Microservices

Design Patterns

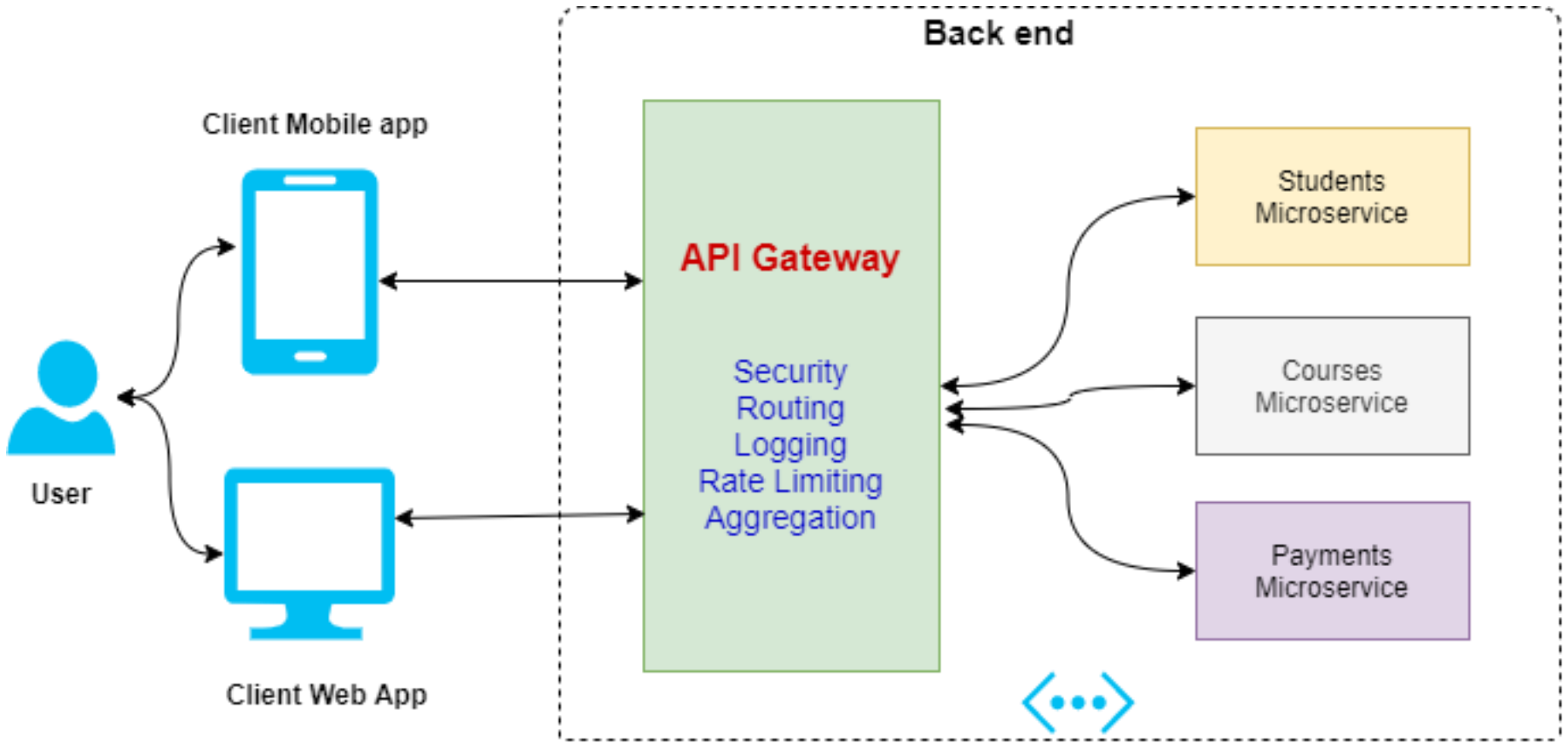
for

Designing and Implementing

Microservices

Microservices

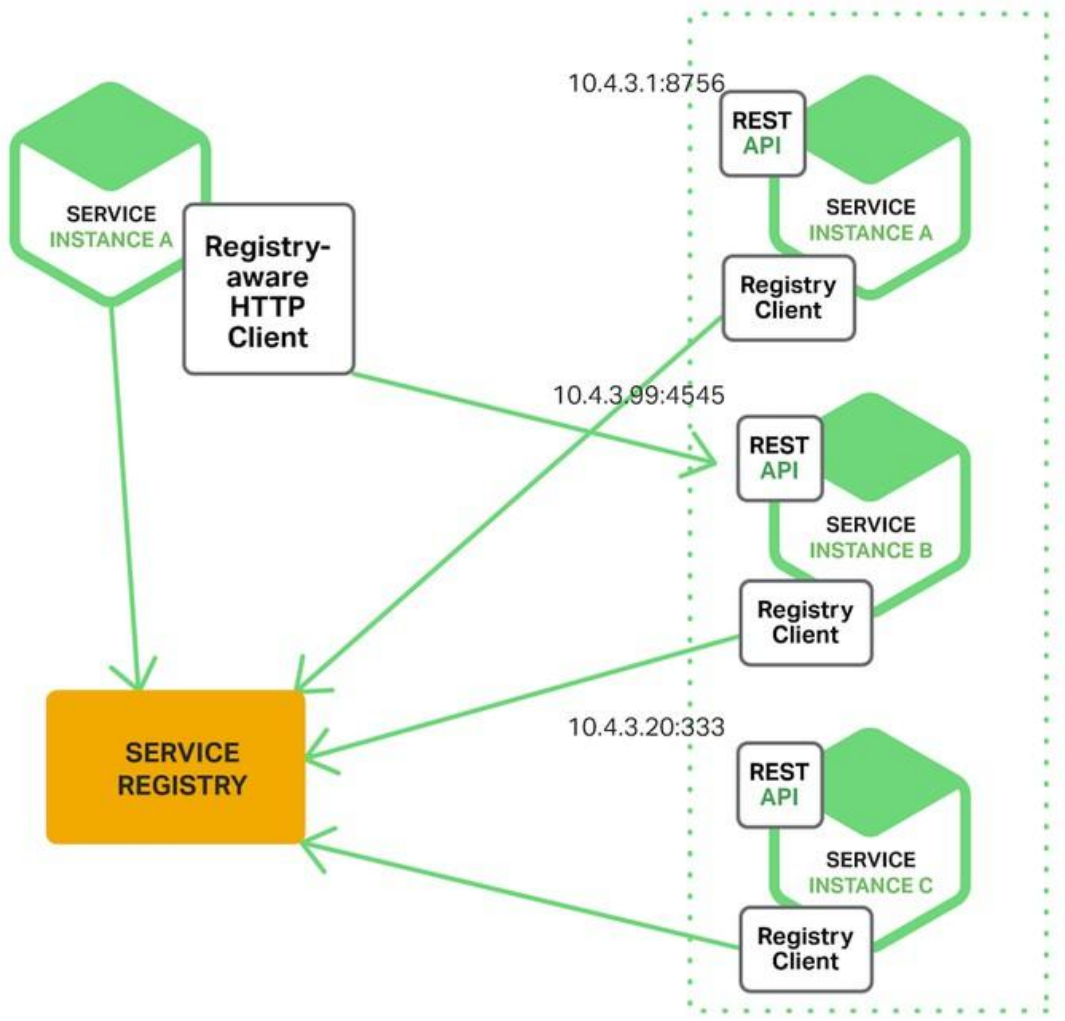
API GATEWAY PATTERN



API Gateway acts as a single entry point in microservices, handling authentication, routing, rate limiting, and response aggregation. It improves security, reduces direct service calls, and manages cross-cutting concerns. However, it can become a bottleneck if not optimized. Examples: Zuul, Spring Cloud Gateway, AWS API Gateway.

Microservices

SERVICE REGISTRY PATTERN

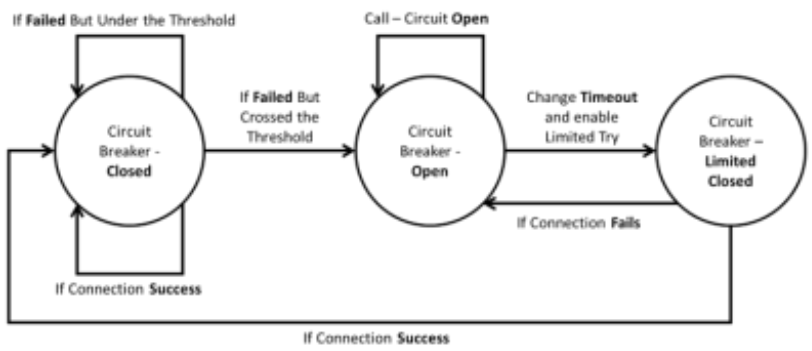


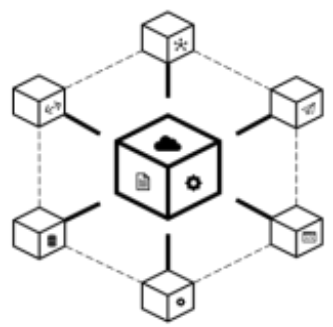
Use Eureka Server for service registration and discovery. Microservices register automatically, and others can discover them dynamically for communication.
Example: Spring Cloud Eureka.

Microservices

CIRCUIT BREAKER PATTERN

What is **Circuit Breaker** Design Pattern?



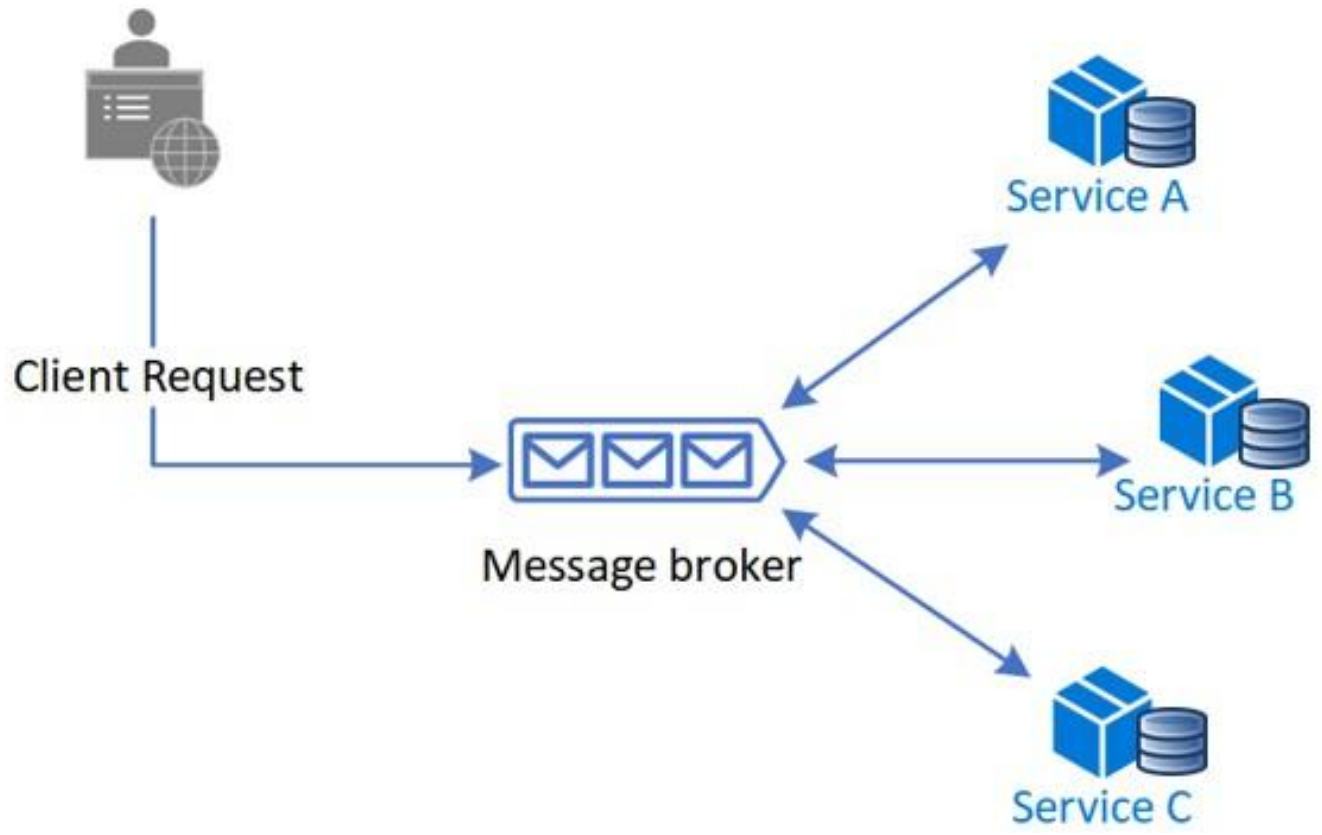


Circuit Breaker In Microservices

The Circuit Breaker pattern is a critical Java design pattern that helps ensure fault tolerance and resilience in microservices and distributed systems. Using Circuit Breaker, it is possible to prevent a system from repeatedly trying to execute an operation likely to fail, allowing it to recover from faults and prevent cascading failures

Microservices

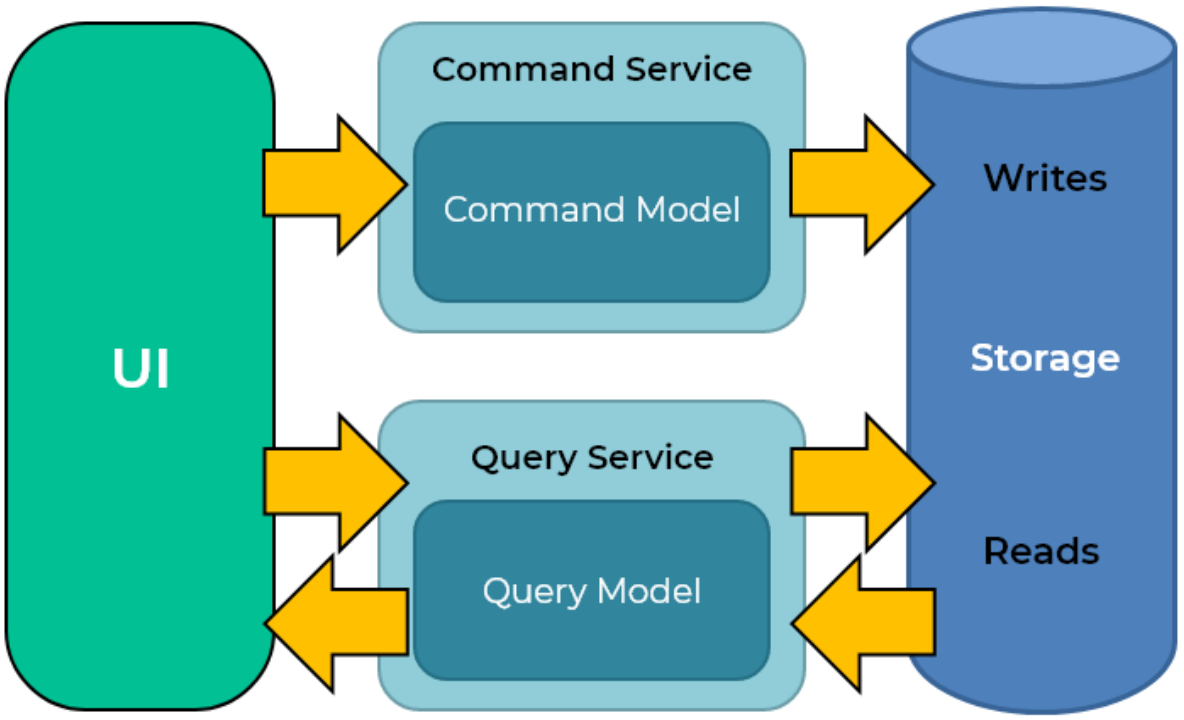
SAGA PATTERN



The Saga Pattern is a design pattern used in distributed systems to handle long-running transactions. Instead of executing a big transaction all at once, it breaks the work into smaller, manageable steps. If one step fails, a compensating action is triggered to undo previous steps, keeping the system consistent even in case of errors.

Microservices

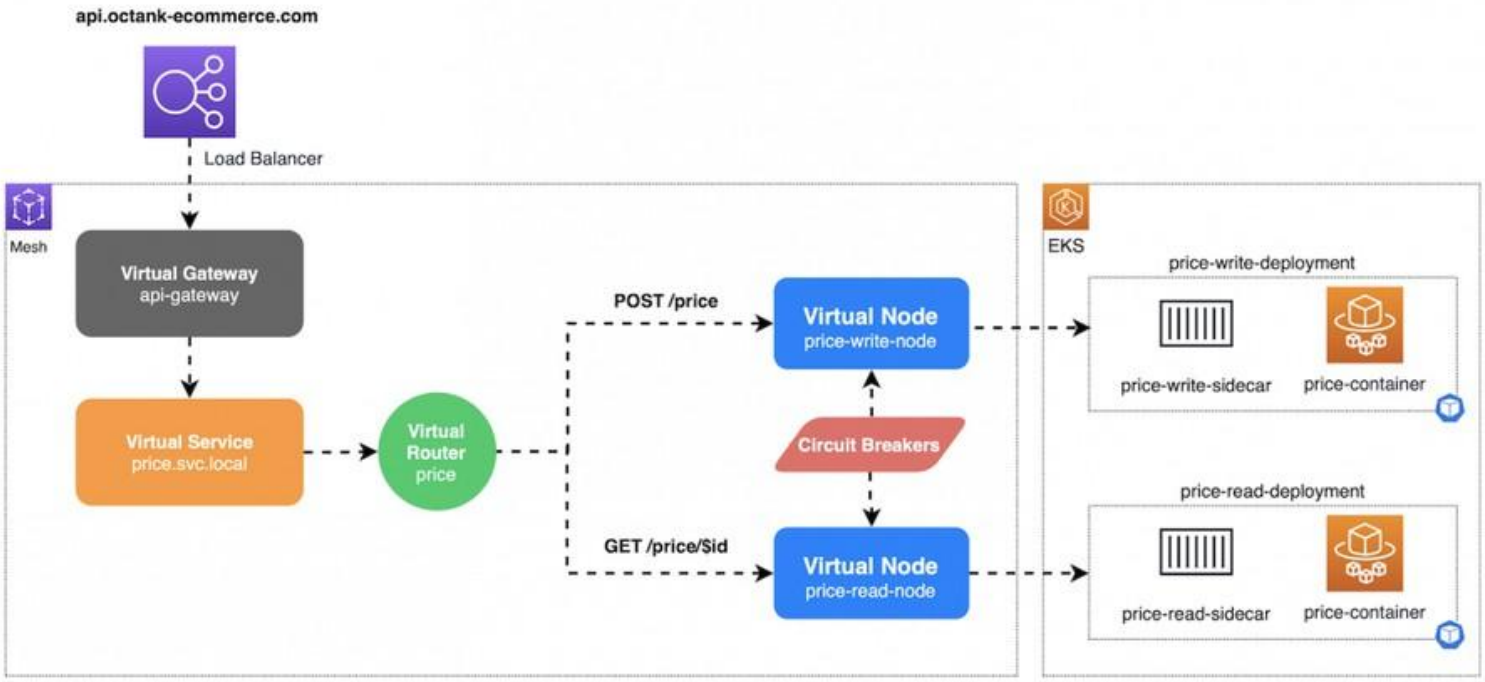
CQRS PATTERN



CQRS Pattern stands for Command Query Responsibility Segregation. It separates your system into two parts: one for commands that change data and another for queries that read data. This separation lets you optimize and scale the read and write parts independently, making your system simpler and more efficient.

Microservices

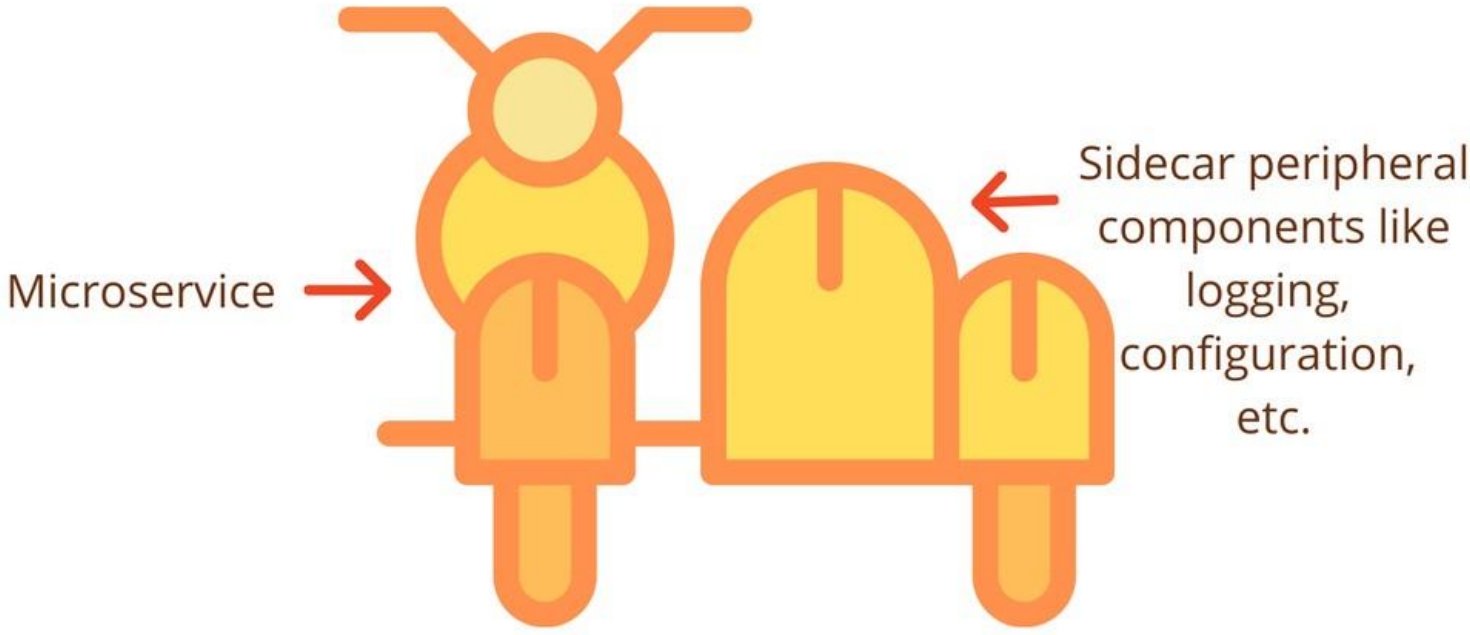
BULKHEAD PATTERN



The Bulkhead Pattern splits a system into separate parts so that if one part fails or gets overloaded, the rest continue working smoothly. It's like having compartments in a ship to prevent a single breach from sinking the entire vessel. Tools like Resilience4j and Hystrix help implement this pattern.

Microservices

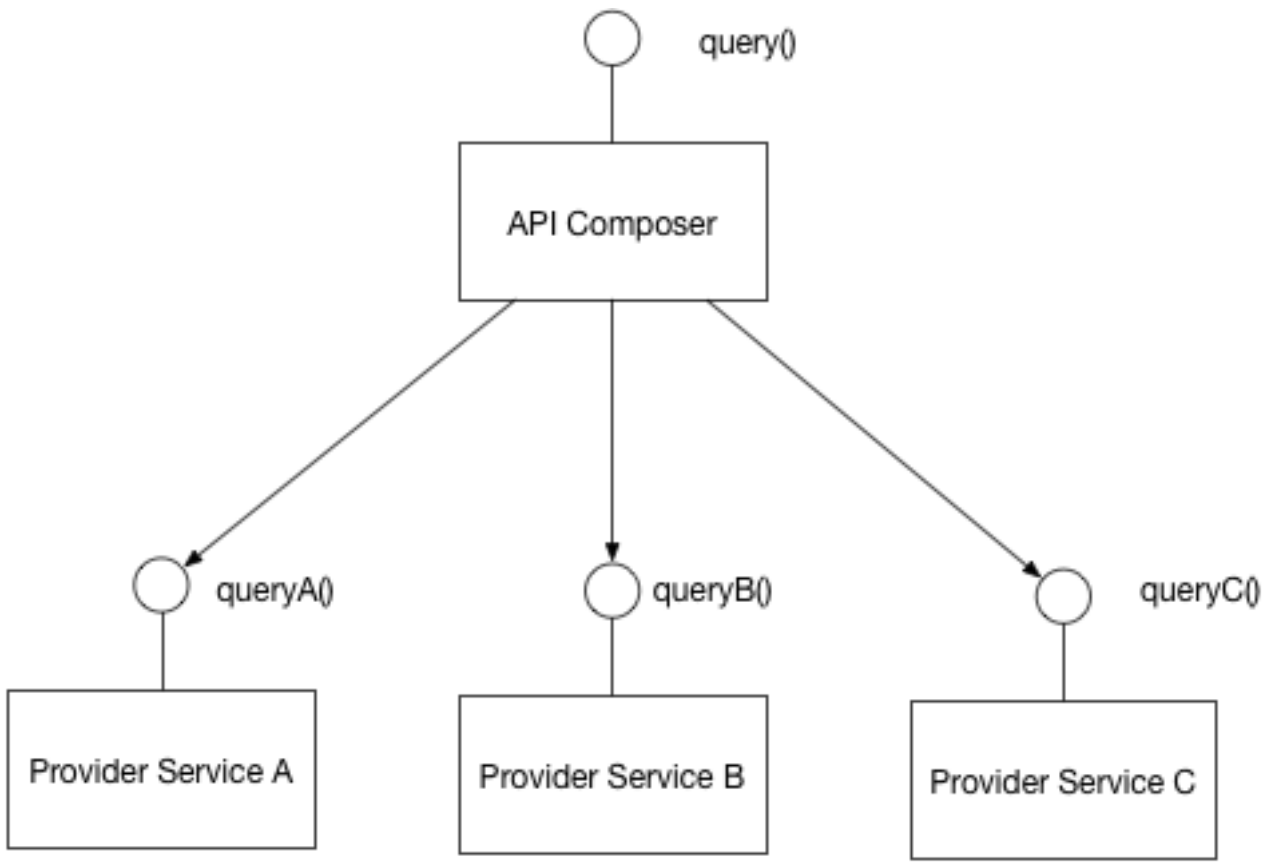
SIDECAR PATTERN



Attach a separate microservice (sidecar) to handle specific tasks like monitoring, logging, or authentication. Sidecar attaches a helper service to a main service for logging, monitoring, or security without modifying the main app.

Microservices

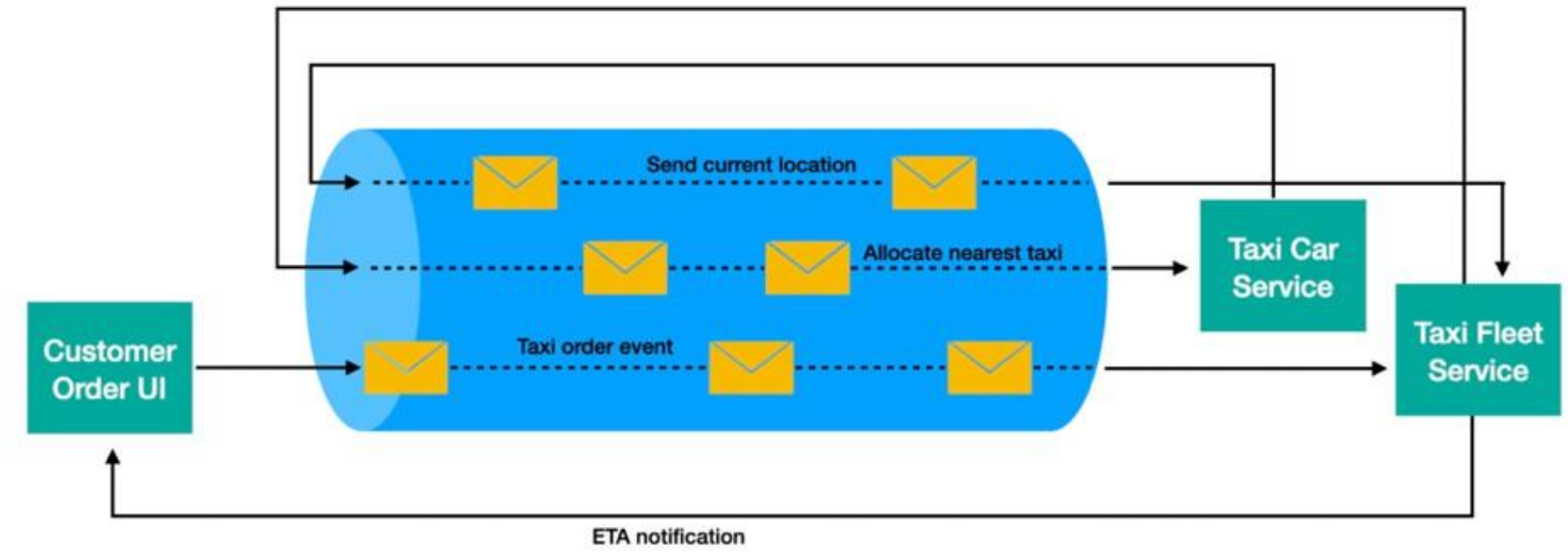
API COMPOSITION PATTERN



Combine multiple microservices to create a more complex and feature-rich API for clients. API Composition aggregates data from multiple microservices in a single response to reduce multiple client calls. It improves performance and efficiency. Examples: GraphQL, API Gateway.

Microservices

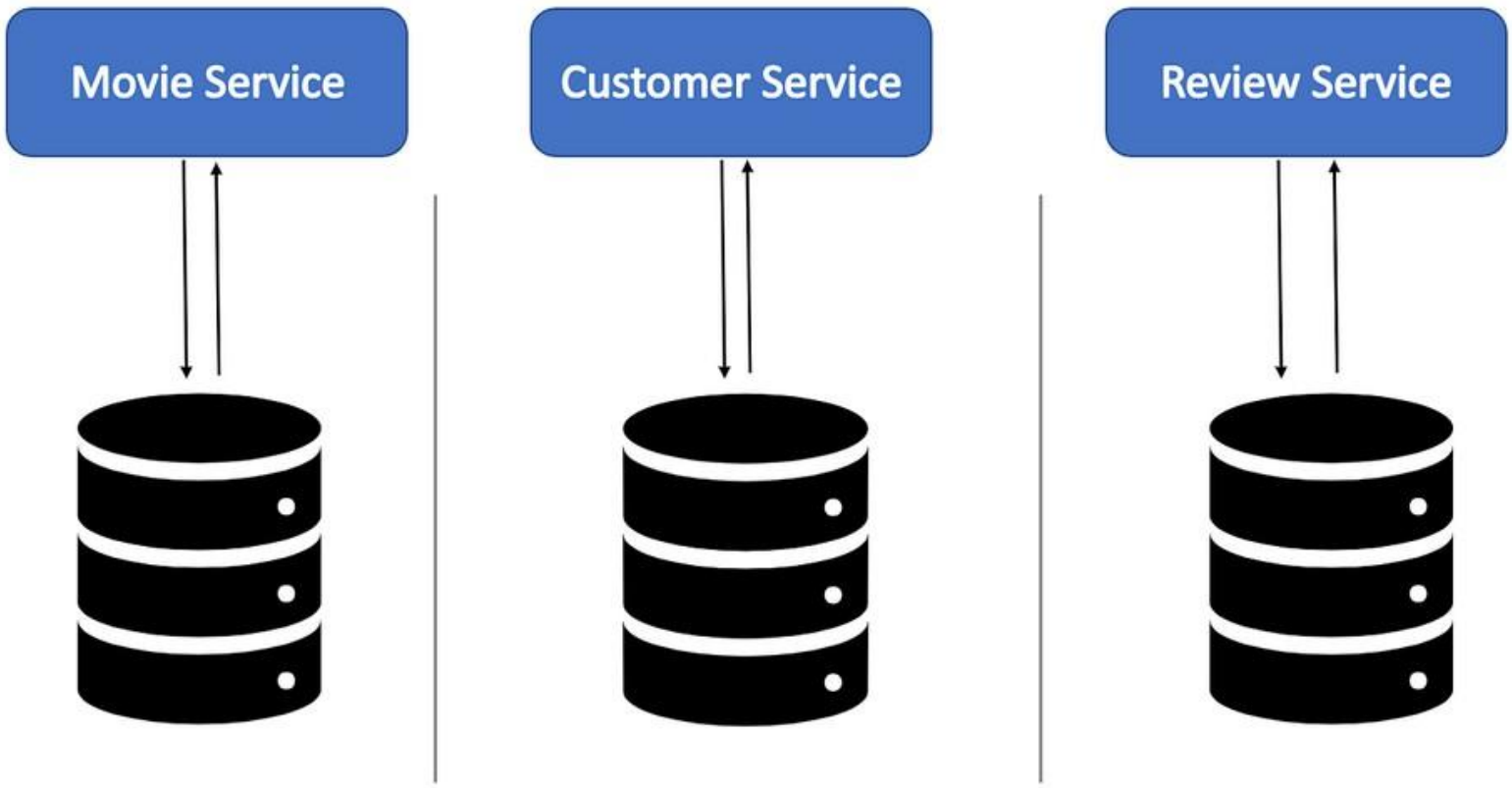
EVENT-DRIVEN ARCHITECTURE PATTERN



Communicate between microservices through events, enabling loose coupling and scalability. Event-Driven enables services to communicate asynchronously using events, improving scalability and decoupling. It enhances real-time processing. Examples: Kafka.

Microservices

DATABASE PER SERVICE PATTERN



The Database per Service Pattern means each service in a microservices system has its own separate database. This keeps services independent and lets them choose the best database for their needs. However, it can make sharing data and maintaining consistency between services a bit more challenging.

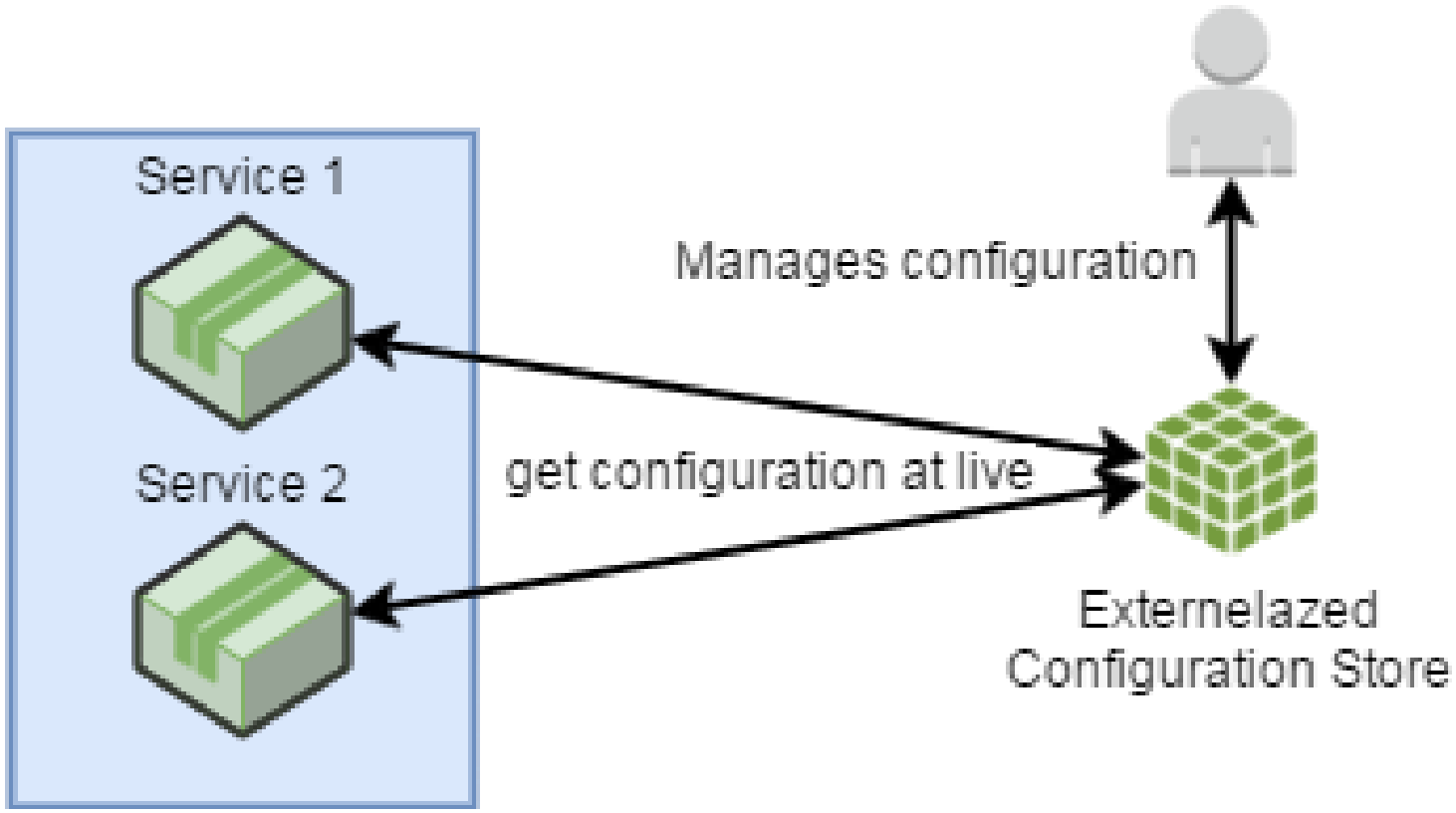
Microservices

RETRY PATTERN

The Retry Pattern is a design strategy where an operation is automatically tried again if it fails due to temporary issues. For example, if a network call fails because of a brief outage, the system will wait a little and try the operation again until it succeeds or reaches a maximum number of retries. This pattern helps improve system reliability by handling transient failures gracefully.

Microservices

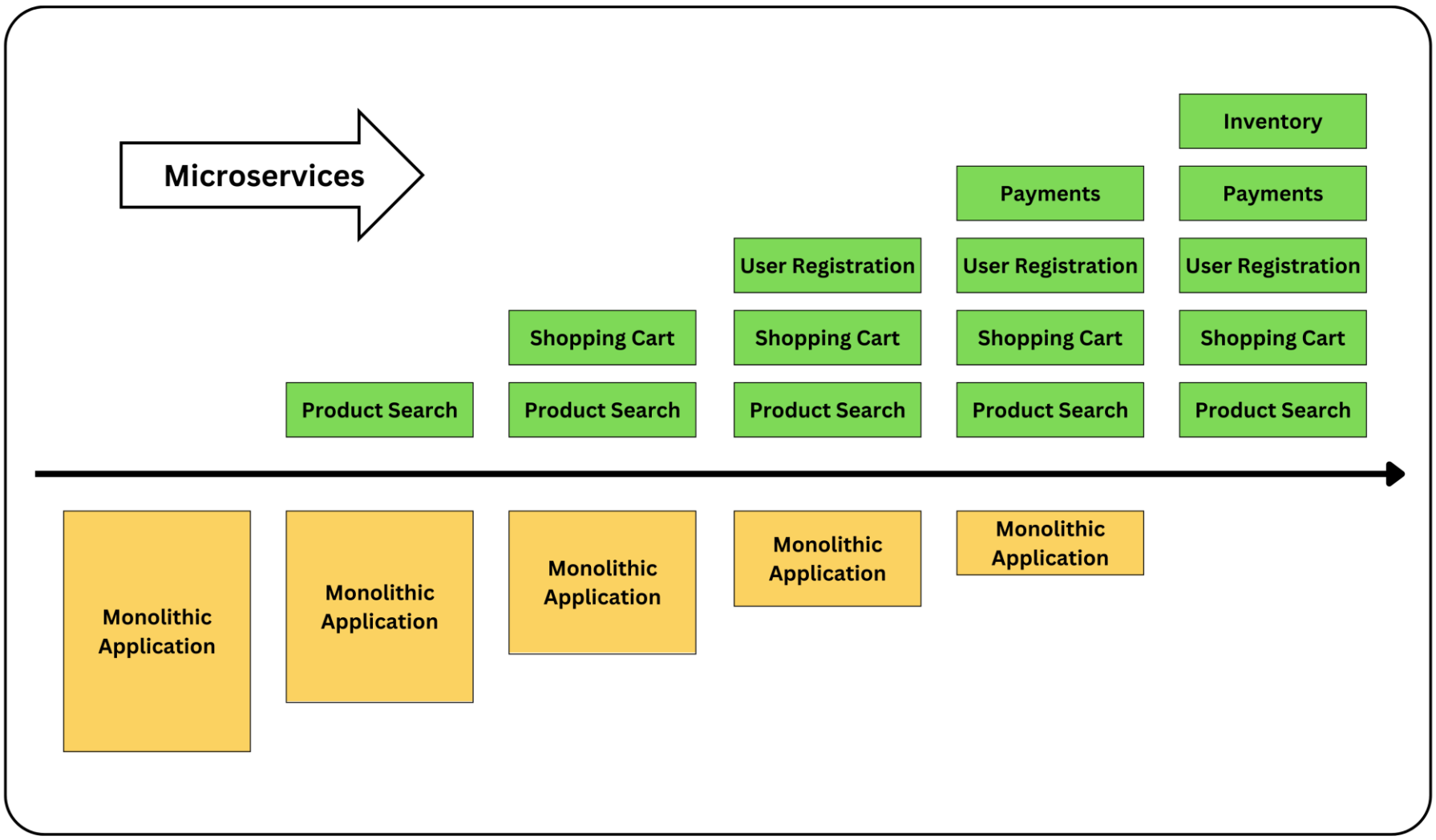
CONFIGURATION EXTERNALIZATION PATTERN



The Configuration Externalization Pattern means keeping settings (like database URLs, API keys, and other configurations) outside your code. This lets you change these details without modifying your program, making it easier to manage and adapt to different environments.

Microservices

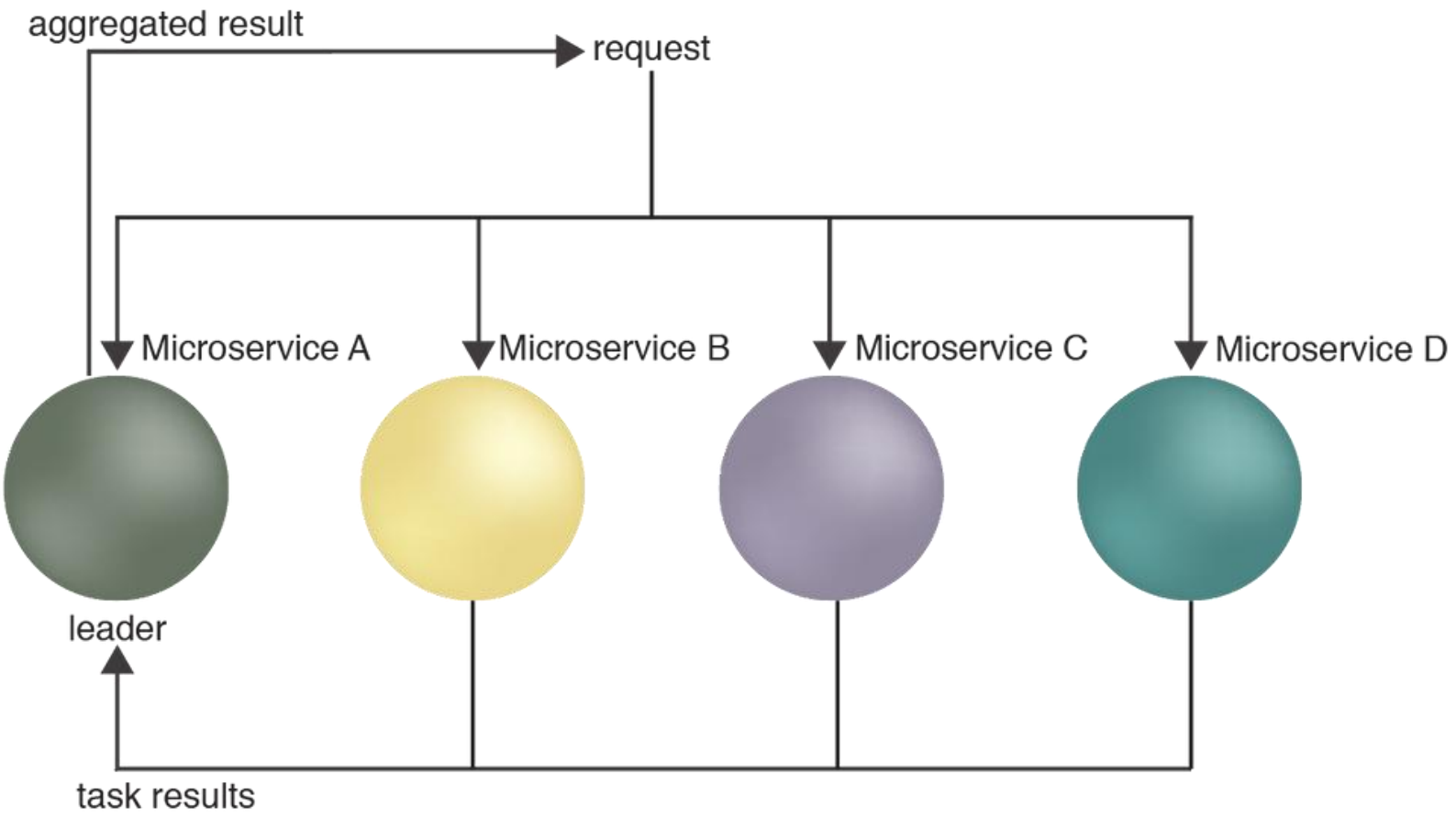
STRANGLER PATTERN



The Strangler Pattern is a way to slowly replace an old system with a new one. Instead of rewriting everything at once, you build new functionality around the old system and gradually take over its responsibilities. Over time, the new parts "strangle" the old system until it can be removed entirely, reducing risk during the transition

Microservices

LEADER ELECTION PATTERN



The Leader Election Pattern is used in distributed systems to choose one node as the leader. The leader coordinates tasks and manages shared resources. If the leader fails, a new leader is elected to keep the system running smoothly. This pattern helps maintain order and reliability in a network of multiple nodes.