Contents

System Design Approach and Key Questions

This document outlines a structured approach and key questions to consider when tackling system design interviews. Use this as a checklist and guide to demonstrate clear, methodical thinking during interviews.

Step-by-Step Approach to Solve Multiple Design Problems

This systematic methodology offers a comprehensive framework for approaching any system design interview. Each phase builds upon previous work, ensuring thorough coverage while showcasing structured analytical thinking to interviewers.

Step 1: Requirements Clarification

Begin by thoroughly understanding the problem scope and objectives. System design problems are typically open-ended with multiple valid solutions, making early clarification essential for interview success. Focus on defining clear goals and boundaries within the available timeframe.

Distinguish Between Functional and Non-Functional Requirements: - Functional Requirements: Define *what* the system should do - the specific features, capabilities, and behaviors that users can directly interact with or observe. These are the core business logic and user-facing features. - Examples: User registration, posting content, searching data, sending notifications, processing payments - **Non-Functional Requirements:** Define *how well* the system should perform - the quality attributes, constraints, and operational characteristics that affect user experience but aren't direct features. - Examples: Performance (latency, throughput), scalability, availability, security, consistency, maintainability

Key Areas to Clarify: - Core user workflows and feature scope - Expected system scale and performance targets - Data consistency and availability requirements - Security and compliance constraints - Integration and platform requirements

Requirement Prioritization Strategy: Always prioritize both functional and non-functional requirements early in the discussion. **This establishes a clear contract between you and the interviewer** about what will be designed and what success looks like: - **Functional Priorities:** Identify must-have features vs. nice-to-have features for MVP and future iterations - **Non-Functional Priorities:** Determine critical quality attributes (e.g., is 99.9% availability more important than sub-100ms latency?) - **PACELC Theorem Application:** When network partitions occur, choose between Consistency and Availability (CAP). Even when the system is running normally (Else), choose between Latency and Consistency. This framework helps prioritize trade-offs systematically. - **Business Impact Assessment:** Understand which requirements directly affect user experience and business success - **Resource Constraints:** Consider time, budget, and team expertise when ranking requirements - **Risk Evaluation:** Prioritize requirements that mitigate the highest business and technical risks

Interview Contract Establishment: Use this phase to create mutual understanding and agreement with your interviewer on: - **Scope boundaries:** What features and components will be included vs. excluded - **Success criteria:** How the system's effectiveness will be measured - **Design constraints:** Technical, business, and resource limitations that guide decisions - **Assumption validation:** Confirm any assumptions about user behavior, traffic patterns, or infrastructure

Step 2: Capacity Planning and Scale Estimation

Calculate the expected system scale including user traffic, data storage requirements, and network bandwidth needs. These estimations inform subsequent architectural decisions around scaling strategies, data partitioning, and infrastructure planning.

Step 3: API and Interface Design

Specify the expected system interfaces and contracts. This establishes clear boundaries and validates requirement understanding. Consider modern API design principles including RESTful patterns, pagination strategies, versioning approaches, and rate limiting mechanisms.

Step 4: Data Architecture and Modeling

Define the system's data entities, their interconnections, and information flow patterns. Address database technology selection (relational vs non-relational), storage infrastructure, and data lifecycle management considerations.

Step 5: System Architecture Overview

Create a high-level architectural diagram showing 5-6 primary system components. Include essential elements like traffic distribution, application logic, data persistence, content storage, performance optimization layers, and specialized services.

Step 6: Deep Dive Design Analysis

Examine 2-3 critical components in detail based on interviewer guidance. Compare alternative approaches with their respective advantages and limitations. Address data distribution strategies, performance optimization techniques, and edge case handling.

Step 7: Risk Assessment and Mitigation

Identify potential system vulnerabilities and propose mitigation strategies. Evaluate failure scenarios, redundancy requirements, operational monitoring needs, and performance optimization opportunities.

Interview Balance Check

Throughout the interview, regularly ensure you're maintaining balance: - Have I covered the core user flows? (Functional) - Have I addressed scalability and performance? (Non-functional) - Have I considered failure scenarios? (Non-functional) - Have I thought about the operational aspects? (Non-functional) - Have I discussed trade-offs for major decisions? - Have I referenced real-world examples or past experience?

Essential System Design Components & Trade-offs

These are fundamental building blocks you should understand and consider when designing systems. Each component category includes both the technical components and their key trade-offs to help you make informed architectural decisions.

Data Storage & Management

Components: - SQL vs NoSQL Databases (PostgreSQL, MongoDB, Redis, Cassandra) - Data Partitioning & Sharding strategies (Horizontal, Vertical, Functional, Consistent Hashing) - Indexes (Primary, Secondary, Composite) and their trade-offs - Replication (Primary-Replica, Peer-to-Peer, Synchronous/Asynchronous) - Queues & Message Systems (Kafka, RabbitMQ, SQS) - Real-time Communication (Polling, WebSockets, Server-Sent Events)

Key Trade-offs: - **SQL vs. NoSQL Databases** - Structured integrity vs. flexible scalability - **Strong vs. Eventual Consistency** - Data accuracy vs. performance and availability - **Polling vs. Long-Polling vs. WebSockets vs. Webhooks** - Simplicity vs. real-time performance

Performance & Scalability

Components: - Caching strategies and types (Client-side, CDN, Application-level, Database) - Cache invalidation methods and schemes (TTL, Write-through, Cache-aside) - Cache eviction policies (LRU, LFU, FIFO) - Performance vs Scalability fundamentals and key distinctions

Key Trade-offs: - **Performance vs. Scalability** - Current speed vs. future growth capacity - **Latency vs. Throughput** - Individual operation speed vs. total processing capacity - **Read-Through vs Write-Through Cache** - Read optimization vs. write consistency - **Server-Side vs. Client-Side Caching** - Centralized control vs. user experience - **CDN Usage vs. Direct Server Serving** - Global performance vs. simplicity

Network Infrastructure & Traffic Management

Components: - Load Balancing (Layer 4/7, algorithms, types) - Proxies (Forward, Reverse, examples) - DNS & Content Delivery (DNS hierarchy, Anycast, CDNs) - API Gateway (purpose, features, trade-offs) - Service Exposure Patterns (Direct vs Gateway/Proxy)

Key Trade-offs: - **Load Balancer vs. API Gateway** - Simple traffic distribution vs. comprehensive API management - **Direct Service Exposure vs. Gateway/Proxy Layer** - Performance vs. centralized control - **API Gateway vs. Reverse Proxy** - Application-aware features vs. high-performance traffic handling

API Design & Communication Patterns

Components: - API Design Patterns (REST, RPC, GraphQL, Webhooks) - API Versioning Strategies (URL, Header, Query Parameter, Content Negotiation)

Key Trade-offs: - **Easy-to-Build APIs vs. Long-Term APIs** - Short-term velocity vs. long-term maintainability - **REST vs. RPC** - Resource-oriented simplicity vs. action-oriented flexibility

Architecture Patterns & State Management

Components: - Stateful vs Stateless Design (characteristics, benefits, challenges) - Client-Server Logic Distribution (Thick Client/Server, Hybrid approaches)

Key Trade-offs: - **UI Complexity vs. Server Complexity** - Client responsiveness vs. server control - **Stateful vs. Stateless Architecture** - Rich UX vs. horizontal scalability

Data Processing Patterns

Components: - Batch Processing vs Stream Processing - Lambda Architecture (hybrid approach) - When to use each processing pattern

Key Trade-offs: - **Batch Processing vs. Stream Processing** - High throughput efficiency vs. low latency responsiveness

---

Trade-off Decision Framework

When evaluating any trade-off, consistently ask: 1. **Current vs. Future Needs:** Are we optimizing for immediate requirements or long-term scalability? 2. **User Impact:** Which choice better serves the end-user experience and business objectives? 3. **Technical Constraints:** What are our team's capabilities, infrastructure limitations, and resource constraints? 4. **Risk Assessment:** What are the failure modes and mitigation strategies for each approach? 5. **Cost Analysis:** What are the development, operational, and maintenance costs over time? 6. **Flexibility:** Which option provides more adaptability for future changes and requirements?