

# Middleware Architectures 1

## Lecture 3: Microservice Architecture

**doc. Ing. Tomáš Vitvar, Ph.D.**

tomas@vitvar.com • @TomasVitvar • <https://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <https://vitvar.com/lectures>



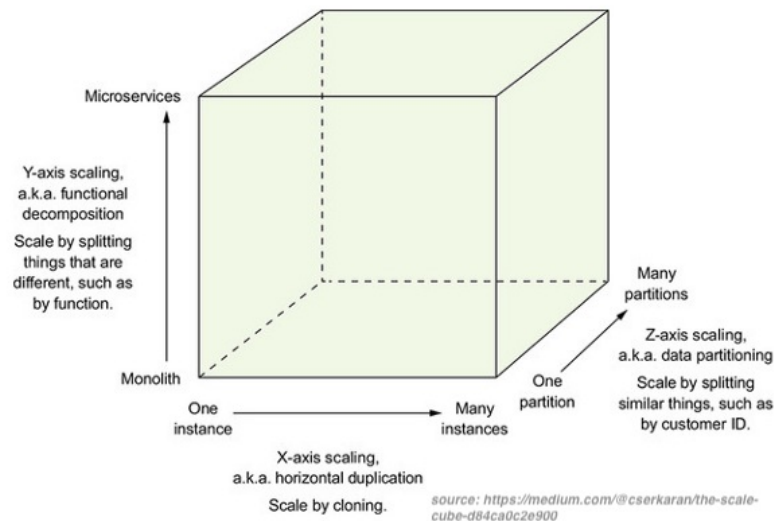
Modified: Sun Oct 26 2025, 20:49:01  
Humla v1.0

## Overview

- **Microservices Architecture**
- Design Patterns

# The Scale Cube

- Three-dimensional scalability model
  - *X-Axis scaling requests across multiple instances*
  - *Y-Axis scaling decomposes an application into micro-services*
  - *Z-Axis scaling requests across "data partitioned" instances*



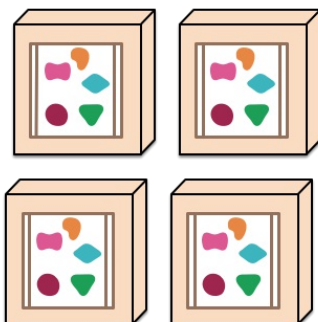
## Overview

- Emerging software architecture
  - *monolithic vs. decoupled applications*
  - *applications as independently deployable services*

A monolithic application puts all its functionality into a single process...



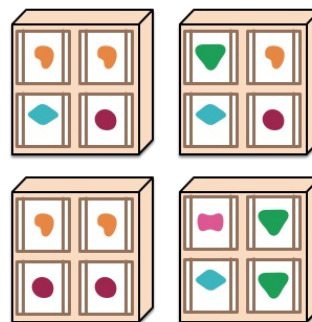
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



## Major Characteristics

- Loosely coupled
  - *Integrated using well-defined interfaces*
- Technology-agnostic protocols
  - *HTTP, they use REST architecture*
- Independently deployable and easy to replace
  - *A change in small part requires to redeploy only that part*
- Organized around capabilities
  - *such as accounting, billing, recommendation, etc.*
- Implemented using different technologies
  - *polyglot – programming languages, databases*
- Owned by a small team

## Overview

- Microservices Architecture
- **Design Patterns**
  - *Data Management Patterns*
  - *Communication Patterns*
  - *Other Patterns*

## Design Patterns

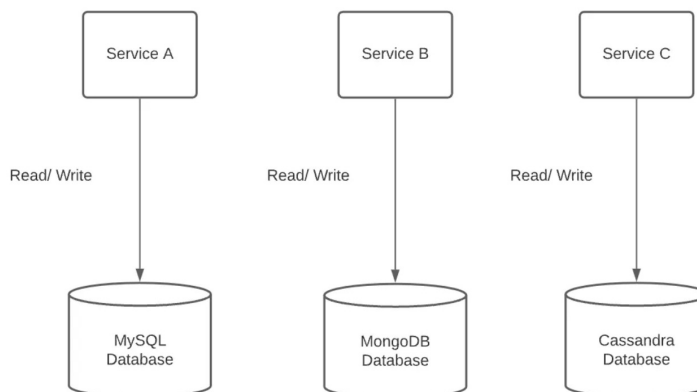
- Data management patterns
  - *Database per service*
  - *Saga pattern*
  - *Command query responsibility segregation (CQRS)*
- Communication patterns
  - *API Gateway*
  - *Aggregator design pattern*
  - *Circuit breaker design pattern*
  - *Sidecar pattern*
- Other patterns
  - *Strangler pattern*

## Overview

- Microservices Architecture
- Design Patterns
  - *Data Management Patterns*
  - *Communication Patterns*
  - *Other Patterns*

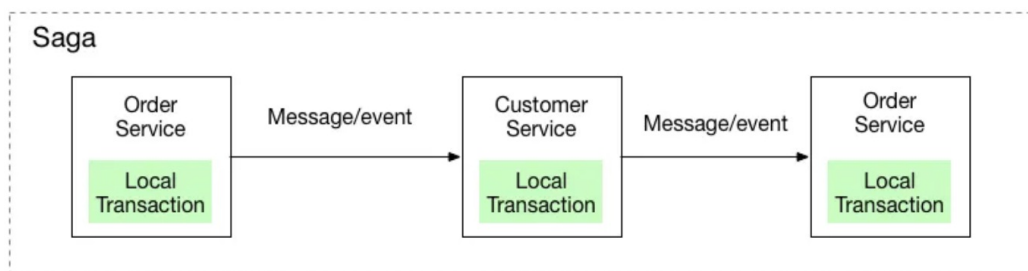
## Database per Service

- Every service has its own database (or at least its own schema)
  - A database dedicated to one service can't be accessed by other services.
  - Decouples services from each other
  - Enables polyglot persistence
    - Different services can use different database technologies
- Challenges
  - Data consistency
  - Complex queries and transactions



## Saga Pattern

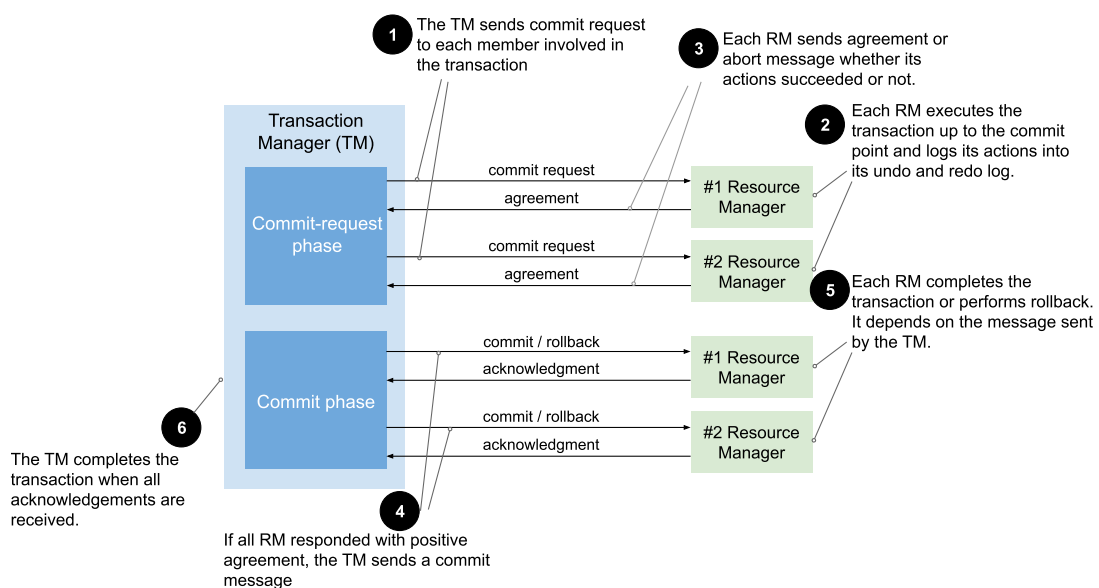
- Manages data consistency across services
  - A series of local transactions
  - This requires **compensating** transactions to undo changes if needed
  - An alternative to Two-phase commit
- Two types of Sagas
  - Choreography-based Sagas
    - Each service produces and listens to events
    - No central coordinator
  - Orchestration-based Sagas
    - Central coordinator (orchestrator) tells the participants what local transactions to execute



## Saga Pattern Examples

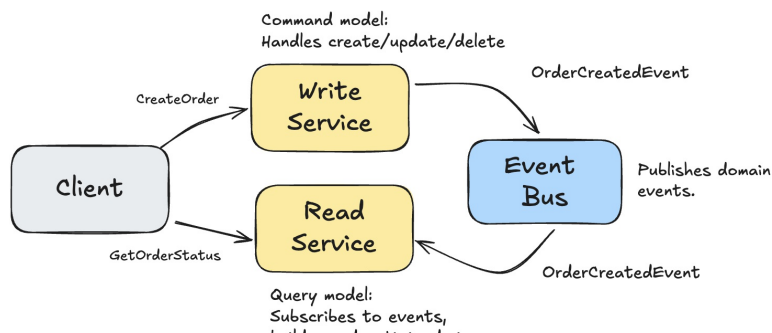
- Example Services
  - Order Service
  - Payment Service
  - Inventory Service
- Choreography (no central coordinator)
  - Order Service → publishes **OrderCreated**
  - Payment Service → listens, reserves funds → publishes **PaymentCompleted**
  - Inventory Service → listens, deducts stock → publishes **InventoryUpdated**
  - Order Service → listens, marks order as **Completed**
- Orchestration (central coordinator)
  - **Orchestrator** → sends **ReservePayment** to Payment Service
  - Payment Service → responds **PaymentConfirmed**
  - **Orchestrator** → sends **ReserveStock** to Inventory Service
  - Inventory Service → responds **StockReserved**
  - **Orchestrator** → calls **CompleteOrder** in Order Service

## Two-phase Commit



## CQRS

- Command Query Responsibility Segregation
- A pattern that separates read and write operations in a system.
  - **Command side:** Handles create/update/delete operations
  - **Query side:** Handles read operations with optimized views
- **Example:** Online Order System
  - User places order → **CreateOrder**
  - Order Service stores order, publishes **OrderCreatedEvent**
  - Read Service updates denormalized **orders\_view**
  - Client queries **GetOrderStatus** → served from **orders\_view**

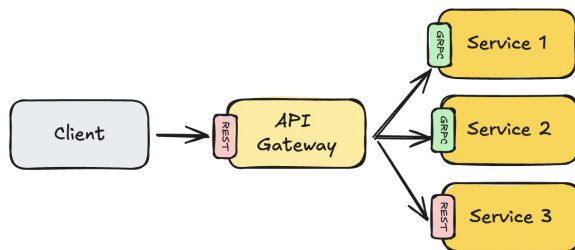


## Overview

- Microservices Architecture
- Design Patterns
  - Data Management Patterns
  - **Communication Patterns**
  - Other Patterns

## API Gateway

- Single entry point for all clients
  - Handles requests by routing them to the appropriate microservice(s)
  - Perform request aggregation, protocol translation, authentication, rate limiting
- Benefits
  - A single entry point for a group of microservices
  - Clients don't need to know how services are partitioned
  - Service boundaries can evolve independently
  - Can implement authentication, TLS termination and caching
- Challenges
  - Potential bottleneck and single point of failure
  - Increased complexity in API Gateway implementation



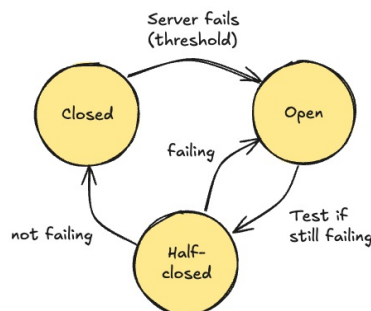
## Aggregator Design Pattern

- Combines data from multiple services into a single response
  - Useful when a client request requires data from multiple microservices
- Benefits
  - Reduces the number of client requests
  - Simplifies client logic
- Challenges
  - Increased complexity in the aggregator service
  - Potential performance bottleneck



## Circuit Breaker

- A service stops trying to execute an operation that is likely to fail
  - Monitors for failures and opens the circuit if failures exceed a threshold
  - When the circuit is **open**, calls to the failing service are blocked for some time
  - After a timeout, the circuit **half-opens** to test if the service has recovered
  - If the test call succeeds, the circuit **closes** and normal operation resumes
- Benefits
  - Improves system resilience and stability
  - Prevents cascading failures in distributed systems
- Challenges
  - Requires careful configuration of thresholds and timeouts



## Circuit Breaker Example

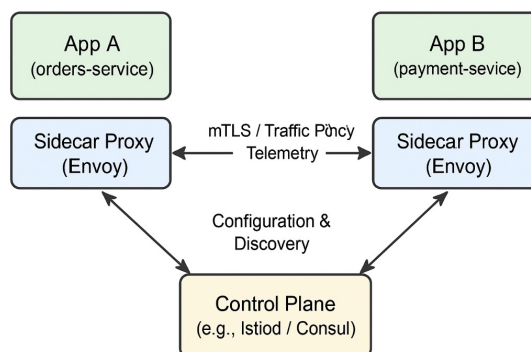
- **Scenario:** Order Service calls Payment Service
  - Under normal conditions → call succeeds, response is fast
  - When Payment Service slows down or fails repeatedly → circuit "opens"
  - Further calls are blocked immediately → fallback response returned
  - After a timeout → circuit "half-opens" to test recovery
  - If test succeeds → circuit "closes" and normal calls resume
- **Example Flow**
  - Order Service → calls Payment API (fails 3×)
  - Circuit opens → returns **"Payment service unavailable"**
  - After 30s → one trial call allowed
  - If trial succeeds → circuit closes and normal traffic resumes

## Sidecar Pattern

- Deploys auxiliary components alongside the main service
  - Handles logging, monitoring, configuration, and networking
  - Runs in a separate process or container but shares the same lifecycle as the main service
- Benefits
  - Decouples auxiliary functionality from the main service
  - Enables reuse of common functionality across multiple services
- Challenges
  - Increased operational complexity
  - Resource overhead due to additional processes/containers

## Sidecar Pattern and Service Mesh

- **Service Mesh:** A dedicated infrastructure layer for managing service-to-service communication
- Service mesh uses sidecar proxies (e.g. Envoy) to manage traffic
- Example: Istio injects Envoy sidecar to handle
  - Service discovery and routing
  - mTLS security
  - Retries, rate limiting, and metrics



## Overview

- Microservices Architecture
- Design Patterns
  - *Data Management Patterns*
  - *Communication Patterns*
  - *Other Patterns*

## Strangler Pattern

- Incrementally replace a monolith with microservices
- New functionality is implemented as microservices
- Existing functionality is gradually "strangled" and replaced
- Benefits
  - *Reduced risk by not rewriting the entire system at once*
  - *Allows for gradual migration and testing*
- Challenges
  - *Complexity in managing both monolith and microservices*
  - *Potential performance overhead during transition*