Microservices Architecture Cheatsheet

A concise reference guide to microservices architecture, covering key concepts, design principles, communication patterns, deployment strategies, and related technologies. This cheatsheet provides a quick overview for architects, developers, and DevOps engineers working with microservices.

Core Concepts

Definition

Microservices: An architectural style that structures an application as a collection of small, autonomous services, modeled around a business domain.
Each service is self-contained and implements a single business capability.
Services communicate through well-defined APIs, often over a network.

Key Principles

Single Responsibility Principle (SRP)	Each service should have one reason to change; focus on a single business capability.
Autonomy	Services should be independently deployable, scalable, and replaceable.
Decentralized Governance	Services can choose their own technology stack, data store, and deployment strategy.
Fault Isolation	Failure of one service should not cascade to other services. Implement circuit breakers and bulkheads.
API-First Design	Design services with well-defined, versioned APIs that are easy to consume.

Benefits

- **Improved Scalability:** Scale individual services based on their specific needs.
- **Increased Agility:** Faster development and deployment cycles.
- **Technology Diversity:** Choose the right technology for each service.
- **Fault Isolation:** Isolate failures and prevent cascading issues.
- **Easier Understanding:** Smaller codebases are easier to understand and maintain.

Communication Patterns

Synchronous Communication

REST (Representational State Transfer)	A widely used architectural style for building web services. Uses standard HTTP methods (GET, POST, PUT, DELETE).
gRPC (gRPC Remote Procedure Calls)	A high-performance, open-source RPC framework developed by Google. Uses Protocol Buffers for serialization.
GraphQL	A query language for your API and a server-side runtime for executing those queries. Clients request only the data they need.

Asynchronous Communication

Message Queues (e.g., RabbitMQ, Kafka)	Enable asynchronous communication between services. Messages are placed in a queue and consumed by other services.
Event-Driven Architecture	Services publish events when something significant happens. Other services subscribe to these events and react accordingly.
Message Brokers	Centralized hub that routes messages between microservices, enabling decoupling and scalability.

API Gateway

Acts as a single entry point for all client requests. Handles routing, authentication, authorization, and rate limiting.

Example: Kong, Tyk, Apigee

Benefits:

- Decouples clients from the underlying microservice architecture.
- Provides a unified interface for clients.
- Enables cross-cutting concerns such as security and monitoring.

Deployment & Infrastructure

Containerization

Containers (e.g., Docker) provide a lightweight and portable way to package and deploy microservices.
Containers encapsulate the service and its dependencies, ensuring consistency across different environments.

Orchestration

Kubernetes	A popular open-source container orchestration platform. Automates deployment, scaling, and management of containerized applications.
Docker Swarm	Docker's native container orchestration tool. Simpler to set up than Kubernetes but less feature-rich.
Serverless Computing	Deploy microservices as functions that are triggered by events. (e.g., AWS Lambda, Azure Functions).

Service Discovery

Services need a way to discover the location of other services. Service discovery mechanisms provide a dynamic registry of service instances.

Examples:

- Consul
- Etcd
- ZooKeeper

Monitoring & Logging

Centralized Logging	Aggregate logs from all services into a central location for analysis. (e.g., ELK stack, Splunk).
Metrics Collection	Collect metrics about service performance and health. (e.g., Prometheus, Grafana).
Distributed Tracing	Track requests as they flow through multiple services. (e.g., Jaeger, Zipkin).

Design Considerations

Domain-Driven Design (DDD)

DDD is an approach to software development that focuses on modeling the domain. Align microservices with bounded contexts in your domain model.
Use DDD concepts like entities, value objects, and aggregates to design cohesive and loosely coupled services.

Data Management

Database per Service	Each service should own its own database. This ensures data autonomy and prevents tight coupling.
Shared Database (Anti-Pattern)	Avoid sharing databases between services. This can lead to tight coupling and contention.
Eventual Consistency	Data consistency across services is often eventual. Use techniques like sagas to manage transactions across services.

Security

Implement authentication and authorization for all services. Use standards like OAuth 2.0 and OpenID Connect.
Secure communication between services using TLS. Consider using a service mesh like Istio for managing security policies.

Testing

Unit Tests	Test individual components of a service.
Integration Tests	Test interactions between services.
End-to-End Tests	Test the entire system from end to end.