



System Design

scalability · databases · caching · APIs · real case studies

Notes by Sandeep Kumar Jha · Senior Full Stack Developer

Revision notes · handwritten style · based on the Gate Smashers System Design playlist (36 lectures)

Part of my interview-revision series

Contents

- 1 · Syllabus, approach & HLD vs LLD (L1, L22)
- 2 · What happens when you open a website (L2, L8)
- 3 · Load Balancer (L3)
- 4 · CDN (L4)
- 5 · Forward vs Reverse Proxy (L5)
- 6 · SSL/TLS Certificates & HTTPS (L6)
- 7 · Caching + Write-through vs Write-back (L7, L16)
- 8 · APIs, API types, SDK, FastAPI (L9, L26, L28, L36)
- 9 · Horizontal vs Vertical Scaling (L14)
- 10 · Partitioning, Sharding, Replication (L10, L11, L24)
- 11 · CAP Theorem · ACID vs BASE (L12, L23)
- 12 · SQL vs NoSQL + 4 NoSQL types (L17-L20, L33)
- 13 · Message Queues (L13)
- 14 · Microservices (L15)
- 15 · HTTP vs WebSocket (L25)
- 16 · Rate Limiting (L27)
- 17 · Security layers · JWT · Session vs JWT (L21, L31, L32)
- 18 · SOLID Principles (L34)
- 19 · Case study — Instagram (L29)
- 20 · Case study — IRCTC Tatkal (L30)
- 21 · Case study — UPI (L35)
- 22 · Rapid-fire Q&A + cheat sheet

How to use these notes

-  Read a chapter, cover it, then **draw the diagram from memory** — system design is drawn, not memorised.
-  Yellow = remember. Orange box = revision / interview. Red = common mistake. Green dashed = where it shows up in my own work.
-  Lecture numbers (L1...L36) map to the Gate Smashers playlist order so I can re-watch the exact video.

1

Syllabus, approach & HLD vs LLD

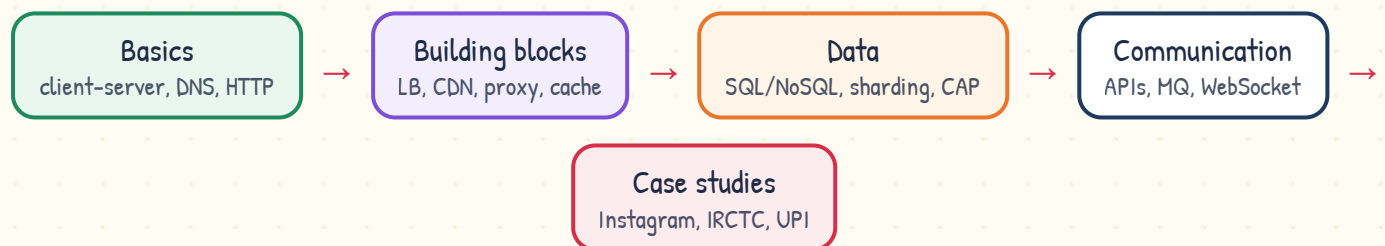
Definition

System design = deciding the architecture, components, data flow and interfaces of a system so it meets its requirements — and keeps meeting them when users grow from 100 to 100 million.

Functional vs Non-functional

Type	Meaning	Instagram example
Functional	WHAT the system does — features.	Upload photo, follow, like, see feed.
Non-functional	HOW WELL it does it — scalability, availability, latency, consistency, reliability, security, cost.	Feed loads < 200 ms, 99.99% uptime, photos never lost.

Roadmap of the course



HLD vs LLD (L22)

Point	HLD — High Level Design	LLD — Low Level Design
View	Bird's-eye: <u>which boxes & how they connect</u>	Zoomed-in: <u>what's inside each box</u>
Contains	Services, DBs, cache, LB, CDN, queues, data flow	Classes, interfaces, methods, DB schema, design patterns
Output	Architecture diagram	Class / sequence / ER diagrams
Made by	Architects, senior devs	Developers
Analogy	Building's floor plan	Wiring & plumbing of each room

My 7-step interview approach

1. **Clarify requirements** — functional + non-functional. Ask, don't assume.
2. **Estimate** — users, QPS, storage, bandwidth (back-of-envelope).
3. **API design** — main endpoints.
4. **Data model** — entities, SQL or NoSQL.
5. **HLD diagram** — client → LB → services → cache → DB.
6. **Deep dive** — bottlenecks, scaling, single points of failure.
7. **Trade-offs** — say what I gave up and why.

back-of-envelope

// 1 day = 86,400 s $\approx 10^5$ s (easy mental maths)

10M DAU $\times$ 10 requests/day = 10^8 req/day

avg QPS = $10^8 / 10^5 \approx 1,000$ QPS

peak QPS $\approx 2-3 \times$ avg $\approx 2,500$ QPS

storage = 1M photos/day $\times$ 500 KB ≈ 500 GB/day ≈ 180 TB/year

Revision

Requirements first (functional + non-functional) → estimate → API → data → HLD → deep dive → trade-offs. HLD = boxes & arrows; LLD = classes & schema.

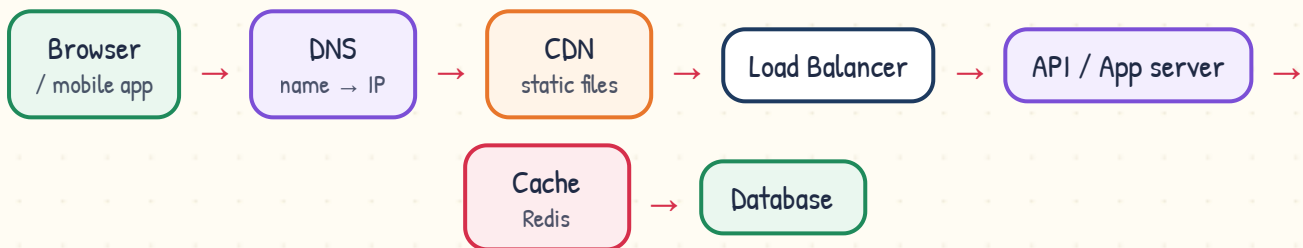
2

What happens when you open a website?

Big picture

Typing `www.example.com` triggers: `DNS lookup` → `TCP connect` → `TLS handshake` → `HTTP request` → `servers` → `response` → `browser renders`. This one flow touches almost every topic in this course.

Full stack request flow (L8)



→ response travels back the same way, browser paints the page

Step by step

1. **Browser cache check** — already have the page / IP? use it.
2. **DNS resolution**: browser cache → OS cache (hosts file) → `recursive resolver` (ISP / 8.8.8.8) → `Root server` → `TLD server` (.com) → `Authoritative server` → returns IP. Cached for its TTL.
3. **TCP 3-way handshake**: SYN → SYN-ACK → ACK.
4. **TLS handshake** (HTTPS): verify certificate, agree a session key (Ch 6).
5. **HTTP request**: `GET / HTTP/1.1` + headers + cookies.
6. Static assets come from the **CDN**; dynamic request goes to **LB** → **app server**.
7. App server checks **cache**; on miss queries the **DB**, fills cache, builds response.
8. **Browser renders**: parse HTML → DOM, CSS → CSSOM, run JS → layout → paint.

DNS records

Record	Meaning
A / AAAA	Domain → IPv4 / IPv6 address
CNAME	Alias: one name → another name (www → example.com, or → CDN host)
MX	Mail server for the domain
NS	Which name servers are authoritative
TXT	Verification, SPF/DKIM for email

Mobile app difference

The app already has its UI installed, so it skips HTML rendering — it only calls APIs that return JSON, then draws native screens. DNS/TCP/TLS/LB/server/DB steps are the same.

Mistake

"DNS is queried on every request" — no, answers are cached at many levels until the **TTL** expires. That's also why DNS changes take time to "propagate".

Revision

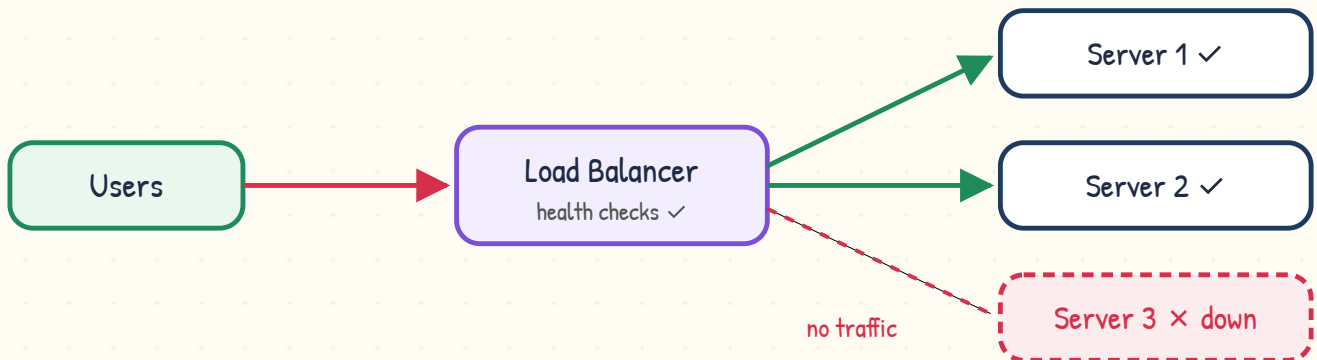
DNS (root → TLD → authoritative) → TCP → TLS → HTTP → CDN / LB → app → cache → DB → render. Be able to draw it in 30 seconds.

3

Load Balancer

Definition

A load balancer sits in front of a group of servers and spreads incoming requests across them, so no single server is overloaded and a dead server gets no traffic. Real life: a **traffic police officer** sending cars to free lanes / a bank token system sending you to a free counter.



Why use one?

- ✎ Scalability — add servers behind it (horizontal scaling).
- ✎ High availability — health checks remove dead servers automatically.
- ✎ No single server is a SPOF; zero-downtime deploys (drain one server at a time).
- ✎ Can also do SSL termination, compression, caching (it's a reverse proxy — Ch 5).

Algorithms (interview favourite)

Algorithm	How	Good for
Round Robin	1 → 2 → 3 → 1 → 2 ... in turn	Equal servers, similar requests
Weighted RR	Bigger server gets more turns (weights 3:1)	Mixed server sizes
Least Connections	Send to server with fewest active connections	Long / uneven requests
Least Response Time	Fewest connections + fastest response	Latency-sensitive
IP Hash	<code>hash(client IP) % N</code> → same user, same server	Sticky sessions
Random	Pick any healthy server	Simple, large pools

L4 vs L7

Layer 4 (transport)

Looks only at IP + port (TCP/UDP). Very fast, can't read URLs. e.g. AWS NLB.

Layer 7 (application)

Reads HTTP: URL, headers, cookies. Can route `/api` → API servers, `/images` → media. e.g. Nginx, AWS ALB.

nginx.conf

```
upstream app_servers {  
    least_conn;                # algorithm  
    server 10.0.0.1:3000 weight=3;  
    server 10.0.0.2:3000;  
    server 10.0.0.3:3000 backup;  
}  
server {  
    listen 80;  
    location / { proxy_pass http://app_servers; }  
}
```

Mistake

The load balancer itself can become the **single point of failure**. Fix: two LBs (active-passive with a floating IP) or a managed LB. Also: storing sessions in server memory breaks when the LB sends the user elsewhere → keep servers **stateless**, sessions in Redis.

Revision

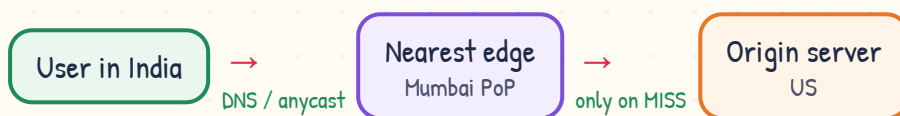
LB = spread traffic + health checks. Algorithms: RR, weighted, least-conn, least-time, IP-hash. L4 = IP/port, L7 = HTTP-aware. Make the LB redundant too.

CDN – Content Delivery Network

Definition

A **CDN** is a network of **edge servers spread across the world** that keep copies of content (images, videos, JS, CSS) **close to users**. A user in Delhi gets files from a Delhi/Mumbai edge instead of the origin in the US. Real life: **Amazon warehouses in every city** instead of one big warehouse.

How it works



1. DNS (often a CNAME to the CDN) sends the user to the **nearest edge** (PoP = Point of Presence).
2. **Cache HIT** → edge serves the file instantly.
3. **Cache MISS** → edge fetches from origin, stores it for the **TTL**, then serves. Next user nearby gets a HIT.

Push vs Pull CDN

Type	How	Use when
Pull	Edge pulls from origin on first request (lazy)	Most websites; frequently changing, high traffic
Push	I upload content to the CDN in advance	Large, rarely changing files (video library, game patches)

Benefits

- ✎ **Lower latency** – shorter physical distance.
- ✎ **Less load on origin** + lower bandwidth bill.
- ✎ Absorbs traffic spikes and **DDoS** (huge distributed capacity).
- ✎ **Availability** – if one edge fails, another serves.

Cache invalidation

- ✎ TTL via **Cache-Control: public, max-age=31536000**.
- ✎ **Purge API** to delete a file from all edges.

✎ Best: **versioned file names** — `app.3f2a9c.js`. New deploy = new name = no stale file.

Mistake

Caching **personalised** responses (my cart, my profile) on a public CDN → other users may see them! Use **Cache-Control: private, no-store** for user-specific data.

Revision

CDN = edge copies near users. HIT → serve, MISS → fetch origin + cache for TTL. Pull (lazy) vs Push (upload).
Examples: Cloudflare, Akamai, AWS CloudFront.

5

Forward Proxy vs Reverse Proxy

Definition

A proxy is a middleman server. **Forward proxy** acts on behalf of clients; **reverse proxy** acts on behalf of servers.



server sees the proxy's IP, not mine

client sees only the proxy, servers hidden

Point	Forward proxy	Reverse proxy
Protects	Client identity	Server identity
Set up by	Client side (office, college, user)	Server side (website owner)
Uses	Privacy / hide IP, content filtering (college blocks sites), caching for many users, bypass geo-blocks	Load balancing, SSL termination, caching, compression, WAF / security, hide internal servers
Examples	Squid, corporate proxy, VPN-like services	Nginx, HAProxy, Cloudflare, AWS ALB
Real life	Friend buys a movie ticket for me — cinema never sees me	Company receptionist — callers never reach employees directly

nginx as reverse proxy + SSL termination

```
server {
    listen 443 ssl;
    server_name api.example.com;
    ssl_certificate      /etc/ssl/fullchain.pem;
    ssl_certificate_key  /etc/ssl/privkey.pem;
    location / {
        proxy_pass http://localhost:3000;           # app stays on plain HTTP inside
        proxy_set_header X-Forwarded-For $remote_addr; # real client IP
    }
}
```

Every **load balancer** is a **reverse proxy**, but not every **reverse proxy** is a **load balancer** (a reverse proxy can sit in front of just one server for SSL + caching).

Revision

Forward = for clients (hide client, filter). Reverse = for servers (LB, SSL, cache, security, hide servers). Remember who is hidden.

SSL/TLS Certificates & HTTPS

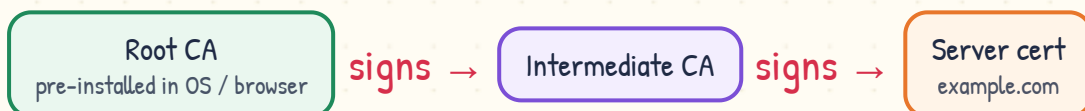
Definition

HTTPS = HTTP + TLS (SSL is the old name). TLS gives 3 things: **Encryption** (nobody can read), **Integrity** (nobody can modify), **Authentication** (I'm really talking to bank.com). An **SSL certificate** is the website's digital ID card, signed by a trusted **Certificate Authority (CA)**.

What's inside a certificate

Domain name(s) · organisation · **public key** · issuer (CA) · validity dates · CA's **digital signature**.

Chain of trust



TLS handshake (simplified)

1. **ClientHello** — TLS versions, cipher suites, random number.
2. **ServerHello + Certificate** — chosen cipher, server's certificate (public key).
3. Browser **verifies** the cert: signed by a trusted CA? domain matches? not expired?
4. **Key exchange** (ECDHE / RSA) using **asymmetric** crypto → both sides derive the same **session key**.
5. All data now encrypted with fast **symmetric** encryption (AES) using that session key.

Point	Asymmetric	Symmetric
Keys	Public + private pair	One shared secret key
Speed	Slow	Fast
Used for	Handshake: identity + agreeing a key	Bulk data after handshake
Examples	RSA, ECDHE	AES, ChaCha20

Certificate types

- **DV** (domain validated, e.g. Let's Encrypt — free, 90 days) · **OV** (organisation checked) · **EV** (extended checks).
- **Wildcard** `*.example.com` · **SAN / multi-domain** — many names in one cert.

```
# free cert + auto nginx config + auto-renew
sudo certbot --nginx -d example.com -d www.example.com
# inspect any site's certificate chain
openssl s_client -connect example.com:443 -showcerts
```

Mistake

"Padlock = safe website" — no. HTTPS only proves the connection is encrypted to *that* domain. A phishing site can have a valid certificate too. Also don't forget **renewal** — expired certs take sites down.

Revision

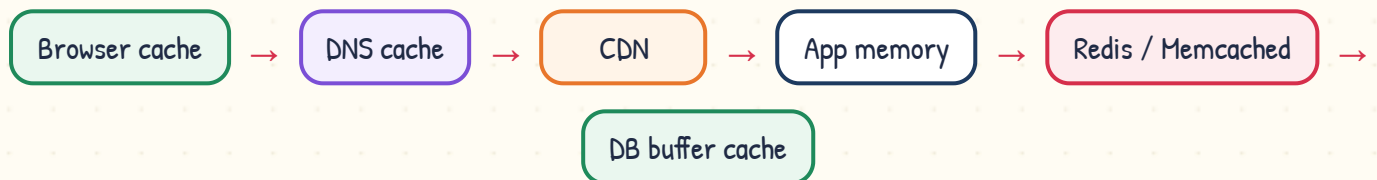
HTTPS = HTTP + TLS → encryption, integrity, authentication. Cert = domain + public key + CA signature. Handshake uses asymmetric to agree a key, then symmetric for data.

Caching + Write-through vs Write-back

Definition

Caching = storing frequently used data in a **faster layer** (usually RAM) so we don't hit the slow source (DB / disk / network) every time. Real life: keeping a **water bottle on my desk** instead of walking to the kitchen each time.

Where caching happens



- Cache hit = found in cache. Miss = go to source. Hit ratio = hits / total — aim high (90%+).
- Cache what is read often, changes rarely: product pages, user profile, config, sessions, results of heavy queries.

Read strategy – Cache-aside (most common)

cache-aside with Redis (Node)

```

async function getUser(id) {
  const key = `user:${id}`;
  const hit = await redis.get(key);
  if (hit) return JSON.parse(hit); // 1. HIT → fast path

  const user = await db.users.findById(id); // 2. MISS → DB
  await redis.set(key, JSON.stringify(user), 'EX', 3600); // 3. fill, TTL 1h
  return user;
}
// on update: write DB, then redis.del(key) so next read refreshes
  
```

Write strategies (L16)

Strategy	How	Pros	Cons
Write-through	Write to cache AND DB together ; ack after both	Cache always consistent, no data loss	Slower writes (2 writes every time)
Write-back (write-behind)	Write to cache only , ack; flush to DB later in batches	Very fast writes, fewer DB hits	Data loss if cache crashes before flush
Write-around	Write to DB only ; cache filled on next read	Cache not polluted with write-once data	First read after write is a miss

Choose: **bank balance** → write-through (can't lose data). **View counts / likes / analytics** → write-back (speed matters, tiny loss is OK). **Logs / bulk uploads** → write-around.

Eviction policies (cache is small!)

 **LRU** — remove Least Recently Used (most common). **LFU** — Least Frequently Used. **FIFO** — oldest first. **TTL** — expire after time.

Cache problems

Problem	What	Fix
Stale data	DB changed, cache still old	TTL + delete key on update
Stampede / thundering herd	Hot key expires → 1000s of requests hit DB at once	Lock / single-flight, random TTL jitter
Penetration	Requests for keys that don't exist always miss	Cache "null" briefly, Bloom filter
Avalanche	Many keys expire at the same moment	Spread TTLs, warm cache



From my work

TanStack / React Query on the frontend is a client-side cache (stale-while-revalidate) — same ideas: cache key, stale time, invalidation after a mutation.

Revision

Cache = fast copy. Cache-aside for reads. Write-through = safe, write-back = fast but risky, write-around = skip cache. Evict with LRU. Watch stale data + stampede.

Definition

An **API** (Application Programming Interface) is a **contract** that defines how one piece of software can ask another for data or actions — without knowing its internals. Real life: the **waiter in a restaurant**: I (client) order from the menu (API docs), the waiter takes it to the kitchen (server) and brings back food (response).

REST – the common one

Verb	Action
GET	Read <code>/users/7</code>
POST	Create <code>/users</code>
PUT / PATCH	Replace / partial update
DELETE	Remove

Code	Meaning
200 / 201	OK / Created
400 / 401 / 403 / 404	Bad request / not logged in / not allowed / not found
429	Too many requests
500 / 503	Server error / unavailable

- REST = **resources as URLs + HTTP verbs + stateless** requests, usually JSON.
- Idempotent** = calling it many times has the same effect as once: GET, PUT, DELETE ✓ · POST × (use an idempotency key for payments).
- Good API: versioning `/v1/`, pagination, filtering, consistent error format, auth, rate limits, docs (OpenAPI/Swagger).

All API types in one table (L28)

Type	Format / transport	Key idea	Use when
REST	JSON over HTTP	Resources + verbs, simple, cacheable	Public & web/mobile APIs
SOAP	XML over HTTP (WSDL contract)	Strict, built-in standards (WS-Security)	Banks, legacy enterprise
GraphQL	Query language, one endpoint	Client asks exactly the fields it needs — no over/under-fetching	Complex UIs, many clients
gRPC	Protobuf (binary) over HTTP/2	Very fast, typed contracts, streaming	Service ↔ service (microservices)
WebSocket	Persistent TCP connection	Two-way, real-time push	Chat, live scores (Ch 15)

API vs SDK (L26)

Point	API	SDK (Software Development Kit)
What	The interface / contract (endpoints)	A toolkit : libraries, docs, samples, tools — often wraps the API
Analogy	The electric socket	The whole toolbox with ready-made plugs
Example	<code>POST /v1/payments</code>	<code>razorpay.orders.create()</code> , AWS SDK, Firebase SDK

From my work

Amazon Chime SDK for doctor video consultations (SDK); Zoho Books API integration + cron sync in the event platform (API); PayU payment flow with server-side hash verification.

FastAPI in 5 minutes (L36)

- Modern **Python** web framework for APIs, built on **Starlette** (async web) + **Pydantic** (validation).
- Type hints → automatic validation + auto docs at `/docs` (Swagger) and `/redoc`.
- `async def` support → high performance; dependency injection with `Depends`.

main.py — run: uvicorn main:app --reload

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class User(BaseModel):
    name: str
    age: int # "abc" → automatic 422 error

@app.get("/users/{user_id}")
async def get_user(user_id: int):
    if user_id != 1: raise HTTPException(404, "Not found")
    return {"id": user_id, "name": "Sandeep"}

@app.post("/users", status_code=201)
async def create_user(user: User):
    return user
```

Revision

API = contract. REST (JSON, verbs, stateless), SOAP (XML, strict), GraphQL (ask exact fields), gRPC (binary, fast, internal), WebSocket (real-time). SDK = toolkit around an API. FastAPI = Python + type hints + auto docs.

Horizontal vs Vertical Scaling

Definition

Scalability = ability to handle more load by adding resources. **Vertical** = make one machine bigger (scale up).

Horizontal = add more machines (scale out).

Vertical ↑

4 GB → 64 GB RAM, 2 → 32 CPUs.

Real life: replace a car with a bigger truck.

Horizontal →

1 server → 10 servers behind a load balancer.

Real life: **more trucks** in the fleet.

Point	Vertical	Horizontal
How	Upgrade CPU / RAM / disk	Add servers + load balancer
Limit	<u>Hardware ceiling</u>	Practically unlimited
Failure	Single point of failure	Fault tolerant — one dies, others serve
Downtime	Often needs restart	Add/remove with no downtime
Complexity	Simple, no code change	Needs LB, stateless apps, data distribution
Cost	High-end hardware gets expensive fast	Commodity machines, pay as you grow
Data consistency	Easy (one machine)	Harder (distributed)
Good for	Early stage, relational DB primary	Web/app tier, large scale

- Horizontal needs stateless servers: no session in memory → put sessions in Redis or use JWT.
- Auto-scaling**: cloud adds/removes servers based on CPU, request count (AWS Auto Scaling Group, Kubernetes HPA).
- Real systems use **both**: scale up a DB first, then scale out with replicas / shards (Ch 10).

Mistake

"Horizontal is always better" — no. It brings network calls, consistency issues and ops overhead. Start simple (vertical) and scale out when needed.

Revision

Vertical = bigger box (simple, limit, SPOF). Horizontal = more boxes (LB, stateless, fault tolerant, complex). Use both.

Definitions

Partitioning = splitting a big table into smaller pieces (can be on the **same** server).

Sharding = horizontal partitioning where pieces live on **different servers** (shards).

Replication = keeping **copies of the same data** on multiple servers.

Horizontal vs Vertical partitioning (L10)

Horizontal (by rows)

Users 1-1M in P1, 1M-2M in P2. Same columns, different rows.

Vertical (by columns)

users(id, name, email) + users_profile(id, bio, photo).
Split rarely used / heavy columns.

Partitioning strategies

Strategy	How	Problem
Range	By range: dates, IDs, A-M / N-Z	Hotspots (all new orders hit latest range)
Hash	<code>hash(user_id) % N</code>	Even spread, but adding a shard reshuffles most keys
List	By category: region = India / US / EU	Uneven sizes
Directory	Lookup table: key → shard	Lookup service is extra hop / SPOF
Consistent hashing	Keys & servers on a ring; key goes to next server clockwise	Adding a server moves only ~1/N keys ✓

MySQL range partition (same server)

```
CREATE TABLE orders (
  id BIGINT, created_at DATE, amount DECIMAL(10,2)
) PARTITION BY RANGE (YEAR(created_at)) (
  PARTITION p2024 VALUES LESS THAN (2025),
  PARTITION p2025 VALUES LESS THAN (2026),
  PARTITION pmax VALUES LESS THAN MAXVALUE
); -- WHERE created_at in 2025 → scans only p2025 (partition pruning)
```

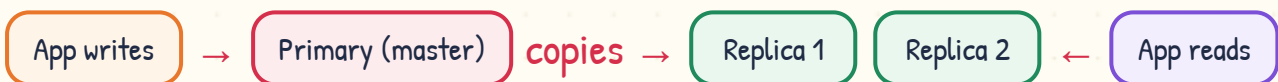
Partitioning vs Sharding (L11)

Point	Partitioning	Sharding
Where	Usually one DB server	Many DB servers
Goal	Faster queries, easier maintenance	Scale writes + storage beyond one machine
Complexity	Handled by DB	App/router must know which shard
Real life	Separate shelves in one library	Separate library branches in each city

Sharding pain points

Bad **shard key** → hotspots (celebrity user). **Cross-shard joins** & transactions are hard. **Resharding** is painful → use consistent hashing. Shard only when you must.

Replication



- Primary-replica:** writes to primary, reads from replicas → scales **reads**, gives failover.
- Multi-primary:** writes on many nodes → conflict resolution needed.
- Sync** = wait for replica (consistent, slower). **Async** = don't wait (fast, **replication lag** → stale reads).

Replication vs Sharding (L24)

Point	Replication	Sharding
Data	Same full copy everywhere	Each shard has a different subset
Solves	Read scaling, high availability, backup	Write scaling, huge data size
Storage	Multiplies storage	Divides storage
Together	Real systems shard the data and replicate each shard (e.g. MongoDB sharded cluster of replica sets).	

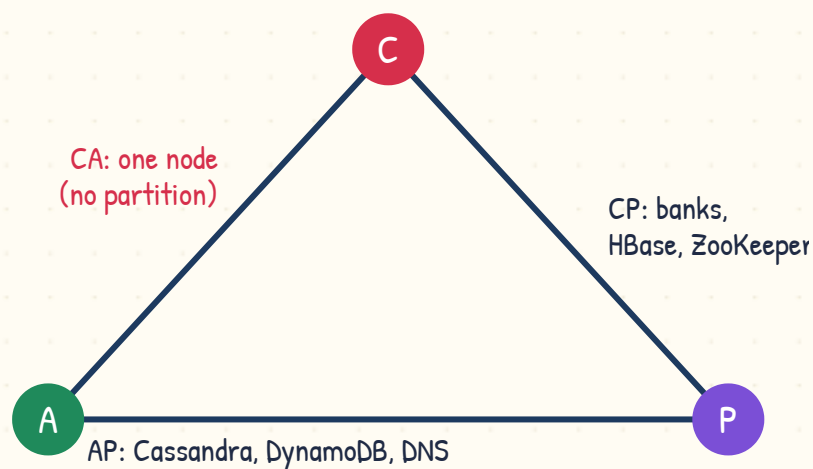
Revision

Partition = split (same server). Shard = split across servers (scale writes). Replicate = copy (scale reads + HA). Pick shard key carefully; consistent hashing for resharding.

CAP theorem

In a distributed system, when a network partition happens you can guarantee only one of: **Consistency** or **Availability**.

- ✎ **Consistency** — every read gets the latest write (all nodes show the same data).
- ✎ **Availability** — every request gets a (non-error) response, even if not the latest.
- ✎ **Partition tolerance** — system keeps working when messages between nodes are lost.



Choice	During a partition...	Use for
CP	Reject / block requests rather than return stale data	Bank balance, payments, inventory/booking
AP	Answer anyway, may be stale; sync later (eventual consistency)	Social feed, likes, product reviews, DNS

Mistake

"Pick any 2 of 3" — in a real distributed system **P is not optional** (networks fail). The real choice is **C vs A** when a partition happens. Bonus: **PACELC** — Else (no partition) you trade **Latency vs Consistency**.

ACID (SQL transactions)

- ✎ **Atomicity** — all or nothing. **Consistency** — rules/constraints always hold. **Isolation** — concurrent transactions don't interfere. **Durability** — committed = survives crash.

```
START TRANSACTION;
  UPDATE accounts SET balance = balance - 500 WHERE id = 1;
  UPDATE accounts SET balance = balance + 500 WHERE id = 2;
COMMIT;    -- any failure → ROLLBACK, money never disappears
```

BASE (many NoSQL)

 Basically Available · Soft state (may change without input as replicas sync) · Eventual consistency (all replicas agree... eventually).

Point	ACID	BASE
Priority	Correctness, strong consistency	Availability, scale, speed
Typical DB	MySQL, PostgreSQL, Oracle	Cassandra, DynamoDB, CouchDB
Scaling	Mostly vertical (harder to distribute)	Horizontal by design
Example	Payments, orders, ledgers	Likes count, feeds, analytics

Revision

CAP: partition will happen → choose C (CP) or A (AP). ACID = strict transactions (SQL). BASE = available + eventually consistent (NoSQL). Money → ACID/CP; social → BASE/AP.

Point	SQL (relational)	NoSQL
Structure	Tables, rows, fixed schema	Documents, key-value, columns, graphs – flexible schema
Relations	JOINS, foreign keys	Usually denormalised / embedded
Transactions	Strong ACID	Often BASE (many now offer ACID per document)
Scaling	Vertical first; sharding is hard	Horizontal by design
Query	SQL (standard)	DB-specific APIs
Examples	MySQL, PostgreSQL, Oracle	MongoDB, Redis, Cassandra, Neo4j
Pick when	Structured data, complex queries, money	Huge scale, fast-changing shape, high write volume

1 · Document DB (L17)

Stores **JSON-like documents**; each can have different fields; nested data in one read.

- ✎ MongoDB, CouchDB, Firestore
- ✎ Use: product catalog, CMS, user profiles

MongoDB doc

```
{ _id: "p101", name: "iPhone 16",
  price: 79900,
  specs: { ram: "8GB", color: "black" },
  tags: ["phone", "apple"] }
```

2 · Key-Value DB (L18)

Simplest: **key** → **value**, like a giant hash map. O(1) lookups, often in-memory.

- ✎ Redis, DynamoDB, Memcached
- ✎ Use: cache, sessions, shopping cart, rate-limit counters, leaderboards

redis-cli

```
SET session:abc123 "user:7" EX 1800
GET session:abc123
INCR page:views:home
ZADD leaderboard 980 "sandeep"
```

3 · Column-Family DB (L19)

Rows identified by a key, columns grouped in **families**; each row can have different columns (wide rows). Data stored by column → great for **huge write volume**.

- ✎ Cassandra, HBase, ScyllaDB
- ✎ Use: time-series, IoT sensors, messages, activity logs

Cassandra CQL

```
CREATE TABLE messages (
  chat_id uuid, sent_at timeuuid,
  sender text, body text,
  PRIMARY KEY (chat_id, sent_at)
) WITH CLUSTERING ORDER BY (sent_at DESC);
```

4 · Graph DB (L20)

Data as **nodes (entities) + edges (relationships)**. Following relationships is fast — no expensive multi-JOINs.

- ✎ **Neo4j**, Amazon Neptune
- ✎ Use: social network ("friends of friends"), recommendations, fraud detection, maps

Neo4j Cypher

```
// friends-of-friends I don't follow yet
MATCH (me:User {name:'Sandeep'})
  -[:FOLLOWS]→()-[:FOLLOWS]→(f)
WHERE NOT (me)-[:FOLLOWS]→(f)
RETURN f.name LIMIT 10;
```

Decision rule: **money / relations / reports** → SQL. Flexible docs → MongoDB. Speed / cache → Redis. Massive writes → Cassandra. Relationships → Neo4j. Big systems use **several** (polyglot persistence).

From my work

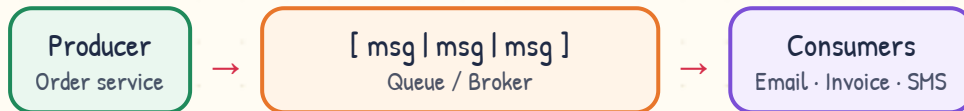
MySQL behind the Livia HMIS NestJS services (relational: patients, appointments, billing need ACID); MongoDB in the Cold Storage tool (flexible calculation inputs).

Revision

SQL = schema + JOIN + ACID. NoSQL = flexible + horizontal. Document (Mongo), Key-value (Redis), Column-family (Cassandra), Graph (Neo4j) — know one use case each.

Definition

A message queue lets services talk asynchronously: a **producer** puts a message in the queue and moves on; a **consumer** picks it up and processes it later. Real life: **restaurant order slips** — the waiter pins the order and goes back to customers; the kitchen cooks at its own pace.



Why use it?

- Decoupling — producer doesn't know/care who consumes.
- Buffer spikes — 10k orders/sec queue up; consumers process at their own speed.
- Async heavy work — user gets "Order placed ✓" instantly; email, PDF invoice, notifications happen in background.
- Reliability — if a consumer is down, messages wait; retries on failure.

Two models

Point-to-point

Each message is consumed by **one** consumer (work queue). e.g. resize image jobs.

Publish / Subscribe

Message goes to a **topic**; **every** subscriber gets a copy. e.g. "OrderPlaced" → email + analytics + inventory.

Delivery guarantees

Guarantee	Meaning
At-most-once	May be lost, never duplicated (fire and forget)
At-least-once	Never lost, <u>may be duplicated</u> → consumers must be idempotent (most common)
Exactly-once	Hard & costly; approximated with idempotency keys / transactions

- Dead Letter Queue (DLQ) — messages that keep failing after N retries go here for inspection, so they don't block the queue.
- Ack — consumer acknowledges after success; un-acked messages are redelivered.

<i>Tool</i>	<i>Style</i>	<i>Best for</i>
Kafka	Distributed append-only log , partitions, messages kept for days, replayable	Event streaming, very high throughput, analytics
RabbitMQ	Classic broker with exchanges & routing	Task queues, complex routing
AWS SQS / SNS	Managed queue / pub-sub	No-ops cloud apps

BullMQ (Redis-backed) in Node

```
// producer - inside the order API
await emailQueue.add('order-confirmation', { orderId, email },
  { attempts: 3, backoff: { type: 'exponential', delay: 2000 } });
return res.status(201).json({ orderId }); // respond immediately

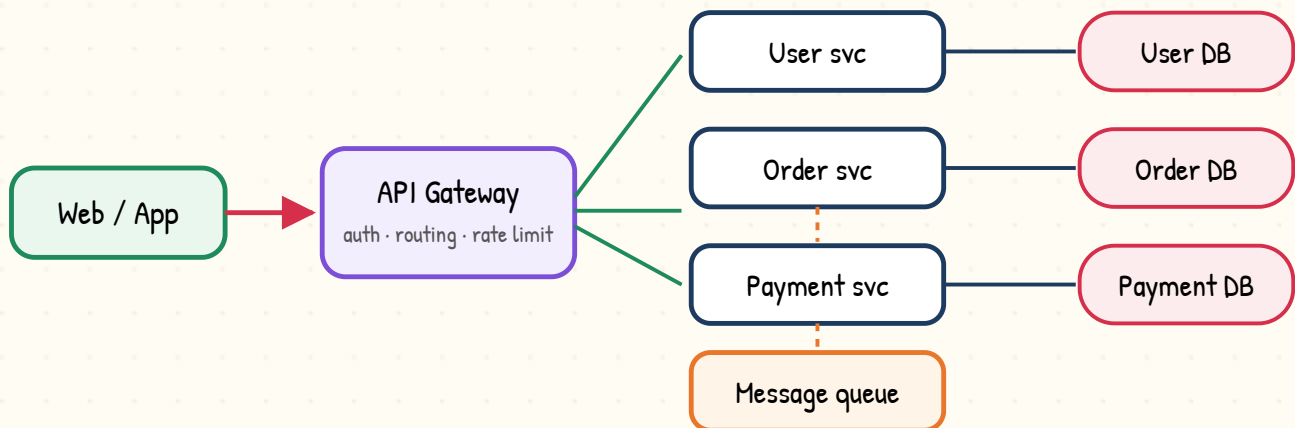
// consumer - separate worker process
new Worker('email', async (job) => {
  await sendMail(job.data.email, job.data.orderId); // make it idempotent!
});
```

Revision

MQ = async + decoupling + spike buffer + retries. P2P (one consumer) vs Pub/Sub (all subscribers). At-least-once
 → idempotent consumers + DLQ. Kafka = log/streaming, RabbitMQ = task broker.

Definition

Microservices = building an app as many small, independent services, each owning **one business capability** and its own **database**, deployed separately, talking over APIs/queues. Opposite: a **monolith** — one codebase, one deploy, one DB.



Point	Monolith	Microservices
Deploy	Whole app every time	Each service independently
Scale	Scale everything together	Scale only the hot service
Failure	One bug can crash all	Isolated (if designed well)
Tech	One stack	Each team picks its stack
Complexity	Simple to start, debug, test	Network calls, distributed data, monitoring, DevOps
Best for	Small team, early product	Large teams, large scale, clear domains

Building blocks

- API Gateway** — single entry: routing, auth, rate limiting, aggregation.
- Service discovery** — find service addresses dynamically (Consul, Kubernetes DNS).
- Communication** — sync REST/gRPC, async message queue / events.
- Database per service** — no sharing tables across services.
- Resilience** — timeouts, retries, **circuit breaker** (stop calling a failing service).
- Observability** — central logs, metrics, distributed tracing (correlation IDs).
- Saga pattern** — a distributed transaction as a chain of local steps + compensating actions (refund if shipping fails).



From my work

Livia HMIS: legacy PHP monolith → 7 NestJS microservices + MySQL, 48 modules migrated, keeping API contracts & business rules unchanged and checked with API parity tests (strangler-style migration).

Mistake

Distributed monolith — services share one DB or must deploy together. You pay microservice costs with none of the benefits. Also: starting a tiny project with microservices.

Revision

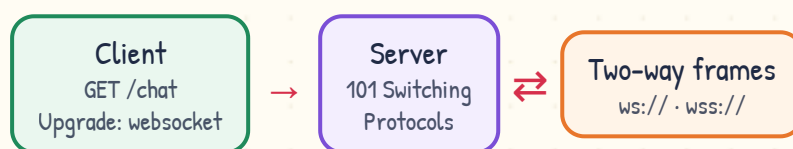
Small services, own DB, independent deploy. Gateway + discovery + async events + circuit breaker + tracing. Saga for cross-service transactions. Monolith first is fine.

HTTP(S) vs WebSocket

Definition

HTTP is request → response: client always asks first, server can't push. **WebSocket** is a persistent, full-duplex connection: after one handshake, both sides can send messages any time. Real life: HTTP = **sending letters**; WebSocket = **a phone call** that stays open.

WebSocket handshake



Point	HTTP	WebSocket
Connection	Short-lived per request (keep-alive reuses TCP)	Long-lived, stays open
Direction	Client → server only (half-duplex)	Both ways (full-duplex)
State	Stateless	Stateful (server tracks connections)
Overhead	Headers on every request	Tiny frames after handshake
Caching / LB	Easy	Harder (sticky connections)
Use	CRUD APIs, pages, files	Chat, live scores, stock ticker, multiplayer games, collaborative editing

Ways to get "real-time"

- ✎ **Short polling** — ask every few seconds (wasteful).
- ✎ **Long polling** — server holds the request until there's data.
- ✎ **SSE (Server-Sent Events)** — one-way server → client stream over HTTP (notifications, live feeds).
- ✎ **WebSocket** — two-way, lowest latency.

```
// server.js (npm i ws)
const wss = new WebSocketServer({ port: 8080 });
wss.on('connection', (ws) => {
  ws.on('message', (msg) => {
    wss.clients.forEach(c => c.send(msg.toString())); // broadcast
  });
});

// browser
const socket = new WebSocket('wss://chat.example.com');
socket.onmessage = (e) => console.log('new:', e.data);
socket.send('Hello!');
```

Scaling gotcha

User A on server 1, user B on server 2 → a message to B must cross servers. Use a pub/sub backbone (Redis Pub/Sub, Kafka) between WebSocket servers, plus sticky load balancing and heartbeats / reconnect logic.

Revision

HTTP = ask → answer, stateless. WebSocket = Upgrade → 101 → open two-way pipe. Polling → long polling → SSE → WebSocket. Scale with Redis pub/sub.

Definition

Rate limiting = controlling how many requests a client can make in a time window (e.g. 100 req/min per user/IP/API key). Extra requests get **429 Too Many Requests**. Real life: an **ATM daily withdrawal limit** / OTP "try again after 30 s".

Why

- Stop abuse: brute-force login, scraping, spam, DDoS. · Fair use between customers. · Protect DB/servers & control cost (paid 3rd-party APIs). · Pricing tiers (free 100/day, pro 10k/day).

Algorithms

Algorithm	How it works	Pros / cons
Token bucket	Bucket holds N tokens, refilled at fixed rate. Each request takes 1 token; empty → reject.	✓ Allows short bursts , memory-light. Used by AWS, Stripe.
Leaky bucket	Requests enter a queue, processed (leak) at a constant rate ; full → drop.	✓ Smooth output. × Bursts wait / dropped.
Fixed window counter	Count per window (12:00–12:01); reset each minute.	✓ Simple. × Boundary burst : 100 at 12:00:59 + 100 at 12:01:00.
Sliding window log	Store timestamp of each request; count those in last 60 s.	✓ Accurate. × Memory heavy.
Sliding window counter	Current count + previous window × overlap %.	✓ Accurate enough + cheap (best balance).

Where to put it



fixed window with Redis (shared across servers)

```
async function rateLimit(req, res, next) {  
  const key = `rl:${req.ip}:${Math.floor(Date.now() / 60000)}`; // per minute  
  const count = await redis.incr(key);  
  if (count === 1) await redis.expire(key, 60);  
  res.set('X-RateLimit-Remaining', Math.max(0, 100 - count));  
  if (count > 100) return res.status(429).set('Retry-After', '60').send('Slow down');  
  next();  
}
```

Mistake

Keeping counters in each server's memory — with 5 servers behind an LB the user really gets 5× the limit. Use a shared store (Redis). Also rate-limit login/OTP endpoints much tighter than normal ones.

Revision

Limit requests per key per window → 429 + Retry-After. Token bucket (bursts), leaky bucket (smooth), fixed window (simple, edge burst), sliding log/counter (accurate). Shared counters in Redis.

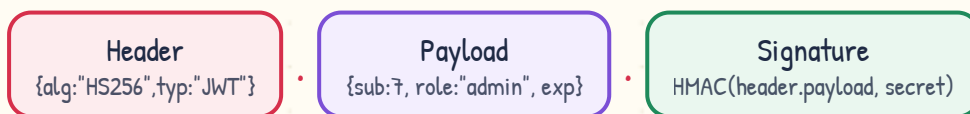
Defense in depth (L21)

Never rely on one lock. Put security at every layer, so if one fails the next stops the attacker. Real life: a bank – guard at the gate, CCTV, ID check at the counter, locker with a key + PIN.

Layer	What protects it
Network	Firewall, VPC / private subnets (DB not public), security groups, DDoS protection (Cloudflare / AWS Shield)
Transport	HTTPS / TLS everywhere, HSTS
Application	<u>Authentication + authorization</u> , input validation, parameterised queries (SQL injection), output escaping (XSS), CSRF tokens, WAF, rate limiting
Data	Encryption at rest, <u>hash passwords</u> (bcrypt/argon2 – never plain/MD5), least-privilege DB users, backups
Monitoring	Audit logs, alerts, intrusion detection, dependency scanning

AuthN (authentication) = *who are you?* (login). **AuthZ** (authorization) = *what are you allowed to do?* (roles: admin / doctor / patient). 401 = not authenticated, 403 = authenticated but not allowed.

JWT – JSON Web Token (L31)



- 3 parts, each **Base64URL**-encoded, joined by dots: `xxxxx.yyyyy.zzzzz`.
- Payload is signed, NOT encrypted – anyone can decode it. Signature only proves it wasn't tampered with.
- HS256** = one shared secret. **RS256** = private key signs, public key verifies (good for many services).

```
// login → issue tokens
const access = jwt.sign({ sub: user.id, role: user.role }, ACCESS_SECRET, { expiresIn: '15m' });
const refresh = jwt.sign({ sub: user.id }, REFRESH_SECRET, { expiresIn: '7d' });
res.cookie('refresh', refresh, { httpOnly: true, secure: true, sameSite: 'strict' });

// middleware → verify on every request
const token = req.headers.authorization?.split(' ')[1]; // "Bearer <token>"
try { req.user = jwt.verify(token, ACCESS_SECRET); next(); }
catch { res.status(401).send('Invalid or expired token'); }
```

Session vs JWT (L32)

Point	Session (stateful)	JWT (stateless)
Stored where	Session data on server (memory/Redis/DB); browser holds only a session ID cookie	All claims inside the token on the client
Verify	Look up session ID in store each request	Check signature — no DB lookup
Scaling	Needs shared session store (Redis) or sticky sessions	Any server can verify → easy horizontal scale, microservices
Logout / revoke	Easy — delete the session	Hard — valid until exp ; use short expiry + refresh tokens / denylist
Size	Small cookie	Bigger token on every request
Best for	Classic server-rendered web apps	APIs, SPAs, mobile apps, service-to-service

From my work

NextAuth in the Marriage Registration portal (session handling); PayU flow where the **server builds and verifies the hash** — never trust amounts or status coming from the browser.

Mistake

Putting passwords / secrets in the JWT payload (it's readable!). Storing tokens in **localStorage** → stolen by XSS; prefer **httpOnly, secure cookies**. Never skip checking **exp** and the algorithm.

Revision

Defense in depth: network → transport → app → data → monitoring. JWT = header.payload.signature, signed not encrypted. Session = server state, easy revoke. JWT = stateless, easy scale, hard revoke → short-lived access + refresh token.

Definition

SOLID = 5 object-oriented design principles (Robert C. Martin) that make code easy to change, test and extend.

Core of LLD interviews.

#	Principle	One line
S	Single Responsibility	A class should have one reason to change .
O	Open / Closed	Open for extension , closed for modification .
L	Liskov Substitution	A subclass must work anywhere its parent works .
I	Interface Segregation	Many small interfaces > one fat interface.
D	Dependency Inversion	Depend on abstractions , not concrete classes.

S – split jobs

```
// × Invoice does maths + PDF + email
class InvoiceCalculator { total() {} }
class InvoicePdf        { render() {} }
class InvoiceMailer      { send() {} }
```

O – add, don't edit

```
interface Payment { pay(amt: number): void }
class Upi implements Payment { pay(a) {} }
class Card implements Payment { pay(a) {} }
// new Wallet? add a class, no if/else edits
```

L – don't break the parent

```
// × Penguin extends Bird { fly() { throw } }
class Bird { eat() {} }
class FlyingBird extends Bird { fly() {} }
class Penguin extends Bird {} // ✓
```

I – small interfaces

```
// × Machine { print(); scan(); fax(); }
interface Printer { print(): void }
interface Scanner { scan(): void }
class BasicPrinter implements Printer {}
```

D – inject the abstraction (NestJS does this for me)

```
interface Notifier { send(to: string, msg: string): Promise<void> }

class OrderService {
  constructor(private notifier: Notifier) {} // not "new EmailService()"
  async place(o) { /* save */ await this.notifier.send(o.email, 'Placed'); }
}

new OrderService(new SmsNotifier()); // swap Email/SMS/mock in tests
```

Revision

S one job · O extend not edit · L child replaces parent safely · I small interfaces · D depend on interfaces (DI). Benefit: changes stay local, tests are easy.

Case study – Instagram

1 · Requirements

Functional

Upload photo/video + caption · follow / unfollow · home feed · like & comment · stories (24 h) · search · notifications.

Non-functional

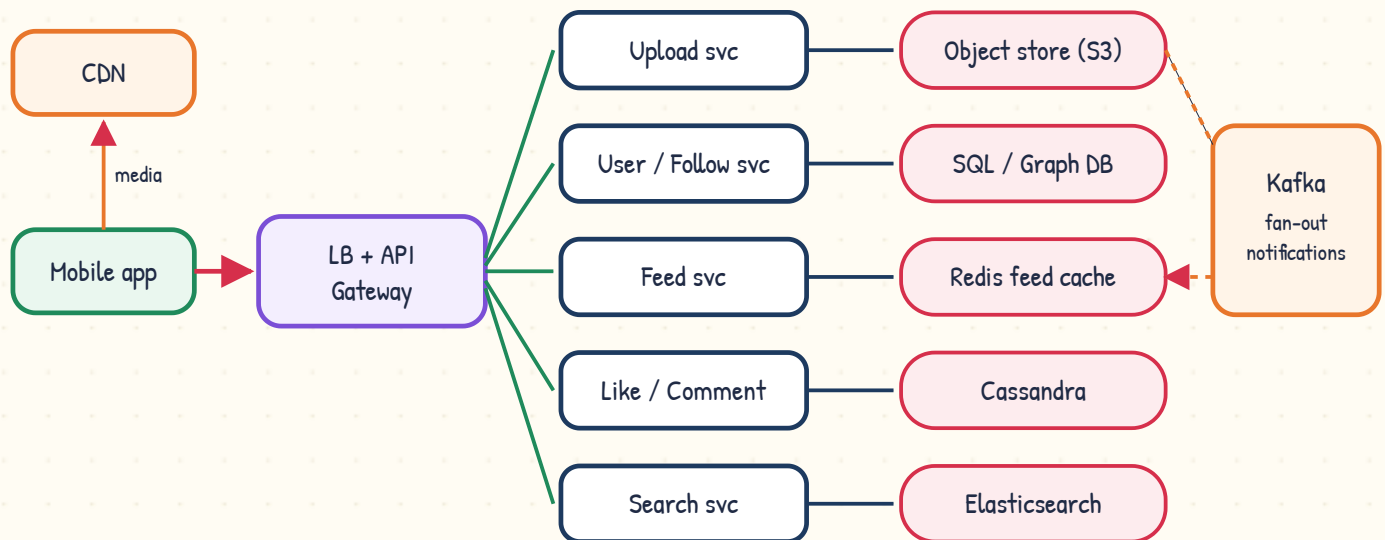
Read-heavy (reads $\gg$ writes) · feed < ~200 ms · highly available (AP is OK for likes/feed) · media never lost · global users.

2 · Estimates (rough)

back-of-envelope

500M DAU, 1 in 10 uploads/day $\rightarrow$ 50M uploads/day $\approx$ 600 uploads/s
 avg media 1 MB (+ resized copies) $\rightarrow$ ~50 TB/day new storage
 each user opens feed ~20x/day $\rightarrow$ 10B feed reads/day $\approx$ 115K reads/s
 $\rightarrow$ read-heavy: cache + CDN everything we can

3 · High-level design



4 · Upload flow

1. App asks Upload svc for a pre-signed URL $\rightarrow$ uploads the file directly to S3 (API servers don't carry big files).
2. Metadata (post id, user id, caption, s3 key, time) saved in DB.
3. Event `PostCreated` $\rightarrow$ Kafka $\rightarrow$ workers make thumbnails / multiple resolutions, update feeds, send notifications.
4. Media served to everyone through the CDN.

5 · The feed – fan-out (key interview point)

Approach	How	Trade-off
Fan-out on write (push)	When I post, push post-id into every follower's pre-computed feed list in Redis	✓ Feed read is instant. × Celebrity with 100M followers = 100M writes
Fan-out on read (pull)	When I open the app, fetch latest posts from everyone I follow and merge	✓ Cheap writes. × Slow reads
Hybrid ✓	Push for normal users; pull celebrity posts at read time and merge	Best of both – what large social apps do

- ✎ Likes/comments: huge write volume → Cassandra; like **counts** in Redis, flushed in batches (write-back!).
- ✎ Stories: stored with **24 h TTL**. Search: Elasticsearch on users/hashtags. Feed paginated with a cursor.

Revision

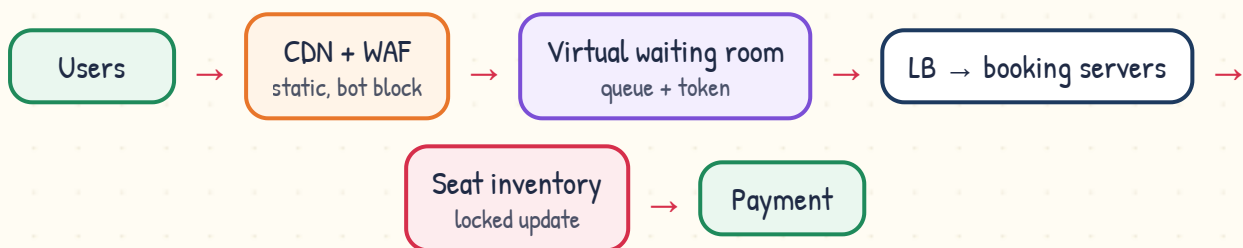
Read-heavy → CDN + cache. Media → pre-signed URL → S3 → CDN. Metadata in DB, events via Kafka. Feed = hybrid fan-out (push normal, pull celebrities). Counters in Redis.

Case study – IRCTC Tatkal

The problem

Tatkal booking opens at a fixed time (10:00 AM AC, 11:00 AM non-AC). Lakhs of users hit "Book" in the same few seconds for a very small number of seats. Need: no crash, **no double booking**, fair order, and payments that don't lose money.

Design – layer by layer



1. **Absorb the spike at the edge** – static pages, train lists and schedules from CDN / cache; login done before 10:00.
2. **Rate limiting + CAPTCHA + bot detection** – stop scripts and agents spamming requests; limit per user/IP.
3. **Virtual waiting room / queue** – users get a place in line; only as many as the booking servers can handle are let in. Smooth load instead of a crash.
4. **Horizontal scaling** – many stateless booking servers behind load balancers; auto-scale before the window opens (the spike is *predictable*).
5. **Search ≠ booking** – availability searches go to **read replicas / cache**; only the final booking touches the primary DB.
6. **Seat inventory with locking** – the critical part: decrement seats atomically so two people never get the same berth.
7. **Temporary hold** – seat is held for the payment window; payment fails / times out → seat released back.
8. **Async after payment** – ticket PDF, SMS, email go through a message queue.

no double booking – atomic conditional update

```

-- only succeeds if a seat is still free; DB row lock makes it atomic
UPDATE train_quota
  SET available = available - 1
 WHERE train_no = '12951' AND journey_date = '2026-11-02'
    AND class = '3A' AND quota = 'TATKAL' AND available > 0;
-- rows affected = 1 → seat held, go to payment
-- rows affected = 0 → sold out / waitlist

// or a fast pre-check in Redis (atomic DECR) before touching the DB
if (await redis.decr('seats:12951:3A:TQ') < 0) { await redis.incr(...); return soldOut(); }
  
```

<i>Locking</i>	<i>How</i>	<i>When</i>
Pessimistic	<code>SELECT ... FOR UPDATE</code> — lock row, others wait	High contention (Tatkal!)
Optimistic	Version column; update fails if version changed → retry	Low contention

Mistake

"Just add more servers" — the bottleneck is the **shared seat inventory**, not web servers. Without atomic updates you get **double booking**; without a queue, the DB melts. This is a CP choice: better to say "try again" than sell one berth twice.

Revision

Predictable spike → pre-scale + CDN + cache. Rate limit + CAPTCHA + waiting room. Search on replicas. Atomic / locked seat decrement, hold during payment, release on timeout. Async ticket/SMS via MQ.

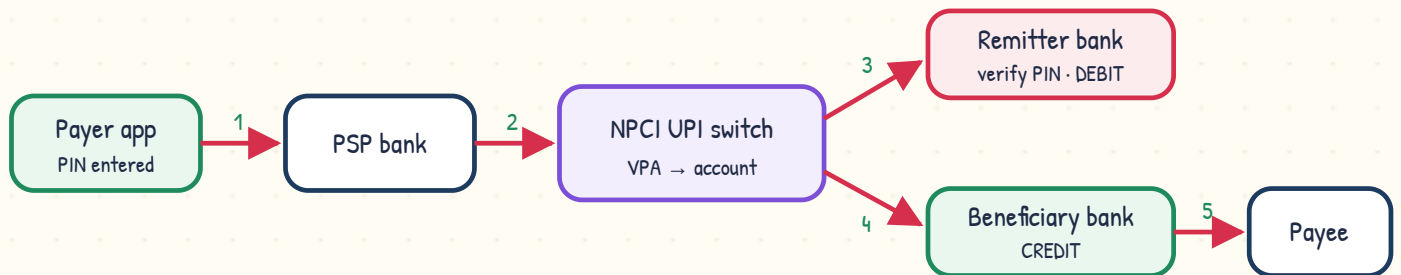
What is UPI?

Unified Payments Interface – India's real-time, 24x7 bank-to-bank payment system run by **NPCI**. Pay using a **VPA** / **UPI ID** (`name@okbank`) or QR code – no account number / IFSC needed.

The actors

Actor	Role
Payer app (TPAP)	PhonePe, Google Pay, Paytm, BHIM – the UI on my phone
PSP bank	Payment Service Provider bank behind the app; connects the app to UPI, owns the <code>@handle</code>
NPCI UPI switch	Central router: resolves VPA → bank account, routes debit/credit, keeps the records
Remitter bank	My bank – verifies UPI PIN, debits my account
Beneficiary bank	Receiver's bank – credits their account

Payment flow (send ₹500)



1. I enter the payee VPA + ₹500 + UPI PIN. The PIN is **encrypted on the device** (NPCI common library) – the app never sees it in plain text.
2. App → PSP bank → **NPCI switch** with a unique **transaction ID**.
3. NPCI resolves payee VPA and sends a **debit request** to my bank → bank verifies PIN, checks balance, debits.
4. NPCI sends a **credit request** to the beneficiary bank → credits payee.
5. Success travels back to both apps in seconds; both get notifications.

System design ideas inside UPI

- Idempotency** – every transaction has a unique ID; retries never debit twice.
- Strong consistency (CP / ACID)** for money: debit + credit must both happen, or the debit is **reversed**.

-  **Timeouts & deemed status** — if a bank doesn't respond, the transaction goes to "pending" and is resolved by status checks + **reconciliation**; failed debits are auto-reversed.
-  **Security** — device binding (SIM + phone) + UPI PIN = 2-factor; everything over encrypted channels; per-transaction & daily limits.
-  **Scale & availability** — billions of transactions/month, 24x7; NPCI switch is horizontally scaled with redundant data centres; bank downtimes are isolated.
-  **Interoperability** — any app ↔ any bank through one central switch (hub-and-spoke), like a standard API contract.

Mistake

Thinking the app (PhonePe/GPay) holds my money — it doesn't. Money moves **bank** → **bank**; the app is only the interface, NPCI is the router.

Revision

App → PSP → NPCI (resolve VPA) → remitter bank (PIN + debit) → beneficiary bank (credit). Unique txn ID = idempotent. Money = consistency first, reversals + reconciliation. Device binding + PIN.

Q. What happens when you type a URL?

A. DNS → TCP → TLS → HTTP → CDN/LB → app → cache → DB → response → render.

Q. Load balancer vs reverse proxy?

A. LB spreads load across many servers; a reverse proxy fronts servers (SSL, cache, security). Every LB is a reverse proxy.

Q. Which LB algorithm for long-lived uneven requests?

A. Least connections.

Q. How does a CDN reduce latency?

A. Serves cached copies from an edge server physically near the user; only misses go to origin.

Q. Write-through vs write-back?

A. Write-through writes cache + DB together (safe, slower). Write-back writes cache, flushes later (fast, risk of loss).

Q. What is cache stampede?

A. A hot key expires and many requests hit the DB at once — fix with locking / single-flight and TTL jitter.

Q. Horizontal vs vertical scaling?

A. Vertical = bigger machine (limit, SPOF). Horizontal = more machines (needs LB + stateless apps).

Q. Sharding vs replication?

A. Sharding splits data (scale writes/storage); replication copies data (scale reads, HA). Use both.

Q. Why consistent hashing?

A. Adding/removing a node moves only $\sim 1/N$ keys instead of reshuffling everything.

Q. CAP in one line?

A. Partitions will happen, so choose consistency (CP) or availability (AP) during one.

Q. ACID vs BASE?

A. ACID = strict transactions, strong consistency. BASE = basically available, eventually consistent.

Q. When NoSQL over SQL?

A. Flexible schema, massive scale / write volume, simple access patterns. SQL for relations, transactions, reporting.

Q. Why a message queue?

A. Decouple services, absorb spikes, do slow work async, retry safely.

Q. At-least-once delivery means...?

A. Messages can repeat → consumers must be idempotent; failed ones go to a DLQ.

Q. When NOT to use microservices?

A. Small team / early product — a modular monolith is simpler; split later along clear domains.

Q. WebSocket vs SSE?

A. WebSocket is two-way; SSE is one-way server → client over plain HTTP.

Q. Best rate-limit algorithm for bursts?

A. Token bucket.

Q. Is JWT encrypted?

A. No — signed. Anyone can read the payload; the signature only prevents tampering.

Q. How to log out a JWT user?

A. Short-lived access token + revocable refresh token (or a denylist).

Q. HLD vs LLD?

A. HLD = components & data flow; LLD = classes, methods, schema inside each component.

Q. How to prevent double booking?

A. Atomic conditional update / row lock on inventory + temporary hold + release on timeout.

One-page cheat sheet

building blocks

DNS	→ name to IP (cached by TTL)
CDN	→ static near users
LB	→ spread + health checks
Rev proxy	→ SSL, cache, hide servers
Cache	→ Redis, cache-aside, LRU
MQ	→ async, decouple, retry
Gateway	→ auth, route, rate limit
WebSocket	→ real-time two-way

data

SQL	→ ACID, joins, money
Document	→ MongoDB
Key-value	→ Redis
Column	→ Cassandra (heavy writes)
Graph	→ Neo4j (relations)
Replicate	→ reads + HA
Shard	→ writes + size
CAP	→ CP money · AP feeds

Next step

Practice drawing: URL shortener, WhatsApp chat, YouTube, Uber, notification system — each in 30 minutes with the 7-step approach from Chapter 1.

— End of System Design · keep drawing!  —