The Complete System Design & Engineering Bible

A Comprehensive Guide from Fundamentals to Advanced Concepts

Table of Contents

PART I: FOUNDATIONS

1. Introduction to System Design

What is System Design?

System design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. Think of it as creating a blueprint for a software system, much like an architect creates blueprints for a building.

Why System Design Matters

In the early days of computing, applications were simple and served few users. Today's applications must:

- Handle millions of users simultaneously
- Process vast amounts of data
- Remain available 24/7

- Scale globally
- Adapt to changing requirements

The Evolution of System Design

1960s-1980s: Mainframe Era

- Centralized computing
- Single point of failure
- Limited scalability

1990s-2000s: Client-Server Era

- Distributed computing emerges
- Three-tier architecture becomes standard
- Internet applications grow

2010s-Present: Cloud and Microservices Era

- Horizontal scaling
- Cloud computing
- Microservices architecture
- DevOps and continuous deployment

Key Stakeholders in System Design

Engineers: Implement the technical aspects **Product Managers**: Define requirements and priorities **Architects**: Design the overall system structure **Operations Teams**: Ensure system reliability and performance **Business Leaders**: Define success metrics and constraints

2. Basic Software Architecture Patterns

MVC (Model-View-Controller)

The MVC pattern separates an application into three interconnected components:

Model: Manages data and business logic

- Represents the data structure
- Handles data validation
- Communicates with databases
- Implements business rules

View: Handles the presentation layer

- Displays data to users
- Captures user input
- Formats information for display
- Provides user interface

Controller: Manages user input and coordinates between Model and View

- Receives user requests
- Processes input
- Updates the Model
- Selects appropriate View

Example: E-commerce Shopping Cart

User clicks "Add to Cart" → Controller receives request →

Controller asks Model to add item → Model updates cart data →

Controller tells View to refresh → View shows updated cart

Benefits of MVC:

- Separation of concerns
- Easier testing
- Code reusability
- Multiple views for same data
- Easier maintenance

MVP (Model-View-Presenter)

Similar to MVC but with different responsibilities:

Model: Same as MVC - handles data and business logic **View**: Passive interface that displays data **Presenter**: Handles all UI logic and acts as intermediary

Key Difference from MVC: The View is completely passive and doesn't directly communicate with the Model.

MVVM (Model-View-ViewModel)

Popular in modern UI frameworks:

Model: Business logic and data **View**: User interface **ViewModel**: Binding layer between View and Model

Benefits:

- Two-way data binding
- Better testability
- Clear separation of UI and business logic

Three-Tier Architecture

A fundamental architectural pattern that organizes applications into three logical layers:

Presentation Tier (Client Layer)

- User interface components
- Web browsers, mobile apps, desktop applications
- Handles user interactions
- Formats data for display

Application Tier (Middle Tier/Business Logic Layer)

- Business logic and rules
- Application servers
- Processes user requests
- Coordinates between presentation and data tiers

Data Tier (Database Layer)

- Data storage and retrieval
- Database management systems
- File systems
- Data warehouses

Example: Online Banking System

- Presentation: Web interface for account access
- Application: Transaction processing, account validation
- Data: Customer accounts, transaction history storage

Benefits:

- Scalability: Each tier can be scaled independently
- Security: Sensitive data is isolated in the data tier
- Maintainability: Changes in one tier don't affect others
- Flexibility: Different technologies can be used for each tier

3. Understanding Requirements

Functional Requirements

Define what the system should do:

Core Features

- User authentication and authorization
- Data processing capabilities
- Business logic implementation
- Integration requirements

User Stories Format "As a [type of user], I want [functionality] so that [benefit]"

Example: "As a customer, I want to search for products so that I can find items I want to purchase"

Non-Functional Requirements

Define how the system should perform:

Performance Requirements

- Response time: How quickly should the system respond?
- Throughput: How many requests per second?
- Latency: Maximum acceptable delay

Scalability Requirements

- Expected number of users
- Data volume growth projections
- Geographic distribution needs

Reliability Requirements

- Uptime expectations (99.9%, 99.99%)
- Disaster recovery needs
- Backup requirements

Security Requirements

- Authentication methods
- Data encryption needs
- Compliance requirements (GDPR, HIPAA)

Requirements Gathering Techniques

Stakeholder Interviews

- Identify key stakeholders
- Conduct structured interviews
- Document requirements systematically

Use Case Analysis

- Define system boundaries
- Identify actors and their interactions
- Map out scenarios

Prototyping

- Create mockups or wireframes
- Validate requirements with stakeholders
- Iterate based on feedback

4. System Design Principles

SOLID Principles

Single Responsibility Principle (SRP) Every class should have only one reason to change.

Example: Separate user authentication from user profile management.

Open/Closed Principle (OCP) Software entities should be open for extension but closed for modification.

Example: Use interfaces to allow new payment methods without changing existing code.

Liskov Substitution Principle (LSP) Derived classes must be substitutable for their base classes.

Interface Segregation Principle (ISP) Clients should not be forced to depend on interfaces they don't use.

Dependency Inversion Principle (DIP) Depend on abstractions, not concretions.

DRY (Don't Repeat Yourself)

Avoid code duplication by:

- Creating reusable functions
- Using configuration files
- Implementing shared libraries
- Creating templates and patterns

KISS (Keep It Simple, Stupid)

Favor simplicity:

- Choose simple solutions over complex ones
- Avoid premature optimization
- Use clear and readable code
- Minimize dependencies

YAGNI (You Aren't Gonna Need It)

Don't implement features until they're actually needed:

- Focus on current requirements
- Avoid speculative development
- Implement incrementally
- Refactor when necessary

PART II: CORE CONCEPTS

5. Scalability Fundamentals

Understanding Scale

Small Scale (1-1000 users)

- Single server can handle everything
- Simple database setup
- Minimal caching needed
- Basic monitoring sufficient

Medium Scale (1K-100K users)

- Need multiple servers
- Database optimization required
- Caching becomes important
- Load balancing necessary

Large Scale (100K-10M users)

- Distributed systems required
- Multiple data centers
- Advanced caching strategies

- Sophisticated monitoring

Internet Scale (10M+ users)

- Global distribution
- Complex distributed systems
- Advanced optimization techniques
- Massive infrastructure

Vertical Scaling (Scaling Up)

Adding more power to existing machines:

Advantages:

- Simple to implement
- No application changes needed
- Easier to manage
- Better for applications requiring shared state

Disadvantages:

- Hardware limits
- Expensive high-end hardware
- Single point of failure
- Downtime required for upgrades

When to Use:

- Early stages of application
- Applications with complex shared state
- Budget constraints for development time
- Legacy applications

Horizontal Scaling (Scaling Out)

Adding more machines to the pool of resources:

Advantages:

- No theoretical limit
- Cost-effective using commodity hardware
- Fault tolerance through redundancy

- Can scale specific components independently

Disadvantages:

- Application complexity increases
- Need to handle distributed system challenges
- Data consistency becomes complex
- More operational overhead

Implementation Strategies:

- Stateless application design
- Database sharding
- Load balancing
- Distributed caching

Auto-Scaling

Automatically adjusting resources based on demand:

Reactive Scaling

- Monitor metrics (CPU, memory, requests)
- Scale when thresholds are exceeded
- Risk of being too late for traffic spikes

Predictive Scaling

- Use historical data and machine learning
- Scale before demand increases
- More cost-effective
- Requires good data and models

Scheduled Scaling

- Scale based on known patterns
- Good for predictable traffic
- Business hours, seasonal patterns
- Combined with reactive scaling

6. Database Design and Management

Relational Databases (SQL)

ACID Properties:

- **Atomicity**: Transactions are all-or-nothing
- **Consistency**: Database remains in valid state
- **Isolation**: Concurrent transactions don't interfere
- **Durability**: Committed transactions survive system failures

When to Use SQL Databases:

- Complex relationships between data
- ACID compliance required
- Complex queries and reporting
- Well-defined schema

Popular SQL Databases:

- PostgreSQL: Advanced features, extensible
- MySQL: Fast, widely used
- Oracle: Enterprise features
- SQL Server: Microsoft ecosystem

NoSQL Databases

Document Databases Store data as documents (JSON, XML):

- MongoDB: Flexible schema, rich queries
- CouchDB: Multi-master replication
- Amazon DocumentDB: Managed MongoDB alternative

Key-Value Stores Simple key-value pairs:

- Redis: In-memory, very fast
- Amazon DynamoDB: Managed, scalable
- Riak: Distributed, fault-tolerant

Column-Family Store data in column families:

- Cassandra: Highly scalable, no single point of failure
- HBase: Hadoop ecosystem integration

Graph Databases Optimized for relationships:

- Neo4j: Popular graph database

- Amazon Neptune: Managed graph database

Database Sharding

Splitting database across multiple servers:

Horizontal Sharding Split rows across databases:

- Range-based: Users A-M on DB1, N-Z on DB2
- Hash-based: Hash user ID to determine database
- Directory-based: Lookup service maps keys to shards

Vertical Sharding Split different tables/features:

- User profile data on one database
- Order data on another database
- Product catalog on third database

Challenges:

- Cross-shard queries are complex
- Rebalancing shards is difficult
- Maintaining referential integrity
- Increased operational complexity

Replication Strategies

Master-Slave Replication

- One master handles writes
- Multiple slaves handle reads
- Slaves replicate from master
- Good for read-heavy workloads

Master-Master Replication

- Multiple masters can handle writes
- More complex conflict resolution
- Better availability
- Risk of conflicting writes

Eventual Consistency

- Changes propagate over time

- System remains available during network partitions
- Good for distributed systems
- Requires application-level conflict resolution

7. Caching Strategies

Cache Hierarchy

Browser Cache

- Stores static resources locally
- Reduces server requests
- Controlled by HTTP headers
- User can clear cache

CDN (Content Delivery Network)

- Geographic distribution of content
- Caches static assets globally
- Reduces latency for users
- Examples: CloudFlare, CloudFront

Application-Level Cache

- In-memory data storage
- Frequently accessed data
- Redis, Memcached
- Shared across application instances

Database Cache

- Query result caching
- Reduces database load
- Built into database systems
- Can be application-managed

Caching Patterns

Cache-Aside (Lazy Loading)

```
if data not in cache:
  data = fetch from database
```

store data in cache

return data from cache

Write-Through

write data to cache

write data to database

Write-Behind (Write-Back)

write data to cache immediately

write data to database asynchronously

Refresh-Ahead

proactively refresh cache before expiration

based on access patterns

Cache Invalidation

TTL (Time-To-Live)

- Set expiration time for cache entries
- Simple but may serve stale data
- Good for data that changes infrequently

Event-Based Invalidation

- Invalidate cache when data changes
- More complex but more accurate
- Requires event system

Manual Invalidation

- Explicit cache clearing
- Used for critical updates
- Requires careful coordination

Distributed Caching

Consistent Hashing

- Distributes cache keys across nodes
- Minimizes redistribution when nodes change
- Used by systems like Memcached clusters

Cache Replication

- Replicate cache data across nodes
- Increases availability
- Higher memory usage

Cache Partitioning

- Divide cache across nodes
- Each node stores subset of data
- Better memory utilization

8. Load Balancing

Load Balancer Types

Layer 4 (Transport Layer)

- Routes based on IP and port
- Fast and simple
- Protocol agnostic
- Limited routing intelligence

Layer 7 (Application Layer)

- Routes based on application data
- Content-based routing
- SSL termination
- More intelligent but slower

Load Balancing Algorithms

Round Robin

- Requests distributed sequentially
- Simple and fair
- Doesn't consider server capacity
- Good for homogeneous servers

Weighted Round Robin

- Assigns weights based on server capacity
- More requests to powerful servers

- Still doesn't consider current load

Least Connections

- Routes to server with fewest active connections
- Good for varying request processing times
- Requires connection tracking

Least Response Time

- Routes to server with fastest response
- Considers both load and performance
- More complex to implement

IP Hash

- Routes based on client IP hash
- Ensures same client goes to same server
- Good for stateful applications
- Can cause uneven distribution

Session Management

Sticky Sessions (Session Affinity)

- Route user to same server
- Maintains server-side session state
- Reduces scalability
- Single point of failure per user

Session Replication

- Replicate session across servers
- Higher availability
- Increased network traffic
- Memory overhead

External Session Store

- Store sessions in shared database/cache
- Stateless application servers
- Better scalability

- Single point of failure for session store

Stateless Design

- No server-side session state
- All state in client or database
- Maximum scalability
- Requires application redesign

PART III: ADVANCED ARCHITECTURE

9. Microservices vs Monolithic Architecture

Monolithic Architecture

Definition: A single deployable unit containing all application functionality.

Structure:

- Single codebase
- Single database
- Single deployment
- Shared runtime environment

Advantages:

- **Simple Development**: Easy to develop, test, and deploy initially
- **Simple Deployment**: Single artifact to deploy
- **Easy Testing**: End-to-end testing is straightforward
- **Performance**: No network latency between components
- **Easier Debugging**: All code in one place

Disadvantages:

- **Technology Lock-in**: Entire application uses same technology stack
- **Scaling Challenges**: Must scale entire application, not individual components
- **Development Bottlenecks**: Large teams working on same codebase
- **Reliability**: Single point of failure can bring down entire application
- **Deployment Risk**: Small changes require full application deployment

When to Use Monoliths:

- Small teams (< 10 developers)
- Early-stage applications
- Simple applications with well-defined boundaries
- Applications with tight coupling requirements
- Organizations new to distributed systems

Microservices Architecture

Definition: Architecture style that structures an application as a collection of small, autonomous services.

Characteristics:

- **Single Responsibility**: Each service has one business function
- **Independently Deployable**: Services can be deployed separately
- **Decentralized**: No central orchestration
- **Technology Agnostic**: Each service can use different technologies
- **Fault Isolated**: Failure in one service doesn't crash others

Advantages:

- **Independent Scaling**: Scale services based on demand
- **Technology Diversity**: Use best tool for each job
- **Team Autonomy**: Teams can work independently
- **Fault Isolation**: Failures are contained
- **Faster Deployment**: Deploy services independently

Disadvantages:

- **Distributed System Complexity**: Network calls, latency, failures
- **Data Consistency**: Managing transactions across services
- **Operational Overhead**: More services to monitor and maintain
- **Service Discovery**: How services find each other
- **Testing Complexity**: Integration testing is more complex

When to Use Microservices:

- Large, complex applications
- Multiple teams working on same product

- Different parts of application have different scaling needs
- Organization has strong DevOps capabilities
- Clear service boundaries can be defined

Microservices Design Patterns

Database per Service

- Each microservice has its own database
- Ensures loose coupling
- Enables independent scaling
- Challenges with distributed transactions

API Gateway

- Single entry point for all client requests
- Handles routing, authentication, rate limiting
- Aggregates responses from multiple services
- Can become a bottleneck

Service Discovery

- Services register themselves with registry
- Services query registry to find other services
- Handles dynamic service locations
- Examples: Consul, Eureka, etcd

Circuit Breaker

- Prevents cascading failures
- Fails fast when downstream service is down
- Allows system to recover
- Provides fallback mechanisms

Saga Pattern

- Manages distributed transactions
- Choreography vs Orchestration approaches
- Handles compensation for failed steps
- Maintains data consistency across services

Migration Strategies

Strangler Fig Pattern

- Gradually replace monolith functionality
- Route traffic between old and new systems
- Reduce risk of big-bang migration
- Allows incremental migration

Database Decomposition

- Start with shared database
- Gradually separate service databases
- Handle data synchronization
- Most challenging part of migration

10. Distributed Systems Fundamentals

Core Challenges

Network Unreliability

- Messages can be lost
- Network partitions occur
- Latency varies unpredictably
- Order of messages may change

Node Failures

- Servers crash unexpectedly
- Services become unresponsive
- Partial failures are common
- Byzantine failures (malicious behavior)

Concurrency

- Multiple nodes processing simultaneously
- Race conditions across network
- Distributed locking challenges
- Ordering of operations

Scalability

- Adding nodes should improve performance
- Coordination overhead increases with nodes
- Bottlenecks shift as system scales
- Non-linear scaling challenges

Consistency Models

Strong Consistency

- All nodes see same data simultaneously
- Requires coordination between nodes
- Higher latency but simpler reasoning
- Examples: Banking transactions

Eventual Consistency

- Nodes will converge to same state eventually
- No guarantees on timing
- Better availability and performance
- Examples: Social media feeds

Weak Consistency

- No guarantees about when all nodes will be consistent
- Application must handle inconsistencies
- Best performance and availability
- Examples: Real-time gaming

Consensus Algorithms

Raft Algorithm

- Leader-based consensus
- Simpler to understand than Paxos
- Strong consistency guarantees
- Used in etcd, Consul

How Raft Works:

1. Elect a leader
2. Leader receives all writes

3. Leader replicates to followers
4. Commit when majority acknowledges

Paxos Algorithm

- More complex but more flexible
- Can handle various failure scenarios
- Theoretical foundation for many systems
- Used in Google's Spanner

Byzantine Fault Tolerance

- Handles malicious or arbitrary failures
- Requires 3f+1 nodes to tolerate f failures
- Used in blockchain systems
- Higher overhead than crash-fault tolerance

Replication Strategies

Synchronous Replication

- Wait for all replicas to acknowledge
- Strong consistency
- Higher latency
- Reduced availability during failures

Asynchronous Replication

- Don't wait for replica acknowledgments
- Lower latency
- Risk of data loss
- Better availability

Semi-Synchronous Replication

- Wait for subset of replicas
- Balance between consistency and availability
- Configurable based on requirements

Distributed Storage

Distributed Hash Tables (DHT)

- Partition data across nodes using consistent hashing
- Each node responsible for range of keys
- Automatic rebalancing when nodes join/leave
- Examples: Amazon Dynamo, Apache Cassandra

Consistent Hashing

- Minimizes redistribution when nodes change
- Virtual nodes for better load distribution
- Hash both data keys and node identifiers
- Foundational technique for distributed systems

Vector Clocks

- Track causality in distributed systems
- Each node maintains its own logical clock
- Helps detect concurrent vs sequential operations
- Used for conflict resolution

11. Message Queues and Event-Driven Architecture

Message Queue Fundamentals

What are Message Queues?

- Asynchronous communication mechanism
- Decouple producers from consumers
- Buffer messages when consumers are slow
- Provide reliability guarantees

Benefits:

- **Decoupling**: Services don't need to know about each other
- **Scalability**: Scale producers and consumers independently
- **Reliability**: Messages aren't lost if consumer is down
- **Flexibility**: Add new consumers without changing producers

Queue Patterns

Point-to-Point

- One producer, one consumer per message

- Message consumed exactly once
- Good for work distribution
- Example: Job processing queue

Publish-Subscribe

- One producer, multiple consumers
- Each consumer gets copy of message
- Good for event notifications
- Example: User registration events

Request-Reply

- Producer sends message and expects response
- Consumer processes and replies
- Synchronous-like behavior over async messaging
- Good for RPC over messaging

Message Queue Technologies

Apache Kafka

- High-throughput, distributed streaming platform
- Excellent for event sourcing and stream processing
- Strong durability and ordering guarantees
- Used by LinkedIn, Netflix, Uber

RabbitMQ

- Traditional message broker
- Rich routing capabilities
- Good for complex routing requirements
- AMQP protocol support

Amazon SQS

- Managed queue service
- Simple to use and scale
- Good for AWS-based architectures
- Serverless-friendly

Redis Pub/Sub

- In-memory messaging
- Very fast but not durable
- Good for real-time notifications
- Simple to implement

Event-Driven Architecture

Event Sourcing

- Store all changes as sequence of events
- Current state derived from event history
- Complete audit trail
- Can replay events to rebuild state

CQRS (Command Query Responsibility Segregation)

- Separate models for reading and writing
- Optimize read and write operations independently
- Often combined with event sourcing
- Better scalability for read-heavy systems

Event Streaming Architecture

- Continuous flow of events
- Real-time processing and analytics
- Events as first-class citizens
- Examples: Kafka Streams, Apache Flink

Handling Event Ordering and Delivery

At-Least-Once Delivery

- Messages guaranteed to be delivered
- May be delivered multiple times
- Consumers must be idempotent
- Most common guarantee

At-Most-Once Delivery

- Messages delivered at most once

- May be lost but never duplicated
- Good for non-critical data
- Lowest overhead

Exactly-Once Delivery

- Messages delivered exactly once
- Most complex to implement
- Highest overhead
- Required for financial transactions

Message Ordering

- Global ordering is expensive
- Partition-level ordering more practical
- Use message keys for related ordering
- Consider event timestamps for processing

12. API Design and Management

REST API Design

REST Principles:

- **Stateless**: Each request contains all necessary information
- **Cacheable**: Responses can be cached when appropriate
- **Uniform Interface**: Consistent interaction patterns
- **Layered System**: Architecture can be composed of layers
- **Client-Server**: Separation of concerns

HTTP Methods:

- **GET**: Retrieve resources (idempotent, safe)
- **POST**: Create new resources
- **PUT**: Update/replace entire resource (idempotent)
- **PATCH**: Partial update of resource
- **DELETE**: Remove resource (idempotent)

Resource Naming:

- Use nouns, not verbs: /users/123 not /getUser/123

- Use plural nouns: /users not /user
- Hierarchical relationships: /users/123/orders
- Use hyphens for readability: /user-profiles

Status Codes:

- **2xx Success**: 200 OK, 201 Created, 204 No Content
- **3xx Redirection**: 301 Moved Permanently, 304 Not Modified
- **4xx Client Error**: 400 Bad Request, 401 Unauthorized, 404 Not Found
- **5xx Server Error**: 500 Internal Server Error, 503 Service Unavailable

GraphQL

What is GraphQL?

- Query language and runtime for APIs
- Single endpoint for all data needs
- Client specifies exactly what data it needs
- Strong type system

Advantages:

- **Flexible Queries**: Get exactly the data you need
- **Single Request**: Fetch related data in one request
- **Introspection**: API is self-documenting
- **Real-time**: Built-in subscription support

Disadvantages:

- **Caching Complexity**: More complex than REST caching
- **Learning Curve**: New concepts to learn
- **Query Complexity**: Need to limit complex queries
- **File Uploads**: Not as straightforward as REST

gRPC

What is gRPC?

- High-performance RPC framework
- Uses Protocol Buffers for serialization
- HTTP/2 transport protocol

- Language-agnostic

Advantages:

- **Performance**: Binary serialization, HTTP/2 multiplexing
- **Code Generation**: Client libraries generated automatically
- **Streaming**: Bidirectional streaming support
- **Type Safety**: Strong typing with Protocol Buffers

When to Use:

- Internal service communication
- High-performance requirements
- Polyglot environments
- Real-time streaming needs

API Versioning

URL Path Versioning

- /v1/users vs /v2/users
- Clear and explicit
- Easy to implement
- URL proliferation

Header Versioning

- Accept: application/vnd.api+json;version=1
- Cleaner URLs
- More complex to implement
- Version not visible in URL

Backward Compatibility

- Add new fields, don't remove existing ones
- Use optional fields for new features
- Deprecate gradually with warnings
- Document breaking changes clearly

API Security

Authentication vs Authorization

- **Authentication**: Who are you?
- **Authorization**: What can you do?

Common Authentication Methods:

- **API Keys**: Simple but not user-specific
- **JWT Tokens**: Stateless, self-contained
- **OAuth 2.0**: Industry standard for authorization
- **Basic Auth**: Simple but requires HTTPS

Security Best Practices:

- Always use HTTPS
- Implement rate limiting
- Validate all inputs
- Use CORS appropriately
- Log security events
- Keep dependencies updated

PART IV: RELIABILITY AND PERFORMANCE

13. Fault Tolerance and Reliability

Understanding Failures

Types of Failures:

- **Hardware Failures**: Disk crashes, memory errors, network failures
- **Software Failures**: Bugs, memory leaks, infinite loops
- **Human Errors**: Misconfigurations, accidental deletions
- **Network Failures**: Partitions, high latency, packet loss

Failure Patterns:

- **Fail-Stop**: System stops completely when error occurs
- **Byzantine**: System continues with incorrect behavior
- **Fail-Slow**: System becomes very slow but doesn't crash
- **Cascading**: One failure triggers additional failures

Building Resilient Systems

Redundancy

- **Active-Active**: Multiple systems handling load simultaneously
- **Active-Passive**: Backup system takes over when primary fails
- **N+1 Redundancy**: One extra component beyond minimum needed
- **Geographic Redundancy**: Systems in multiple locations

Circuit Breaker Pattern

States: Closed, Open, Half-Open

Closed (Normal Operation):

- Allow requests through

- Monitor failure rate

- Open circuit if threshold exceeded

Open (Failing Fast):

- Reject requests immediately

- Return cached response or error

- Periodically test if service recovered

Half-Open (Testing Recovery):

- Allow limited requests through

- Close circuit if requests succeed

- Open circuit if requests still fail

Retry Mechanisms

- **Exponential Backoff**: Increase delay between retries
- **Jitter**: Add randomness to prevent thundering herd
- **Circuit Breaker Integration**: Stop retrying when circuit is open
- **Idempotency**: Ensure retries don't cause side effects

Bulkhead Pattern

- Isolate critical resources

- Separate thread pools for different operations
- Prevent one failing component from affecting others
- Examples: Database connection pools, worker thread pools

Disaster Recovery

Recovery Time Objective (RTO)

- Maximum acceptable downtime
- How quickly must system be restored?
- Drives infrastructure and process decisions

Recovery Point Objective (RPO)

- Maximum acceptable data loss
- How much data can you afford to lose?
- Drives backup and replication strategies

Disaster Recovery Strategies:

Cold Standby

- Backup systems not running
- Longest recovery time
- Lowest cost
- Manual intervention required

Warm Standby

- Backup systems running but not serving traffic
- Medium recovery time and cost
- Some manual steps required

Hot Standby

- Backup systems actively serving traffic
- Fastest recovery time
- Highest cost
- Automatic failover

High Availability Patterns

Health Checks

- Monitor system components continuously
- Remove unhealthy instances from load balancer
- Different types: shallow, deep, dependency checks
- Balance between accuracy and overhead

Graceful Degradation

- Provide reduced functionality when components fail
- Prioritize core features over nice-to-have features
- Example: Show cached data when database is slow

Timeouts and Deadlines

- Set maximum time for operations
- Prevent resource exhaustion
- Fail fast rather than hang indefinitely
- Configure appropriate timeout values

14. Performance Optimization

Performance Metrics

Latency Metrics

- **Response Time**: Time to complete single request
- **Percentiles**: P50, P95, P99 response times
- **Tail Latency**: Worst-case response times
- **Time to First Byte (TTFB)**:

The Complete System Design & Engineering Bible

A Comprehensive Guide from Fundamentals to Advanced Concepts

Table of Contents

PART I: FOUNDATIONS

PART I: FOUNDATIONS

1. Introduction to System Design

What is System Design?

System design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. Think of it as creating a blueprint for a software system, much like an architect creates blueprints for a building.

Why System Design Matters

In the early days of computing, applications were simple and served few users. Today's applications must:

- Handle millions of users simultaneously
- Process vast amounts of data
- Remain available 24/7
- Scale globally
- Adapt to changing requirements

The Evolution of System Design

1960s-1980s: Mainframe Era

- Centralized computing
- Single point of failure
- Limited scalability

1990s-2000s: Client-Server Era

- Distributed computing emerges
- Three-tier architecture becomes standard
- Internet applications grow

2010s-Present: Cloud and Microservices Era

- Horizontal scaling
- Cloud computing
- Microservices architecture
- DevOps and continuous deployment

Key Stakeholders in System Design

Engineers: Implement the technical aspects **Product Managers**: Define requirements and priorities **Architects**: Design the overall system structure **Operations Teams**: Ensure system reliability and performance **Business Leaders**: Define success metrics and constraints

2. Basic Software Architecture Patterns

MVC (Model-View-Controller)

The MVC pattern separates an application into three interconnected components:

Model: Manages data and business logic

- Represents the data structure
- Handles data validation
- Communicates with databases
- Implements business rules

View: Handles the presentation layer

- Displays data to users
- Captures user input
- Formats information for display
- Provides user interface

Controller: Manages user input and coordinates between Model and View

- Receives user requests
- Processes input
- Updates the Model

- Selects appropriate View

Example: E-commerce Shopping Cart

User clicks "Add to Cart" → Controller receives request →

Controller asks Model to add item → Model updates cart data →

Controller tells View to refresh → View shows updated cart

Benefits of MVC:

- Separation of concerns
- Easier testing
- Code reusability
- Multiple views for same data
- Easier maintenance

MVP (Model-View-Presenter)

Similar to MVC but with different responsibilities:

Model: Same as MVC - handles data and business logic **View**: Passive interface that displays data **Presenter**: Handles all UI logic and acts as intermediary

Key Difference from MVC: The View is completely passive and doesn't directly communicate with the Model.

MVVM (Model-View-ViewModel)

Popular in modern UI frameworks:

Model: Business logic and data **View**: User interface **ViewModel**: Binding layer between View and Model

Benefits:

- Two-way data binding
- Better testability
- Clear separation of UI and business logic

Three-Tier Architecture

A fundamental architectural pattern that organizes applications into three logical layers:

Presentation Tier (Client Layer)

- User interface components
- Web browsers, mobile apps, desktop applications

- Handles user interactions
- Formats data for display

Application Tier (Middle Tier/Business Logic Layer)

- Business logic and rules
- Application servers
- Processes user requests
- Coordinates between presentation and data tiers

Data Tier (Database Layer)

- Data storage and retrieval
- Database management systems
- File systems
- Data warehouses

Example: Online Banking System

- Presentation: Web interface for account access
- Application: Transaction processing, account validation
- Data: Customer accounts, transaction history storage

Benefits:

- Scalability: Each tier can be scaled independently
- Security: Sensitive data is isolated in the data tier
- Maintainability: Changes in one tier don't affect others
- Flexibility: Different technologies can be used for each tier

3. Understanding Requirements

Functional Requirements

Define what the system should do:

Core Features

- User authentication and authorization
- Data processing capabilities
- Business logic implementation
- Integration requirements

User Stories Format "As a [type of user], I want [functionality] so that [benefit]"

Example: "As a customer, I want to search for products so that I can find items I want to purchase"

Non-Functional Requirements

Define how the system should perform:

Performance Requirements

- Response time: How quickly should the system respond?
- Throughput: How many requests per second?
- Latency: Maximum acceptable delay

Scalability Requirements

- Expected number of users
- Data volume growth projections
- Geographic distribution needs

Reliability Requirements

- Uptime expectations (99.9%, 99.99%)
- Disaster recovery needs
- Backup requirements

Security Requirements

- Authentication methods
- Data encryption needs
- Compliance requirements (GDPR, HIPAA)

Requirements Gathering Techniques

Stakeholder Interviews

- Identify key stakeholders
- Conduct structured interviews
- Document requirements systematically

Use Case Analysis

- Define system boundaries
- Identify actors and their interactions
- Map out scenarios

Prototyping

- Create mockups or wireframes
- Validate requirements with stakeholders
- Iterate based on feedback

4. System Design Principles

SOLID Principles

Single Responsibility Principle (SRP) Every class should have only one reason to change.

Example: Separate user authentication from user profile management.

Open/Closed Principle (OCP) Software entities should be open for extension but closed for modification.

Example: Use interfaces to allow new payment methods without changing existing code.

Liskov Substitution Principle (LSP) Derived classes must be substitutable for their base classes.

Interface Segregation Principle (ISP) Clients should not be forced to depend on interfaces they don't use.

Dependency Inversion Principle (DIP) Depend on abstractions, not concretions.

DRY (Don't Repeat Yourself)

Avoid code duplication by:

- Creating reusable functions
- Using configuration files
- Implementing shared libraries
- Creating templates and patterns

KISS (Keep It Simple, Stupid)

Favor simplicity:

- Choose simple solutions over complex ones
- Avoid premature optimization
- Use clear and readable code
- Minimize dependencies

YAGNI (You Aren't Gonna Need It)

Don't implement features until they're actually needed:

- Focus on current requirements

- Avoid speculative development
- Implement incrementally
- Refactor when necessary

PART II: CORE CONCEPTS

5. Scalability Fundamentals

Understanding Scale

Small Scale (1-1000 users)

- Single server can handle everything
- Simple database setup
- Minimal caching needed
- Basic monitoring sufficient

Medium Scale (1K-100K users)

- Need multiple servers
- Database optimization required
- Caching becomes important
- Load balancing necessary

Large Scale (100K-10M users)

- Distributed systems required
- Multiple data centers
- Advanced caching strategies
- Sophisticated monitoring

Internet Scale (10M+ users)

- Global distribution
- Complex distributed systems
- Advanced optimization techniques
- Massive infrastructure

Vertical Scaling (Scaling Up)

Adding more power to existing machines:

Advantages:

- Simple to implement
- No application changes needed
- Easier to manage
- Better for applications requiring shared state

Disadvantages:

- Hardware limits
- Expensive high-end hardware
- Single point of failure
- Downtime required for upgrades

When to Use:

- Early stages of application
- Applications with complex shared state
- Budget constraints for development time
- Legacy applications

Horizontal Scaling (Scaling Out)

Adding more machines to the pool of resources:

Advantages:

- No theoretical limit
- Cost-effective using commodity hardware
- Fault tolerance through redundancy
- Can scale specific components independently

Disadvantages:

- Application complexity increases
- Need to handle distributed system challenges
- Data consistency becomes complex
- More operational overhead

Implementation Strategies:

- Stateless application design

- Database sharding
- Load balancing
- Distributed caching

Auto-Scaling

Automatically adjusting resources based on demand:

Reactive Scaling

- Monitor metrics (CPU, memory, requests)
- Scale when thresholds are exceeded
- Risk of being too late for traffic spikes

Predictive Scaling

- Use historical data and machine learning
- Scale before demand increases
- More cost-effective
- Requires good data and models

Scheduled Scaling

- Scale based on known patterns
- Good for predictable traffic
- Business hours, seasonal patterns
- Combined with reactive scaling

6. Database Design and Management

Relational Databases (SQL)

ACID Properties:

- **Atomicity**: Transactions are all-or-nothing
- **Consistency**: Database remains in valid state
- **Isolation**: Concurrent transactions don't interfere
- **Durability**: Committed transactions survive system failures

When to Use SQL Databases:

- Complex relationships between data
- ACID compliance required

- Complex queries and reporting
- Well-defined schema

Popular SQL Databases:

- PostgreSQL: Advanced features, extensible
- MySQL: Fast, widely used
- Oracle: Enterprise features
- SQL Server: Microsoft ecosystem

NoSQL Databases

Document Databases Store data as documents (JSON, XML):

- MongoDB: Flexible schema, rich queries
- CouchDB: Multi-master replication
- Amazon DocumentDB: Managed MongoDB alternative

Key-Value Stores Simple key-value pairs:

- Redis: In-memory, very fast
- Amazon DynamoDB: Managed, scalable
- Riak: Distributed, fault-tolerant

Column-Family Store data in column families:

- Cassandra: Highly scalable, no single point of failure
- HBase: Hadoop ecosystem integration

Graph Databases Optimized for relationships:

- Neo4j: Popular graph database
- Amazon Neptune: Managed graph database

Database Sharding

Splitting database across multiple servers:

Horizontal Sharding Split rows across databases:

- Range-based: Users A-M on DB1, N-Z on DB2
- Hash-based: Hash user ID to determine database
- Directory-based: Lookup service maps keys to shards

Vertical Sharding Split different tables/features:

- User profile data on one database
- Order data on another database
- Product catalog on third database

Challenges:

- Cross-shard queries are complex
- Rebalancing shards is difficult
- Maintaining referential integrity
- Increased operational complexity

Replication Strategies

Master-Slave Replication

- One master handles writes
- Multiple slaves handle reads
- Slaves replicate from master
- Good for read-heavy workloads

Master-Master Replication

- Multiple masters can handle writes
- More complex conflict resolution
- Better availability
- Risk of conflicting writes

Eventual Consistency

- Changes propagate over time
- System remains available during network partitions
- Good for distributed systems
- Requires application-level conflict resolution

7. Caching Strategies

Cache Hierarchy

Browser Cache

- Stores static resources locally
- Reduces server requests

- Controlled by HTTP headers
- User can clear cache

CDN (Content Delivery Network)

- Geographic distribution of content
- Caches static assets globally
- Reduces latency for users
- Examples: CloudFlare, CloudFront

Application-Level Cache

- In-memory data storage
- Frequently accessed data
- Redis, Memcached
- Shared across application instances

Database Cache

- Query result caching
- Reduces database load
- Built into database systems
- Can be application-managed

Caching Patterns

Cache-Aside (Lazy Loading)

```
if data not in cache:
    data = fetch from database
    store data in cache
return data from cache
```

Write-Through

```
write data to cache
write data to database
```

Write-Behind (Write-Back)

```
write data to cache immediately
write data to database asynchronously
```

Refresh-Ahead

proactively refresh cache before expiration

based on access patterns

Cache Invalidation

TTL (Time-To-Live)

- Set expiration time for cache entries
- Simple but may serve stale data
- Good for data that changes infrequently

Event-Based Invalidation

- Invalidate cache when data changes
- More complex but more accurate
- Requires event system

Manual Invalidation

- Explicit cache clearing
- Used for critical updates
- Requires careful coordination

Distributed Caching

Consistent Hashing

- Distributes cache keys across nodes
- Minimizes redistribution when nodes change
- Used by systems like Memcached clusters

Cache Replication

- Replicate cache data across nodes
- Increases availability
- Higher memory usage

Cache Partitioning

- Divide cache across nodes
- Each node stores subset of data
- Better memory utilization

8. Load Balancing

Load Balancer Types

Layer 4 (Transport Layer)

- Routes based on IP and port
- Fast and simple
- Protocol agnostic
- Limited routing intelligence

Layer 7 (Application Layer)

- Routes based on application data
- Content-based routing
- SSL termination
- More intelligent but slower

Load Balancing Algorithms

Round Robin

- Requests distributed sequentially
- Simple and fair
- Doesn't consider server capacity
- Good for homogeneous servers

Weighted Round Robin

- Assigns weights based on server capacity
- More requests to powerful servers
- Still doesn't consider current load

Least Connections

- Routes to server with fewest active connections
- Good for varying request processing times
- Requires connection tracking

Least Response Time

- Routes to server with fastest response
- Considers both load and performance

- More complex to implement

IP Hash

- Routes based on client IP hash
- Ensures same client goes to same server
- Good for stateful applications
- Can cause uneven distribution

Session Management

Sticky Sessions (Session Affinity)

- Route user to same server
- Maintains server-side session state
- Reduces scalability
- Single point of failure per user

Session Replication

- Replicate session across servers
- Higher availability
- Increased network traffic
- Memory overhead

External Session Store

- Store sessions in shared database/cache
- Stateless application servers
- Better scalability
- Single point of failure for session store

Stateless Design

- No server-side session state
- All state in client or database
- Maximum scalability
- Requires application redesign

PART III: ADVANCED ARCHITECTURE

9. Microservices vs Monolithic Architecture

Monolithic Architecture

Definition: A single deployable unit containing all application functionality.

Structure:

- Single codebase
- Single database
- Single deployment
- Shared runtime environment

Advantages:

- **Simple Development**: Easy to develop, test, and deploy initially
- **Simple Deployment**: Single artifact to deploy
- **Easy Testing**: End-to-end testing is straightforward
- **Performance**: No network latency between components
- **Easier Debugging**: All code in one place

Disadvantages:

- **Technology Lock-in**: Entire application uses same technology stack
- **Scaling Challenges**: Must scale entire application, not individual components
- **Development Bottlenecks**: Large teams working on same codebase
- **Reliability**: Single point of failure can bring down entire application
- **Deployment Risk**: Small changes require full application deployment

When to Use Monoliths:

- Small teams (< 10 developers)
- Early-stage applications
- Simple applications with well-defined boundaries
- Applications with tight coupling requirements
- Organizations new to distributed systems

Microservices Architecture

Definition: Architecture style that structures an application as a collection of small, autonomous services.

Characteristics:

- **Single Responsibility**: Each service has one business function
- **Independently Deployable**: Services can be deployed separately
- **Decentralized**: No central orchestration
- **Technology Agnostic**: Each service can use different technologies
- **Fault Isolated**: Failure in one service doesn't crash others

Advantages:

- **Independent Scaling**: Scale services based on demand
- **Technology Diversity**: Use best tool for each job
- **Team Autonomy**: Teams can work independently
- **Fault Isolation**: Failures are contained
- **Faster Deployment**: Deploy services independently

Disadvantages:

- **Distributed System Complexity**: Network calls, latency, failures
- **Data Consistency**: Managing transactions across services
- **Operational Overhead**: More services to monitor and maintain
- **Service Discovery**: How services find each other
- **Testing Complexity**: Integration testing is more complex

When to Use Microservices:

- Large, complex applications
- Multiple teams working on same product
- Different parts of application have different scaling needs
- Organization has strong DevOps capabilities
- Clear service boundaries can be defined

Microservices Design Patterns

Database per Service

- Each microservice has its own database
- Ensures loose coupling
- Enables independent scaling

- Challenges with distributed transactions

API Gateway

- Single entry point for all client requests
- Handles routing, authentication, rate limiting
- Aggregates responses from multiple services
- Can become a bottleneck

Service Discovery

- Services register themselves with registry
- Services query registry to find other services
- Handles dynamic service locations
- Examples: Consul, Eureka, etcd

Circuit Breaker

- Prevents cascading failures
- Fails fast when downstream service is down
- Allows system to recover
- Provides fallback mechanisms

Saga Pattern

- Manages distributed transactions
- Choreography vs Orchestration approaches
- Handles compensation for failed steps
- Maintains data consistency across services

Migration Strategies

Strangler Fig Pattern

- Gradually replace monolith functionality
- Route traffic between old and new systems
- Reduce risk of big-bang migration
- Allows incremental migration

Database Decomposition

- Start with shared database

- Gradually separate service databases
- Handle data synchronization
- Most challenging part of migration

10. Distributed Systems Fundamentals

Core Challenges

Network Unreliability

- Messages can be lost
- Network partitions occur
- Latency varies unpredictably
- Order of messages may change

Node Failures

- Servers crash unexpectedly
- Services become unresponsive
- Partial failures are common
- Byzantine failures (malicious behavior)

Concurrency

- Multiple nodes processing simultaneously
- Race conditions across network
- Distributed locking challenges
- Ordering of operations

Scalability

- Adding nodes should improve performance
- Coordination overhead increases with nodes
- Bottlenecks shift as system scales
- Non-linear scaling challenges

Consistency Models

Strong Consistency

- All nodes see same data simultaneously
- Requires coordination between nodes

- Higher latency but simpler reasoning
- Examples: Banking transactions

Eventual Consistency

- Nodes will converge to same state eventually
- No guarantees on timing
- Better availability and performance
- Examples: Social media feeds

Weak Consistency

- No guarantees about when all nodes will be consistent
- Application must handle inconsistencies
- Best performance and availability
- Examples: Real-time gaming

Consensus Algorithms

Raft Algorithm

- Leader-based consensus
- Simpler to understand than Paxos
- Strong consistency guarantees
- Used in etcd, Consul

How Raft Works:

1. Elect a leader
2. Leader receives all writes
3. Leader replicates to followers
4. Commit when majority acknowledges

Paxos Algorithm

- More complex but more flexible
- Can handle various failure scenarios
- Theoretical foundation for many systems
- Used in Google's Spanner

Byzantine Fault Tolerance

- Handles malicious or arbitrary failures
- Requires 3f+1 nodes to tolerate f failures
- Used in blockchain systems
- Higher overhead than crash-fault tolerance

Replication Strategies

Synchronous Replication

- Wait for all replicas to acknowledge
- Strong consistency
- Higher latency
- Reduced availability during failures

Asynchronous Replication

- Don't wait for replica acknowledgments
- Lower latency
- Risk of data loss
- Better availability

Semi-Synchronous Replication

- Wait for subset of replicas
- Balance between consistency and availability
- Configurable based on requirements

Distributed Storage

Distributed Hash Tables (DHT)

- Partition data across nodes using consistent hashing
- Each node responsible for range of keys
- Automatic rebalancing when nodes join/leave
- Examples: Amazon Dynamo, Apache Cassandra

Consistent Hashing

- Minimizes redistribution when nodes change
- Virtual nodes for better load distribution
- Hash both data keys and node identifiers

- Foundational technique for distributed systems

Vector Clocks

- Track causality in distributed systems
- Each node maintains its own logical clock
- Helps detect concurrent vs sequential operations
- Used for conflict resolution

11. Message Queues and Event-Driven Architecture

Message Queue Fundamentals

What are Message Queues?

- Asynchronous communication mechanism
- Decouple producers from consumers
- Buffer messages when consumers are slow
- Provide reliability guarantees

Benefits:

- **Decoupling**: Services don't need to know about each other
- **Scalability**: Scale producers and consumers independently
- **Reliability**: Messages aren't lost if consumer is down
- **Flexibility**: Add new consumers without changing producers

Queue Patterns

Point-to-Point

- One producer, one consumer per message
- Message consumed exactly once
- Good for work distribution
- Example: Job processing queue

Publish-Subscribe

- One producer, multiple consumers
- Each consumer gets copy of message
- Good for event notifications
- Example: User registration events

Request-Reply

- Producer sends message and expects response
- Consumer processes and replies
- Synchronous-like behavior over async messaging
- Good for RPC over messaging

Message Queue Technologies

Apache Kafka

- High-throughput, distributed streaming platform
- Excellent for event sourcing and stream processing
- Strong durability and ordering guarantees
- Used by LinkedIn, Netflix, Uber

RabbitMQ

- Traditional message broker
- Rich routing capabilities
- Good for complex routing requirements
- AMQP protocol support

Amazon SQS

- Managed queue service
- Simple to use and scale
- Good for AWS-based architectures
- Serverless-friendly

Redis Pub/Sub

- In-memory messaging
- Very fast but not durable
- Good for real-time notifications
- Simple to implement

Event-Driven Architecture

Event Sourcing

- Store all changes as sequence of events

- Current state derived from event history
- Complete audit trail
- Can replay events to rebuild state

CQRS (Command Query Responsibility Segregation)

- Separate models for reading and writing
- Optimize read and write operations independently
- Often combined with event sourcing
- Better scalability for read-heavy systems

Event Streaming Architecture

- Continuous flow of events
- Real-time processing and analytics
- Events as first-class citizens
- Examples: Kafka Streams, Apache Flink

Handling Event Ordering and Delivery

At-Least-Once Delivery

- Messages guaranteed to be delivered
- May be delivered multiple times
- Consumers must be idempotent
- Most common guarantee

At-Most-Once Delivery

- Messages delivered at most once
- May be lost but never duplicated
- Good for non-critical data
- Lowest overhead

Exactly-Once Delivery

- Messages delivered exactly once
- Most complex to implement
- Highest overhead
- Required for financial transactions

Message Ordering

- Global ordering is expensive
- Partition-level ordering more practical
- Use message keys for related ordering
- Consider event timestamps for processing

12. API Design and Management

REST API Design

REST Principles:

- **Stateless**: Each request contains all necessary information
- **Cacheable**: Responses can be cached when appropriate
- **Uniform Interface**: Consistent interaction patterns
- **Layered System**: Architecture can be composed of layers
- **Client-Server**: Separation of concerns

HTTP Methods:

- **GET**: Retrieve resources (idempotent, safe)
- **POST**: Create new resources
- **PUT**: Update/replace entire resource (idempotent)
- **PATCH**: Partial update of resource
- **DELETE**: Remove resource (idempotent)

Resource Naming:

- Use nouns, not verbs: /users/123 not /getUser/123
- Use plural nouns: /users not /user
- Hierarchical relationships: /users/123/orders
- Use hyphens for readability: /user-profiles

Status Codes:

- **2xx Success**: 200 OK, 201 Created, 204 No Content
- **3xx Redirection**: 301 Moved Permanently, 304 Not Modified
- **4xx Client Error**: 400 Bad Request, 401 Unauthorized, 404 Not Found
- **5xx Server Error**: 500 Internal Server Error, 503 Service Unavailable

GraphQL

What is GraphQL?

- Query language and runtime for APIs
- Single endpoint for all data needs
- Client specifies exactly what data it needs
- Strong type system

Advantages:

- **Flexible Queries**: Get exactly the data you need
- **Single Request**: Fetch related data in one request
- **Introspection**: API is self-documenting
- **Real-time**: Built-in subscription support

Disadvantages:

- **Caching Complexity**: More complex than REST caching
- **Learning Curve**: New concepts to learn
- **Query Complexity**: Need to limit complex queries
- **File Uploads**: Not as straightforward as REST

gRPC

What is gRPC?

- High-performance RPC framework
- Uses Protocol Buffers for serialization
- HTTP/2 transport protocol
- Language-agnostic

Advantages:

- **Performance**: Binary serialization, HTTP/2 multiplexing
- **Code Generation**: Client libraries generated automatically
- **Streaming**: Bidirectional streaming support
- **Type Safety**: Strong typing with Protocol Buffers

When to Use:

- Internal service communication

- High-performance requirements
- Polyglot environments
- Real-time streaming needs

API Versioning

URL Path Versioning

- /v1/users vs /v2/users
- Clear and explicit
- Easy to implement
- URL proliferation

Header Versioning

- Accept: application/vnd.api+json;version=1
- Cleaner URLs
- More complex to implement
- Version not visible in URL

Backward Compatibility

- Add new fields, don't remove existing ones
- Use optional fields for new features
- Deprecate gradually with warnings
- Document breaking changes clearly

API Security

Authentication vs Authorization

- **Authentication**: Who are you?
- **Authorization**: What can you do?

Common Authentication Methods:

- **API Keys**: Simple but not user-specific
- **JWT Tokens**: Stateless, self-contained
- **OAuth 2.0**: Industry standard for authorization
- **Basic Auth**: Simple but requires HTTPS

Security Best Practices:

- Always use HTTPS
- Implement rate limiting
- Validate all inputs
- Use CORS appropriately
- Log security events
- Keep dependencies updated

PART IV: RELIABILITY AND PERFORMANCE

13. Fault Tolerance and Reliability

Understanding Failures

Types of Failures:

- **Hardware Failures**: Disk crashes, memory errors, network failures
- **Software Failures**: Bugs, memory leaks, infinite loops
- **Human Errors**: Misconfigurations, accidental deletions
- **Network Failures**: Partitions, high latency, packet loss

Failure Patterns:

- **Fail-Stop**: System stops completely when error occurs
- **Byzantine**: System continues with incorrect behavior
- **Fail-Slow**: System becomes very slow but doesn't crash
- **Cascading**: One failure triggers additional failures

Building Resilient Systems

Redundancy

- **Active-Active**: Multiple systems handling load simultaneously
- **Active-Passive**: Backup system takes over when primary fails
- **N+1 Redundancy**: One extra component beyond minimum needed
- **Geographic Redundancy**: Systems in multiple locations

Circuit Breaker Pattern

States: Closed, Open, Half-Open

Closed (Normal Operation):

- Allow requests through

- Monitor failure rate

- Open circuit if threshold exceeded

Open (Failing Fast):

- Reject requests immediately

- Return cached response or error

- Periodically test if service recovered

Half-Open (Testing Recovery):

- Allow limited requests through

- Close circuit if requests succeed

- Open circuit if requests still fail

Retry Mechanisms

- **Exponential Backoff**: Increase delay between retries
- **Jitter**: Add randomness to prevent thundering herd
- **Circuit Breaker Integration**: Stop retrying when circuit is open
- **Idempotency**: Ensure retries don't cause side effects

Bulkhead Pattern

- Isolate critical resources
- Separate thread pools for different operations
- Prevent one failing component from affecting others
- Examples: Database connection pools, worker thread pools

Disaster Recovery

Recovery Time Objective (RTO)

- Maximum acceptable downtime
- How quickly must system be restored?
- Drives infrastructure and process decisions

Recovery Point Objective (RPO)

- Maximum acceptable data loss
- How much data can you afford to lose?
- Drives backup and replication strategies

Disaster Recovery Strategies:

Cold Standby

- Backup systems not running
- Longest recovery time
- Lowest cost
- Manual intervention required

Warm Standby

- Backup systems running but not serving traffic
- Medium recovery time and cost
- Some manual steps required

Hot Standby

- Backup systems actively serving traffic
- Fastest recovery time
- Highest cost
- Automatic failover

High Availability Patterns

Health Checks

- Monitor system components continuously
- Remove unhealthy instances from load balancer
- Different types: shallow, deep, dependency checks
- Balance between accuracy and overhead

Graceful Degradation

- Provide reduced functionality when components fail
- Prioritize core features over nice-to-have features
- Example: Show cached data when database is slow

Timeouts and Deadlines

- Set maximum time for operations
- Prevent resource exhaustion
- Fail fast rather than hang indefinitely
- Configure appropriate timeout values

14. Performance Optimization

Performance Metrics

Latency Metrics

- **Response Time**: Time to complete single request
- **Percentiles**: P50, P95, P99 response times
- **Tail Latency**: Worst-case response times
- **Time to First Byte (TTFB)**: Server processing time before first byte sent

Throughput Metrics

- **Requests Per Second (RPS)**: Number of requests handled per second
- **Transactions Per Second (TPS)**: Business transactions completed per second
- **Bandwidth**: Data transfer rate (MB/s, GB/s)
- **Concurrent Users**: Number of simultaneous active users

Resource Utilization

- **CPU Utilization**: Percentage of CPU capacity used
- **Memory Usage**: RAM consumption patterns
- **Disk I/O**: Read/write operations per second
- **Network I/O**: Incoming/outgoing data rates

Performance Testing

Load Testing

- Test system under expected normal load
- Verify system meets performance requirements
- Identify bottlenecks under normal conditions
- Baseline for other testing types

Stress Testing

- Test system beyond normal capacity
- Find breaking point of the system
- Observe system behavior under extreme load
- Plan for capacity limits

Spike Testing

- Test sudden increases in load
- Simulate viral content or flash sales
- Verify auto-scaling mechanisms
- Test system recovery after spike

Volume Testing

- Test with large amounts of data
- Verify performance with full datasets
- Test database performance with realistic data
- Identify storage-related bottlenecks

Database Performance Optimization

Indexing Strategies

- **B-Tree Indexes**: Good for range queries and equality
- **Hash Indexes**: Fast for equality lookups
- **Composite Indexes**: Multiple columns, order matters
- **Partial Indexes**: Index subset of data with conditions

Query Optimization

- **Explain Plans**: Understand query execution path
- **Avoid SELECT ***: Only fetch needed columns
- **Use Appropriate Joins**: Understand join types and costs
- **Limit Result Sets**: Use LIMIT/TOP clauses

Connection Management

- **Connection Pooling**: Reuse database connections
- **Pool Sizing**: Balance between resource usage and performance
- **Connection Timeout**: Prevent resource leaks

- **Prepared Statements**: Reduce parsing overhead

Database Sharding Performance

- **Shard Key Selection**: Distribute load evenly
- **Avoid Cross-Shard Queries**: Minimize distributed queries
- **Shard Rebalancing**: Handle hot spots dynamically
- **Aggregation Strategies**: Efficient cross-shard aggregations

Application Performance Optimization

Code-Level Optimizations

- **Algorithm Complexity**: Choose appropriate algorithms ($O(n)$ vs $O(n^2)$)
- **Data Structure Selection**: Arrays vs Lists vs Maps
- **Memory Management**: Avoid memory leaks and excessive allocations
- **Lazy Loading**: Load data only when needed

Caching Optimizations

- **Cache Hit Ratios**: Monitor and optimize cache effectiveness
- **Cache Warming**: Pre-populate cache with likely-needed data
- **Cache Invalidation**: Ensure cache consistency
- **Multi-Level Caching**: Browser, CDN, application, database caches

Asynchronous Processing

- **Non-Blocking I/O**: Don't wait for I/O operations
- **Message Queues**: Decouple heavy processing from user requests
- **Background Jobs**: Process non-critical tasks asynchronously
- **Event-Driven Architecture**: React to events rather than polling

Frontend Performance

Resource Optimization

- **Minification**: Reduce file sizes (CSS, JS, HTML)
- **Compression**: Gzip/Brotli compression for text files
- **Image Optimization**: Appropriate formats and sizes
- **Bundle Splitting**: Load only necessary code

Network Optimization

- **CDN Usage**: Serve static assets from edge locations
- **HTTP/2**: Multiplexing, server push, header compression
- **Prefetching**: Load resources before they're needed
- **Lazy Loading**: Load images and content as needed

Rendering Performance

- **Critical Rendering Path**: Minimize render-blocking resources
- **Code Splitting**: Load JavaScript in chunks
- **Service Workers**: Cache resources and enable offline functionality
- **Progressive Web App**: App-like experience in browsers

15. Monitoring and Observability

Three Pillars of Observability

Metrics

- Numerical measurements over time
- Examples: CPU usage, response time, error rate
- Good for alerts and dashboards
- Aggregated data with lower storage costs

Logs

- Detailed records of events
- Examples: Error messages, user actions, system events
- Good for debugging and audit trails
- High storage costs but rich information

Traces

- Request flows through distributed systems
- Shows how requests travel between services
- Good for understanding system behavior
- Essential for debugging distributed systems

Key Metrics to Monitor

Application Metrics

- **Error Rate**: Percentage of failed requests

- **Response Time**: P50, P95, P99 latencies
- **Throughput**: Requests per second
- **Availability**: Uptime percentage

Infrastructure Metrics

- **CPU Utilization**: Processor usage across instances
- **Memory Usage**: RAM consumption patterns
- **Disk I/O**: Read/write operations and queue depth
- **Network I/O**: Bandwidth usage and packet loss

Business Metrics

- **Conversion Rates**: Business goal completion rates
- **User Engagement**: Active users, session duration
- **Revenue Impact**: Financial metrics tied to system performance
- **Customer Satisfaction**: User experience metrics

Logging Best Practices

Structured Logging

- Use consistent log formats (JSON)
- Include contextual information (user ID, request ID)
- Standardize log levels (DEBUG, INFO, WARN, ERROR)
- Make logs machine-readable for analysis

Log Levels

- **DEBUG**: Detailed information for diagnosing problems
- **INFO**: General information about system operation
- **WARN**: Potentially harmful situations
- **ERROR**: Error events that might allow application to continue
- **FATAL**: Severe errors that cause application to abort

Centralized Logging

- Collect logs from all system components
- Use tools like ELK Stack (Elasticsearch, Logstash, Kibana)
- Provide searchable, filterable log interface

- Correlate logs across different services

Distributed Tracing

Trace Components

- **Trace**: Complete request journey through system
- **Span**: Individual operation within a trace
- **Context**: Metadata passed between services
- **Baggage**: Additional context information

Tracing Tools

- **Jaeger**: Open-source distributed tracing
- **Zipkin**: Distributed tracing system
- **AWS X-Ray**: Managed tracing service
- **Google Cloud Trace**: Cloud-native tracing

Implementation Considerations

- **Sampling**: Trace subset of requests to reduce overhead
- **Context Propagation**: Pass trace context between services
- **Performance Impact**: Minimize tracing overhead
- **Storage**: Manage trace data storage and retention

Alerting and Incident Response

Alerting Principles

- **Alert on Symptoms**: Alert on user-facing issues, not just component failures
- **Meaningful Alerts**: Every alert should be actionable
- **Alert Fatigue**: Avoid too many false positives
- **Escalation**: Define escalation paths for unresolved alerts

SLA/SLO/SLI Framework

- **SLI (Service Level Indicator)**: Metrics that matter to users
- **SLO (Service Level Objective)**: Target values for SLIs
- **SLA (Service Level Agreement)**: Contractual obligations
- **Error Budgets**: Acceptable amount of unreliability

Incident Response Process

1. **Detection**: Automated alerts or user reports
2. **Response**: Acknowledge and begin investigation
3. **Mitigation**: Restore service quickly
4. **Resolution**: Fix root cause
5. **Post-Mortem**: Learn from incident without blame

16. Security in System Design

Security Principles

Defense in Depth

- Multiple layers of security controls
- If one layer fails, others provide protection
- Examples: Firewalls, authentication, encryption, monitoring

Principle of Least Privilege

- Give users/systems minimum access needed
- Regularly review and revoke unnecessary permissions
- Use role-based access control (RBAC)

Zero Trust Architecture

- Never trust, always verify
- Verify every user and device
- Encrypt all communications
- Monitor all network traffic

Fail Securely

- When systems fail, they should fail to a secure state
- Default deny policies
- Secure error handling
- Don't expose sensitive information in errors

Authentication and Authorization

Authentication Methods

- **Password-Based**: Traditional username/password
- **Multi-Factor Authentication (MFA)**: Something you know, have, are

- **Single Sign-On (SSO)**: One login for multiple systems
- **OAuth 2.0/OpenID Connect**: Delegated authorization

Authorization Models

- **Role-Based Access Control (RBAC)**: Users assigned to roles
- **Attribute-Based Access Control (ABAC)**: Context-aware permissions
- **Access Control Lists (ACL)**: Permissions per resource
- **Capability-Based**: Unforgeable tokens representing permissions

Token Management

- **JWT (JSON Web Tokens)**: Self-contained tokens
- **Refresh Tokens**: Long-lived tokens for obtaining new access tokens
- **Token Expiration**: Balance security and user experience
- **Token Revocation**: Ability to invalidate tokens

Data Security

Encryption

- **Encryption at Rest**: Protect stored data
- **Encryption in Transit**: Protect data during transmission
- **Key Management**: Secure key storage and rotation
- **End-to-End Encryption**: Data encrypted from sender to recipient

Data Classification

- **Public**: No harm if disclosed
- **Internal**: Shouldn't be disclosed outside organization
- **Confidential**: Serious harm if disclosed
- **Restricted**: Extreme harm if disclosed

Data Privacy

- **GDPR Compliance**: European privacy regulation
- **CCPA Compliance**: California privacy law
- **Data Minimization**: Collect only necessary data
- **Right to Deletion**: Users can request data deletion

Network Security

Firewalls

- **Perimeter Firewalls**: Control traffic entering/leaving network
- **Application Firewalls**: Filter based on application protocols
- **Web Application Firewalls (WAF)**: Protect web applications
- **Network Segmentation**: Isolate network segments

DDoS Protection

- **Rate Limiting**: Limit requests per user/IP
- **Traffic Shaping**: Prioritize legitimate traffic
- **CDN Protection**: Absorb attack traffic at edge
- **Anomaly Detection**: Identify unusual traffic patterns

Secure Communication

- **TLS/SSL**: Encrypt HTTP communications
- **Certificate Management**: Proper certificate handling
- **Perfect Forward Secrecy**: Past communications remain secure
- **HSTS**: Force HTTPS connections

Application Security

Common Vulnerabilities

- **SQL Injection**: Malicious SQL in user input
- **Cross-Site Scripting (XSS)**: Malicious scripts in web pages
- **Cross-Site Request Forgery (CSRF)**: Unauthorized actions on behalf of user
- **Insecure Direct Object References**: Access to unauthorized objects

Security Testing

- **Static Application Security Testing (SAST)**: Analyze source code
- **Dynamic Application Security Testing (DAST)**: Test running application
- **Interactive Application Security Testing (IAST)**: Real-time testing
- **Penetration Testing**: Simulate real attacks

Secure Development Practices

- **Security by Design**: Consider security from the beginning
- **Input Validation**: Validate all user inputs

- **Output Encoding**: Properly encode output data
- **Security Code Reviews**: Review code for security issues

PART V: DATA AND CONSISTENCY

17. Data Storage Patterns

Choosing the Right Database

Relational Databases (RDBMS)

- **Use Cases**:
    - Complex relationships between entities
    - ACID compliance requirements
    - Complex queries and reporting
    - Well-defined, stable schema
- **Examples**: PostgreSQL, MySQL, Oracle, SQL Server
- **Advantages**: Mature ecosystem, strong consistency, complex queries
- **Disadvantages**: Harder to scale horizontally, rigid schema

Document Databases

- **Use Cases**:
    - Flexible, evolving schemas
    - JSON-like data structures
    - Content management systems
    - Catalogs and user profiles
- **Examples**: MongoDB, CouchDB, Amazon DocumentDB
- **Advantages**: Flexible schema, natural object mapping
- **Disadvantages**: Less mature query capabilities, potential for data duplication

Key-Value Stores

- **Use Cases**:
    - High-performance caching
    - Session storage
    - Real-time recommendations

- Shopping carts

- **Examples**: Redis, Amazon DynamoDB, Riak
- **Advantages**: Extremely fast, simple model, highly scalable
- **Disadvantages**: Limited query capabilities, no relationships

Wide Column Stores

- **Use Cases**:
    - Time-series data
    - IoT sensor data
    - Logging and analytics
    - High write throughput
- **Examples**: Cassandra, HBase, Amazon DynamoDB
- **Advantages**: Excellent for write-heavy workloads, scalable
- **Disadvantages**: Complex data modeling, eventual consistency

Graph Databases

- **Use Cases**:
    - Social networks
    - Recommendation engines
    - Fraud detection
    - Knowledge graphs
- **Examples**: Neo4j, Amazon Neptune, ArangoDB
- **Advantages**: Excellent for relationship queries, intuitive modeling
- **Disadvantages**: Specialized use cases, can be complex to scale

Polyglot Persistence

Definition: Using multiple database technologies within the same application to handle different data storage needs.

Benefits:

- Use the right tool for each job
- Optimize for specific use cases
- Avoid forcing all data into one model

Challenges:

- Increased operational complexity
- Data consistency across systems
- Multiple skillsets required
- Transaction boundaries

Implementation Strategies:

- **Database per Service**: Each microservice owns its data
- **CQRS**: Separate read and write models
- **Event Sourcing**: Maintain event log, project to different views
- **Data Synchronization**: Keep systems in sync through events

Data Modeling Patterns

Normalization vs Denormalization

Normalization:

- Eliminate data redundancy
- Reduce storage space
- Maintain data consistency
- More complex queries

Denormalization:

- Duplicate data for performance
- Faster read queries
- Increased storage requirements
- Risk of data inconsistency

Schema Design Patterns

Embedded Documents (Document Databases):

```
{
 "user_id": "123",
 "name": "John Doe",
 "addresses": [
  {
```

```
      "type": "home",
      "street": "123 Main St",
      "city": "Anytown"
    }
  ]
}
```

Reference Documents:

```
{
  "user_id": "123",
  "name": "John Doe",
  "address_ids": ["addr1", "addr2"]
}
```

Hybrid Approach:

- Embed frequently accessed data
- Reference large or infrequently accessed data
- Balance between performance and consistency

Data Lifecycle Management

Data Archiving

- Move old data to cheaper storage
- Maintain performance of active data
- Implement transparent data access
- Define retention policies

Data Partitioning

- **Horizontal Partitioning**: Split rows across tables/databases
- **Vertical Partitioning**: Split columns across tables
- **Functional Partitioning**: Split by feature/domain
- **Time-based Partitioning**: Split by date ranges

Data Backup and Recovery

- **Full Backups**: Complete database copy

- **Incremental Backups**: Only changed data
- **Point-in-Time Recovery**: Restore to specific moment
- **Cross-Region Replication**: Geographic disaster recovery

18. Consistency Models

Understanding Consistency

Strong Consistency

- All nodes see the same data simultaneously
- Reads always return the most recent write
- Requires coordination between nodes
- Higher latency, lower availability

Eventual Consistency

- System will become consistent over time
- No guarantee when consistency will be achieved
- Nodes may return different values temporarily
- Better performance and availability

Weak Consistency

- No guarantees about when data will be consistent
- Application must handle inconsistencies
- Best performance characteristics
- Used in systems where some inconsistency is acceptable

Consistency Levels

Monotonic Read Consistency

- If a process reads a value, subsequent reads return the same or newer value
- Prevents reading older values after newer ones
- Important for user experience

Monotonic Write Consistency

- Writes by a single process are seen by all nodes in the order they were written
- Prevents out-of-order write operations
- Maintains causal relationships

Read Your Writes Consistency

- After writing a value, the same process will always read that value or a newer one
- Prevents confusion where user doesn't see their own changes
- Common requirement for user-facing applications

Writes Follow Reads Consistency

- If a process reads a value and then writes, the write is guaranteed to take place after the read
- Maintains causal consistency
- Important for maintaining logical ordering

Implementing Consistency

Quorum-Based Consistency

- Require majority of nodes to agree
- Read + Write replicas > Total replicas
- Configurable consistency levels
- Used in systems like Cassandra, DynamoDB

Vector Clocks

- Track causal relationships between events
- Each node maintains logical clock
- Detect concurrent vs sequential operations
- Resolve conflicts deterministically

Consensus Protocols

- **Raft**: Leader-based consensus algorithm
- **Paxos**: More general consensus algorithm
- **PBFT**: Byzantine fault tolerant consensus
- **Blockchain Consensus**: Proof of Work, Proof of Stake

Conflict Resolution

Last Write Wins (LWW)

- Use timestamp to determine winning write
- Simple but may lose data
- Requires synchronized clocks

- Used in systems like Riak

Multi-Value Returns

- Return all conflicting values
- Let application decide resolution
- Preserves all data
- Requires application logic

Conflict-Free Replicated Data Types (CRDTs)

- Data structures that automatically resolve conflicts
- Mathematical properties ensure convergence
- Examples: Counters, sets, maps
- Used in collaborative editing systems

19. CAP Theorem and Trade-offs

Understanding CAP Theorem

Consistency: All nodes see the same data at the same time **Availability**: System remains operational and responsive **Partition Tolerance**: System continues despite network failures

The Theorem: In a distributed system, you can only guarantee two of the three properties when network partitions occur.

CAP Theorem in Practice

CP Systems (Consistency + Partition Tolerance)

- Choose consistency over availability during partitions
- System may become unavailable but data remains consistent
- Examples: MongoDB (default), HBase, Redis Cluster
- Good for: Financial systems, inventory management

AP Systems (Availability + Partition Tolerance)

- Choose availability over consistency during partitions
- System remains available but may return stale data
- Examples: Cassandra, DynamoDB, CouchDB
- Good for: Social media, content delivery, catalogs

CA Systems (Consistency + Availability)

- Only possible without network partitions
- Traditional single-node databases
- Examples: PostgreSQL, MySQL (single instance)
- Good for: Applications where network partitions are rare

PACELC Theorem

Extension of CAP: In case of network Partitioning, choose between Availability and Consistency, Else choose between Latency and Consistency.

PA/EL Systems:

- Partition: Choose Availability
- Else: Choose Latency
- Examples: Cassandra, DynamoDB

PC/EC Systems:

- Partition: Choose Consistency
- Else: Choose Consistency
- Examples: MongoDB, HBase

PC/EL Systems:

- Partition: Choose Consistency
- Else: Choose Latency
- Examples: Yahoo! PNUTS

Making CAP Trade-offs

Business Requirements Drive Decisions

- Financial applications need consistency
- Social media can tolerate some inconsistency
- Real-time gaming needs low latency
- E-commerce balances consistency and availability

Hybrid Approaches

- Different consistency levels for different data
- Eventual consistency for reads, strong consistency for writes
- Compensating transactions for distributed operations

- Flexible consistency models

Tunable Consistency

- Configure consistency levels per operation
- Read/write quorums in systems like Cassandra
- Application-level consistency guarantees
- Dynamic adjustment based on conditions

20. ACID vs BASE

ACID Properties

Atomicity

- Transactions are all-or-nothing
- Either all operations succeed or all fail
- No partial updates to database
- Implemented through transaction logs

Consistency

- Database remains in valid state after transactions
- All constraints and rules are enforced
- No invalid data states
- Referential integrity maintained

Isolation

- Concurrent transactions don't interfere
- Each transaction sees consistent view
- Multiple isolation levels available
- Prevents dirty reads, phantom reads

Durability

- Committed transactions survive system failures
- Data is permanently stored
- Changes survive crashes and power failures
- Implemented through write-ahead logging

BASE Properties

Basically Available

- System remains available most of the time
- May degrade gracefully under load
- Some operations may fail or timeout
- Prioritizes availability over consistency

Soft State

- Data may be inconsistent temporarily
- System doesn't enforce immediate consistency
- State may change due to eventual consistency
- Application must handle inconsistent reads

Eventual Consistency

- System will become consistent over time
- No guarantee when consistency will be achieved
- Conflicts resolved through various mechanisms
- Good enough for many applications

When to Use ACID vs BASE

Use ACID When:

- Financial transactions
- Inventory management
- User account management
- Regulatory compliance requirements
- Data integrity is critical

Use BASE When:

- Social media feeds
- Content delivery
- Analytics and reporting
- Real-time recommendations
- High-scale web applications

Implementing ACID in Distributed Systems

Two-Phase Commit (2PC)

- Coordinator manages distributed transaction
- Phase 1: Prepare - ask all participants to prepare
- Phase 2: Commit - tell all participants to commit
- Blocking protocol - failure of coordinator blocks progress

Three-Phase Commit (3PC)

- Adds pre-commit phase to 2PC
- Reduces blocking scenarios
- More complex protocol
- Still has edge cases

Saga Pattern

- Long-running transactions as sequence of smaller transactions
- Each step has compensating action
- Choreography vs Orchestration
- Eventual consistency with compensation

PART VI: REAL-WORLD APPLICATIONS

21. Case Studies: Major Tech Companies

Google's Approach to Scale

Google File System (GFS)

- Designed for large files and append operations
- Assumes commodity hardware failures
- Single master, multiple chunk servers
- Optimized for throughput over latency

BigTable

- Distributed storage system for structured data
- Column-family data model
- Automatic sharding and rebalancing
- Inspired many NoSQL databases

MapReduce

- Programming model for processing large datasets
- Fault-tolerant distributed computing
- Inspired Hadoop ecosystem
- Now superseded by more flexible frameworks

Spanner

- Globally distributed database
- Provides ACID transactions across data centers
- Uses atomic clocks for consistency
- Combines benefits of relational and NoSQL databases

Amazon's Architecture Philosophy

Service-Oriented Architecture

- Every team owns a service
- Services communicate through APIs
- Two-pizza team rule
- Decentralized decision making

Dynamo

- Eventually consistent key-value store
- Inspired by Amazon's shopping cart requirements
- Gossip protocol for membership
- Vector clocks for conflict resolution

Microservices at Scale

- Thousands of services
- Each service has specific purpose
- Autonomous teams
- Fault isolation and independent scaling

Infrastructure as Code

- Everything defined as code
- Automated provisioning and deployment

- Immutable infrastructure
- Cloud-native from the beginning

Netflix's Resilience Engineering

Chaos Engineering

- Deliberately introduce failures
- Chaos Monkey randomly terminates instances
- Chaos Kong simulates region failures
- Build confidence in system resilience

Microservices Architecture

- Hundreds of services
- Each service is independently deployable
- Service discovery and load balancing
- Circuit breakers for fault isolation

Global Content Delivery

- Content caches at ISP locations
- Predictive caching algorithms
- Multiple CDN providers
- Optimized for video streaming

Data-Driven Decisions

- A/B testing for all features
- Real-time analytics
- Machine learning for recommendations
- Personalization at scale

Facebook's Social Graph

Graph Structure

- Users, posts, comments, likes as graph nodes
- Relationships as edges
- Complex queries on social connections
- Billions of nodes and edges

TAO (The Associations and Objects)

- Graph data model
- Geographically distributed
- Read-heavy workload optimization
- Cache-aside pattern

Feed Generation

- Personalized content for each user
- Real-time and batch processing
- Machine learning for ranking
- Billions of feeds generated daily

Real-Time Messaging

- WhatsApp, Messenger, Instagram messaging
- Eventual consistency for messages
- Optimized for mobile networks
- End-to-end encryption

Uber's Microservices Evolution

Monolith to Microservices

- Started with monolithic architecture
- Gradually extracted services
- Domain-driven design approach
- Challenges with distributed transactions

Real-Time Systems

- Driver-rider matching
- Dynamic pricing
- Real-time location tracking
- Event-driven architecture

Data Processing

- Batch processing for analytics
- Stream processing for real-time features

- Data lake for historical analysis
- Machine learning pipelines

Global Expansion

- Multi-region deployments
- Localized features and regulations
- Data residency requirements
- Cultural and technical challenges

22. Common System Design Interview Questions

URL Shortener (like bit.ly)

Requirements:

- Shorten long URLs
- Redirect to original URL
- Handle 100M URLs per day
- Analytics on click counts

High-Level Design:

1. **URL Encoding**: Base62 encoding (a-z, A-Z, 0-9)
2. **Database**: Store mapping between short and long URLs
3. **Cache**: Cache popular URLs
4. **Load Balancer**: Distribute traffic across servers

Detailed Design:

- **URL Generation**: Counter-based or hash-based
- **Database Schema**: url_id, short_url, long_url, created_at, expires_at
- **Caching Strategy**: LRU cache for hot URLs
- **Analytics**: Separate service for click tracking

Scale Considerations:

- **Read vs Write Ratio**: 100:1 read-heavy
- **Database Sharding**: Shard by URL hash
- **CDN**: Cache redirects globally
- **Rate Limiting**: Prevent abuse

Chat System (like WhatsApp)

Requirements:

- Send and receive messages
- Online status
- Group chats
- Message history
- Push notifications

High-Level Design:

1. **WebSocket Connections**: Real-time messaging
2. **Message Queue**: Reliable message delivery
3. **Database**: Store messages and user data
4. **Notification Service**: Push notifications

Detailed Design:

- **Connection Management**: WebSocket servers
- **Message Routing**: Route messages to correct recipients
- **Database Schema**: Users, conversations, messages
- **Offline Handling**: Store messages for offline users

Scale Considerations:

- **Connection Scaling**: Horizontal scaling of WebSocket servers
- **Message Ordering**: Ensure message order within conversations
- **Group Chat Scaling**: Optimize for large groups
- **Global Distribution**: Multiple data centers

Social Media Feed (like Twitter)

Requirements:

- Users can post tweets
- Users can follow other users
- Generate timeline for users
- Handle celebrity users with millions of followers

High-Level Design:

1. **User Service**: Manage users and relationships
2. **Tweet Service**: Store and retrieve tweets
3. **Timeline Service**: Generate user timelines
4. **Notification Service**: Real-time updates

Detailed Design:

- **Tweet Storage**: Optimized for write-heavy workload
- **Timeline Generation**: Push vs Pull models
- **Fan-out Strategies**: Immediate vs lazy fan-out
- **Caching**: Cache popular tweets and timelines

Scale Considerations:

- **Celebrity Problem**: Hybrid push-pull for popular users
- **Timeline Ranking**: Machine learning for relevance
- **Media Handling**: Separate service for images/videos
- **Real-time Updates**: WebSocket for live updates

Design Cache System (like Redis)

Requirements:

- Get and Put operations
- Distributed across multiple servers
- Handle server failures
- LRU eviction policy

High-Level Design:

1. **Consistent Hashing**: Distribute keys across servers
2. **Replication**: Multiple copies for reliability
3. **Client Library**: Handle server communication
4. **Monitoring**: Health checks and metrics

Detailed Design:

- **Hashing Strategy**: Consistent hashing with virtual nodes
- **Replication Factor**: 3 replicas per key
- **Conflict Resolution**: Last-write-wins or vector clocks

- **Client Routing**: Smart client with server topology

Scale Considerations:

- **Hot Spots**: Identify and handle popular keys
- **Memory Management**: Efficient memory usage
- **Network Optimization**: Minimize network calls
- **Monitoring**: Track hit rates and performance

Ride-Sharing Service (like Uber)

Requirements:

- Match drivers with riders
- Real-time location tracking
- Pricing and payments
- Trip history and ratings

High-Level Design:

1. **Location Service**: Track driver and rider locations
2. **Matching Service**: Find suitable driver for rider
3. **Trip Service**: Manage trip lifecycle
4. **Payment Service**: Handle payments and pricing

Detailed Design:

- **Geospatial Indexing**: Efficient location queries
- **Matching Algorithm**: Optimize for distance and time
- **Real-time Updates**: WebSocket for location updates
- **State Management**: Trip state machine

Scale Considerations:

- **Geographic Partitioning**: Shard by city/region
- **Real-time Processing**: Stream processing for locations
- **Demand Prediction**: Machine learning for supply/demand
- **Fault Tolerance**: Handle service failures gracefully

23. Best Practices and Anti-Patterns

System Design Best Practices

Start Simple

- Begin with monolithic architecture
- Extract services when needed
- Avoid premature optimization
- Understand your domain first

Design for Failure

- Assume components will fail
- Implement circuit breakers
- Use timeouts and retries
- Plan for disaster recovery

Monitor Everything

- Implement observability from the beginning
- Use structured logging
- Track business metrics
- Set up meaningful alerts

Security by Design

- Implement authentication and authorization
- Encrypt sensitive data
- Use HTTPS everywhere
- Regular security audits

Performance Considerations

- Measure before optimizing
- Identify bottlenecks systematically
- Use appropriate caching strategies
- Optimize database queries

Common Anti-Patterns

Distributed Monolith

- Microservices that are tightly coupled
- Synchronous communication everywhere

- Shared databases across services
- Cannot deploy services independently

Database as Integration Point

- Multiple services accessing same database
- Shared database schema
- No clear service boundaries
- Difficult to scale and maintain

Chatty Interfaces

- Too many small network calls
- N+1 query problems
- Inefficient data transfer
- High latency and bandwidth usage

Synchronous Everything

- Blocking calls everywhere
- No asynchronous processing
- Poor user experience
- Cascading failures

Premature Optimization

- Optimizing before measuring
- Complex solutions for simple problems
- Over-engineering from the start
- Ignoring actual bottlenecks

Code Quality Best Practices

Clean Code Principles

- Meaningful naming
- Small, focused functions
- Clear comments where necessary
- Consistent formatting

Testing Strategy

- Unit tests for individual components
- Integration tests for service interactions
- End-to-end tests for critical paths
- Performance tests for scale

Documentation

- Architecture decision records
- API documentation
- Runbooks for operations
- Code comments for complex logic

Version Control

- Meaningful commit messages
- Feature branches
- Code reviews
- Continuous integration

Operational Excellence

Deployment Practices

- Blue-green deployments
- Canary releases
- Automated rollbacks
- Infrastructure as code

Monitoring and Alerting

- SLO-based alerting
- Runbooks for common issues
- On-call procedures
- Post-incident reviews

Capacity Planning

- Regular capacity reviews
- Load testing
- Auto-scaling policies

- Cost optimization

Security Practices

- Regular security updates
- Principle of least privilege
- Security scanning
- Incident response procedures

24. Future Trends in System Design

Emerging Technologies

Serverless Computing

- Function-as-a-Service (Faa