# System Design Patterns

System design involves creating the architecture of software systems to meet specific requirements. Key considerations include scalability, reliability, availability, and performance.

Microservices architecture decomposes applications into small, independent services that communicate via APIs. Each service is responsible for a specific business capability and can be developed, deployed, and scaled independently.

Load balancing distributes incoming network traffic across multiple servers. Common algorithms include round-robin, least connections, and weighted distribution. Hardware and software load balancers serve different use cases.

Caching is a technique to store frequently accessed data in fast storage. Strategies include write-through, write-back, and write-around. Cache invalidation remains one of the hardest problems in computer science.

Database sharding horizontally partitions data across multiple database instances. Sharding strategies include range-based, hash-based, and directory-based sharding. Each approach has tradeoffs in query complexity and data distribution.

# Distributed Systems

The CAP theorem states that a distributed system cannot simultaneously guarantee Consistency, Availability, and Partition tolerance. In practice, systems must choose between CP and AP when network partitions occur.

Consensus algorithms like Raft and Paxos enable distributed systems to agree on values even when some nodes fail. Raft is designed to be more understandable than Paxos while providing the same guarantees.

Message queues like Kafka, RabbitMQ, and SQS enable asynchronous communication between services. They provide durability, ordering guarantees, and backpressure handling for distributed architectures.

Event sourcing stores all changes to application state as a sequence of events. Instead of storing current state, the system stores the complete history of state transitions, enabling audit trails and temporal queries.

Circuit breakers prevent cascading failures in distributed systems. When a service fails repeatedly, the circuit breaker trips, preventing further calls and allowing the failing service time to recover.

# API Design Best Practices

RESTful APIs use HTTP methods semantically: GET for retrieval, POST for creation, PUT for full updates, PATCH for partial updates, and DELETE for removal. Resources are identified by URIs.

GraphQL provides a flexible query language for APIs. Clients specify exactly what data they need, eliminating over-fetching and under-fetching. Schema definitions provide strong typing and self-documentation.

Rate limiting protects APIs from abuse. Token bucket and sliding window algorithms are commonly used. Rate limits should be communicated via HTTP headers like X-RateLimit-Limit and X-RateLimit-Remaining.

API versioning strategies include URI versioning (/v1/users), header versioning, and query parameter versioning. Each approach has different implications for backward compatibility and client migration.

Authentication mechanisms include API keys, OAuth 2.0, JWT tokens, and mutual TLS. The choice depends on security requirements, client types, and the sensitivity of the data being accessed.