FAANG System Design Interview Guide

Master Large-Scale System Architecture

From Fundamentals to Complex Distributed Systems

OBJECTIVE: Design systems that scale to millions of users
COVERAGE: 25+ Core Concepts + Real Interview Questions
APPROACH: Structured methodology for system design
TARGET: Senior Engineer and Staff Engineer Positions

Based on 2024 FAANG Interview Analysis

September 28, 2025

Contents

1 System Design Interview Overview

1.1 What Interviewers Evaluate

System design interviews assess your ability to:

- **Architectural Thinking**: Break down complex problems into manageable components
- **Scalability Reasoning**: Design systems that handle growth from thousands to millions of users
- **Trade-off Analysis**: Make informed decisions between competing design choices
- **Communication**: Explain technical concepts clearly and drive the conversation
- **Practical Experience**: Demonstrate understanding of real-world constraints

1.2 Common Interview Format

System Design Framework

45-60 Minute Structure:

1. **Clarification (5-10 min):** Understand requirements and constraints
2. **High-level Design (10-15 min):** Architecture overview and main components
3. **Deep Dive (20-25 min):** Detailed design of critical components
4. **Scale & Optimize (10-15 min):** Handle bottlenecks and scale to millions
5. **Wrap-up (5 min):** Summary and additional considerations

1.3 Key Success Factors

- **Start Simple**: Begin with a basic design and evolve it
- **Think Out Loud**: Verbalize your thought process
- **Ask Questions**: Clarify ambiguities early and often
- **Consider Trade-offs**: Every design decision has pros and cons
- **Estimate Scale**: Use back-of-envelope calculations

2 Core System Design Concepts

2.1 1. Scalability

Scalability - Handling Growth

Definition: A system's ability to handle increased load gracefully
Types: Vertical (scale up) vs Horizontal (scale out)
Goal: Maintain performance as user base grows

2.1.1 Vertical Scaling (Scale Up)

- **Approach**: Add more power (CPU, RAM, Storage) to existing machine
- **Pros**: Simple, no code changes required, strong consistency
- **Cons**: Hardware limits, single point of failure, expensive
- **Use Cases**: Traditional databases, monolithic applications

2.1.2 Horizontal Scaling (Scale Out)

- **Approach**: Add more servers to handle increased load
- **Pros**: No hardware limits, fault tolerant, cost effective
- **Cons**: Complex implementation, eventual consistency, data consistency challenges
- **Use Cases**: Web servers, microservices, distributed databases

Scale Considerations

Scale Estimation Framework:

- Daily Active Users (DAU): 10M users
- Requests per day: 10M × 10 requests = 100M requests/day
- Requests per second: 100M / (24 × 3600) ≈ 1,200 RPS
- Peak load (3x average): 3,600 RPS
- Storage per user: 1KB × 10M = 10GB

2.2 2. Load Balancing

Load Balancing - Traffic Distribution

Purpose: Distribute incoming requests across multiple servers
Benefits: Prevents overload, improves availability, enables horizontal scaling
Types: Layer 4 (Transport) vs Layer 7 (Application)

2.2.1 Load Balancing Algorithms

- **Round Robin**: Requests distributed sequentially
- **Weighted Round Robin**: Servers get requests based on capacity
- **Least Connections**: Route to server with fewest active connections
- **IP Hash**: Route based on client IP hash
- **Geographic**: Route based on client location

2.2.2 Load Balancer Types

- **Layer 4 (Transport Layer)**:
  - Routes based on IP and port
  - Faster, lower latency
  - Cannot inspect application data
- **Layer 7 (Application Layer)**:
  - Routes based on HTTP headers, URLs, cookies
  - More intelligent routing decisions
  - Higher latency due to content inspection

2.3 3. Caching

Caching - Performance Optimization

Purpose: Store frequently accessed data in fast storage
Benefit: Reduces latency and database load
Principle: Locality of reference - recently accessed data likely to be accessed again

2.3.1 Cache Levels

- **Browser Cache**: Client-side caching (CSS, JS, images)
- **CDN**: Geographic distribution of static content
- **Reverse Proxy**: Nginx, Apache cache responses
- **Application Cache**: In-memory caching (Redis, Memcached)
- **Database Cache**: Query result caching

2.3.2 Cache Patterns

- **Cache-Aside (Lazy Loading)**:
  - Application manages cache directly
  - Read: Check cache → DB if miss → Update cache
  - Write: Update DB → Invalidate cache
- **Write-Through**:
  - Write to cache and DB simultaneously
  - Ensures cache consistency
  - Higher write latency
- **Write-Behind (Write-Back)**:
  - Write to cache immediately, DB asynchronously
  - Lower write latency
  - Risk of data loss
- **Refresh-Ahead**:
  - Proactively refresh cache before expiration
  - Good for predictable access patterns

2.3.3 Cache Eviction Policies

- **LRU (Least Recently Used)**: Remove oldest accessed item
- **LFU (Least Frequently Used)**: Remove least accessed item
- **FIFO (First In, First Out)**: Remove oldest item
- **TTL (Time To Live)**: Remove expired items

2.4 4. Database Design & Scaling

Database Scaling - Managing Data Growth

Challenge: Single database becomes bottleneck as data and traffic grow
Solutions: Replication, Sharding, Partitioning
Trade-offs: Consistency vs Availability vs Partition tolerance (CAP Theorem)

2.4.1 Database Replication

- **Master-Slave**:
  - One master (writes), multiple slaves (reads)
  - Improves read performance
  - Single point of failure for writes
- **Master-Master**:
  - Multiple masters, both read and write
  - Better availability
  - Complex conflict resolution

2.4.2 Database Sharding

- **Horizontal Partitioning**: Split data across multiple databases
- **Sharding Strategies**:
  - **Range-based**: Shard by ID ranges (1-1000, 1001-2000)
  - **Hash-based**: Use hash function on shard key
  - **Geographic**: Shard by user location
  - **Directory-based**: Lookup service to find shard
- **Challenges**: Cross-shard queries, rebalancing, hot spots

2.4.3 SQL vs NoSQL

Aspect	SQL (RDBMS)	NoSQL
Schema	Fixed schema, predefined structure	Flexible schema, dynamic structure
ACID Properties	Strong ACID compliance	Eventually consistent (BASE)
Scaling	Vertical scaling primarily	Horizontal scaling
Query Language	SQL (standardized)	Varied (document, key-value, graph)
Use Cases	Complex queries, transactions	Large scale, rapid development
Examples	MySQL, PostgreSQL, Oracle	MongoDB, Cassandra, Redis

2.5 5. Microservices Architecture

Microservices - Service Decomposition

Definition: Architectural style organizing application as loosely coupled services
Benefits: Independent deployment, technology diversity, fault isolation
Challenges: Network complexity, data consistency, service discovery

2.5.1 Microservices vs Monolith

2.5.2 Microservices Patterns

- **API Gateway**: Single entry point for all client requests
- **Service Discovery**: Mechanism to locate services dynamically
- **Circuit Breaker**: Prevent cascade failures
- **Bulkhead**: Isolate resources to prevent total system failure
- **Saga Pattern**: Manage distributed transactions

Aspect	Monolith	Microservices
Deployment	Single deployable unit	Independent service deployment
Technology Stack	Single technology stack	Technology per service
Development	Centralized development	Distributed teams
Testing	Easier integration testing	Complex end-to-end testing
Network	In-process communication	Network communication
Data Management	Shared database	Database per service

2.6 6. Message Queues & Event Streaming

Asynchronous Communication - Decoupling Services

Purpose: Enable services to communicate without direct coupling
Benefits: Improved scalability, reliability, and fault tolerance
Types: Message queues, Pub/Sub, Event streaming

2.6.1 Message Queue Patterns

- **Point-to-Point**: One producer, one consumer per message
- **Publish-Subscribe**: One producer, multiple consumers
- **Request-Reply**: Synchronous-like communication over async transport
- **Dead Letter Queue**: Handle failed message processing

2.6.2 Message Queue Technologies

- **RabbitMQ**: Traditional message broker with routing
- **Apache Kafka**: High-throughput event streaming platform
- **Amazon SQS**: Managed message queue service
- **Redis Pub/Sub**: Lightweight publish-subscribe messaging

2.7 7. Content Delivery Network (CDN)

CDN - Global Content Distribution

Purpose: Deliver content from geographically distributed servers
Benefits: Reduced latency, improved performance, reduced origin load
Content Types: Static assets, dynamic content, video streaming

2.7.1 CDN Strategies

- **Push CDN**: Content uploaded to CDN during deployment
- **Pull CDN**: Content cached on first request (origin pull)
- **Cache Invalidation**: Remove stale content from edge servers
- **Geographic Routing**: Route users to nearest edge server

2.8 8. Consistency Models

Data Consistency - Managing Distributed State

Challenge: Maintaining consistent data across multiple nodes
Spectrum: Strong consistency ↔ Eventual consistency
Trade-off: Consistency vs Availability vs Partition tolerance

2.8.1 Consistency Levels

- **Strong Consistency**: All nodes see same data simultaneously
- **Eventual Consistency**: System becomes consistent over time
- **Weak Consistency**: No guarantees about when data becomes consistent
- **Causal Consistency**: Maintains causal relationships between operations

2.8.2 CAP Theorem

- **Consistency**: All nodes return same data simultaneously
- **Availability**: System remains operational
- **Partition Tolerance**: System continues despite network failures
- **Reality**: Can only guarantee 2 out of 3 during network partitions

3 System Design Interview Framework

3.1 Step 1: Clarify Requirements (5-10 minutes)

System Design Framework

Functional Requirements:

- What specific features does the system need?
- What are the core user workflows?
- What actions can users perform?

Non-Functional Requirements:

- How many users will the system support?
- What is the expected read/write ratio?
- What are the latency requirements?
- What is the availability requirement (99.9%)?

Constraints & Assumptions:

- What is the scale (DAU, storage, bandwidth)?
- Are there any specific technology constraints?
- What is the budget/timeline?

3.1.1 Example Questions to Ask

- "How many daily active users are we expecting?"
- "What's the read to write ratio?"
- "Do we need to support real-time features?"
- "What's our availability target?"
- "Are there any compliance requirements?"
- "What devices/platforms need to be supported?"

3.2 Step 2: Back-of-Envelope Estimation (5 minutes)

Scale Considerations

Common Estimation Framework:

- **Users**: 100M DAU
- **Usage**: 10 requests per user per day
- **Total Requests**: 1B requests/day
- **QPS**: 1B / (24 × 3600) ≈ 11,600 QPS
- **Peak QPS**: 11,600 × 3 = 35,000 QPS
- **Storage**: 1KB per request × 1B = 1TB/day
- **Bandwidth**: 35K QPS × 1KB = 35MB/s

3.2.1 Quick Reference Numbers

- L1 cache reference: 0.5 ns
- L2 cache reference: 7 ns
- Main memory reference: 100 ns
- Send 1KB over network: 10 μs
- SSD random read: 150 μs
- Disk seek: 10 ms
- Network round trip: 500 μs (same datacenter)

3.3 Step 3: High-Level Design (10-15 minutes)

System Design Framework

Design Components:

1. **Client**: Web browser, mobile app
2. **Load Balancer**: Distribute traffic
3. **Web Servers**: Handle HTTP requests
4. **Application Servers**: Business logic
5. **Database**: Data storage
6. **Cache**: Performance optimization
7. **CDN**: Static content delivery

3.3.1 Standard Web Architecture

```
Client -> CDN -> Load Balancer -> Web Servers -> App Servers -> Database
                                      |
                                   Cache
```

3.4 Step 4: Deep Dive (20-25 minutes)

Focus on 2-3 critical components based on interviewer interest:

3.4.1 Database Design

- Define data schema and relationships
- Choose SQL vs NoSQL based on requirements
- Design indexing strategy
- Plan for data partitioning/sharding

3.4.2 API Design

- Define RESTful endpoints
- Specify request/response formats
- Consider authentication and authorization
- Plan for API versioning

3.4.3 Key Algorithms

- News feed generation algorithm
- Recommendation algorithm
- Search ranking algorithm
- Matching algorithm (for ride-sharing, dating)

3.5 Step 5: Scale the Design (10-15 minutes)

Scale Considerations

Bottleneck Identification:

- Database becomes read/write bottleneck
- Single points of failure
- Network bandwidth limitations
- Memory constraints
- Storage capacity limits

Scaling Solutions:

- Database replication and sharding
- Horizontal scaling of services
- Caching at multiple levels
- CDN for global distribution
- Message queues for async processing

3.6 Step 6: Address Follow-up Questions

Common follow-up topics:

- **Monitoring & Logging**: How to monitor system health
- **Security**: Authentication, authorization, data protection
- **Disaster Recovery**: Backup and recovery strategies

- **Analytics**: Data pipeline for business intelligence
- **A/B Testing**: Feature rollout and experimentation

4 Common System Design Questions

4.1 1. Design a URL Shortener (bit.ly)

4.1.1 Requirements

- Shorten long URLs to short URLs
- Redirect short URLs to original URLs
- Custom aliases support
- Analytics (click tracking)
- Scale: 100M URLs per day, 100:1 read/write ratio

4.1.2 High-Level Design

```
Client -> Load Balancer -> Web Servers -> Application Servers -> Database
                                                    |
                                                  Cache
```

4.1.3 Database Schema

```
URLs Table:
- short_url (varchar, primary key)
- long_url (varchar)
- user_id (int)
- created_at (timestamp)
- expires_at (timestamp)

Analytics Table:
- short_url (varchar)
- timestamp (timestamp)
- user_agent (varchar)
- ip_address (varchar)
```

4.1.4 Algorithm: Base62 Encoding

```
Characters: [a-z, A-Z, 0-9] = 62 characters
For 7-character short URL: 62^7 $\approx$ 3.5 trillion combinations

function encode(id):
    chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    result = ""
    while id > 0:
        result = chars[id % 62] + result
        id = id // 62
    return result
```

4.1.5 Scaling Considerations

- **Caching**: Cache popular URLs in Redis
- **Database Sharding**: Shard by short_url hash
- **Rate Limiting**: Prevent abuse
- **Analytics**: Separate service for click tracking

4.2 2. Design a Chat System (WhatsApp/Messenger)

4.2.1 Requirements

- One-on-one and group messaging
- Real-time message delivery
- Message history storage
- Online presence indicators
- Push notifications
- Scale: 1B users, 10B messages per day

4.2.2 High-Level Architecture

```
Mobile Client <-> Load Balancer <-> Chat Servers <-> Message Queue
                                          |
                                   Message Database
                                          |
                                    Push Notification Service
```

4.2.3 Real-Time Communication

- **WebSocket**: Persistent connection for real-time messaging
- **Long Polling**: Alternative for environments without WebSocket
- **Server-Sent Events (SSE)**: One-way communication from server

4.2.4 Message Storage

```
Messages Table:
- message_id (uuid, primary key)
- chat_id (uuid)
- sender_id (uuid)
- content (text)
- message_type (enum: text, image, video)
- timestamp (timestamp)
- status (enum: sent, delivered, read)

Chats Table:
- chat_id (uuid, primary key)
- type (enum: one_on_one, group)
- created_at (timestamp)
- last_message_id (uuid)

Participants Table:
- chat_id (uuid)
- user_id (uuid)
- joined_at (timestamp)
- role (enum: admin, member)
```

4.2.5 Scaling Strategies

- **Message Sharding**: Shard messages by chat_id
- **Connection Management**: Use connection pools and load balancing
- **Message Queue**: Apache Kafka for message distribution
- **Caching**: Cache recent messages and user sessions

4.3 3. Design a Social Media Feed (Instagram/Facebook)

4.3.1 Requirements

- Users can create posts (text, images, videos)
- Users can follow other users
- Generate personalized news feed
- Like, comment, and share posts
- Scale: 1B users, 500M daily posts

4.3.2 Core Components

- **User Service**: Manage user profiles and relationships
- **Post Service**: Handle post creation and storage
- **Feed Generation Service**: Create personalized feeds
- **Notification Service**: Handle likes, comments, follows

4.3.3 Feed Generation Approaches

Pull Model (Fan-out on Read):

- Generate feed when user requests it
- Query posts from all followed users
- Pros: Less storage, real-time updates
- Cons: Slow for users following many people

Push Model (Fan-out on Write):

- Pre-generate feeds when posts are created
- Store feeds in user's feed cache
- Pros: Fast feed retrieval
- Cons: High storage cost, celebrity problem

Hybrid Approach:

- Pull model for celebrities (users with many followers)
- Push model for regular users
- Merge results in real-time

4.3.4 Database Design

```
Users Table:
- user_id (uuid, primary key)
- username (varchar, unique)
- email (varchar, unique)
- profile_picture_url (varchar)

Posts Table:
- post_id (uuid, primary key)
- user_id (uuid)
- content (text)
- media_urls (json array)
```

```
- created_at (timestamp)
- likes_count (int)
- comments_count (int)

Follows Table:
- follower_id (uuid)
- followee_id (uuid)
- created_at (timestamp)
- primary key (follower_id, followee_id)

Feeds Table (for push model):
- user_id (uuid)
- post_id (uuid)
- created_at (timestamp)
- primary key (user_id, post_id)
```

4.4 4. Design a Video Streaming Service (YouTube/Netflix)

4.4.1 Requirements

- Upload and stream videos
- Support multiple video qualities
- Global content delivery
- Video recommendations
- View analytics
- Scale: 1B users, 500 hours uploaded per minute

4.4.2 Architecture Components

- **Upload Service**: Handle video uploads and processing
- **Encoding Service**: Convert videos to multiple formats
- **CDN**: Global video distribution
- **Metadata Service**: Store video information
- **Recommendation Service**: Generate personalized recommendations

4.4.3 Video Processing Pipeline

```
Upload -> Validation -> Encoding -> Storage -> CDN Distribution
          |              |          |
      Metadata      Thumbnails   Search Index
```

4.4.4 Video Encoding

- **Multiple Resolutions**: 144p, 240p, 360p, 480p, 720p, 1080p, 4K
- **Adaptive Bitrate**: Adjust quality based on network conditions
- **Codecs**: H.264, H.265, VP9 for compression
- **Containers**: MP4, WebM for different platforms

4.4.5 Storage Strategy

- **Object Storage**: Amazon S3, Google Cloud Storage for video files
- **CDN**: CloudFront, CloudFlare for global distribution
- **Caching**: Cache popular videos at edge locations
- **Tiered Storage**: Hot, warm, and cold storage based on popularity

5 Advanced Topics

5.1 Distributed System Patterns

5.1.1 Circuit Breaker Pattern

- **Purpose**: Prevent cascade failures in distributed systems
- **States**: Closed (normal), Open (failing), Half-open (testing)
- **Implementation**: Monitor failure rate, trip circuit when threshold exceeded
- **Recovery**: Gradually test service recovery

5.1.2 Bulkhead Pattern

- **Purpose**: Isolate resources to prevent total system failure
- **Implementation**: Separate connection pools, thread pools, resources
- **Example**: Separate database connections for read/write operations

5.1.3 Saga Pattern

- **Purpose**: Manage distributed transactions across microservices
- **Types**: Choreography vs Orchestration
- **Compensation**: Define compensation actions for failed steps

5.2 Security Considerations

5.2.1 Authentication & Authorization

- **OAuth 2.0**: Industry standard for authorization
- **JWT**: JSON Web Tokens for stateless authentication
- **API Keys**: Simple authentication for service-to-service
- **Multi-factor Authentication**: Additional security layer

5.2.2 Data Protection

- **Encryption**: Data at rest and in transit
- **HTTPS**: TLS encryption for web traffic
- **Database Encryption**: Encrypt sensitive data fields
- **Key Management**: Secure key storage and rotation

5.2.3 Common Security Threats

- **SQL Injection**: Use parameterized queries
- **XSS**: Sanitize user inputs
- **CSRF**: Use CSRF tokens
- **DDoS**: Rate limiting and traffic filtering

5.3 Monitoring & Observability

5.3.1 Metrics

- **RED Method**: Rate, Errors, Duration
- **USE Method**: Utilization, Saturation, Errors
- **SLIs**: Service Level Indicators (latency, availability, throughput)
- **SLOs**: Service Level Objectives (99.9% availability)

5.3.2 Logging

- **Structured Logging**: Use JSON format for machine parsing
- **Log Levels**: Debug, Info, Warning, Error, Fatal
- **Centralized Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)
- **Log Aggregation**: Collect logs from all services

5.3.3 Tracing

- **Distributed Tracing**: Track requests across multiple services
- **Trace ID**: Unique identifier for request flow
- **Span**: Individual operation within a trace
- **Tools**: Jaeger, Zipkin, AWS X-Ray

6 Integrated Study Schedule

6.1 How to Combine with Coding Interview Prep

System Design Framework

Integrated 6-Week Schedule:

- **Weeks 1-4**: Focus on coding patterns (existing schedule)
- **Weeks 5-6**: Intensive system design preparation
- **Daily**: 15 minutes system design concepts review
- **Weekly**: 2-3 hours dedicated system design practice

6.2 Week-by-Week Integration

6.2.1 Weeks 1-2: Foundation + Basic Concepts

Coding Focus: Two Pointers, Sliding Window, DFS, BFS
System Design: 15 min/day reading fundamental concepts

- Day 1: Scalability concepts
- Day 2: Load balancing
- Day 3: Caching strategies
- Day 4: Database basics
- Day 5: Review and practice estimation
- Weekend: Design a simple URL shortener (1 hour)

6.2.2 Weeks 3-4: Algorithms + Intermediate Concepts

Coding Focus: Binary Search, Dynamic Programming, Backtracking
System Design: 15 min/day + 1 design session per week

- Day 1: Microservices architecture
- Day 2: Message queues
- Day 3: CDN and caching
- Day 4: Database sharding
- Day 5: CAP theorem
- Weekend: Design a chat system (1.5 hours)

6.2.3 Week 5: Advanced Data Structures + System Design Deep Dive

Coding Focus: Heaps, Union-Find, Trie
System Design: 30 min/day + 2 design sessions

- Day 1: Design social media feed
- Day 2: Design video streaming service
- Day 3: Design search engine
- Day 4: Design ride-sharing service
- Day 5: Design e-commerce platform
- Weekend: Mock system design interview (2 hours)

6.2.4 Week 6: Special Algorithms + System Design Mastery

Coding Focus: Topological Sort, Special Algorithms
System Design: 45 min/day + daily practice

- Day 1: Design distributed cache
- Day 2: Design notification system
- Day 3: Design analytics platform
- Day 4: Design file storage system
- Day 5: Design monitoring system
- Weekend: Full mock interviews (3 hours)

6.3 Daily Routine Enhancement

6.3.1 Morning Routine (45 minutes total)

- **Coding Templates (15 min):** Write 2-3 templates from memory
- **System Design Concepts (15 min):** Review 1-2 fundamental concepts
- **Estimation Practice (15 min):** Quick back-of-envelope calculations

6.3.2 Evening Session (90-120 minutes total)

- **Coding Problems (60 min):** 2-3 coding problems
- **System Design Practice (30-60 min):**
  - Weeks 1-4: Concept review and simple designs
  - Weeks 5-6: Full system design practice

7 System Design Practice Framework

7.1 30-Minute Practice Session Structure

System Design Framework

Quick Practice Format:

1. **Problem Statement (2 min):** Read and understand requirements
2. **Clarification (3 min):** List key questions and assumptions
3. **High-level Design (10 min):** Draw main components
4. **Deep Dive (10 min):** Focus on 1-2 critical components
5. **Scaling (5 min):** Identify bottlenecks and solutions

7.2 60-Minute Full Practice Session

System Design Framework

Complete Interview Simulation:

1. **Requirements Gathering (10 min):** Thorough clarification
2. **Estimation (5 min):** Back-of-envelope calculations
3. **High-level Design (15 min):** Complete architecture diagram
4. **Detailed Design (20 min):** Database schema, APIs, algorithms
5. **Scaling & Optimization (10 min):** Handle growth and bottlenecks

7.3 Self-Evaluation Checklist

After each practice session, evaluate:

☐ Did I ask clarifying questions?

☐ Did I estimate scale and performance requirements?

☐ Did I start with a simple design and evolve it?

- ☐ Did I consider trade-offs for each design decision?
- ☐ Did I identify and address potential bottlenecks?
- ☐ Did I discuss monitoring and failure scenarios?
- ☐ Did I stay within the time limit?

8 Company-Specific Preparation

8.1 Google

- **Focus**: Scalability, distributed systems, algorithms
- **Common Questions**: Design Google Search, Gmail, Google Drive
- **Key Concepts**: MapReduce, BigTable, distributed consensus
- **Prep Time**: 2-3 weeks dedicated system design study

8.2 Amazon

- **Focus**: Service-oriented architecture, reliability, cost optimization
- **Common Questions**: Design Amazon cart, recommendation system, inventory
- **Key Concepts**: Microservices, DynamoDB, eventual consistency
- **Leadership Principles**: Customer obsession, ownership, bias for action

8.3 Meta (Facebook)

- **Focus**: Social features, real-time systems, mobile-first
- **Common Questions**: Design Facebook feed, Messenger, Instagram
- **Key Concepts**: Graph databases, real-time messaging, content ranking
- **Scale**: Billion-user scale considerations

8.4 Netflix

- **Focus**: Content delivery, personalization, streaming
- **Common Questions**: Design video streaming, recommendation engine
- **Key Concepts**: CDN optimization, A/B testing, microservices
- **Specialization**: Media processing and global distribution

9 Final Preparation Tips

9.1 Common Mistakes to Avoid

- Jumping into details without understanding requirements
- Over-engineering the initial solution
- Ignoring non-functional requirements
- Not considering failure scenarios
- Focusing on implementation details too early
- Not communicating trade-offs clearly

9.2 Day Before Interview

- Review fundamental concepts (30 minutes)
- Practice one complete system design (60 minutes)
- Review common estimation numbers
- Prepare questions to ask the interviewer
- Get good sleep and stay confident

9.3 During the Interview

- Ask clarifying questions before designing
- Think out loud and explain your reasoning
- Start simple and build complexity gradually
- Draw diagrams to visualize your design
- Discuss trade-offs for major decisions
- Be prepared to defend your choices
- Stay calm if you don't know something - reason through it

Trade-offs Analysis

Remember: System design interviews are about demonstrating your thought process, not finding the "perfect" solution. Show how you approach complex problems, make trade-offs, and design systems that can evolve with changing requirements.

Success Formula: Fundamentals + Practice + Communication
Master the concepts, practice regularly, and think out loud!