Harisha Lakshan Warnakulasuriya(BSc.(ousl)) • 2nd
Software Engineer | Designing Innovative Technology for Industrial Sectors | 10+ Years of Expe...
2h •

Follow

1. Define the Business Capabilities (Domain Driven Design)

Identify Core Domains (Order Management, Payment, Inventory, etc.)

Each Microservice should own one clear business responsibility.

> Example: In an E-commerce system, separate services for Cart, Orders, Payments, and User Management.

2. Design Microservices Boundaries

Small, autonomous, and independently deployable services.

Use Bounded Context concepts: no hidden dependencies between services.

3. Choose Communication Style (Sync or Async)

Synchronous (HTTP/REST, gRPC) → Immediate response needed.

Asynchronous (Kafka, RabbitMQ, Event Bus) → Decoupled, scalable.

4. Choose the Tech Stack Per Service

Polyglot Architecture: Services can use different languages/tools.

Examples:

User Service - Python (FastAPI)

Payment Service - Java (Spring Boot)

Recommendation Service - Node.js (Express)

5. Define APIs Clearly (API Contracts)

Use OpenAPI (Swagger), gRPC Protobufs, or GraphQL schemas.

Public API = Contract between services.

6. Set up Service Discovery Mechanism

Services should automatically find each other without hardcoding IPs.

Use tools like:

Eureka

Consul

Kubernetes Internal DNS

7. Implement API Gateway

Single entry point for clients to route traffic to appropriate services.

Responsibilities:

Authentication

Load balancing

Rate limiting

Routing

8. Define Database per Service (Decentralized Data Management)

Each service must own its database.

Avoid sharing databases across services!

> Example: CartService -> MongoDB, OrderService -> PostgreSQL

9. Enable Inter-Service Communication

REST, gRPC, Kafka, RabbitMQ, NATS for messages/events.

Ensure Retries, Timeouts, and Circuit Breakers are implemented.

10. Implement Centralized Authentication and Authorization

Use a centralized Identity Provider (like OAuth2, Keycloak, Auth0).

Tokens (e.g., JWT) to pass identity across services.

11. Build Resiliency (Failure Management)

Retry Mechanism

Circuit Breakers (Hystrix, Resilience4j)

Fallbacks if dependent service fails

12. Deploy in Containers

Use Docker to containerize each service.

Dockerfiles should be minimal, reproducible.

13. Orchestrate with Kubernetes (K8s)

Manage deployments, auto-scaling, service discovery, secret management.

Use Helm Charts or Kustomize for deployment templates.

14. Secure the Microservices

Mutual TLS (mTLS) between services.

API Gateway security (rate-limiting, API keys).

OWASP top 10 vulnerabilities protection.

19. Automate Horizontal Scaling

Scale services independently based on:

CPU utilization

Message queue length

Traffic spikes

Bonus: Visual Simple Flow Diagram

Client (Browser, App)

↓

API Gateway

↓

Service Discovery

↓

Microservices (Cart, Order, Payment, Shipping, etc.)

↓

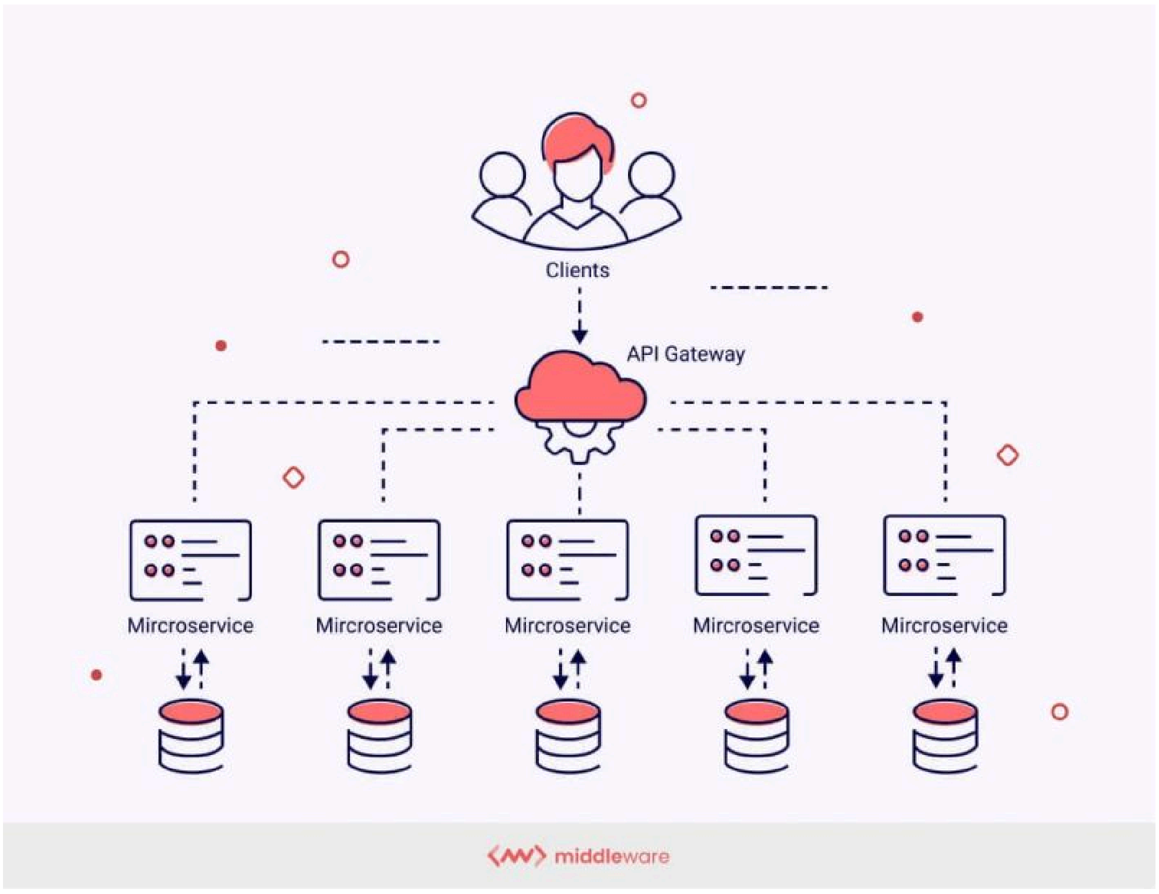
Each Microservice's Database (Postgres, MongoDB, Redis, etc.)

↘️ ↙

Event Bus (Kafka/NATS) for Asynchronous Communication

↘️ ↙

Centralized Logging, Monitoring, Tracing



👍 4