

System Design

Part II

1. Architecture:

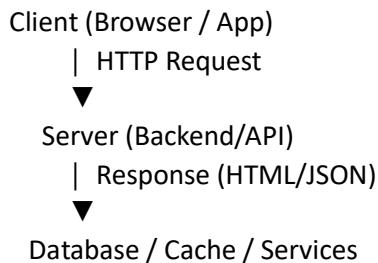
1. Client Server Architecture:

Client–server architecture is a **distributed system design** where:

Client → requests services/data

Server → processes request and sends back response

They communicate mainly through **HTTP/HTTPS** in web applications, Through APIs.



Client Responsibilities:

- Display UI
- Take input from user
- Render server response
- Store tokens/cookies

Examples: React, Angular, Vue, mobile apps, browsers.

Server Responsibilities:

- Process client requests
- Apply business logic
- Validate input
- Authenticate / authorize users
- Interact with database
- Send JSON/HTML responses

Examples: Express (NodeJS), Spring Boot, Django, Laravel.

TYPES OF CLIENT–SERVER ARCHITECTURE:

1. 1-Tier Architecture:

1-Tier architecture means the application, logic, and data are stored in the same machine or environment—no separation between client and server.

Everything in one single system

UI + Logic + Database together.

Example: Microsoft Excel, Local desktop applications

Characteristics of 1-Tier Architecture:

- Application executes on **one machine**

- Database is stored **locally**
- No external API calls
- Fast execution (no network calls)
- Simple to build but not scalable

Examples:

- MS Excel: Data (sheet) + Logic (formulas) + UI (Excel window) and Everything runs inside your computer
- Desktop Applications: Notepad, calculator, Paint etc
- Single-Machine Database Systems: Offline billing software used in shops, Microsoft Access

Advantages:

- Very fast (no network)
- Easy to develop
- Low cost
- No complex deployment

Disadvantages:

- Not scalable (only one machine)
- Hard to share data (local machine only)
- Limited users (single user environment)
- Security risk if the machine is damaged

2. 2-Tier Architecture (Client ↔ Server)

Most simple web architecture: Client ↔ Server (API + DB)

Server (Tier 2) → Database + heavy business logic

The client communicates **directly** with the server, usually through SQL queries or an API.

There are only **two layers**, hence "2-tier."

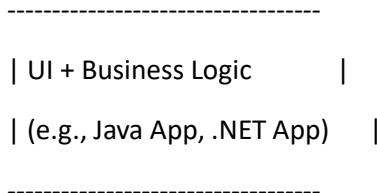
In 2-tier architecture, the client talks directly to the server without any middle layer.

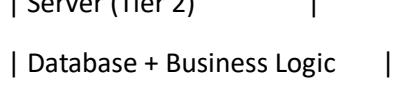
Database usually connects directly to backend

Backend handles business logic + DB operations

Diagram:

Client (Tier 1)





How it works:

- User interacts with a **client application**.
- Client sends request **directly** to the server.
- Server processes logic and queries the **database**.
- Server returns results to the client.

Where 2-Tier is used?

- Medium-sized systems
- Small organizations
- Internal applications

Real life Examples:

- Desktop App ↔ Database: Hospital management desktop apps, Library management systems, Android App directly calling Java backend, React App calling a Node.js REST API without microservices

Types of 2-Tier Architecture:

1. **Fat Client / Thick Client:**
Most logic on the client.
Example: Desktop software with forms, Software sends SQL queries directly to DB
Client = Big
Server = Only DB
2. **Fat Server / Thin Client:**
Most logic on the **server**.
Example: Web app where the client is only HTML/JS, Server stores business logic + DB
Client = Simple UI
Server = Heavy logic

Advantages:

- Fast communication
- Simple architecture
- Good for small team

- Less latency

Disadvantages:

- Not scalable
- Security Issues
- Hard maintenance
- No load balancing

3. 3-Tier Architecture (Most Common in Web Apps):

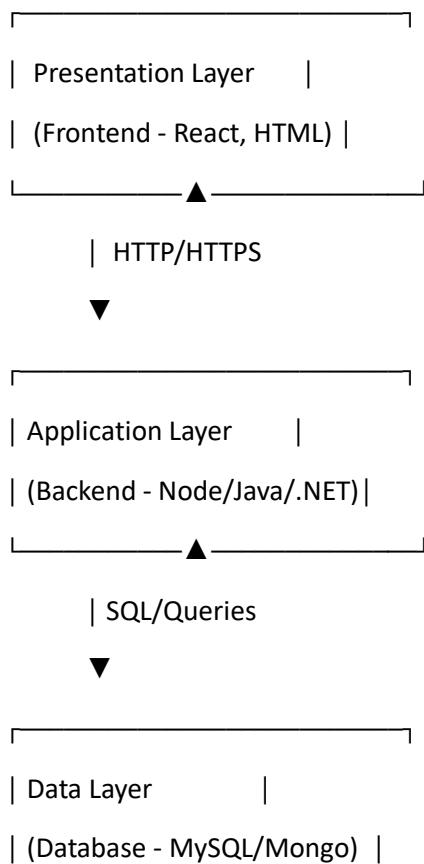
Client → Application Server → Database Server

- a. Client Layer : React / HTML / Mobile Apps
- b. Application Layer (Backend): NodeJS, Django, Spring Boot
Handles business logic, authentication, routing
- c. Database Layer: MongoDB, MySQL, Postgres

3-Tier Architecture is the most common architecture used in **web applications** and **SaaS products**.

3-tier architecture separates UI, backend logic, and database into 3 layers so the system becomes scalable, maintainable, and secure.

Diagram:





Real-World Examples:

1. Instagram
2. Flipkart/Amazon
3. Any web application

Benefits:

1. High Scalability
2. Maintainability
3. Security
4. Reusability
5. Easy Deployment

Disadvantages:

1. Slightly complex to build initially
2. Needs more infrastructure
3. More network calls = slightly slower than 2-tier

4. N-Tier (Multi-Tier) Architecture:

N-Tier Architecture, also called Multi-Tier Architecture, is an extension of 3-tier architecture.

Instead of just 3 layers (UI → Backend → DB), it contains multiple layers (N layers), where each layer has a specific, independent responsibility.

You can add layers like:

- Cache Layer
- API Gateway Layer
- Authentication Layer
- Microservices Layer
- Message Queue Layer
- Analytics Layer
- Load Balancer Layer
- Logging Layer

Used by large companies:

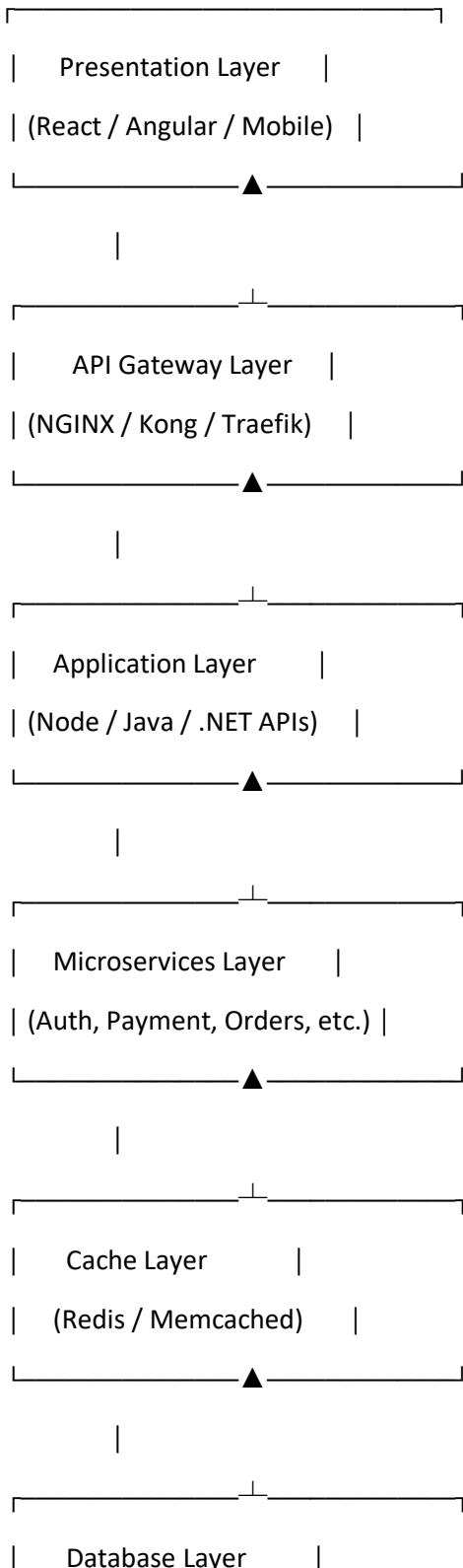
Client → API Gateway → Microservices → DBs → Queues → Cache

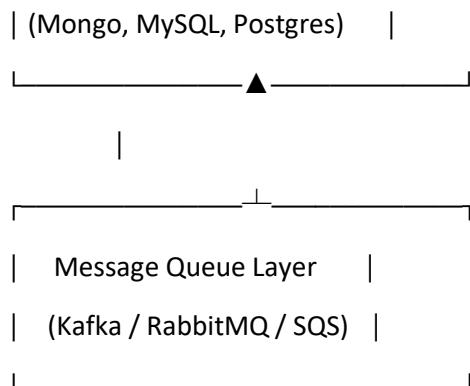
Components include:

- Load balancer
- Multiple backends (microservices)
- Cache (Redis)

- Message Queue (Kafka/RabbitMQ)
- Search Engine (Elasticsearch)

Diagram:





Tiers of N-Tier Architecture:

1. Presentation Layer

User UI

- React
- Angular
- Vue
- Android/iOS

Handles only UI/UX.

2. API Gateway Layer

Acts as a single-entry point for all client requests.

- NGINX
- Kong
- API Gateway (AWS)
-

Does:

- Routing
- Load balancing
- Rate limiting
- Authentication

3. Application Layer (Backend Services):

REST APIs or GraphQL APIs using:

- Node.js
- Java Spring Boot
- Django
- .NET
-

Handles:

- Business logic
- Data validation

- Authorization
- Integrations

4. Microservices Layer:

Instead of one backend, functionality is divided into multiple services:

- Auth Service
- Payment Service
- User Service
- Order Service

Each service has its own:

- Codebase
- Database
- Deployment

5. Cache Layer:

Improves performance and speed.

Examples:

- Redis
- Memcached

Stores:

- Session data
- Frequently used queries
- Token data

6. Database Layer:

Databases for persistent storage.

- MySQL
- PostgreSQL
- MongoDB
- DynamoDB
- Cassandra

7. Messaging Layer:

Used for asynchronous communication between services.

Examples:

- Kafka
- RabbitMQ
- AWS SQS

8. Logging + Monitoring Layer:

Used for logs, metrics, and monitoring.

Examples:

- ELK Stack
- Prometheus + Grafana
- CloudWatch

5. Peer-to-Peer (P2P) Architecture:

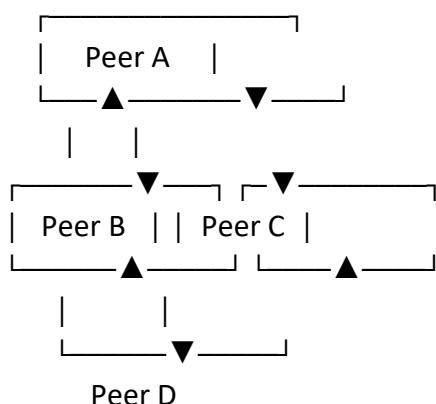
Clients act as both **server AND client**.

Every device in the network (called a **peer**) can act as:

- **Client** → requesting data
- **Server** → providing data

This means **each peer is equal**, and they communicate **directly** with each other.

Diagram:



Examples:

- Torrent systems
- Blockchain
- Skype (old architecture)

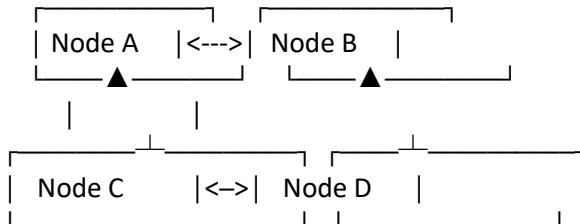
6. Distributed Architecture:

Distributed Architecture is a system design where an application is **divided into multiple components** that run on **different machines**, but work together as **one unified system**.

These machines (called **nodes**) communicate over a **network**.

Simple definition: Distributed architecture is a model where computation, storage, and control are spread across multiple networked computers (nodes), each performing part of the overall task.

Diagram:



All nodes share workload & communicate

Key characteristics:

- Multiple independent nodes
- Communicate over network
- High availability
- Parallel execution
- Scalability
- Resource sharing

Used in:

- Google Cloud
- AWS
- Distributed databases
- Scalable backend systems

Types of Distributed Architectures

- Distributed Database Architecture:
- Distributed Computing Architecture
- Microservices-Based Distributed Architecture
- Distributed File Systems

N-Tier Architecture vs Distributed Architecture

<u>Feature</u>	<u>N-Tier Architecture</u>	<u>Distributed Architecture</u>
Definition	Logical separation into multiple layers (tiers)	System spread across multiple physical nodes

<u>Feature</u>	<u>N-Tier Architecture</u>	<u>Distributed Architecture</u>
Focus	Structure and responsibilities	Scalability, reliability, node distribution
Deployment	Can be on single or multiple servers	Always multiple servers/nodes
Communication	Usually vertical (UI → API → DB)	Can be any pattern (peer-to-peer, message passing, replication)
Fault Tolerance	Not automatically fault-tolerant	High fault tolerance by design
Examples	3-tier web app, ERP systems	Microservices, distributed DBs, cloud systems
Scaling	Tier-based scaling (scale UI separately)	Distributed node scaling (horizontal)
Coupling	Typically more tightly coupled	Loosely coupled microservices or nodes

Use cases:

Small to Medium Systems → N-Tier Architecture

Large-Scale, High-Traffic Systems → Distributed Architecture

In simple words:

N-Tier = How the app is built.

Distributed = How the app runs at scale.