System Design Interview Questions with Sample Answers

This document provides system design interview questions along with comprehensive answers and architectural approaches.

System Design Framework

1. **Clarify Requirements** - Understand functional and non-functional requirements
2. **Estimate Scale** - Calculate data volume, QPS, storage needs
3. **High-Level Design** - Create overall architecture diagram
4. **Detailed Design** - Deep dive into critical components
5. **Scale the Design** - Address bottlenecks and scaling challenges
6. **Discuss Tradeoffs** - Analyze different approaches and their implications

---

Data & AI/ML Systems

1. Design a machine learning model monitoring system

Difficulty: Medium

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                    ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

2. Design a multi-tenant analytics platform

Difficulty: Hard

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                    ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

3. Design a data quality monitoring system

Difficulty: Medium

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                    ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

Distributed Systems & Infrastructure

1. Design a distributed auto-scaling system

Difficulty: Medium

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                    ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

2. Design a distributed health check system

Difficulty: Easy

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                          ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

3. Design a distributed circuit breaker system

Difficulty: Medium

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                  ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

Product & Platform Systems

1. Design a password manager platform

Difficulty: Medium

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                       ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

2. Design a photo storage platform like Google Photos

Difficulty: Medium

Company: Google

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                       ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

3. Design a news aggregation platform like Reddit

Difficulty: Medium

Company: Reddit

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                    ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

Real-time & Communication Systems

1. Design a real-time social media feed

Difficulty: Hard

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                     ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

2. Design a real-time price monitoring system

Difficulty: Medium

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                     ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---

3. Design a real-time collaborative code editor

Difficulty: Hard

Company: General

Sample System Design Answer:

1. Requirements Clarification: - Functional: Core features and user interactions - Non-functional: Scale (users, data), performance, availability - Constraints: Budget, timeline, existing systems

2. Scale Estimation: - Daily Active Users: 10M - Read/Write Ratio: 100:1 - Storage: 1TB per year - Bandwidth: 1000 QPS peak

3. High-Level Architecture:

```
[Client] → [Load Balancer] → [API Gateway] → [Microservices]
                                                  ↓
[Cache Layer] ← [Database Cluster] ← [Message Queue]
```

4. Detailed Components: - **Load Balancer:** Nginx/HAProxy for traffic distribution - **API Gateway:** Rate limiting, authentication, routing - **Microservices:** Domain-specific services with clear boundaries - **Database:** Primary-replica setup with sharding - **Cache:** Redis/Memcached for frequently accessed data - **Message Queue:** Kafka/RabbitMQ for async processing

5. Scaling Strategies: - Horizontal scaling of stateless services - Database sharding by user ID or geographic region - CDN for static content delivery - Auto-scaling based on metrics

6. Key Tradeoffs: - Consistency vs Availability (CAP theorem) - SQL vs NoSQL based on data structure - Synchronous vs Asynchronous processing - Cost vs Performance optimization

---