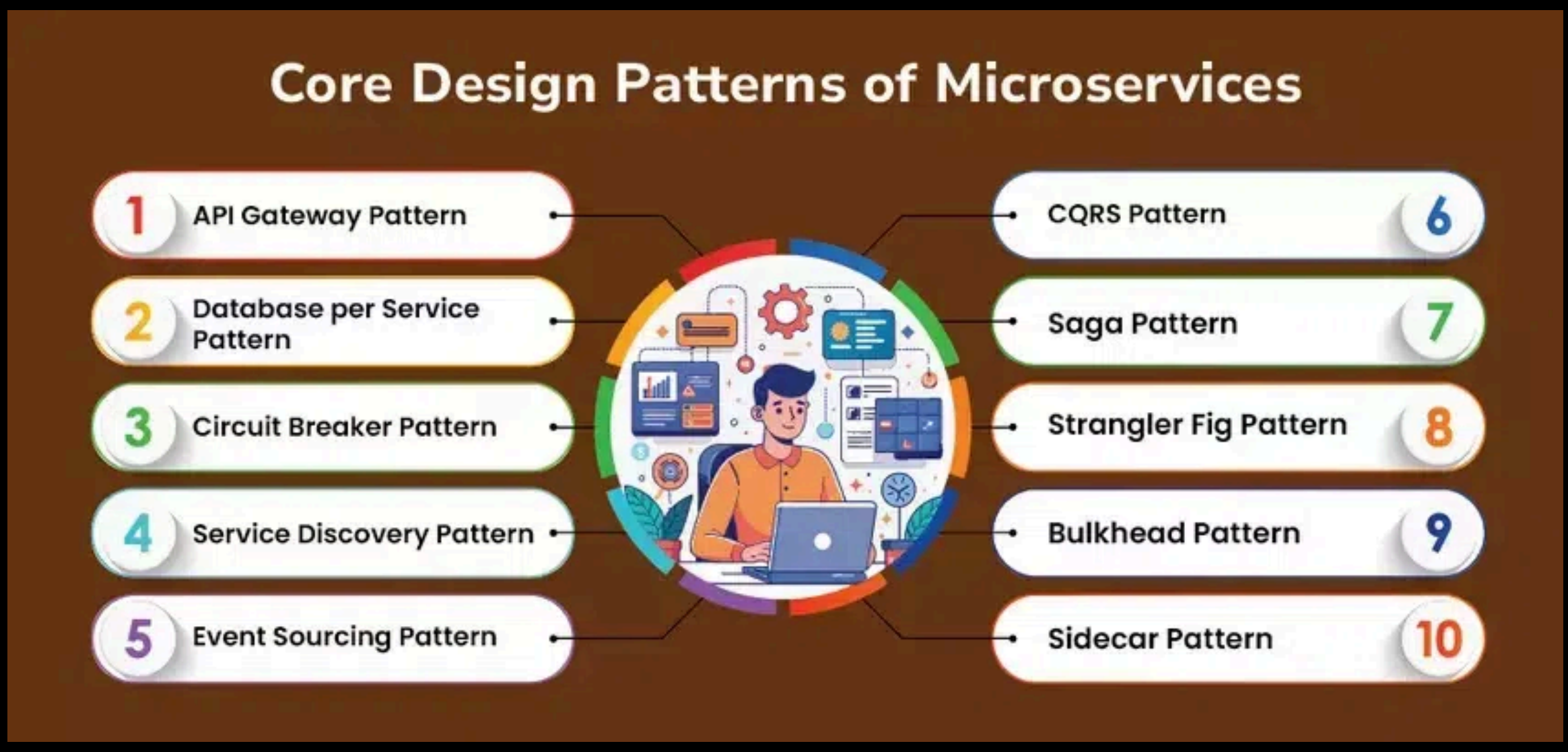
Core Design Patterns of Microservices with Examples

Overview

Here’s an overview of the core design patterns in microservices along with their use cases. These design patterns address various challenges in microservices architecture, promoting scalability, reliability, and maintainability. Each pattern is suitable for specific use cases and can be combined to create a robust microservices-based system

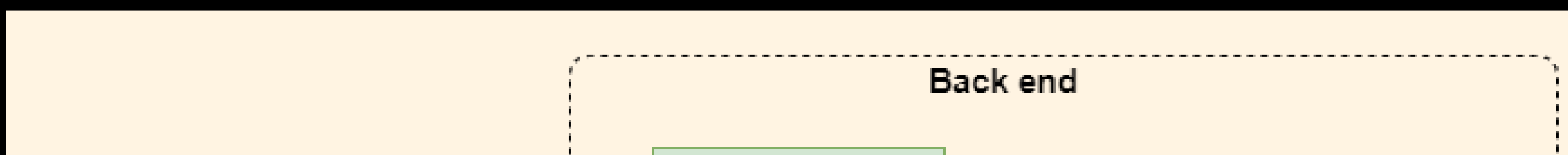


These core patterns are essential for designing resilient and scalable distributed systems.

Choosing the right combination of these patterns depends on your application’s specific needs.

API Gateway Pattern

An API Gateway acts as a single entry point for all clients, routing requests to the appropriate microservices. It can handle cross-cutting concerns such as authentication, logging, rate limiting, and load balancing.

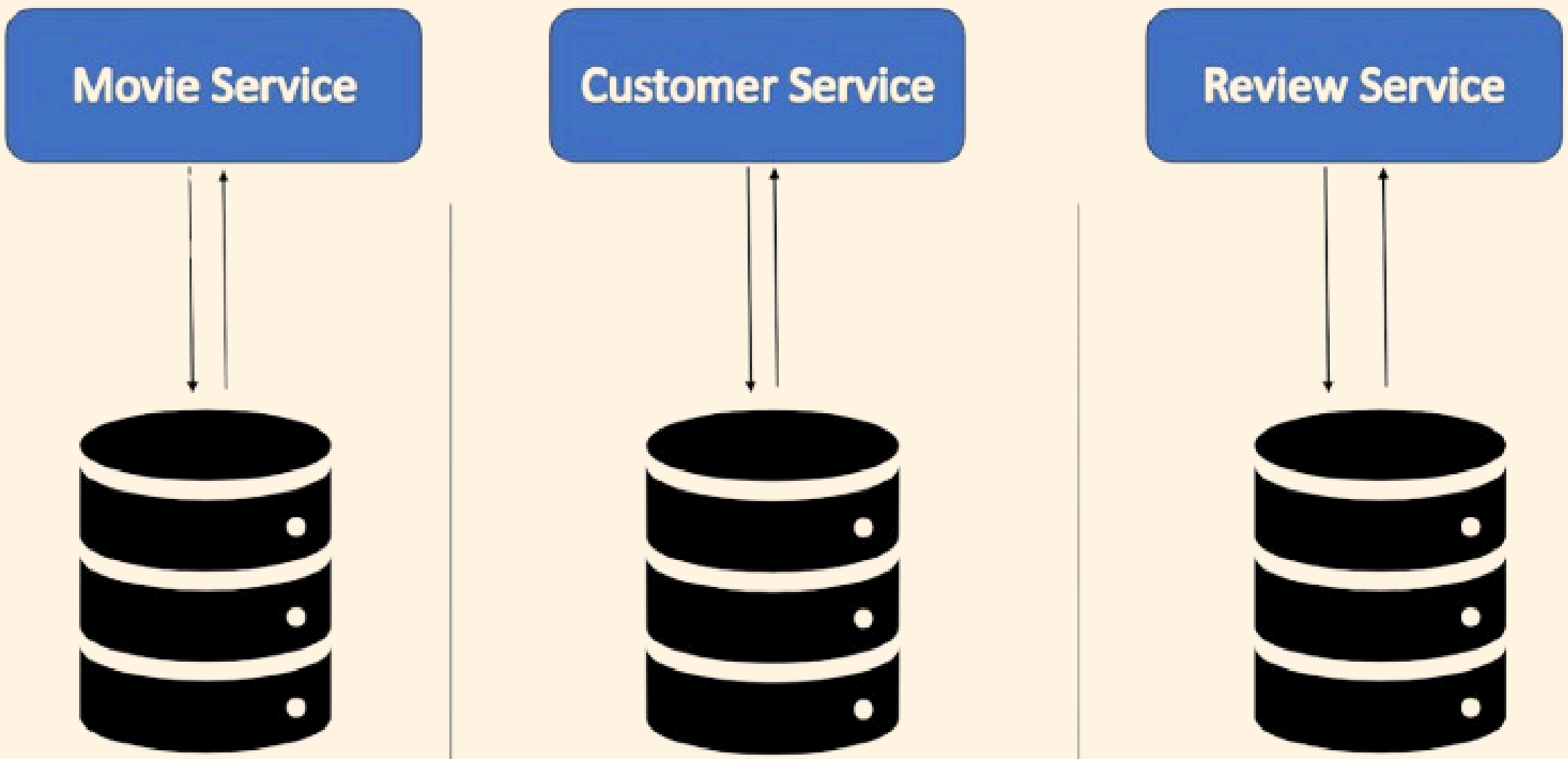


A large e-commerce platform where multiple clients (web, mobile, third-party) need to interact with various services (catalog, user management, orders).

The API Gateway simplifies client communication by providing a unified interface and handling complexities like security and routing.

Database per Service Pattern

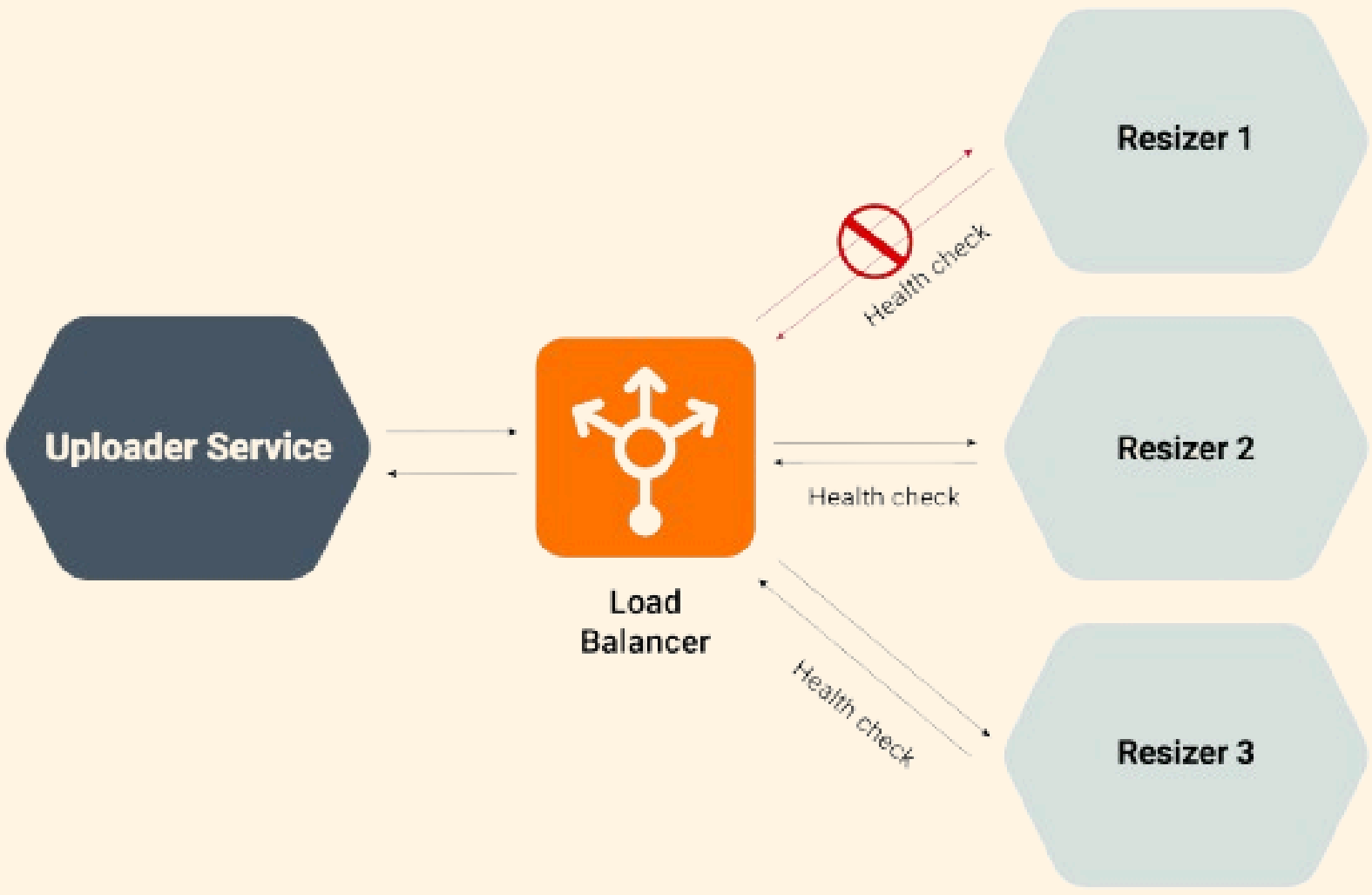
Each microservice has its own database, ensuring loose coupling and independent data management. This pattern avoids a single point of failure and allows services to use different types of databases suited to their needs.



A SaaS application with multiple microservices such as billing, user management, and analytics. Each service requires different database technologies (e.g., relational for billing, NoSQL for user profiles, time-series for analytics), allowing optimized performance and scalability.

Circuit Breaker Pattern

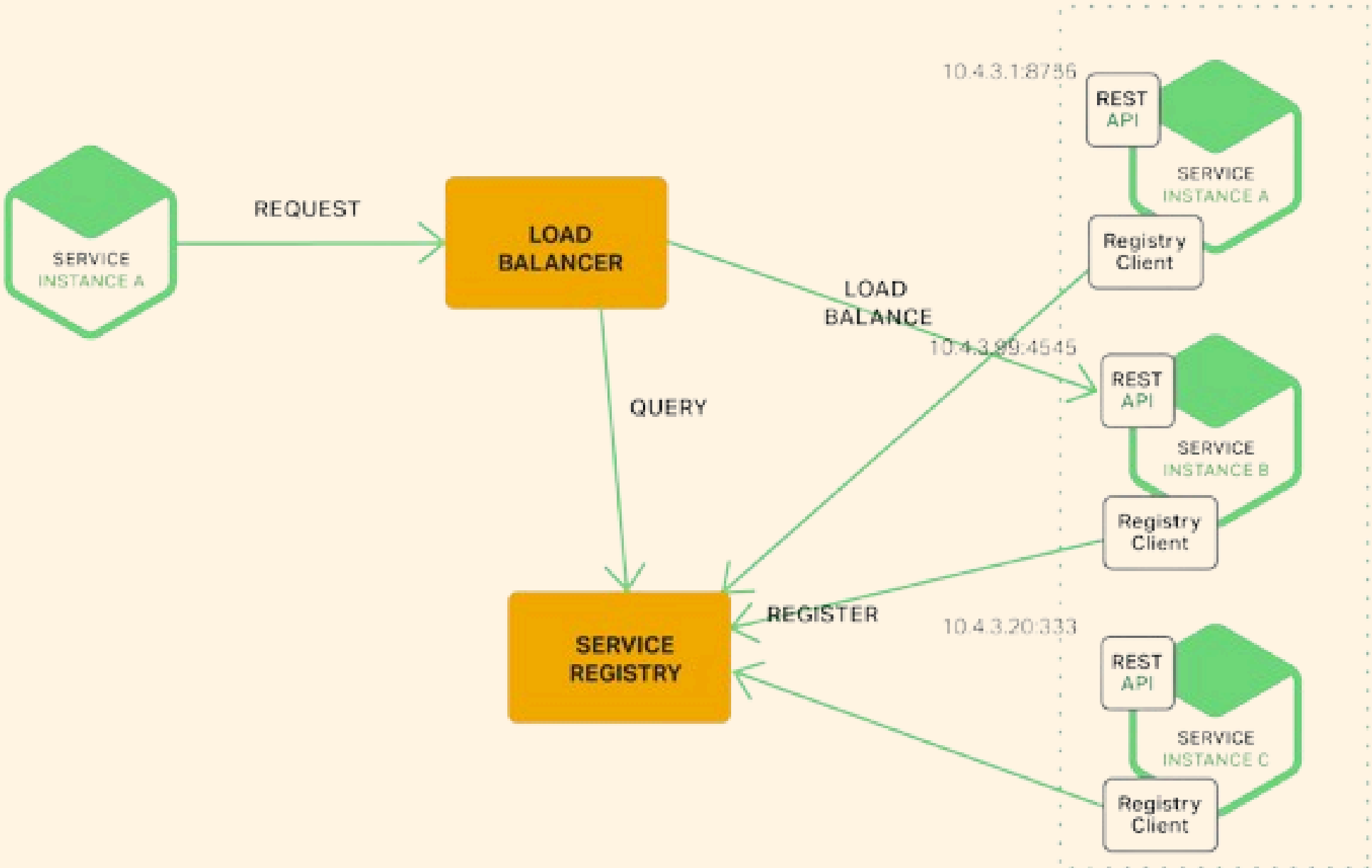
This pattern prevents service failure by providing a fallback mechanism when a service is unreachable or fails. It monitors service calls and "breaks" the circuit to prevent further calls when failures exceed a threshold.



A travel booking system where multiple external services (airline, hotel, car rental) are integrated. If one service is slow or fails, the circuit breaker prevents cascading failures and provides a default response to maintain system stability.

Service Discovery Pattern

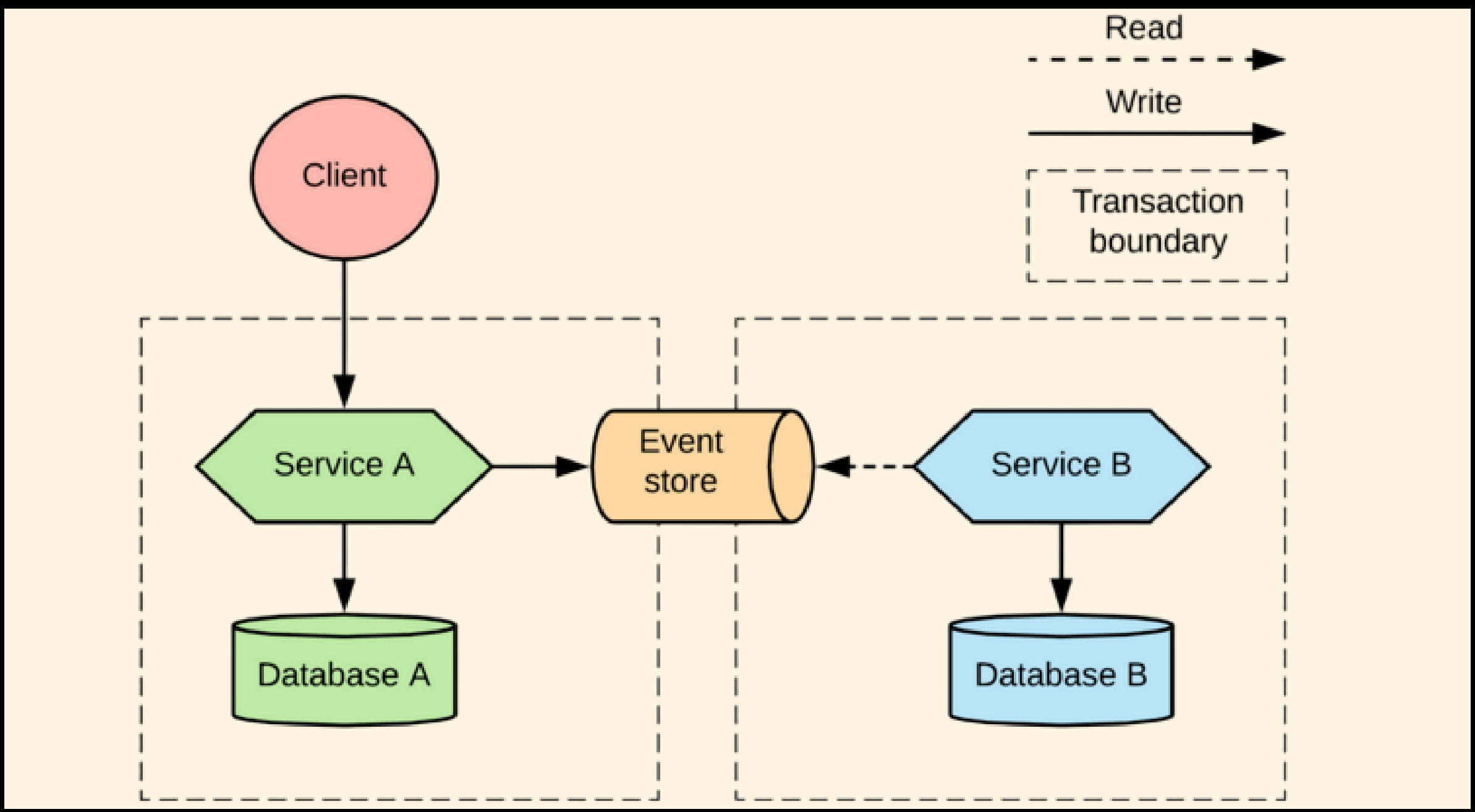
Service Discovery allows microservices to find and communicate with each other dynamically. It involves a service registry where services register themselves and look up other services.



A microservices-based application deployed in a cloud environment where instances of services start and stop frequently. Service discovery ensures that services can locate each other without manual configuration, enabling automatic scaling and resilience.

Event Sourcing Pattern

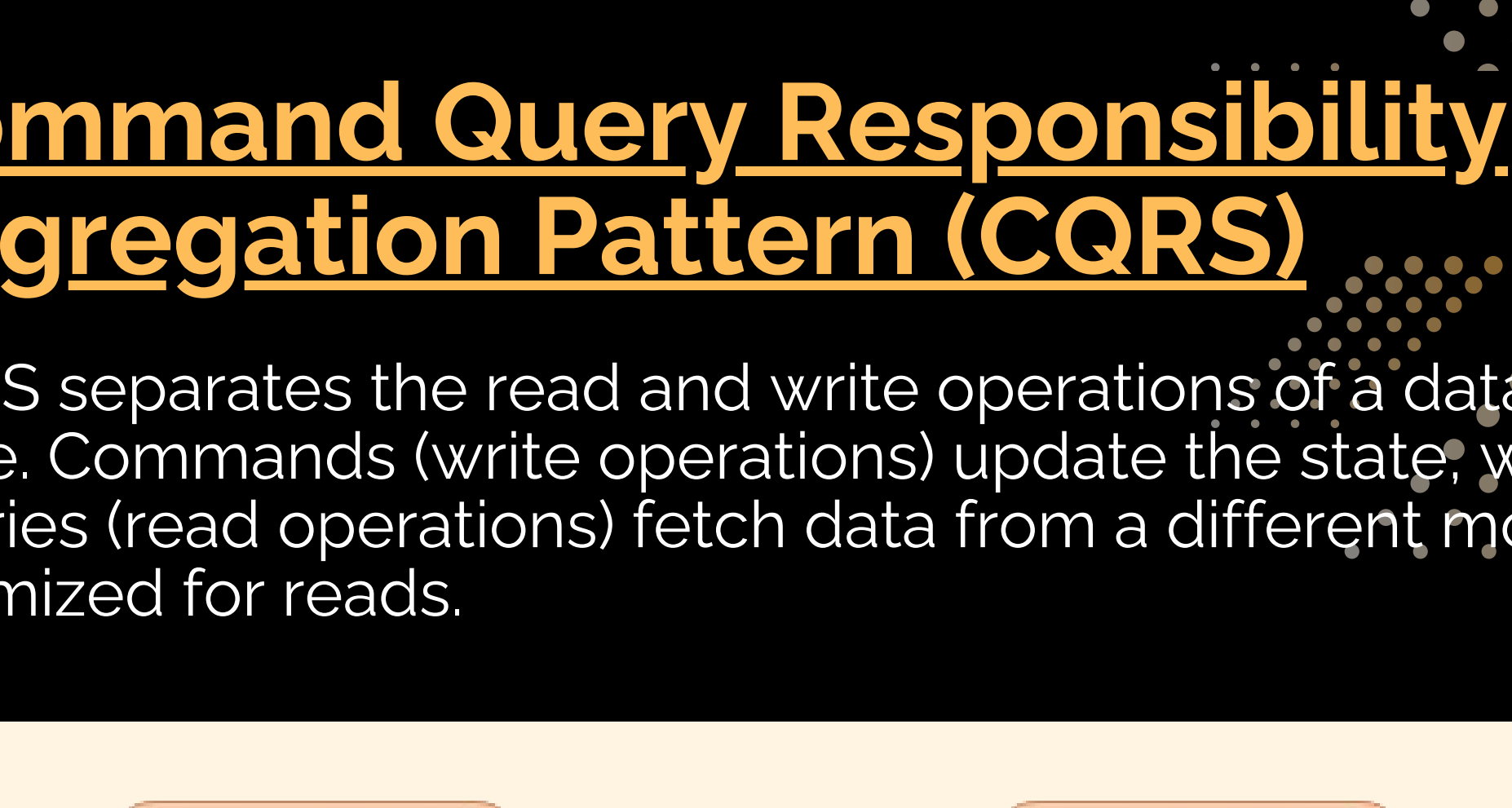
This pattern captures changes to an application state as a sequence of events. Instead of storing just the current state, it stores the state changes (events), allowing the system to reconstruct past states and audit trails.



A financial application managing transactions and accounts. By using event sourcing, the system can reconstruct account histories, track every transaction, and provide audit trails for regulatory compliance.

Command Query Responsibility Segregation Pattern (CQRS)

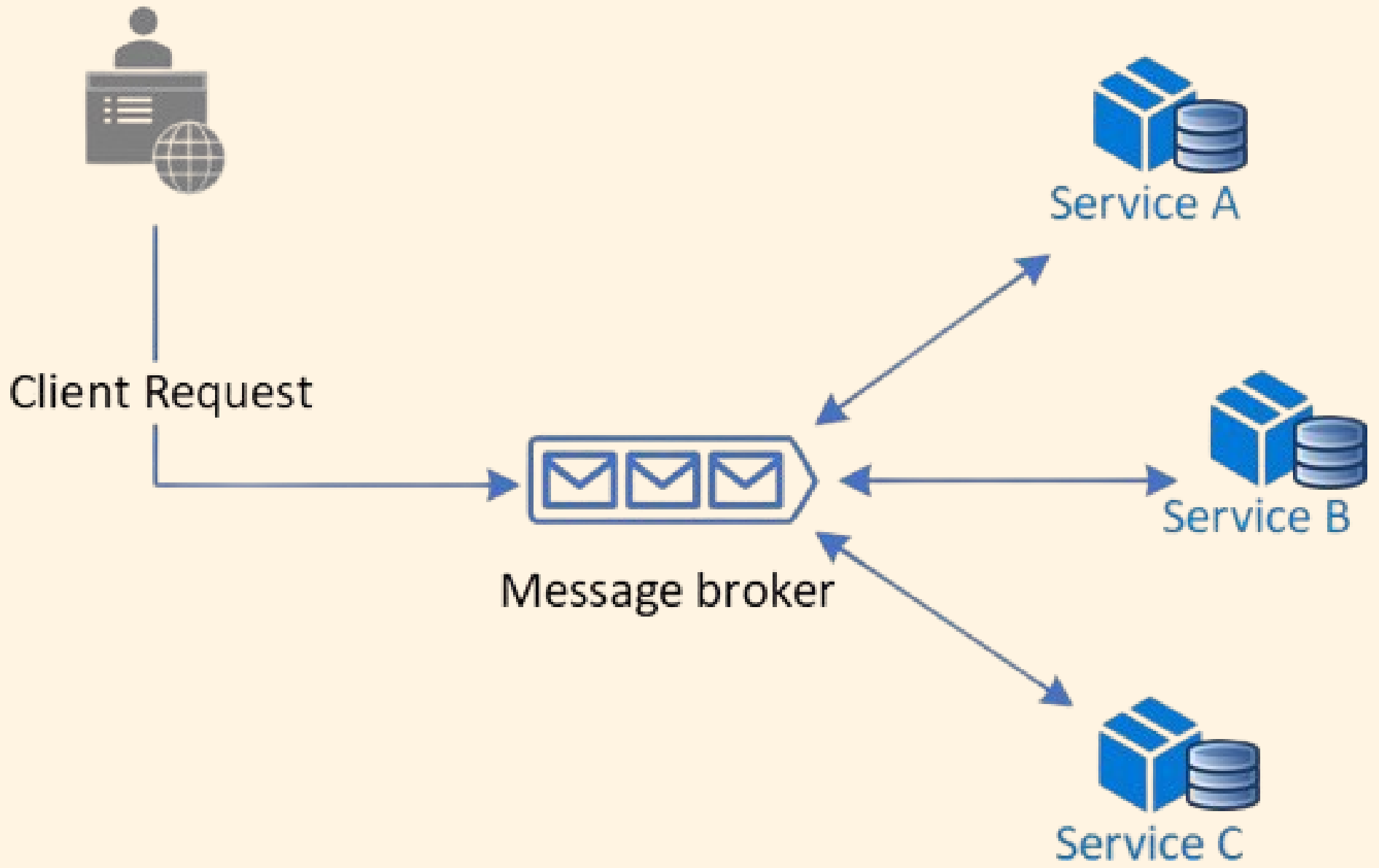
CQRS separates the read and write operations of a data store. Commands (write operations) update the state, while queries (read operations) fetch data from a different model optimized for reads.



An online retail application where the product catalog requires frequent updates and fast queries. Using CQRS, the write model ensures consistency when updating product information, while the read model provides quick responses for customer queries.

Saga Pattern

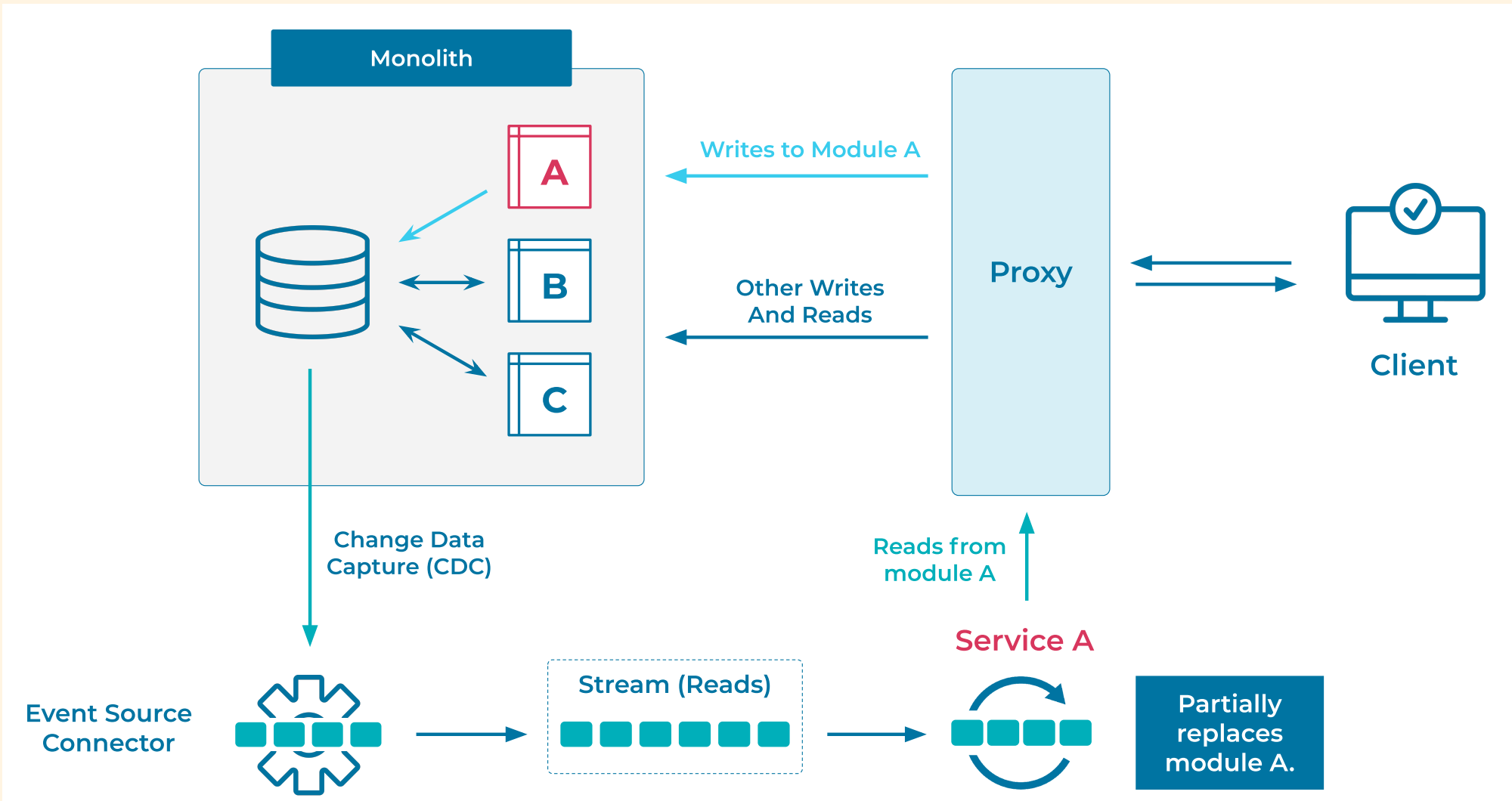
Saga manages distributed transactions across multiple microservices by coordinating a sequence of local transactions. Each service performs its transaction and publishes an event triggering the next service's transaction. If a transaction fails, compensating transactions undo the changes.



An order processing system where placing an order involves multiple services (payment, inventory, shipping). The saga pattern ensures that all steps are completed successfully, and if any step fails, compensating actions roll back the previous steps.

Strangler Fig Pattern

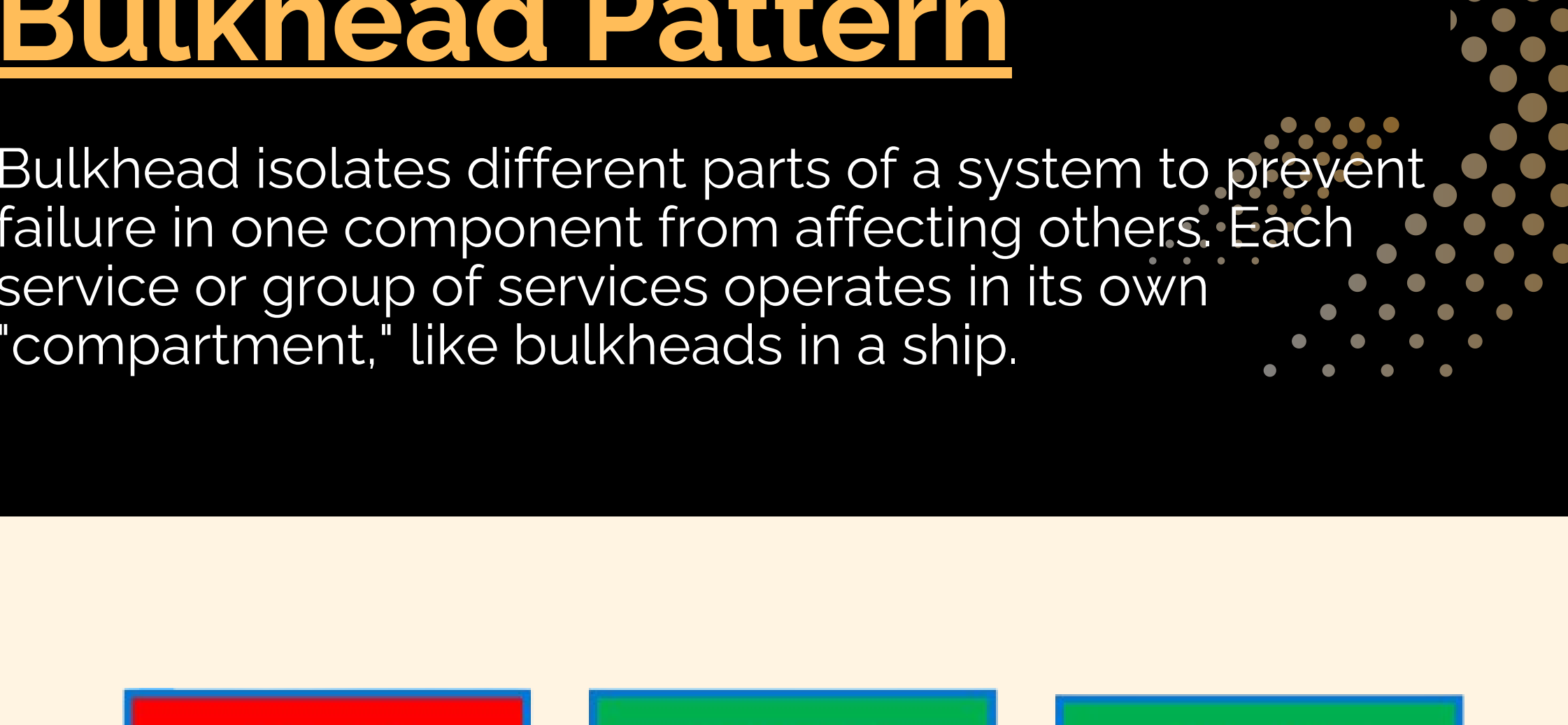
This pattern incrementally replaces a legacy system with a microservices architecture. The new system gradually takes over the functionality of the old system until the legacy system is entirely replaced.



Migrating a monolithic insurance application to a microservices architecture. The strangler fig pattern allows the new microservices to take over functionalities one by one, reducing the risk and complexity associated with a big-bang migration.

Bulkhead Pattern

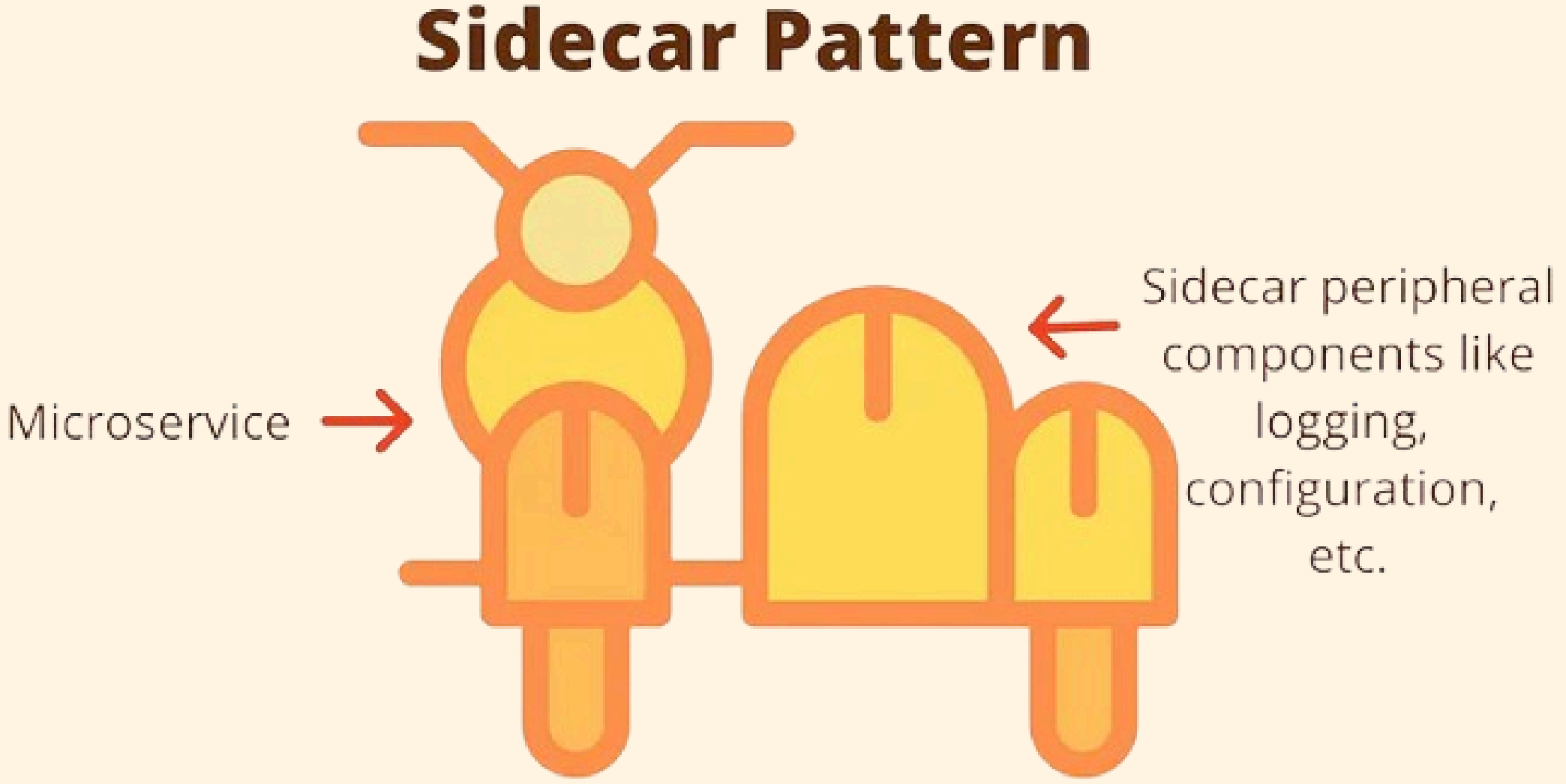
Bulkhead isolates different parts of a system to prevent failure in one component from affecting others. Each service or group of services operates in its own "compartment," like bulkheads in a ship.



A streaming service with different microservices for user management, video playback, and recommendations. Using bulkheads ensures that a failure in the recommendation service doesn't impact video playback or user management, maintaining overall system stability.

Sidecar Pattern

The Sidecar pattern deploys helper components (sidecars) alongside the main microservices. These sidecars handle cross-cutting concerns like logging, monitoring, and configuration management, allowing the main services to focus on business logic.



An application running in a Kubernetes cluster, where each microservice is accompanied by a sidecar for logging and monitoring. This pattern centralizes these concerns and simplifies the main service's codebase.

Created By:
Mohamed El Laithy

Want more tips?

Follow me and share this post with others who might find it helpful!

Scan me!