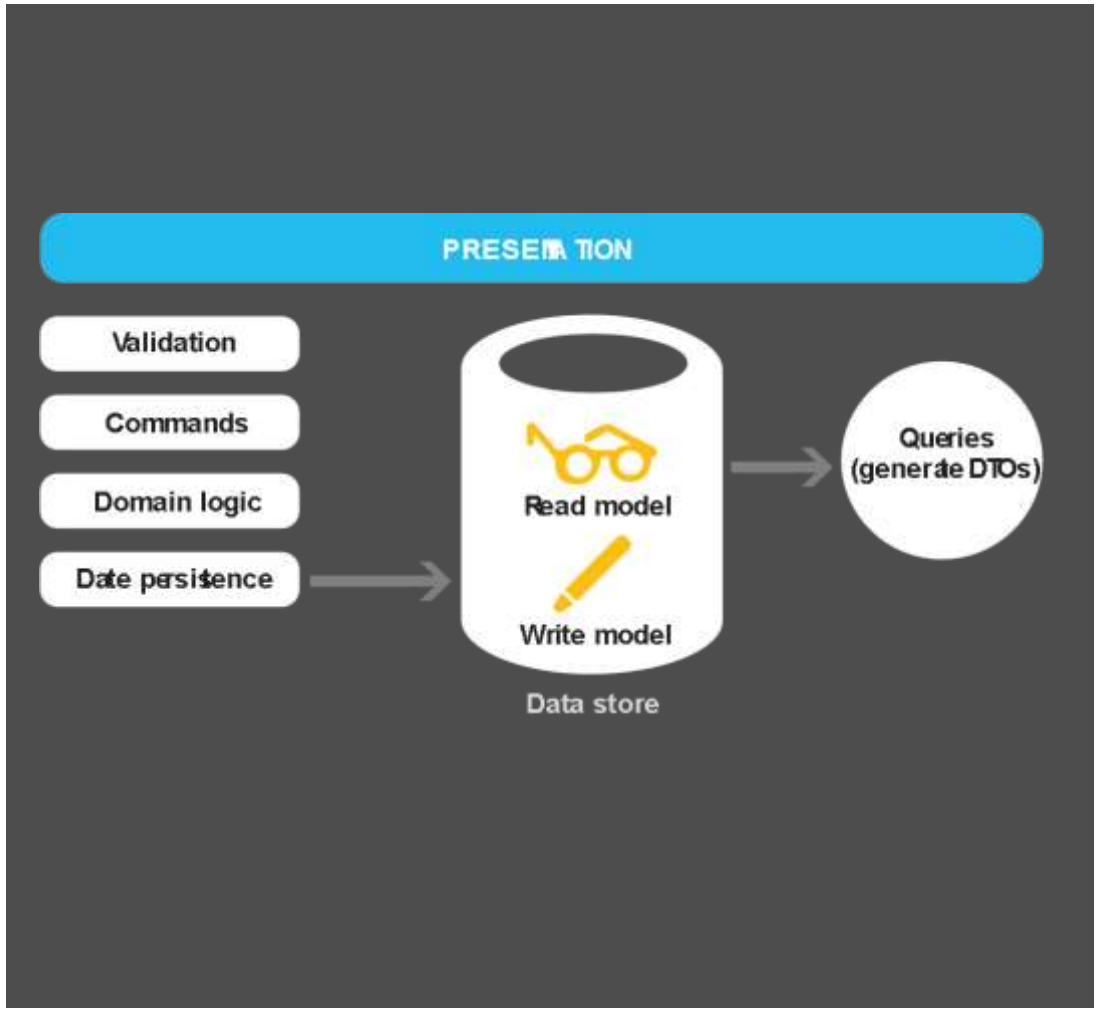
Cache-aside

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



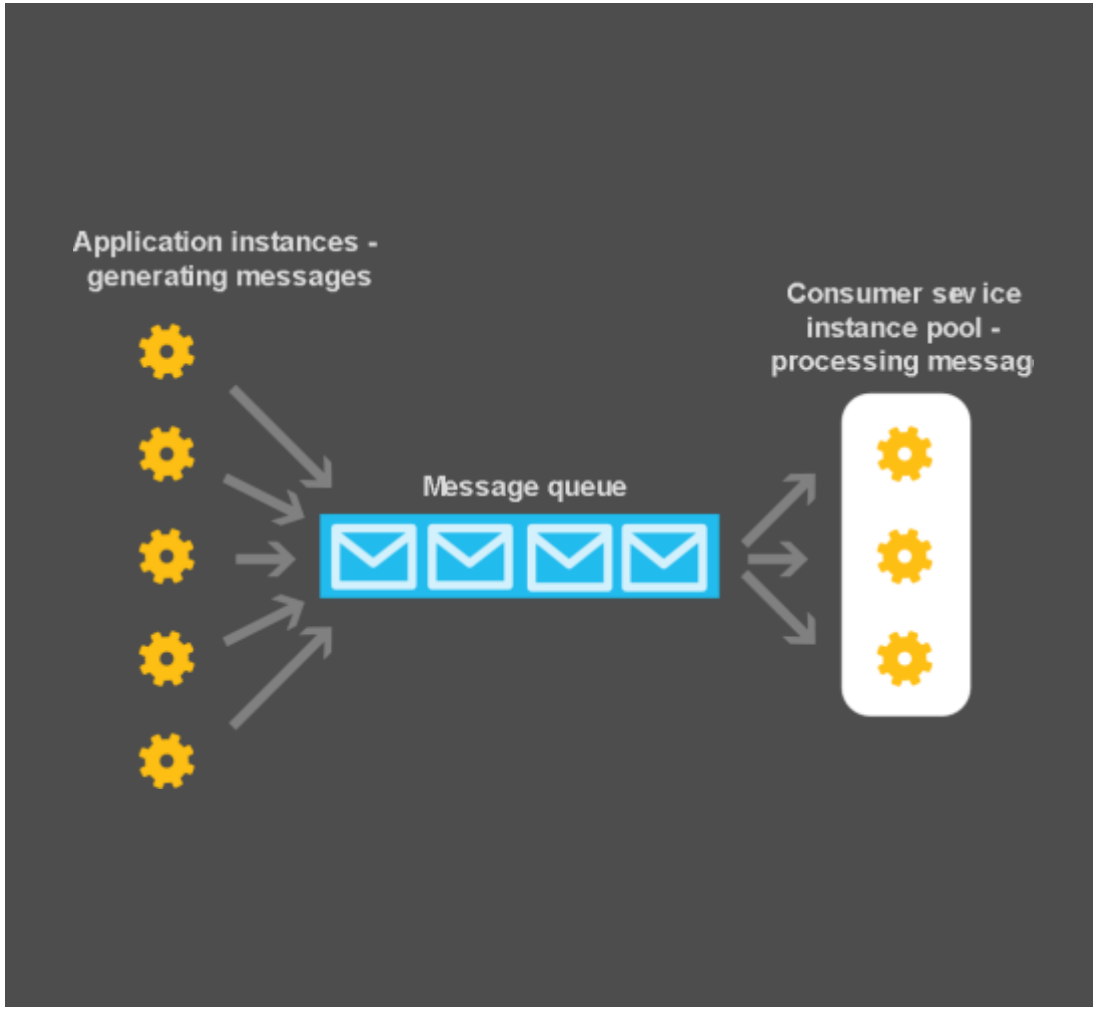
Command and Query Responsibility Segregation (CQRS)

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



Compensating Consumers

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

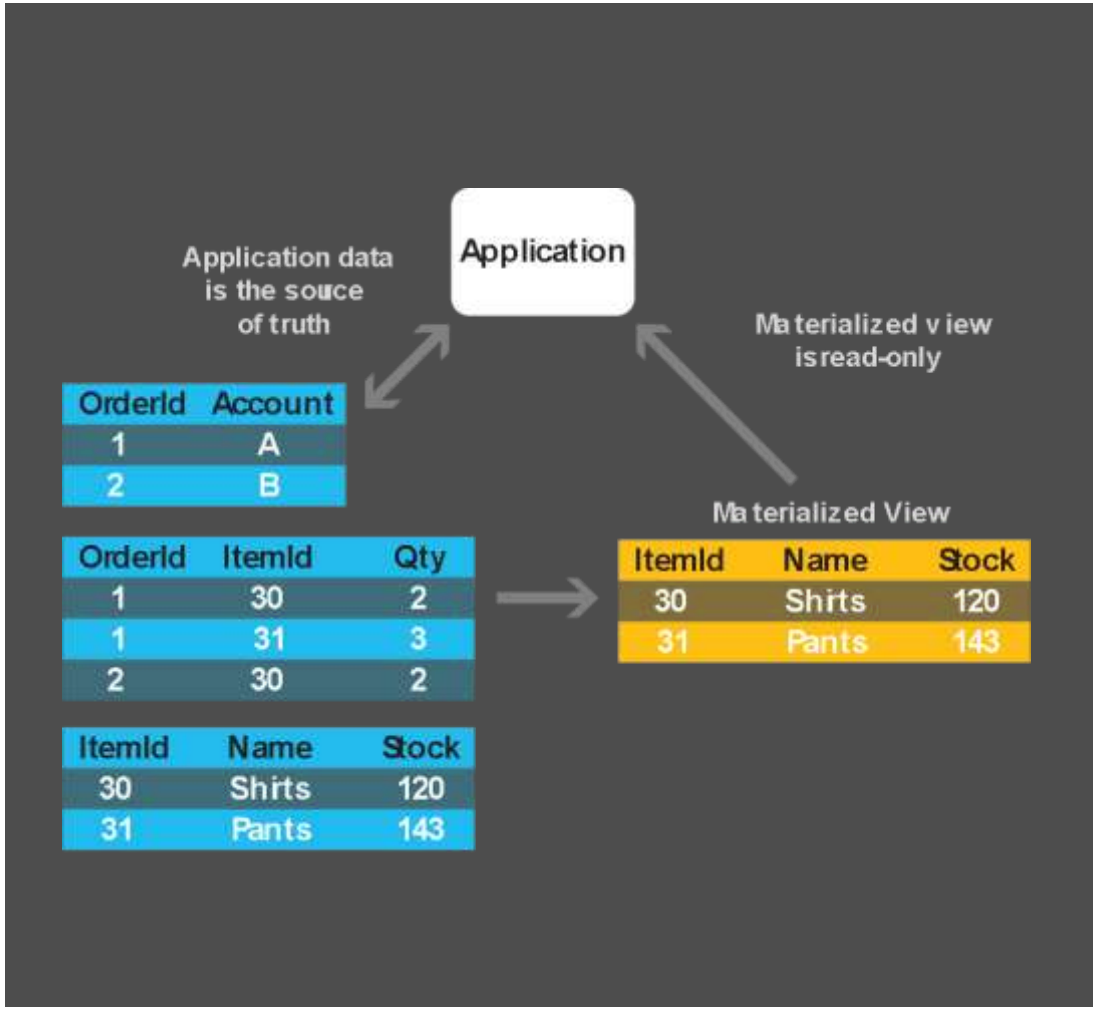


Index Table

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

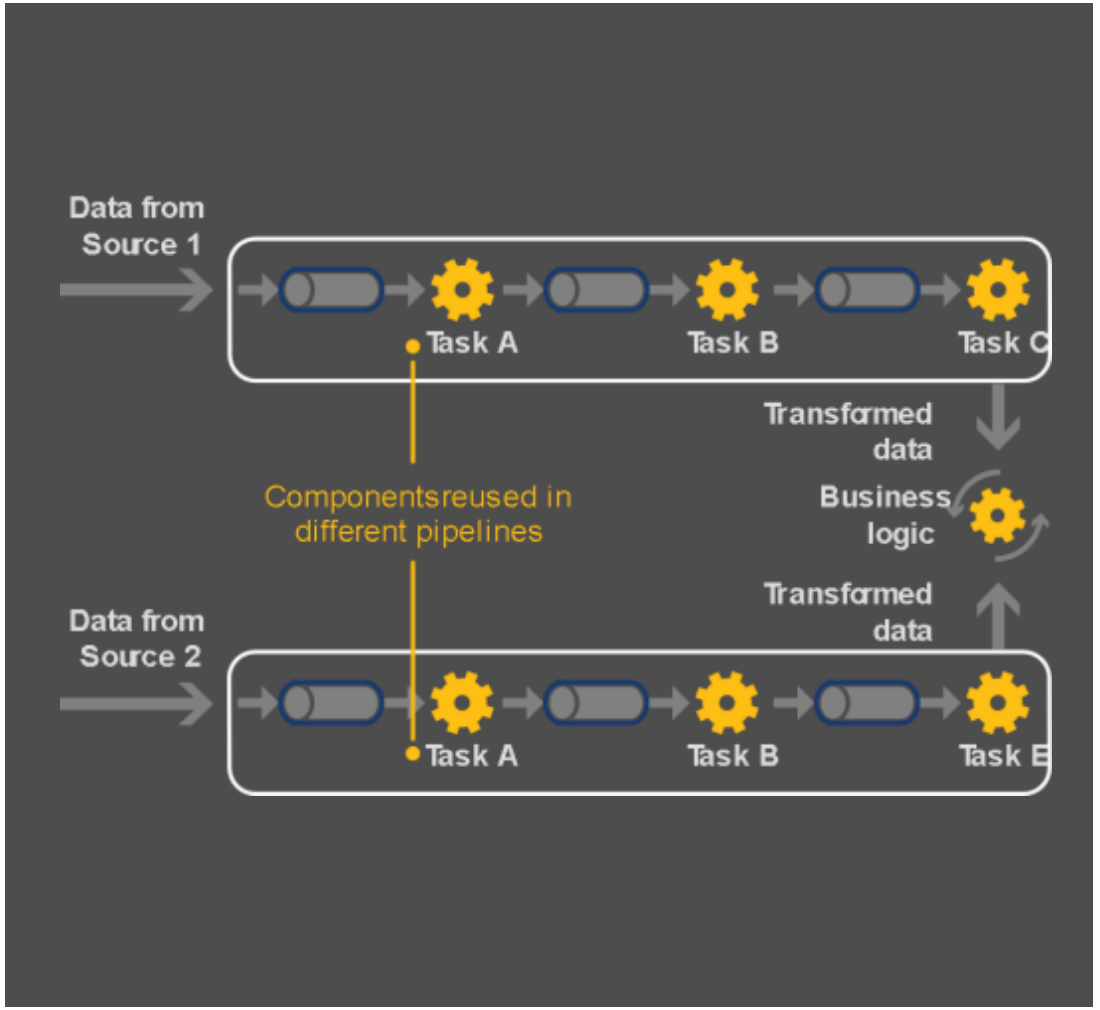
Materialized View

Generate pre-populated views over the data in one or more data stores when the data is formatted in a way that does not favor the required query operations. This pattern can help to support efficient querying and data extraction, and improve performance.



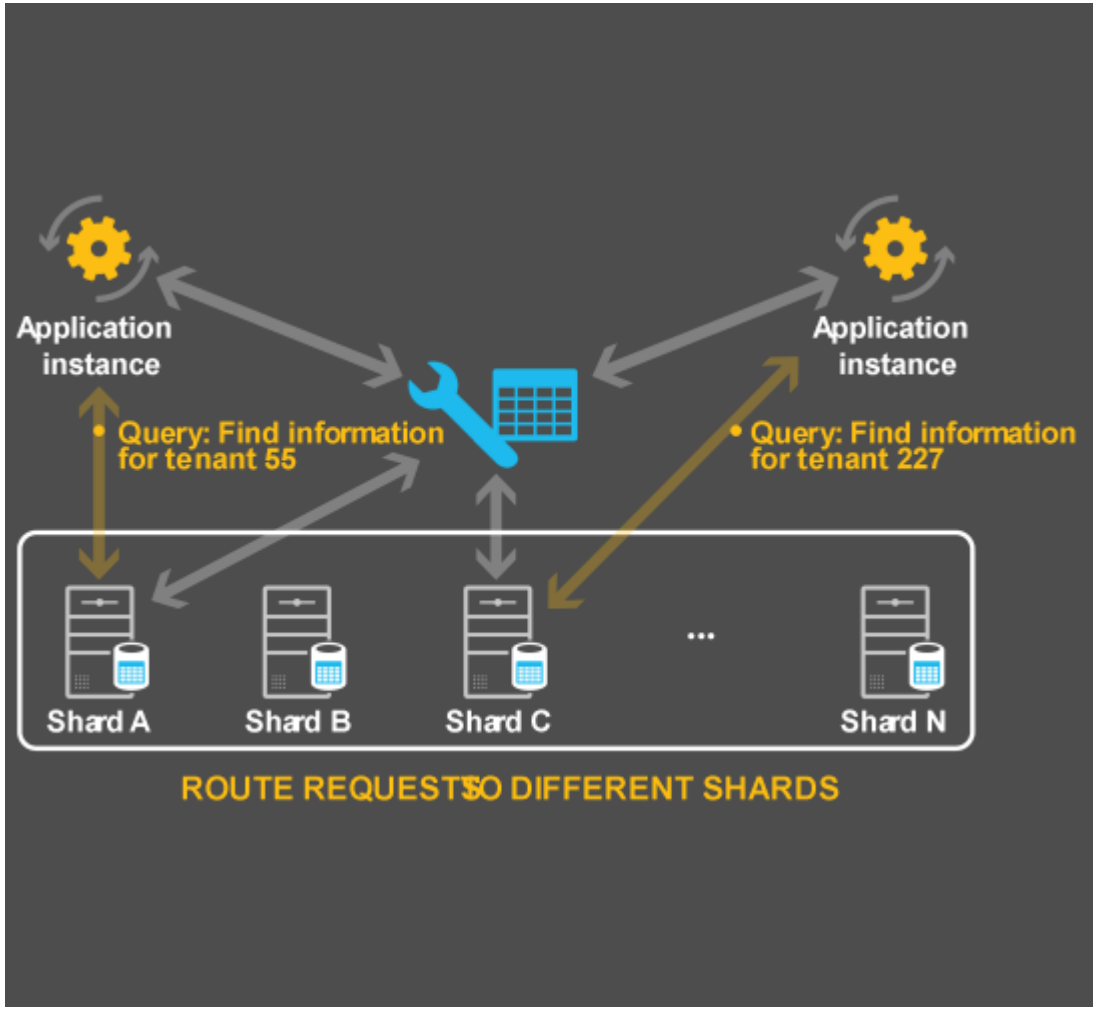
Priority Queue

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

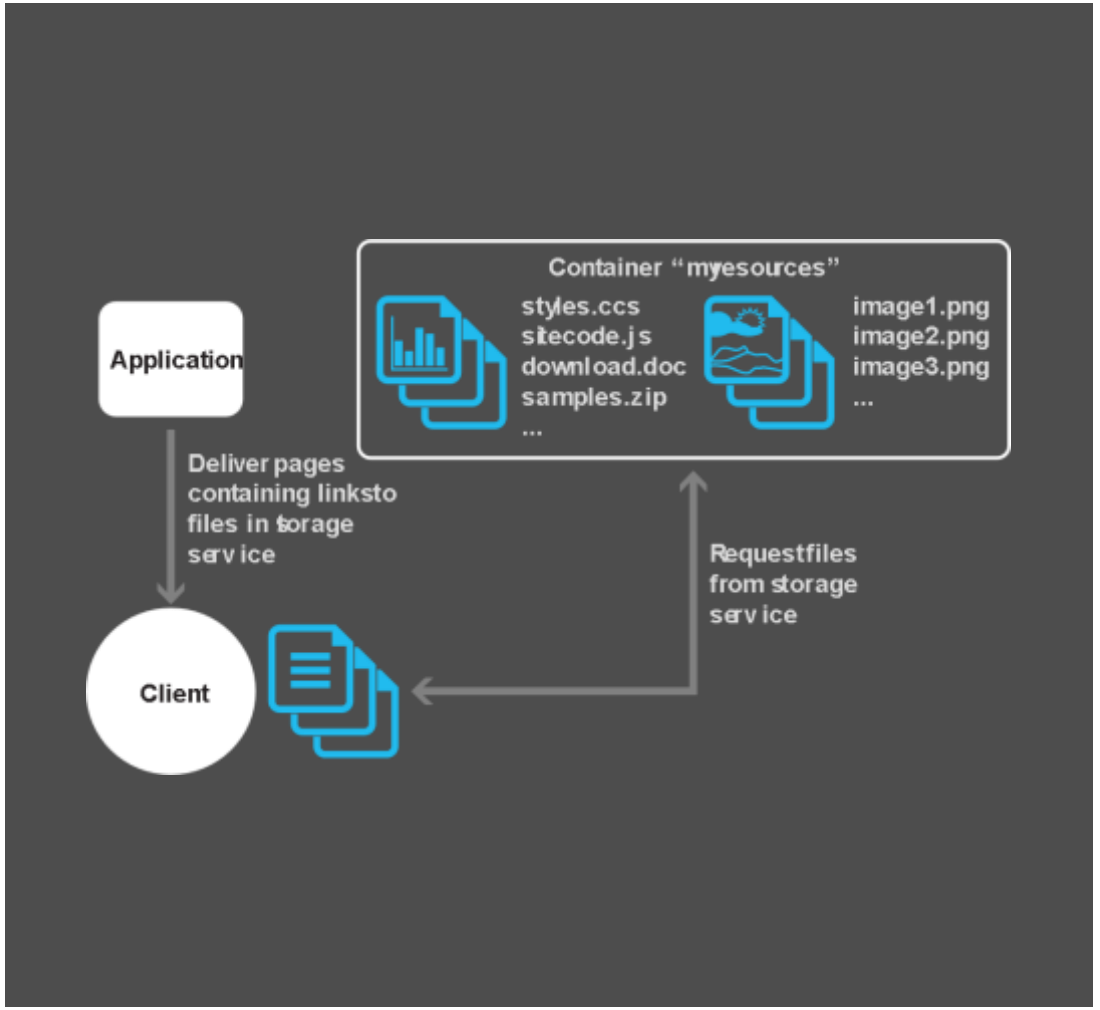


Sharding

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

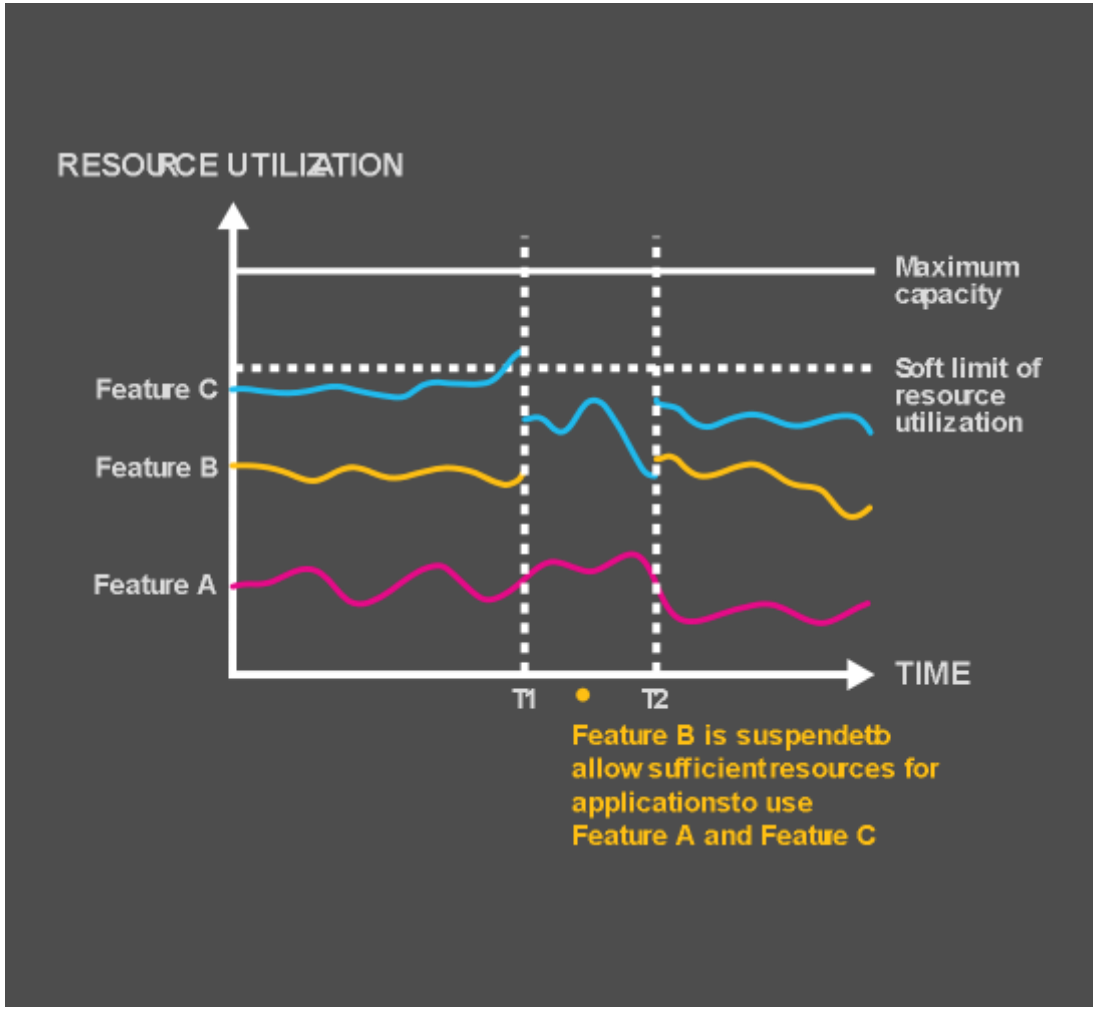


Static Content Hosting



Throttling

Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service. This pattern 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.



“ Resiliency

Resiliency

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

Bulkhead

Circuit Breaker

Compensating Transaction

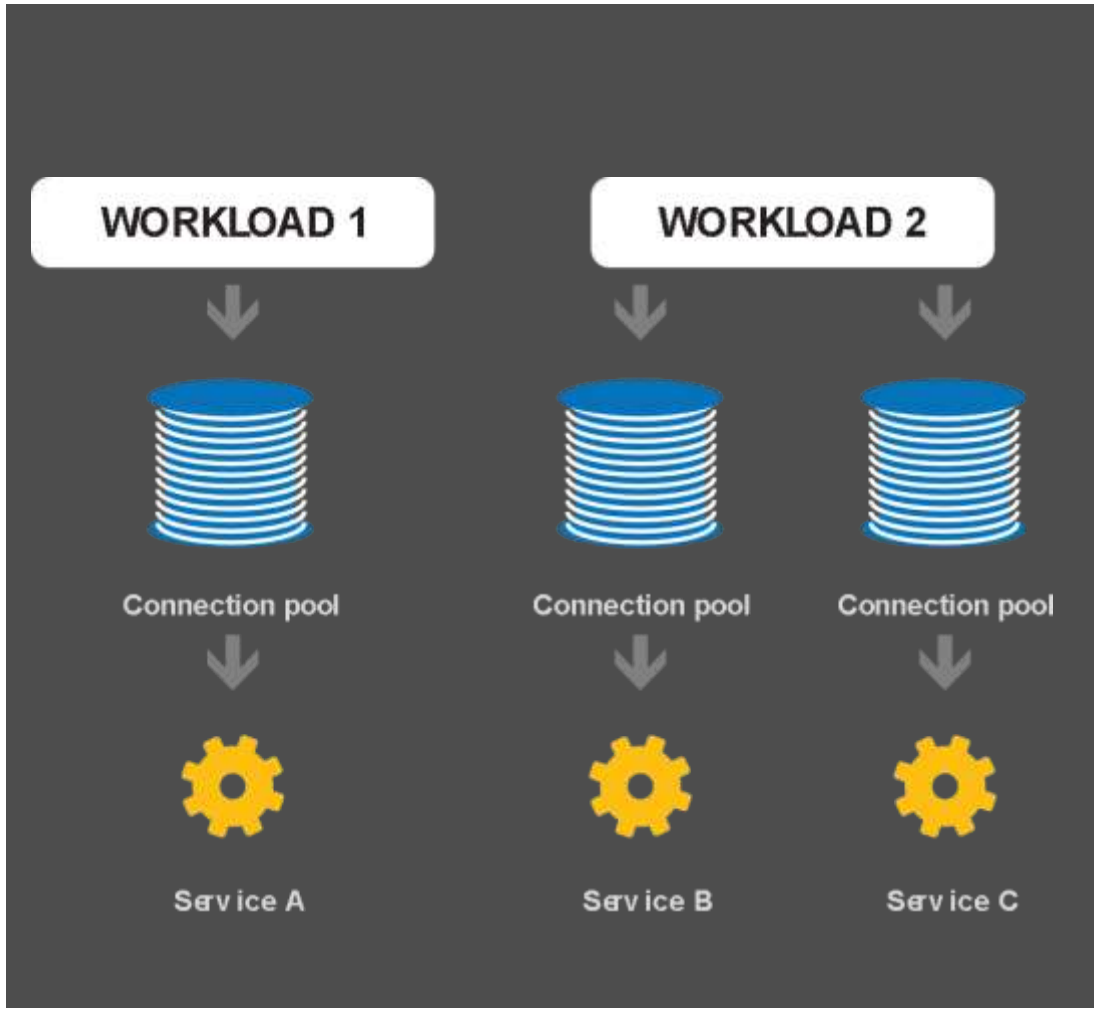
Leader Election

Retry

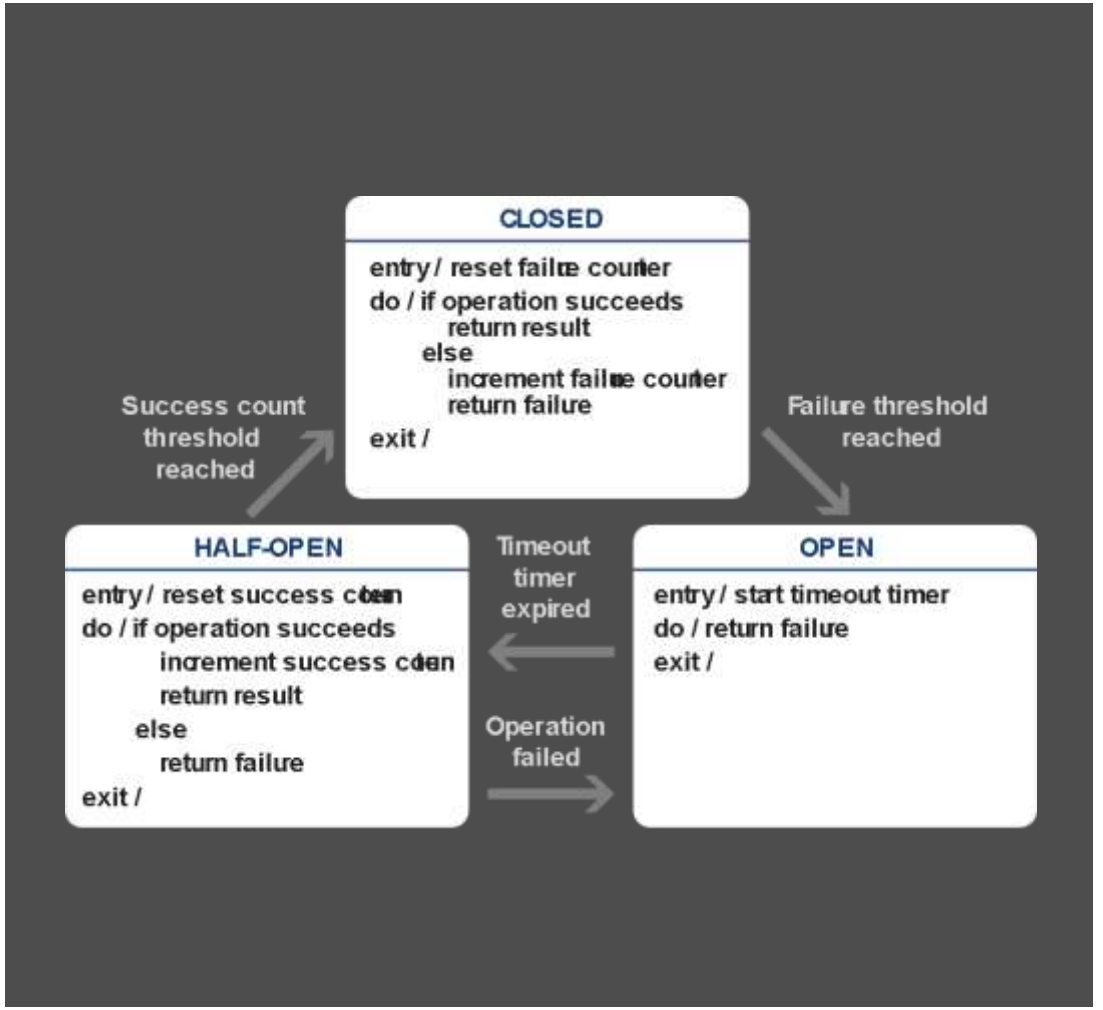
Scheduler Agent Supervisor

Bulkhead

Isolate the elements of an application like the sectioned partitions of a ship's hull. If the hull of a ship that uses bulkheads is compromised, only the damaged section of the ship will fill with water, and the ship will not sink. Using the same concept in application design, this pattern isolates instances of a service or clients into pools so that if one fails, the others will continue to function.



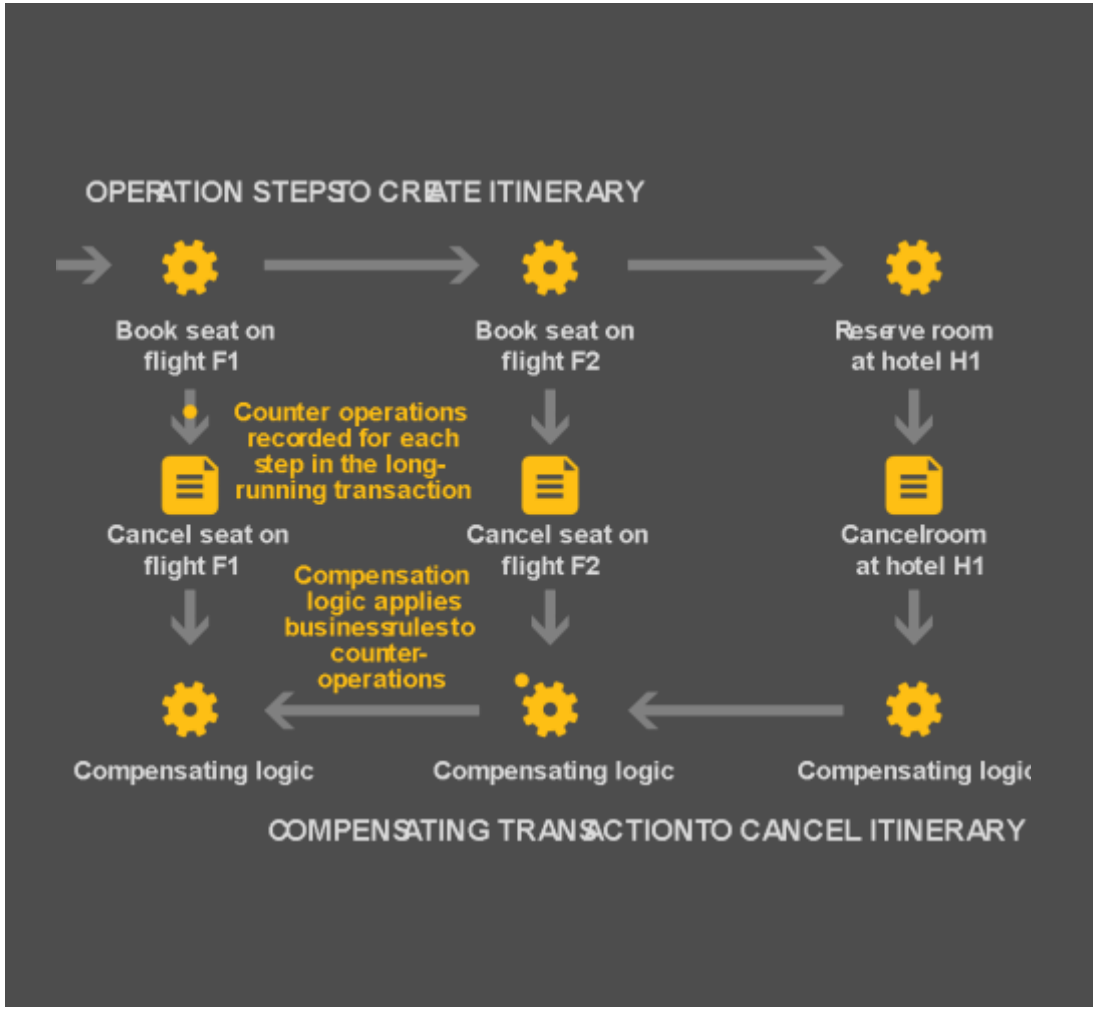
Circuit Breaker



Handle faults that may take a variable amount of time to rectify when connecting to a remote service or resource. This pattern can improve the stability and resiliency of an application.

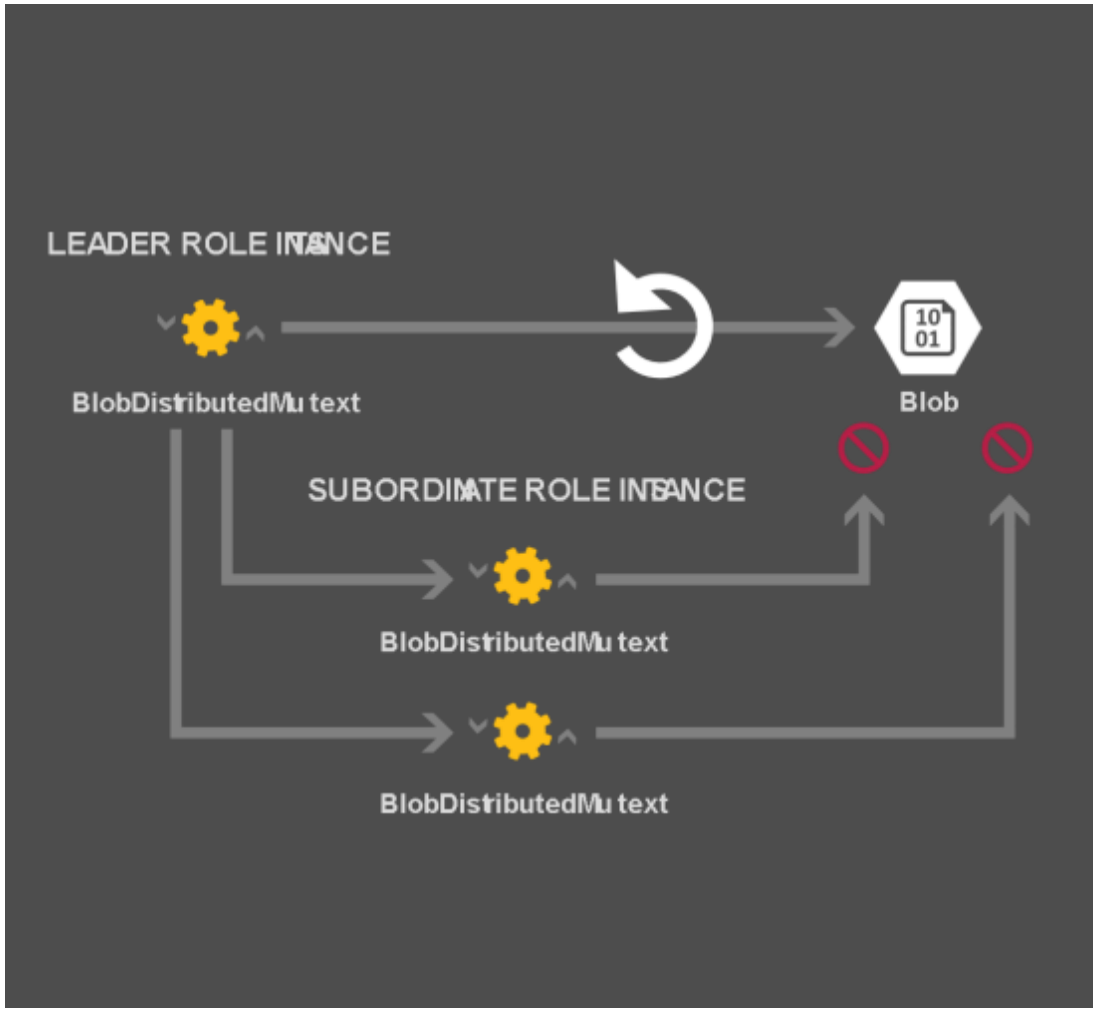
Compensating Transaction

Undo the work performed by a series of steps, which together define an eventually consistent operation, if one or more of the operations fails. Operations that follow the eventual consistency model are commonly found in cloud-hosted applications that implement complex business processes and workflows.



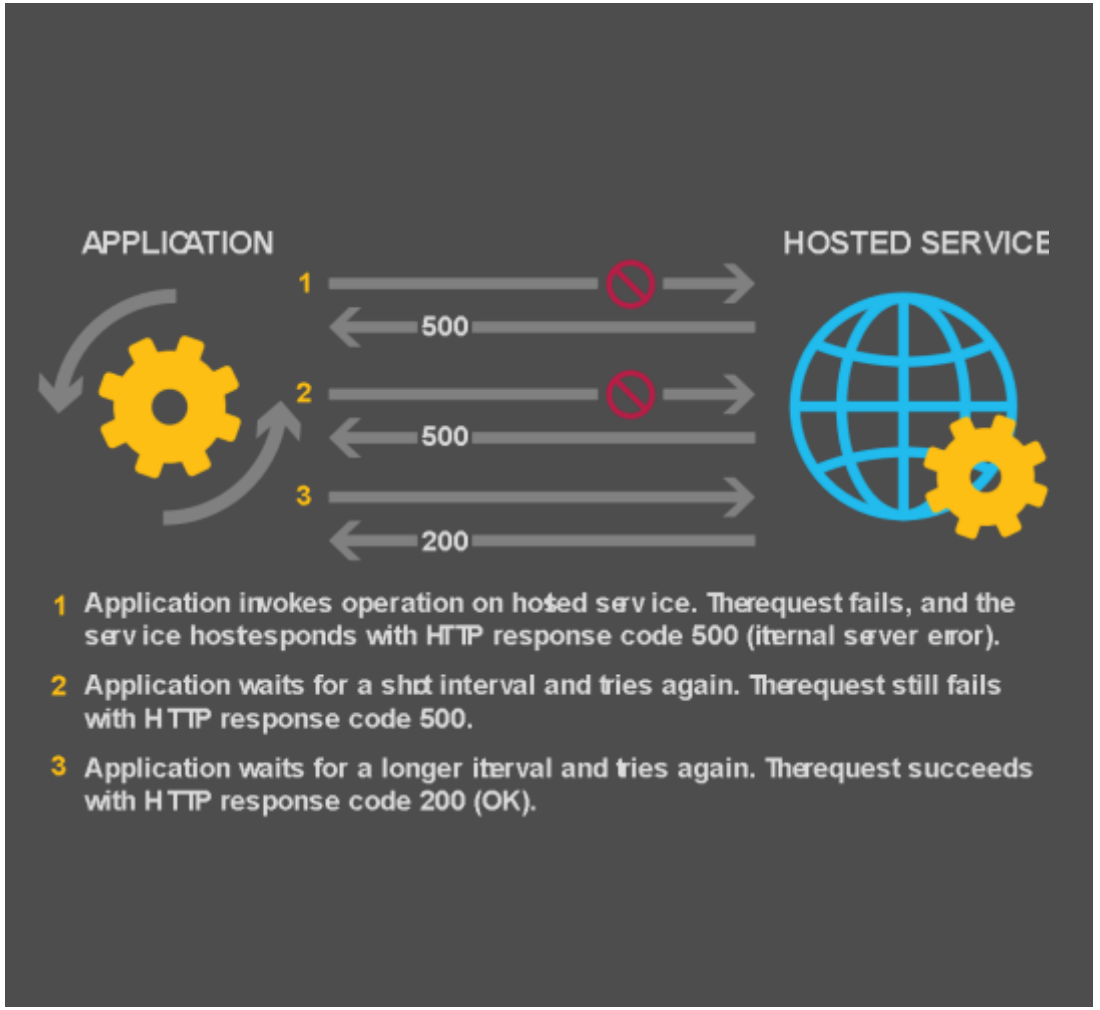
Leader Election

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. This pattern can help to ensure that task instances do not conflict with each other, cause contention for shared resources, or inadvertently interfere with the work that other task instances are performing.



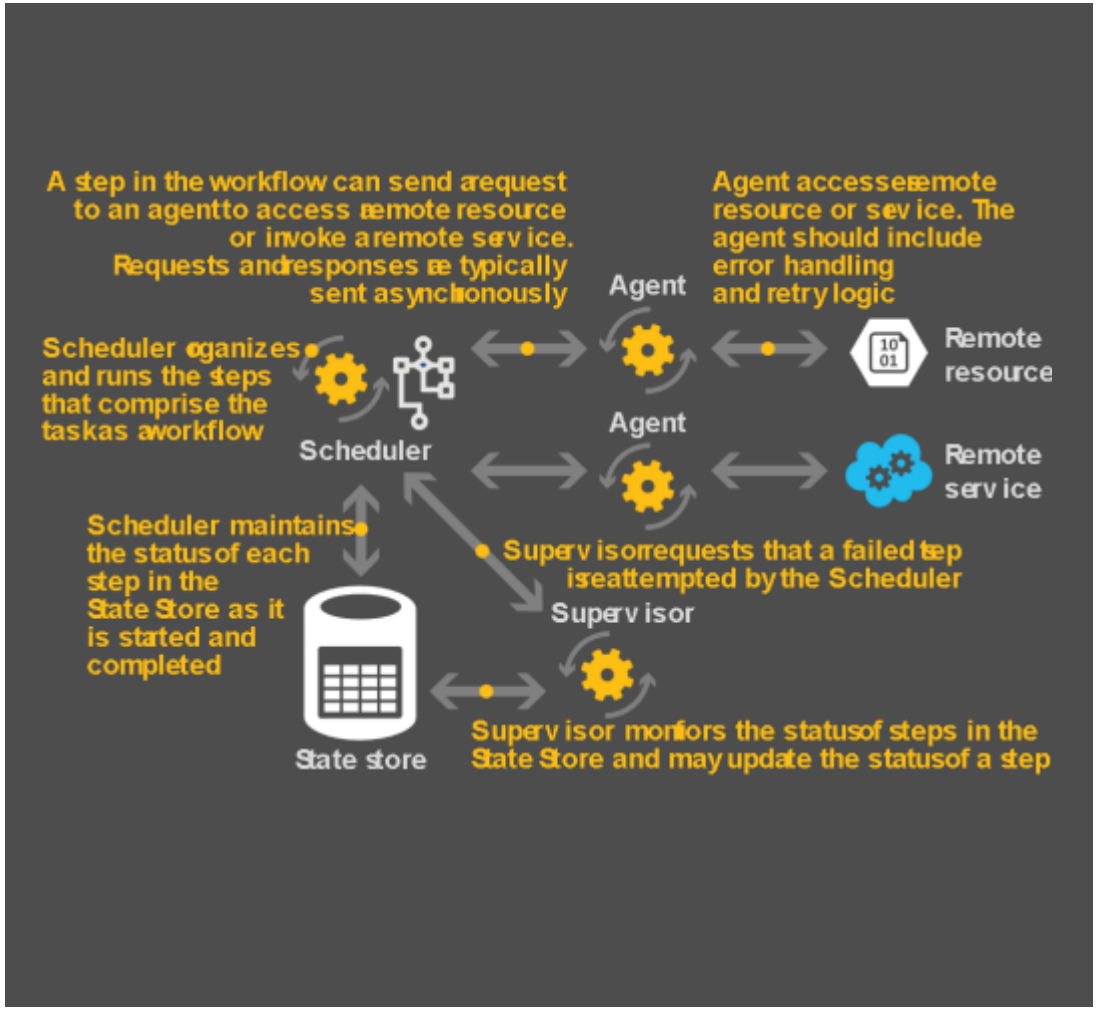
Retry

Enable an application to handle anticipated, temporary failures when it attempts to connect to a service or network resource by transparently retrying an operation that has previously failed in the expectation that the cause of the failure is transient. This pattern can improve the stability of the application.



Scheduler Agent Supervisor

Coordinate a set of actions across a distributed set of services and other remote resources, attempt to transparently handle faults if any of these actions fail, or undo the effects of the work performed if the system cannot recover from a fault. This pattern can add resiliency to a distributed system by enabling it to recover and retry actions that fail due to transient exceptions, long-lasting faults, and process failures.



“Security

Security

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

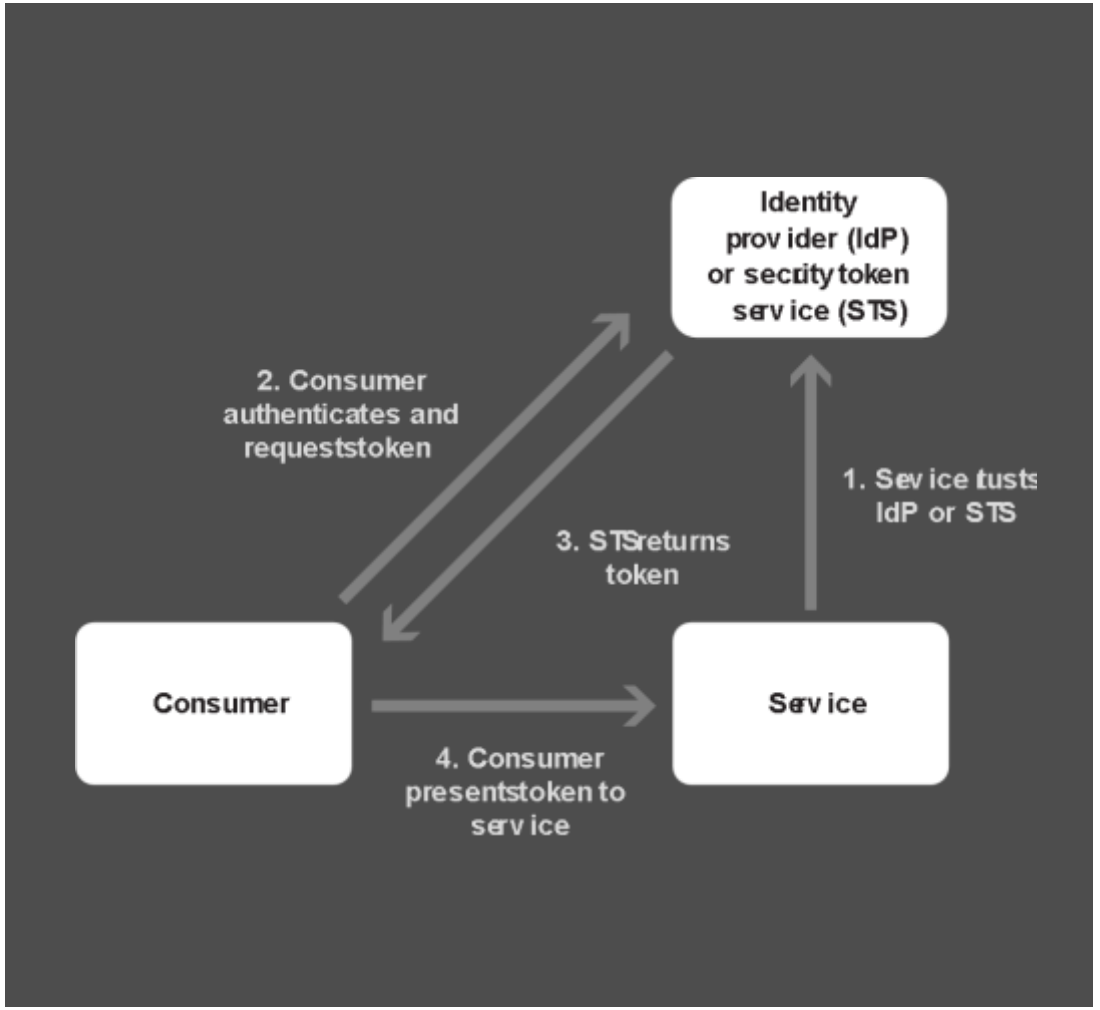
Federated Identity

Gatekeeper

Valet key

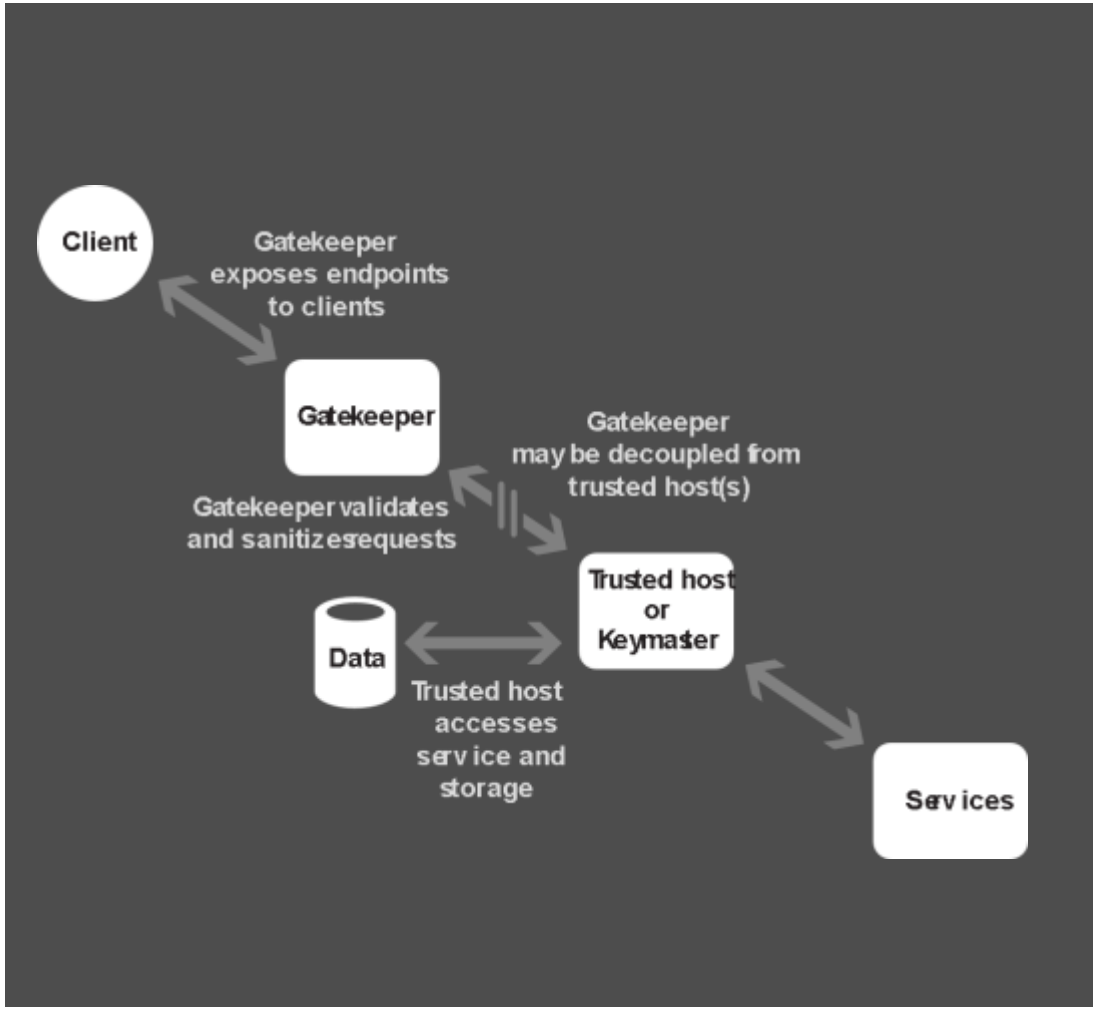
Federated Identity

Delegate authentication to an external identity provider. This pattern can simplify development, minimize the requirement for user administration, and improve the user experience of then application.



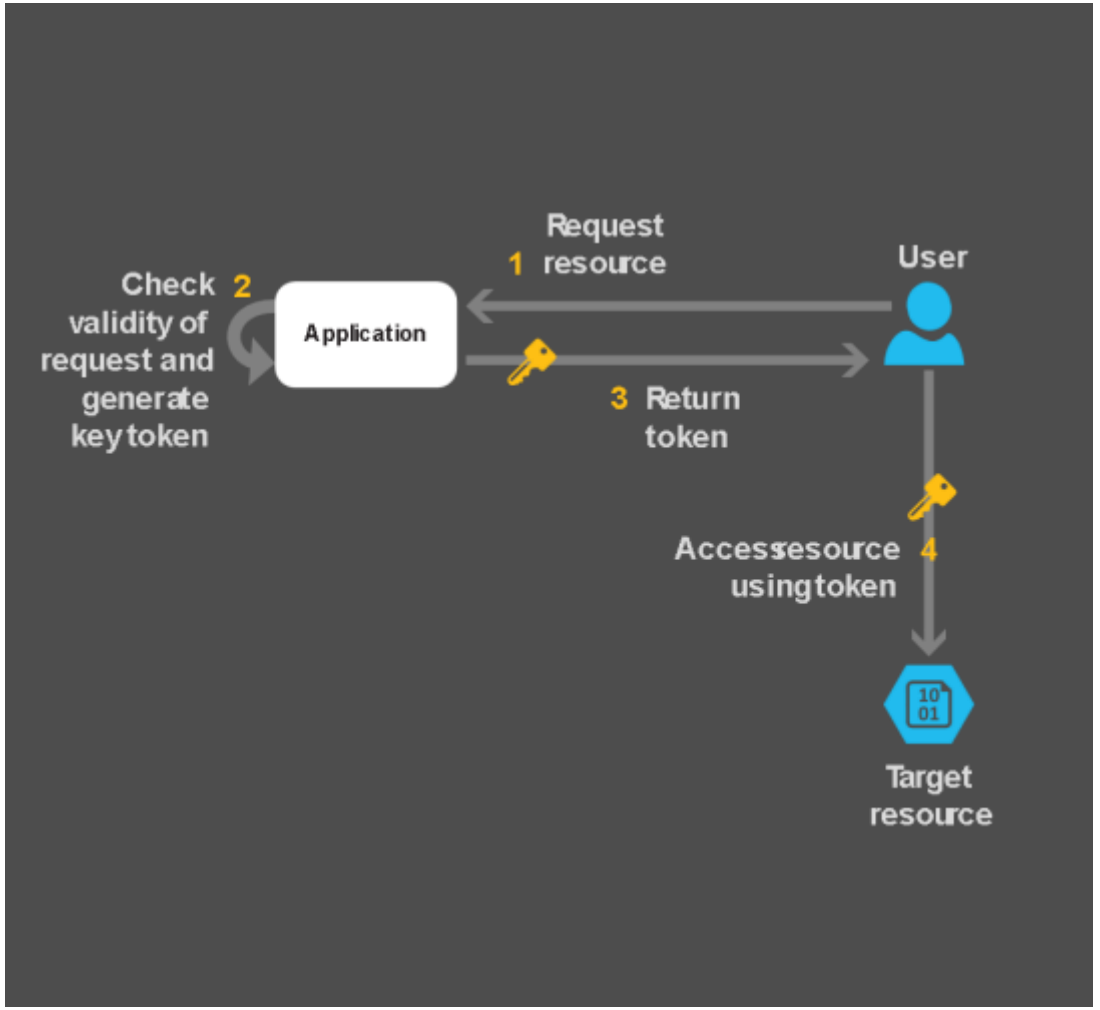
Gatekeeper

Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them. This pattern can provide an additional layer of security, and limit the attack surface of the system.



Valet key

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



“

Let’s restart from single patterns

Valet key

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

Security
Data Management

Queue-Based Load Leveling

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

Cache-aside

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

Command and Query Responsibility Segregation (CQRS)

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

Index Table

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

Materialized View

Generate pre-populated views over the data in one or more data stores when the data is formatted in a way that does not favor the required query operations. This pattern can help to support efficient querying and data extraction, and improve performance.