

# Design Patterns for Micro Services

---

Presented By,  
Anvesh Valeti.

# What are the Challenges?

Scalability

Availability

Resiliency

Independent,  
autonomous

Decentralized  
governance

Failure isolation

Auto-  
Provisioning

Continuous  
delivery through  
DevOps

# Agenda

## Decomposition

- Decompose by business capability
- Decompose by subdomain (DDD)

## Integration

- API Gateway
- Aggregator
- Client-Side UI Composition

## Database

- Database per Service
- Shared Database
- Command Query Responsibility Segregation (CQRS)
- Saga (Choreography & Orchestrator)

## Communication

- Synchronous
- Asynchronous

## Cross Cutting Concern

- External Configuration
- Service Discovery
- Circuit Breaker

## Observability

- Log Aggregation
- Performance metrics
- Distributed Tracing



# Decomposition Patterns

# Decompose by Business Capability

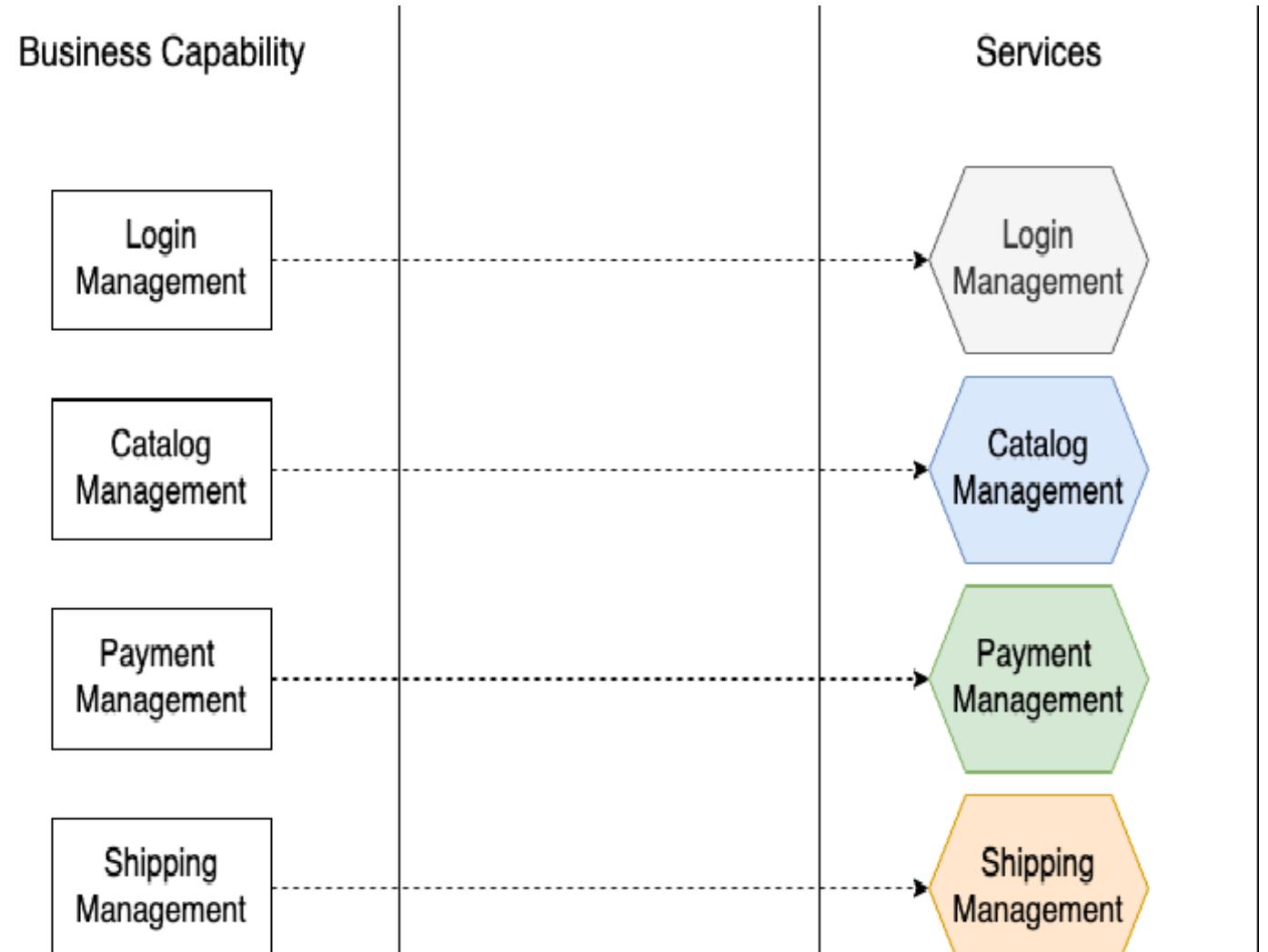
- **Problem**

Microservices is all about making services loosely coupled, applying the single responsibility principle. However, breaking an application into smaller pieces has to be done logically. How do we decompose an application into small services?

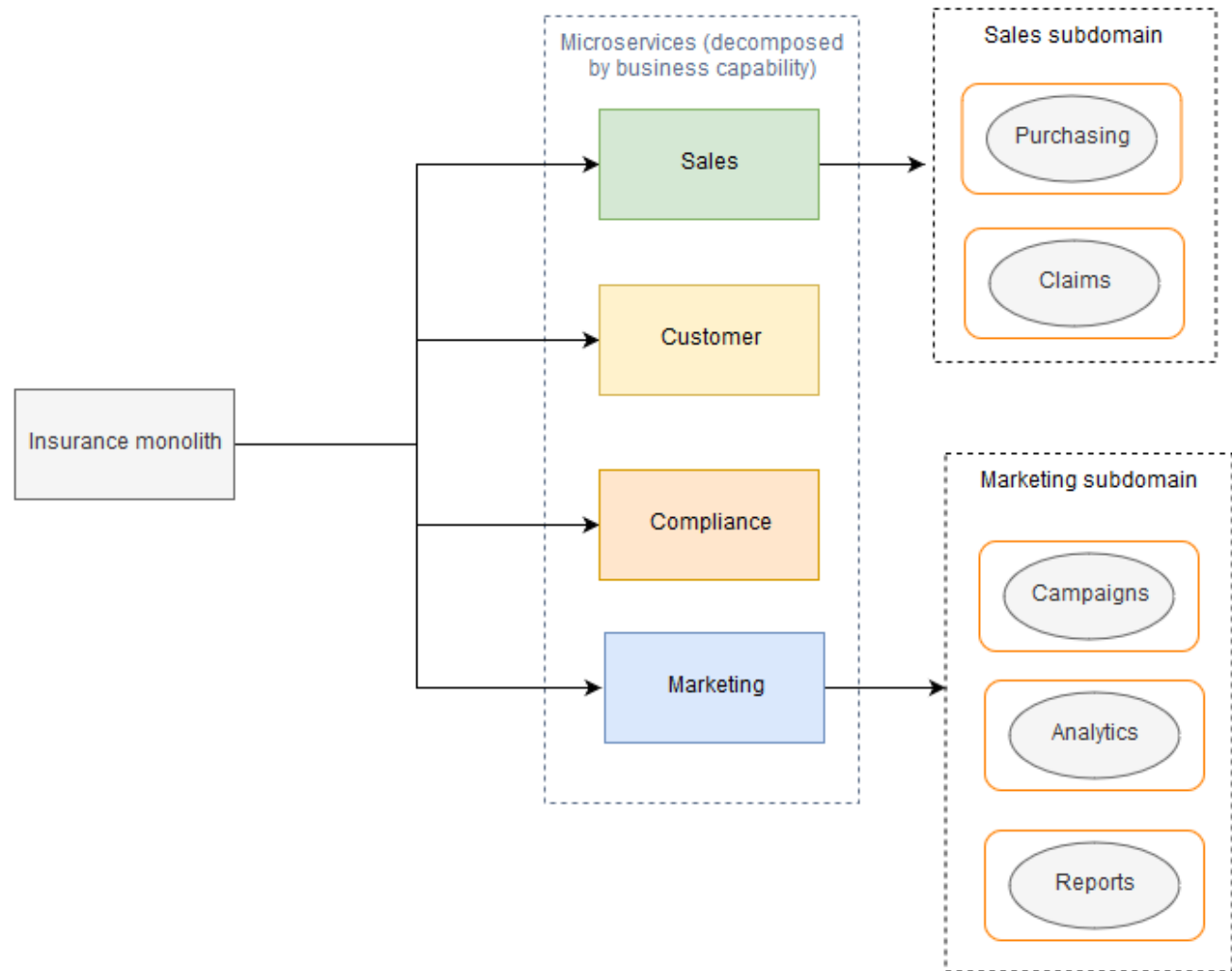
- **Solution**

- decompose by business capability
- DDD (Domain-Driven Design) - It uses subdomains and bounded context concepts to solve this problem

# Business capability



DDD

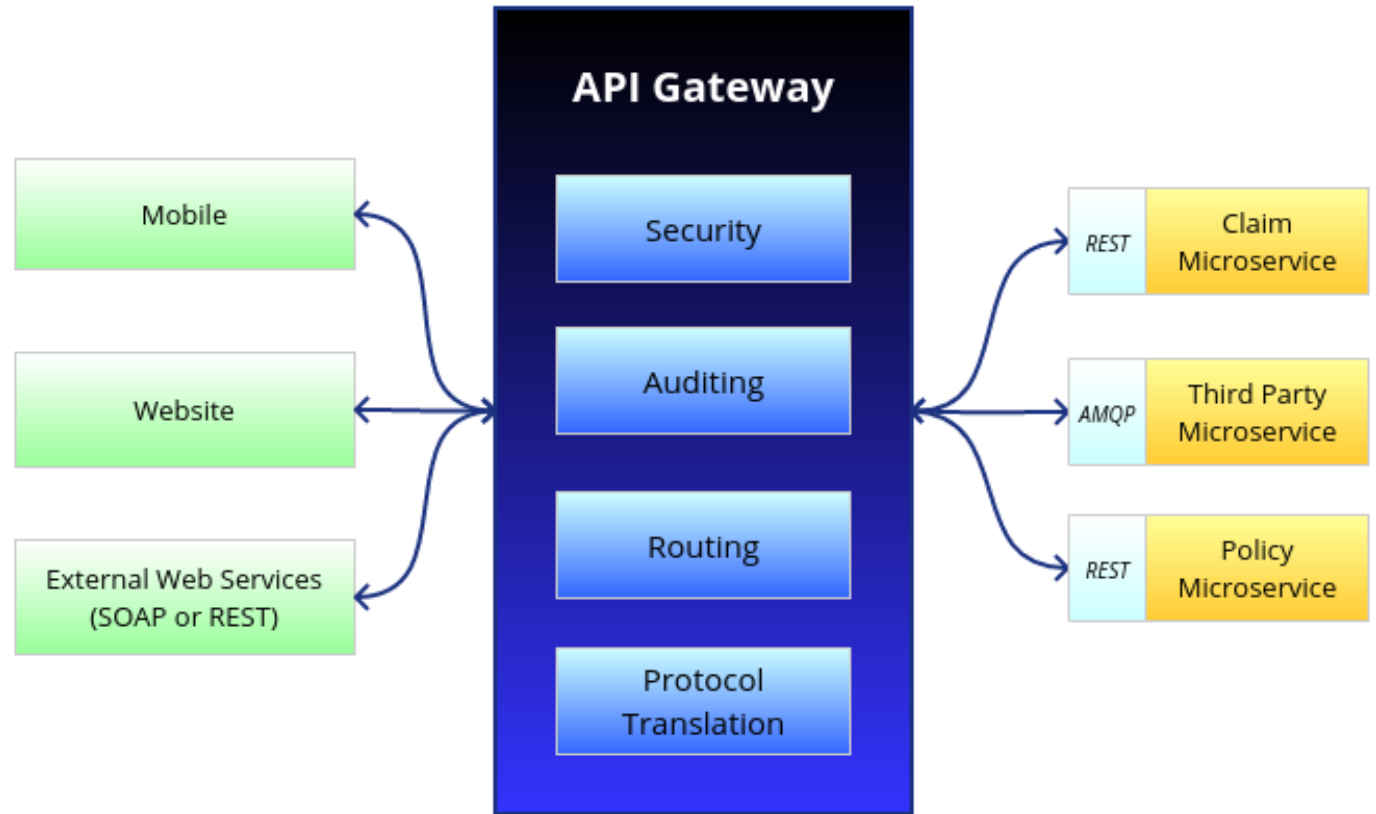


# Integration Patterns





# API Gateway Pattern



# Aggregator Pattern

- **Problem**

When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service.

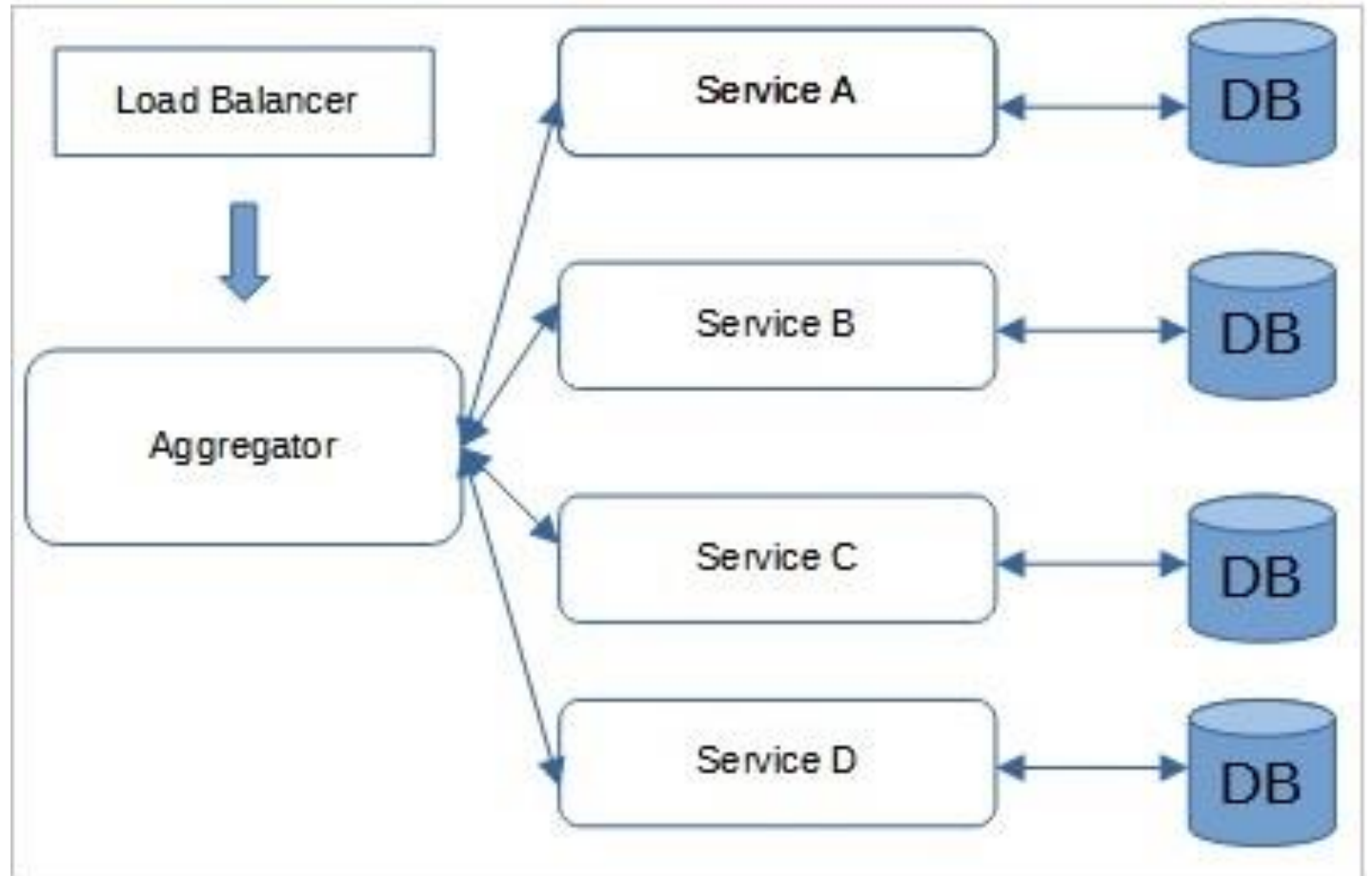
- **Solution**

1. A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.

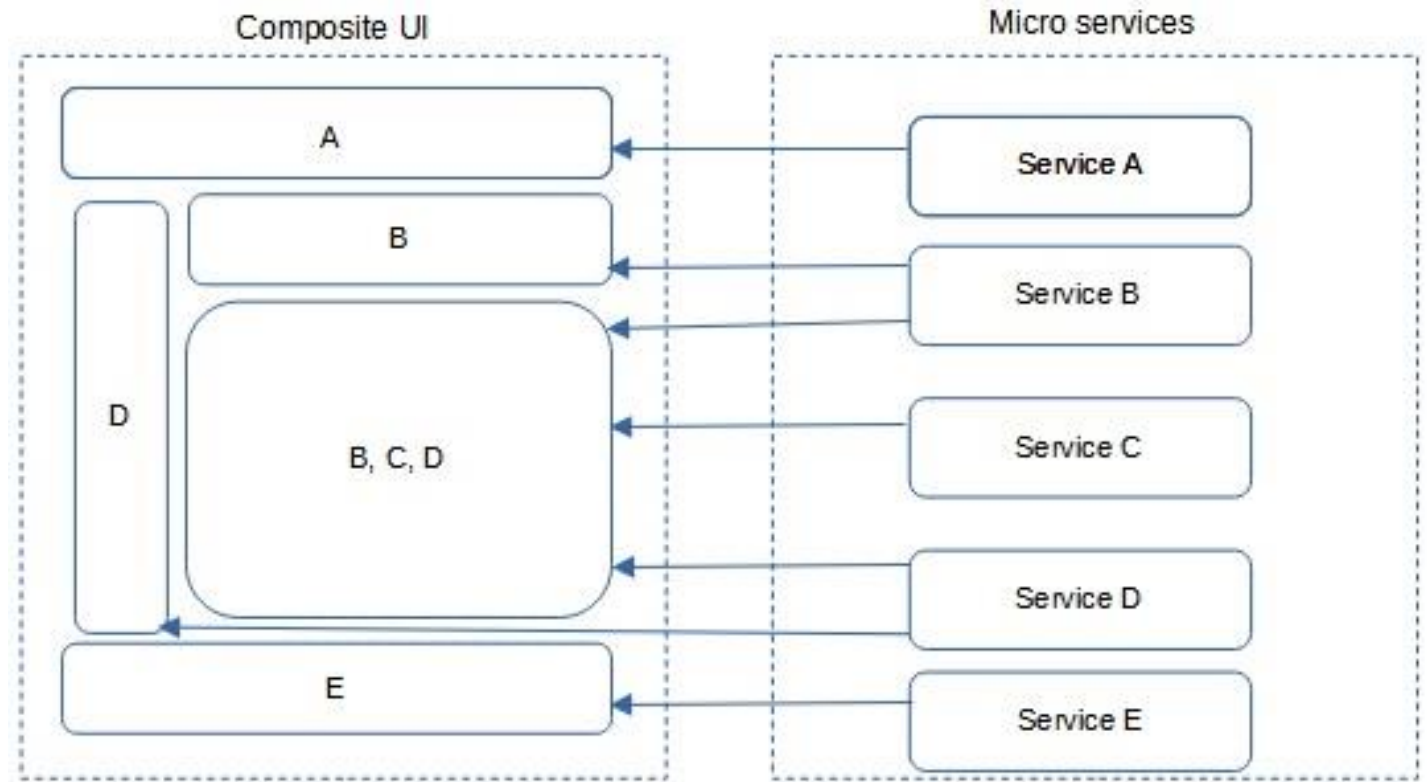
2. An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

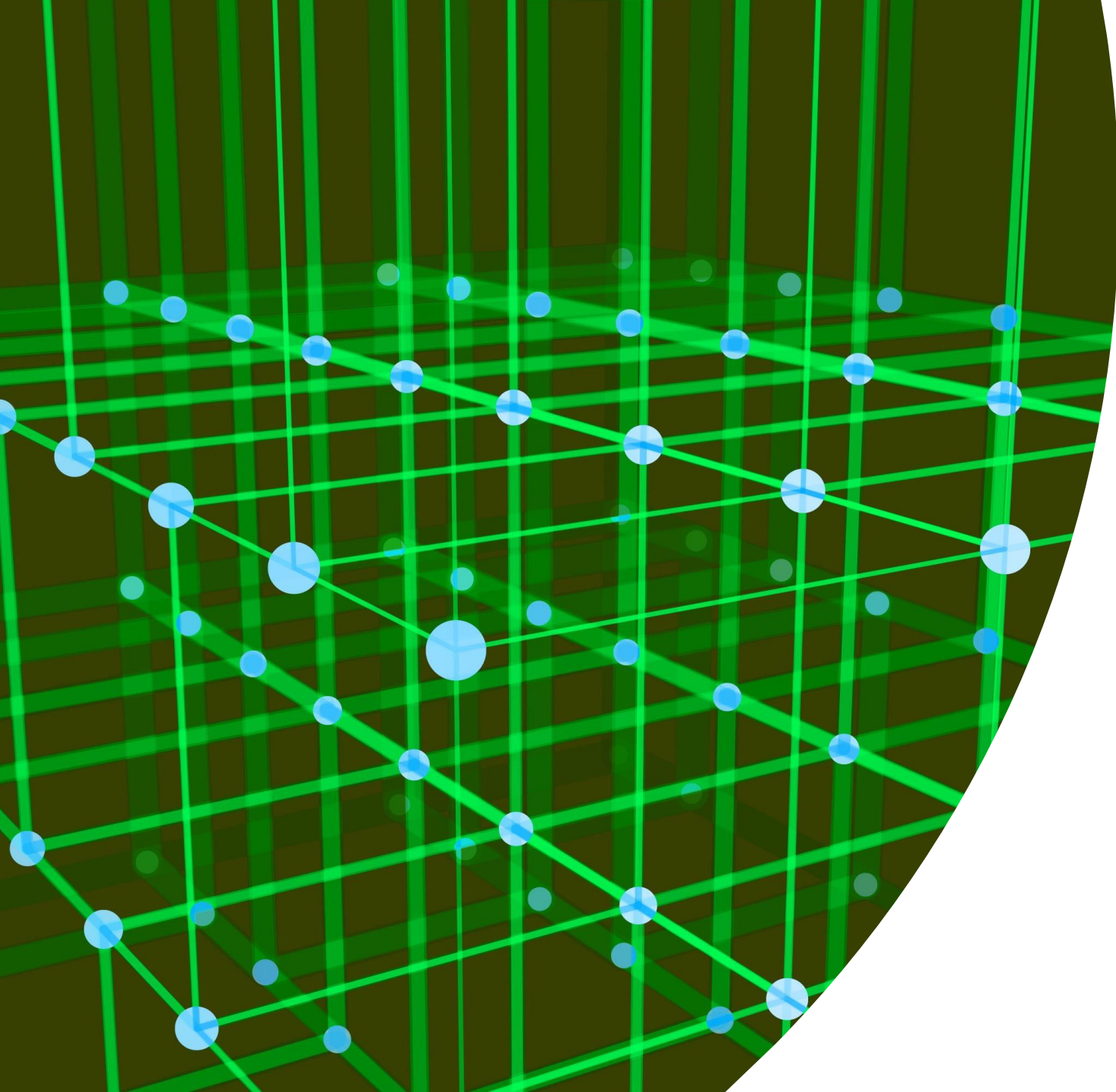
- It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

# Composite Service



# Client-Side UI Composition Pattern





# Database Patterns

# Database per Service

- **Problem**

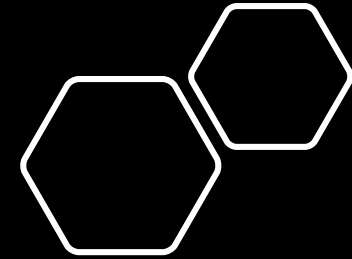
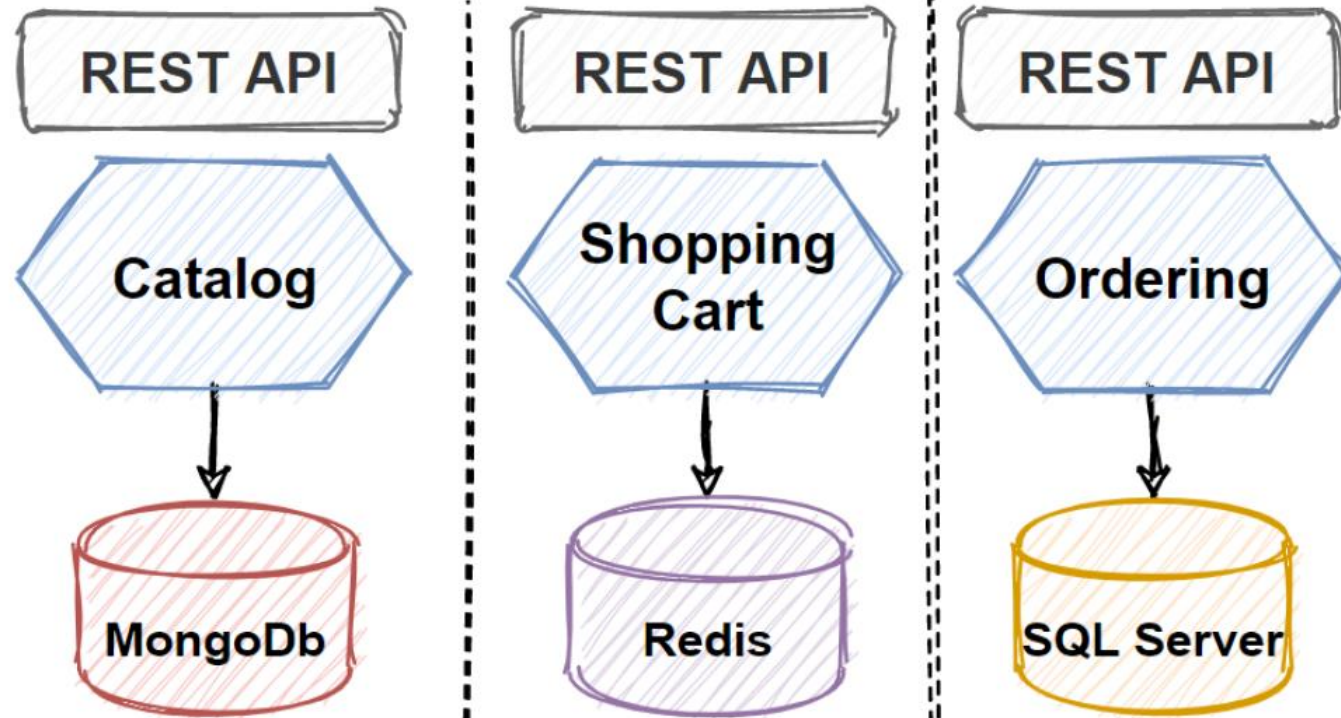
There is a problem of how to define database architecture for microservices.

Following are the concerns to be addressed:

1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.
2. Business transactions may enforce invariants that span multiple services.
3. Some business transactions need to query data that is owned by multiple services.
4. Databases must sometimes be replicated and sharded in order to scale.
5. Different services have different data storage requirements.



### The Database-per-Service pattern



# Shared Database per Service

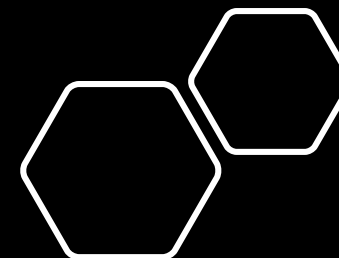
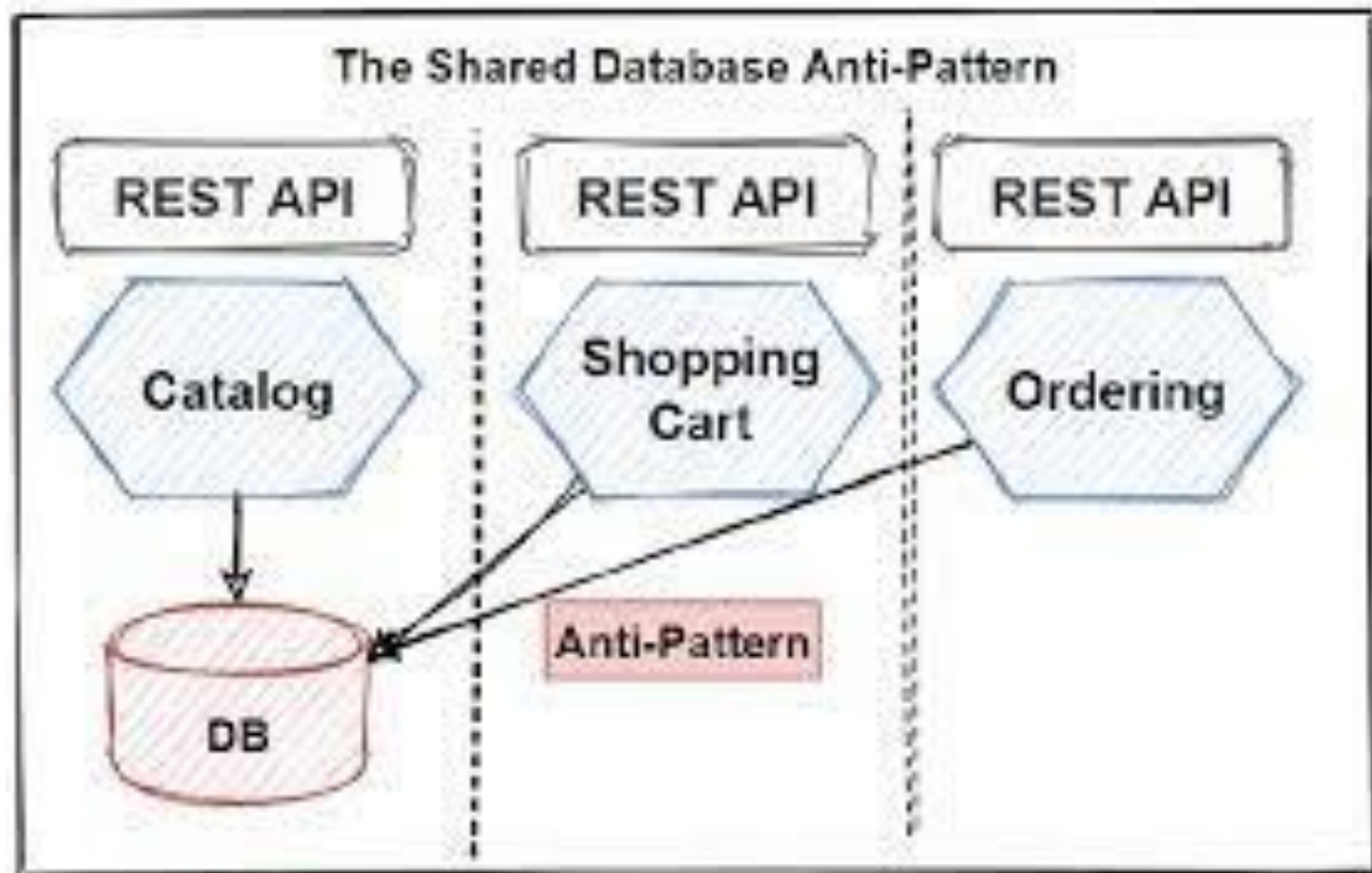
- **Problem**

We have talked about one database per service being ideal for microservices, but that is possible when the application is greenfield and to be developed with DDD. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. What is the suitable architecture in that case?

- **Solution**

- A shared database





# Command Query Responsibility Segregation (CQRS)

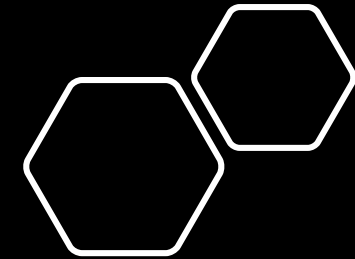
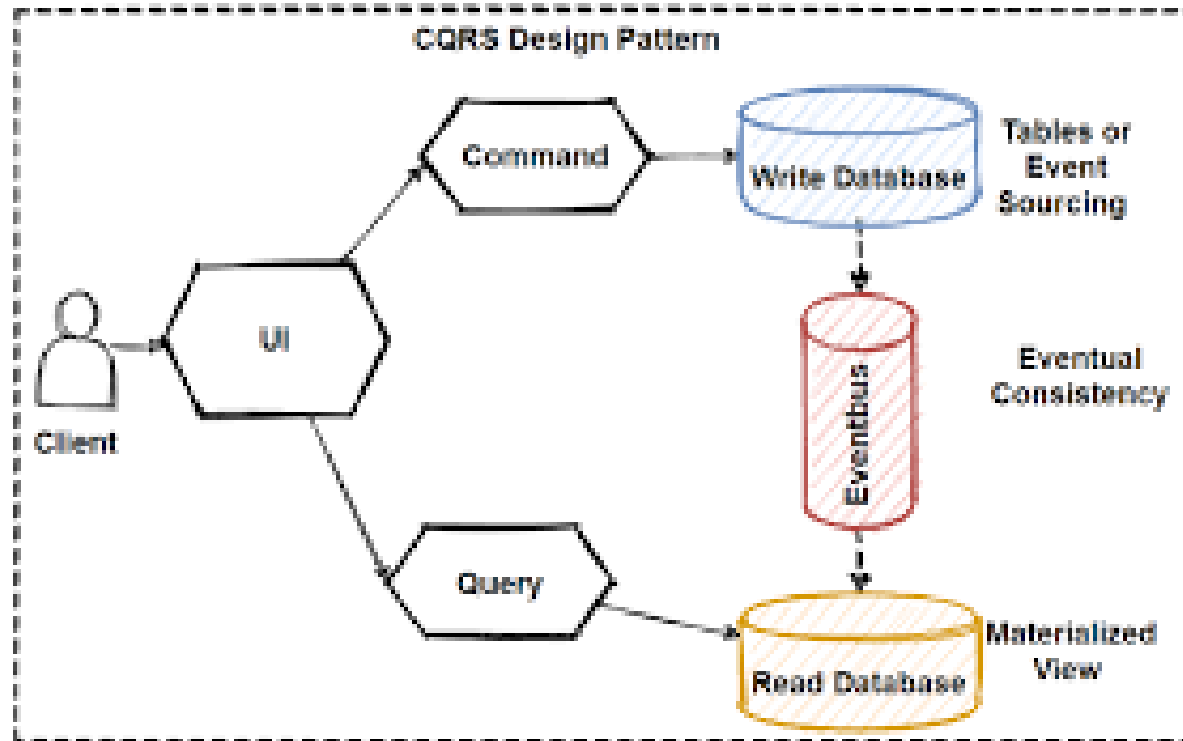
- **Problem**

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible. Then, how do we implement queries in microservice architecture?

- **Solution**

-CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query part by using the materialized views.

-The **event sourcing pattern** is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.



# Saga Pattern

- **Problem**

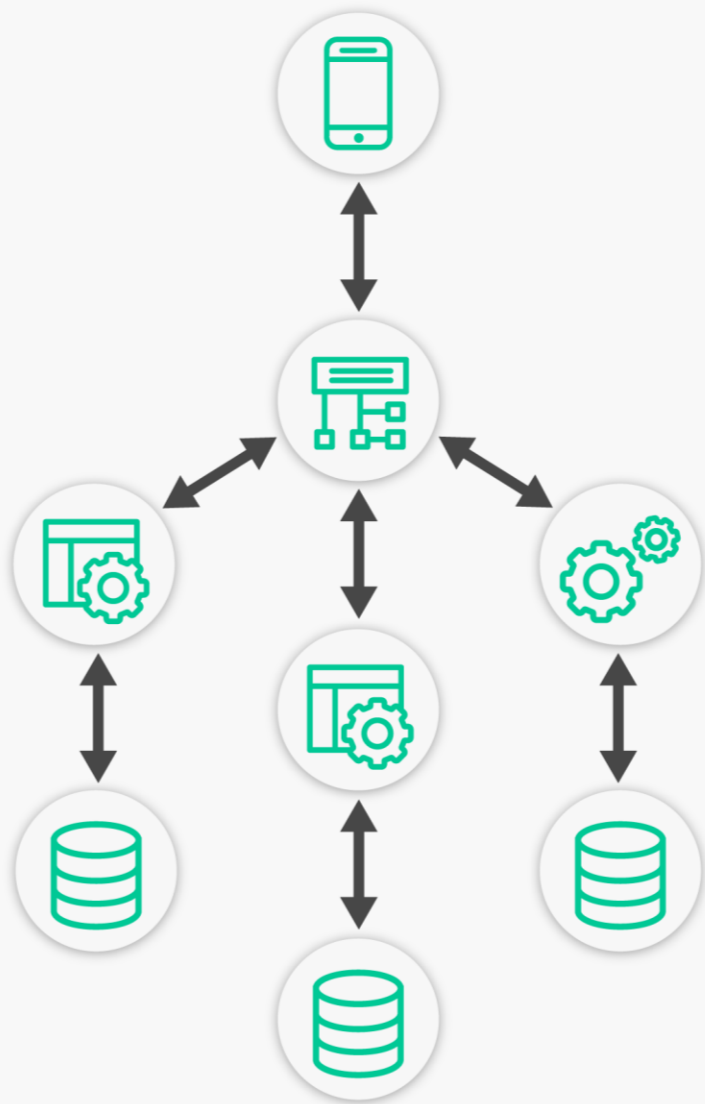
When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services? For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer's credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

- **Solution**

A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

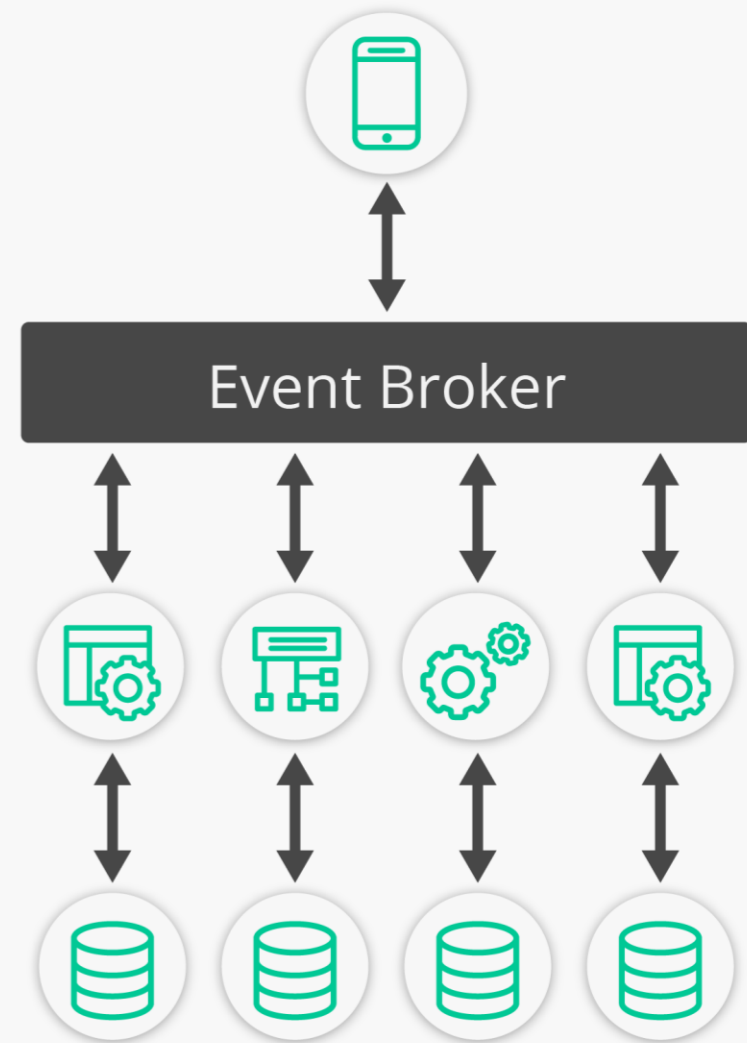
1. Choreography — When there is no central coordination, each service produces and listens to another service's events and decides if an action should be taken or not.
2. Orchestration — An orchestrator (object) takes responsibility for a saga's decision making and sequencing business logic.

## Orchestration



VS

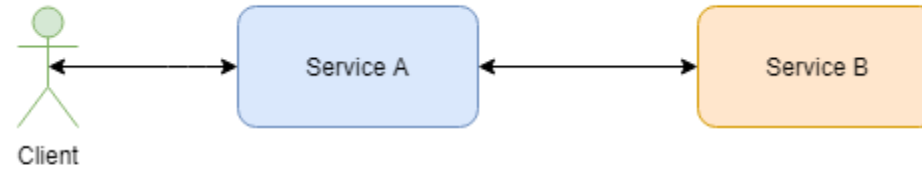
## Choreography



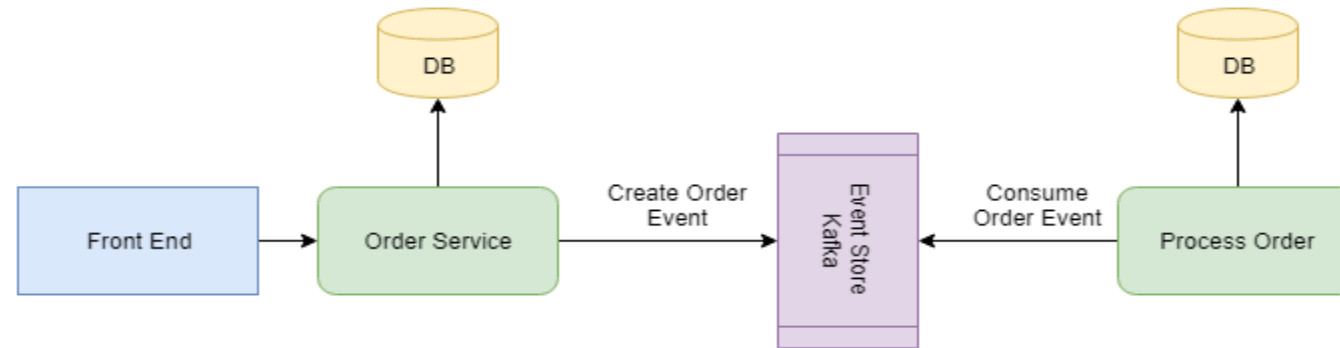


# **Communication Patterns**

Synchronous  
Asynchronous



Synchronous Flow -1



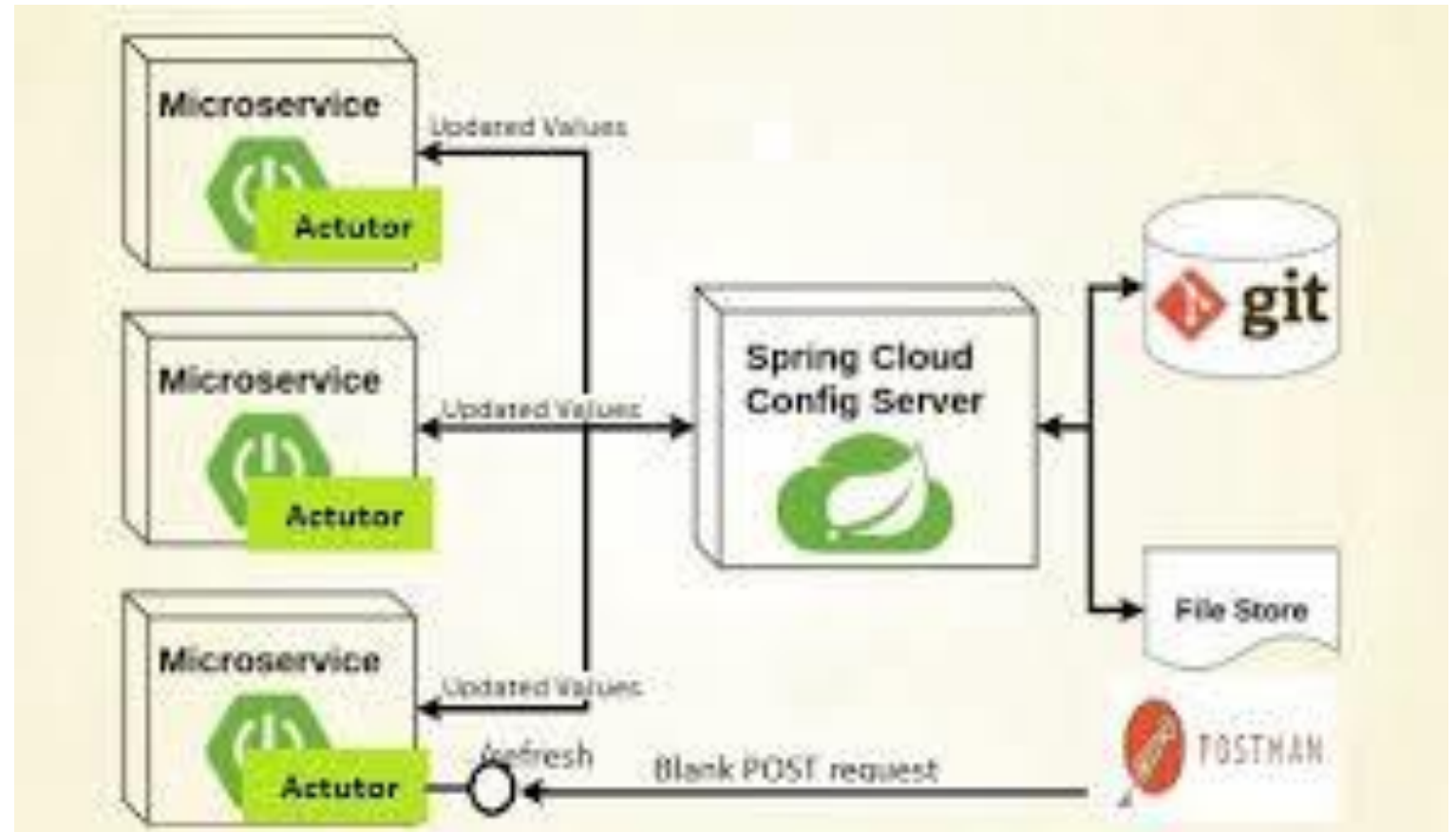
Event Driven - Asynchronous Flow



# **Cross Cutting Concern Patterns**

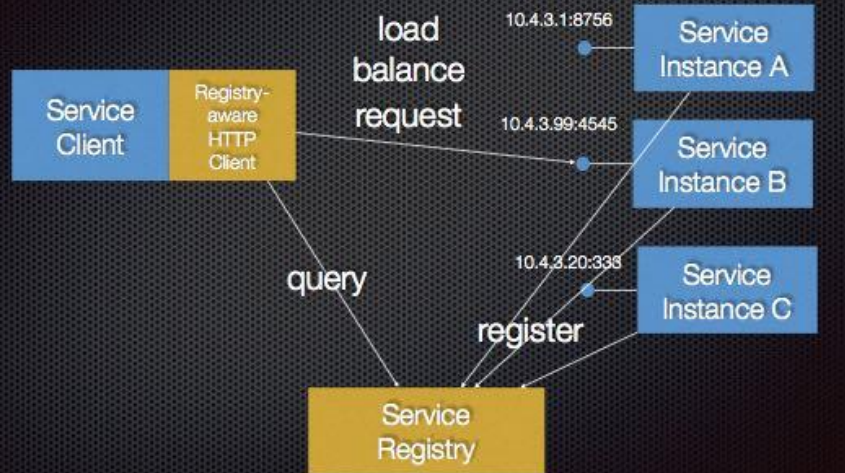


## Spring Cloud Config Server



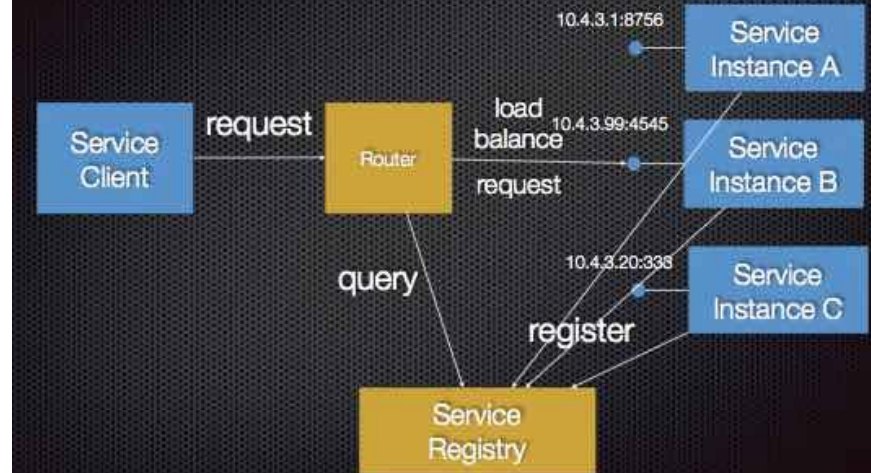
- Service Discovery

### Pattern: Client-side discovery

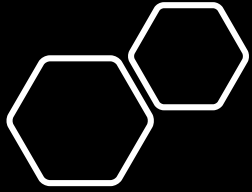


@crichardson

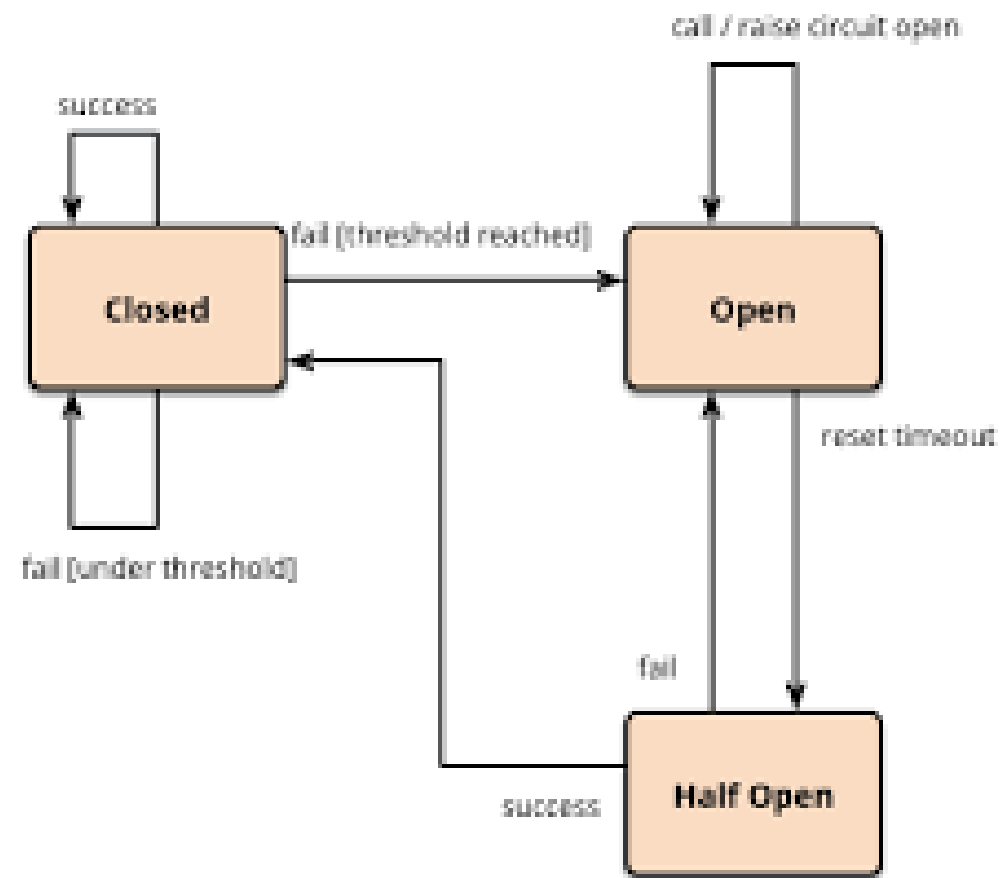
### Pattern: Server-side discovery

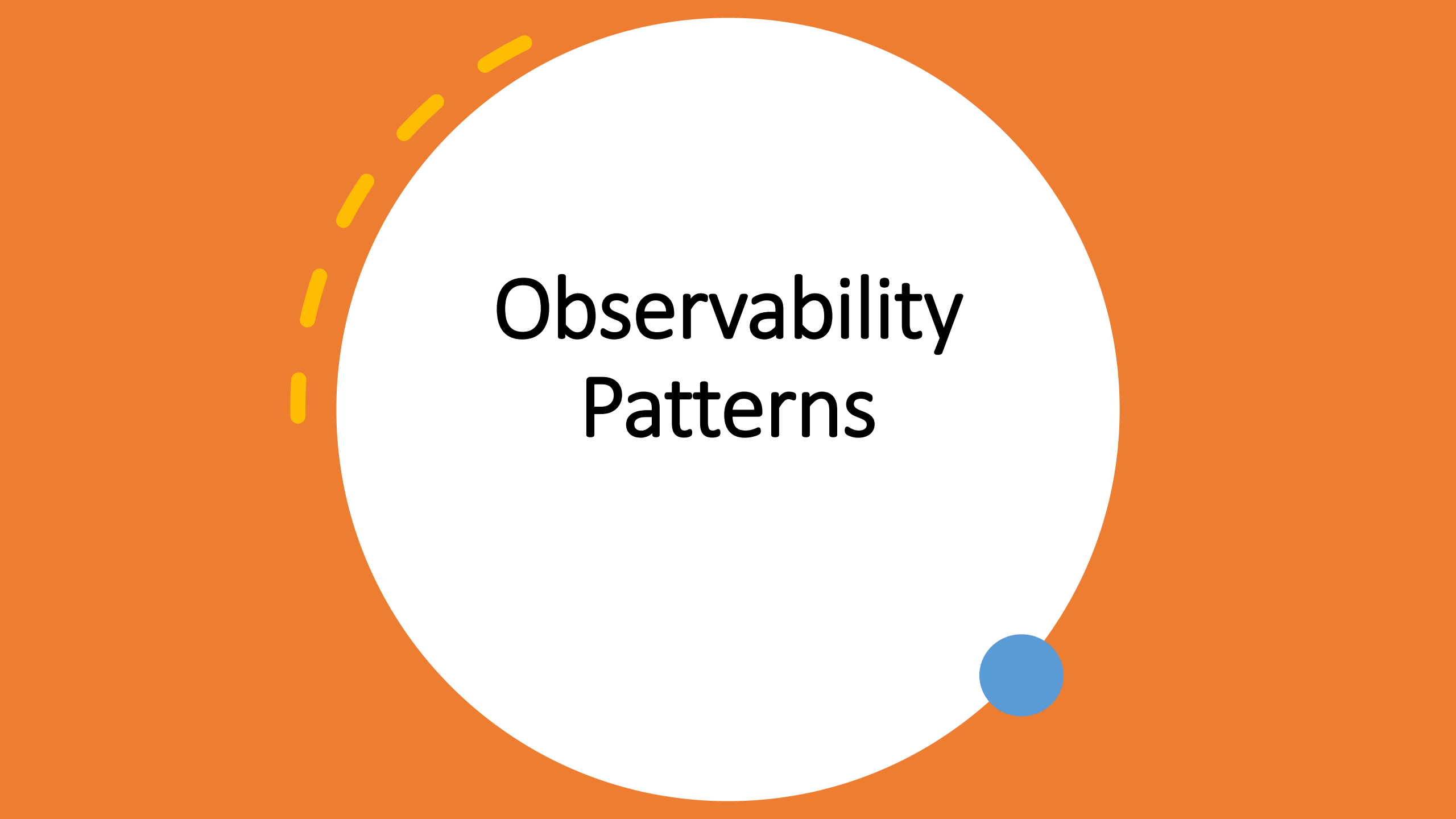


@crichardson



- Circuit Breaker





# Observability Patterns

# Log Aggregation

- **Centralize and Externalize Log Storage**

As microservices are running on multiple hosts, you should send all the generated logs across the hosts to an external, centralized place. From there, you can easily get the log information from one place. It might be another physical system that is highly available or an S3 bucket or any other storage option. If you are hosting your environment on AWS, you can leverage CloudWatch, and other cloud providers generally offer similarly appropriate services.

- **Correlation IDs**

-Generate a correlation ID when you are making the first microservice call and pass that same ID to downstream services. Log the correlation ID across all microservice calls. That way, we can use the correlation ID coming from the response to trace out the logs.

-If you are using Spring Cloud to develop microservices, you can use the [Spring Sleuth](#) module along with [Zipkin](#).

# Health Check

## Problem

When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. In that case, how do you ensure a request doesn't go to those failed instances? With a load balancing pattern implementation.

## Solution

- Each service needs to have an endpoint which can be used to check the health of the application, such as `/health`. This API should check the status of the host, the connection to other services/infrastructure, and any specific logic.
- Spring Boot Actuator does implement a `/health` endpoint and the implementation can be customized, as well.

# Performance Metrics

- **Problem**

How should we collect metrics to monitor application performance?

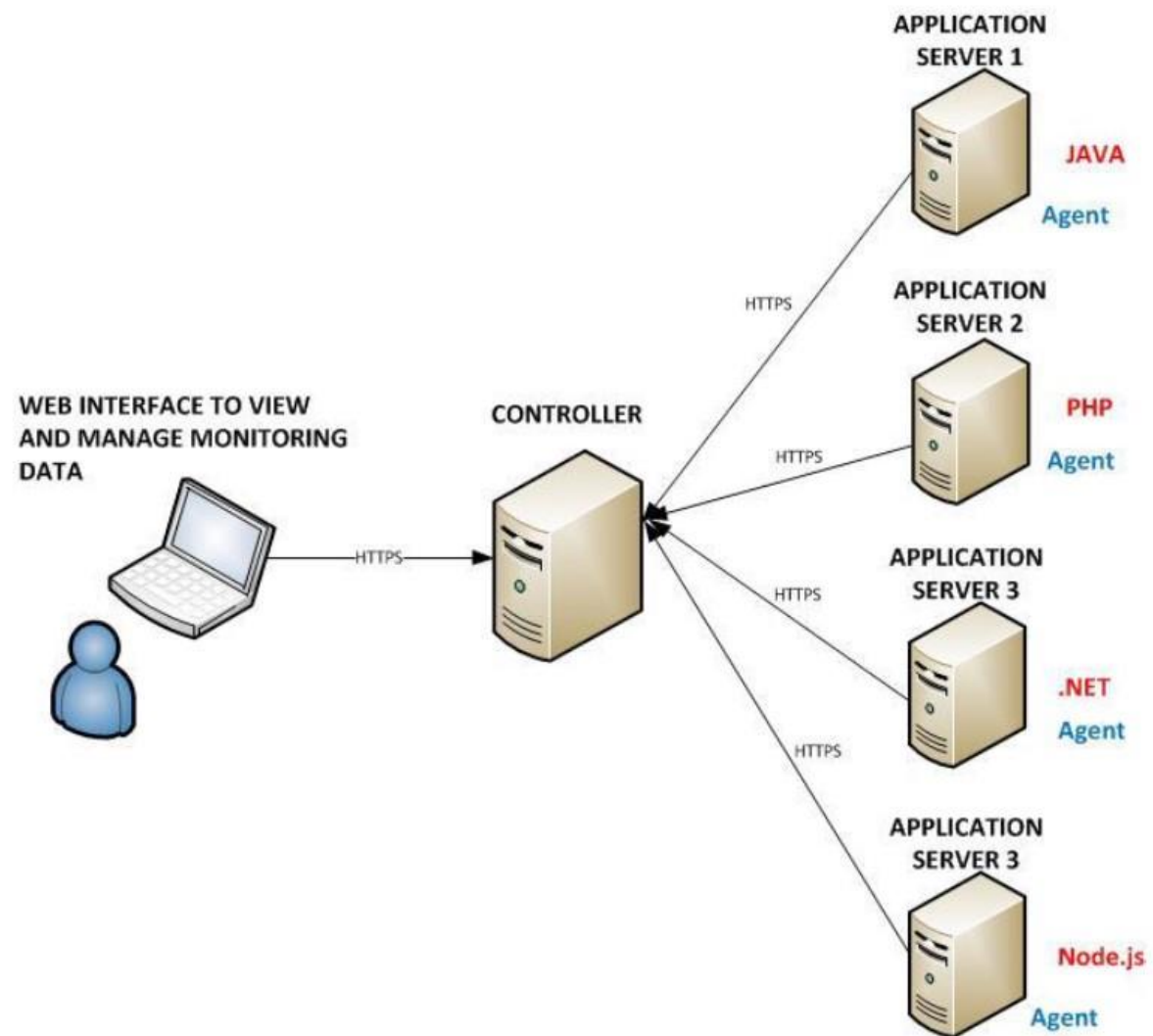
- **Solution**

A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:

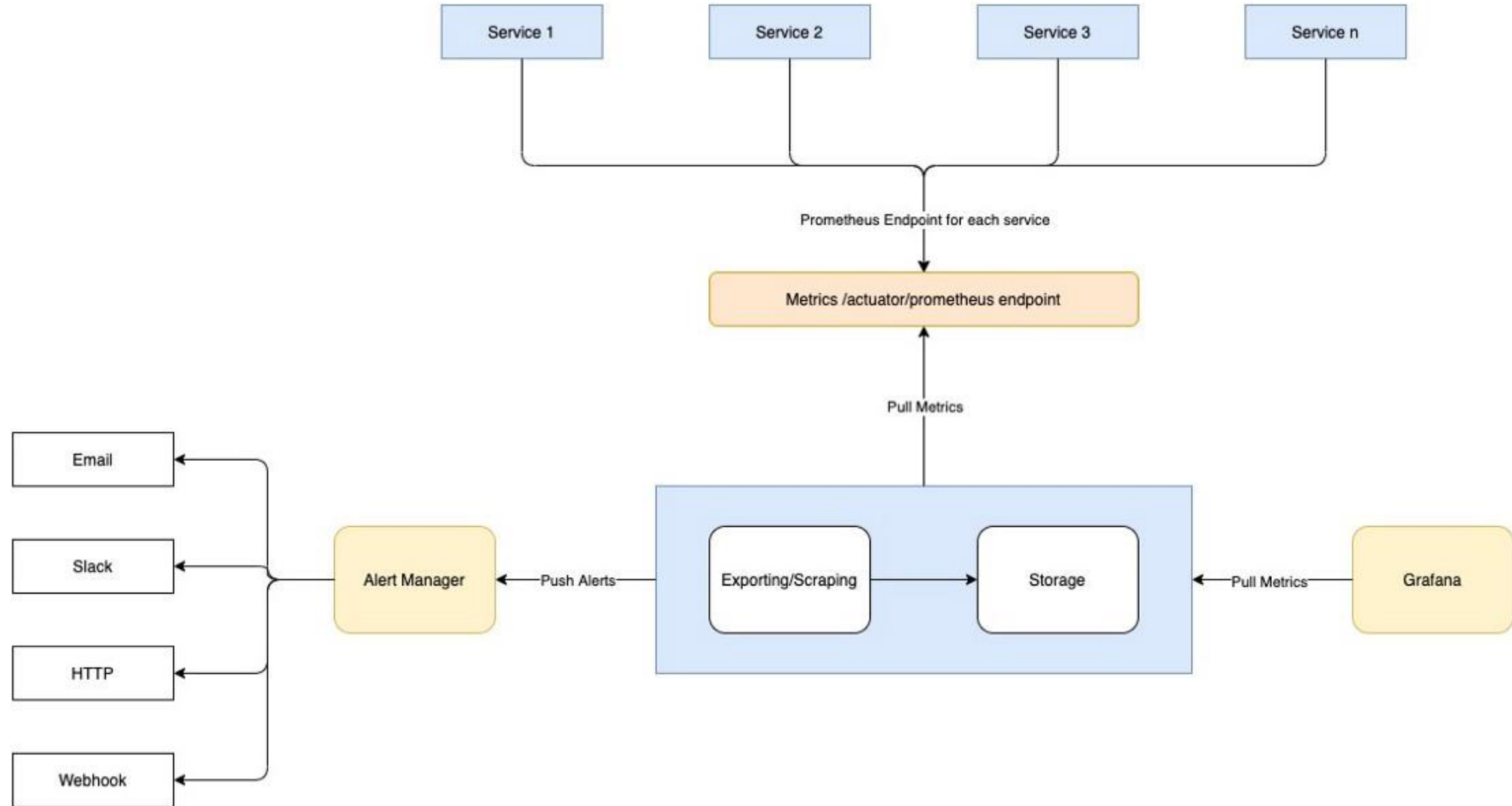
- Push — the service pushes metrics to the metrics service e.g. NewRelic, **AppDynamics**
- Pull — the metrics services pulls metrics from the service e.g. **Prometheus**



## AppDynamics High Level Architecture







# Distributed Tracing

- **Problem**

In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. Then, how do we trace a request end-to-end to troubleshoot the problem?

- **Solution**

- We need a service which

- Assigns each external request a unique external request id.
- Passes the external request id to all services.
- Includes the external request id in all log messages.
- Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

Note: Spring Cloud Sleuth, along with Zipkin server, is a common implementation.



Q&A





Thank You.

