**15-Day LLD + System Design Roadmap**

**📅 Week 1 → Foundations of LLD (OOP + Principles)**

**Day 1: OOP Mastery**

* Revise OOP in Java (Class, Object, Inheritance, Polymorphism, Encapsulation, Abstraction)
* Code small examples (Bank Account, Shape hierarchy, Vehicle)
* 🎯 Resource: *Head First Java* (OOP sections)

**Day 2: SOLID Principles**

* S → Single Responsibility
* O → Open/Closed
* L → Liskov Substitution
* I → Interface Segregation
* D → Dependency Inversion
* Implement each with **real code examples** (e.g., designing Logger, PaymentProcessor).
* 🎯 Resource: *Clean Code by Robert C. Martin*

**Day 3: Design Principles & Coding Best Practices**

* DRY, KISS, YAGNI principles
* Composition > Inheritance
* Interface vs Abstract class
* Example: Design a Notification Service (Email, SMS, Push).

**Day 4: Creational Design Patterns**

* Singleton, Factory, Builder, Prototype
* Code in Java with UML diagrams
* Example: Connection Pool, Pizza Builder

**Day 5: Structural Design Patterns**

* Adapter, Decorator, Composite, Proxy
* Example: Payment Gateway Adapter (PayPal, Razorpay, Stripe)

**Day 6: Behavioral Design Patterns**

* Strategy, Observer, Command, State, Template
* Example: Strategy for Sorting Algorithms, Observer for Stock Price Alert

**Day 7: LLD Practice Day**

* Implement 2–3 case studies:
  + Parking Lot
  + Library Management System
  + Logging Framework

**📅 Week 2 → LLD Projects + System Design (HLD Basics)**

**Day 8: Interview-Focused LLD Questions**

* Elevator System
* Tic Tac Toe
* Snake & Ladder
* Splitwise
* 🎯 Practice on *Grokking LLD* (takeuforward has free versions too)

**Day 9: More LLD Questions**

* Ride Sharing (Uber/Ola)
* Food Delivery (Zomato/Swiggy)
* ATM Machine
* Cache (LRU, LFU)

**Day 10: HLD Foundations**

* Client-Server Architecture
* APIs (REST vs gRPC)
* Databases (SQL vs NoSQL, Sharding, Replication)
* Caching (Redis, Memcached)
* Load Balancers, Message Queues (Kafka, RabbitMQ)

**Day 11: HLD Case Studies (Basics)**

* URL Shortener
* TinyInstagram (Feed system)
* WhatsApp Messaging Flow

**Day 12: HLD Case Studies (Advanced for Freshers)**

* Design YouTube (Video storage + Streaming basics)
* Design Twitter (Followers, Feed generation)
* Design Netflix (basic high-level, CDN usage)

**Day 13: End-to-End Mini Projects**  
Pick 1–2 and do **LLD + HLD together**:

* Parking Lot
* Food Delivery System
* BookMyShow Ticket Booking

**Day 14: Mock Interviews + Whiteboarding**

* Solve 2–3 LLD + 1 HLD in interview format
* Record yourself explaining UML + decisions

**Day 15: Final Revision + Cheat Sheet Prep**

* Prepare **20 common LLD interview problems**
* Prepare **8 common HLD systems**
* Summarize OOP, SOLID, Patterns in 2–3 pages

**✅ What You’ll Have After 15 Days**

* **Solid OOP + Design Patterns knowledge** (you can explain + code them)
* **10+ LLD case studies implemented** (very high chance of being asked in interviews)
* **5–6 HLD systems explained with tradeoffs** (expected at MNCs)
* **Confidence in coding + explaining UML/architecture diagrams on whiteboard**

**🎯 Extra Resources for Cracking 50 LPA Interviews**

1. **Books**
   * *Head First Design Patterns*
   * *Clean Code*
   * *Designing Data-Intensive Applications (DDIA)* → Just skim major chapters
2. **Courses**
   * Take U Forward LLD/HLD YouTube series (best for freshers)
   * Grokking the System Design (LLD + HLD)
3. **Practice**
   * Do **LLD + DSA combo** → many MNCs (Amazon, Microsoft, Uber) first check DSA, then LLD.
   * 30–40 LPA roles usually = DSA (LeetCode) + LLD + Basics of HLD.

**15-Day LLD + System Design + Interview Concepts Roadmap**

**📅 Week 1 → Core LLD + OOP + Design Patterns**

**Day 1: OOP (Object-Oriented Programming)**

**Concepts to Learn**

* Class, Object, Constructor, Methods
* Inheritance (IS-A), Composition (HAS-A)
* Polymorphism (Overloading vs Overriding)
* Encapsulation, Abstraction
* Access Modifiers

**Interview Qs**

* What is the difference between **Composition and Inheritance**?
* Explain **Overriding vs Overloading** with real examples.
* Why is **Encapsulation important**?

**Coding Practice**: Bank Account System, Shape Hierarchy (Circle, Square).

**Day 2: SOLID Principles**

**Concepts to Learn**

* **S**: Single Responsibility – one class = one job
* **O**: Open/Closed – open for extension, closed for modification
* **L**: Liskov Substitution – subtypes must be usable as parent type
* **I**: Interface Segregation – no “fat” interfaces
* **D**: Dependency Inversion – depend on abstractions, not concretions

**Interview Qs**

* What is the difference between **OCP and ISP**?
* Give a real-world example of **Dependency Inversion**.

**Coding Practice**: Logger System (FileLogger, DBLogger, CloudLogger).

**Day 3: Design Principles**

**Concepts to Learn**

* DRY (Don’t Repeat Yourself)
* KISS (Keep It Simple, Stupid)
* YAGNI (You Aren’t Gonna Need It)
* Composition > Inheritance
* Cohesion vs Coupling

**Interview Qs**

* Why should we prefer **composition over inheritance**?
* What is **tight coupling** vs **loose coupling**?

**Coding Practice**: Notification System (SMS, Email, Push).

**Day 4: Creational Patterns**

**Concepts to Learn**

* **Singleton** (thread-safe version)
* **Factory Pattern** (object creation centralized)
* **Builder Pattern** (complex objects step-by-step)
* **Prototype Pattern** (cloning objects)

**Interview Qs**

* How to implement a **Thread-Safe Singleton**?
* Difference between **Factory vs Builder**?

**Coding Practice**:

* DB Connection Pool (Singleton)
* Pizza Builder (Builder Pattern).

**Day 5: Structural Patterns**

**Concepts to Learn**

* **Adapter** (convert one interface to another)
* **Decorator** (add behavior at runtime)
* **Composite** (tree structure)
* **Proxy** (control access)

**Interview Qs**

* Example of **Adapter Pattern** in Java libraries?
* Difference between **Decorator and Proxy**?

**Coding Practice**:

* Payment Gateway Adapter (PayPal, Stripe).
* Data Compression Decorator.

**Day 6: Behavioral Patterns**

**Concepts to Learn**

* **Strategy** (choose algorithms at runtime)
* **Observer** (publisher-subscriber)
* **Command** (encapsulate request)
* **State** (object behavior changes with state)
* **Template Method**

**Interview Qs**

* How is **Observer used in real life** (e.g., Stock price alert)?
* Difference between **Strategy vs State Pattern**?

**Coding Practice**:

* Sorting Strategy (QuickSort, MergeSort).
* Observer for Weather Alerts.

**Day 7: LLD Practice Day**

* Parking Lot System
* Library Management System
* Elevator System

**Interview Qs**

* How will you design a **Parking Lot** that supports Cars, Bikes, Trucks?
* How do you ensure **thread safety in LLD systems**?

**📅 Week 2 → Advanced LLD + HLD Basics**

**Day 8: LLD Case Studies**

* Splitwise
* Snake & Ladder
* ATM Machine

**Interview Qs**

* How will you design **Splitwise** to handle groups & settlements?
* How will you design an **ATM** that supports multiple banks?

**Day 9: More LLD Case Studies**

* Food Delivery (Zomato/Swiggy)
* Ride Sharing (Uber/Ola)
* Cache (LRU, LFU)

**Interview Qs**

* How would you implement **LRU Cache**?
* What classes are required for **Ride Sharing system**?

**Day 10: HLD Basics**

**Concepts to Learn**

* Client-Server Model
* REST vs gRPC APIs
* SQL vs NoSQL
* Sharding, Replication
* Caching (Redis, Memcached)
* Load Balancing, Message Queues (Kafka, RabbitMQ)

**Interview Qs**

* When to use **SQL vs NoSQL**?
* Why do we need **Message Queues**?

**Day 11: HLD Case Studies (Easy)**

* URL Shortener
* TinyInstagram (Feed system)
* WhatsApp Messaging

**Interview Qs**

* How would you design **URL Shortener like bit.ly**?
* How to design **WhatsApp group messaging**?

**Day 12: HLD Case Studies (Advanced)**

* Twitter (Followers + Timeline)
* YouTube (Video storage + streaming basics)
* Netflix (basic CDN usage)

**Interview Qs**

* How would you design **Twitter feed**?
* How does **Netflix stream videos worldwide**?

**Day 13: End-to-End System Design**

Pick 2–3 and do **LLD + HLD together**:

* Parking Lot + Scaling with multiple lots
* Food Delivery (LLD classes + HLD caching, DB choice)
* BookMyShow Ticket Booking (Concurrency Handling)

**Interview Qs**

* How to handle **double booking in BookMyShow**?
* Which DB will you use for **Food Delivery orders**?

**Day 14: Mock Interviews**

* Solve 2–3 LLD + 1 HLD **with UML + explanations**
* Record yourself and review clarity

**Day 15: Revision & Cheat Sheet**

* 20 Common **LLD Problems** (Parking Lot, Splitwise, Elevator, ATM, Ride Sharing, Cache, Snake & Ladder, etc.)
* 8 Common **HLD Problems** (URL Shortener, Twitter, YouTube, WhatsApp, Food Delivery, BookMyShow, etc.)
* 2–3 Pages of **OOP, SOLID, Patterns Theory**

**🎯 Key Takeaway for Interviews**

✅ **For Fresher (30–50 LPA)**:

* Round 1 → DSA (LeetCode medium/hard)
* Round 2 → LLD (Parking Lot, Splitwise, Cache, Food Delivery)
* Round 3 → HLD basics (URL Shortener, WhatsApp, Twitter feed)
* HR → Communication + Projects

**🔑 What a Good System Design Course Covers:**

✅ **Foundations (Week 1–2)**

* Client-server model, HTTP, DNS, CDN, load balancing
* Caching (Redis, Memcached)
* Databases (SQL vs NoSQL, indexing, sharding, replication)
* Message queues (Kafka, RabbitMQ)

✅ **Core Design Patterns (Week 3–4)**

* Consistent hashing
* Rate limiting (Token Bucket, Leaky Bucket)
* Leader election & consensus (Paxos, Raft, Zookeeper)
* CAP theorem, PACELC theorem

✅ **Case Studies (Week 5–6)**

* Design **URL shortener** (TinyURL)
* Design **Instagram / WhatsApp**
* Design **Uber / Swiggy / Flipkart**
* Design **YouTube / Netflix Recommendation System**
* Design **Distributed Cache (like Redis)**

✅ **Advanced Topics (Week 7–8)**

* Microservices vs Monoliths
* Event-driven architecture
* Observability: logging, monitoring, tracing
* Cloud-native design (AWS, GCP, Kubernetes)
* Real-time streaming (Kafka, Flink, Spark)

**🎯 Outcomes of a System Design Course:**

* You’ll be able to **crack interviews** at Amazon, Microsoft, Uber, etc.
* You’ll know how to **scale an app from 1k → 100M users**.
* You’ll understand **trade-offs** (SQL vs NoSQL, consistency vs availability).
* You’ll build **intuition for designing production systems**.

👉 I can give you:

1. **A structured roadmap (30–60 days)** to master System Design.
2. **Best courses/playlists** (paid & free) for both interview prep and real-world system design.
3. **High-quality case studies with diagrams + explanations**.

Do you want me to make you a **System Design learning roadmap (30/60 days)** with courses + practice problems, tailored for **interviews (MNC 50 LPA)** or for **practical backend dev**?

**Start**

**1️⃣ What is a System?**

A **system** is a collection of **components** that work together to achieve a goal.

🔹 Key properties:

* Input → Processing → Output
* Multiple components interacting (hardware, software, users, data)
* Has **boundaries** and **purpose**

📌 **Examples of Systems**

* **Computer System**: Hardware (CPU, RAM, Disk) + Software (OS, Apps) → gives output like running programs.
* **Transport System**: Roads, vehicles, traffic signals → move people from A to B.
* **E-commerce System (Amazon/Flipkart)**: Users + Website + Database + Payment gateway → lets you buy products online.

👉 In tech interviews, “system” usually means **software + infrastructure working together** (like Twitter, YouTube, Uber).

**2️⃣ What is Design?**

**Design** is the **process of planning and structuring** components to solve a problem effectively.

🔹 Key properties:

* Looks at **requirements** (what the system must do)
* Considers **trade-offs** (speed vs cost, availability vs consistency)
* Aims for **clarity, efficiency, scalability**

📌 **Examples of Design**

* **Bridge Design**: Engineers decide material, length, load capacity before construction.
* **House Design**: Architect plans rooms, ventilation, structure before building.
* **Database Design**: Choosing schema (tables, indexes, relationships) to handle data effectively.

👉 In software, “design” means **how different parts interact** (API, database, cache, load balancer, etc.).

**3️⃣ What is System Design?**

**System Design = Designing the architecture of a system to meet specific requirements.**

It’s about breaking down a **big problem** (like building YouTube) into **smaller components** (video upload, storage, streaming, recommendations, scaling) and designing **how they interact**.

🔹 Key properties:

* Defines **architecture** (frontend, backend, DB, infra)
* Ensures **scalability, reliability, availability, maintainability**
* Involves **trade-offs** (SQL vs NoSQL, cache vs DB, consistency vs latency)

📌 **Examples of System Design**

1. **URL Shortener (TinyURL / Bitly)**
   * Requirement: Convert a long URL into a short one.
   * Design:
     + API server → takes long URL, generates hash code.
     + Database → stores mapping (short → long).
     + Redirect service → fetches original URL when short is clicked.
2. **WhatsApp**
   * Requirement: Send/receive messages instantly.
   * Design:
     + Client app → sends message request.
     + Server → stores message in DB.
     + Push service → delivers message to recipient.
     + Use queues for reliability + caching for speed.
3. **YouTube**
   * Requirement: Upload, store, and stream videos worldwide.
   * Design:
     + Upload service → receives video.
     + Storage (distributed file system, like Google File System).
     + Transcoding service → converts video to different resolutions.
     + CDN → delivers videos fast globally.
     + Recommendation system → suggests videos using ML.

**Summary in Simple Words**

* ***System*** *= Group of components working together (e.g., WhatsApp, a car engine).*
* ***Design*** *= Plan/blueprint of how to build something (e.g., house plan, DB schema).*
* **System Design** = Blueprint of a big system (e.g., how YouTube stores & streams videos, how Amazon handles millions of users).

**What are Components in System Design?**

In **System Design**, a **component** is an individual building block (piece) of the system that performs a specific function.

* Each component has a **clear responsibility**.
* Multiple components interact → to form the complete system.
* Think of components as **LEGO blocks** in a big system.

Example: In **WhatsApp**,

* **Client App** = one component
* **Message Server** = another component
* **Database** = another component  
  Together, they form the messaging system.

**Two Main Types of Components in System Design**

**1. Functional Components (Logical Components)**

👉 These are about **what the system does** (business logic).

* Handle **features & functionality**.
* Usually implemented in **application layer**.
* Focus is on **requirements** (e.g., login, payments, messaging).

📌 Examples:

* **Authentication Service** → login/signup
* **Recommendation Engine** → suggests YouTube videos
* **Payment Service** → processes transactions
* **Search Service** → finds items in Flipkart

**2. Infrastructure Components (Physical / Supporting Components)**

👉 These are about **how the system runs** (performance, scalability, reliability).

* Handle **storage, communication, scalability, fault tolerance**.
* Usually sit in **infrastructure & networking layer**.
* Focus is on **system qualities** (speed, scale, security).

📌 Examples:

* **Database** (SQL, NoSQL, sharding, replication)
* **Cache** (Redis, Memcached) → speedup
* **Load Balancer** → distributes traffic
* **Message Queue** (Kafka, RabbitMQ) → async processing
* **CDN** (Content Delivery Network) → fast global delivery
* **Monitoring & Logging** (Prometheus, ELK)

**📝 Analogy**

* **Functional components** = What a restaurant offers (menu: food, drinks).
* **Infrastructure components** = How it delivers service (kitchen, staff, electricity, tables).

Both together make the **system (restaurant)** work.

✅ Summary:

* **Components** = building blocks of system design.
* **Type 1: Functional** → Features (auth, search, payments).
* **Type 2: Infrastructure** → Support (DB, cache, load balancer).

**🔑 Entities in System Design Components**

When designing systems, we often divide components into **two entity categories**:

**1. Logistic Entities (*Business* / Domain Entities)**

👉 These represent the **real-world business objects** and **process flow** the system needs to support.

* They define **what the system must do** (business requirements).
* Closely tied to the **domain** (e.g., e-commerce, transport, healthcare).

Examples in different systems:

* **E-commerce (Amazon/Flipkart)**: Product, Order, Cart, Payment, User
* **Logistics (Uber/Swiggy)**: Rider, Driver, Trip, Delivery Slot, Route
* **Banking System**: Account, Transaction, Customer, Loan

🔹 These are like the **“nouns”** in the problem statement.  
(“User places an order → system processes payment → order is delivered”).

**2. Technology Entities (Technical Components)**

👉 These represent the **infrastructure + technical implementation** used to support the logistics.

* They define **how the system will work**.
* Tied to **software/hardware solutions**.

📌 Examples in different systems:

* **Databases** → SQL (Postgres, MySQL), NoSQL (MongoDB, Cassandra)
* **Caching systems** → Redis, Memcached
* **Load Balancers** → Nginx, HAProxy
* **Message Queues** → Kafka, RabbitMQ, SQS
* **CDN (Content Delivery Network)** → Cloudflare, Akamai
* **Monitoring & Logging** → Prometheus, Grafana, ELK stack

🔹 These are like the **“verbs + tools”** that bring the system to life.  
(“Store order in DB → use Redis cache → notify delivery agent via Kafka queue”).

Think of **Flipkart** as an example:

* **Logistic Entities**:
  + User, Cart, Product, Order, Payment, Delivery Partner.
* **Technology Entities**:
  + Database (Postgres for orders, MongoDB for product catalog).
  + Cache (Redis for hot products).
  + Message Queue (Kafka for delivery updates).
  + CDN (Akamai for images).
  + Load Balancer (Nginx for scaling).
* Logistic entities = **what exists in the business**.
* Technology entities = **what tech supports it**.

✅ **Summary:**

* **Logistic Entities** = Business objects & workflow (user, order, delivery, trip).
* **Technology Entities** = Technical infrastructure (DB, cache, load balancer, queue).
* Both are needed in system design → logistic tells us **what** to design, technology tells us **how** to implement.

**What is a Communication Protocol?**

A **communication protocol** is a set of **rules and standards** that define **how data is exchanged** between two or more systems, devices, or software components.

Think of it like a **common language** so that both sides can **understand each other**.

* Without protocols → systems can’t talk.
* With protocols → we ensure **format, timing, error handling, security**.

**🟢 3 Examples of Communication Protocols**

**1. HTTP / HTTPS (Hypertext Transfer Protocol)**

* **Used for:** Communication between web browsers and servers.
* **How it works:**
  + Client (browser) sends a request → “GET /product/123”.
  + Server responds → “200 OK + product details in JSON/HTML”.
* **Why important:**
  + Basis of the **web** (Google, Flipkart, YouTube).
  + HTTPS = Secure (encrypted with SSL/TLS).

📌 Example:  
When you open [**www.amazon.in**](http://www.amazon.in/), your browser uses **HTTPS** to request the page from Amazon’s server.

**2. TCP/IP (Transmission Control Protocol / Internet Protocol)**

* **Used for:** Reliable communication over the internet.
* **How it works:**
  + TCP splits data into packets, ensures delivery, reorders if needed.
  + IP handles addressing → “send packet to this IP address”.
* **Why important:**
  + Basis of all **internet communication**.
  + Used in WhatsApp messages, emails, file transfers.

📌 Example:  
When sending a WhatsApp message, TCP ensures the entire message is delivered **without loss or duplication**.

**3. WebSocket Protocol**

* **Used for:** Real-time, two-way communication between client and server.
* **How it works:**
  + Unlike HTTP (request → response), WebSocket allows **continuous open connection**.
  + Both client & server can send messages anytime.
* **Why important:**
  + Needed for **chat apps, live updates, stock trading platforms, multiplayer games**.

📌 Example:  
In **Swiggy live order tracking**, WebSocket is used to push real-time updates of your delivery partner’s location.

**⚖️ Analogy**

* **HTTP/HTTPS** → Like a postal service: you send a letter, wait for reply.
* **TCP/IP** → Like a courier with tracking: ensures your package arrives, in order.
* **WebSocket** → Like a phone call: continuous two-way communication.

**Summary**

* A **communication protocol** = agreed rules for exchanging data.
* **Example 1: HTTP/HTTPS** → Web browsing (request/response).
* **Example 2: TCP/IP** → Internet backbone (reliable packet transfer).
* **Example 3: WebSocket** → Real-time communication (chat, tracking).

**🔑 What is Client-Server Architecture?**

**Client-Server Architecture** is a design model where:

* **Client** = Requests a service (*frontend*, mobile app, browser).
* **Server** = Response a service (*backend*, database, API).
* Communication happens via a **network** (like the internet).

👉 In simple words:

* **Client asks, Server answers.**
* They work together but have **different roles**.

**🟢 How it Works (Step-by-Step)**

1. Client (e.g., your browser) sends a request → "GET /homepage".
2. Server (e.g., Amazon backend) processes it → fetches data from DB.
3. Server responds → HTML/JSON back to the client.
4. Client displays it to the user.

**📌 Examples of Client-Server Architecture**

**1. Web Browsing (HTTP/HTTPS)**

* **Client** → Chrome/Edge browser
* **Server** → Web server (Apache, Nginx, Node.js backend)
* **Process**:
  + You type [*www.flipkart.com*](http://www.flipkart.com/).
  + Browser (client) requests page.
  + Flipkart server sends HTML/CSS/JS to browser.
  + Browser renders site for you.

**2. WhatsApp Messaging**

* **Client** → WhatsApp app on your phone
* **Server** → WhatsApp message server
* **Process**:
  + You send "Hello" to a friend.
  + Client app sends request → server stores message → forwards to recipient’s device.
  + Receiver’s app gets the message from server.

**3. Online Banking System**

* **Client** → Mobile banking app
* **Server** → Bank’s backend + database
* **Process**:
  + You request to transfer ₹500.
  + Server verifies balance, updates database, processes payment.
  + Server replies with confirmation → client shows “Transaction Successful ✅”.

**⚖️ Analogy**

* **Client** = Customer in a restaurant (places order).
* **Server** = Kitchen (prepares and delivers food).
* ***Waiter (protocol like HTTP/TCP)*** *= Communication medium.*

**Summary**

* **Client-Server Architecture** = A model where client requests and server responds.
* **Client** → Frontend (browser, app).
* **Server** → Backend (database, APIs).
* Examples: Web browsing, WhatsApp, Online banking.
* Basis of **modern internet applications**.

**1. What is a client?**

* A **client** is a device or software that **requests services or resources** from a server.
* Usually **frontend-facing** (used by the end user).
* Can be a **web browser, mobile app, desktop app, IoT device**.

📌 Examples:

* Chrome browser requesting Amazon’s homepage.
* WhatsApp app sending a message.
* ATM machine requesting balance from the bank server.

**2. What is a Server?**

* A **server** is a powerful computer/software that **provides services/resources** to clients.
* Always **on, listening, and responding**.
* Can be a **web server, application server, database server, file server**.

📌 Examples:

* *Amazon backend server sending product data.*
* WhatsApp server forwarding messages.
* Bank database server processing transactions.

**🔑 3. Difference between Client and Server**

| **Feature** | **Client** | **Server** |
| --- | --- | --- |
| **Role** | Requests service | Provides service |
| **Who uses it?** | End-user interacts | Runs in background, no direct user |
| **Examples** | Browser, Mobile app, ATM | Web server, DB server, Mail server |
| **Hardware Needs** | Lightweight (PC, phone) | High performance (powerful CPU, RAM, storage) |
| **Initiates?** | Always initiates request | Always waits for request |

👉 In short:

* **Client 🡪 asks**, **Server 🡪 answers**.

**4. Types of Client-Server Architectures**

**🟢 A) 2-Tier Client-Server Architecture**

* **Layers**: Client + Server (only 2 tiers).
* Client directly communicates with the database.

📌 Example:

* A desktop app directly connected to a database.
* ATM machine → Bank Database.

**Pros:** Simple, fast.  
**Cons:** Not scalable, tight coupling, less secure.

**🟢 B) 3-Tier Client-Server Architecture**

* **Layers**: Client + Application Server + Database Server.
* Client talks to **application server** (business logic), which talks to **database server**.

📌 Example:

* Flipkart website:
  + Client (Browser/Mobile App) → Application Server (handles cart, orders, payments) → Database Server (stores users, products).

**Pros:** Better scalability, separation of concerns, more secure.  
**Cons:** More complex than 2-tier.

**🟢 C) N-Tier (Multi-Tier) Client-Server Architecture**

* **Layers**: Client + Multiple middle layers + Database.
* More than 3 layers → often includes **load balancer, cache, API gateways, microservices, analytics layer**.

📌 Example: Netflix System

1. Client (mobile/web app).
2. API Gateway → routes requests.
3. Microservices (auth, video service, recommendations).
4. Cache layer (Redis, CDN).
5. Database (user DB, video DB).

**Pros:**

* Very scalable and flexible.
* Fault isolation (failure in one service doesn’t crash all).
* Used in **real-world large systems** (Amazon, Uber, Netflix).

**Cons:**

* Complex to design & maintain.
* Higher cost.

**⚖️ Analogy**

* **2-Tier** = You (client) directly calling your bank manager (server).
* **3-Tier** = You (client) → receptionist (app server) → manager (DB server).
* **N-Tier** = You (client) → receptionist → clerk → finance team → bank database.

✅ **Summary:**

* **Client** = Requests service.
* **Server** = Provides service.
* **Difference** = Client initiates, server responds.
* **2-Tier** = Client ↔ Database directly.
* **3-Tier** = Client ↔ Application Server ↔ Database.
* **N-Tier** = Client ↔ Multiple layers (API Gateway, Microservices, Cache, DB).

**2-Tier Client-Server Architecture**

👉 **Definition**: Client communicates **directly** with the database. No middle layer.

📌 **Examples**:

* **ATM Machine** → Directly talks to **Bank Database** for balance & transactions.
* **Desktop Applications** like MS Access → Directly connects to SQL Server.
* **Simple Inventory Management System** → Client app installed on PC directly queries the central DB.

**Flow**:  
Client (ATM / App) ↔ Database Server

**3-Tier Client-Server Architecture**

👉 **Definition**: Client communicates with an **Application Server**, which then communicates with the **Database**.

📌 **Examples**:

* **Flipkart / Amazon Website**:
  + Client (browser/app) → Application Server (business logic, product search, order handling) → Database Server (products, users, orders).
* **Online Banking Website**:
  + Client (mobile/web app) → App Server (validates user, processes transaction rules) → Database (stores accounts, balances).
* **University Portal**:
  + Student (client) → Application Server (handles login, course registration) → Database (stores students, results).

**Flow**:  
Client ↔ Application Server ↔ Database

**🟢 3. N-Tier (Multi-Tier) Client-Server Architecture**

👉 **Definition**: More than 3 layers, with additional services like **load balancers, microservices, cache, API gateway, CDN**.

📌 **Examples**:

* **Netflix / YouTube**:
  + **Client** (app/website)
  + *Load Balancer*
  + *API Gateway*
  + *Microservices (auth, video streaming, recommendations, payments)*
  + *Cache (Redis, CDN for video delivery)*
  + **Database (user DB, video metadata, analytics DB)**
* **Uber / Swiggy**:
  + Client (app)
  + API Gateway
  + Services (ride matching, driver tracking, payments, notifications)
  + Cache (Redis for active drivers, surge pricing)
  + Database (user data, trip history)
* **E-commerce Giant (Amazon)**:
  + Client (browser/app)
  + CDN (images, static files)
  + API Gateway
  + Services (product catalog, cart, payment, order, shipping)
  + Queue (Kafka for order updates)
  + Database (RDS for orders, DynamoDB for cart, Elasticsearch for search)

**Flow**:  
Client ↔ CDN/Load Balancer ↔ API Gateway ↔ Microservices ↔ Cache ↔ Database

**⚖️ Summary with Quick Examples**

* **2-Tier** → ATM ↔ Bank Database, MS Access ↔ SQL Server
* **3-Tier** → Flipkart (Client ↔ App Server ↔ DB), Online Banking, University Portal
* **N-Tier** → Netflix, YouTube, Uber, Swiggy, Amazon (with microservices + cache + CDN)

**🔑 What is a Client?**

A **client** is the part of the system that interacts with the end user and requests services from the server.  
But depending on **how much processing the client does itself**, we classify it as **Thin Client** or **Thick Client**.

**🟢 1. Thin Client**

👉 A **thin client** does **very little processing** itself.

* Most of the work (processing, storage, logic) is done on the **server**.
* Client only handles **display + input/output**.

📌 Examples:

* **Web Browser** (Chrome, Firefox):
  + Just sends requests to server, displays HTML/JS response.
  + All business logic is processed on the server.
* **Google Docs (Web Version)**:
  + Most processing/storage on Google’s servers.
* **ATM Machine**:
  + Just sends transaction requests, server handles validation & balance updates.

✅ Advantages:

* Simple, lightweight, cheaper.
* Easy to maintain (since logic lives on server).  
  ❌ Disadvantages:
* Depends heavily on server.
* Needs continuous network connection.

**2. Thick Client (Fat Client / Rich Client)**

👉 A **thick client** does **most processing itself**.

* It contains more **business logic, processing power, and sometimes storage**.
* Server is used mainly for data exchange or backup.

📌 Examples:

* **MS Outlook (Desktop App)**:
  + Stores emails locally, processes them, syncs with mail server.
* **Video Games**:
  + Game client does rendering, logic → server just handles multiplayer sync.
* **Zoom / Microsoft Teams App**:
  + Client handles audio/video encoding, while server manages routing.

✅ Advantages:

* Less server load.
* Works even with limited connectivity (offline mode).  
  ❌ Disadvantages:
* More complex to maintain/update (each client needs software updates).
* Higher hardware requirements on client side.

**⚖️ Thin vs Thick Client Comparison**

| **Feature** | **Thin Client** | **Thick Client** |
| --- | --- | --- |
| **Processing** | Done mostly on **server** | Done mostly on **client** |
| **Client Hardware** | Lightweight (low CPU/RAM) | Powerful (needs high CPU/RAM) |
| **Dependency** | High server dependency | Less server dependency |
| **Examples** | Web browser, ATM, Google Docs | MS Outlook, Zoom, Video Games |
| **Offline Support** | No (needs server always) | Yes (can work offline sometimes) |

**📝 Analogy**

* **Thin Client** = Student attending class online → teacher (server) explains everything, student just listens.
* **Thick Client** = Student studying with full textbooks → does most work themselves, asks teacher only for doubts.

✅ **Summary:**

* **Thin Client** → Lightweight, relies on server (Browser, ATM, Google Docs).
* **Thick Client** → Heavy, processes itself (MS Outlook, Zoom, Games).

**🔹 What is a Proxy?**

A **proxy** is an **intermediate server** that sits between a client and the internet (or another server).

* It acts as a **middleman**, forwarding requests and responses.
* Clients don’t talk **directly** to the destination server → instead, they talk to the proxy, which talks to the server.

**🔹 Types of Proxies in System Design**

**1️⃣ Forward Proxy**

👉 A forward proxy sits **in front of the client** and forwards client requests to servers.

* The **server doesn’t know** who the real client is.
* Often used for **privacy, filtering, or caching**.

**✅ Pros:**

* Hides client’s identity (IP masking).
* Can enforce access policies (block sites, control browsing).
* Improves performance with caching.

**❌ Cons:**

* Adds latency (extra hop).
* Single point of failure.
* Needs configuration on client side.

**📌 Use Cases:**

* **Corporate networks** blocking social media sites.
* **Schools/universities** controlling which sites students can access.
* **Anonymous browsing** (VPNs are advanced forward proxies).

**💡 Example:**

* A company employee wants to visit YouTube. The request first goes to the **forward proxy** → proxy checks if allowed → if yes, it fetches from YouTube → returns to employee.

**2️⃣ Reverse Proxy**

👉 A reverse proxy sits **in front of servers** and forwards requests from clients to backend servers.

* The **client doesn’t know** which server handled the request.
* Often used for **load balancing, security, and caching**.

**✅ Pros:**

* Protects backend servers (IP hidden).
* Distributes load across multiple servers.
* Can serve cached content to reduce server load.
* Enables SSL termination (offload HTTPS encryption).

**❌ Cons:**

* Adds complexity to system design.
* Single point of failure (unless multiple reverse proxies).

**📌 Use Cases:**

* **Load Balancer** (NGINX, HAProxy, AWS Elastic Load Balancer).
* **CDN (Content Delivery Network)** like Cloudflare/Akamai.
* **Security Gateways** filtering DDoS or malicious requests.

**💡 Example:**

* You open **Netflix.com** → request goes to **Cloudflare (reverse proxy)** → Cloudflare decides which Netflix server to send it to → response comes back via Cloudflare.

**🔹 Forward vs Reverse Proxy (Quick Difference)**

| **Feature** | **Forward Proxy 🧑➡️🌍** | **Reverse Proxy 🌍➡️🖥️** |
| --- | --- | --- |
| Sits in front of | Client | Server |
| Hides identity of | Client | Server |
| Controlled by | Client/user | Server provider |
| Common use cases | Anonymity, access control, censorship bypass | Load balancing, caching, DDoS protection |
| Example | VPN, Squid Proxy | NGINX, Cloudflare, AWS ELB |

**✅ Interview-ready Summary:**

* **Proxy** = Middle server between client & server.
* **Forward Proxy** = Hides **client identity**; used for **privacy & filtering**.
* **Reverse Proxy** = Hides **server identity**; used for **load balancing, caching, security**.

**🔹 1. What is Data?**

* **Definition**: Data is raw information that the system processes, stores, and transmits.
* **Forms of Data**: Text, numbers, images, logs, metrics, videos, etc.
* **Example**:
  + In **Instagram**: Usernames, passwords, images, likes, comments are data.
  + In **Uber**: Pickup location, drop location, ride cost, driver details.

**🔹 2. What is Data Flow?**

* **Definition**: Data flow describes how data moves through the system — from input to processing to storage to output.
* **Steps in Data Flow**:
  1. **Input** → User enters data (form, click, sensor reading, API request).
  2. **Processing** → Application processes the request (validation, transformation, business logic).
  3. **Storage** → Data saved in databases, cache, or message queues.
  4. **Output** → Data sent back to client or another system.
* **Example**:
  1. In **Amazon Order Placement**:
     1. User → clicks “Buy” (input).
     2. System → checks stock, payment (processing).
     3. DB → stores order details (storage).
     4. User → gets confirmation email + delivery tracking (output).

**🔹 3. What is Data Representation?**

* **Definition**: How data is structured and formatted for communication and storage.
* **Common Representations**:
  + **JSON** → { "name": "Alice", "age": 25 }
  + **XML** → <user><name>Alice</name><age>25</age></user>
  + **Binary** → efficient but unreadable to humans (e.g., Protobuf, Avro).
* **Example**:
  + **REST API** → uses JSON.
  + **IoT Sensors** → often send binary data for efficiency.

**🔹 4. Layers in Data Flow (OSI-inspired in System Design)**

Think of data traveling through **layers**:

1. **Presentation Layer** → How data is shown (UI, API response).
   * Example: Mobile app showing “₹500 deducted.”
2. **Application Layer** → Business logic.
   * Example: Deduct wallet, update order, send receipt.
3. **Transport Layer** → Ensures reliable delivery.
   * Example: TCP ensures API request isn’t lost.
4. **Storage Layer** → Persists data.
   * Example: SQL/NoSQL DB saves transaction.
5. **Communication Layer** → Defines format/protocol.
   * Example: HTTP, gRPC, WebSocket.

**🔹 5. Data Stores (Where Data Lives)**

In system design, data stores are building blocks.

**1. Database**

* SQL (Postgres, MySQL) → relational, strong consistency.
* NoSQL (MongoDB, Cassandra) → scalable, flexible schema.
* Example: Amazon stores orders in SQL, product catalog in NoSQL.

**2. Cache**

* Stores frequently accessed data in memory (Redis, Memcached).
* Very fast (ms response).
* Example: Facebook profile pictures in cache → faster load.

**3. Message Queue**

* Stores messages temporarily to ensure asynchronous communication.
* Examples: Kafka, RabbitMQ, SQS.
* Example: Uber → ride request pushed into queue → drivers notified.

**4. Indexes**

* Data structures (B-trees, inverted indexes) to speed up queries.
* Example: Google Search → inverted index of words to pages.

**🔹 6. Data Flow Mechanisms**

How data **moves** between stores, services, and clients.

1. **APIs (Request/Response)**
   * Synchronous → client waits for reply.
   * Example: Weather app fetches “current temperature” via REST API.
2. **Messages (Queue-based)**
   * Asynchronous → sender doesn’t wait.
   * Example: Payment service sends “payment successful” message to order service.
3. **Events (Pub/Sub)**
   * Broadcast → multiple services subscribe.
   * Example: In YouTube → “New video uploaded” event triggers → Notifications service, Recommendation service, Feed service.

**🔹 7. Interaction Example (End-to-End)**

📌 **Example: Placing a Food Order in Swiggy**

1. **User → API Request**
   * Client sends order via REST API (JSON).
2. **API Gateway → Service Layer**
   * Order Service receives request.
3. **Service → Data Stores**
   * Order Service writes order into SQL DB.
   * Cache used for quick menu lookup.
4. **Service → Queue/Event**
   * Publishes “New Order Placed” event to Kafka.
5. **Other Services Consume Event**
   * Delivery Service finds drivers.
   * Payment Service processes payment.
   * Notification Service sends confirmation.
6. **Client Receives Response**
   * User app shows “Order Confirmed.”

✅ This shows **data flow + data representation + data stores + communication** in action.

**🔹 8. Pros & Cons of Different Data Flows**

| **Mechanism** | **Pros** | **Cons** | **Example** |
| --- | --- | --- | --- |
| **APIs** | Simple, real-time | Tight coupling | REST, gRPC |
| **Messages** | Reliable async | Latency | Kafka, SQS |
| **Events** | Scalable, decoupled | Debugging hard | Pub/Sub, Kafka Streams |

👉 So in **System Design**, you need to **pick the right combination**:

* Cache for speed,
* DB for persistence,
* Queue for async reliability,
* Events for scalability.

**1. SQL Databases (Relational Databases)**

➡️ Store data in **tables (rows and columns)** with a fixed **schema**.  
➡️ Based on **Relational Model** (relations = tables).  
➡️ Follows **ACID properties** for strong consistency.

**🔑 Key Concepts**

* **Schema:** Defines table structure (columns, data types, constraints).
* **ACID properties:**
  + **A (Atomicity):** All operations succeed or none.
  + **C (Consistency):** Always valid data after transaction.
  + **I (Isolation):** Transactions don’t interfere.
  + **D (Durability):** Data persists after commit, even if system crashes.

**✅ Advantages:**

* Strong **consistency**.
* Supports **complex queries** (joins, aggregations).
* Mature ecosystem (MySQL, PostgreSQL, Oracle, MS SQL).
* Good for structured, predictable data.

**❌ Disadvantages:**

* **Scalability** is harder (vertical scaling preferred).
* Schema is **rigid** (difficult to handle changing data).
* Can be **slow for very large datasets** with frequent writes.

**📌 Examples:**

* Banking systems (transactions must be ACID).
* E-commerce inventory.

**2. NoSQL Databases (Non-Relational)**

➡️ Designed for **scalability, flexibility, high performance**.  
➡️ Types: **Key-Value, Document, Column, Graph**.

**2.1. Key-Value Databases**

➡️ Data stored as **key → value** pairs.  
➡️ No schema. Very fast lookups.

**✅ Advantages:**

* Extremely **fast** (O(1) lookup).
* Great for **caching, sessions**.
* Easy to scale horizontally.

**❌ Disadvantages:**

* No **relationships** (no joins).
* Limited querying capabilities (can only get by key).

**📌 Examples:**

* **Redis, DynamoDB, Riak**.
* Use case: Storing **user sessions, caching frequently accessed data**.

**2.2. Document Databases**

➡️ Store **semi-structured data** (JSON, BSON, XML).  
➡️ Each document = flexible schema.

**✅ Advantages:**

* Schema-less, flexible.
* Supports **nested data**.
* Scales horizontally.
* Queries are easier compared to key-value stores.

**❌ Disadvantages:**

* No strong ACID guarantee (usually **eventual consistency**).
* Complex joins are inefficient.

**📌 Examples:**

* **MongoDB, CouchDB, Firestore**.
* Use case: **Product catalogs, content management systems, user profiles**.

**2.3. Column-Oriented Databases**

➡️ Store data **by columns instead of rows**.  
➡️ Good for **analytical queries** (aggregate on large datasets).

**✅ Advantages:**

* Very efficient for **OLAP** (analytical processing).
* Can scan columns faster than rows.
* High compression rates.

**❌ Disadvantages:**

* Not good for frequent **row-level updates**.
* More suited for **read-heavy systems**.

**📌 Examples:**

* **Apache Cassandra, HBase, ClickHouse**.
* Use case: **Data warehouses, analytics, logging, time-series data**.

**2.4. Search Databases (Search Engines)**

➡️ Optimized for **full-text search** and **indexing**.  
➡️ Stores inverted indexes for fast retrieval.

**✅ Advantages:**

* Excellent for **text search, ranking, filtering**.
* Can handle unstructured/semi-structured text.
* Advanced features (fuzzy search, autocomplete, relevance scoring).

**❌ Disadvantages:**

* Not great for **transactions**.
* High memory usage.
* Consistency can lag behind writes.

**📌 Examples:**

* **Elasticsearch, Solr, OpenSearch**.
* Use case: **Google-like search, e-commerce product search, log search (ELK stack)**.

**3. Comparison Table**

| **Database Type** | **Structure** | **Best For** | **Advantages** | **Disadvantages** | **Examples** |
| --- | --- | --- | --- | --- | --- |
| **SQL (RDBMS)** | Tables (rows/columns) | Structured data, transactions | ACID, mature ecosystem, joins | Hard to scale, rigid schema | MySQL, PostgreSQL |
| **Key-Value** | Key → Value | Caching, sessions | Very fast, scalable | Limited queries | Redis, DynamoDB |
| **Document** | JSON/XML | Semi-structured, flexible | Schema-less, scalable | Weak joins, eventual consistency | MongoDB, CouchDB |
| **Column** | Column families | Analytics, OLAP | Great for aggregations | Poor row updates | Cassandra, HBase |
| **Search** | Inverted index | Text search, logs | Full-text search, relevance | Not transactional | Elasticsearch, Solr |

✅ **In System Design Interviews**

* Use **SQL** when **strong consistency, transactions** are required (banking, orders).
* Use **NoSQL (Document/Key-Value)** when **scalability, flexibility, performance** is more important.
* Use **Column stores** for **analytics & reporting**.
* Use **Search DB** when **search is the core requirement**.

**1. Scalability**

**Definition**: The ability of a system to handle increasing load (users, data, or requests) by adding resources (hardware or software).

**Types of Scalability:**

1. **Vertical Scalability (Scale Up)**
   * Add more power (CPU, RAM, SSD) to a ***single server.***
   * Example: Upgrading from 16 GB RAM → 64 GB RAM.
   * ✅ Pros: Simple to implement, no code changes.
   * ❌ Cons: Hardware limits; single point of failure.
2. **Horizontal Scalability (Scale Out)**
   * ***Add more servers***/nodes and distribute the load.
   * Example: Adding more web servers behind a **load balancer**.
   * ✅ Pros: Virtually unlimited scaling.
   * ❌ Cons: Complex architecture (load balancing, consistency issues).

📌 Example:

* Amazon’s shopping system scales horizontally during **Black Friday sales**.
* A startup may begin with vertical scaling, then shift to horizontal once traffic grows.

**🔹 2. Sharding**

**Definition**: A database partitioning technique that splits data into smaller, faster, more manageable pieces (called *shards*) and distributes them across servers.

**How it works:**

* Imagine a **user database** with 100M records.
* If on 1 server → queries become slow.
* With **sharding**:
  + Shard 1: Users with ID 1–25M
  + Shard 2: Users with ID 25M–50M
  + Shard 3: Users with ID 50M–75M
  + Shard 4: Users with ID 75M–100M

**Pros:**

* High performance (smaller DB per shard).
* *Scales horizontally.*

**Cons:**

* Complexity in query routing.
* Joins across shards are expensive.
* Rebalancing shards when data grows unevenly.

📌 Example:

* Twitter shards **tweets** by user ID.
* MongoDB has **built-in sharding support**.

**🔹 3. Flexibility**

**Definition**: The ability of a system (or database) to adapt to changing requirements, new features, or new data structures.

**SQL vs NoSQL (Flexibility view):**

* **SQL (Rigid schema)**
  + All rows in a table must follow the schema (columns fixed).
  + ✅ Data consistency guaranteed.
  + ❌ Hard to change schema later.
  + Example: Banking applications (must enforce schema).
* **NoSQL (Flexible schema)**
  + JSON-like documents or key-value stores.
  + ✅ Easy to add new fields anytime.
  + ❌ Weaker consistency sometimes.
  + Example: E-commerce product catalog (different items have different attributes).

📌 Real-world:

* Netflix uses NoSQL (Cassandra) for flexibility in video metadata.
* Banks use SQL (Postgres/Oracle) for financial transactions.

**🔹 4. Related Concepts**

**(a) Replication**

* Copying the same data to multiple nodes.
* Types:
  + ***Master-Slave (Primary-Replica)****: Writes go to master, reads from replicas.*
  + **Multi-Master**: Multiple nodes accept writes.
* ✅ Pros: High availability, fault tolerance.
* ❌ Cons: Consistency issues.

📌 Example:

* YouTube videos are replicated across global data centers.

**(b) Consistency vs Availability (CAP Theorem)**

* **Consistency (C):** Every read gets the latest write.
* **Availability (A):** Every request gets a response, even if not the latest.
* **Partition Tolerance (P):** System works despite network failures.

⚖️ In distributed systems, you must **trade-off**:

* CP (Consistency + Partition Tolerance): Banking.
* AP (Availability + Partition Tolerance): Social Media feeds.

**(c) Caching**

* Store frequently accessed data in **memory** for faster reads.
* Example: Redis, Memcached.
* Use case:
  + User sessions
  + Product price lookups
  + Recommendation results

**(d) Queueing & Event-Driven Systems**

* Queue = temporary storage for async communication.
* Examples: Kafka, RabbitMQ, AWS SQS.
* Use case: Order placement → Payment service processes later.

**🔹 5. Combined Example: E-Commerce System**

* **Frontend**: Scales horizontally (multiple servers behind load balancer).
* **Database**: SQL for orders (strong consistency), NoSQL for product catalog.
* **Sharding**: User DB sharded by User ID.
* **Caching**: Redis stores session + top-selling products.
* **Queue**: Kafka processes events like “Order Placed → Payment Service”.
* **Replication**: Product images replicated in global CDNs for fast access.

✅ **In interviews**, companies like Amazon, Microsoft, Flipkart, and Swiggy often ask:

* *“How will you scale a user database with 100M users?”*
* *“Explain sharding with an example.”*
* *“Why would you choose SQL vs NoSQL in your project?”*

**🔹 Applications vs Services**

**1. Application**

**👉 What is it?**

An **application** is a software program designed for end-users (humans). It provides a **complete functionality** to perform tasks like banking, chatting, shopping, or booking tickets.

**👉 Why use it?**

Because users need an **interface** (mobile app, website, or desktop app) to interact with the system and use business functionalities.

**👉 Use Cases:**

* Banking applications (SBI app, PayPal, etc.)
* E-commerce (Amazon, Flipkart)
* Social media (Instagram, WhatsApp)

**👉 Advantages:**

✅ Provides end-user experience (UI/UX).  
✅ Integrates with backend services to provide full functionality.  
✅ Easy for users to interact with complex systems.

**👉 Disadvantages:**

❌ Heavier to maintain (UI + business logic + data connection).  
❌ Needs frequent updates for compatibility & user features.  
❌ Can be less reusable (tightly coupled to business logic).

**👉 Examples:**

* **WhatsApp app** on your phone (Application)
* **Amazon website** (Application)
* **Zoom Desktop app**

**2. Service**

**👉 What is it?**

A **service** is a **backend component** that performs a specific **business function** or **technical task**. Services are usually reusable and talk to applications or other services using **APIs, events, or messages**.

* Services are typically **invisible** to the user directly (users only interact through an application).
* Example: Payment Service, Notification Service, Authentication Service.

**👉 Why use it?**

Because instead of building a **monolithic app**, we divide the system into **services (microservices)** that are independent, scalable, and reusable.

**👉 Use Cases:**

* Payment Gateway service (Stripe, Razorpay, PayPal API).
* Authentication service (Google Sign-In, Auth0).
* Notification service (Email/SMS push notifications).

**👉 Advantages:**

✅ Reusability across multiple apps.  
✅ Independent scaling (scale only the payment service if traffic is high).  
✅ Easier to maintain and deploy (microservices model).  
✅ Encourages modular design.

**👉 Disadvantages:**

❌ Adds network overhead (extra latency since calls go via APIs).  
❌ Complex monitoring & debugging (distributed system).  
❌ Requires service discovery, load balancing, and orchestration tools (Kubernetes, Consul, etc.).

**👉 Examples:**

* **Google Maps API** (Service) used inside **Uber App** (Application).
* **Stripe Payment API** (Service) used inside **E-commerce Website** (Application).
* **Firebase Authentication** service used in mobile apps.

**🔹 Applications vs Services – Side by Side**

| **Feature** | **Application** | **Service** |
| --- | --- | --- |
| **End User Facing?** | ✅ Yes | ❌ No |
| **Purpose** | Provide complete functionality with UI | Provide backend business/technical capability |
| **Examples** | WhatsApp, Amazon Website | Stripe API, Auth0, Firebase |
| **Scalability** | Scale whole app | Scale individual service |
| **Coupling** | Tightly coupled | Loosely coupled (microservices) |
| **Maintenance** | Harder (UI + logic) | Easier (independent deployments) |

**🔹 Real-Life Example (Banking System)**

* **Application**: SBI Mobile App → lets users log in, transfer money, pay bills.
* **Services** inside:
  + Authentication Service → verifies login.
  + Payment Service → processes transactions.
  + Notification Service → sends SMS/Email alerts.

So the **app = face for the user**, while **services = invisible workers inside**.

Great question 👍 Let’s go **deep into Application Programming Interface (API)**, since it’s one of the most important concepts in **System Design and Software Engineering**.

**🌐 What is an API?**

* **API (Application Programming Interface)** is a **contract/bridge** that allows one software application to interact with another.
* Think of it like a **restaurant menu**:
  + The **menu (API)** describes what dishes (services/functions) are available.
  + You (the client) place an order through the menu.
  + The **kitchen (server/system)** prepares the dish and gives it back.
  + You don’t care how it’s cooked inside → just that you get the result.

👉 So, an **API hides complexity** and provides a **defined way to communicate** between systems.

**🛠️ Why APIs are needed**

1. **Integration** – Allow different systems/apps to talk (e.g., Google Maps API used in Uber).
2. **Abstraction** – Users don’t need to know the internal logic (just call the API).
3. **Reusability** – Same backend logic can serve multiple clients (web, mobile, 3rd-party apps).
4. **Scalability** – APIs help in microservices & distributed system design.
5. **Security** – APIs can enforce authentication/authorization (tokens, keys).

**📊 Types of APIs**

**1. Web APIs (most common)**

* Used over HTTP/HTTPS
* Example: REST, GraphQL, gRPC
* Use case: Web apps, mobile apps, microservices communication.

**2. Library APIs**

* Functions exposed by libraries/frameworks (e.g., Java Collections API, NumPy API).
* Use case: Software development.

**3. Operating System APIs**

* OS exposes APIs for apps (e.g., Windows API, POSIX API).
* Use case: File I/O, process handling.

**4. Hardware APIs**

* Allow apps to interact with hardware (e.g., camera, GPS, GPU APIs).
* Use case: Games, IoT, mobile devices.

**🔑 Common API Protocols/Styles**

**1. REST (Representational State Transfer)**

* HTTP-based, stateless.
* Data in JSON/XML.
* Example: GET /users/1
* ✅ Simple, scalable, widely used.
* ❌ Not efficient for complex queries.

**2. GraphQL**

* Query language for APIs.
* Client asks exactly what it needs.
* Example: { user(id:1) { name, email } }
* ✅ Reduces over-fetching/under-fetching.
* ❌ Complex server setup.

**3. gRPC (Google Remote Procedure Call)**

* Uses HTTP/2, Protocol Buffers (binary).
* ✅ Super fast, great for microservices.
* ❌ Harder to debug, not human-readable.

**4. SOAP (Simple Object Access Protocol)**

* XML-based, strict rules.
* ✅ High security, standardized.
* ❌ Heavy and slower.

**⚡ API Use Cases**

1. **Payment Gateways** – Stripe, PayPal APIs for transactions.
2. **Social Media Integration** – Facebook Graph API, Twitter API.
3. **Maps & Location** – Google Maps API, OpenStreetMap API.
4. **Cloud Services** – AWS S3 API, Azure API.
5. **Internal Microservices** – APIs between services in an organization.

**✅ Advantages of APIs**

* Interoperability (connect any two systems).
* Code reusability.
* Scalability (serve multiple clients).
* Security (controlled access).
* Faster development (use existing services).

**❌ Disadvantages of APIs**

* Dependency on third-party services (if API is down, your app suffers).
* Maintenance overhead (APIs evolve, need versioning).
* Security risks (if authentication isn’t strong).
* Rate limits (many APIs restrict requests).

**🔄 API Example (REST)**

Suppose you’re building a **Weather App**.

* **Client request (GET)**
* GET https://api.weather.com/v1/current?city=London
* **Server response (JSON)**
* {
* "city": "London",
* "temperature": "18°C",
* "condition": "Cloudy"
* }
* You didn’t calculate weather yourself → API did it for you.

**🏗️ API in System Design**

* In **2-tier architecture**: Client directly talks to DB (not recommended).
* In **3-tier architecture**: API layer (server) interacts between client and DB.
* In **Microservices**: Services communicate using APIs (REST/gRPC).
* In **Event-driven systems**: APIs trigger events/messages between components.

👉 In short:  
**API = a contract/interface that enables structured, secure, and efficient communication between software components.**

Great question 🚀 — caching is **one of the most important concepts in system design**, and interviewers love to test your depth here. Let’s go step by step in detail:

**🔹 What is Caching?**

**Caching** is the process of storing a copy of frequently accessed data in a **temporary storage layer (cache)** so that future requests for that data can be served **faster**, without going back to the original (slower) source like a database, disk, or API.

👉 Think of it as keeping your **favorite book on your desk** instead of searching the library every time.

* **Where caches live?**
  + In-memory (RAM, e.g., **Redis, Memcached**)
  + Browser cache (local storage, cookies)
  + CDN (edge caching for static files)
  + Application-level cache

**🔹 Why Caching?**

* **Performance** → Reduce latency (faster responses).
* **Scalability** → Reduce load on DB and backend.
* **Cost efficiency** → Minimize expensive computations or API calls.
* **Availability** → Serve data even if DB is down (to some extent).

**🔹 Cache Patterns**

These are **strategies for where and how caching is applied**:

**1. Cache-aside (Lazy Loading)**

* App checks cache → if data **not found**, fetch from DB → put it in cache.
* **Example:** Amazon product page details.

✅ Pros: Cache only stores what is needed.  
❌ Cons: First request is slow (cache miss).

**2. Read-through**

* App **always reads through the cache**. If not present, cache fetches from DB and updates itself.
* **Example:** Redis + RDS with library integration.

✅ Pros: Transparent for app.  
❌ Cons: Cache becomes a bottleneck if misconfigured.

**3. Write-through**

* Every **write goes to cache first**, then DB.
* **Example:** User profile update → update cache + DB.

✅ Pros: Cache and DB always in sync.  
❌ Cons: Higher write latency.

**4. Write-behind (Write-back)**

* Writes go to cache first, and DB update happens **asynchronously**.
* **Example:** High-frequency logging system.

✅ Pros: Fast writes.  
❌ Cons: Risk of data loss if cache crashes before DB update.

**5. Refresh-ahead**

* Cache predicts and **refreshes data before it expires**, based on access patterns.
* **Example:** Netflix pre-fetching trending shows for next requests.

✅ Pros: Reduces cache misses.  
❌ Cons: Might cache unused data.

**🔹 Cache Invalidation**

**Problem:** How do we keep cache and DB consistent?  
👉 "The two hardest problems in CS are: naming things, cache invalidation, and off-by-one errors."

**Strategies:**

1. **Time-to-Live (TTL)**
   * Cache expires automatically after X seconds/minutes.
   * Example: Weather app cache → expires every 10 min.
2. **Manual Invalidation**
   * App explicitly removes/updates cache after a DB write.
   * Example: Update product price → invalidate cache entry.
3. **Event-driven Invalidation**
   * Cache listens for events from DB (like **CDC – Change Data Capture**) and updates itself.

**🔹 Cache Eviction Policies**

When cache is **full**, old data must be removed:

* **LRU (Least Recently Used)** → Remove least accessed recently (most common).
* **LFU (Least Frequently Used)** → Remove least frequently used.
* **FIFO (First In First Out)** → Remove oldest added.
* **Random** → Remove random entry (simple but unpredictable).

**🔹 Examples of Caching in Real Systems**

* **CDN (Akamai, Cloudflare):** Cache static assets (images, CSS, JS).
* **Redis / Memcached:** Store session tokens, user data, leaderboard, shopping cart.
* **Database query results:** Popular queries cached for fast retrieval.
* **OS Page Cache:** Keep disk pages in memory.
* **Browser Cache:** Cache static files locally.

**🔹 Advantages of Caching**

✅ Faster response times (low latency).  
✅ Reduces load on databases and servers.  
✅ Improves scalability (handle more users).  
✅ Can act as a **buffer** in case of DB downtime.

**🔹 Disadvantages of Caching**

❌ Cache inconsistency with DB (stale data).  
❌ Increased complexity in invalidation logic.  
❌ Memory/storage overhead.  
❌ May cause **cache stampede** (too many requests when cache expires at once).

**🔹 Use Case Example**

📌 **E-commerce app (like Flipkart / Amazon)**

* **Product details** → Cached using read-through.
* **Shopping cart/session** → Cached in Redis (write-through).
* **Search results** → Cached with TTL (cache-aside).
* **Recommendations** → Cached with refresh-ahead to pre-warm results.
* **Static images** → Cached at CDN (edge caching).

👉 So caching is not **one-size-fits-all**. The right **pattern** + **invalidation strategy** + **eviction policy** depends on the system’s **read/write ratio**, **freshness requirement**, and **traffic pattern**.

Do you want me to also **draw a layered system diagram** (DB + API + cache flow for each pattern) so it’s crystal clear for interview prep?

For interview preparation follow below channel.,

**ByteByteGo**