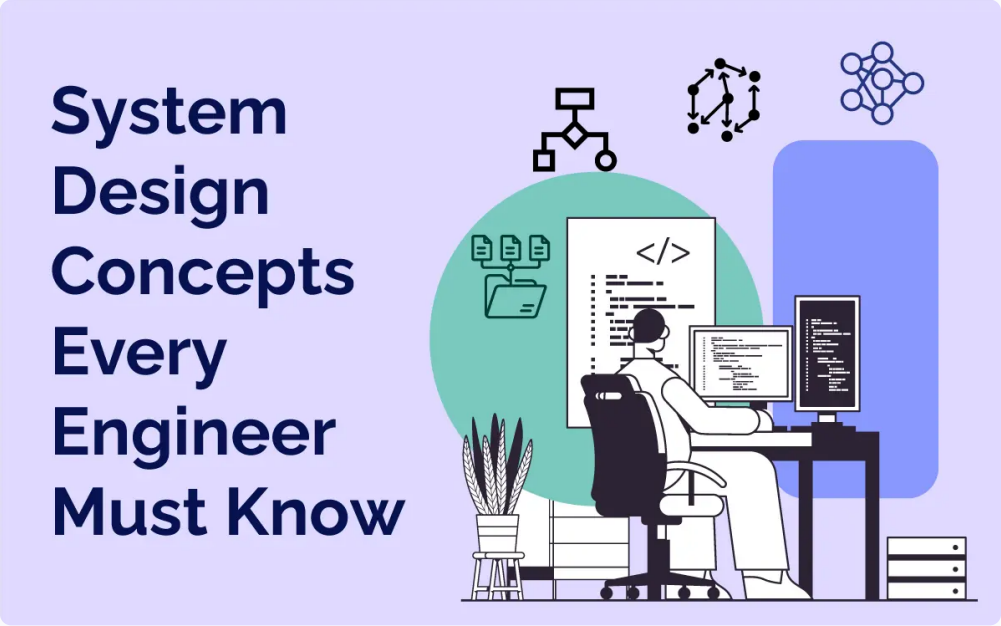
25 Fundamental System Design Concepts Engineers Must Know Before the Interview

Arslan Ahmad
September 11th, 2025

Preparing for a system design interview? Learn 25 fundamental system design concepts – from caching and load balancing to the CAP theorem – and get the insights you need to ace your interview.



ON THIS PAGE

This blog covers the 25 most essential system design concepts that every software engineer should understand before an interview. From caching and load balancing to consistency models and sharding, each topic is explained with real-world relevance—perfect for beginners preparing for FAANG interviews.

Confused by system design concepts?

You're not alone.

If terms like load balancing, sharding, or CAP theorem sound intimidating—you're exactly who this guide is for.

Understanding system design fundamentals, such as **Load Balancing**, **Caching**, **Partitioning**, **Replication**, **Databases**, and **Proxies** is important for any system design interview.

Based on my own experiences, I have identified 25 key concepts that can greatly improve your ability to tackle system design problems.

These concepts include understanding the intricacies of API gateway, mastering load-balancing techniques, grasping the importance of CDNs, and appreciating the role of caching in modern distributed systems.

By the end of this blog, you will have a comprehensive understanding of these essential system design concepts.

Here is a quick definition of each:

25 System Design Fundamentals

1. **Domain Name System (DNS)** – Translates human-friendly website names (like google.com) into IP addresses computers use.
2. **Load Balancer** – Distributes incoming traffic across multiple servers to keep the system fast and reliable.
3. **API Gateway** – A single entry point that routes client requests to backend services in a microservices system.
4. **CDN (Content Delivery Network)** – A network of servers that delivers static content (like images or videos) from locations closer to the user.
5. **Forward Proxy vs Reverse Proxy** – A forward proxy hides the client from the server; a reverse proxy hides the server from the client.
6. **Caching** – Temporarily stores frequently used data to speed up future access.
7. **Data Partitioning** – Splits large datasets into smaller chunks (shards) to spread the load across servers.
8. **Database Replication** – Creates and maintains copies of data across different servers for better availability and fault tolerance.
9. **Distributed Messaging Systems** – Enable services to communicate by sending messages asynchronously via a queue (e.g. Kafka, RabbitMQ).
10. **Microservices** – A system design where different features are split into small, independent services that work together.
11. **NoSQL Database** – A flexible, schema-less database designed for fast storage and retrieval of large amounts of data (e.g. MongoDB).
12. **Database Index** – Like a table of contents for a database that helps find data faster.
13. **Distributed File Systems** – A system that lets you store and access files across many servers as if they were on one machine.
14. **Notification System** – Sends alerts or updates to users through email, push notifications, or SMS based on events.
15. **Full-Text Search** – Lets you search for words or phrases in documents or data like Google search does.
16. **Distributed Coordination Services** – Tools (like ZooKeeper) that help manage and synchronize tasks across distributed systems.

The Observer vs Pub-Sub Pattern: Understanding Event...
Arslan Ahmad Sep 8th, 2025

Mastering the Meta Technical Screen: A Comprehensive Guide fo...
Arslan Ahmad Feb 3rd, 2024

17. **Heartbeat** – A small, regular signal sent between components to check if they're still alive and responsive.
18. **Checksum** – A small piece of data used to detect errors in files or messages.
19. **Latency vs Throughput** – Latency is the time it takes to process one request; throughput is how many requests are handled per second.
20. **Availability vs Reliability** – Availability means the system is up and running; reliability means it works correctly over time without failure.
21. **CAP Theorem** – In a distributed system, you can only guarantee two out of three: consistency, availability, and partition tolerance.
22. **Consistency Patterns** – Strategies for making sure all users see the same data, either instantly or eventually.
23. **Scalability** – The ability of a system to handle increasing traffic, data, or users without breaking or slowing down.
24. **Long Polling vs WebSockets** – Long polling repeatedly asks the server for updates; WebSockets keep a live connection for real-time communication.
25. **Rate Limiting** – Controls how many requests a user or client can make in a certain time to protect the system from abuse.

Mastering these system design concepts will give you an edge in any system design interview. Let's cover each one of these in detail.

1. Domain Name System (DNS)

The Domain Name System (DNS) serves as a fundamental component of the internet infrastructure, translating user-friendly domain names into their corresponding IP addresses.

It acts as a phonebook for the internet, enabling users to access websites and services by entering easily memorable domain names, such as www.designgurus.io, rather than the numerical IP addresses like "192.0.2.1" that computers utilize to identify each other.

When you input a domain name into your web browser, the DNS is responsible for finding the associated IP address and directing your request to the appropriate server.

This process begins with your computer sending a query to a recursive resolver, which then searches a series of DNS servers, beginning with the root server, followed by the Top-Level Domain (TLD) server, and ultimately the authoritative name server.

Once the IP address is located, the recursive resolver returns it to your computer, allowing your browser to establish a connection with the target server and access the desired content.

DNS

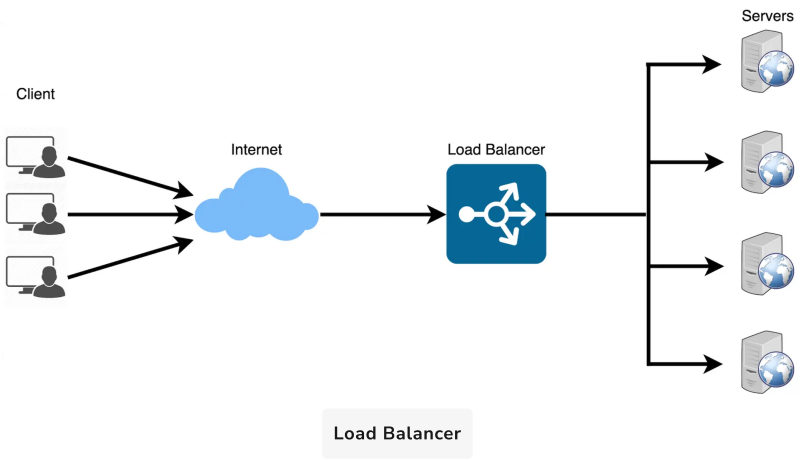
2. Load Balancer

A load balancer is a networking device or software designed to distribute incoming network traffic across multiple servers, ensuring optimal resource utilization, reduced latency, and maintained high availability. It plays a crucial role in scaling applications and efficiently managing server workloads, particularly in situations where there is a sudden surge in traffic or uneven distribution of requests among servers.

Load balancers employ various load balancing algorithms to determine the distribution of incoming traffic. Some common algorithms include:

- **Round Robin:** Requests are sequentially and evenly distributed across all available servers in a cyclical manner.
- **Least Connections:** The load balancer assigns requests to the server with the fewest active connections, giving priority to less-busy servers.
- **IP Hash:** The client's IP address is hashed, and the resulting value is used to determine which server the request should be directed to. This method ensures that a specific client's requests are consistently routed to the same server, helping maintain session persistence.

Learn more about the concept of Load Balancing.



Load Balancer

3. API Gateway

API Gateway is pivotal in system design - it serves as a server or service that functions as an intermediary between external clients and the internal microservices or API-based backend services of an application. It is a vital component in contemporary architectures, particularly in microservices-based systems, where it streamlines the communication process and offers a single entry point for clients to access various services.

The primary functions of an API Gateway encompass:

1. Request Routing: The API Gateway directs incoming API requests from clients to the appropriate backend service or microservice, based on predefined rules and configurations.
2. Authentication and Authorization: The API Gateway manages user authentication and authorization, ensuring that only authorized clients can access the services. It verifies API keys, tokens, or other credentials before routing requests to the backend services.

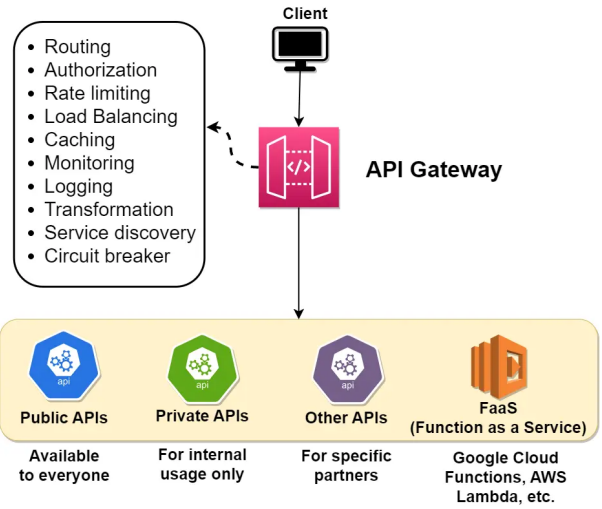
3. Rate Limiting and Throttling: To safeguard backend services from excessive load or abuse, the API Gateway enforces rate limits or throttles requests from clients according to predefined policies. Learn about common rate limiting algorithms.

4. Caching: In order to minimize latency and backend load, the API Gateway caches frequently-used responses, serving them directly to clients without the need to query the backend services.

5. Request and Response Transformation: The API Gateway can modify requests and responses, such as converting data formats, adding or removing headers, or altering query parameters, to ensure compatibility between clients and services.

Integrating an API Gateway is a strategic system design move, particularly for microservices architecture.

It simplifies software design, enhances security, and manages traffic effectively.



API Gateway

> Check Grokking System Design Fundamentals course for a list of common system design concepts.

4. CDN

A Content Delivery Network (CDN) is a distributed network of servers that store and deliver content, such as images, videos, stylesheets, and scripts, to users from locations that are geographically closer to them.

CDNs are designed to enhance the performance, speed, and reliability of content delivery to end-users, irrespective of their location relative to the origin server.

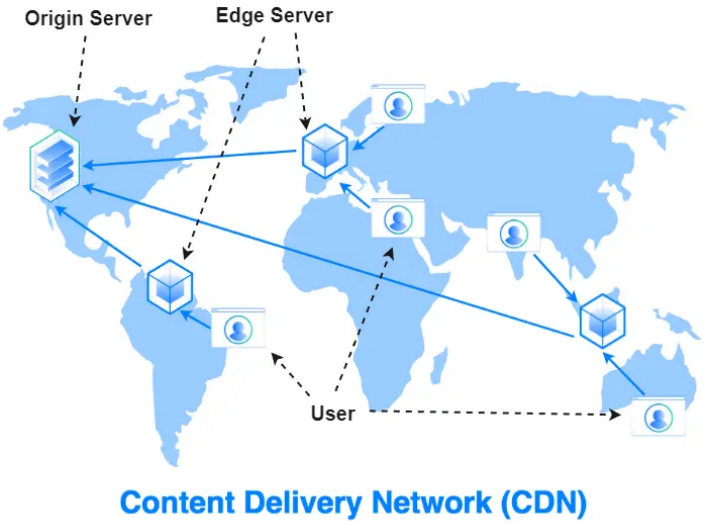
Here's how CDN operates:

1. When a user requests content from a website or application, the request is directed to the nearest CDN server, also known as an edge server.

2. If the edge server has the requested content cached, it directly serves the content to the user. This process reduces latency and improves the user experience, as the content travels a shorter distance.

3. If the content is not cached on the edge server, the CDN retrieves it from the origin server or another nearby CDN server. Once the content is fetched, it is cached on the edge server and served to the user.

4. To ensure the content stays up-to-date, the CDN periodically checks the origin server for changes and updates its cache accordingly.



CDN

5. Forward Proxy vs Reverse Proxy

A forward proxy, also referred to as a "proxy server" or simply "proxy," is a server positioned in front of one or more client machines, acting as an intermediary between the clients and the internet.

When a client machine requests a resource on the internet, the request is initially sent to the forward proxy.

The forward proxy then forwards the request to the internet on behalf of the client machine and returns the response to the client machine.

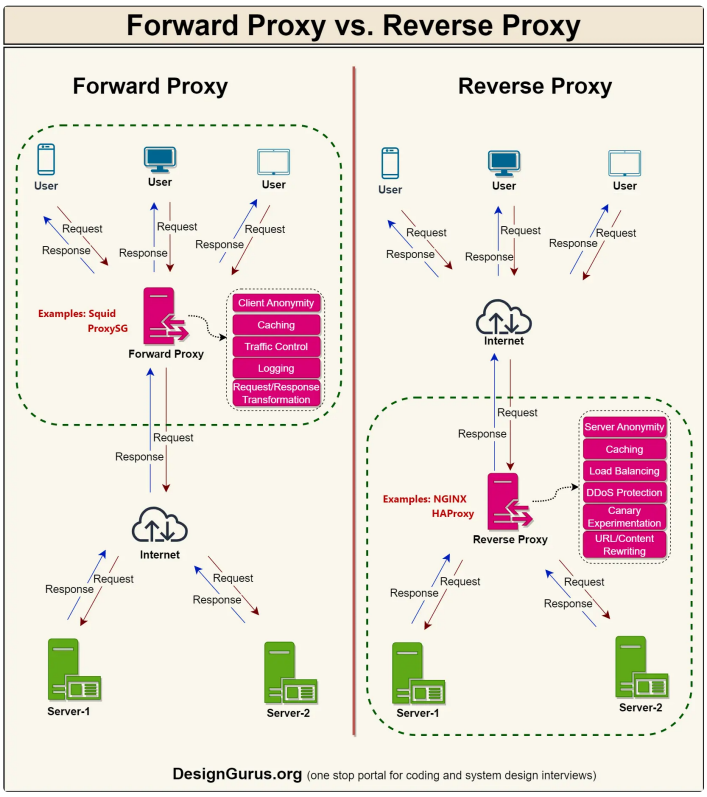
On the other hand, a reverse proxy is a server that sits in front of one or more web servers, serving as an intermediary between the web servers and the internet.

When a client requests a resource on the internet, the request is first sent to the reverse proxy.

The reverse proxy then forwards the request to one of the web servers, which returns the response to the reverse proxy.

Finally, the reverse proxy returns the response to the client.

For more details, see difference between proxy and reverse proxy.



Forward Proxy vs. Reverse Proxy

> Check Grokking System Design Fundamentals for a list of common system design concepts.

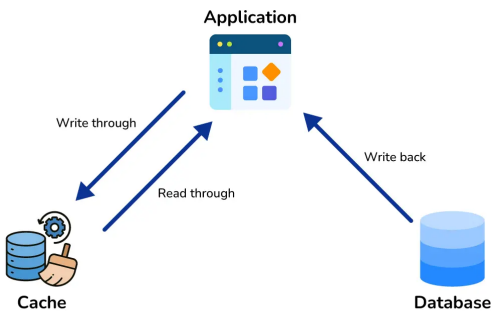
6. Caching: A Foundational System Design Concept

A cache is a high-speed storage layer positioned between the application and the original data source, such as a database, file system, or remote web service.

When an application requests data, the cache is checked first. If the data is present in the cache, it is returned to the application.

If the data is not found in the cache, it is retrieved from its original source, stored in the cache for future use, and then returned to the application.

In a distributed system, caching can occur in multiple locations, including the client, DNS, CDN, load balancer, API gateway, server, database, and more.

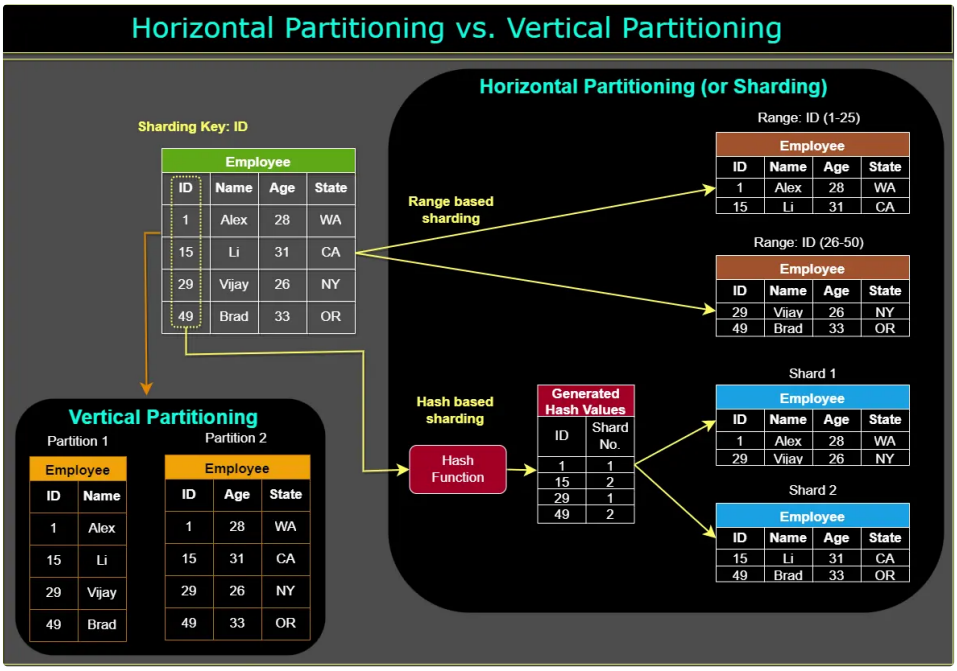


Caching

7. Data Partitioning

In a database, **horizontal partitioning**, often referred to as **sharding**, entails dividing the rows of a table into smaller tables and storing them on distinct servers or database instances. This method is employed to distribute the database load across multiple servers, thereby enhancing performance.

Conversely, **vertical partitioning** involves splitting the columns of a table into separate tables. This technique aims to reduce the column count in a table and boost the performance of queries that only access a limited number of columns.



Data partitioning

8. Database Replication: Enhancing Reliability in System Design

Database replication is a method employed to maintain multiple copies of the same database across various servers or locations.

The main objective of database replication is to enhance data availability, redundancy, and fault tolerance, ensuring the system remains operational even in the face of hardware failures or other issues.

In a replicated database configuration, one server serves as the primary (or master) database, while others act as replicas (or slaves).

This process involves synchronizing data between the primary database and replicas, ensuring all possess the same up-to-date information.

Database replication provides several advantages, including:

1. Improved Performance: By distributing read queries among multiple replicas, the load on the primary database can be reduced, leading to faster query response times.
2. High Availability: If the primary database experiences failure or downtime, replicas can continue to provide data, ensuring uninterrupted access to the application.

3. Enhanced Data Protection: Maintaining multiple copies of the database across different locations helps safeguard against data loss due to hardware failures or other disasters.
4. Load Balancing: Replicas can handle read queries, allowing for better load distribution and reducing overall stress on the primary database.

9. Distributed Messaging Systems: Scaling Communication in System Design

Distributed messaging systems provide a reliable, scalable, and fault-tolerant means for exchanging messages between numerous, possibly geographically dispersed applications, services, or components.

These systems facilitate communication by decoupling sender and receiver components, enabling them to develop and function independently.

Distributed messaging systems are especially valuable in large-scale or intricate systems, like those seen in microservices architectures or distributed computing environments.

Examples of these systems include Apache Kafka and RabbitMQ.

10. Microservices: Embracing Modular System Design

Microservices represent an architectural style wherein an application is organized as an assembly of small, loosely coupled, and autonomously deployable services.

Each microservice is accountable for a distinct aspect of functionality or domain within the application and communicates with other microservices via well-defined APIs.

This method deviates from the conventional monolithic architecture, where an application is constructed as a single, tightly-coupled unit.

The primary characteristics of microservices include:

1. Single Responsibility: Adhering to the Single Responsibility Principle, each microservice focuses on a specific function or domain, making the services more straightforward to comprehend, develop, and maintain.
2. Independence: Microservices can be independently developed, deployed, and scaled, offering increased flexibility and agility in the development process. Teams can work on various services simultaneously without impacting the entire system.

3. Decentralization: Typically, microservices are decentralized, with each service possessing its data and business logic. This approach fosters separation of concerns and empowers teams to make decisions and select technologies tailored to their unique requirements.
4. Communication: Microservices interact with each other using lightweight protocols, such as HTTP/REST, gRPC, or message queues. This fosters interoperability and facilitates the integration of new services or the replacement of existing ones.
5. Fault Tolerance: As microservices are independent, the failure of one service does not necessarily result in the collapse of the entire system, enhancing the application's overall resiliency.

Microservices

11. NoSQL Databases

NoSQL databases, or “Not Only SQL” databases, are non-relational databases designed to store, manage, and retrieve unstructured or semi-structured data.

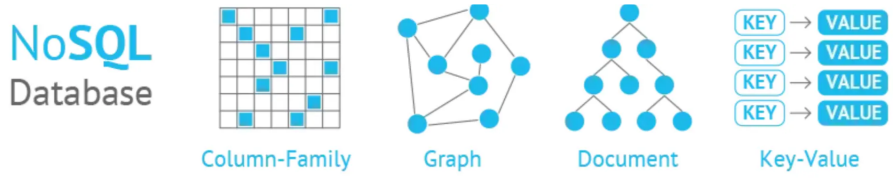
They offer an alternative to traditional relational databases, which rely on structured data and predefined schemas.

NoSQL databases have become popular due to their flexibility, scalability, and ability to handle large volumes of data, making them well-suited for modern applications, big data processing, and real-time analytics.

NoSQL databases can be categorized into four main types:

1. Document-Based: These databases store data in document-like structures, such as JSON or BSON. Each document is self-contained and can have its own unique structure, making them suitable for handling heterogeneous data. Examples of document-based NoSQL databases include MongoDB and Couchbase.
2. Key-Value: These databases store data as key-value pairs, where the key acts as a unique identifier, and the value holds the associated data. Key-value databases are highly efficient for simple read and write operations, and they can be easily partitioned and scaled horizontally. Examples of key-value NoSQL databases include Redis and Amazon DynamoDB.
3. Column-Family: These databases store data in column families, which are groups of related columns. They are designed to handle write-heavy workloads and are highly efficient for querying data with a known row and column keys. Examples of column-family NoSQL databases include Apache Cassandra and HBase.
4. Graph-Based: These databases are designed for storing and querying data that has complex relationships and interconnected structures, such as social networks or recommendation systems. Graph databases use nodes, edges, and properties to represent and store data,

making it easier to perform complex traversals and relationship-based queries. Examples of graph-based NoSQL databases include Neo4j and Amazon Neptune.



Types of NoSQL databases

12. Database Index

Database indexes are data structures that enhance the speed and efficiency of query operations within a database.

They function similarly to an index in a book, enabling the database management system (DBMS) to swiftly locate data associated with a specific value or group of values, without the need to search through every row in a table.

By offering a more direct route to the desired data, indexes can considerably decrease the time required to retrieve information from a database.

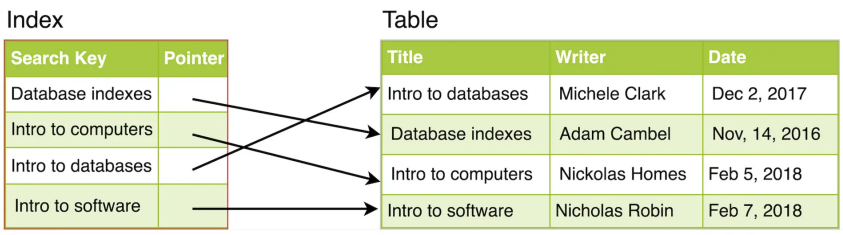
Indexes are typically constructed on one or more columns of a database table.

The B-tree index is the most prevalent type, organizing data in a hierarchical tree structure, which allows for rapid search, insertion, and deletion operations.

Other types of indexes, such as bitmap indexes and hash indexes, exist as well, each with their particular use cases and advantages.

Although indexes can significantly enhance query performance, they also involve certain trade-offs:

- **Storage Space:** Indexes require additional storage space since they generate and maintain separate data structures alongside the original table data.
- **Write Performance:** When data is inserted, updated, or deleted in a table, the corresponding indexes must also be updated, which may slow down write operations.



Database Index

13. Distributed File Systems: Managing Data Across System Design Architectures

Distributed file systems are storage systems designed to manage and grant access to files and directories across multiple servers, nodes, or machines, frequently distributed across a network.

They allow users and applications to access and modify files as though they were situated on a local file system, despite the fact that the actual files may be physically located on various remote

servers.

Distributed file systems are commonly employed in large-scale or distributed computing environments to offer fault tolerance, high availability, and enhanced performance.

14. Notification System: Real-Time Updates in System Design

These are used to send notifications or alerts to users, such as emails, push notifications, or text messages.

15. Full-Text Search: Efficient Data Retrieval in System Design

Full-Text search allows users to search for particular words or phrases within an application or website. Upon receiving a user query, the application or website delivers the most relevant results.

To accomplish this rapidly and effectively, full-text search utilizes an inverted index, a data structure that associates words or phrases with the documents where they are found.

Elasticsearch is an example of such systems.

16. Distributed Coordination Services: Orchestrating Complex System Design

Distributed coordination services are systems engineered to regulate and synchronize the actions of distributed applications, services, or nodes in a dependable, efficient, and fault-tolerant way.

They assist in maintaining consistency, addressing distributed synchronization, and overseeing the configuration and state of diverse components in a distributed setting.

Distributed coordination services are especially valuable in large-scale or intricate systems, like those encountered in microservices architectures, distributed computing environments, or clustered databases.

Apache ZooKeeper, etcd, and Consul are examples of such services.

In other words, these services (like ZooKeeper or etcd) act like referees or orchestrators, making sure all the parts of a distributed system play nicely together.

17. Heartbeat: Monitoring Availability in Distributed System Design

In a distributed environment, work/data is distributed among servers. To efficiently route requests in such a setup, servers need to know what other servers are part of the system.

Furthermore, servers should know if other servers are alive and working.

In a decentralized system, whenever a request arrives at a server, the server should have enough information to decide which server is responsible for handling that request.

This makes the timely detection of server failure an important task, which also enables the system to take corrective actions and move the data/work to another healthy server and stop the environment from further deterioration.

To solve this, each server periodically sends a heartbeat message to a central monitoring server or other servers in the system to show that it is still alive and functioning.

Heartbeating is one of the mechanisms for detecting failures in a distributed system.

If there is a central server, all servers periodically send a heartbeat message to it.

If there is no central server, all servers randomly choose a set of servers and send them a heartbeat message every few seconds. This way, if no heartbeat message is received from a server for a while, the system can suspect that the server might have crashed.

If there is no heartbeat within a configured timeout period, the system can conclude that the server is not alive anymore and stop sending requests to it and start working on its replacement.

18. Checksum: Ensuring Data Integrity in System Design

In a distributed system, while moving data between components, it is possible that the data fetched from a node may arrive corrupted. This corruption can occur because of faults in a storage device, network, software, etc. How can a distributed system ensure data integrity, so that the client receives an error instead of corrupt data?

To solve this, we can calculate a checksum and store it with data.

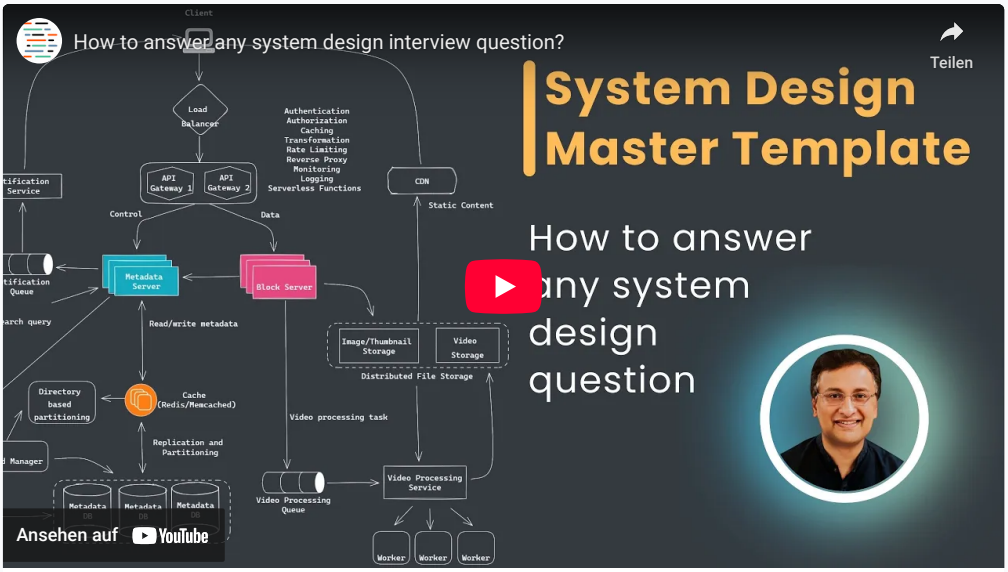
To calculate a checksum, a cryptographic hash function like MD5, SHA-1, SHA-256, or SHA-512 is used.

The hash function takes the input data and produces a string (containing letters and numbers) of fixed length; this string is called the checksum.

When a system is storing some data, it computes a checksum of the data and stores the checksum with the data.

When a client retrieves data, it verifies that the data it received from the server matches the checksum stored. If not, then the client can opt to retrieve that data from another replica.

These fundamental system design concepts serve as basic principles for scalable system architecture.



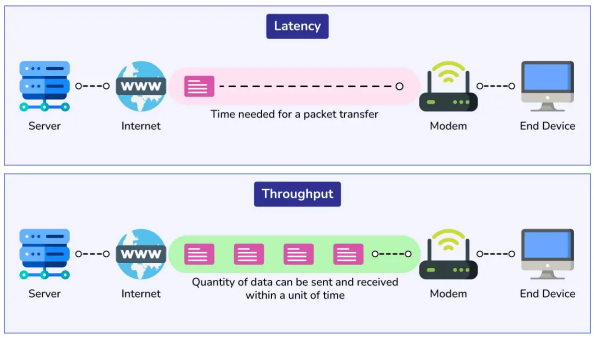
How to answer any system design interview question?

19. Latency vs Throughput

Latency is the time it takes to complete a single request—from the moment it's sent to the moment a response is received. Think of it like the time it takes for one person to get a coffee at a café.

Throughput is the number of requests a system can handle in a given amount of time—like how many coffees that café can serve in a minute.

A system can have low latency (fast response) but low throughput (serves few users), or high throughput but higher latency.



Latency vs. throughput

In interviews, you'll often be asked to design for **both**—fast responses for users (low latency) and the ability to handle scale (high throughput).

20. Availability vs Reliability

Availability means your system is up and running when users try to access it. If it's 99.99% available, it's down for only a few minutes a year.

Reliability means the system works correctly over time without failure—consistently delivering the expected result.

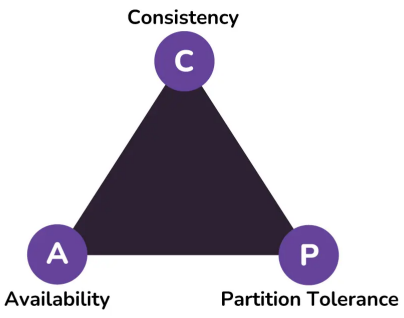
A service can be available but unreliable (it responds but gives wrong results), or reliable but unavailable (always accurate but frequently down).

You aim for both in critical systems like payments or healthcare, where downtime and errors aren't acceptable.

21. CAP Theorem

The **CAP Theorem** says that in a distributed system, you can only guarantee **two out of three**:

- **Consistency**: Every user sees the same data at the same time
- **Availability**: The system responds to every request (even if data is slightly outdated)
- **Partition Tolerance**: The system keeps running even if parts of the network fail



CAP Theorem

In practice, network issues (partition tolerance) are inevitable, so systems often choose between **consistency** and **availability** depending on the use case. For example, banking favors consistency, while social media may lean toward availability.

22. Consistency Patterns

These define how up-to-date and uniform data is across distributed systems.

- **Strong consistency** ensures every read reflects the latest write.
- **Eventual consistency** allows data to be out of sync temporarily but consistent over time.
- **Causal and session consistency** fall somewhere in between, depending on user actions or session context.

Different applications choose different patterns—for example, shopping carts might use eventual consistency, while financial systems must use strong consistency.

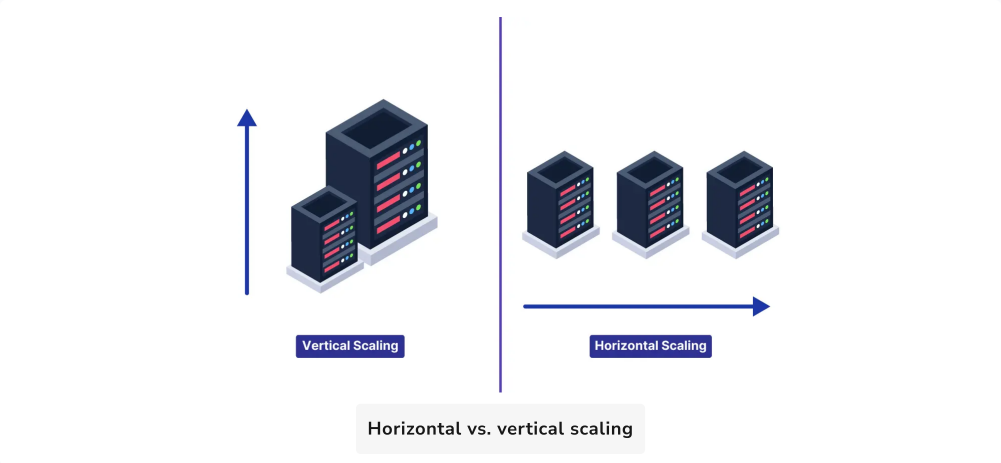
23. Scalability

Scalability is a system's ability to grow and handle more load—whether it's more users, more data, or more requests per second.

There are two types:

- **Vertical scaling** means adding more power (CPU, RAM) to one server.

- **Horizontal scaling** means adding more servers to spread the load.



Horizontal vs. vertical scaling

A well-designed system should scale without needing a complete rewrite—think of how Instagram grew from thousands to millions of users.

24. Long Polling vs WebSockets

Long polling is a technique where the client repeatedly asks the server for updates, and the server holds the connection until new data is available.

It works but is inefficient, especially when updates are frequent—it keeps making and closing HTTP connections.

WebSockets allow a **persistent, two-way connection** between client and server, perfect for real-time updates like chat apps or stock tickers.

WebSockets are preferred when you need live data, but long polling is simpler and still useful for less frequent updates.

25. Rate Limiting

Rate limiting controls how many requests a user or client can make in a certain time window (like "100 requests per minute").

It protects your system from abuse, overuse, or accidental traffic spikes.

Without it, a single user—or bot—could overload your servers or unfairly consume resources. Common techniques include token buckets, leaky buckets, or fixed windows.

Rate limiting is especially important in APIs, login systems, and public-facing services.

Conclusion: Putting System Design Concepts into Practice

Mastering these 25 system design concepts significantly boosts your ability to build scalable, robust applications and ace system design interviews.

Whether you're tackling DNS resolution, balancing traffic via load balancers, or enhancing performance with caching, each concept lays the groundwork for a resilient, high-performing system.

By applying the system design master template and diving into detailed discussions about critical components, you'll be equipped to handle complex questions during your interviews.

Keep exploring topics such as data partitioning, distributed messaging, and advanced system design patterns to stay at the forefront of modern software architecture.

I have also developed a system design master template that should guide you in answering any system design interview question.

For further reading, here is a list of **common system design interview questions**:

- Designing a File-sharing Service Like Google Drive or Dropbox
- Designing a Video Streaming Platform
- Designing a URL Shortening Service
- Designing a Web Crawler
- Designing Uber
- Designing Facebook Messenger
- Designing Twitter Search
- Designing Instagram

Best of luck on your system design interview journey—may your architectures be scalable, reliable, and fully prepped for success!

FAQs – System Design Interview Fundamentals

Q1: What are the most important concepts to know for system design interviews?

Key concepts include load balancing, caching, CAP theorem, consistency models, data partitioning, replication, queues, proxies, and scalability strategies. Mastering these fundamentals helps you structure reliable, scalable systems.

Q2: What is the CAP theorem in system design?

The CAP theorem states that in a distributed system, you can only have two out of three: consistency, availability, and partition tolerance. You must choose based on your system's priorities.

Q3: What is the difference between latency and throughput?

Latency is the time it takes to complete a single request, while throughput is the number of requests a system can handle per unit of time.

Q4: Why is caching important in system design?

Caching stores frequently accessed data closer to the user or app, reducing load on the backend and speeding up response times.

Q5: What is load balancing in system design?

Load balancing distributes incoming traffic across multiple servers to ensure no single server is overwhelmed, improving performance and fault tolerance.

Q6: Do I need to know these system design fundamentals for interviews at FAANG companies?

Yes. FAANG and top tech companies expect candidates to understand these fundamentals to demonstrate architectural thinking and build scalable solutions during interviews.

Q7: How should you use these system design concepts in an interview setting?

When facing a **system design interview**, treat concepts like caching, scalability, the **CAP theorem**, fault tolerance, and sharding as tools in your toolbox – to be applied thoughtfully rather than just recited.

Interviewers won't usually ask you to define these concepts in isolation; instead, they expect you to **bring them up when relevant** to solving the design problem at hand.

Here's how to effectively use these system design concepts during your interview:

- **Identify Relevant Concepts for the Problem:** Start by understanding the question's requirements (e.g. high traffic, big data, low latency, high reliability). Then **choose the appropriate concepts** that address those needs. For example, if the system must handle millions of users, discuss **scalability** strategies (like sharding or load balancing); if low latency is crucial, mention **caching**; if data consistency is a concern, invoke the **CAP theorem** to reason about consistency vs. availability trade-offs. Use each concept only where it **logically fits**, rather than force-fitting every buzzword.
- **Explain and Justify Each Concept's Use:** When you introduce a concept, **define it briefly in simple terms** and immediately tie it to your design's needs. This shows the interviewer you truly understand it. For instance, you might say, *"To improve response times, I would add **caching**, which stores frequent results in memory to reduce database load."* Always follow up with **why** that concept helps in this context. If you decide on a NoSQL database for scalability, explain that choice (e.g. flexible schema, easier horizontal scaling) in comparison to alternatives. Every design decision should have a clear rationale, as this makes your answer **strong and logical**.
- **Discuss Trade-offs and Downsides:** Demonstrating an ability to **reason about trade-offs** is critical in system design interviews. Whenever you propose a concept, briefly discuss its pros and cons. For example, if you introduce caching, note that it improves speed but **can serve stale data** if not updated, so you might need an eviction policy or cache invalidation strategy. If you suggest sharding the database for scalability, acknowledge the added **complexity in rebalancing or joining data across shards**, and perhaps mention a technique like consistent hashing to mitigate that. By bringing up potential challenges (e.g. cache consistency, shard rebalancing, replication lag) and how to handle them, you show you understand the **practical implications** of these concepts. This ability to balance trade-offs is exactly what interviewers look for in strong candidates.
- **Communicate Clearly and Structure Your Answer:** Using system design concepts effectively isn't just *what* you say but *how* you say it. Organize your answer in a clear framework (e.g. start with requirements, then high-level design, then dive into specifics) so that when you introduce a concept, it fits naturally into your design flow. Make sure to **speak clearly and methodically**, almost as if you are teaching the concept in context. Don't just name-drop "CAP theorem" or "fault tolerance" – explain what those mean for the design (*e.g. "Because we need high availability, I'd replicate services across regions for fault tolerance, so the system stays up even if one data center fails"*). Engage your interviewer in a two-way discussion: you can even ask if they have concerns about a particular approach. Remember that **communication skills are being assessed too**. Top candidates *"ask questions, consider trade-offs, and justify their choices"* as they walk through the design. By clearly articulating how each concept solves a problem (and at what cost), you demonstrate both your technical knowledge and your reasoning ability.

In summary, use these **system design concepts** as building blocks to construct a robust solution tailored to the question.

Show that you understand each concept's purpose and **apply it judiciously** to meet the system's requirements.

Like Save Share

System Design Fundamentals | System Design Interview | Data Partitioning | Load Balancer | Reliability

What our users say

ABHISHEK GUPTA

My offer from the top tech company would not have been possible without this course. Many thanks!!

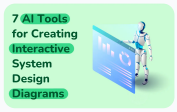
Eric

I've completed my first pass of "grokking the System Design Interview" and I can say this was an excellent use of money and time. I've grown as a developer and now know the secrets of how to build these really giant internet systems.

AHMET HANIF

Whoever put this together, you folks are life savers. Thank you :)

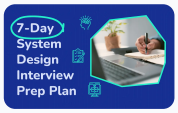
More From Designgurus



Top 7 Tools for Creating System Design Diagrams

Which tools are best for creating interactive system design diagrams?

Arslan Ahmad — Apr 30th, 2024



Your 7-Day System Design Interview Prep Plan

Need to prep fast for your system design interview? This 7-day crash course breaks down what to study each day—fundamentals, real-world examples, mock questions, and expert tips. Includes a free checklist...

Arslan Ahmad — Jul 3rd, 2025

Last-Minute System Design Prep: Key Focus Areas

Need to prepare for a system design interview fast? This last-minute system design prep guide covers the key concepts, must-know technologies, and high-impact strategies to help you focus on what truly matte...

Arslan Ahmad — Jan 30th, 2025

Large-Scale System Design Questions: How to Design Systems at Scale

Facing questions about large-scale system design? Learn how to architect systems for millions of users with high availability, reliability, and performance. This guide breaks down key concepts to help you desig...

Arslan Ahmad — Jan 14th, 2025

System Design Interview Guide – 7 Steps to Ace It

Confused by system design interviews? Follow this 7-step framework to structure your answers like a pro. Perfect for FAANG prep and 2025 interviews.

Arslan Ahmad — Feb 22nd, 2023

Top 10 System Design Challenges Every Developer Must Master for 2025

Discover the 10 system design challenges to handle in 2025

Arslan Ahmad — Dec 23rd, 2024

How to Design a URL Shortener Service (System Design Interview Guide)

Learn how to design a URL shortening service (like TinyURL or Bit.ly) in a system design interview. We cover the full design: requirements (e.g. generating short links, redirection, custom aliases), system...

Arslan Ahmad — Feb 23rd, 2023

Why Practicing System Design Is Crucial for Software Engineers

Find out why practicing system design is crucial for software engineers and how they can build scalable systems, sharpen problem-solving, and ace interviews to advance their careers.

Arslan Ahmad — May 9th, 2025

How To Clear System Design Interview: A Quick Guide

What You Must Know To Clear a System Design Interview

Arslan Ahmad — Apr 22nd, 2024

Netflix System Design Interview Questions: An In-Depth Guide

Preparing for Netflix's system design interview? This guide covers top Netflix design questions—like building a video streaming platform and CDN architecture—and shows how to approach them the Netflix...

Arslan Ahmad — Jul 13th, 2023