Manoj Ganapathi, CodeOps

Cloud Design Patterns with Microsoft Azure

About Me

- Manoj is a seasoned IT professional with more than 20 years of experience. He has extensive experience in enterprise & solution architecture, design and implementation of large & complex enterprise systems. As an architect and technology consultant, he has consulted with several large, fortune 500 enterprises and worked with ISVs and startups. In his career, he has worked in multiple technology-oriented and leadership roles across all phases of software development life cycle. He is experienced in building and running technical communities and has been a speaker in several technology conferences.
- Over the last decade, he has worked extensively on consulting, architecture and implementation of Cloud-based solutions, specializing on building highly scalable, resilient systems and DevOps practices on the Microsoft Cloud.
- Currently, he is the Chief Architect at CodeOps Technologies (http://codeops.tech/) and a Digital Technology Consultant.
- LinkedIn profile: https://www.linkedin.com/in/manojg
- @manojgr, manoj@codeops.tech

Key Design Principles

1) Design for self-healing
2) Make all things redundant
3) Minimize coordination
4) Design to scale out
5) Partition around limits
6) Design for operations
7) Use managed services
8) Use the best data store for the job
9) Design for evolution
10) Build for the needs of business

- Design principles for Azure applications - Azure Architecture Center | Microsoft Docs

Azure Well-Architected Framework

Microsoft Azure Well-Architected Framework - Microsoft Azure Well-Architected Framework introduction | Microsoft Docs

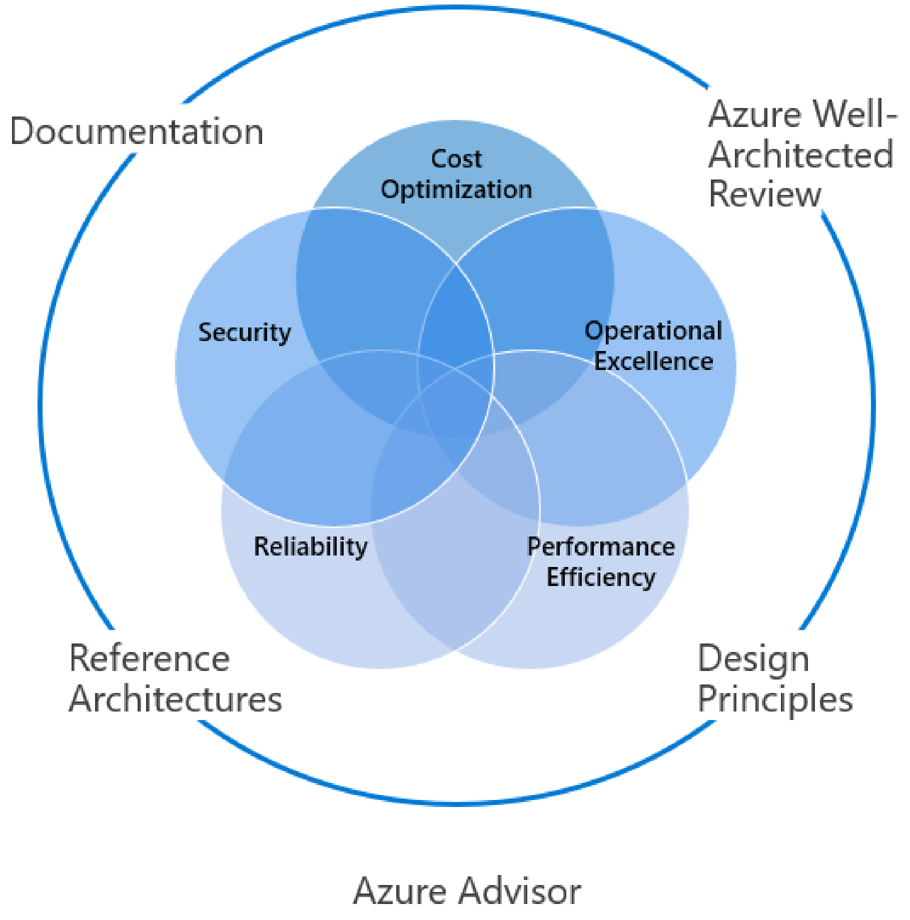


Image Source: Azure Well-Architected Framework - AzToso.com

Why Cloud Patterns?

Broad definition of a Pattern:

- General reusable solution to a recurring problem
- Incorporate best practices
- Allow for better communication.

Cloud Patterns

- These design patterns are useful for building reliable, scalable, secure applications in the cloud.
- Influenced by the different categories of challenges/aspects:
  - Data Management
  - Messaging
  - Design and Implementation
  - Performance Efficiency
  - Operational Excellence
  - Reliability
  - Security

Data Management

Influences most of the quality attributes.

High availability, recoverability and performance

Maintaining consistency

Synchronization across different locations.

Design and Implementation

Consistency and coherence in component design and deployment

Maintainability to simplify administration and development

Reusability to allow components and subsystems to be used in other applications and in other scenarios.

Have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.

Messaging

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

Asynchronous messaging is widely used, and provides many benefits, but also brings challenges:

- Ordering of messages
- Poison message management
- Idempotency

Performance Efficiency

Ability of your workload to scale to meet the demands placed on it by users in an efficient manner.

Scalability is ability of a system either to handle increases in load without impact on performance or for the available resources to be readily increased.

Operational Excellence

Applications must expose runtime information that administrators and operators can use to manage and monitor the system

Supporting changing business requirements and customization without requiring the application to be stopped or redeployed.

Reliability

Availability is measured as a percentage of uptime and defines the proportion of time that a system is functional and working.

Availability is affected by system errors, infrastructure problems, malicious attacks, and system load.

Cloud applications typically provide users with a service level agreement (SLA), which means that applications must be designed and implemented to maximize availability.

Security

Security provides confidentiality, integrity, and availability assurances against malicious attacks on information

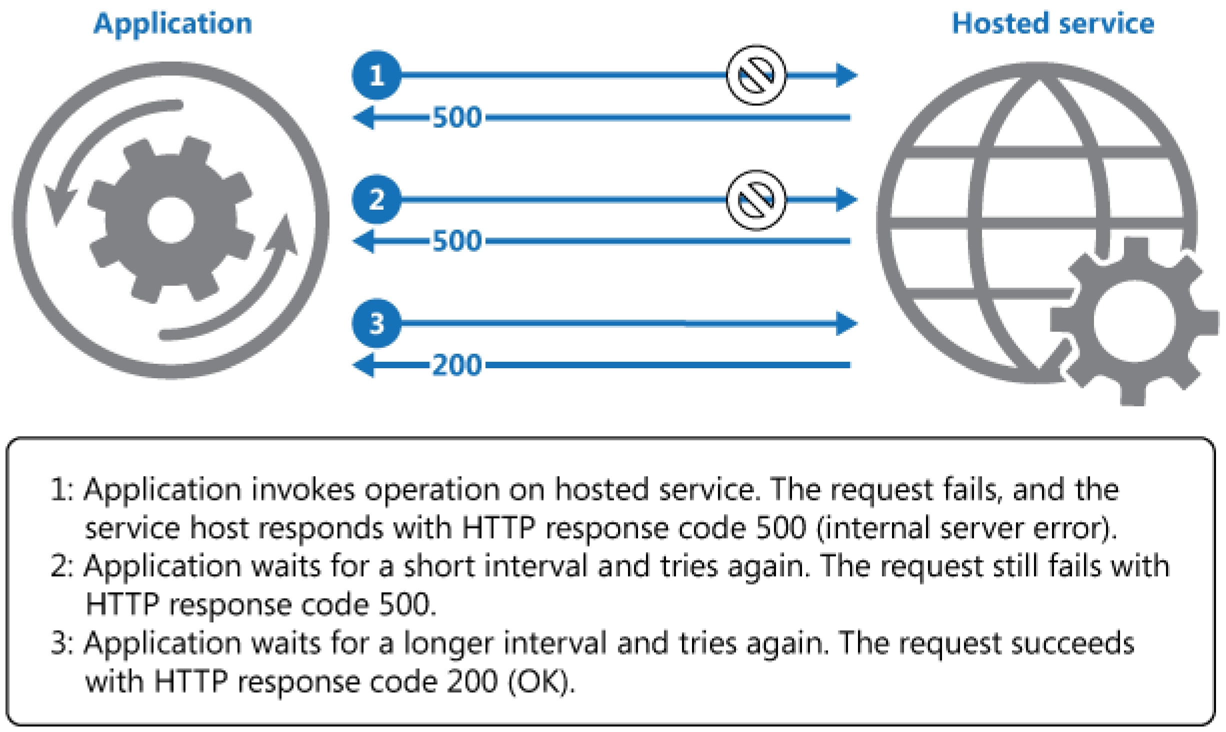
Losing these assurances can negatively impact your business operations and revenue, as well as your organization’s reputation in the marketplace.

Maintaining security requires following well-established practices (security hygiene) and being vigilant to detect and rapidly remediate vulnerabilities and active attacks.

Lab Set 1

Problem Statement	Solution	Alternatives/Related Patterns
You are an architect of a company that provides Weather APIs. You rely on a large number of external partner APIs which have a history of intermittent failures and reliability issues. You want to ensure these issues do not adversely affect the stability and experience of your services.	You implement the Circuit Breaker Pattern using the Polly package to detect repeated failures on the 3rd party service and temporarily disable the endpoint. This way the dependent service can fallback gracefully. The Circuit breaker works in concert with the Health Endpoint Monitoring pattern - a dedicated URL exposed by the endpoint to check on the health and liveness periodically. This is done using the ASP.NET Core built-in health checks framework.	The Retry pattern can be implemented to retry requests on a 3rd party service which has intermittent/transient failures and back-off after repeated failures (supported by Polly)

Retry



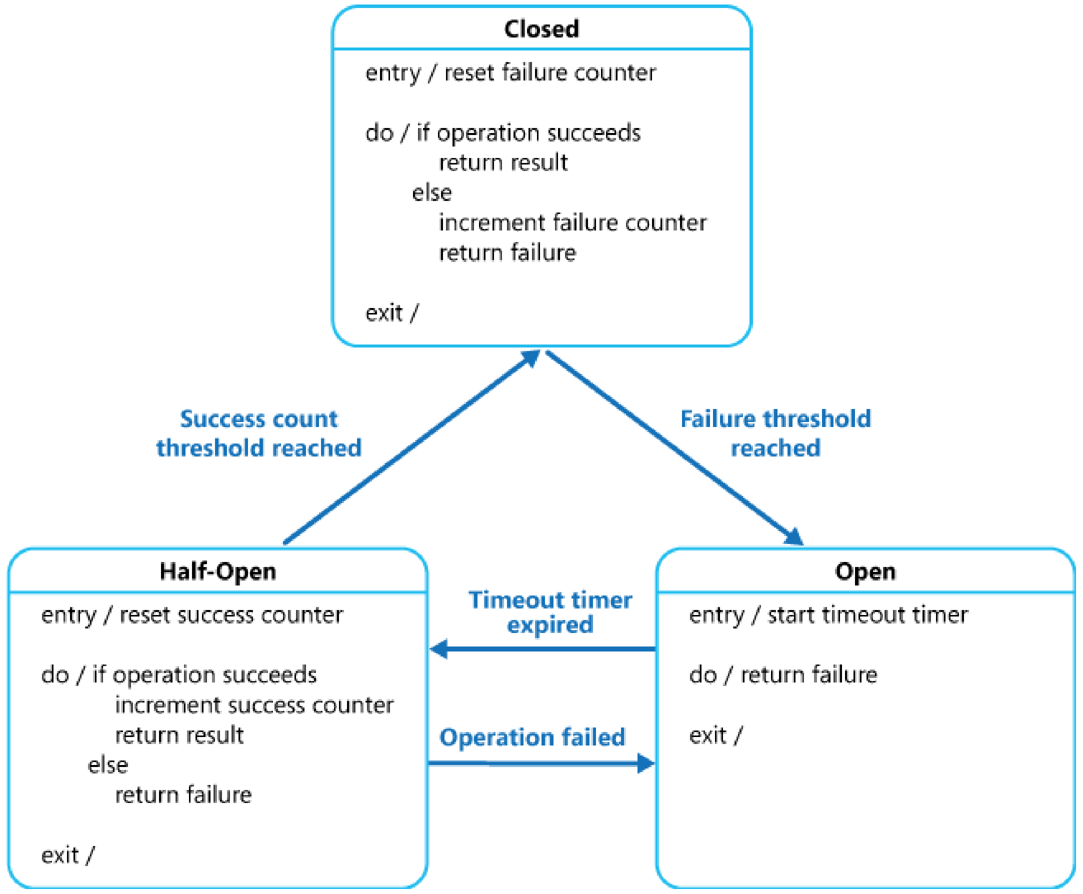
1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

- Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation. This can improve the stability of the application.

Retry guidance for Azure services - Best practices for cloud applications | Microsoft Docs

Retry pattern - Cloud Design Patterns | Microsoft Docs

Circuit Breaker



Closed
entry / reset failure counter
do / if operation succeeds
return result
else
increment failure counter
return failure
exit /
Success count threshold reached
Failure threshold reached
Half-Open
entry / reset success counter
do / if operation succeeds
increment success counter
return result
else
return failure
exit /
Timeout timer expired
Operation failed
Open
entry / start timeout timer
do / return failure
exit /

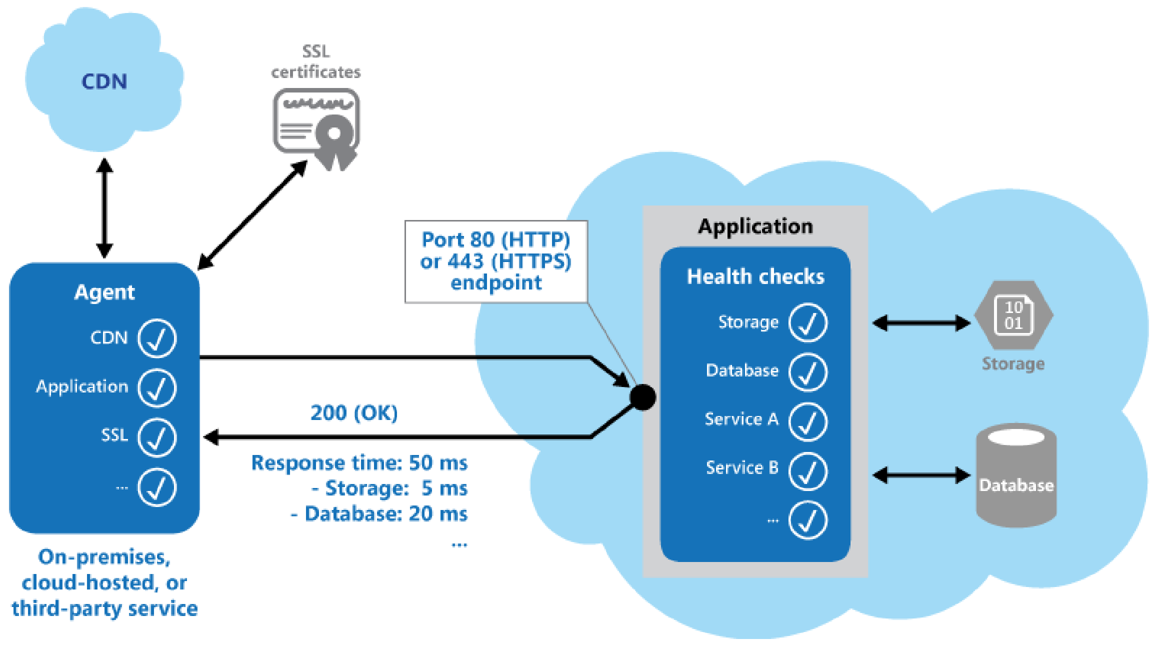
- Handle faults that might take a variable amount of time to recover from, when connecting to a remote service or resource.
- This can improve the stability and resiliency of an application.

Implementing the Circuit Breaker pattern | Microsoft Docs

Transient fault handling and proactive resilience engineering · App-vNext/Polly Wiki (github.com)

Using Polly Circuit Breakers for Resilient .NET Web Service Consumers - Twilio

Health Endpoint Monitoring



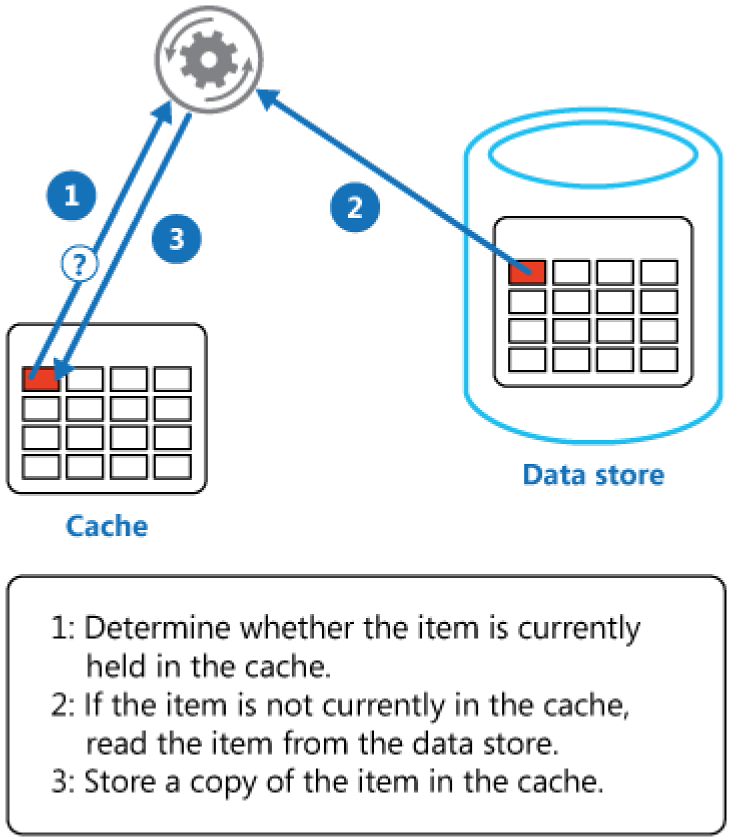
- Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks and return an indication of its status.
- A health monitoring check typically combines two factors:
  - The checks (if any) performed by the application or service in response to the request to the health verification endpoint.
  - Analysis of the results by the tool or framework that performs the health verification check.

[Health Endpoint Monitoring pattern - Cloud Design Patterns | Microsoft Docs]

Lab Set 2

Problem Statement	Solution	Alternatives/Related Patterns
You are an architect of a company that provides Weather APIs. You want to improve the performance (response time) of your API by caching results of your temperature API (for which you rely on a partner).	You use the Cache Aside pattern to cache responses based on a policy. You use Polly's Cache policy for the HttpClient with an InMemoryCache provider (optionally, a distributed cache with Azure Redis Cache when deployed in a cluster).	

Cache Aside



1: Determine whether the item is currently held in the cache.
2: If the item is not currently in the cache, read the item from the data store.
3: Store a copy of the item in the cache.

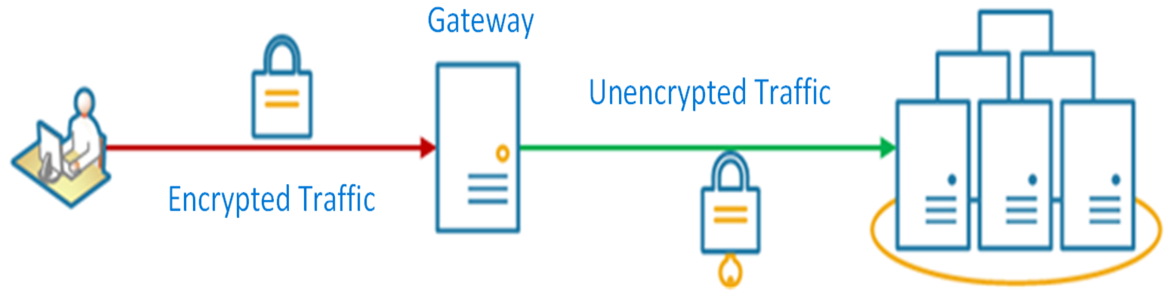
- Load data on demand into a cache from a data store. This can improve performance and also helps to maintain consistency between data held in the cache and data in the underlying data store.

[Cache-Aside pattern - Cloud Design Patterns | Microsoft Docs]

Lab Set 3

Problem Statement	Solution	Alternatives/Related Patterns
You are an architect of a company that provides Weather APIs. You want to add resilience to your APIs by adding features like throttling. You also want the flexibility of supporting multiple concurrent versions for your backend APIs. You rely on a 3rd party for the Temperature API, which is considered legacy. You want to decouple your technology implementation from this and stick to modern protocols and message formats	You employ the Gateway Offloading and Throttling patterns by leveraging Azure API management service- using it to wrap your API and configure appropriate rules. You employ the Ambassador patterns to wrap the legacy API and gives it a modern façade by employing appropriate transformation policies	Additionally, Gateway Routing pattern can be implemented by configuring routing rules to connect to different micro service backends using a single endpoint (differentiated by URL paths).

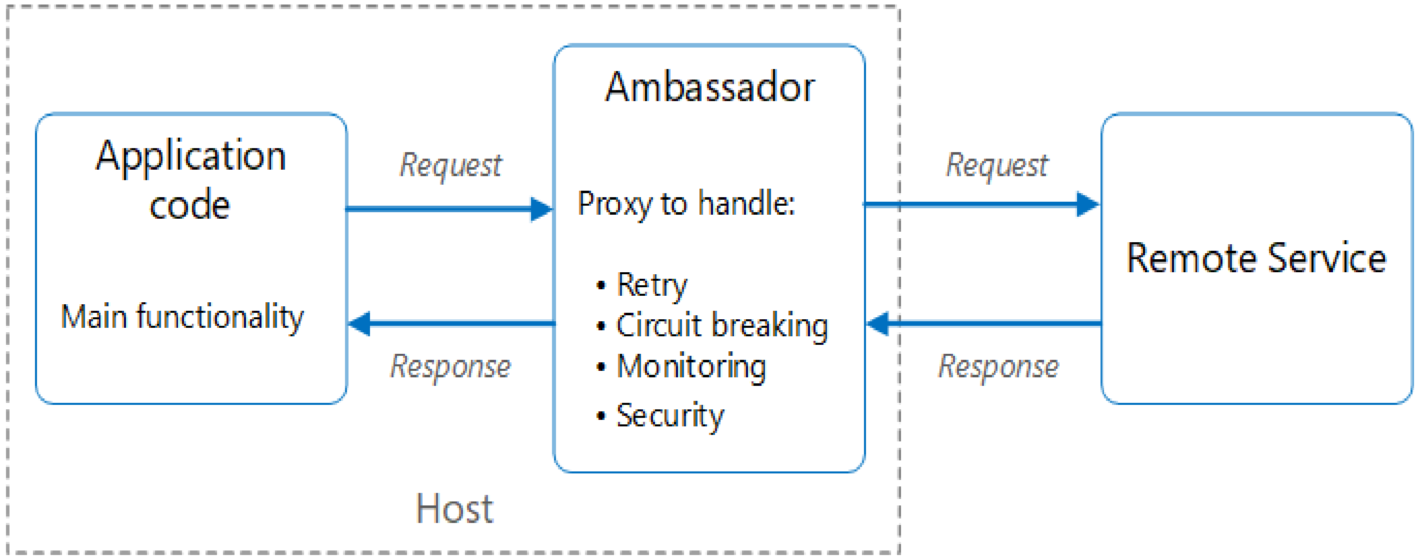
Gateway Offloading



- Offload some features into a gateway, particularly cross-cutting concerns such as certificate management, authentication, SSL termination, monitoring, protocol translation, or throttling.

[Gateway Offloading pattern - Cloud Design Patterns | Microsoft Docs]

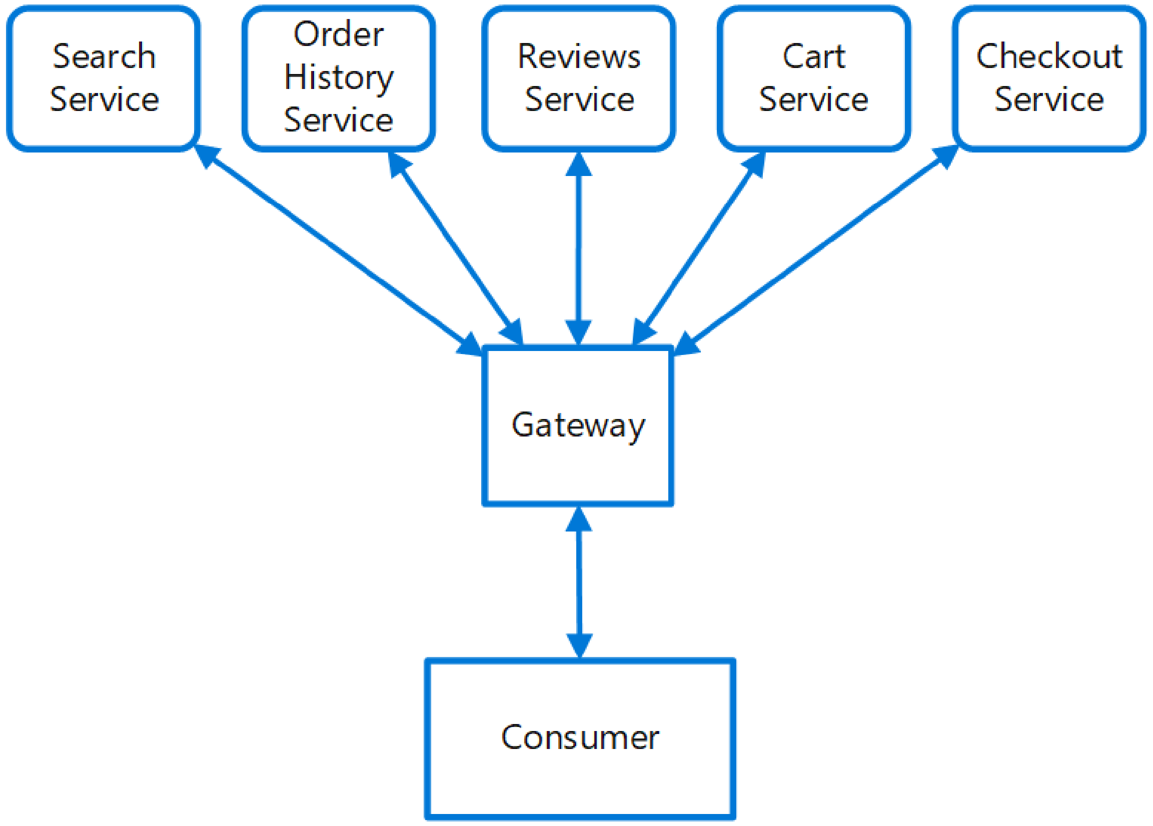
Ambassador



- 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.

[Ambassador pattern - Cloud Design Patterns | Microsoft Docs]

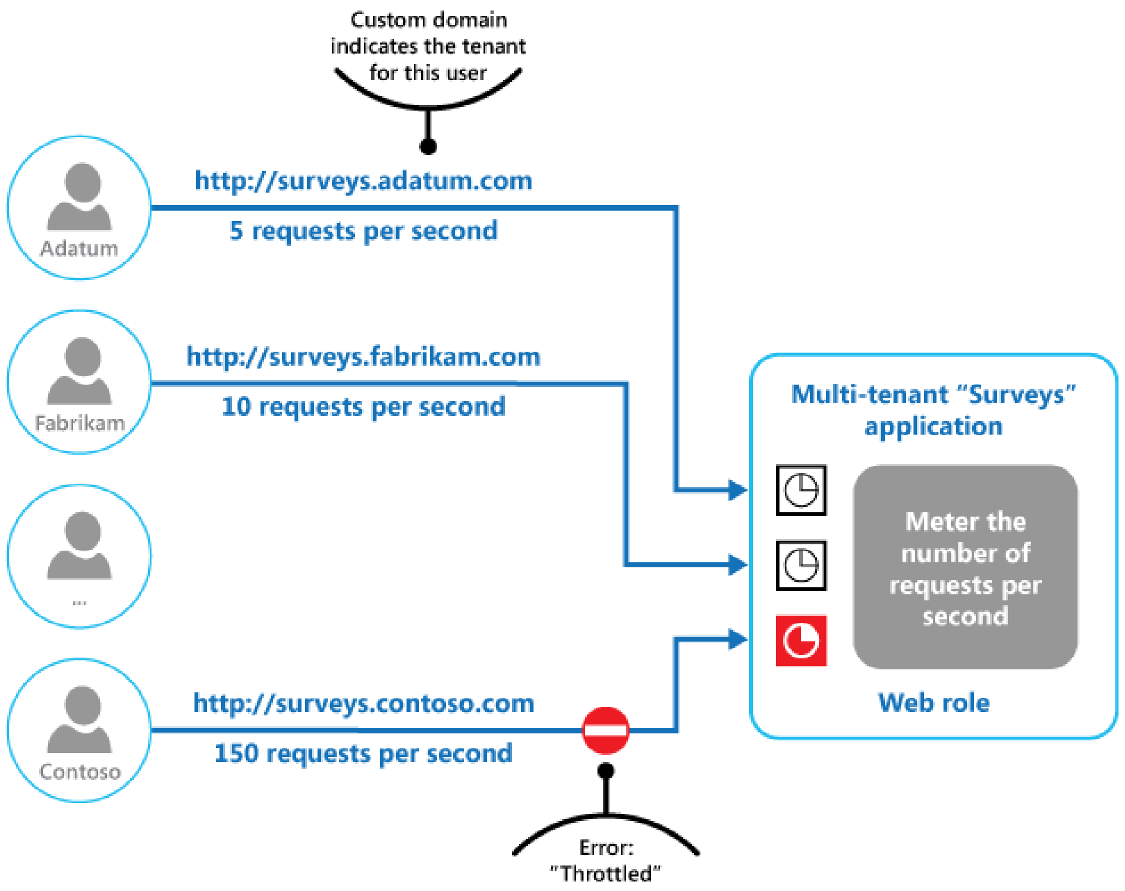
Gateway Routing



- Place a gateway in front of a set of applications, services, or deployments. Use application Layer 7 routing to route the request to the appropriate instances.
- With this pattern, the client application only needs to know about and communicate with a single endpoint. If a service is consolidated or decomposed, the client does not necessarily require updating. It can continue making requests to the gateway, and only the routing changes.
- A gateway also lets you abstract backend services from the clients, allowing you to keep client calls simple while enabling changes in the backend services behind the gateway. Client calls can be routed to whatever service or services need to handle the expected client behavior, allowing you to add, split, and reorganize services behind the gateway without changing the client.

Gateway Routing pattern - Cloud Design Patterns | Microsoft Docs

Throttling


Custom domain indicates the tenant for this user
Adatum
http://surveys.adatum.com
5 requests per second
Fabrikam
http://surveys.fabrikam.com
10 requests per second
...
Contoso
http://surveys.contoso.com
150 requests per second
Error: "Throttled"
Multi-tenant "Surveys" application
Meter the number of requests per second
Web role

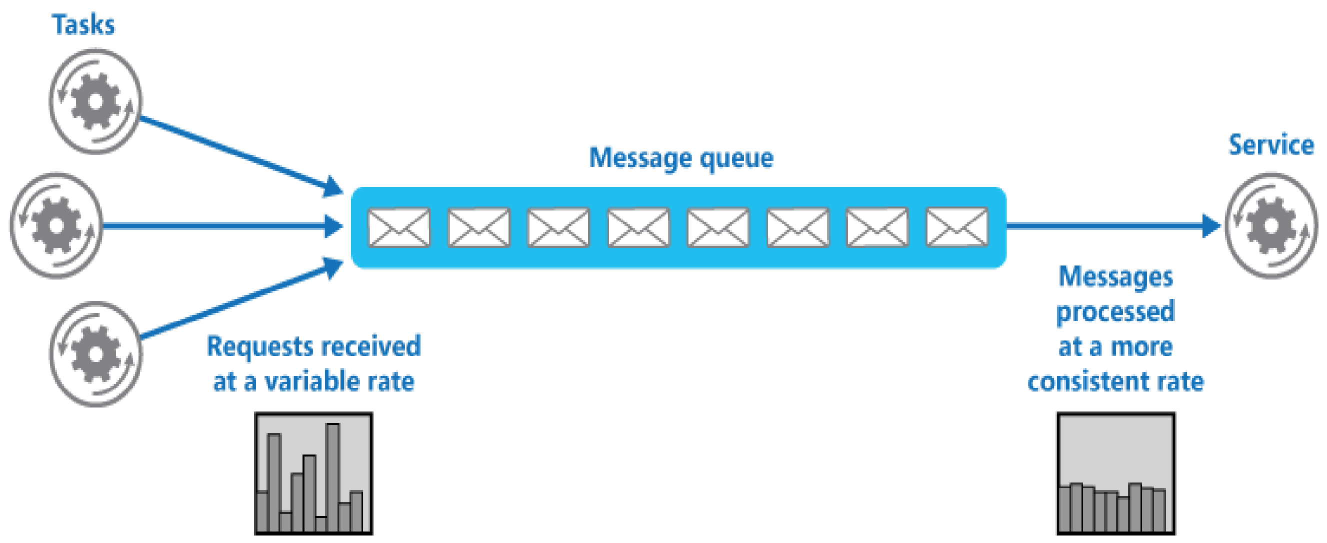
- An alternative strategy to autoscaling is to allow applications to use resources only up to a limit, and then throttle them when this limit is reached.
- The system should monitor how it's using resources so that, when usage exceeds the threshold, it can throttle requests from one or more users.
-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

Throttling pattern - Cloud Design Patterns | Microsoft Docs

Lab Set 4

Problem Statement	Solution	Alternatives/Related Patterns
You are building a solution for COVID front-line workers. You want to build a scalable solution for transcribing audio report files uploaded by front line workers. Also, you want to support a background process which performs text analytics on the transcribed data you get from the first step, but you want to do this without affecting the performance of your end-user facing systems.	You implement the Competing Consumer pattern with Azure Functions and a Blob Trigger. The function does the work of transcribing with another blob configured as the output binding. This in turn, triggers a series of text analytics tasks (like Entity Recognition etc.) which are invoked by different Azure functions in parallel. The transcribing and the text analytics tasks are decoupled by leveraging the Queue based load levelling pattern	A slight variant is the use of the Pipes & Filters pattern, where you use an Azure Queue based pipeline and a series of Azure Functions which perform different tasks on the same message, enriching it along the way

Queue-Based Load Levelling

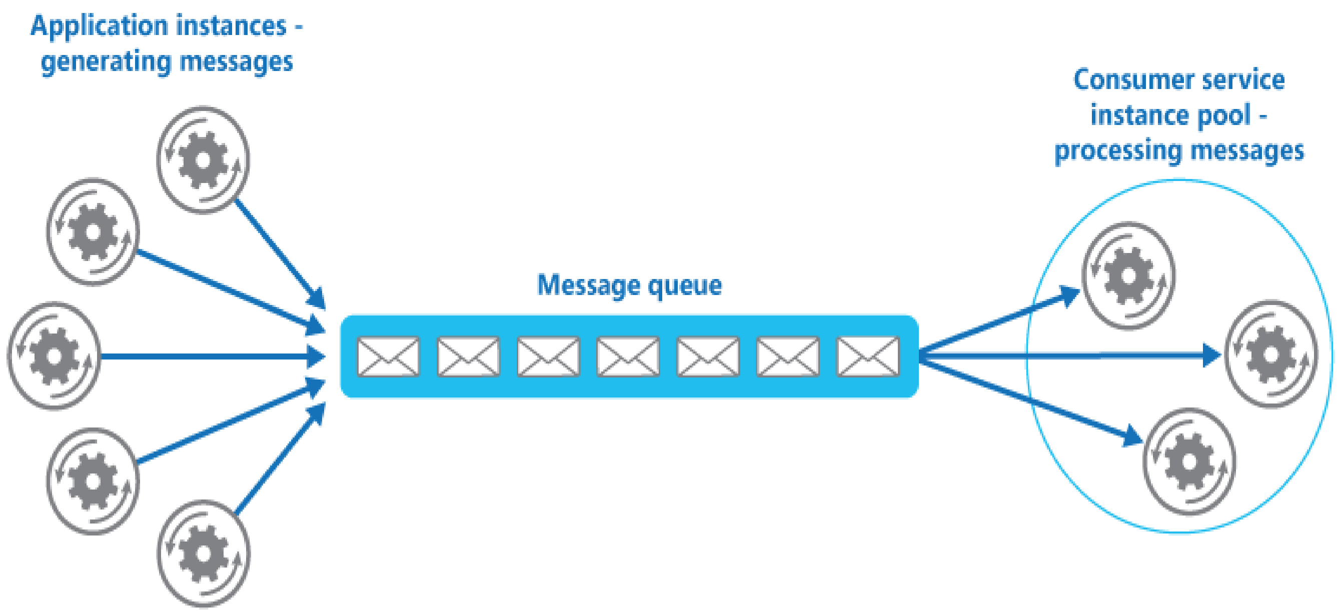

Tasks
Message queue
Service
Requests received at a variable rate
Messages processed at a more consistent rate

- 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

Compare Azure Storage queues and Service Bus queues - Azure Service Bus | Microsoft Docs

Queue-Based Load Leveling pattern - Cloud Design Patterns | Microsoft Docs

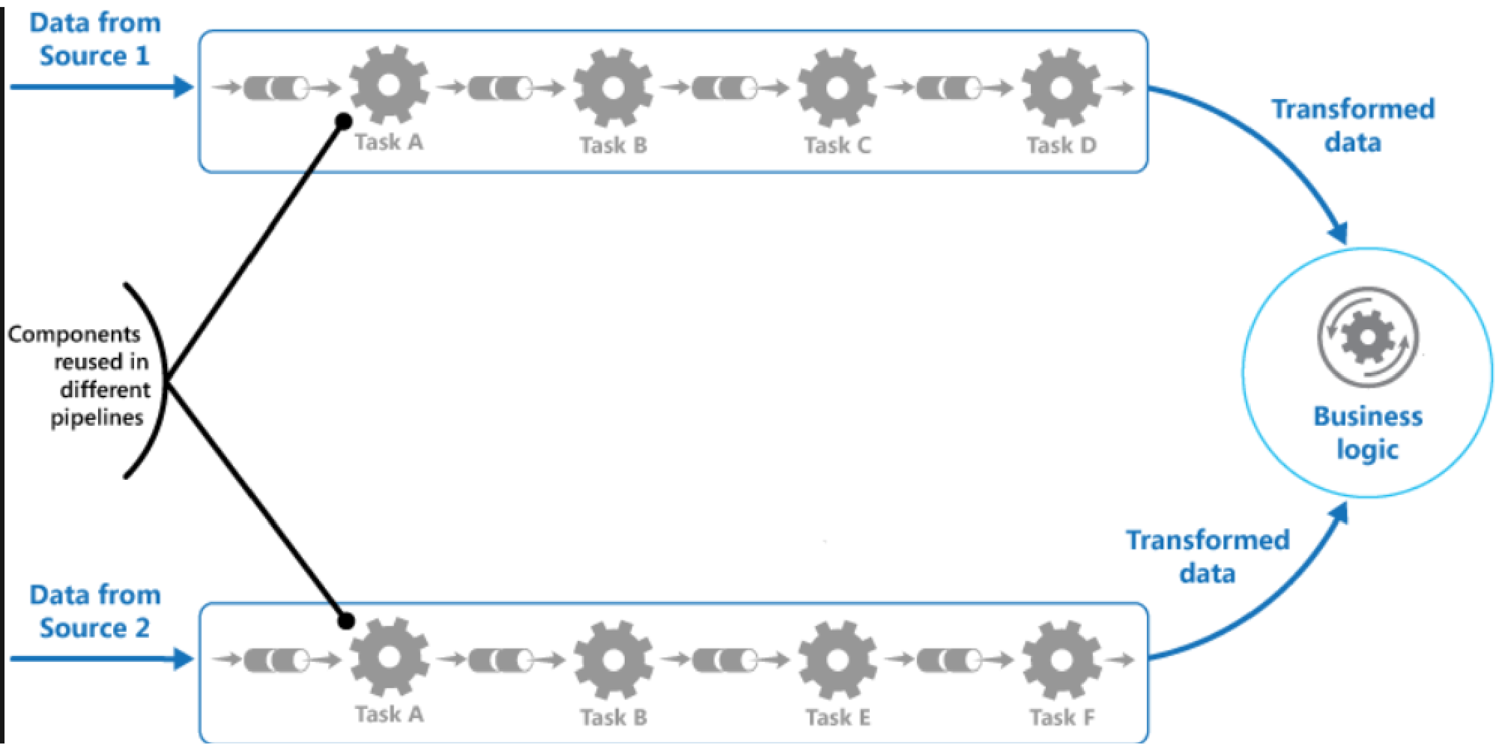
Competing Consumer


Application instances - generating messages
Message queue
Consumer service instance pool - processing messages

- Use a message queue to implement the communication channel between the application and the instances of the consumer service.
- The application posts requests in the form of messages to the queue, and the consumer service instances receive messages from the queue and process them.
- This approach enables the same pool of consumer service instances to handle messages from any instance of the application

Competing Consumers pattern - Cloud Design Patterns | Microsoft Docs

Pipes and Filters



- Break down the processing required for each stream into a set of separate components (or filters), each performing a single task.
- By standardizing the format of the data that each component receives and sends, these filters can be combined together into a pipeline.
- This helps to avoid duplicating code, and makes it easy to remove, replace, or integrate additional components if the processing requirements change.

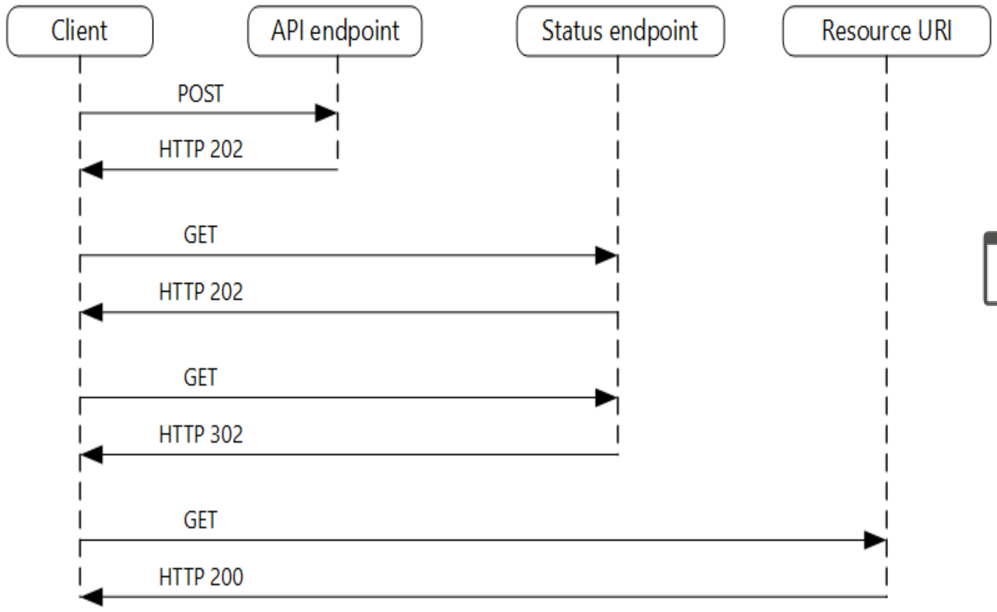
Pipes and Filters pattern - Cloud Design Patterns | Microsoft Docs

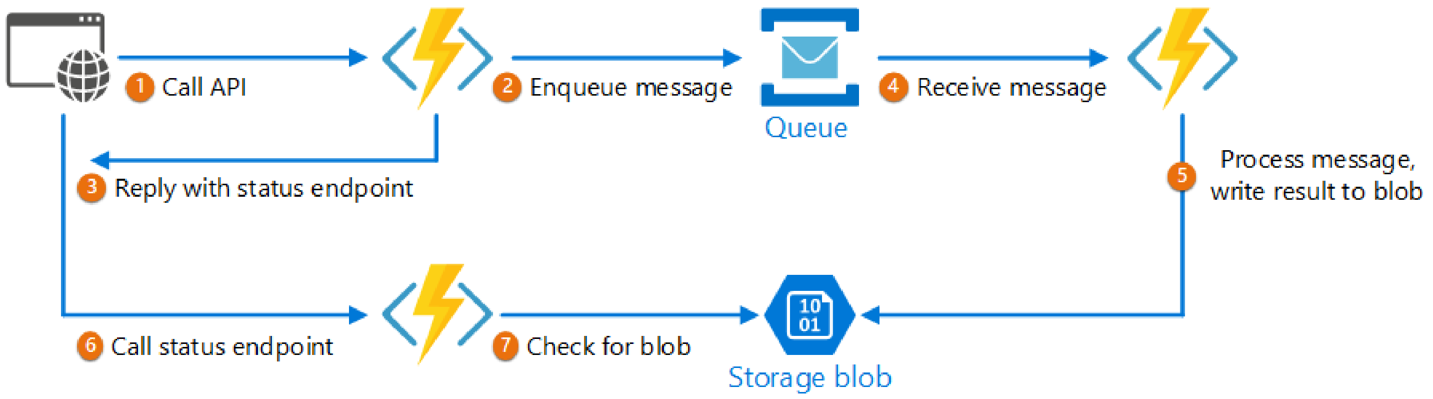
Asynchronous Request-Reply pattern

Decouple backend processing from a frontend host, where backend processing needs to be asynchronous, but the frontend still needs a clear response.

- Two implementation approaches:

1. Http Polling
2. Service-side persistent network connections such as WebSockets or SignalR



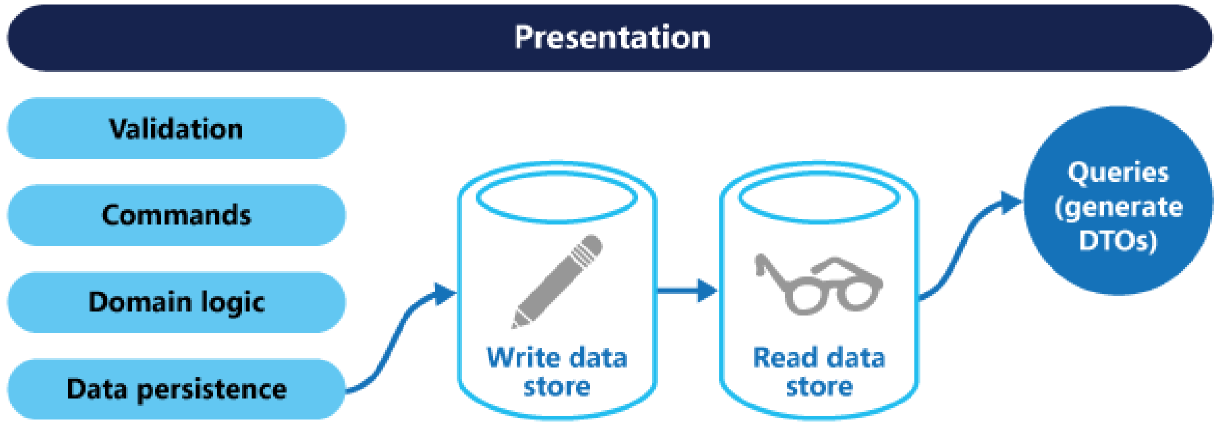


[Asynchronous Request-Reply Pattern - Azure Architecture Center | Microsoft Docs]()

Lab Set 5

Problem Statement	Solution	Alternatives/Related Patterns
You are the architect in an ecommerce company, where the response times for the end user application is critical. While the application sees large volumes of order created by users, you want to allow users to search their order data and use different type of queries. You don't want the query load of the application to affect the transaction processing performance of the orders	You employ the CQRS pattern to segregate the order query processing logic and data source and the order creation logic and its data source (represented as two separate Cosmos DB containers). MediatR is used to decouple the command and query processing logic from the consuming service. They are deployed and scaled independently without impacting one other. On the data storage a separate Materialized View is created just to support querying in another Cosmos DB container. The materialized view is updated using the Change feed feature of Cosmos DB, leveraging an Azure Function	The Change feed feature also supports the Event Sourcing pattern where specific change events can be consumed and replayed

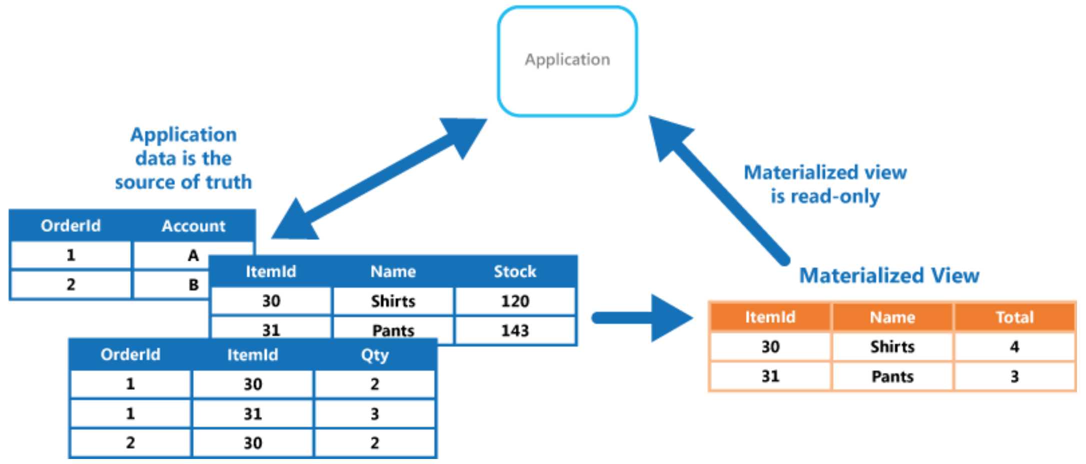
CQRS



- 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.

[CQRS pattern - Azure Architecture Center | Microsoft Docs]

Materialized View


Application
Application data is the source of truth
Materialized view is read-only
Materialized View

OrderId	Account
1	A
2	B

ItemId	Name	Stock
30	Shirts	120
31	Pants	143

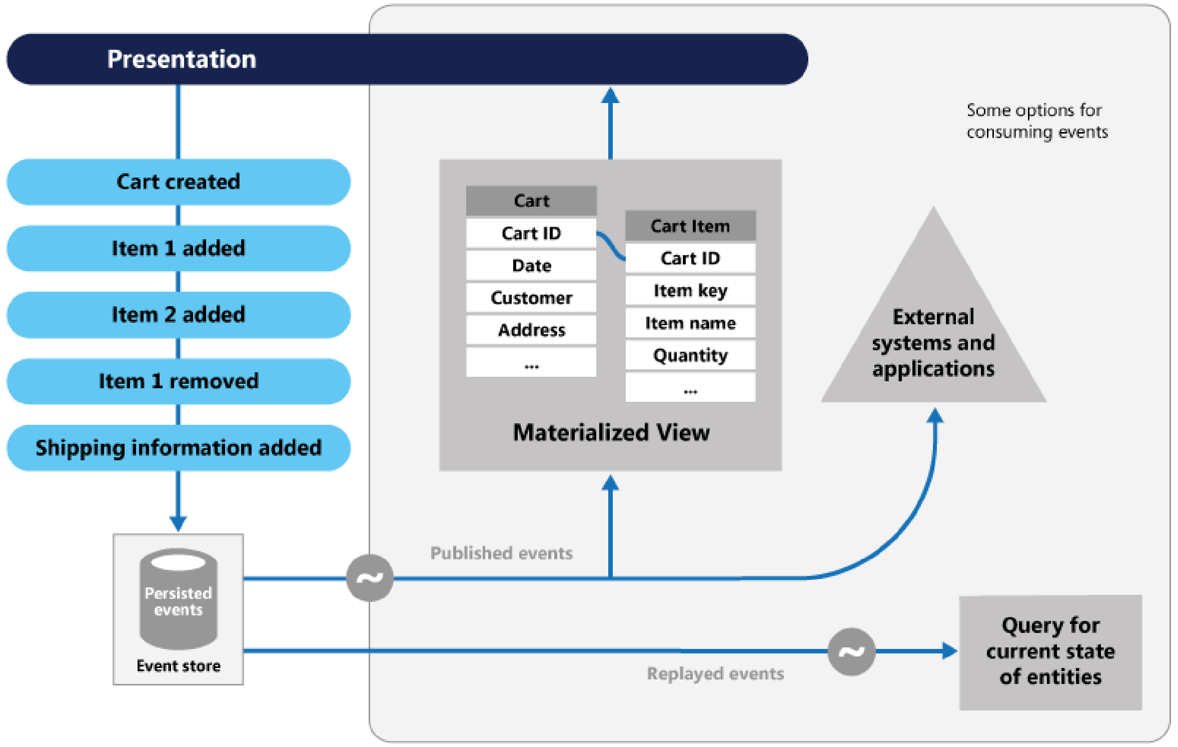
OrderId	ItemId	Qty
1	30	2
1	31	3
2	30	2

ItemId	Name	Total
30	Shirts	4
31	Pants	3

- 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.
- 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.

[Materialized View pattern - Cloud Design Patterns | Microsoft Docs]

Event Sourcing



- Approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store.
- Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they're persisted.
- Each event represents a set of changes to the data (such as AddedItemToOrder).

[Event Sourcing pattern - Cloud Design Patterns | Microsoft Docs]

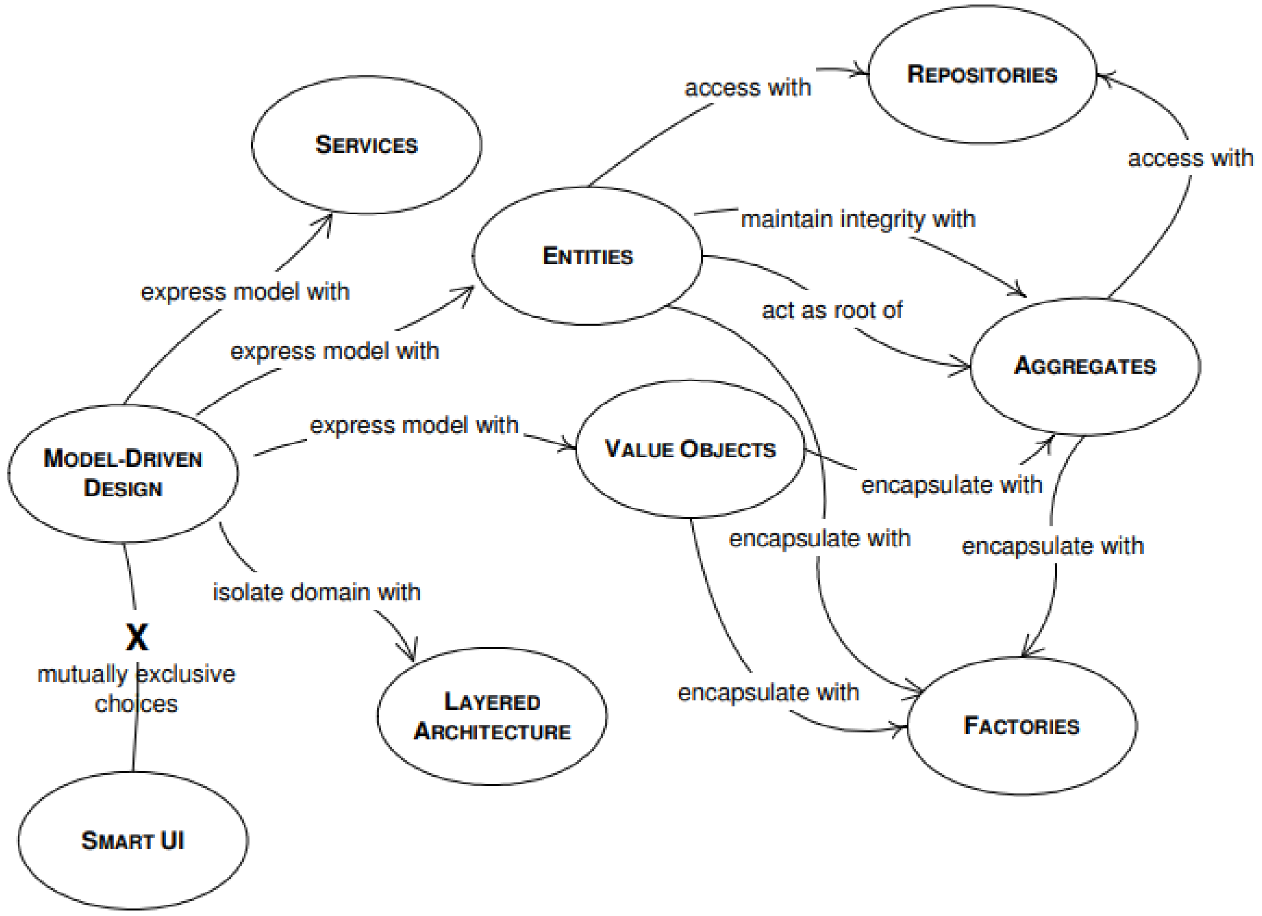
Microservices

Principles of Microservices

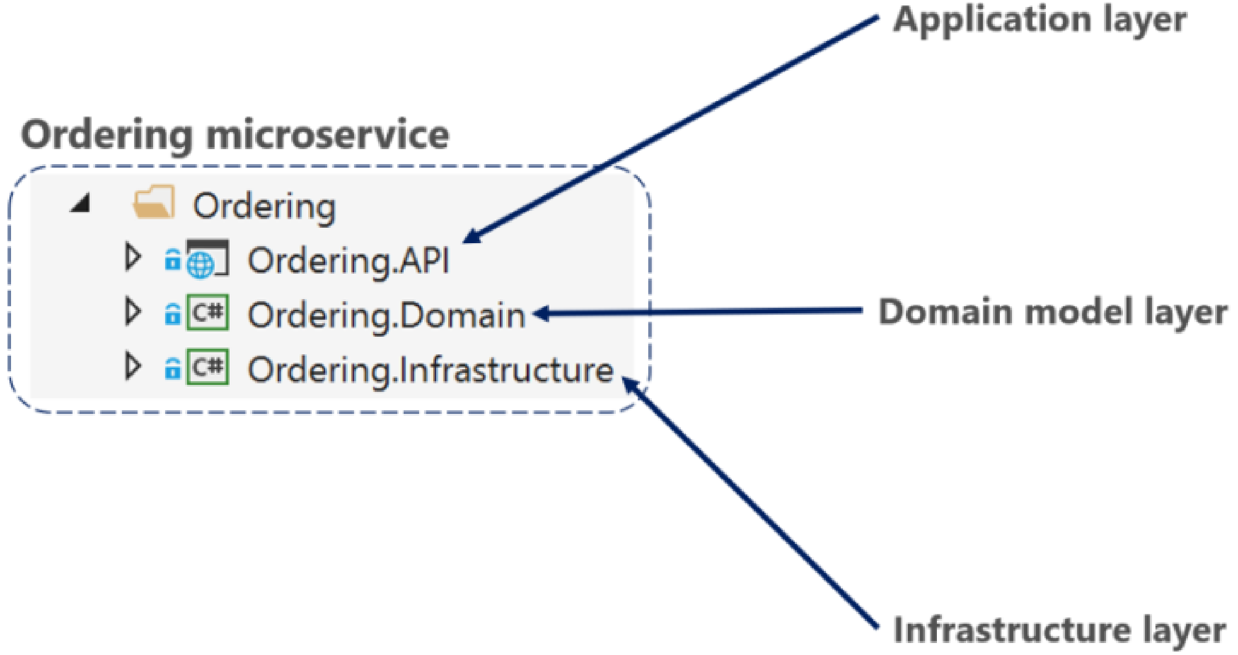
- Architectural Style, not a pattern by itself
- Every MicroService follows the 10 principles/12-Factor App
- Modeled around business domain
- Decentralized, Autonomous - developed, built and deployed independently - could use its own tech stack, DB Choices, etc.
- Availability valued more than consistency
- Event-Driven
- Loose coupling
- Single Responsibility & Interface segregation
- Observable - Monitoring, Health Checks, Alerts

Domain Driven Design

- Advocates modeling based on the reality of business as relevant to your use cases. In the context of building applications, DDD talks about problems as domains.
- It describes independent problem areas as Bounded Contexts (each Bounded Context correlates to a microservice) and emphasizes a common language to talk about these problems (ubiquitous language).
- It also suggests many technical concepts and patterns, like domain entities with rich models (no anemic-domain model), value objects, aggregates, and aggregate root (or root entity) rules to support the internal implementation



Layers in a Domain-Driven Design Microservice



Application layer

- ASP.NET Web API
- Network access to microservice
- API contracts/implementation
- Commands and command handlers
- Queries (when using an CQS approach)
  - Micro ORMs like Dapper

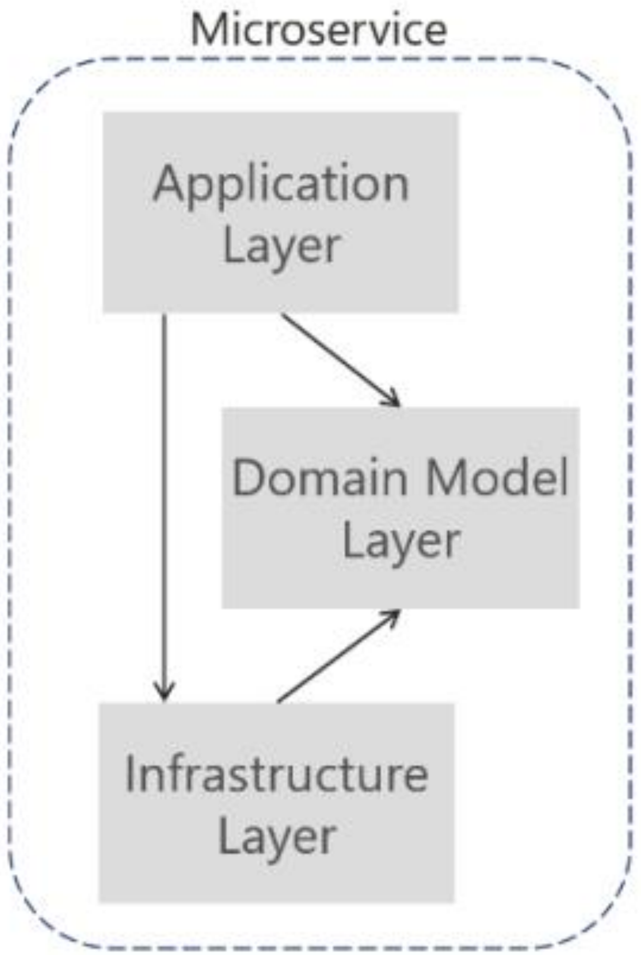
Domain model layer

- Domain entity model
- POCO entity classes (clean C# code)
- Domain entities with data + behavior
- DDD patterns:
  - Domain entity, aggregate
  - Aggregate root, value object
  - Repository contracts/interfaces

Infrastructure layer

- Data persistence infrastructure
  - Repository implementation
- Use of ORMs or data access API:
  - Entity Framework Core or any ORM
  - ADO.NET
  - Any NoSQL database API
- Other infrastructure implementation used from the application layer
  - Logging, cryptography, search engine, etc.

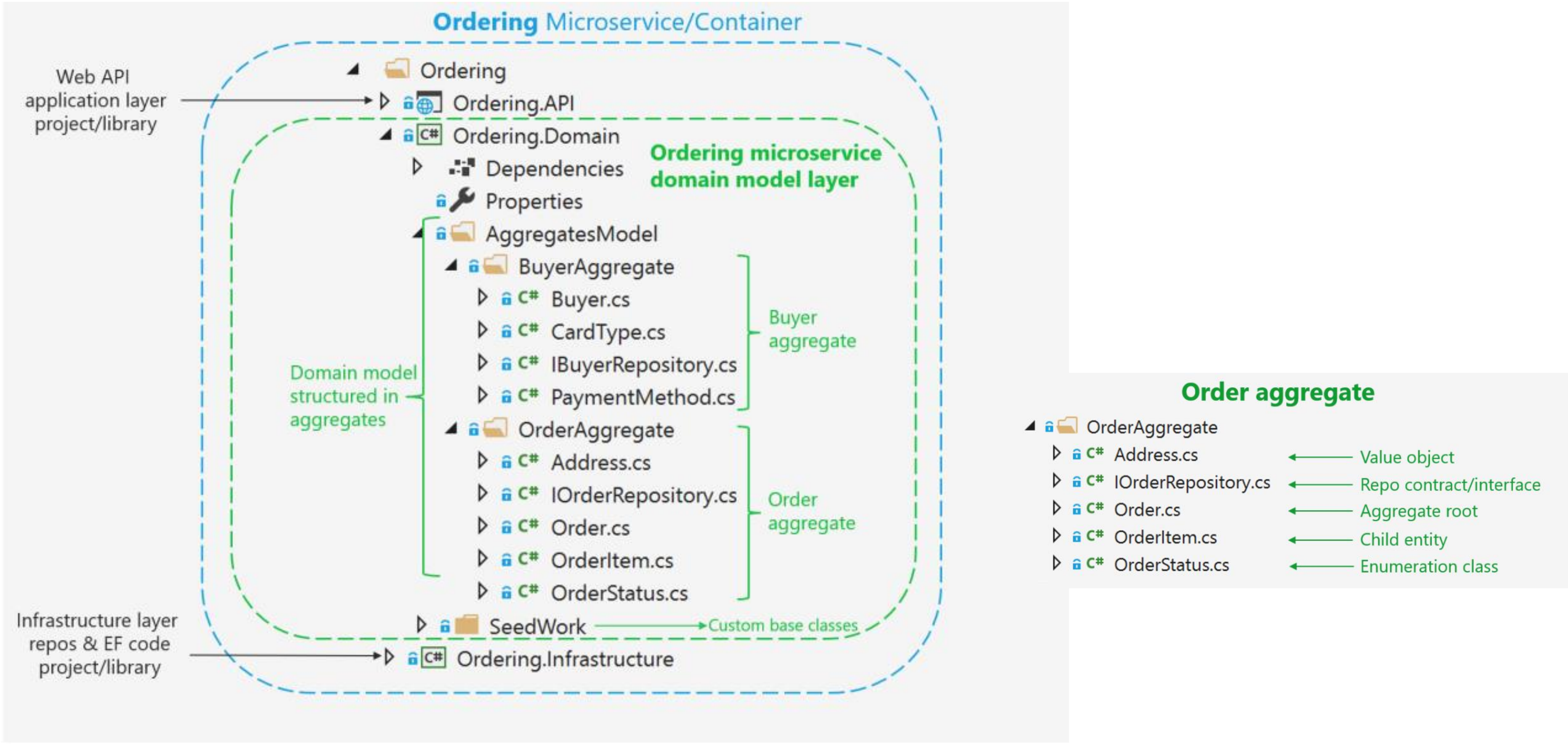
Dependencies between Layers in a Domain-Driven Design service



- Depends on the Domain-Model Layer so it can:
  - Use entity objects
  - Use Repository Interfaces/Contracts
- Depends on the Infrastructure Layer (thru DI) so it can:
  - Use Repository implementation classes, ideally through DI

- Ideally, it must NOT take dependency on any other layer
- It implements:
  - Domain Entities, Aggregate-Roots and Value-Objects
  - Repository Contracts/Interfaces (to be used in DI)

- Depends on the Domain-Model Layer so it can:
  - Use entity objects.
    - Like EF updating a database through mapped entities
- Direct dependency on infrastructure frameworks like EF Core or any other database, cache or infrastructure API

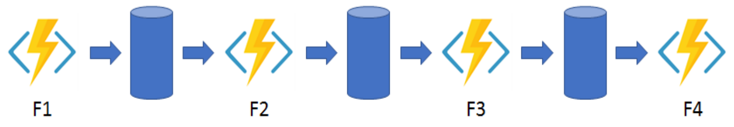

Ordering Microservice/Container
Web API application layer project/library
Ordering
Ordering.API
Ordering.Domain
Dependencies
Properties
Ordering microservice domain model layer
AggregatesModel
BuyerAggregate
Buyer.cs
CardType.cs
IBuyerRepository.cs
PaymentMethod.cs
Buyer aggregate
Domain model structured in aggregates
OrderAggregate
Address.cs
IOrderRepository.cs
Order.cs
OrderItem.cs
OrderStatus.cs
Order aggregate
SeedWork
Custom base classes
Infrastructure layer repos & EF code project/library
Ordering.Infrastructure
Order aggregate
OrderAggregate
Address.cs
Value object
IOrderRepository.cs
Repo contract/interface
Order.cs
Aggregate root
OrderItem.cs
Child entity
OrderStatus.cs
Enumeration class

Other common patterns

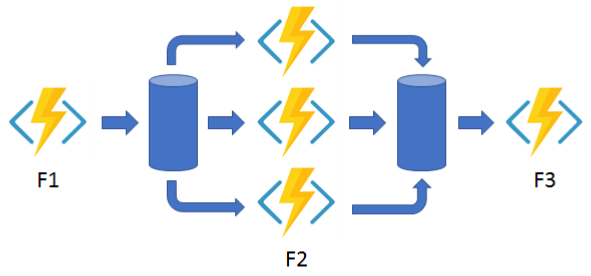
Lab Set 6

Problem Statement	Solution	Alternatives/Related Patterns
You are the architect of a HRM software company that is creating a business process to onboard employees. The process involves multiple steps that can be long running. Manger’s approval should also be factored in.	You use the Orchestration Pattern (Type of Saga) to model the process using Azure Durable Functions. You leverage Durable entities to persist state in a long running process	

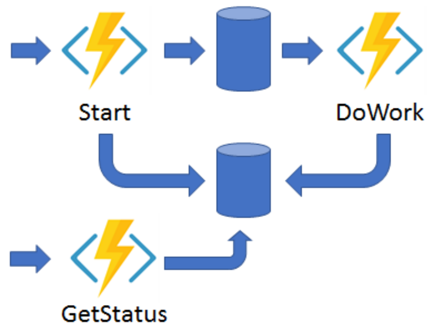
Durable Functions - Patterns



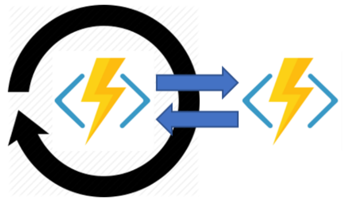
Pattern #1: Function chaining



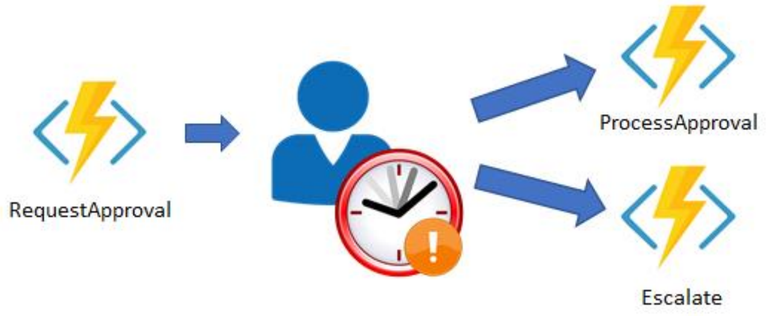
Pattern #2: Fan out/fan in



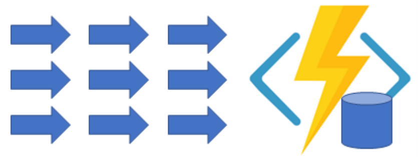
Pattern #3: Async HTTP APIs



Pattern #4: Monitor



Pattern #5: Human interaction



Pattern #6: Aggregator (stateful entities)

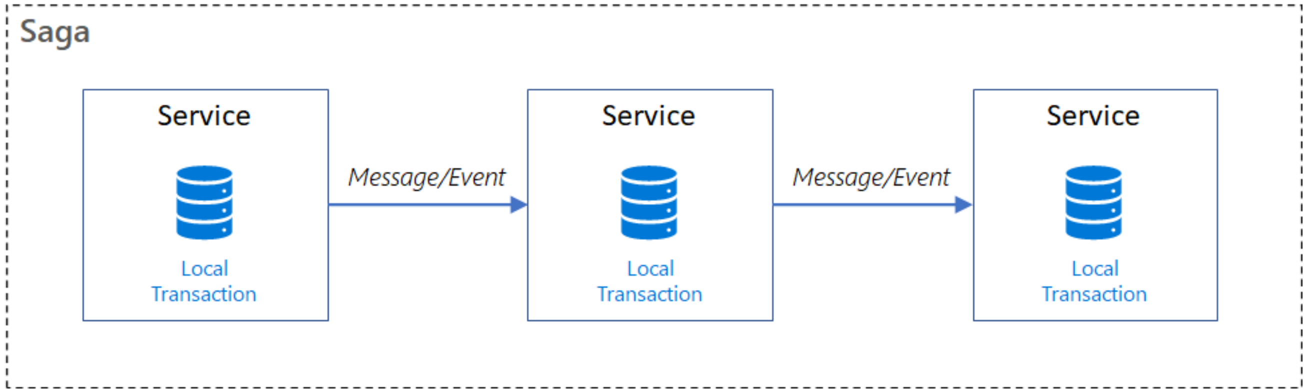
[Durable Functions Overview - Azure | Microsoft Docs]()

Saga

- The saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios.
- A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step.
- If a step fails, the saga executes compensating transactions that counteract the preceding transactions.

Solution

- The saga pattern provides transaction management using a sequence of local transactions - a local transaction is the atomic work effort performed by a saga participant.
- Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.
- If a local transaction fails, the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



Key Considerations:

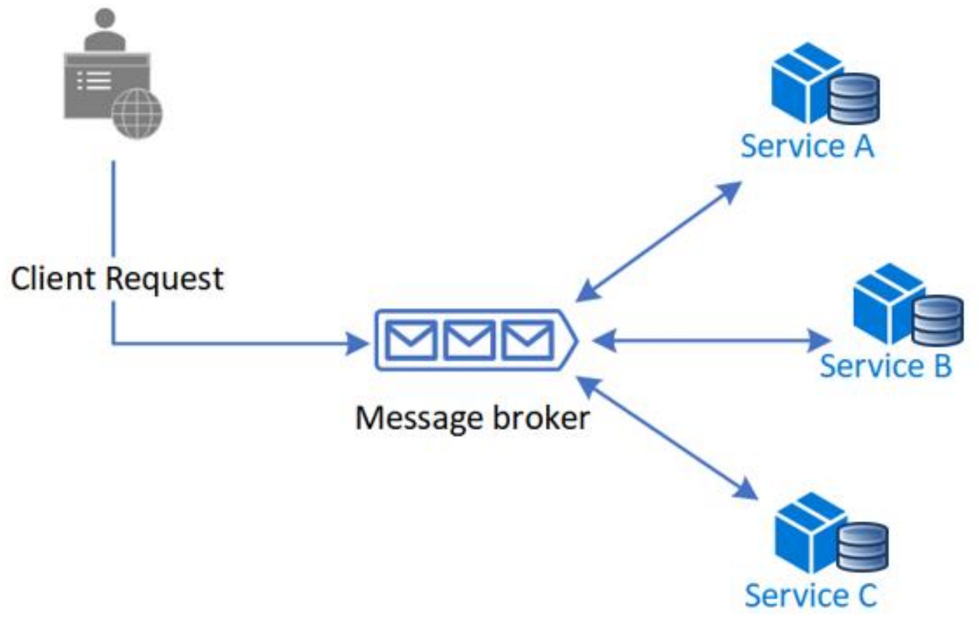
1. Implement Observability
2. The implementation must be capable of handling a set of potential transient failures and provide idempotence for reducing side-effects and ensuring data consistency
3. Implement countermeasures to reduce durability/data anomalies

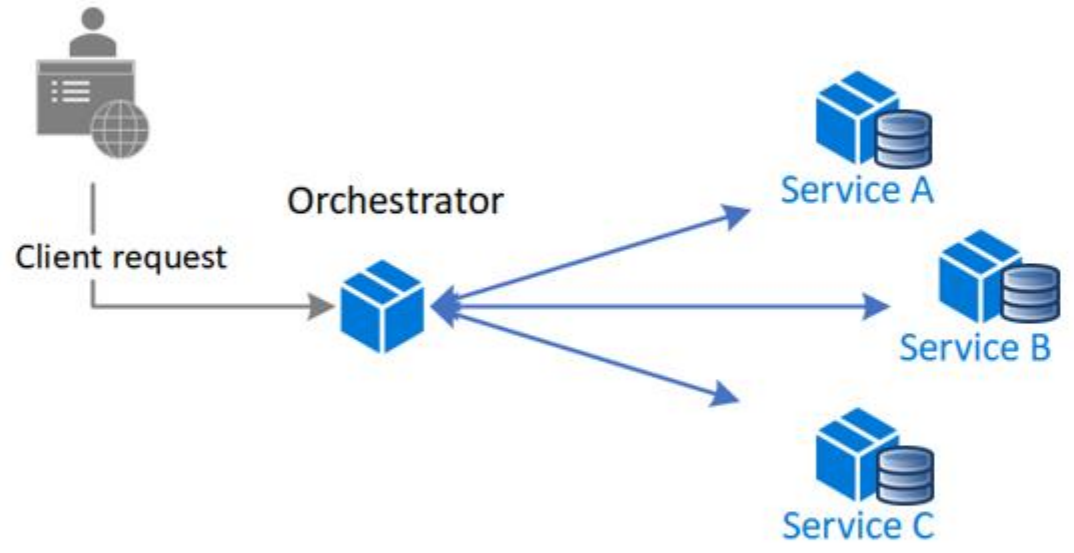
[Saga distributed transactions - Azure Design Patterns | Microsoft Docs]

Two patterns to implement Saga

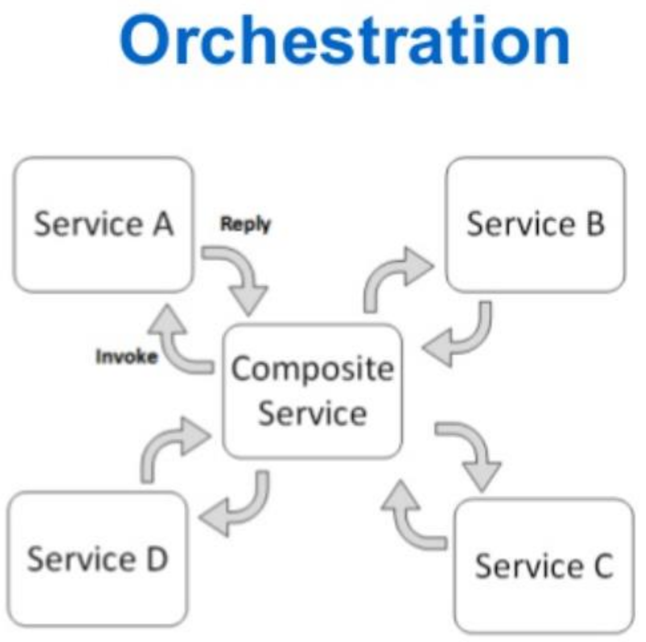
- **Choreography** is a way to coordinate sagas where participants exchange events without a centralized point of control.
- With choreography, each local transaction publishes domain events that trigger local transactions in other services.

- **Orchestration** is a way to coordinate sagas where a centralized controller tells the saga participants what local transactions to execute.
- The saga orchestrator handles all the transactions and tells the participants which operation to perform based on events.
- The orchestrator executes saga requests, stores and interprets the states of each task, and handles failure recovery with compensating transactions.

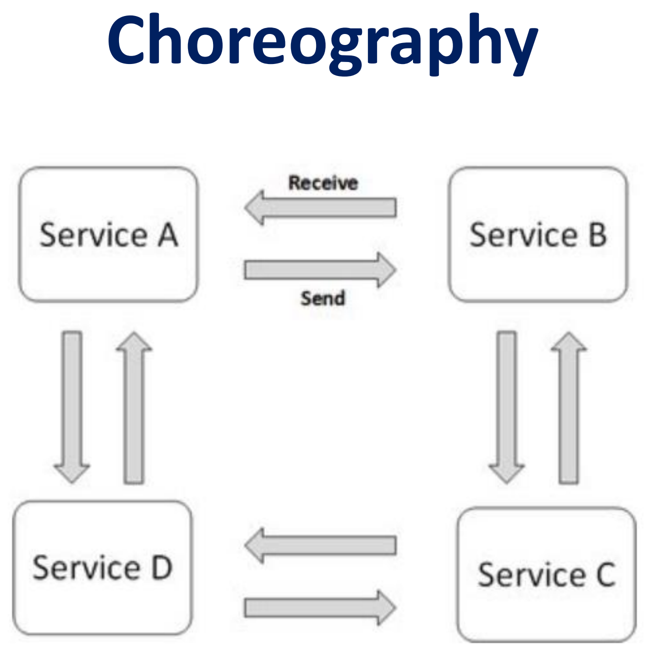




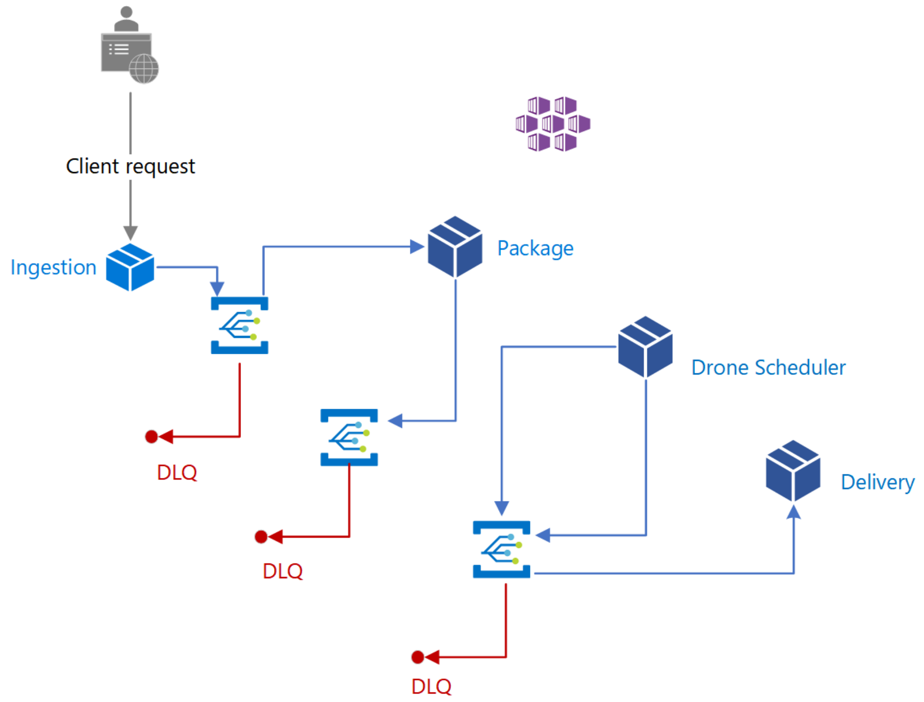
Orchestration v/s Choreography



Imperative

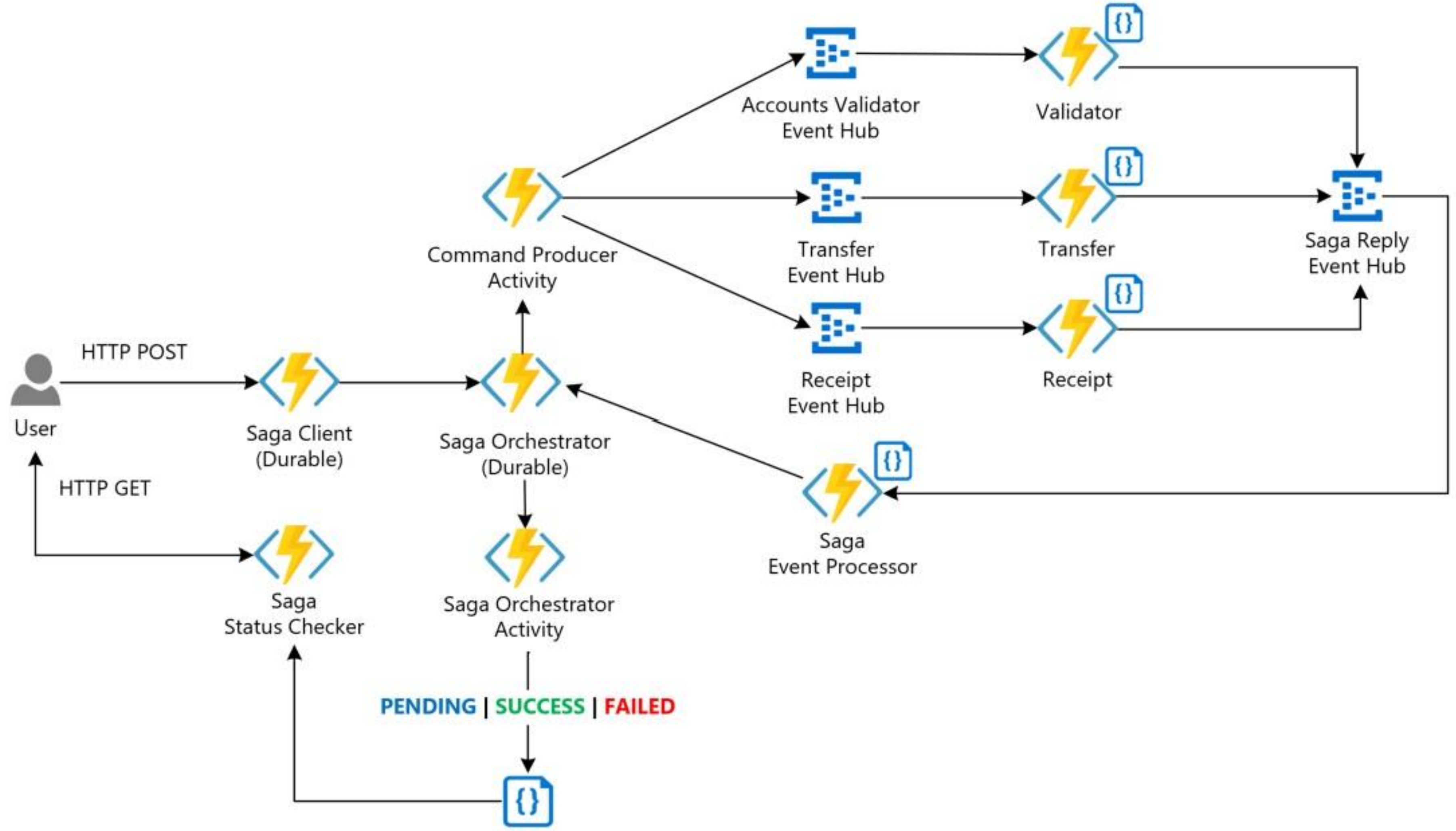


Declarative



[Choreography - Cloud Design Patterns | Microsoft Docs]()

Saga Reference Implementation for Azure



[Azure-Samples/saga-orchestration-serverless: An orchestration-based saga implementation reference in a serverless architecture (github.com)]

Sharding

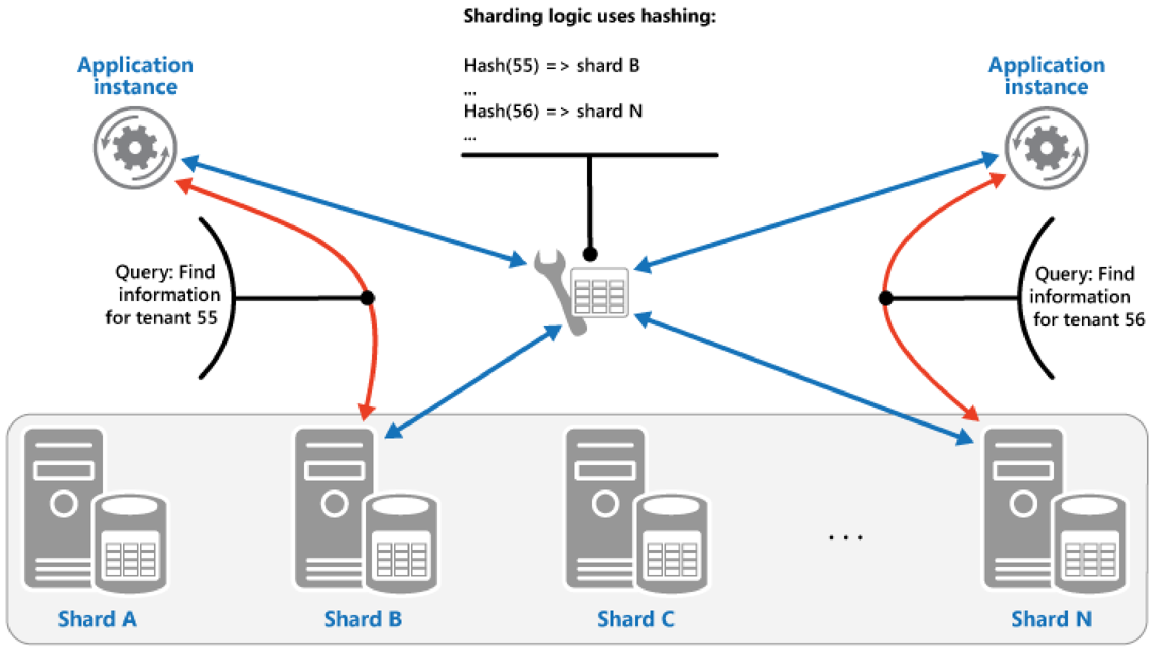
Divide the data store into horizontal partitions or shards. Each shard has the same schema, but holds its own distinct subset of the data.

Sharding Strategies

1) **Lookup**: routes a request for data to the shard that contains that data by using the shard key
2) **Range:** This strategy groups related items together in the same shard
3) **Hash:** The sharding logic computes the shard in which to store an item based on a hash of one or more attributes of the data

Key Considerations:

1. Keep Shards balanced, reduce hotspots
2. Monitor for data skew
3. Use stable data for Shard key (avoid data movement)
4. For many applications, creating a larger number of small shards can be more efficient than having a small number of large shards because they can offer increased opportunities for load balancing



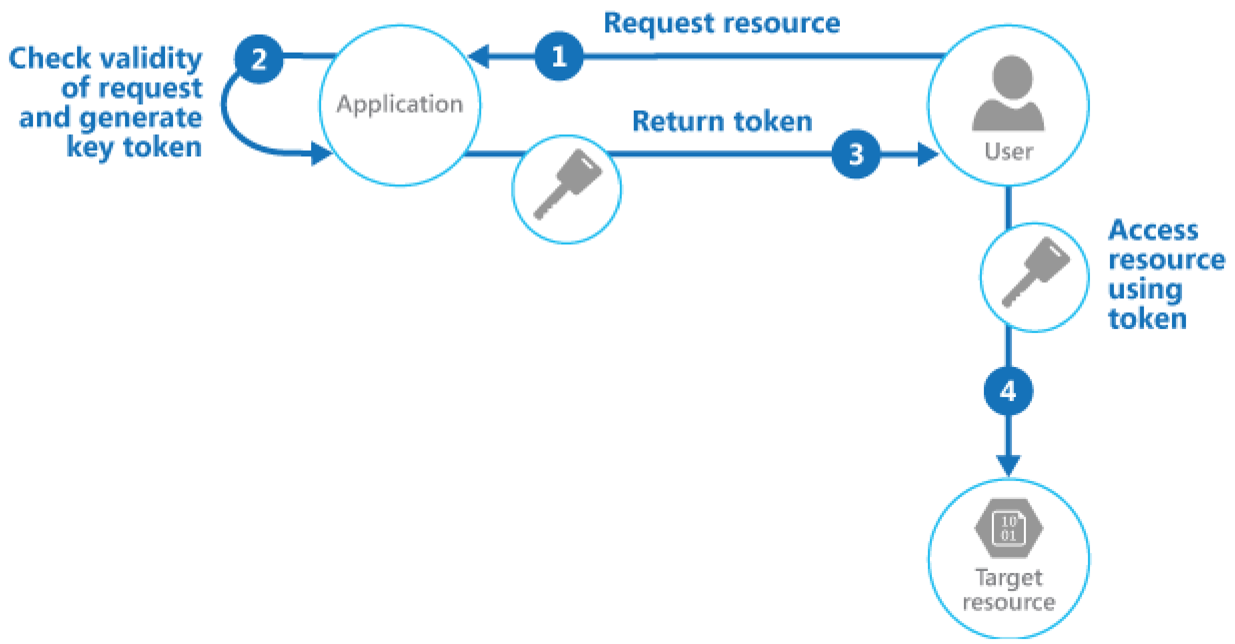
[Sharding Pattern | Microsoft Docs](#)

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.

Key Considerations:

1. Manage the validity status and period of the key.
2. Control the level of access the key will provide
3. Validate, and optionally sanitize, all uploaded data
4. Consider how to control users' behavior.
5. Audit all operations
6. Deliver the key securely
7. Protect sensitive data in transit

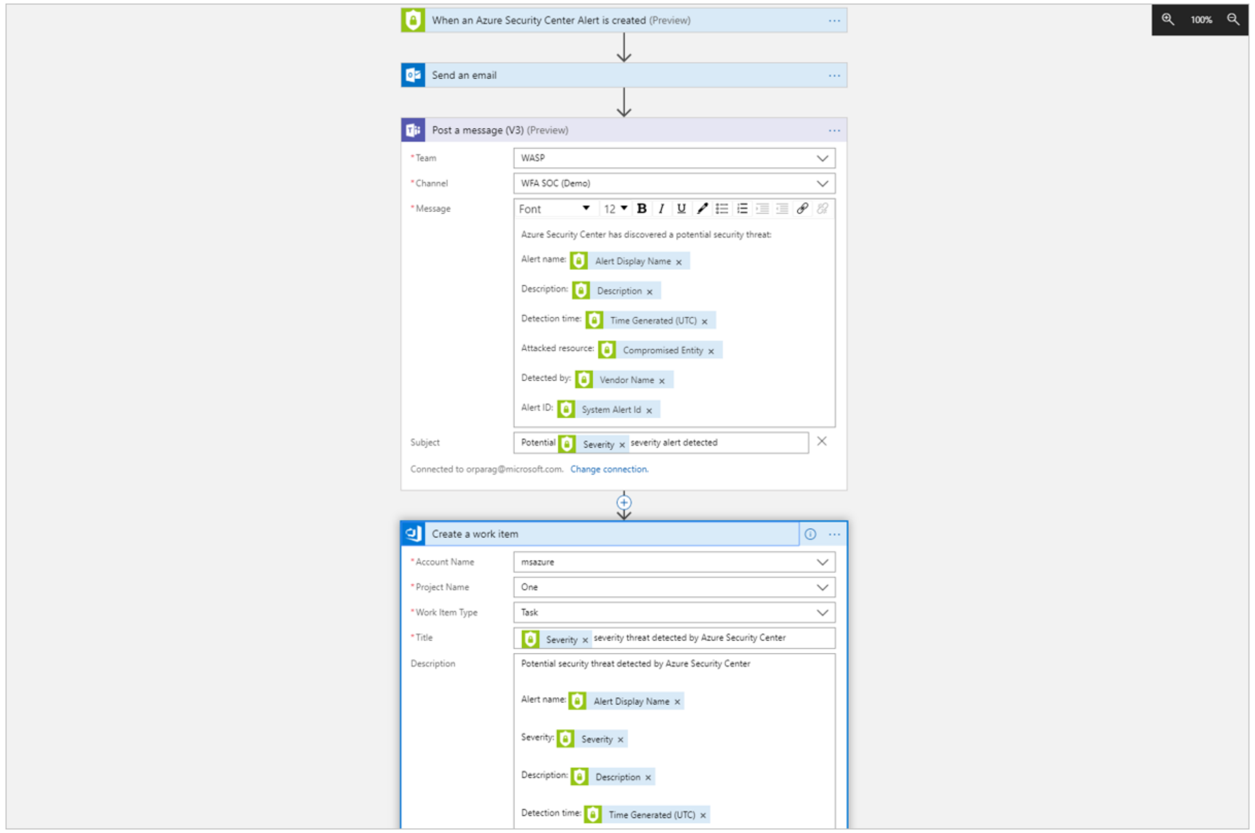


Request resource
1
Check validity of request and generate key token
2
Application
Return token
3
User
Access resource using token
4
Target resource

Valet Key Pattern | Microsoft Docs

Serverless Business Process Automation

- Use Logic Apps (serverless) to automate business processes triggered by events
- Example: Incident Response from Security Center


When an Azure Security Center Alert is created (Preview)
Send an email
Post a message (V3) (Preview)
Create a work item
100%

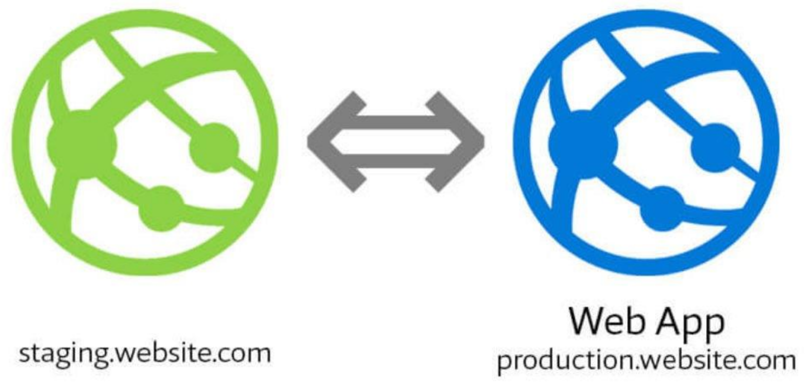
*Team: WASP
*Channel: WFA SOC (Demo)
*Message:
Azure Security Center has discovered a potential security threat:
Alert name: Alert Display Name
Description: Description
Detection time: Time Generated (UTC)
Attacked resource: Compromised Entity
Detected by: Vendor Name
Alert ID: System Alert Id
Subject: Potential Severity severity alert detected
Connected to orparag@microsoft.com. Change connection.

*Account Name: msazure
*Project Name: One
*Work Item Type: Task
*Title: Severity severity threat detected by Azure Security Center
Description:
Potential security threat detected by Azure Security Center
Alert name: Alert Display Name
Severity: Severity
Description: Description
Detection time: Time Generated (UTC)

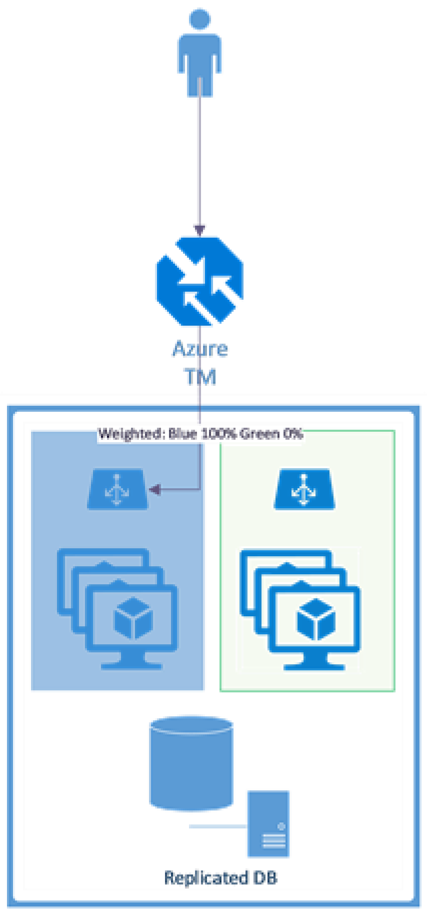
[Workflow automation in Azure Security Center | Microsoft Docs]()

Deployment patterns – Blue Green

- Blue-Green deployment is a type of deployment that reduces downtime and risk by running two identical production environments known as Blue and Green.
- At any time, only one of the environments is live, with the live environment serving all production traffic.
- Switch from blue to green for cutting over to a new version. Switch back if case of issues



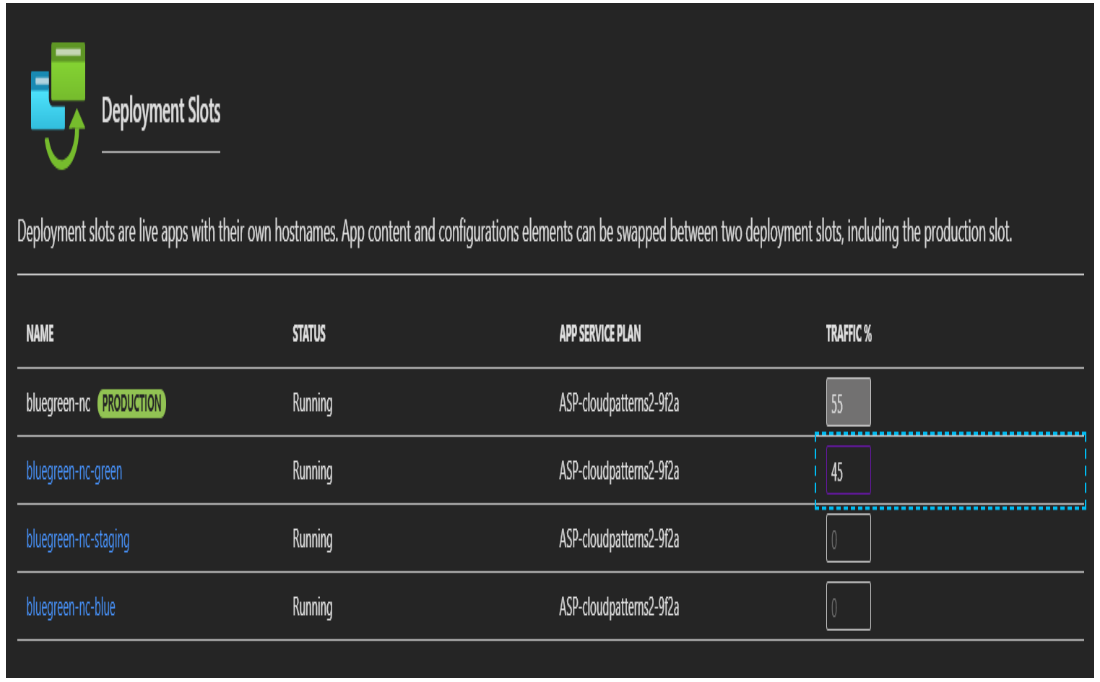
Azure App Services



Azure Traffic Manager

Deployment patterns – A/B Testing

- Route a % of traffic to the A site and the rest to the B site

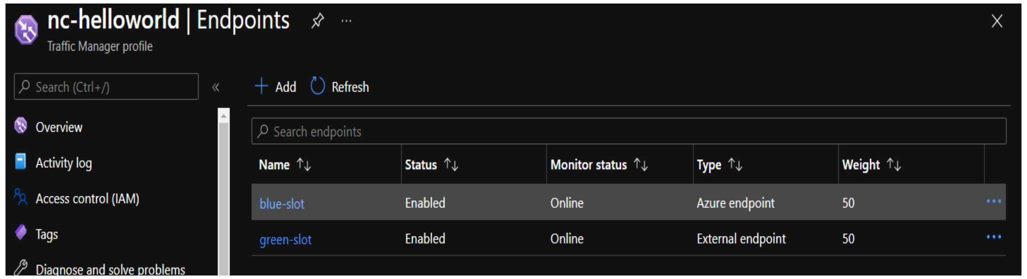


Deployment Slots

Deployment slots are live apps with their own hostnames. App content and configurations elements can be swapped between two deployment slots, including the production slot.

NAME	STATUS	APP SERVICE PLAN	TRAFFIC %
bluegreen-nc PRODUCTION	Running	ASP-cloudpatterns2-9f2a	55
bluegreen-nc-green	Running	ASP-cloudpatterns2-9f2a	45
bluegreen-nc-staging	Running	ASP-cloudpatterns2-9f2a	0
bluegreen-nc-blue	Running	ASP-cloudpatterns2-9f2a	0

Azure App Services



nc-helloworld | Endpoints

Traffic Manager profile

Name	Status	Monitor status	Type	Weight
blue-slot	Enabled	Online	Azure endpoint	50
green-slot	Enabled	Online	External endpoint	50

Azure Traffic Manager

References

- Azure Well Architected Framework
- Pattern Index
- eShop Reference Implementation
- GitHub Repo (Samples)