

# 11 Microservices Patterns

API Gateway Pattern

Circuit Breaker Pattern

Service Registry Pattern

Service Mesh Pattern

Bulkhead Pattern

Event-Driven Pattern

Strangler Pattern

Sidecar Pattern

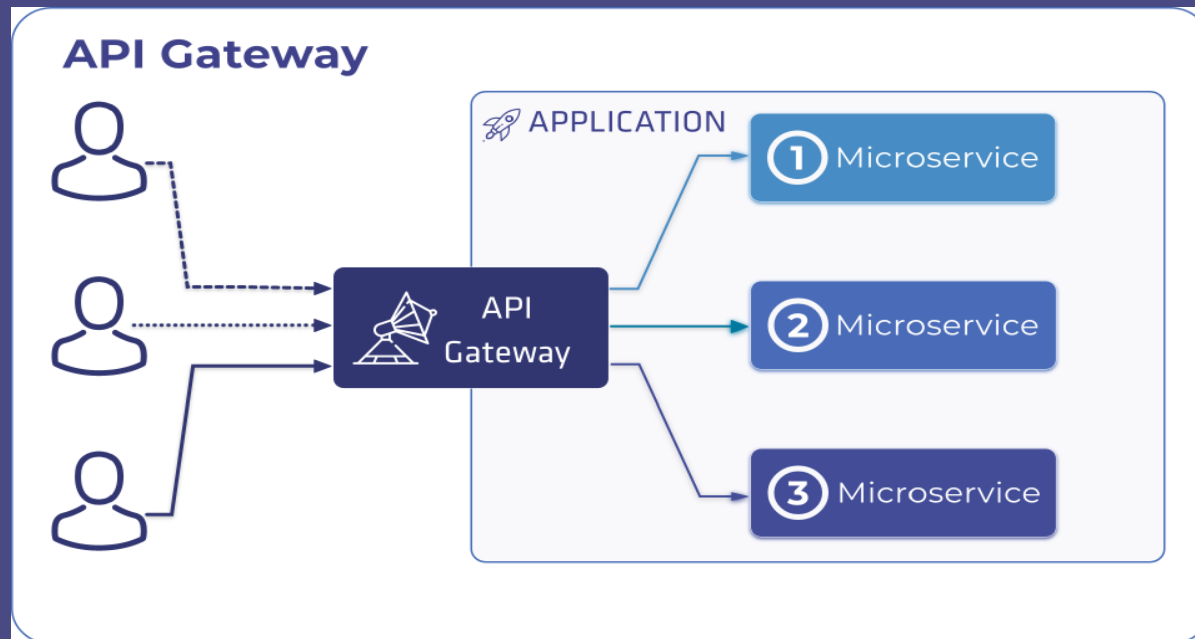
CQRS Pattern

Saga Pattern

Database per Service Pattern

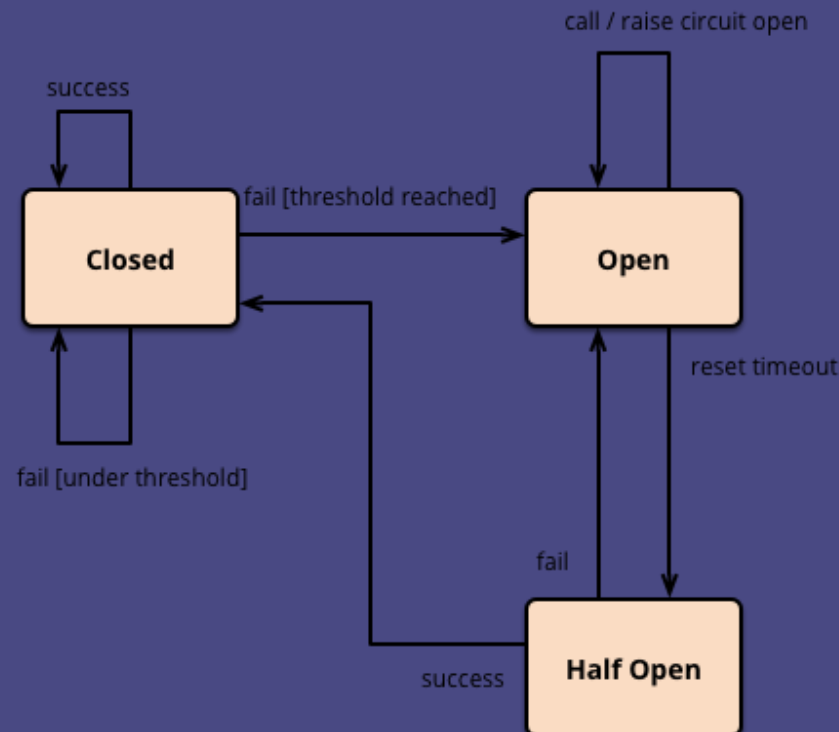
# API Gateway Pattern

This pattern provides a **single entry point** to a microservices based system. It acts as a reverse proxy and routes incoming requests to the appropriate microservice. It can also perform authentication, rate limiting, and other security-related tasks.



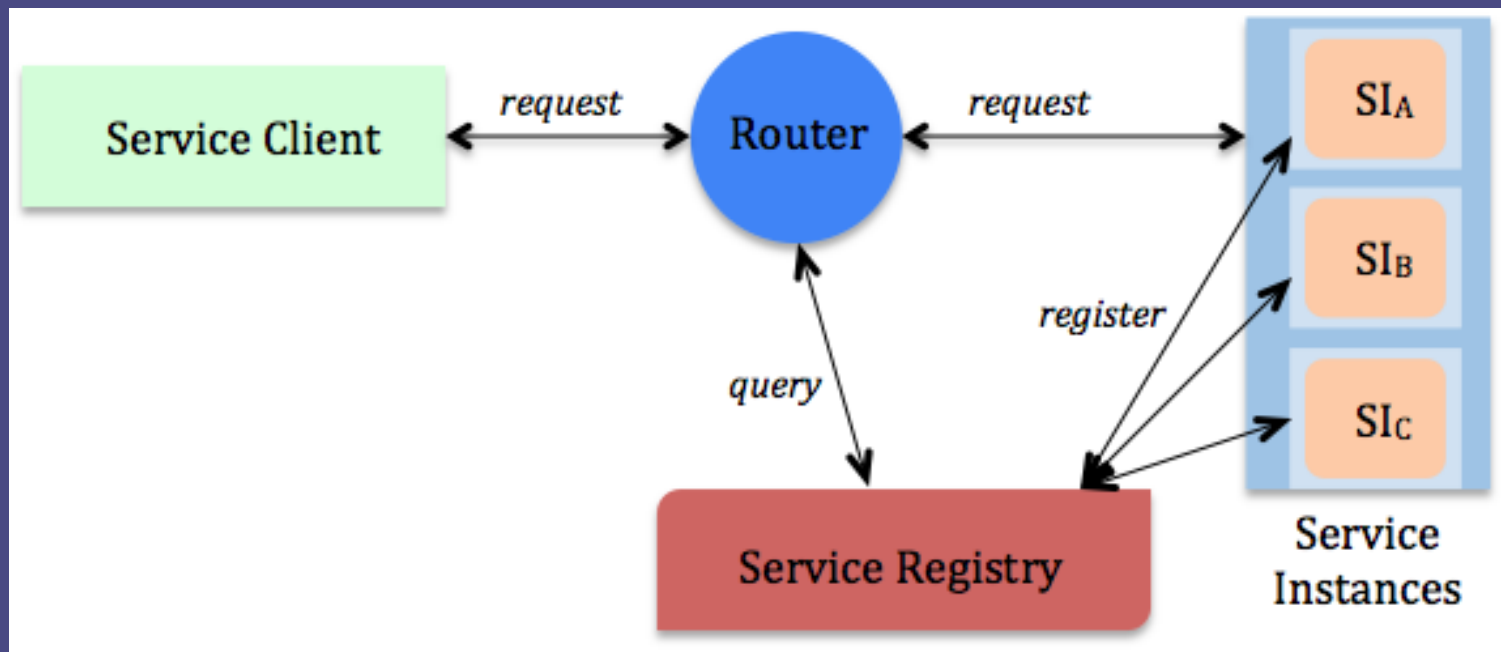
# Circuit Breaker Pattern

This pattern is used to **prevent cascading failures** in a distributed system. It monitors the availability of a service and, if it detects a failure, it can quickly isolate the problematic service and prevent other services from being affected.



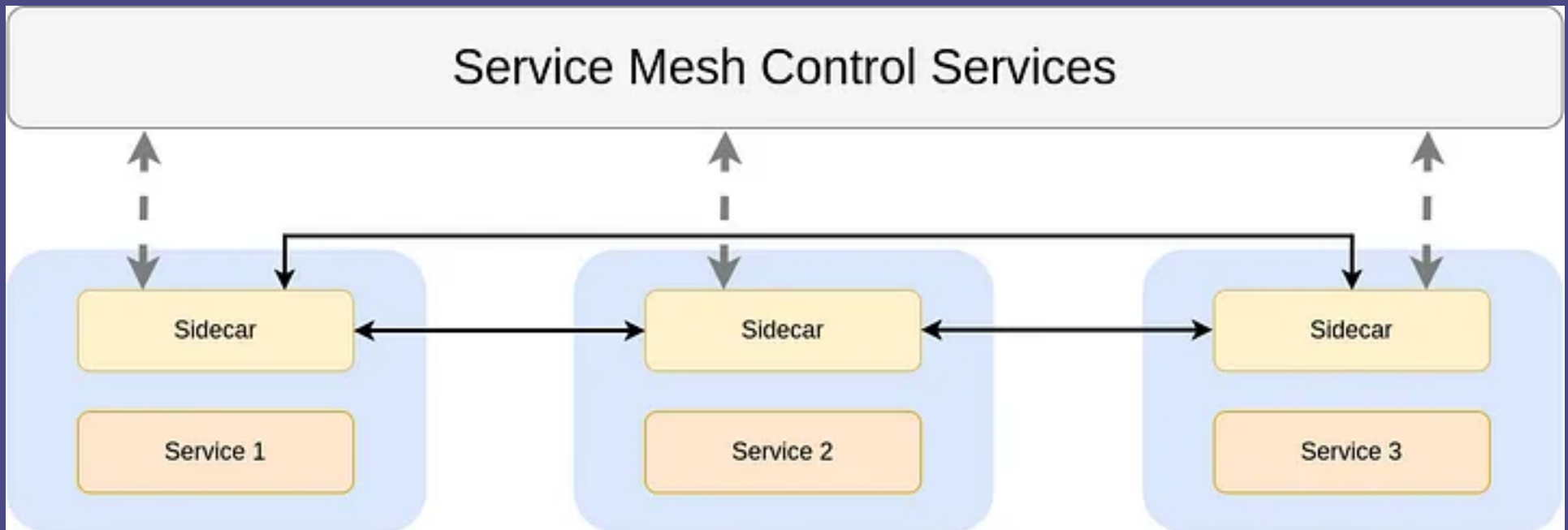
# Service Registry Pattern

This pattern involves managing the **locations of services** in a distributed system. It maintains a list of all available services and their locations, which can be queried by other services to find and communicate with them.



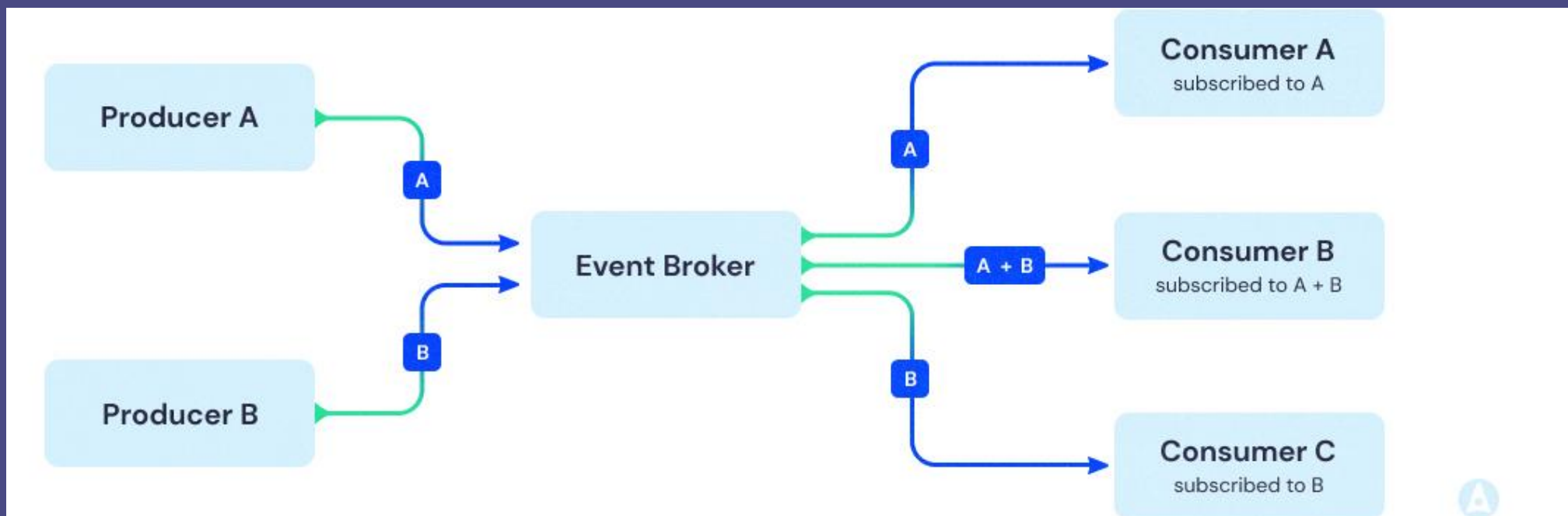
# Service Mesh Pattern

A pattern that involves adding a layer of infrastructure between microservices to handle **cross-cutting concerns** such as service discovery, load balancing and security



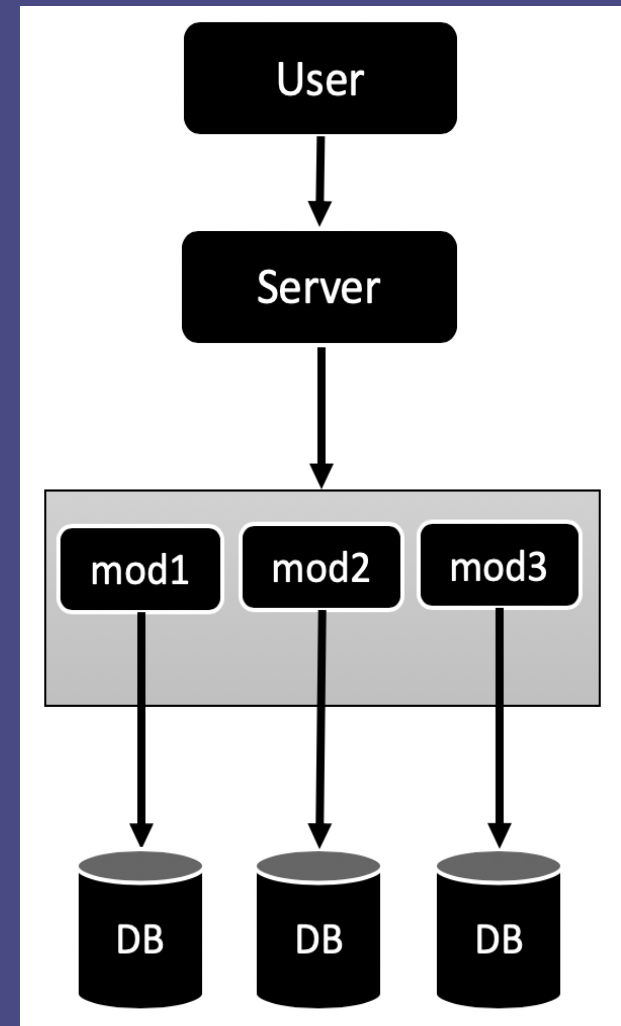
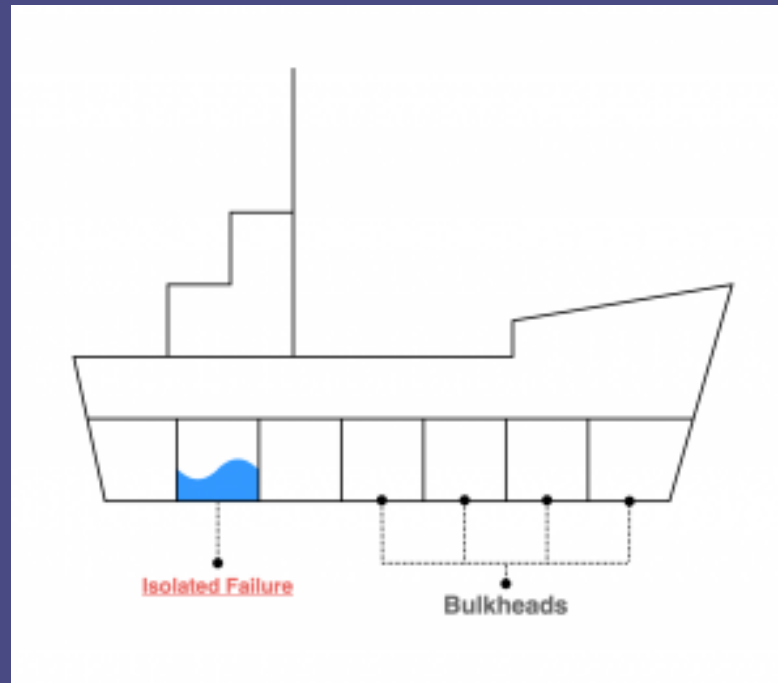
# Event-Driven Pattern

A pattern that involves using **events to communicate** between microservices. Each microservice can push events and subscribe to events published by other microservices



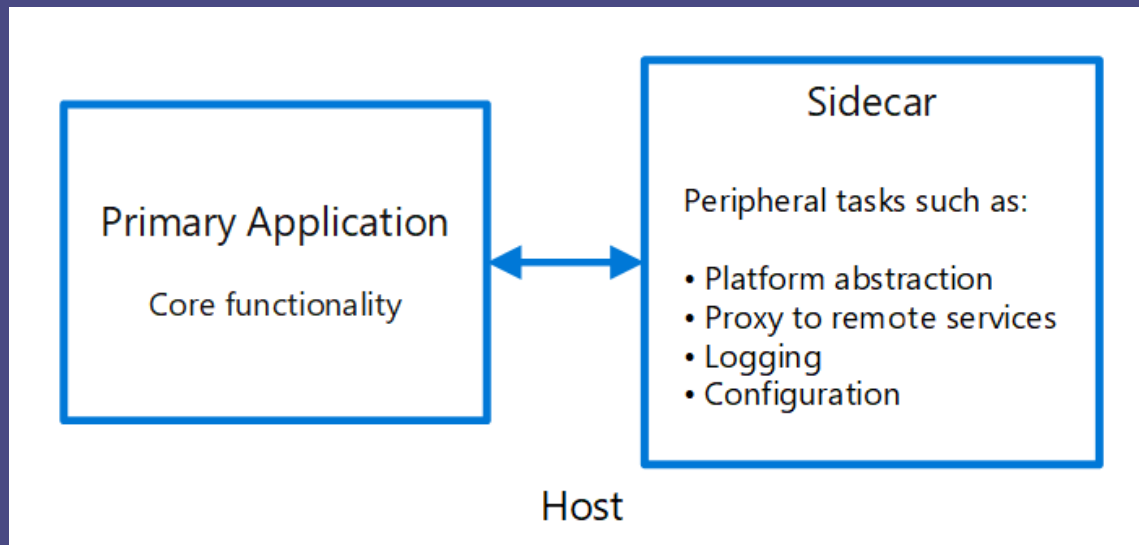
# Bulkhead Pattern

A pattern used to **isolate failures** is a microservices Architecture. Each microservice is placed in a separate container, so if one microservice fails, it does not affect other microservices.



# Sidecar Pattern

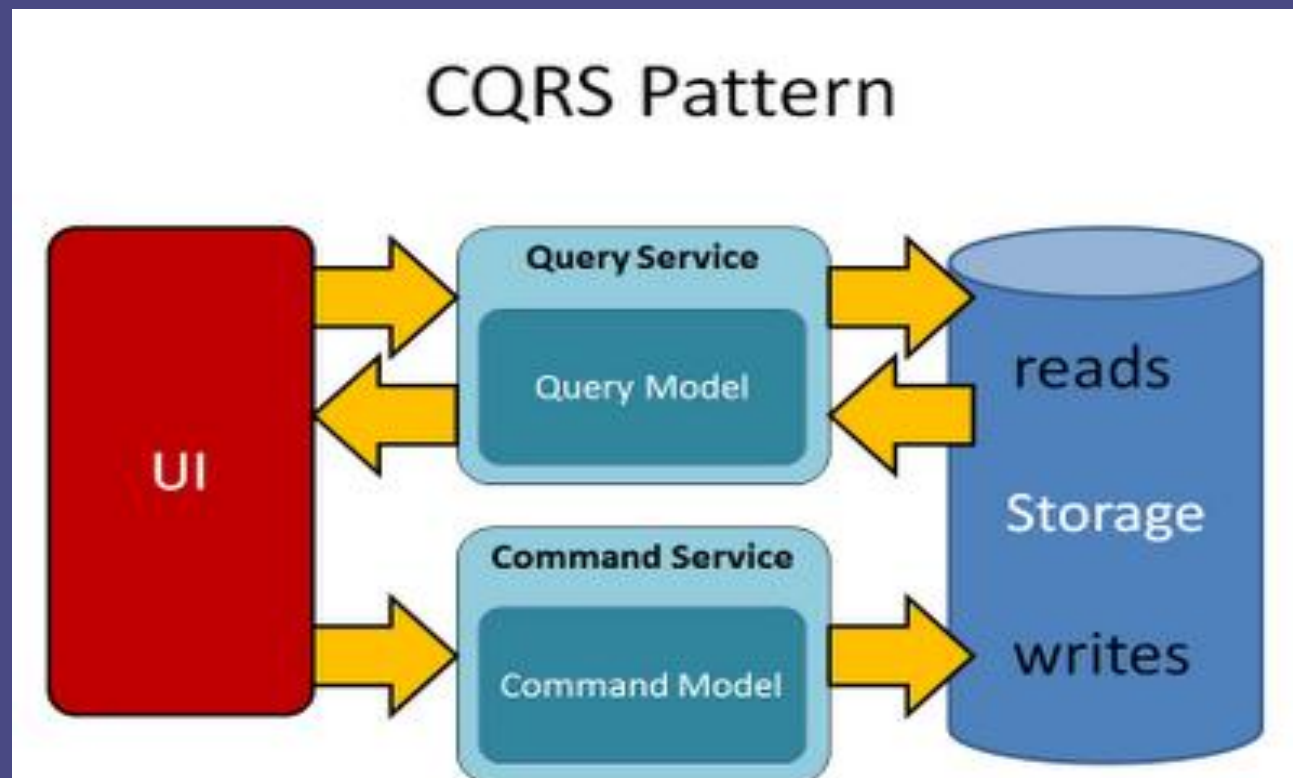
A pattern that involves deploying a **separate container** alongside each microservice to handle cross-cutting concerns such as logging, monitoring and security.





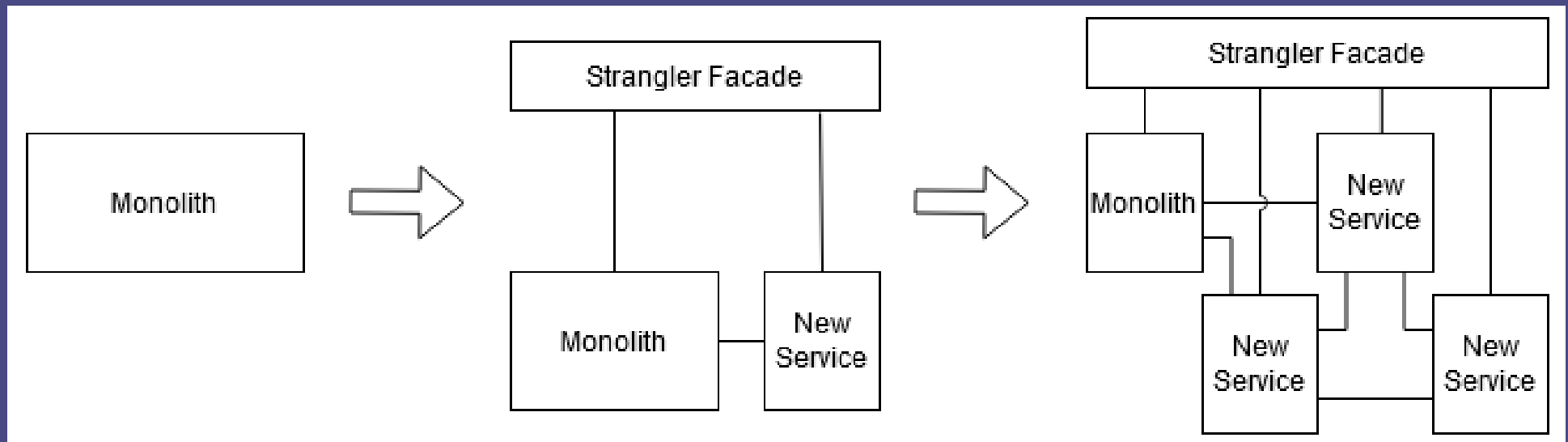
# CQRS Pattern

This pattern **separates the read and writes operations** of a system. It uses separate models for reads and writes, which allows for the optimization and scalability of each. This pattern is particularly useful for systems with high read-and-write workloads.



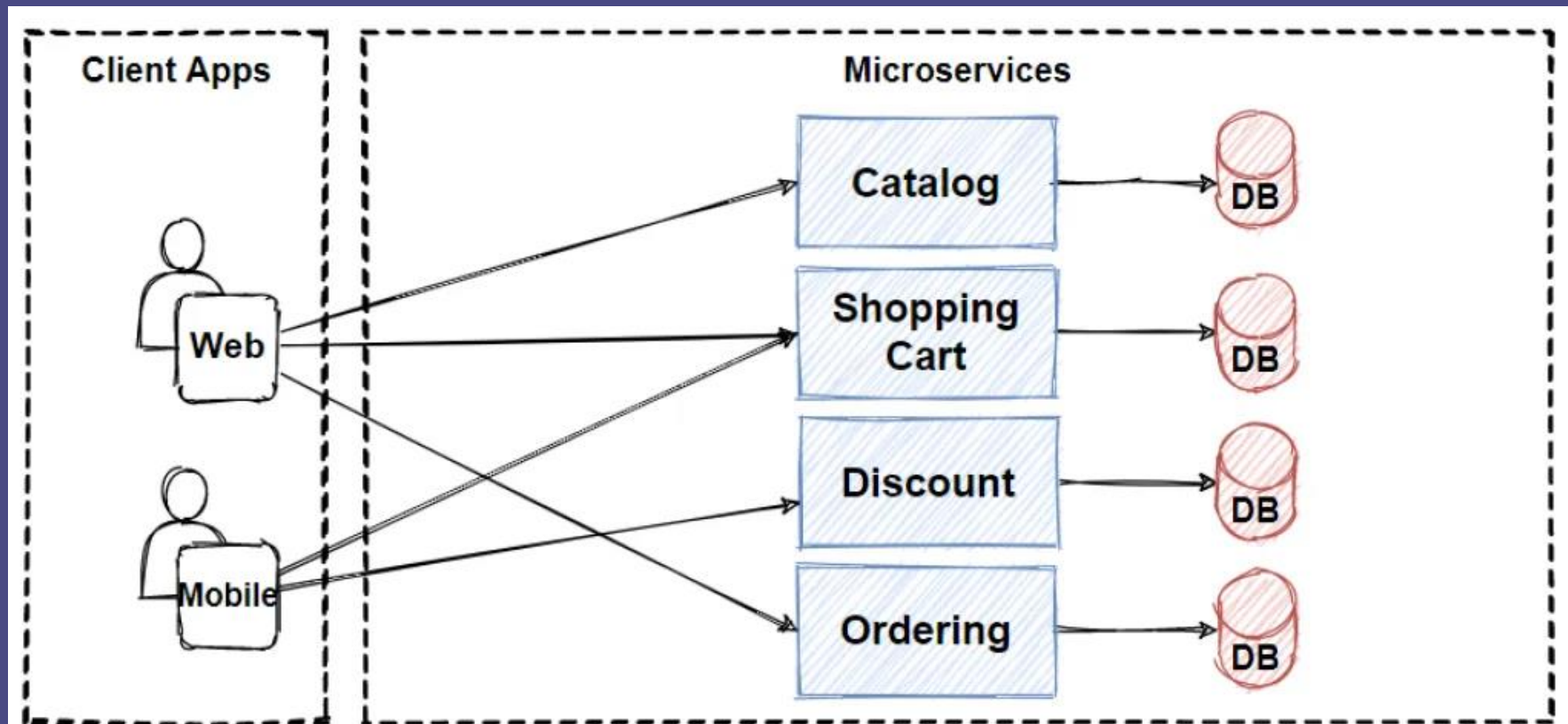
# Strangler Pattern

A pattern that involves **gradually replicating** a monolithic application with microservices by gradually adding new microservices and removing functionality from the monolithic.



# Database per Service Pattern

This pattern involves using a **separate database** for each microservice. This ensures that each microservice has its own data store, which can be optimized for its specific needs. It also helps to prevent coupling between services



# Saga Pattern

This The saga design pattern is provide to manage **data consistency across microservices** in distributed transaction cases. Basically, saga patterns offers to create set of transactions that update microservices sequentially, and publish events to trigger the next transaction for the next microservices.

If one of the step is failed, than saga patterns trigger to rollback transactions which is basically do reverse operations with publishing rollback events to previous microservices.

There are 2 Saga types :

- ✓ Orchestration Saga Pattern
- ✓ Choreography Saga Pattern

Thank you for reading!

If you like this content or similar content, please follow me in bellow channels.

