

# ☆ Top 50 System Design Interview Questions ☆

## Page 1 (Questions 1 - 5)

### 1. What is Distributed System ?

- A distributed system is a collection of independent computers that appear to users as a single system.
- These computers communicate over a network to achieve a common goal.
- It improves scalability, availability, fault tolerance and performance.
- Examples: Google, Amazon, WhatsApp, etc.

### 2. Difference between Vertical and Horizontal Scaling.

| Vertical Scaling (Scale Up)                                                                                                                                                                                                                                | Horizontal Scaling (Scale Out)                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Add more power (CPU, RAM, Disk) to a single server.</li><li>• Limited by max capacity of the server.</li><li>• Downtime may be required.</li><li>• Example: Upgrading a server from 8GB RAM to 32GB RAM.</li></ul> | <ul style="list-style-type: none"><li>• Add more servers to the system.</li><li>• Nearly unlimited scalability.</li><li>• No downtime (in most cases).</li><li>• More complex to manage.</li><li>• Example: Adding more instances behind a load balancer.</li></ul> |

### 3. What is Load Balancing and what are common algorithms ?

- Load Balancing distributes incoming traffic across multiple servers.
- It prevents any single server from getting overloaded and improves availability and reliability.
- Common Algorithms:
  - ① Round Robin - Requests are distributed one by one in circular order.
  - ② Least Connections - Request goes to server with least active connections.
  - ③ IP Hash - Same IP always goes to the same server.
  - ④ Weighted Round Robin - Servers have different weights.
  - ⑤ Least Response Time - Request goes to server with least response time.

### 4. Difference between SQL and NoSQL databases, and when to use each.

| Feature        | SQL (Relational DB)                                        | NoSQL (Non-Relational DB)                                            |
|----------------|------------------------------------------------------------|----------------------------------------------------------------------|
| Data Structure | Table (Rows & Columns)                                     | Key-Value, Document, Column, Graph                                   |
| Schema         | Fixed Schema                                               | Flexible Schema                                                      |
| Scalability    | Vertical Scaling                                           | Horizontal Scaling                                                   |
| ACID Support   | Full ACID                                                  | Usually BASE (Eventual Consistency)                                  |
| Examples       | MySQL, PostgreSQL, Oracle                                  | MongoDB, Cassandra, Redis, DynamoDB                                  |
| When to Use    | When you need strong consistency and complex transactions. | When you need high scalability, high availability and flexible data. |

### 5. What is Caching, and where can caching be applied in a system ?

- Caching is storing frequently accessed data in a fast storage (cache) so that future requests can be served quickly.
- It reduces latency, database load and improves performance.
- Where it can be applied:
  - Browser Cache (Static files, Images, CSS, JS)
  - Application Cache (Redis, Memcached)
  - Database Query Cache
  - CDN Cache (for content delivery)

# ★ Top 50 System Design Interview Questions ★

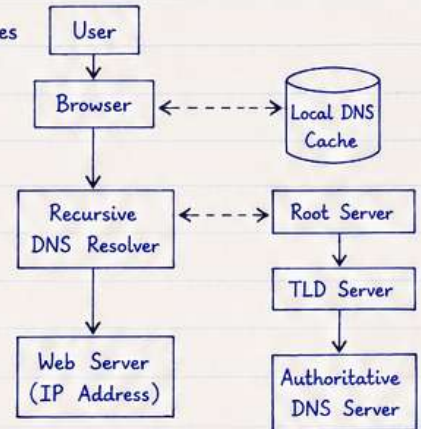
## Page 2 (Questions 6 - 10)

### 6. Explain CDN (Content Delivery Network) and its benefits.

- A CDN is a distributed network of edge servers placed in multiple locations to deliver content to users.
- It caches static content (images, videos, CSS, JS, files) close to users.
- Benefits: Reduced latency, lower bandwidth cost, improved availability and scalability, protects origin server from traffic spikes.
- Examples: Cloudflare CDN, AWS CloudFront, Akamai.

### 7. What is DNS and how does it work in system design?

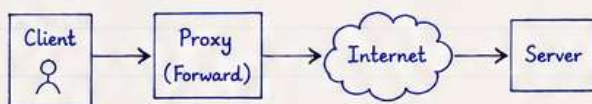
- DNS (Domain Name System) translates human-readable domain names (e.g., www.example.com) into IP addresses.
- It works in a hierarchical and distributed manner.
- Steps:
  - ① User enters a domain name.
  - ② Browser checks cache (local DNS cache).
  - ③ If not found, request goes to Recursive DNS Resolver.
  - ④ Resolver queries Root → TLD → Authoritative DNS Server.
  - ⑤ Gets IP address and returns to the user.
  - ⑥ Browser connects to the server using IP.



### 8. What is a proxy vs a reverse proxy?

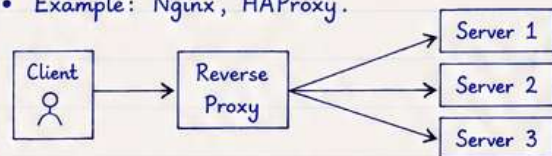
#### Proxy (Forward Proxy)

- Acts on behalf of clients.
- Hides client IP from the server.
- Used for caching, access control, anonymity.
- Example: Corporate proxy, browser proxy.



#### Reverse Proxy

- Acts on behalf of servers.
- Hides server details from clients.
- Used for load balancing, SSL termination, caching, security.
- Example: Nginx, HAProxy.



### 9. Difference between latency and throughput.

| Latency                                                                                                                                                                                                                                        | Throughput                                                                                                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Time taken for a single request from start to finish.</li> <li>• Measured in ms (milliseconds).</li> <li>• Lower latency is better.</li> <li>• Example: API response time, page load time.</li> </ul> | <ul style="list-style-type: none"> <li>• Number of requests processed per unit time.</li> <li>• Measured in RPS (requests per second).</li> <li>• Higher throughput is better.</li> <li>• Example: Server can handle 1000 RPS.</li> </ul> |

### 10. What is database indexing and how does it improve performance?

- Indexing is a data structure that improves the speed of data retrieval operations.
- It works like the index in a book.
- Instead of scanning the entire table, database uses the index to locate rows faster.
- Improves performance for read-heavy queries (SELECT, WHERE, JOIN, ORDER BY).
- Trade-off: Uses extra storage and slows down write operations (INSERT, UPDATE, DELETE).

### 11. Explain the client-server architecture model.

- It is a computing model where clients request services or resources from servers.
- Clients send requests and servers process them and send back responses.
- It enables centralized data management, security and easy maintenance.
- Examples: Web applications, Email services, File servers.

### 12. What is an API Gateway and why is it used?

- API Gateway is a server that acts as a single entry point for all client requests.
- It routes requests to appropriate microservices.
- It handles authentication, authorization, rate limiting, monitoring, logging, caching, etc.
- Benefits: Simplifies client communication, improves security, and provides centralized control.

### 13. Difference between REST and GraphQL.

| REST                                                  | GraphQL                                      |
|-------------------------------------------------------|----------------------------------------------|
| • Uses multiple endpoints for different resources.    | • Uses a single endpoint for all requests.   |
| • Over-fetching and under-fetching of data can occur. | • Client requests only the data it needs.    |
| • Uses HTTP methods (GET, POST, PUT, DELETE).         | • Uses a single HTTP method (POST).          |
| • Versioning is required.                             | • No versioning, schema is self-descriptive. |
| • Examples: Twitter API, GitHub REST API.             | • Example: Facebook, Shopify.                |

### 14. What is database replication and why is it needed?

- Replication is the process of copying data from one database (master) to one or more databases (replicas).
- It improves availability, read scalability and disaster recovery.
- Types: ① Master-Slave Replication  
② Master-Master Replication
- Use Case: Handle read-heavy workloads, backup and failover.

### 15. What is database sharding and how is it different from partitioning?

- Sharding is horizontal scaling where data is distributed across multiple databases/servers based on a shard key.
- Partitioning is splitting a table into smaller parts within the same database.
  - Sharding: Across multiple databases/servers.
  - Partitioning: Within the same database.
  - Sharding improves scalability and handles larger data.
  - Partitioning improves performance and manageability.

## 16. What is the CAP theorem and how does it affect design decisions?

- CAP theorem states that a distributed system can only guarantee two out of the following three properties at the same time.
  - Consistency - All nodes see the same data at the same time.
  - Availability - Every request gets a response.
  - Partition Tolerance - System continues to work despite network partitions.
- Design decisions depend on which two are more important for the system.
- Examples: CP systems (e.g., HBase), AP systems (e.g., Cassandra), CA systems (rare in distributed systems).

## 17. Strong consistency vs eventual consistency.

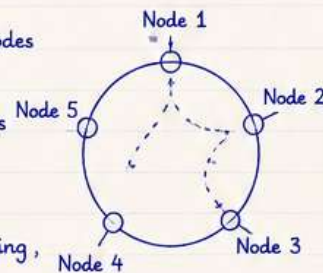
| Strong Consistency                                                                                                                                                                                                                                       | Eventual Consistency                                                                                                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>All reads return the most latest write across all nodes.</li><li>Data is consistent at all times.</li><li>Lower availability and higher latency.</li><li>Example: Banking systems, Inventory management.</li></ul> | <ul style="list-style-type: none"><li>Data will be consistent eventually, not immediately.</li><li>Higher availability and better performance.</li><li>Temporary inconsistencies may occur.</li><li>Example: Social media feeds, DNS, Shopping carts.</li></ul> |

## 18. What is a message queue - Kafka vs RabbitMQ, when to use which?

| Feature     | Kafka                                                  | RabbitMQ                                                  |
|-------------|--------------------------------------------------------|-----------------------------------------------------------|
| Model       | Publish-Subscribe (Log based)                          | Message Broker (Queue based)                              |
| Performance | High throughput, optimized for scale                   | Good for smaller scale                                    |
| Persistence | Messages stored on disk                                | Messages can be persistent or in-memory                   |
| Ordering    | Ordered within a partition                             | Ordered within a queue                                    |
| Use Case    | Real-time streaming, events, logs, analytics pipelines | Task queues, background jobs, microservices communication |
| When to Use | High volume, real-time data streaming                  | Reliable message delivery, task queues, complex routing   |

## 19. Explain consistent hashing and where it's used.

- Consistent hashing is a technique used to distribute data across nodes in a distributed system.
- When a node is added or removed, only a small portion of keys need to be redistributed.
- It minimizes data movement and improves scalability.
- Used in: Distributed caches (e.g., Redis Cluster), Load balancing, CDN, Databases.



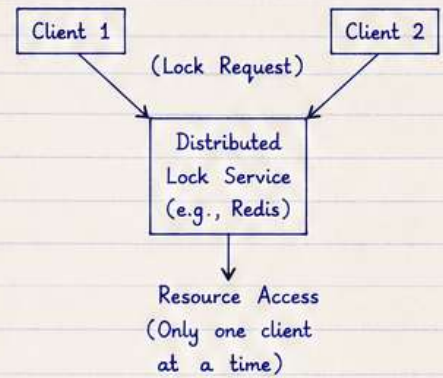
Keys are mapped to the next node in the clockwise direction.

## 20. Synchronous vs asynchronous communication.

| Synchronous                                                                                                                                                                                                                                            | Asynchronous                                                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>Client waits for the response from the server.</li><li>Communication happens in real-time.</li><li>Blocking - The client is blocked until response is received.</li><li>Example: REST APIs, RPC calls.</li></ul> | <ul style="list-style-type: none"><li>Client does not wait for the response.</li><li>Communication happens in the background.</li><li>Non-blocking - Client can continue other work.</li><li>Example: Message queues, Event-driven systems, Webhooks.</li></ul> |

## 21. What is a distributed lock and why is it needed ?

- A distributed lock is a mechanism that ensures only one process or instance can access a shared resource at a time across multiple machines.
- It prevents issues like race conditions, duplicate processing, and data inconsistencies.
- Common implementations: Redis (SETNX with expiry), Zookeeper, etcd.
- Use Cases: Scheduling jobs, inventory updates, leader election, preventing duplicate payments.



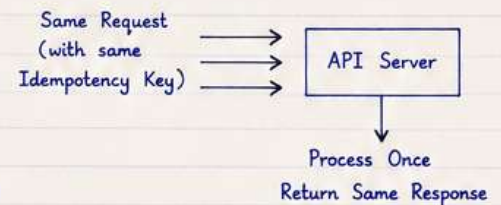
## 22. Explain database transactions and ACID properties.

- A transaction is a sequence of one or more database operations treated as a single unit of work.
- ACID properties ensure reliable processing of transactions.

|             |                                                                                         |
|-------------|-----------------------------------------------------------------------------------------|
| Atomicity   | All operations in a transaction are completed successfully, or none are applied.        |
| Consistency | Transaction brings the database from one valid state to another, maintaining integrity. |
| Isolation   | Concurrent transactions do not interfere with each other.                               |
| Durability  | Once a transaction is committed, changes are permanent even in case of failures.        |

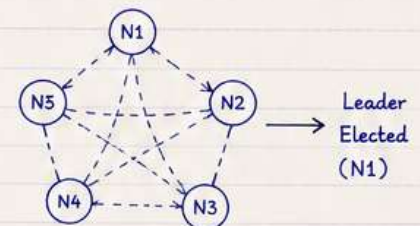
## 23. What is idempotency, and why does it matter in API design?

- Idempotency means that multiple identical requests have the same effect as a single request.
- It is important in APIs to handle retries safely without causing duplicate operations.
- How: Use idempotency keys, unique request IDs, or versioning.
- Example: Payment API should not charge the user multiple times if the same request is retried.



## 24. Explain the leader election problem in distributed systems.

- In a distributed system, leader election is the process of selecting one node as the leader among many nodes.
- The leader is responsible for coordinating tasks, assigning work, or managing resources.
- Challenges: Network failures, split-brain, and ensuring only one leader.
- Solutions: Zookeeper, etcd, Raft, Paxos.

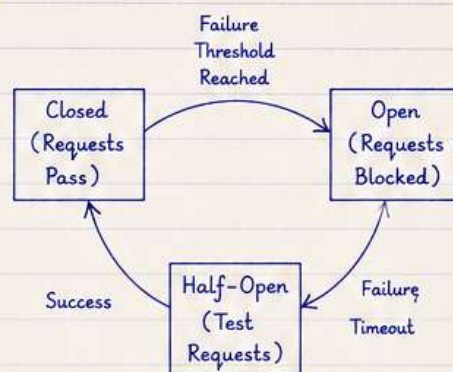


## 25. Monolith vs microservices architecture.

| Monolith Architecture                                                                                                                                                                                                                                                                               | Microservices Architecture                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Entire application is built and deployed as a single unit.</li> <li>• All components share the same codebase and database.</li> <li>• Easier to develop and deploy initially.</li> <li>• Harder to scale and maintain as the application grows.</li> </ul> | <ul style="list-style-type: none"> <li>• Application is divided into small, independent services.</li> <li>• Each service has its own codebase and database.</li> <li>• Easier to scale, deploy and maintain.</li> <li>• More complex due to distributed nature, network calls, and data consistency.</li> </ul> |

## 26. What is a circuit breaker pattern and why is it used?

- Circuit Breaker is a design pattern used to detect failures in external services and prevent continuous requests to a failing service.
- It improves system resilience and prevents cascading failures.
- It has three states.
  - ① Closed — Normal operation, requests pass through.
  - ② Open — Failure detected, requests are blocked.
  - ③ Half-Open — Limited requests allowed to check if service is recovered.
- Used in microservices, API calls, databases, third-party services.



## 27. What is database denormalization and when should you use it?

- Denormalization is the process of adding redundant data or combining tables to reduce joins and improve read performance.
- It increases read speed but may affect write performance and data consistency.
- Use it when:
  - Read operations are very frequent.
  - Performance is critical.
  - Data changes are infrequent.
  - Reporting and analytics queries.

## 28. Explain optimistic vs pessimistic locking.

| Optimistic Locking                  | Pessimistic Locking                         |
|-------------------------------------|---------------------------------------------|
| Assumes conflicts are rare.         | Assumes conflicts are frequent.             |
| No locks during read.               | Locks data during read/write.               |
| Checks for conflict during commit.  | Prevents others from accessing locked data. |
| Better for read-heavy systems.      | Better for write-heavy systems.             |
| Example: Version number, Timestamp. | Example: DB row locks, SELECT FOR UPDATE.   |

## 29. How does Kafka ensure message ordering and fault tolerance at scale?

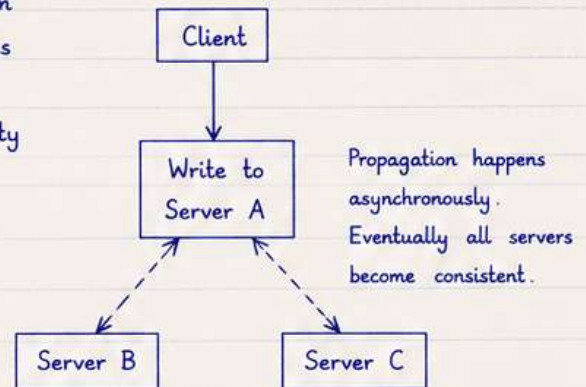
- Kafka ensures ordering by maintaining order within a partition.
- All messages with the same key go to the same partition.
- Each partition is replicated across multiple brokers for fault tolerance.
- If a broker fails, leader election happens and a replica becomes the new leader.
- Uses acknowledgment (acks) and ISR (In-Sync Replicas) to ensure durability.

## 30. Explain the concept of quorum in distributed databases.

- Quorum is the minimum number of nodes that must agree on a read or write operation for it to be successful.
- It helps maintain consistency and availability.
- Types:
  - Read Quorum (R): Minimum nodes required to read.
  - Write Quorum (W): Minimum nodes required to write.
- Rule:  $R + W > N$  ( $N$  = total number of nodes)
- Example: In a 5-node system, if  $R = 3$  and  $W = 3$ , then  $R + W = 6 > 5$ .
- Used in systems like Cassandra, DynamoDB.

### 32. What is eventual consistency and where is it used ?

- Eventual consistency is a consistency model where the system will become consistent over time, given that no new updates are made.
- It allows for temporary inconsistencies to improve availability and partition tolerance.
- Data may be inconsistent for a short period after a write.
- Used in systems that prioritize availability and partition tolerance over immediate consistency.
- Examples: DNS, Cassandra, DynamoDB, Amazon S3.



### 33. What is load balancing and what are its types ?

- Load balancing distributes incoming traffic across multiple servers to ensure no single server is overwhelmed.
- It improves availability, scalability and reliability.
- Types:
  - ① Round Robin - Distributes requests sequentially.
  - ② Least Connections - Sends request to server with least active connections.
  - ③ IP Hash - Uses client IP to determine the server.
  - ④ Least Response Time - Sends request to server with lowest response time.
  - ⑤ Weighted Round Robin - Distributes based on server weights.

### 34. Explain the difference between vertical scaling and horizontal scaling.

| Vertical Scaling (Scale Up)                                                                                                                                                                                                                                                                                            | Horizontal Scaling (Scale Out)                                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Add more power (CPU, RAM, storage) to an existing server.</li><li>• Limited by the capacity of a single machine.</li><li>• Downtime may be required.</li><li>• Cost increases quickly at high scale.</li><li>• Example: Upgrading a server from 8GB RAM to 32GB RAM.</li></ul> | <ul style="list-style-type: none"><li>• Add more servers (nodes) to handle increased load.</li><li>• Almost unlimited scalability.</li><li>• High availability and fault tolerance.</li><li>• More cost-effective at large scale.</li><li>• Example: Adding more instances behind a load balancer.</li></ul> |

### 35. What is caching and what are the different caching strategies ?

- Caching stores frequently accessed data in a fast storage layer to reduce latency and load on backend systems.
- It improves response time and scalability.
- Caching Strategies:
  - ① Cache Aside (Lazy Loading) - App checks cache first, if miss, load from DB and update cache.
  - ② Read Through - Cache handles read and loads data from DB on cache miss.
  - ③ Write Through - Data is written to cache and DB simultaneously.
  - ④ Write Behind (Write Back) - Data is written to cache first, then to DB asynchronously.
  - ⑤ Refresh Ahead - Cache is refreshed before expiration.

## 36. Design a URL Shortener

### Requirements

- Convert long URLs to short URLs.
- Redirect short URLs to original URLs.
- Track analytics (clicks, time, location etc.).

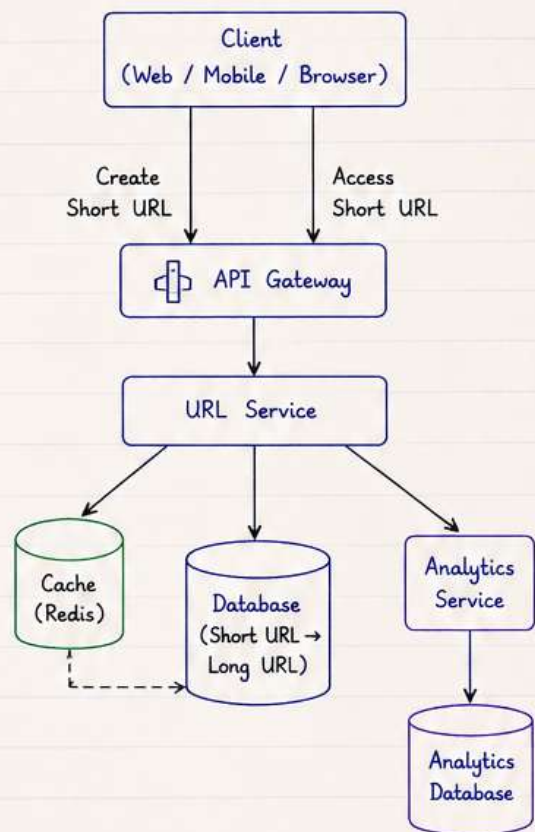
### High Level Design - Key Components

- Client - Web / Mobile / Browser
- API Gateway - Handles all requests
- URL Service - Generates short code, stores and fetches URLs
- Database - Stores mapping of short URL → long URL
- Cache (Redis) - Fast lookup for redirects
- Analytics Service - Tracks and stores click metrics
- Load Balancer - Distributes traffic across services

### How it works

1. User submits a long URL.
2. System generates a short code and stores mapping in DB (and cache).
3. When short URL is accessed, system fetches original URL and redirects.
4. Analytics service records the click.

### High Level Design Diagram



## 37. Design a Rate Limiter

### Requirements

- Limit the number of requests from a user/IP within a given time window.
- Prevent abuse and ensure system stability.
- Should be fast and scalable.

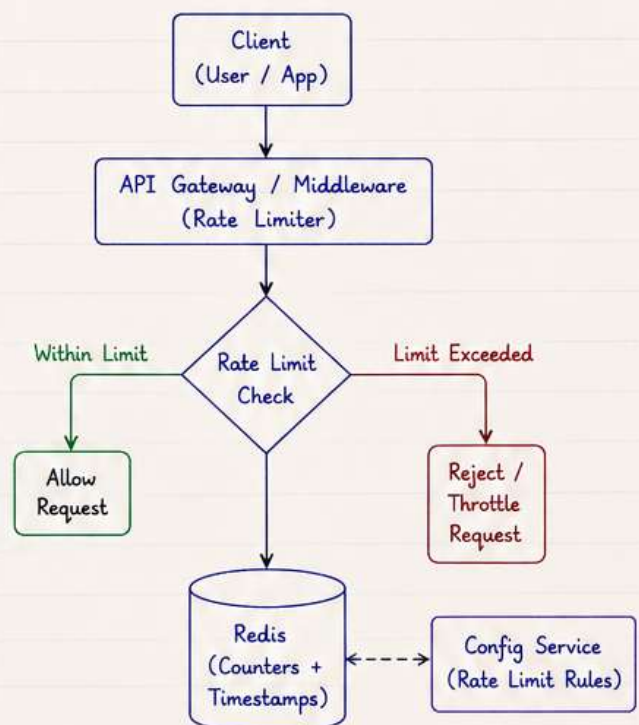
### High Level Design - Key Components

- Client - Application / User
- API Gateway / Middleware - Intercepts and checks requests
- Rate Limiter Service - Applies rate limiting logic
- Data Store (Redis) - Stores counters and timestamps
- Configuration Service - Manages rate limit rules (per user/IP/endpoint)

### How it works

1. On each request, rate limiter checks how many requests the user/IP has made in the time window.
2. If count < limit → Allow request and increment count.
3. If count ≥ limit → Reject or throttle the request.
4. Counters expire after the time window.

### High Level Design Diagram



## 38. Design a Parking Lot System

### Requirements

- Park vehicles of different types (car, bike, truck).
- Find available parking slot.
- Park in and park out (checkout).
- Calculate parking fee.

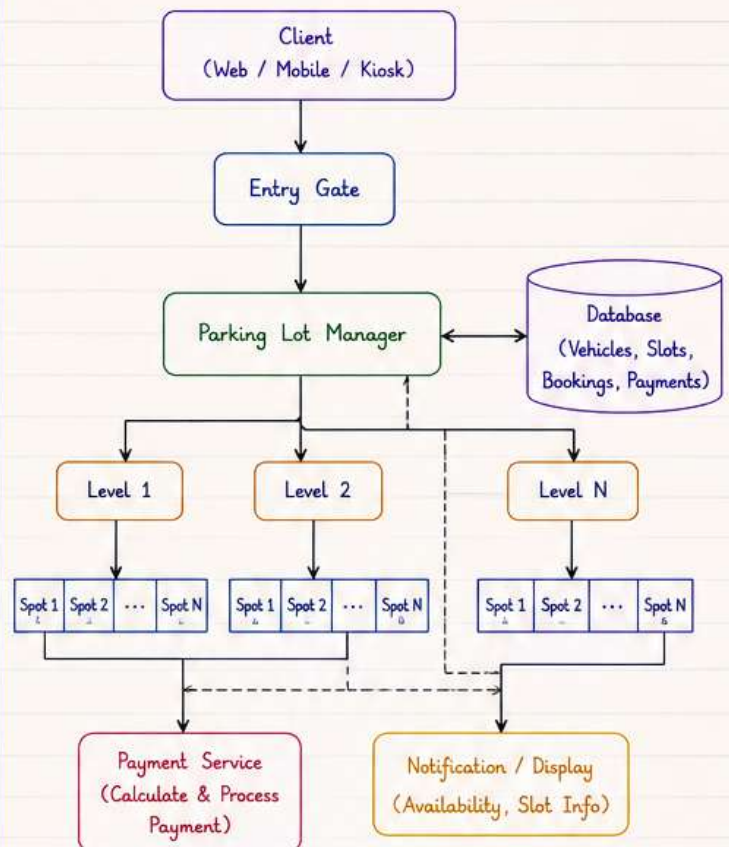
### High Level Design - Key Components

- Client - Web / Mobile / Kiosk
- Entry / Exit Gate - Handles vehicle entry/exit
- Parking Lot Manager - Manages slots, assignments and pricing
- Parking Level - Represents each floor/level
- Parking Spot - Individual slot
- Payment Service - Handles payments
- Database - Stores parking records, vehicle info, payments
- Notification / Display (optional) - Shows availability, slot info

### How it works

1. Vehicle enters, system finds available slot.
2. Slot is assigned and vehicle is parked.
3. On exit, system calculates fee.
4. Payment is processed and slot is released.

### High Level Design Diagram



## 39. Design a Notification System

### Requirements

- Send notifications to users through multiple channels (push, email, SMS, in-app).
- Support high delivery volume.
- Handle failures and retries.
- Notification preferences and scheduling.

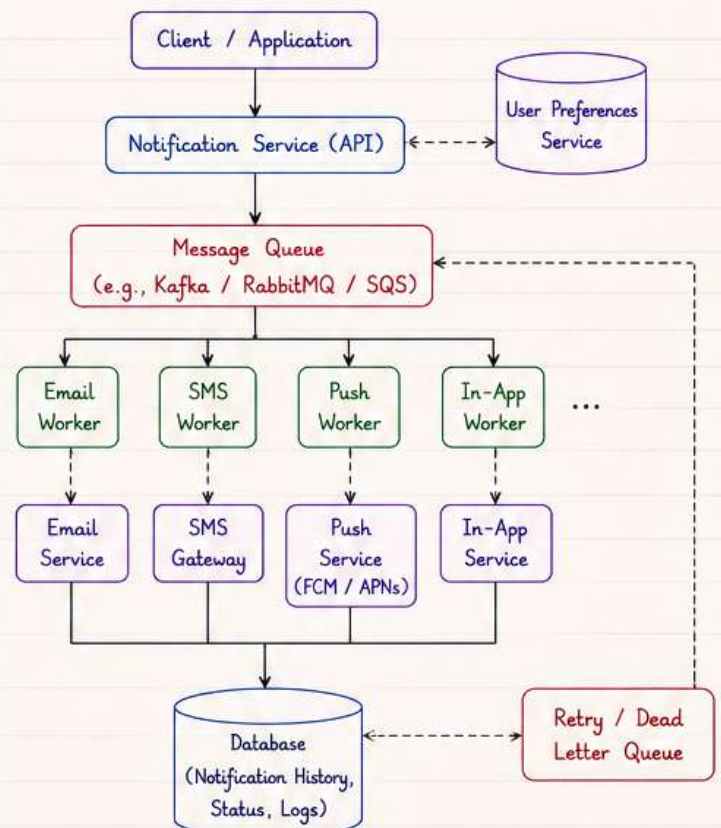
### High Level Design - Key Components

- Client / Application - Triggers notification
- Notification Service (API) - Accepts and validates requests
- Message Queue - Decouples producer and consumers
- Notification Workers - Pick messages and deliver via respective channels
- Channel Services - Email, SMS, Push, In-App
- User Preferences Service - Stores user preferences
- Database - Stores notification history, status
- Retry / Dead Letter Queue - Handles failures

### How it works

1. Application sends notification request to API.
2. API validates and puts message in queue.
3. Workers consume messages and send via channel.
4. Status is updated in DB. Failures go to retry queue.

### High Level Design Diagram



## 40. Design WhatsApp (High Level)

### Requirements

- Send 1-to-1 and group messages.
- Real-time message delivery and read receipts.
- Support media (images, videos, docs, voice).
- Offline message storage and sync.
- Show online status, last seen.

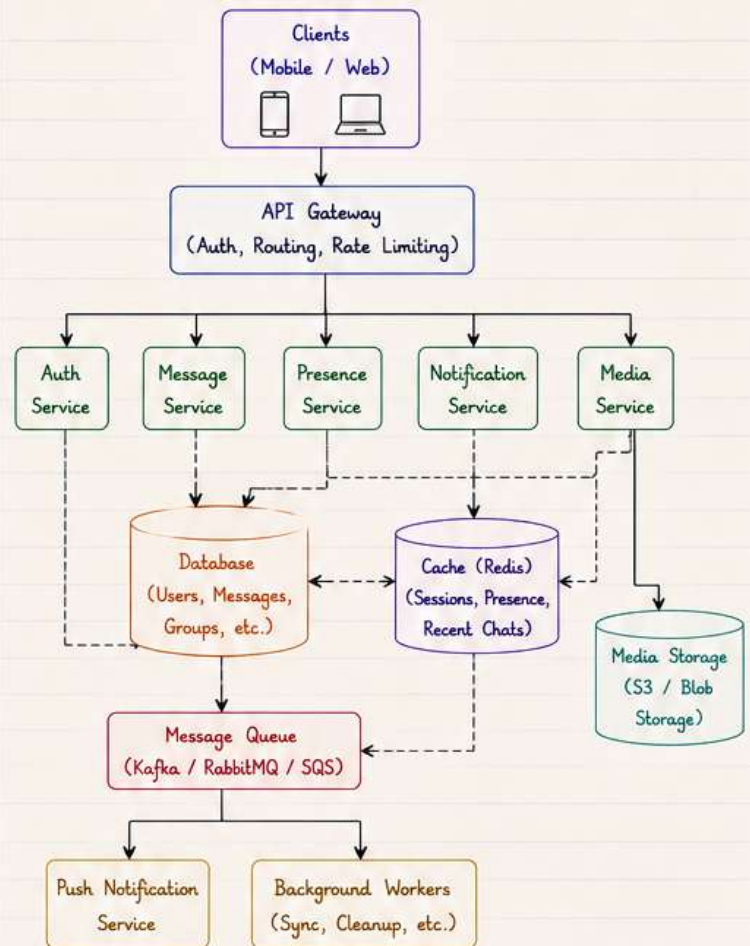
### High Level Design - Key Components

- Client - Mobile / Web application.
- API Gateway - Auth, routing, rate limiting.
- Authentication Service - Handles login, session, tokens.
- Message Service - Send, store and fetch messages.
- Presence Service - Online status, last seen.
- Notification Service - Push notifications.
- Media Service - Upload and serve media files.
- Database - Stores users, messages, groups.
- Cache (Redis) - Recent chats, presence, sessions.
- Message Queue - For async processing and push notifications.

### How it works

1. User sends message → Message Service.
2. Message stored in DB and pushed to recipients in real-time.
3. If recipient offline, message stored and delivered later.
4. Notification Service sends push notification.
5. Clients sync chats when they come online.

### High Level Design Diagram



## 41. Design E-commerce Checkout System

### Requirements

- Add items to cart and place order.
- Apply coupons and calculate totals.
- Multiple payment methods.
- Handle payment success / failure.
- Send order confirmation.

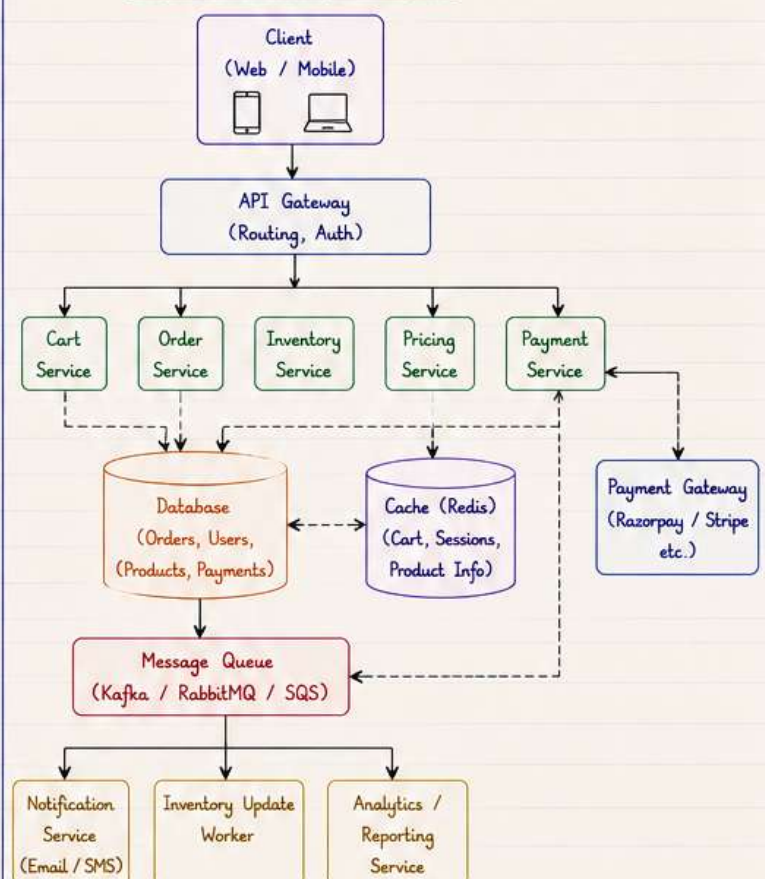
### High Level Design - Key Components

- Client - Web / Mobile application.
- API Gateway - Routes and secures requests.
- Cart Service - Manage cart, items.
- Order Service - Create and manage orders.
- Inventory Service - Check and reserve stock.
- Pricing Service - Calculate price, apply discounts.
- Payment Service - Process payments.
- Notification Service - Email / SMS confirmations.
- Database - Orders, users, payments, products.
- Message Queue - For async operations.

### How it works

1. User adds items to cart.
2. User clicks checkout → Pricing & Inventory checks.
3. Order created → Payment initiated.
4. On payment success → Order confirmed, inventory updated, notification sent.
5. On failure → Order cancelled / retry.

### High Level Design Diagram



## 42. Design a File Storage System (e.g., Google Drive / Dropbox)

### Requirements

- Store and manage large files.
- Upload, download, delete, update files.
- Share files and manage permissions.
- Support versioning and restore.
- Scalable, highly available and durable.

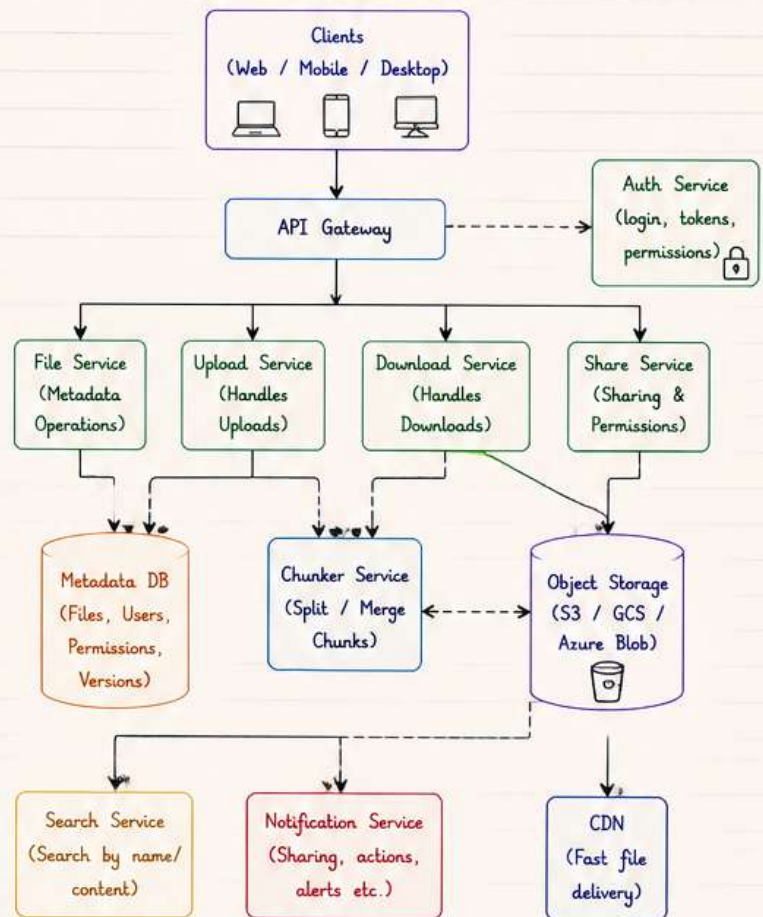
### High Level Design - Key Components

- Client - Web / Mobile / Desktop application.
- API Gateway - Handles requests, auth, throttling.
- Auth Service - Login, authorization, tokens.
- File Service - Handles file metadata operations.
- Storage Service - Stores file chunks in object storage.
- Metadata DB - Stores file info, owners, permissions, versions.
- Chunker Service - Splits large files into chunks.
- Search Service - Search files by name/content.
- Notification Service - File share, actions etc.
- CDN - Fast download of files.

### How it works

1. On upload, file is split into chunks and stored in object storage.
2. Metadata is saved in DB.
3. On download, required chunks are fetched and assembled.
4. Permissions checked before sharing access.
5. Versioning keeps older copies for restore.

### High Level Design Diagram



## 43. Design an Authentication System

### Requirements

- Secure user registration and login.
- Support login with password and OAuth.
- Issue and validate tokens (session/JWT).
- Handle password reset / change.
- Support multi-factor authentication (MFA).
- Scalable and secure.

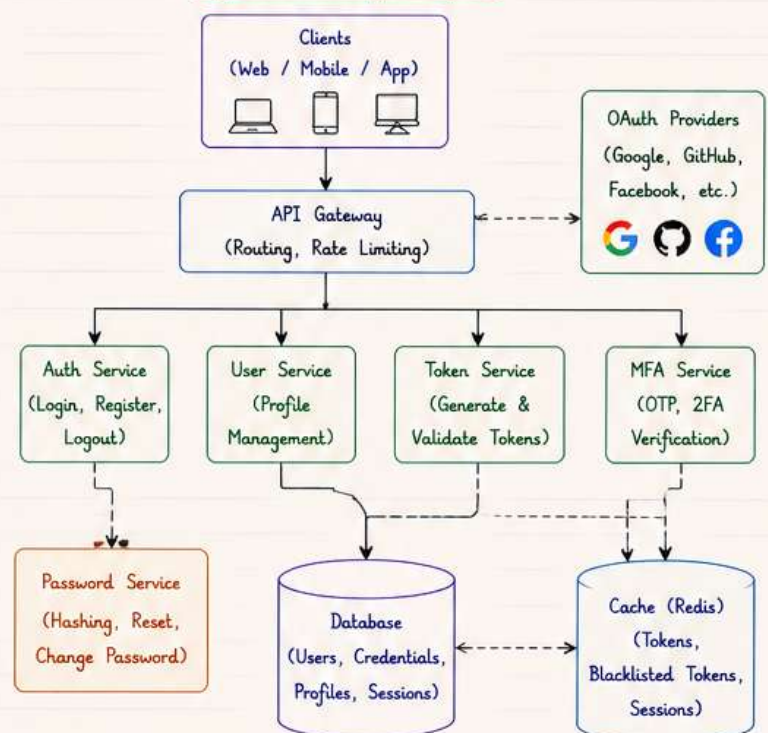
### High Level Design - Key Components

- Client - Web / Mobile / App.
- API Gateway - Routes and secures requests.
- Auth Service - Handles login, register, logout.
- User Service - Manages user profile.
- Token Service - Generates and validates tokens.
- Password Service - Hashing, reset, change.
- MFA Service - Sends OTP / verifies.
- Database - Stores users, credentials, sessions.
- Cache (Redis) - Stores tokens / blacklisted tokens.

### How it works

1. User registers or logs in via client.
2. Auth Service validates credentials.
3. On success, Token Service issues access token (and refresh token).
4. Token is used for authenticated requests.
5. For protected actions, token is validated.
6. Tokens can be revoked / expired.

### High Level Design Diagram



### Common Considerations (Both Designs)

- \* Scalability
- \* High Availability
- \* Security
- \* Monitoring & Logging
- \* Backups & Disaster Recovery
- \* Cost Optimization

#### 42. What are read replicas ? When should you use them ?

- Read replicas are copies of the primary database that replicate data asynchronously.
  - They help in scaling read operations and improving system performance.
  - Data is written to the primary DB and then replicated to replicas.
  - Replicas handle read queries, reducing load on the primary DB.
  - Use Cases :
    - ① Read-heavy applications
    - ② Reporting and analytics
    - ③ Global distribution (read from nearest region)
    - ④ High availability and disaster recovery
- 

#### 43. What is service discovery in microservices ?

- Service discovery is the mechanism by which services find and communicate with each other.
  - In microservices, instances can change (scale up/down), so hardcoding not possible.
  - Services register themselves with a service registry.
  - Other services query the registry to find available instances.
  - Common Tools : Eureka, Consul, Zookeeper, Kubernetes DNS.
  - Benefits : Dynamic scaling, load balancing, fault tolerance.
- 

#### 44. What are WebSockets ? How are they different from HTTP ?

- WebSockets provide a persistent, full-duplex communication channel over a single TCP connection.
- They enable real-time, low-latency data exchange between client and server.
- Use Cases : Chat applications, live notifications, online gaming, stock updates.
- Differences from HTTP :

| HTTP                                                                                                                                                                                   | WebSockets                                                                                                                                                                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Connection is short-lived</li><li>• Request-Response model</li><li>• Each request creates a new connection</li><li>• Higher overhead</li></ul> | <ul style="list-style-type: none"><li>• Connection is long-lived</li><li>• Full-Duplex (bi-directional)</li><li>• Single connection reused</li><li>• Lower latency and overhead</li></ul> |

---

#### 45. How do retries, exponential backoff and timeouts improve system reliability ?

- Retries : If a request fails due to temporary issues, retrying increases chances of success.
  - Exponential Backoff : Wait time between retries increases exponentially (1s, 2s, 4s, 8s...).
  - Timeouts : Prevent hanging requests; if a service doesn't respond in time, fail fast.
  - Jitter : Add random delay to avoid retry storms when many clients retry at once.
  - These techniques improve resiliency and reduce failure impact.
- 

#### 46. What is observability ? Explain Logs, Metrics and Traces.

- Observability is the ability to understand the internal state of a system based on its outputs.
  - It helps in monitoring, debugging and improving system performance.
    - ① Logs : Discrete events with timestamp. Useful for debugging issues.
    - ② Metrics : Numerical measurements over time (CPU usage, request count, latency).
    - ③ Traces : Track a request as it flows across multiple services. Helps in root cause analysis.
  - Together, they provide visibility into system health and performance.
- 
-

#### 47. What are hot partitions and how can you avoid them?

- Hot partitions occur when a particular partition receives much more traffic than others.
  - This causes uneven load, high latency and can even lead to failures.
  - Common in systems using sharding or partitioning (e.g., databases, Kafka, Cassandra).
  - How to avoid :
    - ① Choose a good partition key that distributes data evenly.
    - ② Use hashing or consistent hashing.
    - ③ Add randomness / salting to the key if necessary.
    - ④ Monitor partition metrics and rebalance if needed.
- 

#### 48. What is data compression and why is it used?

- Data compression reduces the size of data by removing redundancy.
  - It can be lossless (all data recovered) or lossy (some data lost for higher compression).
  - Why it is used :
    - ① Reduce storage space.
    - ② Reduce network bandwidth and transmission time.
    - ③ Improve performance (faster transfers).
    - ④ Lower cost for storage and network usage.
- 

#### 49. What is rate limiting vs throttling?

- Both are used to control the number of requests, but they are different.
- Rate Limiting : Restricts the number of requests a client can make in a given time window.  
(e.g., 100 requests per minute)
- Throttling : Reduces the rate of requests when the system is under heavy load to maintain stability.
- Techniques used :

##### Rate Limiting Algorithms

- ① Token Bucket
- ② Leaky Bucket
- ③ Fixed Window Counter
- ④ Sliding Window Log

##### Throttling Techniques

- ① Slow down request rate dynamically
  - ② Queue requests
  - ③ Reject excess requests
  - ④ Use backpressure
- 

#### 50. What are Bloom Filters? Where are they used?

- Bloom Filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set.
  - It can return false positives but never false negatives.
  - It uses a bit array and multiple hash functions.
  - Where it is used :
    - ① Caching systems (e.g., check if item may exist before querying DB)
    - ② Databases to reduce disk lookups.
    - ③ Web browsers to check if a URL is malicious.
    - ④ Distributed systems for membership tests.
    - ⑤ Search engines, DNS systems, and spam filters.
- 
-