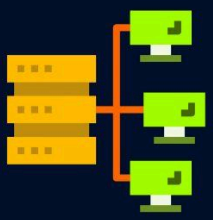
FREE

SYSTEM DESIGN CRASH COURSE



Load Balancer

Data Partitioning

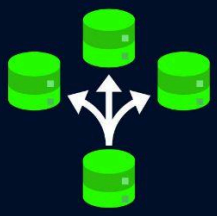
Sharding

CAP Theorem

CDN

Message Queue

API

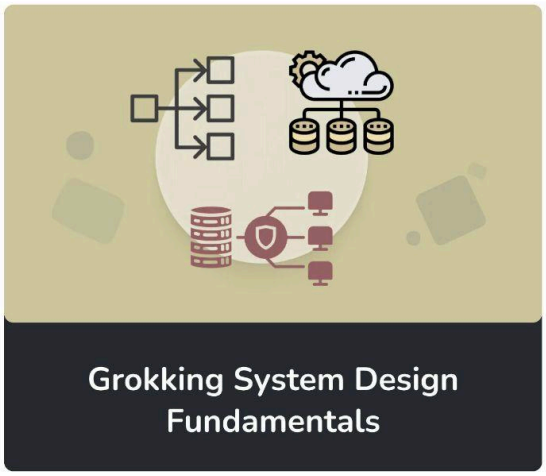


Scalability

Caching

DesignGurus.io

Check our top-selling **System Design Courses** available on DesignGurus.io.



Grokking System Design Fundamentals

Grokking the System Design Interview

Grokking the Advanced System Design Interview

Grokking Microservices Design Patterns

Grokking the Object Oriented Design Interview

Grokking Design Patterns for Engineers and Managers

System Design Crash Course

System design interviews can feel overwhelming for beginners, but they don't have to be.

This crash course breaks down the fundamental concepts you need to know in a clear and concise manner.

Whether you're preparing for a tech interview or just learning the basics, these bite-sized explanations will help you grasp key system design ideas quickly.

Let's demystify system design and get you confident with the essentials!

The System Design Master Template

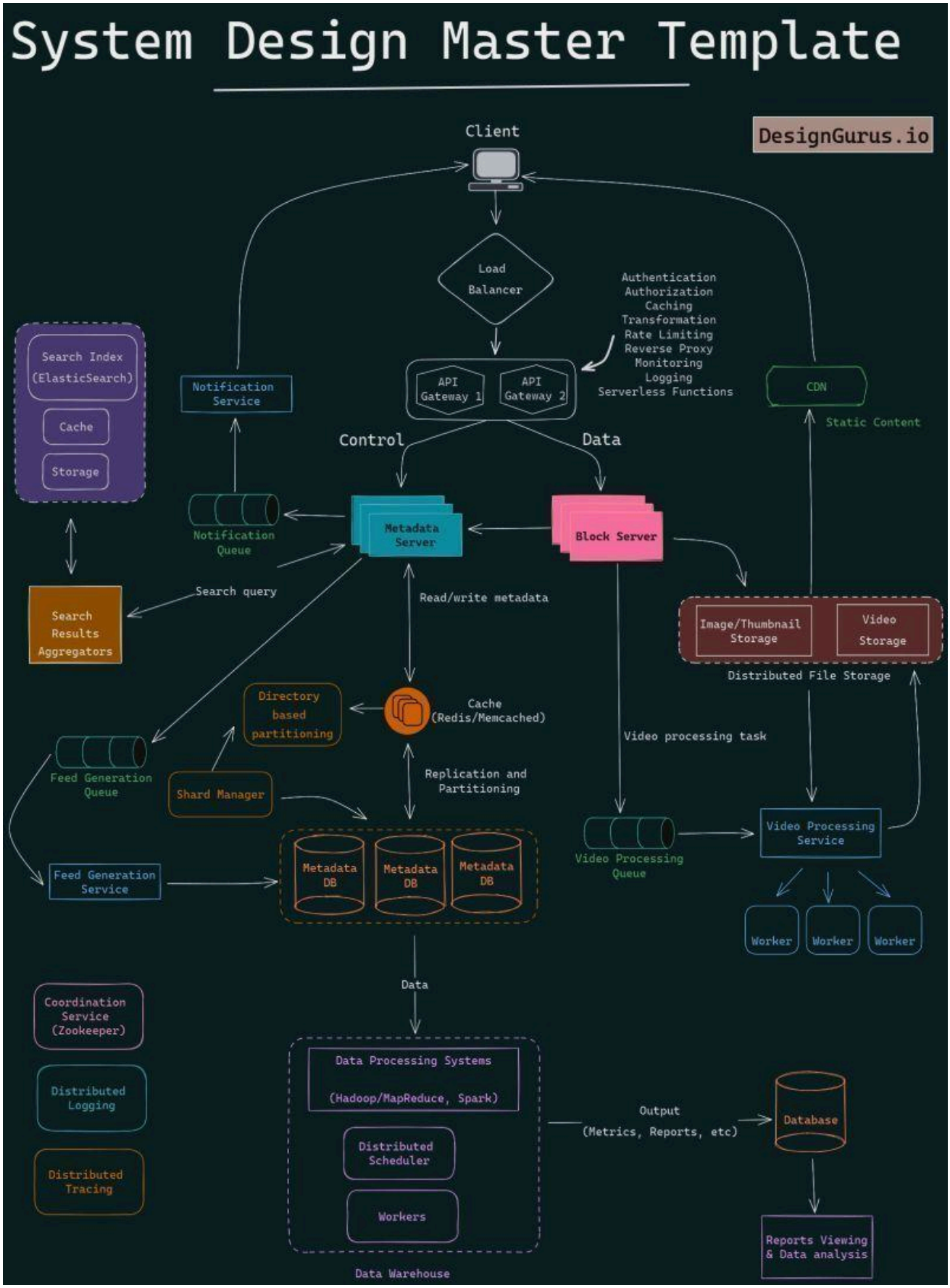
Before diving into individual concepts, it's helpful to have a roadmap.

DesignGurus.io offers a **System Design Master Template** – a blueprint of standard components and steps to consider when tackling any system design problem.

The idea is to ensure you cover all critical areas of a system, from how users find your service (DNS) to how data is stored and served.

This template includes key building blocks, such as load balancers, caches, databases, data partitioning, message queues, and microservice architecture, among others.

Here is the **System Design Master Template** we will be following:


System Design Master Template
DesignGurus.io
Client
Load Balancer
Authentication
Authorization
Caching
Transformation
Rate Limiting
Reverse Proxy
Monitoring
Logging
Serverless Functions
API Gateway 1
API Gateway 2
CDN
Static Content
Search Index (ElasticSearch)
Cache
Storage
Notification Service
Control
Data
Notification Queue
Metadata Server
Block Server
Search query
Search Results Aggregators
Read/write metadata
Image/Thumbnail Storage
Video Storage
Distributed File Storage
Directory based partitioning
Cache (Redis/Memcached)
Video processing task
Feed Generation Queue
Shard Manager
Replication and Partitioning
Video Processing Service
Video Processing Queue
Feed Generation Service
Metadata DB
Metadata DB
Metadata DB
Worker
Worker
Worker
Data
Coordination Service (Zookeeper)
Distributed Logging
Distributed Tracing
Data Processing Systems
(Hadoop/MapReduce, Spark)
Distributed Scheduler
Workers
Output
(Metrics, Reports, etc)
Database
Reports Viewing & Data analysis
Data Warehouse

By following a master template, you can systematically address requirements, outline a high-level architecture, and make sure you discuss all important aspects during an interview. It's like a checklist that reminds you to think about things like scalability, reliability, and performance for every design.

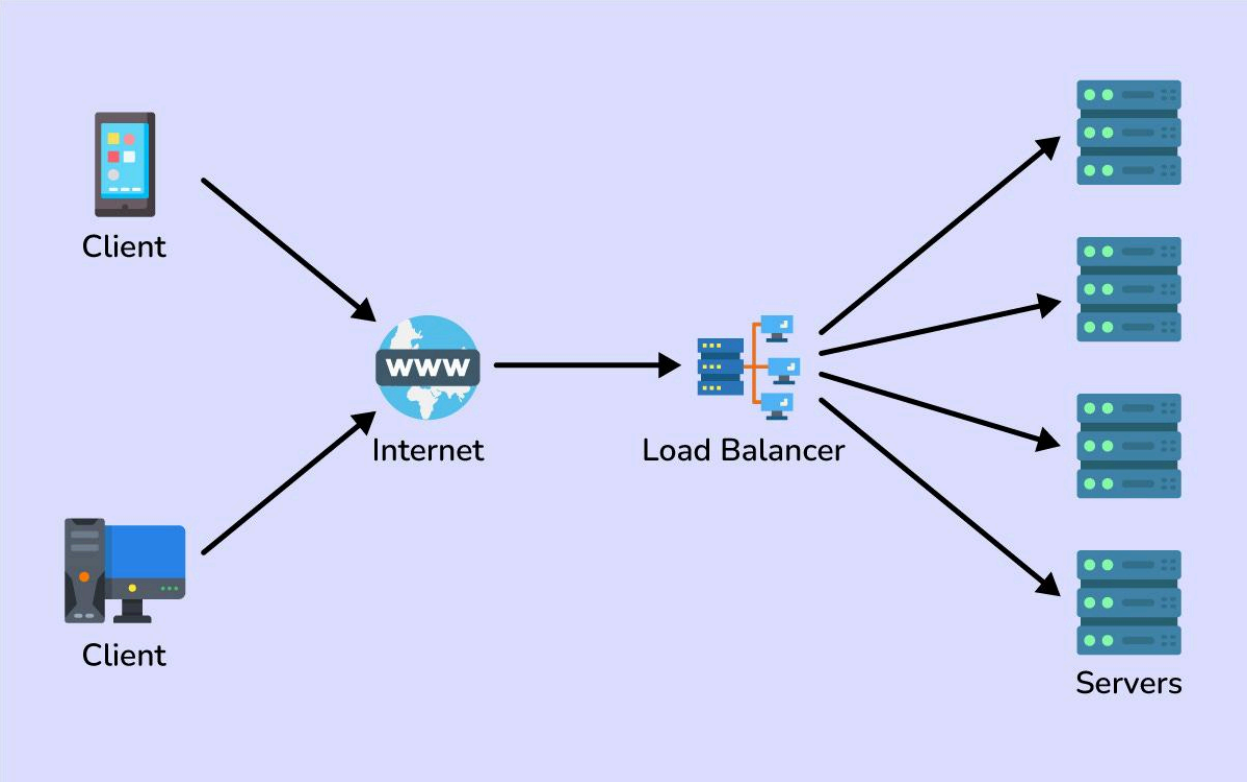
With that in mind, let's jump into the core concepts one by one.

1. Load Balancer

A **load balancer** is a tool (either hardware or software) that distributes incoming traffic across multiple servers.

Think of it as a smart traffic cop for your servers: it makes sure no single server gets overwhelmed by requests.

By spreading out requests, load balancers help improve a system's performance and prevent downtime.



Load balancing

Load balancers can use different algorithms to decide where each request goes (common ones include **Round Robin**, which cycles through servers one by one, and **Least Connection**s, which picks the server with the fewest active users).

They also often perform health checks on servers, so if one server goes down, the load balancer stops sending traffic to it.

This way, your application stays **highly available** and responsive even under heavy load.

Further Reading (Load Balancer):

- Step-by-Step Approach for Load Balancing in System Design Interviews
- Load Balancer vs Reverse Proxy vs API Gateway
- What is Load Balancing and How Does It Improve System Reliability and Performance?
- How to Design a Load Balancer from Scratch
- How to Understand Load Balancing for System Design Interviews
- What are Different Load Balancer Algorithms?

2. Data Partitioning

Data partitioning means splitting a large dataset into smaller parts (partitions) so each part can be managed separately. It's like dividing a big library into sections by topic – each section is easier to handle than the entire library at once.

Partitioning improves performance and scalability because queries can target just one partition instead of the whole dataset.

There are different ways to partition data.

Horizontal partitioning (sharding) splits rows of data across multiple databases or tables (each one holds a subset of records), while **vertical partitioning** splits by columns (grouping certain attributes together).

The goal is to distribute load and storage, so no single database becomes a bottleneck.

Proper partitioning can make your system more efficient, but it adds complexity in keeping track of where data lives.

Further Reading (Data Partitioning):

- Data Replication Strategies in System Design

- Horizontal vs Vertical Partitioning: Key Differences
- Understanding Data Partitioning and Sharding for Interviews
- Implementing Data Partitioning in Microservices
- Sharding vs Partitioning: What's the Difference?

3. Sharding

Sharding is a specific type of data partitioning where you break up a database into smaller pieces called *shards*, each on a different server.

Each shard holds a portion of the data (for example, users A-M on shard 1, N-Z on shard 2).

Sharding is used when a single database can't handle the scale – by spreading data across multiple machines, you can handle more traffic and larger datasets.

A critical decision in sharding is choosing a **shard key**, which is the attribute used to decide which shard a particular piece of data goes to.

For instance, you might use user ID as a shard key to determine which database holds that user's data.

Good sharding design ensures data is evenly distributed (to avoid hot spots where one shard gets most of the load).

Keep in mind that sharding increases complexity: queries that need data from multiple shards are harder, and you'll need to manage things like rebalancing data if one shard grows too large.

Further Reading (Sharding):

- What is Database Sharding?
- Sharding vs Partitioning: Key Differences
- How to Choose a Sharding Key
- Data Partitioning and Sharding Basics
- How to Handle Database Sharding

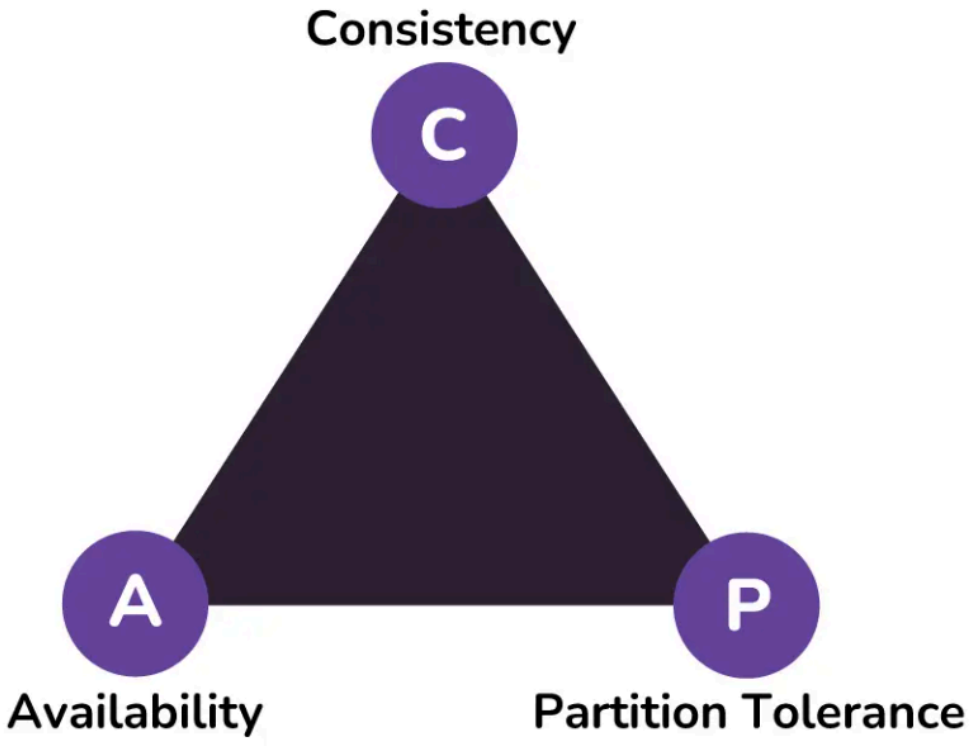
4. CAP Theorem

The **CAP Theorem** is a fundamental principle in distributed systems that describes a trade-off between three qualities: **Consistency**, **Availability**, and **Partition Tolerance**.

It states that in the presence of a network partition (i.e., when some nodes can't communicate with others), a distributed system can only guarantee either consistency or availability – but not both at the same time.

- **Consistency (C):** Every read receives the latest write or an error. In simple terms, all nodes see the same data at the same time.
- **Availability (A):** Every request receives a (non-error) response, but without a guarantee that it contains the latest write. The system remains operational and responsive.
- **Partition Tolerance (P):** The system continues to work despite network splits or communication breakdowns between nodes.

According to CAP, you must make a choice (especially during a network fault): either prioritize consistency (might return an error or block updates to keep data in sync) or prioritize availability (always return some response, even if data might not be up-to-date).



For example, many NoSQL databases choose availability over strong consistency (they're "AP" systems), whereas some relational databases might choose consistency over availability ("CP" systems) during a partition.

Understanding CAP helps in deciding which database or caching strategy to use based on the needs of your system.

Further Reading (CAP Theorem):

- What is the CAP Theorem?
- CAP Theorem and Trade-offs for Distributed Databases
- Understanding CAP Theorem for System Design Interviews
- Consistency vs Availability (CAP)
- CAP vs PACELC (Beyond CAP)

5. CDN (Content Delivery Network)

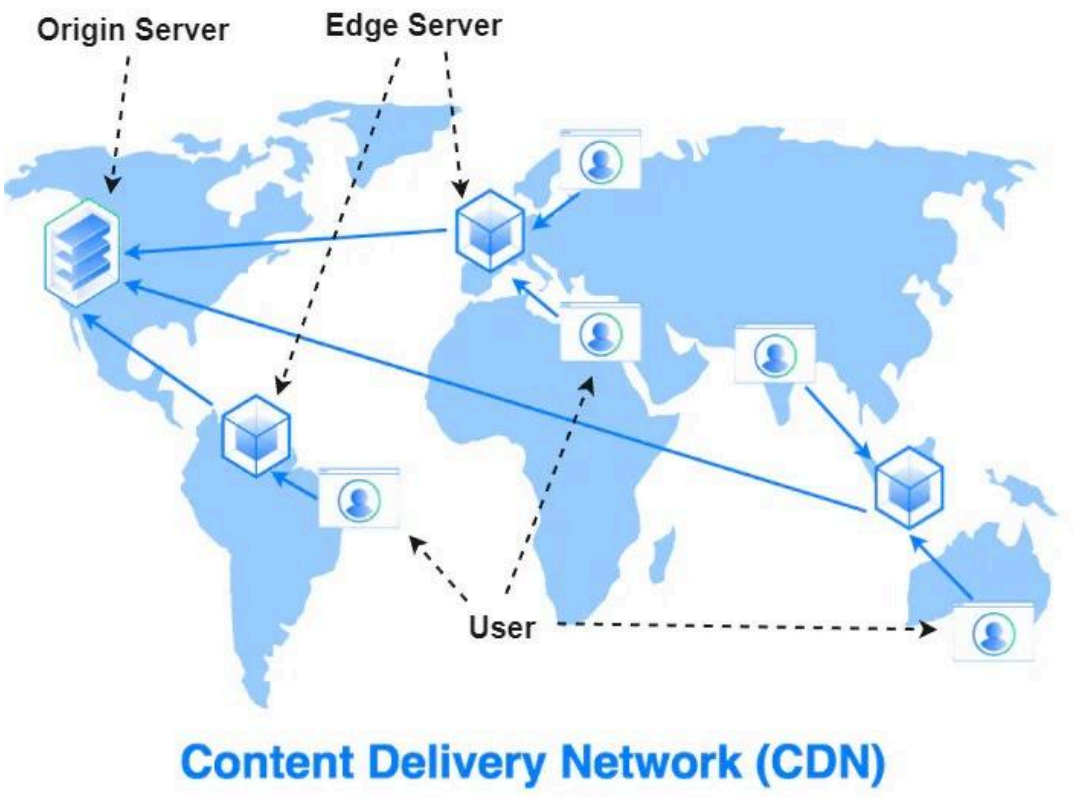
A **Content Delivery Network (CDN)** is a network of servers distributed around the world that delivers content (like images, videos, and web pages) to users from the nearest server location.

The main idea is to reduce latency – the delay users experience when loading content.

By serving content from a server geographically closer to the user, load times are faster.

Here's how it works: if you're in London and request a photo from a website that normally hosts its data in New York, a CDN will have a cached copy of that photo on a London server.

You get the photo quicker because it travels a shorter distance.



CDN

CDNs also take load off the origin servers because popular content is cached at the edges.

This improves reliability and can protect against traffic spikes (even some DDoS attacks) since the content is distributed.

In short, a CDN makes your content load faster and your service more robust for users all over the world.

Further Reading (CDN):

- What is a CDN and How Does It Speed Up Content Delivery?
- Understanding CDN (Content Delivery Network) in System Design
- CDN System Design Basics
- What is a Content Delivery Network (CDN)?
- Explain CDN “Stale-While-Revalidate”

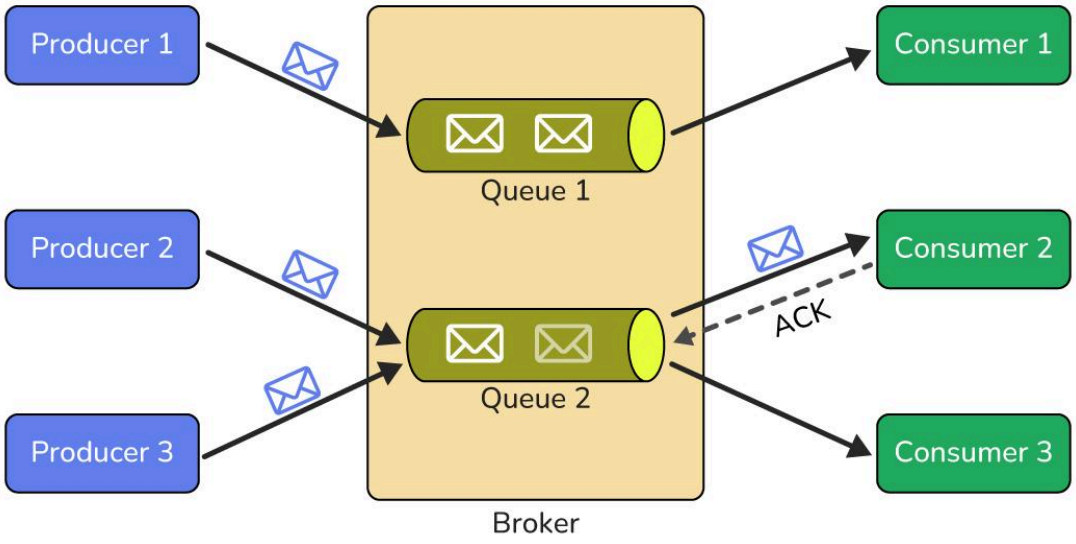
6. Message Queue

A **message queue** is a communication mechanism that uses a queue for sending and receiving messages between services or components.

Imagine a line (queue) at a ticket counter: producers (senders) post messages into the queue, and consumers (receivers) process them in order, one by one.

This setup decouples the producer from the consumer – the producer can continue working after placing the message, and the consumer will handle it whenever it's ready.

Message queues enable **asynchronous processing**, which is crucial for building scalable systems.



For example, when a user uploads a photo, your app can quickly put a "process this photo" message in a queue and immediately respond to the user, while a background worker will eventually resize and store the photo.

Queues help smooth out traffic spikes and ensure no data is lost if a receiver is temporarily busy or down (the message stays in the queue until processed).

Overall, they improve system resilience and flexibility by buffering work and letting components work independently.

Further Reading (Message Queue):

- What is a Message Queue and Why Are Queues Used in Scalable System Design?
- Understanding Message Queues for System Design Interviews
- How to Design a Message Queue for System Design Interviews
- Differences Between Message Brokers and Message Queues

7. Message Broker

A **message broker** is like the postal service for your system – it routes and manages messages between different parts of your application.

While a message queue is the actual data structure that holds messages, the broker is the software that organizes the queues, handles routing logic, and sometimes offers additional features like message persistence, pub/sub (publish-subscribe) patterns, and delivery guarantees.

Popular message brokers include **RabbitMQ**, **Apache Kafka**, and **ActiveMQ**.

A broker can have multiple queues (and topics) and knows how to deliver each message to the right consumer(s).

For instance, in an **event-driven architecture**, different services communicate through a broker: one service publishes an event (message) to the broker, and one or more other services subscribe to receive that event.

The broker ensures the message gets to its intended destinations.

By using a message broker, systems become more **loosely coupled** – services don't call each other directly, they just send messages through the broker.

This improves scalability and fault tolerance, because each service can operate and be scaled independently.

Further Reading (Message Broker):

- Message Brokers vs Message Queues (Differences)

- What is a Message Broker (e.g. Kafka or RabbitMQ) and How Is It Used in Event-Driven Design?
- Types of Message Brokers
- Popular Message Broker Technologies
- RabbitMQ vs Kafka vs ActiveMQ (System Design)

8. API

An **API (Application Programming Interface)** is a set of rules and interfaces that allows different software programs to communicate with each other.

In simpler terms, an API is like a menu at a restaurant: it tells you what you can order (the operations you can perform) and returns the result without exposing the kitchen (the internal implementation).

For web services, APIs are often exposed as endpoints (URLs) that clients can send requests to (usually using HTTP).

APIs enable integration between different systems. For example, a mobile app uses an API to fetch data from a server.

There are different types of APIs – **RESTful APIs** (the most common style for web services, using HTTP verbs like GET, POST), **GraphQL** (a query language for APIs), and others.

There are also categories like **internal vs external APIs** (internal for within an organization, external for third-party developers).

In system design, understanding APIs is crucial because they define how clients interact with your service and how microservices talk to each other.

Good API design means clear documentation, consistent structure, and proper versioning so that updates don't break things for users.

Further Reading (API):

- What is an API (Application Programming Interface)?
- Mastering the API Interview: Common Questions and Expert Answers
- What is a RESTful API?

- What are the 4 Types of API?
- How Do I Prepare for API (Interview)?
- Understanding APIs for Software Engineering Interviews

9. Replication

Replication means keeping copies of data on multiple machines. In databases, replication is used to increase reliability and improve read performance.

For example, you might have one primary database that accepts all writes, and multiple secondary replicas that copy the data from the primary.

If the primary goes down, a replica can take over (improving reliability).

And for reads (like fetching data), your application can query the replicas, which spreads the load and makes reads faster for users in different locations.

There are a couple of common replication strategies:

- **Primary-Replica (Master-Slave):** One primary node handles all writes, and one or more replica nodes receive copies of the data. Reads can be done from replicas.
- **Multi-Master (Peer-to-Peer):** Multiple nodes can handle writes and sync with each other. This is more complex but can handle writes in multiple locations.

Replication is key for high **availability** and **fault tolerance** – even if one database instance fails, others have the data. It also helps in scaling read-heavy workloads.

However, replication can introduce consistency challenges (e.g., there might be a slight delay before a replica has the latest data). It's important to handle conflicts (if using multi-master) and decide on synchronous vs asynchronous replication.

Synchronous replication means a write is confirmed only when all replicas have saved it (strong consistency, but slower), while asynchronous means the primary confirms immediately and replicas update afterward (faster, but risk of some data loss if a failure occurs right after a write).

Further Reading (Replication):

- Data Replication Strategies in System Design
- Database Replication (Overview)

- Primary-Replica vs Peer-to-Peer Replication
- Redundancy vs Replication (Differences)
- Handling Data Replication in Microservices Architecture
- What is Database Replication (for Reliability and Read Performance)?

10. Microservices Architecture

Microservices architecture is a design approach where an application is broken into many small, independent services that work together.

Each microservice focuses on a specific feature or domain (for example, user service, payment service, notification service) and has its own logic and database. These services communicate with each other through APIs or messaging systems, but they are developed, deployed, and scaled independently.

This is different from a **monolithic architecture**, where the entire application is one big unit.

Microservices offer a lot of benefits: each service can be developed and deployed independently (so teams can work in parallel), you can scale services individually based on their load, and if one service goes down, it might not bring down the whole system (improving fault isolation).

However, microservices also add complexity: there's overhead in managing many services, network calls between them, and challenges in monitoring and debugging a distributed system.

For beginners, it's important to understand the trade-offs.

Microservices shine for large, complex applications with many teams, while simpler or smaller apps might start as a monolith and only break into microservices when needed.

Further Reading (Microservices Architecture):

- What are Microservices? (Explained)
- What are Microservices (Architecture)?
- Microservices vs Monolithic Architecture
- Monolithic vs SOA vs Microservice Architecture
- Microservices vs “Normal” Services (What's the Difference?)

11. SQL vs NoSQL

When choosing a database, a common design decision is **SQL vs NoSQL**. **SQL databases** are relational databases (like MySQL, PostgreSQL) that use structured tables, schemas, and the SQL query language.

They are great for structured data, complex queries (like JOINs across multiple tables), and ensuring data consistency (they typically support ACID transactions, meaning operations are Atomic, Consistent, Isolated, Durable).

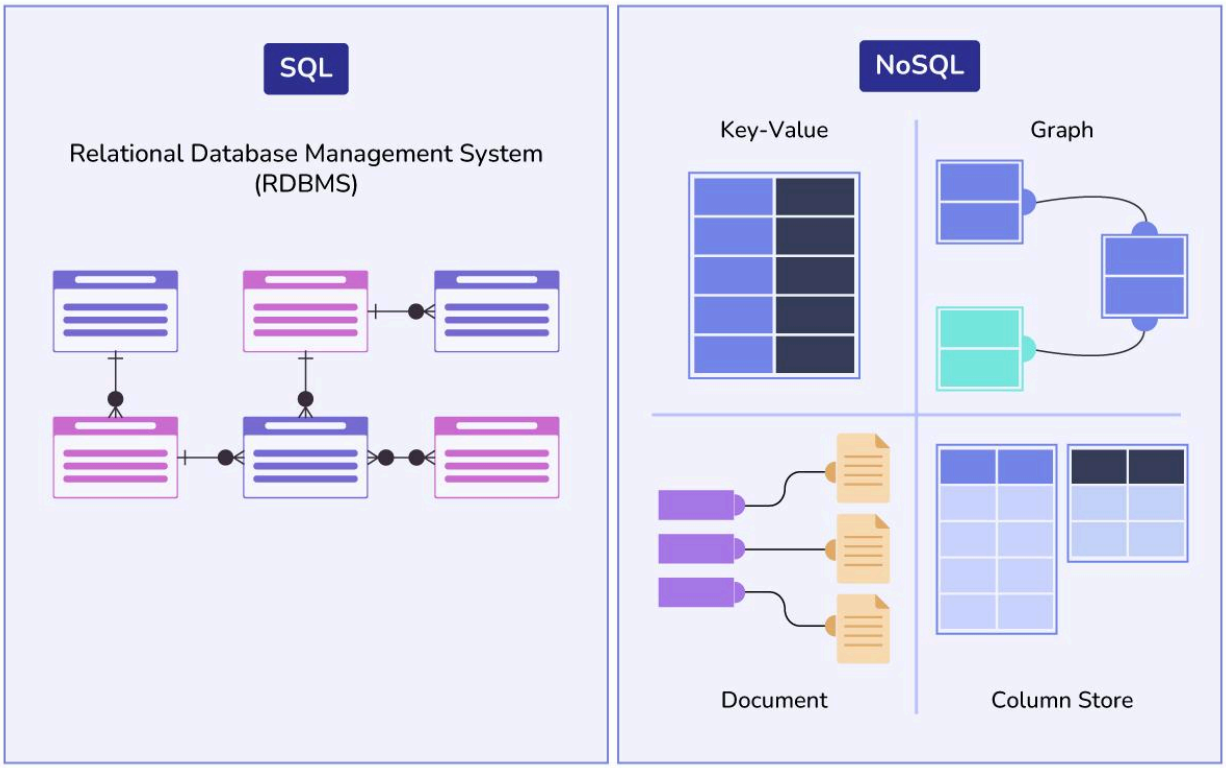
SQL databases are a solid choice when your data is highly relational or you need strict consistency and integrity.

NoSQL databases are "non-relational" databases, which come in various types: document stores (like MongoDB), key-value stores (like Redis), wide-column stores (like Cassandra), graph databases (like Neo4j), etc. NoSQL systems are designed for flexibility and scalability.

They often handle huge volumes of data and high traffic by scaling horizontally (adding more servers).

NoSQL might be schemaless or allow dynamic schemas, which means you can store data without a fixed structure – great for evolving requirements or big JSON-like data. They often prioritize availability and partition tolerance over immediate consistency (many are eventually consistent).

In practice, NoSQL is useful for big data, real-time analytics, content management, or whenever the relational model is too limiting or slow.



The key differences: SQL is structured and consistent but can be harder to scale horizontally; NoSQL is flexible and scalable but might sacrifice some consistency or query ability.

Many modern systems use a mix of both based on needs (this is called **polyglot persistence** – using different storage technologies for different parts of the application).

Further Reading (SQL vs NoSQL):

- SQL vs NoSQL: Key Differences
- SQL vs NoSQL: When to Use Each for System Design
- NoSQL Database Basics
- Normalization vs Denormalization
- Trade-offs Between NoSQL and SQL (in Interviews)
- Understanding NoSQL Databases for System Design Interviews

12. Scalability

Scalability is the ability of a system to handle increased load (more users, more data, more requests) by adding resources rather than having to redesign the system.

If your app goes viral and user traffic doubles, a scalable design can accommodate that growth smoothly.

There are two main ways to scale:

- **Vertical Scaling:** Add more power to a single server (e.g., upgrade CPU, RAM). It's like making one machine stronger. This is simpler but has limits (you can only get so big).
- **Horizontal Scaling:** Add more servers to share the load. It's like handling a growing crowd by opening more counters at a bank rather than building one super counter. Horizontal scaling is key for very large systems – using load balancers, distributing data (through partitioning or sharding), etc., to spread out work.

Horizontal vs Vertical Scaling

Designing for scalability often means avoiding single points of failure, enabling statelessness for easy replication of servers, and using distributed systems techniques.

Keep in mind that scaling can introduce complexity – more servers mean more coordination.

A good system design addresses how to grow without sacrificing performance or reliability.

Scalability isn't just about traffic; it can also refer to scaling the development process (for example, microservices help with scaling teams and codebases).

In interviews, be ready to discuss how you'd scale a system up or down and the trade-offs of different scaling strategies.

Further Reading (Scalability):

- What is Scalability in a Distributed System?
- Scalability in System Design: Vertical vs Horizontal
- Grokking System Design Scalability

- Scaling SQL Databases
- Approaching System Scalability Questions in Interviews
- Learning Tracks for System Design & Scalability Topics

13. Rate Limiting

Rate limiting is a mechanism used to control how many requests or actions a user (or a client like an API consumer) can perform in a given time window. It's like setting a speed limit for traffic to a service.

If someone (or something, like a script) tries to hit an API too many times too quickly, rate limiting will start blocking or throttling those requests.

This ensures that no single user can overwhelm the system and that resources are shared fairly among users.

Common scenarios for rate limiting include protecting APIs from abuse (e.g., preventing a single IP from making thousands of requests per second), stopping spam (limiting how many comments a user can post per minute), or generally preventing accidental overload due to bugs or aggressive usage.

There are different algorithms to implement rate limiting:

- **Token Bucket** and **Leaky Bucket** algorithms control the flow of requests by allowing tokens to fill a bucket at a fixed rate (and each request takes a token).
- **Fixed Window** and **Sliding Window** counters track request counts in time intervals (like 100 requests per minute).

The specifics of these algorithms are usually hidden behind the scenes in frameworks.

As a designer, you should know why rate limiting is important: it keeps your system **reliable** and **secure** by preventing abuse and ensuring stability.

It also affects user experience (for example, if a user hits a limit, the system might return an HTTP 429 "Too Many Requests" response or slow down responses).

Overall, rate limiting is a simple concept that greatly improves a system's robustness.

Further Reading (Rate Limiting):

- What is Rate Limiting?

- Grokking Rate Limiters
- Understanding API Rate Limiting (for Interviews)
- Rate Limiting Algorithms

14. Reliability

Reliability in system design refers to the ability of a system to function correctly and consistently over time.

A reliable system does what it's supposed to do, accurately, and can handle failures gracefully without losing data or crashing unexpectedly. It's about trustworthiness: can users trust that the system will behave correctly and maintain data integrity?

Several factors contribute to reliability:

- **Fault Tolerance:** The system can keep working even if some components fail (often through redundancy and backups).
- **Consistency:** The system processes requests accurately and doesn't produce corrupted data.
- **Recovery:** If something does go wrong, the system can recover quickly (for example, through automatic failovers or data restoration from backups).

Reliability often goes hand-in-hand with availability, but they're not identical.

Availability is about uptime (is the system up and reachable?), while reliability is about correct operation (is it doing the right thing without errors?).

For instance, a system could be up (available) but responding with errors or wrong data (not reliable).

In design, you increase reliability by eliminating single points of failure, using replication and backups, handling exceptions properly, and testing thoroughly.

A reliable system not only stays up, but also consistently delivers the expected results.

Further Reading (Reliability):

- The 4 Basic Pillars of System Design
- Availability vs Reliability: What's the Difference?

- Evaluating System Safety and Reliability in Design Scenarios

15. Availability

Availability is about ensuring that your system is up and reachable when users need it. It's often measured as a percentage of uptime.

For example, "99.9% availability" means the system may be down for at most 0.1% of the time (about 43 minutes in a month).

High availability is achieved by designing systems that have minimal downtime, even in the face of component failures or maintenance.

Key techniques to achieve high availability include:

- **Redundancy:** Having backup components or servers so that if one fails, another can take over. For example, multiple server instances behind a load balancer, or a primary-secondary database setup.

- **Failover Mechanisms:** Automatic detection of failures and instant switching to a healthy backup component when something goes wrong.

- **Eliminating Single Points of Failure:** Ensuring that no single component can bring down the entire system. This might involve using multiple data centers, backup networks, and distributed services.

High availability often requires trade-offs, like extra cost and complexity (more servers, sophisticated routing, etc.), but it's crucial for systems that need to run 24/7 (like online banking, e-commerce, or critical infrastructure).

Remember, availability is about **uptime** – making sure users can access the service whenever they want.

Combined with reliability (the system doing the right thing), it forms the backbone of user trust in your service.

Further Reading (Availability):

- High Availability in System Design (Basics)

- Availability vs Reliability (Differences)

16. Caching

Caching is one of the most powerful ways to improve system performance and reduce load on your backend servers. It works by temporarily storing frequently accessed data in a faster storage layer (like memory) so future requests can be served quickly without always hitting the main database.

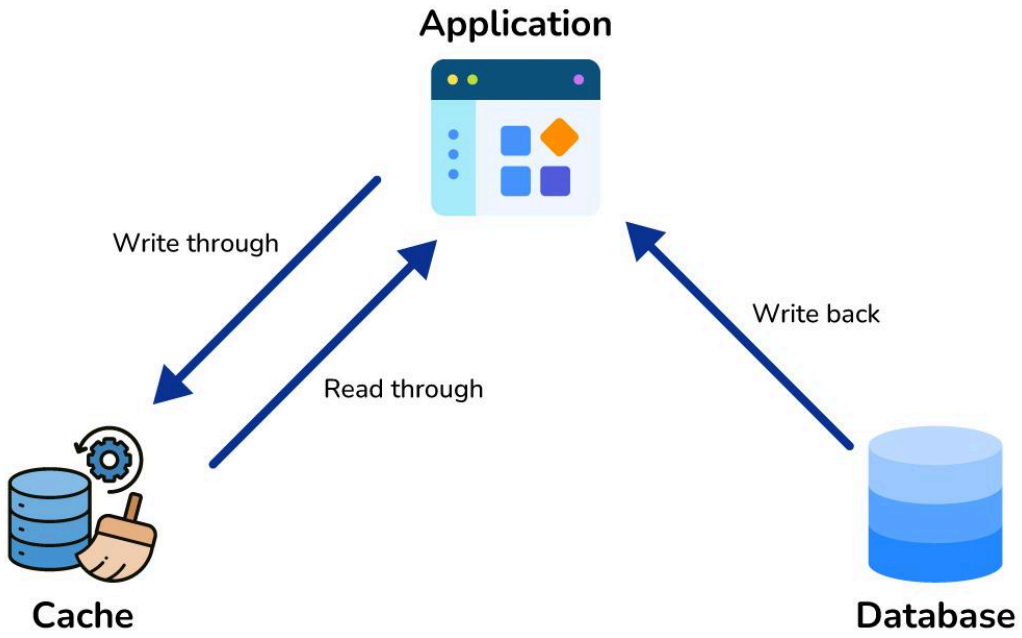
In system design interviews, caching often comes up when you need to make a system more scalable or responsive.

For example, when thousands of users request the same data, fetching it each time from a slow data store can overload your system.

A cache acts as a shortcut—it stores the result of that query, and when another user requests the same thing, it's returned instantly from memory.

There are different caching strategies:

- **Client-side caching** (in browsers or mobile apps) helps reduce server calls.
- **Server-side caching** (using systems like Redis or Memcached) stores results on your servers for faster reuse.
- **CDN caching** stores static files like images or videos closer to users around the world.



Caching

Another important concept is **cache eviction**—deciding what to remove when the cache is full. Common strategies include:

- **LRU (Least Recently Used)** – removes data that hasn't been accessed recently.
- **LFU (Least Frequently Used)** – removes data used the least often.
- **FIFO (First In, First Out)** – removes the oldest cached data first.

Caching can drastically speed up systems but introduces trade-offs like cache inconsistency (stale data) and complexity in invalidation (deciding when to update or clear cache).

Further Reading:

- Caching in System Design Interviews
- Cache Eviction Strategies

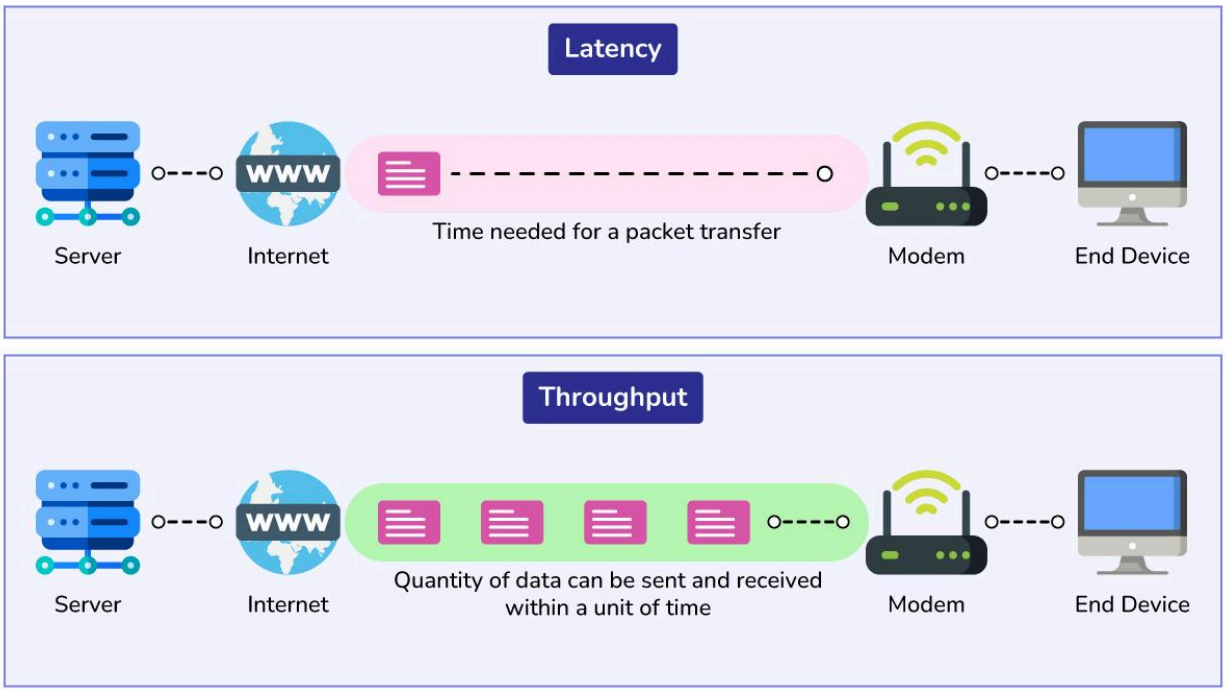
17. Latency vs Throughput

Latency and throughput are two core measures of system performance.

Latency is the time it takes for a single request to travel through the system — from when it's sent to when a response is received.

Think of it as the delay a user feels when clicking a button.

Throughput, on the other hand, is the total number of requests or operations the system can handle per second — like the width of a highway that determines how many cars can pass at once.



In system design, improving one can often hurt the other.

For example, batching requests can increase throughput but may also add delay to each request, raising latency.

The key is to balance both depending on your use case: for gaming or chat apps, prioritize latency; for data pipelines or streaming systems, aim for throughput.

To maintain good performance, engineers often track p95 or p99 latency (the slowest 5% or 1% of requests) and manage bottlenecks using techniques like load balancing and backpressure.

Further Reading:

- Understanding Long Tail Latency
- Trade-offs Between Latency and Throughput
- Refining Architectural Targets
- Handling High Throughput Requirements

18. Forward Proxy vs Reverse Proxy

A **proxy** acts as a middleman for requests — but depending on where it sits, its role changes.

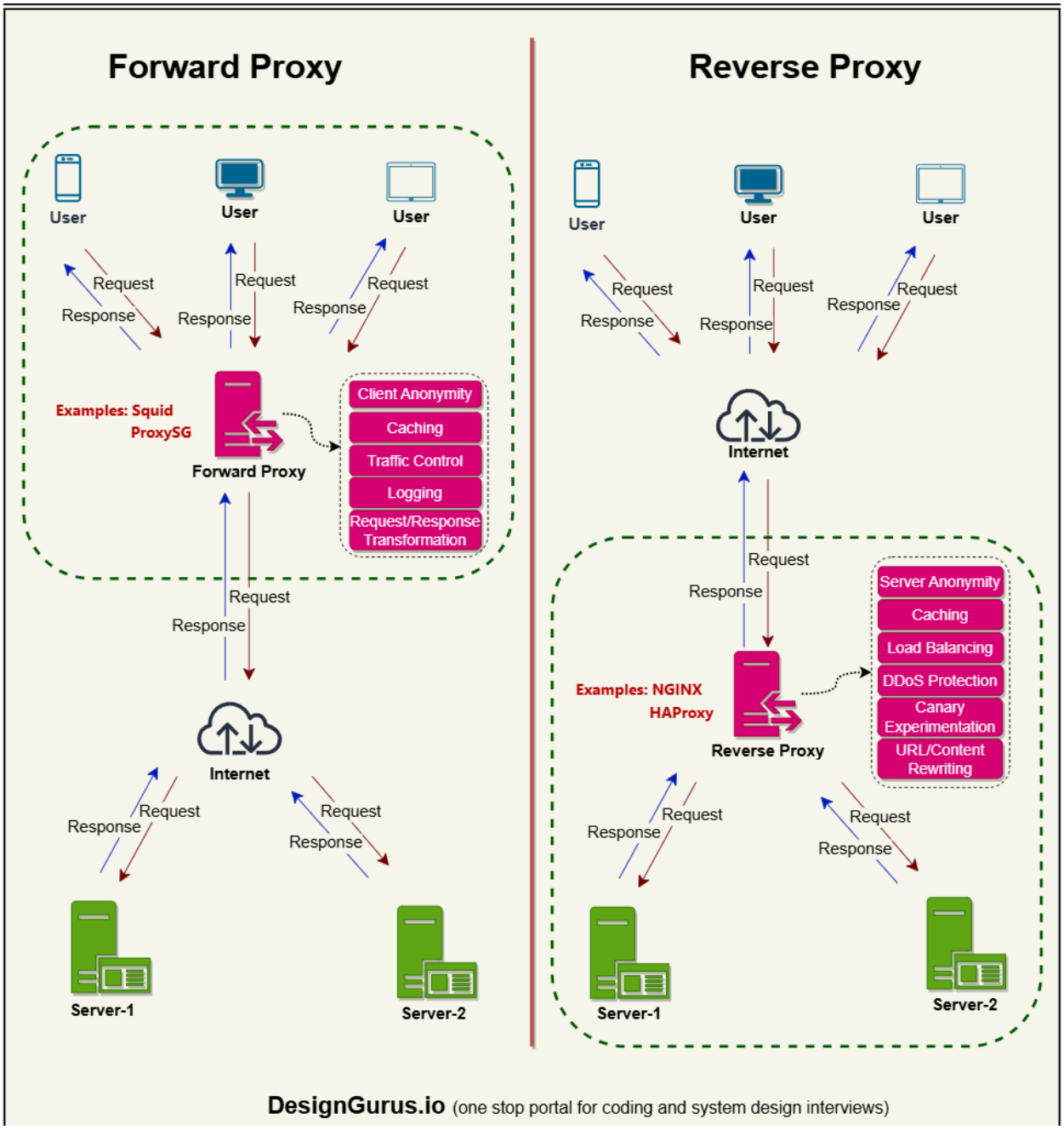
A **forward proxy** stands in front of the client and sends outbound requests on its behalf.

It's often used in corporate networks or VPNs to filter traffic, enforce security policies, and cache content.

Users connect to the proxy, and it fetches the data for them.

A **reverse proxy**, however, sits in front of servers. It handles inbound traffic from clients, routes requests to the right backend servers, and can handle caching, SSL termination, and authentication.

Reverse proxies improve scalability, load balancing, and security by hiding the backend infrastructure.



Many popular tools like Nginx or HAProxy serve as reverse proxies.

In interviews, remember: **forward proxy** protects clients; **reverse proxy** protects servers.

Further Reading:

- Forward Proxy vs Reverse Proxy Server
- Proxy vs Reverse Proxy Differences

- Reverse Proxy vs Load Balancer

19. Domain Name System (DNS)

The **Domain Name System (DNS)** is like the phone book of the internet — it translates human-friendly domain names (like google.com) into machine-readable IP addresses.

When you type a website into your browser, your device sends a DNS query to a resolver, which looks up the address through a hierarchy: root servers, top-level domains (.com, .org), and the site's authoritative name server.

To speed up lookups, DNS relies heavily on **caching**, using a Time-To-Live (TTL) value to decide how long results are stored.

DNS not only connects users to websites but also supports complex architectures — such as global load balancing, failover routing, and CDNs — by mapping traffic to the closest or healthiest data center.

Understanding DNS is crucial in system design, as slow or misconfigured DNS can cause significant latency.

Further Reading:

- What is DNS and Why It's Important in System Design
- DNS Basics

20. Long Polling vs WebSockets

Both **long polling** and **WebSockets** enable real-time communication between clients and servers, but they work differently.

Long polling uses regular HTTP requests that the server holds open until new data is available.

Once the server responds, the client immediately opens a new request. This makes it simple to implement but less efficient for high-frequency updates.

WebSockets, however, create a persistent two-way connection after an initial handshake.

This allows continuous data exchange without reopening connections — ideal for chat apps, gaming, or live dashboards where instant updates matter.

WebSockets reduce overhead but require stateful connections and extra care for scaling and reconnects.

Some systems also use **Server-Sent Events (SSE)** for one-way updates when full duplex isn't needed.

In summary, long polling is simpler and firewall-friendly; WebSockets are faster and more interactive.

Further Reading:

- WebSockets vs Server-Sent Events vs Long Polling
- Polling vs Long Polling vs Webhooks
- Understanding HTTP Long Polling
- How WebSockets Enable Real-Time Communication

By understanding these core concepts and how they interplay in real systems, you'll be well on your way to mastering system design.

Remember, the key in interviews is not just knowing the definitions but being able to connect these ideas to build systems that meet specific needs.

Good luck with your interviews and happy designing!

FAQs

Q: What is system design, and why is it important in tech interviews?
System design is the process of defining the architecture, components, and data flow of a software system to meet specific requirements. In tech interviews, it's important because it shows how you approach building scalable, reliable systems in real-world scenarios. Interviewers want to see if you can break down a complex problem, consider trade-offs, and design a solution that handles large scale, potential failures, and evolving requirements.

Q: How should a beginner start with system design preparation?
Beginners should start by understanding the core concepts (like the ones in this crash course) and how they fit together. It's helpful to follow a structured approach, such as the System Design Master Template, to cover all bases. Begin with small design problems (e.g., design a URL shortener or a simple chat app) and practice outlining the components — clients, servers, databases, caches, etc. Gradually tackle more complex systems. Also, review case studies of real system architectures to see how the pros do it. Remember, it's okay to ask clarifying questions in an interview and to start simple, then refine your design with more details.

Q: What's the difference between a load balancer and an API gateway?
A load balancer and an API gateway are both entry points in a system, but they have different roles. A **load balancer** distributes incoming requests across multiple servers to balance the load and keep the service available. An **API gateway** acts as a single entry point for clients to

access various services or microservices; it handles things like routing requests to the right service, authentication, rate limiting, and sometimes response caching or transformations. In essence, a load balancer is about traffic distribution for reliability and scaling, while an API gateway is about request management and protocol translation for APIs. In many architectures, you might use both: the load balancer sits in front to distribute traffic to multiple instances of your API gateway or service, and then the API gateway handles directing each request to the appropriate backend service.

Q: When should I use SQL vs NoSQL databases?
Use **SQL (relational)** databases when your data is structured and relational (tables with defined schema), you need complex joins or transactions, and strong consistency is crucial (e.g., for financial data or user account information). SQL databases excel at ensuring data integrity and supporting complex queries. Use **NoSQL** when you need to handle massive scale or unstructured data, require flexible schemas (e.g., storing varied data without a strict schema), or have use cases like caching, real-time analytics, or big data where horizontal scaling is key. NoSQL databases are often used for high-speed reads/writes, distributed data storage, and situations where eventual consistency is acceptable. Sometimes you might use both in a system: for example, an e-commerce site might use SQL for transactions and a NoSQL store for caching and sessions.

Q: What are some common pitfalls in system design interviews and how can I avoid them?
Common pitfalls include: jumping into a design without clarifying requirements, focusing on one aspect (like scalability) and neglecting others (like security or simplicity), designing something overly complex for the given needs, and not considering failure scenarios. To avoid these, always start by gathering requirements (ask about use cases, constraints, scale). Outline your high-level approach before diving into details. Discuss trade-offs openly (every design decision has pros and cons). Think about bottlenecks and how the system behaves under failure (e.g., "What if this component goes down?"). And manage your time – it's better to cover the main components and their interactions than to get lost in the weeds of one part of the system.

Q: How do microservices communicate with each other?
Microservices can communicate in two primary ways: **synchronously** or **asynchronously**. Synchronous communication typically means using direct requests — for example, Service A calls Service B's API (often over HTTP/REST or gRPC) and waits for a response. This is straightforward but can create tight coupling (if B is slow or down, A is affected). Asynchronous communication uses messaging: Service A publishes a message or event to a **message broker** (like a queue or topic), and Service B (and others) can consume that message on their own time. This decouples services because A doesn't wait; it just sends the message. Asynchronous patterns (using message queues and brokers) improve resilience and allow for things like event-driven architecture, but they're a bit more complex to design and reason about. In practice, many systems use a mix: critical requests might be direct calls, while background tasks and updates use asynchronous events.

Q: Can you briefly explain the difference between horizontal and vertical scaling?
Vertical scaling means making one server or instance more powerful — add CPU, RAM, etc. It's like upgrading a computer to handle more load. **Horizontal scaling** means adding more servers or instances to handle load — like adding more workers to share a job. Vertical scaling is simpler (no change in architecture) but has physical limits and can be expensive at the high end. Horizontal scaling is more flexible for very large scale; you keep adding machines as needed and distribute the work (often using load balancers, caches, partitioning data, etc.). Most large systems rely on horizontal scaling to grow, because you can only make a single machine so powerful, but designing for horizontal scale requires thinking about how to split work and data across machines.