1. Scalability

Meaning:
Scalability is the system's ability to handle **increasing load** (more users, more data, more requests) *without degrading performance*.

Two types:

1. **Vertical scaling (Scale-Up)**
   Increase the power of a single machine (more RAM, CPU).
   ✔ Simple
   ✘ Limited by hardware
   ✘ Downtime during upgrades

2. **Horizontal scaling (Scale-Out)**
   Add more machines to distribute the load.
   ✔ Practically unlimited
   ✔ Fault-tolerant
   ✘ Requires distributed systems complexity (sharding, load balancing)

Example:
If your e-commerce website goes from **10K to 1M users/day**, scalability ensures the system doesn't slow down or break.

Interview Tip:
Mention how caching, sharding, and microservices help achieve scalability.

---

2. Availability

Meaning:
Availability means the system is **up and operational** whenever users need it.

Formally expressed as **percentage uptime**, e.g.,

- 99% → ~3.65 days downtime/year
- 99.9% (3-nines) → ~8.7 hours downtime/year
- 99.999% (5-nines) → ~5 minutes downtime/year

How availability is achieved:

- Redundancy (multiple replicas)
- Load balancing
- Failover systems
- Removing single points of failure

Example:
Amazon product page must always load—even if one server fails, traffic should move to another replica automatically.

---

3. Latency Constraints

Meaning:
Latency is the **time taken** for the system to respond to a request.
It's not throughput; it's the per-request "speed".

Key Types:

- **Network latency** – time for data to travel between client and server
- **Processing latency** – time taken to compute/process request
- **Disk I/O latency** – read/write delays
- **DB query latency**

Typical latency expectations:

- API should respond < 100ms
- Cache lookup ~1–2ms
- DB read ~10–50ms
- Cross-region request ~100–200ms

Interview Tip:
Talk about using caching (Redis), CDNs, read replicas, and denormalized data to reduce latency.

---

4. Consistency vs Availability (CAP Consideration)

Background:
In distributed systems, due to network failures, you can only guarantee **two out of three**:

- **C — Consistency:**
  All nodes return the **same data** at any given time.

- **A — Availability:**
  Every request gets a response, even if some nodes are down.

- **P — Partition Tolerance:**
  System continues working even when communication between nodes breaks.

Since **network partition is unavoidable**, real systems must choose between:

- **CP (Consistency + Partition tolerance):**
  May reject requests (reduced availability) to ensure consistent data.
  Example: **HBase, MongoDB in strong-consistency mode**

- **AP (Availability + Partition tolerance):**
  Always respond, but data might be temporarily inconsistent.
  Example: **DynamoDB, Cassandra**

Interview Tip:
Explain that system design choices depend on the business domain:

- Financial transactions → Prefer **Consistency**

- Social media feeds → Prefer **Availability**

---

5. Fault Tolerance

Meaning:
Fault tolerance means the system continues to work correctly even if **some components fail**.

Why it matters:
In large distributed systems, failures are **not rare**—they're expected. Servers, networks, disks, and even entire data centers can fail unexpectedly.

How fault tolerance is achieved:

- Replication (multiple copies of data/services)
- Automatic failover (detecting failure and routing to healthy nodes)
- Retry mechanisms
- Circuit breakers
- Chaos testing (Netflix's “Chaos Monkey”)

Example:
If one Redis node crashes, another replica automatically takes over, ensuring no downtime.

✨ Final Summary Table

NFR	What It Ensures	Achieved Through
Scalability	Can handle growing load	Sharding, caching, load balancing
Availability	System stays up	Replication, redundancy, health checks, failover
Latency	Fast response times	CDN, caching, indexing, distributed architecture
Consistency vs Availability	Data correctness vs uptime during failures	CAP tradeoffs, quorum reads/writes
Fault Tolerance	System survives failures	Replicas, retries, circuit breakers, monitoring

🟫 What is a CDN?

A **Content Delivery Network** is a collection of servers located in different regions of the world. These servers **cache static content** so users can fetch it from a **nearby edge server** instead of your main backend server.

✔ Without CDN

User → Your server (maybe far away) → High latency
Images load slowly → Bad user experience

✔ With CDN

User → Nearest CDN server → Very low latency
Images load instantly → Fast and smooth UI

🟫 Why do we use a CDN?

1. Faster load times

- Product images, thumbnails, CSS, JavaScript, and videos load quickly.
- Distance between user and content is minimized.

2. Reduces load on your backend

- Your application server doesn't have to store or serve images.
- It handles only business operations (orders, carts, payment).

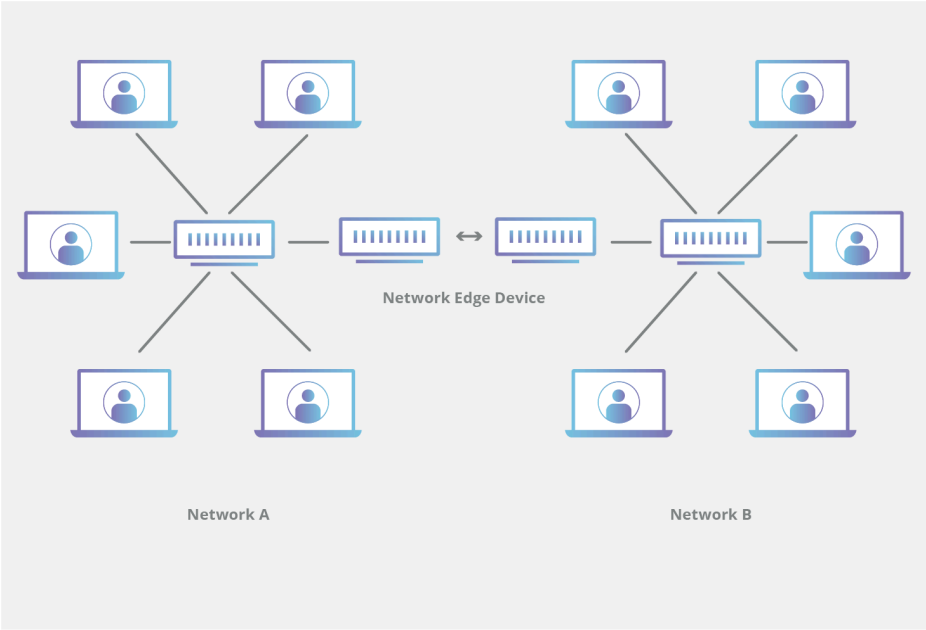
3. Highly scalable

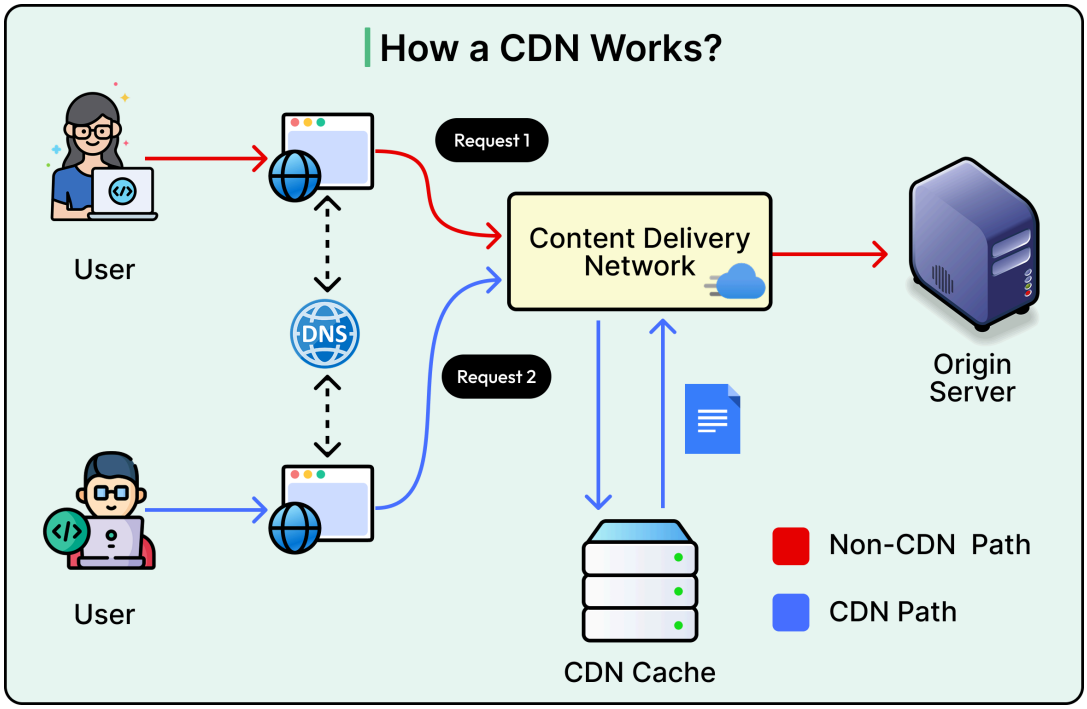
- CDNs are built to handle **millions of parallel requests**.
- No need to scale your own servers for static file delivery.

4. Better availability

- Even if one CDN node fails, the request is served by another nearby node.

🟫 CDN in E-commerce Example





Imagine your e-commerce website (like Amazon/Flipkart):

Users want to see:

- Product images
- Category banners
- Icons
- CSS files
- JS bundles

If these files are served from your backend server, **every request hits your infra**, slowing down both frontend & backend.

Instead:

Flow With CDN

1. Product images uploaded → Stored in object storage (Amazon S3, Google Cloud Storage)
2. CDN (CloudFront, Akamai, Cloudflare) fetches the image once and caches it.
3. Next time a user from any location requests the same image:
    - CDN serves from *nearest edge server*, not the origin server.

This reduces:

- Latency
- Backend load
- Bandwidth usage
- Cost

🟫 How CDN works technically

Step-by-step:

1. **User requests an image**
   `cdn.yourstore.com/product123.png`

2. CDN checks nearest edge node cache:
   - If **hit** → Serve instantly (1–10ms)
   - If **miss** → Fetch from origin server → Cache → Serve

Cache rules:

- TTL (Time To Live) decides how long to cache
- Invalidation tells CDN to clear outdated files

🟫 Important Terms

Term	Meaning
Edge Server	A server close to user's region that serves cached content
Origin Server	Your main storage (like S3) where the original file lives
Cache Hit	CDN found the content in cache
Cache Miss	CDN didn't have the file → fetch from origin
TTL	How long CDN keeps content cached

🟫 In short: What is CDN here?

👉 A CDN is a **network of globally distributed caching servers** used in system design to deliver **static content (product images, CSS, JS, etc.) quickly** to users with minimal latency.

It is one of the core non-functional components of any **e-commerce architecture**, improving:

- Speed
- Scalability
- Reliability
- User experience

If you want, I can also show **how CDN fits into the high-level design diagram of an e-commerce system**.

Do you like this personality?

🌐 What Is the OSI Networking Model?

The **OSI (Open Systems Interconnection) model** is a *conceptual framework* that standardizes how data moves across a network — from one device to another.

It divides networking into **7 layers**, each with a specific purpose. Think of it as a pipeline that data flows through.

Even though modern internet protocols don't strictly follow OSI, the model is still used everywhere in interviews to explain:

- How data transmission works
- Where certain technologies operate (e.g., load balancers, SSL, TCP)
- What breaks when the network fails

📶 The 7 OSI Layers (Simple, Interview-Friendly Explanation)

Below is a clear, intuitive explanation of each layer.

Layer 1 – Physical Layer

🔹 *"Bits travelling across wires."*

- Sends raw 0s and 1s over a physical medium
- Includes cables (fiber, copper), WiFi radio waves
- Hardware: NIC, hubs, repeaters

💡 System design relevance:
Not much. Only matters when discussing "network partition due to physical link failure."

Layer 2 – Data Link Layer

🔹 *"Local delivery within a LAN."*

- Responsible for **MAC addresses**
- Ensures error detection and correction within a single physical network
- Breaks data into frames

Examples:
Ethernet, Wi-Fi (802.11), switches

💡 Relevance:
Useful if discussing

- ARP

- MAC flooding
- Switch-level segmentation

Layer 3 – Network Layer

🔹 *“Finds the path across networks.”*

- Handles **routing**
- Uses **IP addresses**
- Decides the best path for packets

Examples:
IPv4, IPv6, routers

💡 Relevance in system design:
⭐ Important
Because your distributed service spans multiple networks and datacenters.

Layer 4 – Transport Layer

🔹 *“Reliable or fast delivery?” TCP vs UDP.”*

Responsible for **end-to-end communication**.

Provides:

- Port numbers
- Reliability (TCP)
- Speed (UDP)
- Flow control

- Retransmissions

Examples:
TCP, UDP

💡 **Extremely important layer**
This is where:

- slow clients
- timeouts
- load balancers (L4)
- connection limits
- backpressure

show up in real systems.

Layer 5 – Session Layer

🔹 *“Maintains connection sessions.”*

Establishes, manages, and terminates sessions.

Not heavily used directly today; rolled into higher layers.

Examples:
Authentication, session handling

💡 Relevance:
Explaining long-lived API connections (WebSockets, gRPC).

Layer 6 – Presentation Layer

🔹 *“Data format + encryption.”*

Handles how data is represented.

Examples:

- JSON, XML, Protobuf
- Data compression
- SSL/TLS encryption (often mapped here)

💡 Relevance:
Yes — when discussing HTTPS, TLS termination at load balancer, or compression.

Layer 7 – Application Layer

🔹 *"Where user-facing apps live."*

This is where application protocols run.

Examples:

- HTTP / HTTPS
- DNS
- SMTP
- API requests

💡 **Most important for system design**
This is what you scale:

- REST APIs
- Microservices
- Web apps

🌟 In System Design Interviews, Only 3 Layers Truly Matter

Your PDF also highlights this:

✔ Layer 7 → Application Layer

(Home of your APIs, microservices, RPC)

✔ Layer 4 → Transport Layer

(How connections behave: TCP limits, UDP for streaming)

✔ Layer 3 → Network Layer

(Data routing, network partitions, IP-level behavior)

These layers decide:

- How clients talk to servers
- How load balancers forward requests
- How failures ripple across regions
- How TCP congestion can slow your backend

📘 Example to Tie It Together

Imagine a user uploading a photo on Instagram.

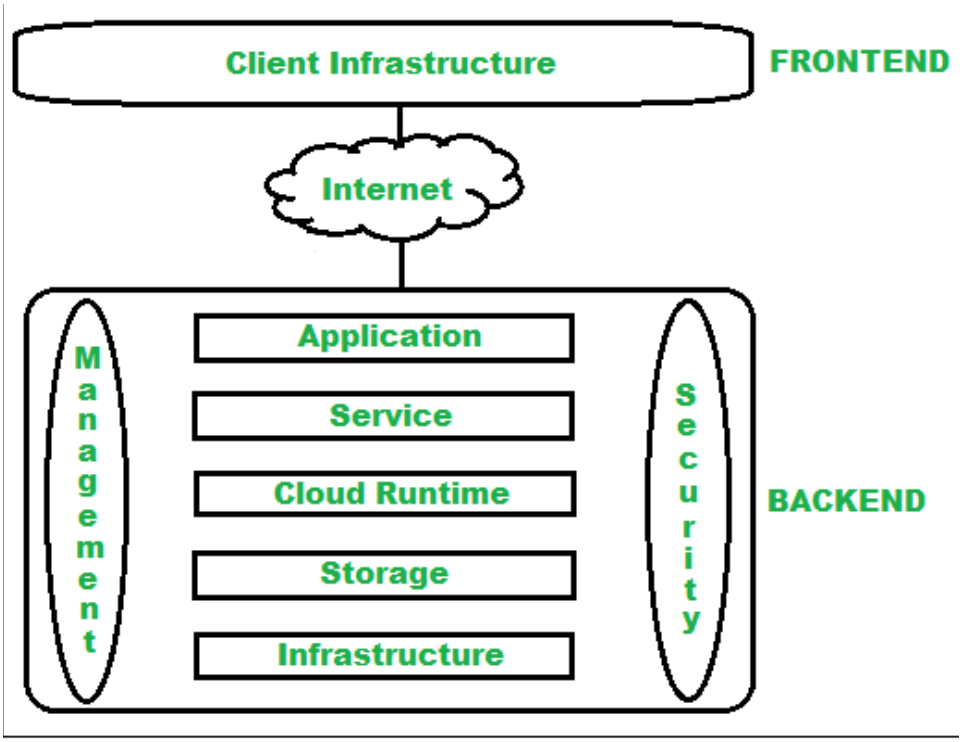
1. **L7 (Application)** — User sends HTTP POST to upload endpoint
2. **L6 (Presentation)** — Data gets encoded (JPEG) + encrypted (TLS)
3. **L4 (Transport)** — TCP ensures reliable delivery
4. **L3 (Network)** — Routers forward packets through the internet
5. **L2/L1** — Bits move over fiber/WiFi

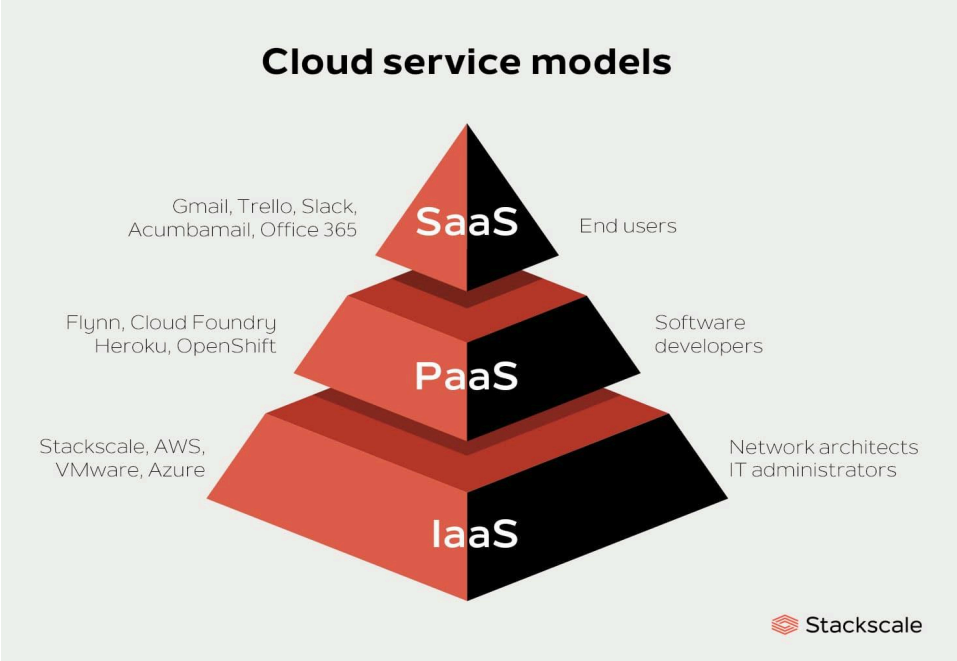
At each layer, something different happens — but all cooperate to move data safely.

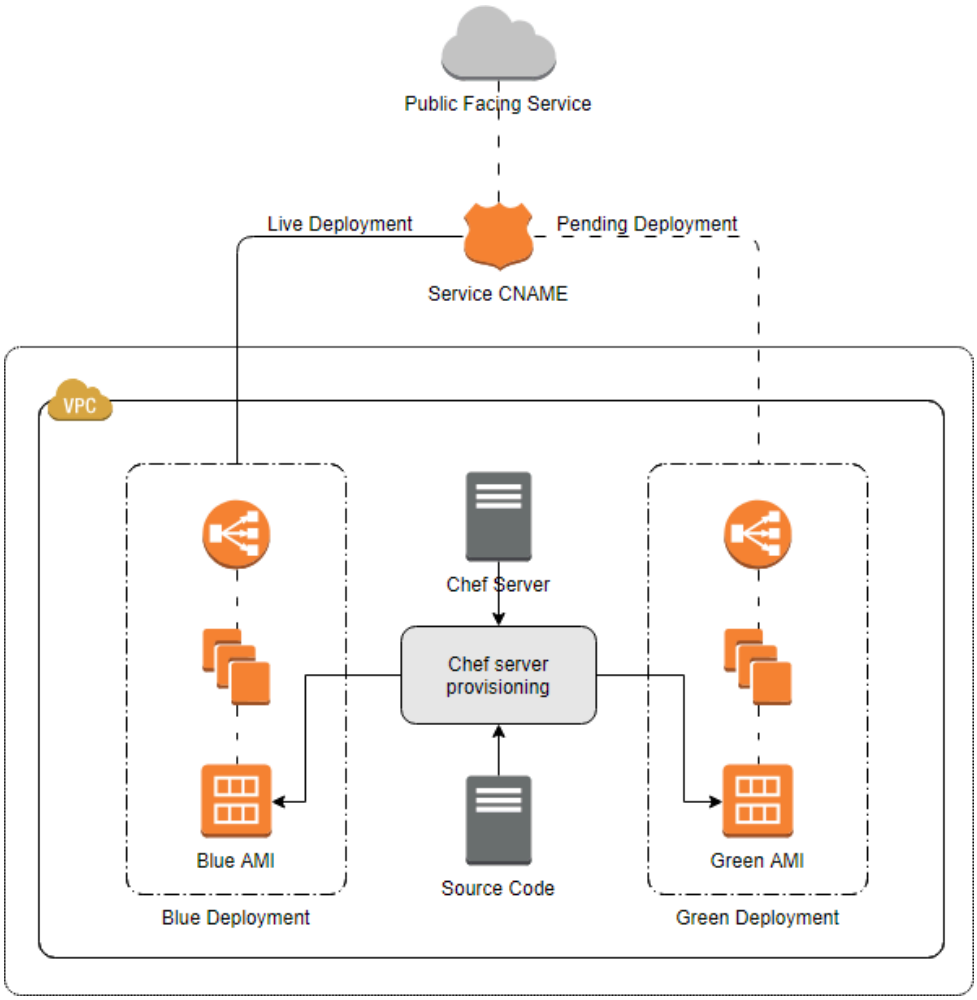
🎯 Final Summary (Easy to Remember)

OSI Layer	What It Does	Why It Matters
7	APIS, HTTP, DNS	System design logic lives here
6	Encryption, data format	TLS termination, compression
5	Sessions	WebSockets, gRPC
4	TCP/UDP	Timeouts, backpressure, scaling
3	Routing, IP	Multi-region routing, partitions
2	MAC, LAN	Rare in interviews
1	Cables, WiFi	Rare unless discussing failures

☁️ 1. What Are Cloud Services?




Cloud service models
Gmail, Trello, Slack,
Acumbamail, Office 365
SaaS
End users
Flynn, Cloud Foundry
Heroku, OpenShift
PaaS
Software
developers
Stackscale, AWS,
VMware, Azure
IaaS
Network architects
IT administrators
Stackscale


Public Facing Service
Live Deployment
Pending Deployment
Service CNAME
VPC
Chef Server
Chef server
provisioning
Blue AMI
Source Code
Green AMI
Blue Deployment
Green Deployment

Simple definition:

Cloud services are computing resources—like servers, storage, databases, and APIs—provided over the internet so that companies don't need to manage physical hardware.

Instead of buying your own servers, you **rent** computing power from providers like **AWS, Azure, GCP** and scale it whenever needed.

🔹 Why Cloud Services Exist

Before cloud computing:

- Companies bought servers → expensive
- Setup + maintenance → slow
- Scaling → hard
- Downtime → common

With cloud:

- Servers available in seconds
- Pay only for what you use
- Auto-scaling
- Global deployment without owning datacenters

🔹 Types of Cloud Services

1. IaaS (Infrastructure as a Service)

You rent virtual machines, storage, load balancers.
Examples: AWS EC2, Azure VM.

2. PaaS (Platform as a Service)

Cloud handles runtime + deployment; you push your app.
Examples: AWS Elastic Beanstalk, Google App Engine.

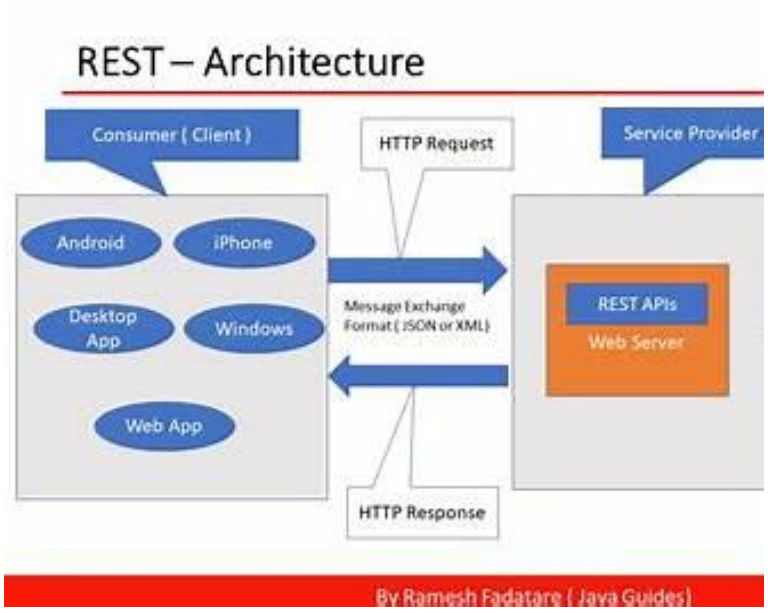
3. SaaS (Software as a Service)

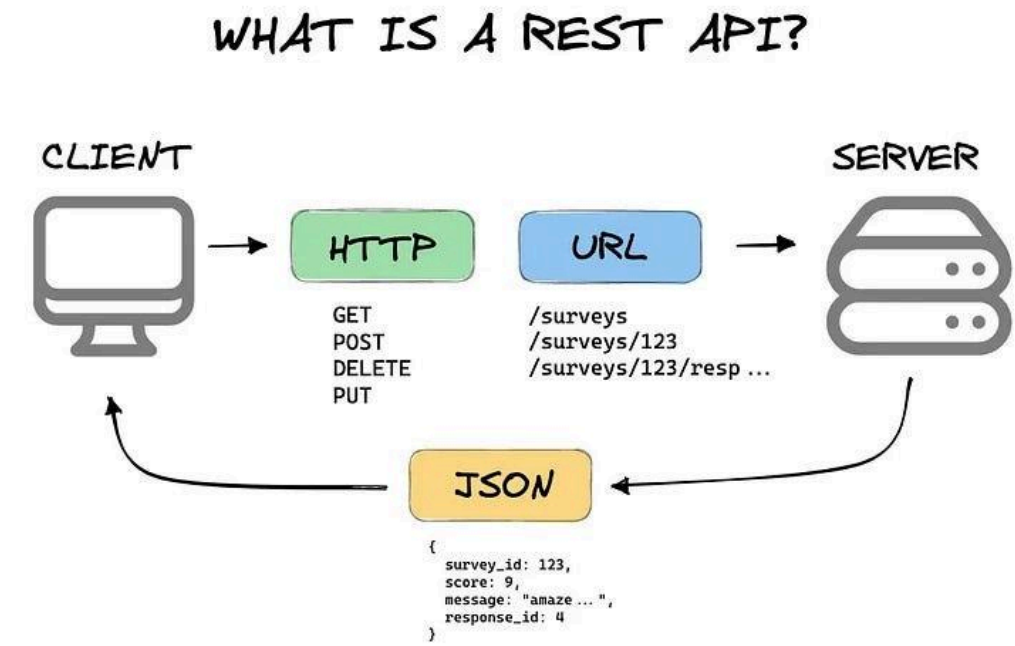
Complete applications delivered on web.
Examples: Gmail, Google Docs, Salesforce.

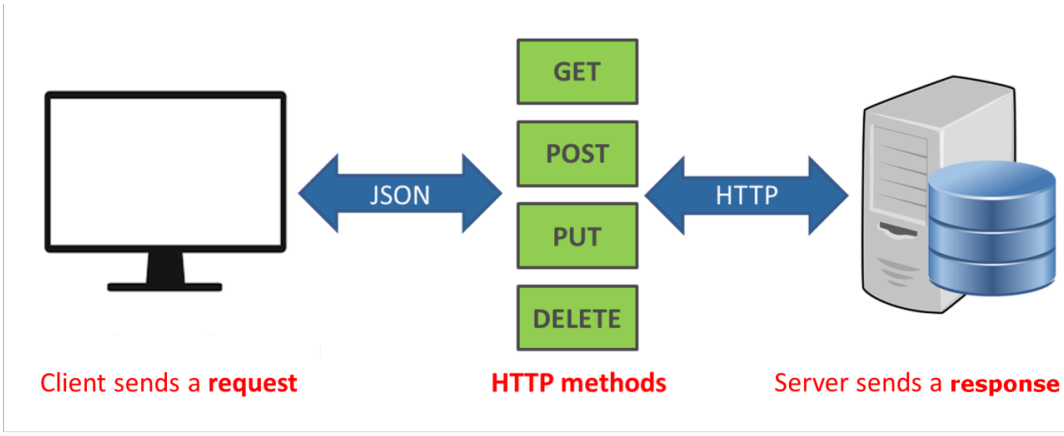
🔹 Real Use Case (Very Common in Interviews)

“You deploy your microservice on AWS EC2. User uploads are stored in S3. Caching uses Redis on ElastiCache. Logs are stored in CloudWatch.”
This entire workflow is **cloud computing**.

🌐 2. What Is a RESTful Service (REST API)?







Simple definition:

A RESTful service is an API that uses **HTTP** to allow different systems to communicate over the internet using standard operations like:

- **GET** → read
- **POST** → create
- **PUT** → update

- **DELETE** → remove

🔹 Principles of REST (easy memory)

1. **Client–Server** → UI and backend separate
2. **Stateless** → server does not store client session
3. **Uniform interface** → predictable request patterns
4. **Cacheable** → responses can be cached
5. **Layered system** → API → Load Balancer → Backend → DB

🔹 Example REST API

Request:

```
GET /users/123
```

Response:

```
{
  "id": 123,
  "name": "Raj"
}
```

This is a **RESTful service** returning user info.

🔹 Why REST APIs are important in system design?

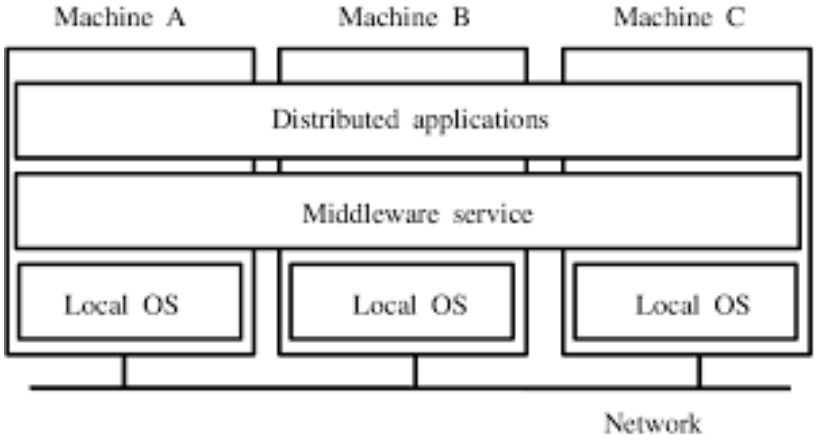
- All microservices communicate via REST or gRPC

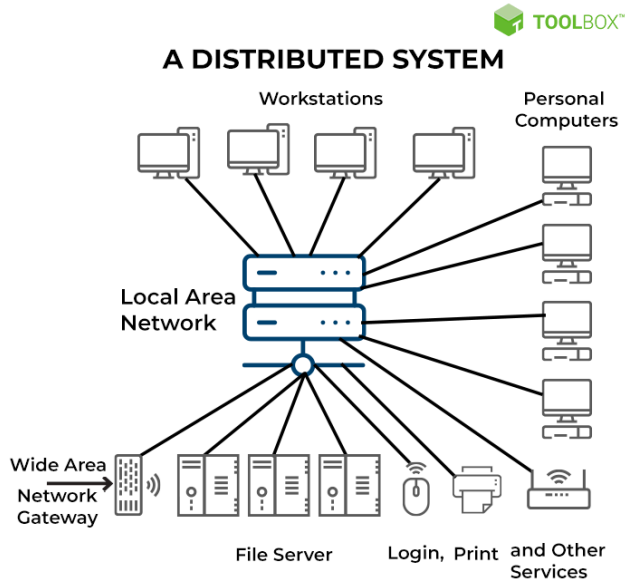
- Powers mobile apps, web apps, IoT
- Acts as the entry point for load balancers, gateways
- Statelessness helps horizontal scaling

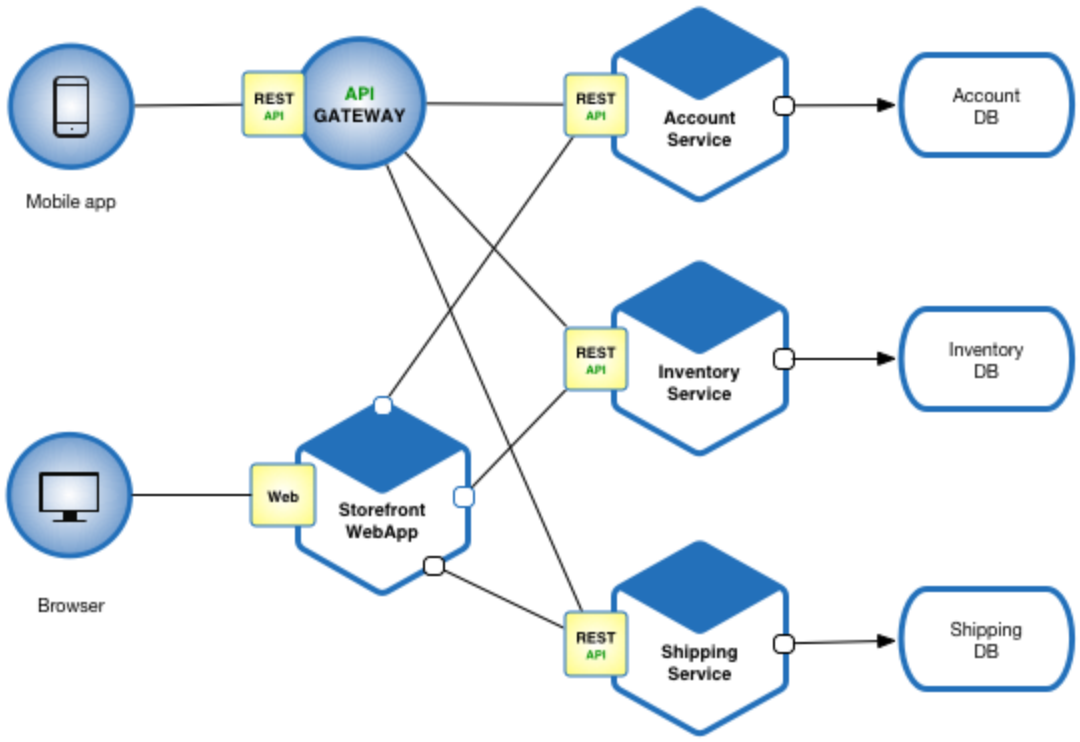
Interviewers often ask:

> “How would you design a REST API for uploading files or fetching feeds?”

🕸️ 3. What Is a Distributed System?







Simple definition:

A distributed system is **a collection of multiple computers (nodes)** working together to appear like **one system** to the user.

Examples:

- Google Search
- Netflix streaming
- Amazon e-commerce
- Apache Kafka clusters
- Microservices architecture

🔹 Key Characteristics

1. Multiple machines working together

Not one big server — many small servers.

2. Communicate over a network

Nodes talk to each other using TCP, HTTP, RPC.

3. Handle failures gracefully

Some machines will fail → system must continue running.

4. Provide scalability

Add more servers to handle more traffic.

🔹 Why Distributed Systems Exist

A single machine cannot handle:

- Millions of users
- Petabytes of data
- Global latency requirements

So companies **distribute**:

- computation
- storage
- traffic
- caching
- messaging

across many machines.

🔹 Common concepts inside distributed systems

- **Load balancing**
- **Replication**
- **Sharding**
- **Caching**
- **Consensus (Raft, Paxos)**
- **Distributed transactions**
- **Event streaming systems (Kafka)**
- **Microservices**

🎯 All Three Concepts Connected (Interview Summary)

Concept	Simple Meaning	Why It Matters
Cloud Services	Renting servers & managed services from AWS/GCP/Azure	Fast deployment, scaling, global availability
RESTful Service	APIs using HTTP + JSON for communication	Microservice communication, client/server interaction
Distributed System	Many machines acting as one system	Large scale, reliability, fault tolerance

🌱 What Are TCP and UDP?

TCP and UDP are **transport layer protocols (Layer 4 in OSI)** responsible for sending data between two machines over the internet.

They decide **HOW** data is delivered—not what the data contains.

🟩 TCP (Transmission Control Protocol)

Simple Definition:

TCP is a reliable, connection-oriented protocol that guarantees your data arrives **completely, correctly, and in order.**

It behaves like sending a parcel via a trusted courier who:

- checks your address
- confirms delivery
- resends if lost
- keeps track of order

✔ Key Features of TCP

1. Reliable Delivery

If packets get lost, TCP **detects it** and **retransmits** them.

2. Ordered Data

TCP reorders packets at the receiver so the application gets them in the correct sequence.

3. Connection-oriented (3-way handshake)

Before sending data, TCP establishes a connection:

1. SYN

2. SYN–ACK
3. ACK

Only then communication starts.

This ensures both sides are ready.

4. Congestion Control

TCP slows down or speeds up depending on network conditions to prevent overload.

5. Flow Control

Ensures receiver is not overwhelmed.

🟦 Where TCP Is Used

Because it is reliable and ordered, TCP is used in protocols where **accuracy matters**:

- **HTTP / HTTPS** (websites, REST APIs)
- **gRPC**
- **WebSockets**
- **SSH**
- **Email (SMTP)**
- **File transfer (FTP)**

Anything that requires correctness uses TCP.

📌 Real system design relevance

TCP matters when discussing:

- long-lived connections (WebSockets, gRPC)
- connection pooling
- slow clients
- applying backpressure
- timeouts and retries

🟥 UDP (User Datagram Protocol)

Simple Definition:

UDP is an unreliable, connectionless protocol that focuses on **speed and low latency**, not perfection.

It behaves like shouting a message—fast, but not guaranteed to be heard or remembered.

✔ Key Features of UDP

1. No reliability

UDP does *not* retry lost packets.

2. No ordering

If packets come out of order, UDP doesn't fix it.

3. No connection setup

No handshake, no session.
Just send packets immediately → **super fast**.

4. Very low overhead

Because there are no checks, retries, or congestion control, UDP has minimal latency.

🟦 Where UDP Is Used

UDP is used where **speed is more important than accuracy**:

1. Real-time streaming

- Video calls
- Online classes
- Live sports steaming
  A few lost packets don't matter—delay would matter more.

2. Gaming

Fast movement updates must arrive instantly, not perfectly.

3. DNS

DNS queries are tiny and can be retried easily.

4. IoT & sensor data

📌 Real system design relevance

UDP is important when selecting protocol for:

- real-time push
- low-latency systems
- CDN streaming
- VoIP

Interviewers may ask:

> "Why isn't video streaming done over TCP?"

Answer: TCP retransmissions cause lag; UDP tolerates packet loss for smoother real-time playback.

🔥 TCP vs UDP — Quick Comparison Table

Feature	TCP	UDP
Reliability	✔ Guaranteed	✖ Not guaranteed
Packet order	✔ Maintains order	✖ No ordering
Speed	Slower	Very fast
Connection	3-way handshake	No connection
Overhead	High	Low
Use cases	Web, APIs, file transfer	Streaming, gaming, DNS

🎯 System Design Interview Summary

Use TCP when:

- correctness matters
- order matters
- data transfer is heavy
  Examples: REST APIs, Kafka client-broker, DB connection

Use UDP when:

- speed matters
- small data packets

- occasional loss is acceptable
  Examples: video calls, games, real-time streams