## **1. Scalability**

### **Horizontal vs Vertical Scaling Evolution**

**Traditional View**: Scale up (vertical) vs scale out (horizontal)

**Modern Reality**: Hybrid approaches with auto-scaling orchestration

#### **Advanced Scaling Patterns:**

* **Elastic Scaling**: Dynamic resource allocation based on predictive algorithms
* **Multi-dimensional Scaling**: Scaling different components independently (CPU, memory, I/O, network)
* **Geographic Scaling**: Edge computing with CDN integration for global distribution

### **Load Balancing Innovation**

**Beyond Round-Robin**:

* **Consistent Hashing**: Minimizes redistribution when nodes are added/removed
* **Weighted Least Connections**: Accounts for server capacity differences
* **Geographic Load Balancing**: Route based on user location and server health
* **Application-Aware Load Balancing**: Routes based on request characteristics

## **2. Caching Strategies**

### **Multi-Layer Caching Architecture**

Browser Cache → CDN → Reverse Proxy → Application Cache → Database Cache

### **Cache Patterns Deep Dive**

#### **Cache-Aside (Lazy Loading)**

if data not in cache:  
 data = fetch\_from\_database()  
 cache.set(key, data, ttl)  
return data

**Pros**: Only cache what's needed, cache failure doesn't break app

**Cons**: Cache miss penalty, potential for stale data

#### **Write-Through**

cache.set(key, data)  
database.save(data)

**Pros**: Data consistency, no cache miss penalty

**Cons**: Write latency, unnecessary caching

#### **Write-Behind (Write-Back)**

cache.set(key, data)  
async\_queue.add(database\_write\_task)

**Pros**: Improved write performance, batching opportunities

**Cons**: Data loss risk, complexity

#### **Refresh-Ahead**

if cache\_entry.time\_to\_live < threshold:  
 async\_refresh\_cache(key)

**Pros**: Prevents cache miss storms, consistent performance

**Cons**: Resource overhead, complexity

### **Cache Invalidation Strategies**

**The Two Hard Problems**: Cache invalidation, naming things, and off-by-one errors

#### **Time-Based Expiration (TTL)**

* Simple but may serve stale data
* Good for data that changes predictably

#### **Event-Based Invalidation**

* Invalidate on data changes
* Requires robust event system
* Risk of invalidation failures

#### **Version-Based Invalidation**

* Each cache entry has a version
* Compare versions before serving
* Handles partial failures better

## **3. CAP Theorem**

### **CAP Theorem Fundamentals**

**Consistency**: All nodes see the same data simultaneously

**Availability**: System remains operational

**Partition Tolerance**: System continues despite network failures

**\*Reality**: You can only guarantee 2 out of 3 during network partitions

### **CAP in Practice**

#### **CP Systems (Consistency + Partition Tolerance)**

* **Examples**: HBase, MongoDB (default), Redis Cluster
* **Trade-off**: Sacrifice availability during partitions
* **Use Case**: Financial systems, inventory management

#### **AP Systems (Availability + Partition Tolerance)**

* **Examples**: Cassandra, DynamoDB, CouchDB
* **Trade-off**: Accept temporary inconsistency
* **Use Case**: Social media, content delivery, analytics

#### **CA Systems (Consistency + Availability)**

* **Examples**: Traditional RDBMS in single data center
* **Reality**: Doesn't handle network partitions well
* **Limited Use**: Single-node or perfect network assumptions

## **4. Microservices Fundamentals & Evolution**

### **Definition & Core Principles**

Microservices architecture structures an application as a collection of loosely coupled services that are:

* **Independently deployable**
* **Business capability focused**
* **Decentralized**
* **Fault tolerant**
* **Technology agnostic**

### **Best Practices**

* API Gateway
* Centralized auth (e.g., Cognito/OAuth)
* Containerization: Docker + Orchestration: Kubernetes

### **Real-World Example**

**Netflix**

* Each microservice handles specific functionality like billing, recommendation, streaming
* Uses Eureka for service discovery
* Asynchronous messaging with Kafka