System Design

Verified 2026-11-02

Requirements and numbers first, then load balancing, caching, data, replication, sharding, queues and limits — plus a 6-step interview plan.

Video walkthrough — scan the code to watch.

Contents

Mental model 2

Requirements and estimation 2

Load balancing 3

Caching layers 4

Database choice and sharding 5

Replication and consistency 6

Queues, async work and idempotency 6

Rate limiting and overload 8

Observability and cost 9

The 45-minute interview 10

Further reading 11

Mental model

Every design is a set of trade-offs forced by three pressures: load (how much), consistency (how correct, how soon) and failure (what breaks and what it costs). Boxes and arrows are the last step, not the first — state the requirements and the numbers, and most components choose themselves. A design without numbers cannot be wrong, which is exactly why it is useless.

Requirements and estimation

Ask	Turns into
Who uses it, doing what?	Functional requirements, API
How many, how often?	Requests per second, peak factor
How much data, for how long?	Storage and growth
How fast, how available?	Latency and availability SLOs
How fresh, how correct?	Consistency model
What budget?	Cost ceiling

Write the numbers down before drawing anything. A day has 86,400 seconds; round to 100,000 for mental maths. State every assumption — peak multiplier, record size, read/write ratio — so a reviewer can challenge it instead of your conclusion.

```
10M DAU x 20 reads  = 200M reads/day
200M / 86,400 s     ~ 2,300 reads/s average
peak (assume 3x)    ~ 7,000 reads/s
10M writes/day      ~ 116 writes/s average
```

```
10M x 1 KB          = 10 GB/day
10 GB x 365         ~ 3.65 TB/year
reads:writes        = 20:1  -> cache reads
```

> **Gotcha:** Averages hide the peak. A service sized for the daily average falls over at 9 a.m.; size for peak and say what multiplier you assumed.

Load balancing

Choice	Use when
L4 (TCP/UDP)	Raw throughput, non-HTTP protocols
L7 (HTTP)	Path routing, TLS termination, retries
Round robin	Uniform, short requests
Least connections	Uneven request cost
Consistent hashing	Keep a key on the same backend

A load balancer spreads requests across interchangeable, stateless instances and stops sending traffic to an instance that fails its health check. Keep session state out of the instance (in a cookie, a cache or a database) so any instance can serve any request.

```
client -> DNS -> L7 LB (TLS, /api, /static)
                  |-> app-1  (health: GET /healthz)
                  |-> app-2
                  '-> app-3  (failing: drained)
```

> **Gotcha:** A health check that also checks the database marks every instance unhealthy when the database blips, and the balancer then has nowhere to send traffic. Check the process; alert on the dependency.

Caching layers

Layer	Holds
Browser	Static assets, per-user responses
CDN	Static and public responses near users
Application cache	Hot objects (Redis, Memcached)
Database buffer	Hot pages, automatically

Cache-aside is the default: read the cache, on a miss read the database and fill the cache with a TTL. Every cache trades freshness for speed, so decide how stale each kind of data may be. Invalidate on write when staleness matters; otherwise let the TTL expire it.

```
get(k):  v = cache.get(k)
         if v: return v
         v = db.read(k)
         cache.set(k, v, ttl=60s)
         return v
put(k):  db.write(k, v); cache.delete(k)
```

> **Gotcha:** When a hot key expires, thousands of requests miss at once and all hit the database (a cache stampede). Let one request refill while others wait or serve stale, and add jitter to TTLs.

Database choice and sharding

Need	Reach for
Relations, transactions, ad hoc queries	Relational (PostgreSQL)
Key lookups at huge scale	Key-value / wide-column
Flexible documents	Document store
Time-ordered metrics	Time-series
Similarity search	Vector index

Start relational unless the access pattern says otherwise. Shard (partition) only when one primary cannot take the write load or hold the data. Range partitioning keeps scans cheap but creates hot spots on sequential keys; hash partitioning spreads load but loses range scans.

```
shard = hash(user_id) mod N     # avoid:
  N -> N+1 moves almost every key
shard = lookup(user_id)         # better:
  many fixed partitions, move whole ones
```

> **Gotcha:** One celebrity user or tenant can saturate a single shard, however good the hash. Watch per-partition load and split or salt hot keys.

Replication and consistency

Setup	Trade-off
Leader + async followers	Fast writes; followers lag
Leader + sync follower	No loss on failover; slower writes
Multi-leader	Writes in many regions; conflicts
Leaderless quorum	`w + r > n` to read the latest write

Replicas give read scale and survive machine loss. Asynchronous replicas lag, so a user can write and then read old data from a follower. Pick the guarantee you need per feature — read-your-writes for a profile page, eventual for a like count — rather than one for the whole system.

```
POST /profile  -> leader (write)
GET  /profile  -> follower (lag 2 s): old value
fix: read from leader for 10 s after
     this user's last write
```

> **Warning:** Failing over to an asynchronous follower loses every write it had not yet received. If losing acknowledged writes is unacceptable, you need synchronous replication or consensus.

Queues, async work and idempotency

Pattern	Use
Queue + workers	Smooth bursts, slow work off the request
Pub/sub	One event, many consumers

Pattern	Use
Outbox table	Write DB row and event atomically
Idempotency key	Safe client retries
Dead-letter queue	Park poison messages

Return `202 Accepted` and let a worker finish slow jobs. Queues deliver at least once, and clients retry on timeouts, so any operation that can run twice must produce the same effect as once: store the result under a client-supplied key and replay it.

```
type Res = { status: number; body: string };
const seen = new Map<string, Promise<Res>>();
function charge(key: string, cents: number) {
  const prior = seen.get(key);
  if (prior) return prior; // replay
  const res = (async (): Promise<Res> => {
    // ...call the payment provider once...
    return { status: 201, body: `${cents}` };
  })();
  seen.set(key, res);
  return res;
}
const a = await charge("key-1", 500);
const b = await charge("key-1", 500);
console.log(a === b); // true
```

> **Tip:** Queue mechanics — acknowledgements, ordering, retries and dead letters — are on the queuing systems sheet.

Rate limiting and overload

Technique	Effect
Token bucket	Average rate plus bounded burst
`429 + Retry-After`	Tell clients when to return
Load shedding	Reject early instead of timing out late
Backoff with jitter	Retries spread out, not synchronised
Timeouts and deadlines	Stop work nobody is waiting for

Limit per client at the edge to protect shared capacity. Under overload, failing some requests fast keeps the rest fast; queueing everything makes every request slow and then all of them time out. Cap retries, because each layer that retries multiplies load on the layer below.

```
function bucket(rate: number, burst: number) {
  let tokens = burst, last = Date.now();
  return () => {
    const now = Date.now();
    const refill = (now - last) / 1000 * rate;
    tokens = Math.min(burst, tokens + refill);
    last = now;
    if (tokens < 1) return false; // 429
    tokens -= 1; return true;
```

```
  };
}
const take = bucket(5, 10); // 5/s, burst 10
let n = 0;
for (let i = 0; i < 12; i++) if (take()) n++;
console.log(n); // 10
```

> **Gotcha:** An in-memory limiter on each of 10 instances allows 10x the intended rate. Share the counter (for example in Redis) or divide the limit by the instance count.

Observability and cost

Signal	Question it answers
Latency	How slow, for successes and failures separately
Traffic	How much demand
Errors	What fraction fails, explicitly or silently
Saturation	How full the tightest resource is
Cost per request	What each unit of load costs

These are the SRE book's four golden signals, plus cost. Set an SLO (for example 99.9% of requests succeed under 300 ms) and the remaining 0.1% is the **error budget** you may spend on releases and risk. Cost and sustainability are pillars of AWS Well-Architected next to reliability and performance, so put a cost per request in the design.

```
SLO      downtime per 30 days
99%      7.2 h
99.9%    43.2 min
99.95%   21.6 min
99.99%   4.32 min
```

> **Gotcha:** Average latency hides the slow tail that users feel, and fast errors pull the average down. Track percentiles (p95, p99), and track failed-request latency separately.

The 45-minute interview

Step	Output
1. Requirements	Functional list, non-functional numbers
2. Estimation	RPS, storage, read/write ratio
3. API and data model	Endpoints, entities, keys
4. High-level design	Components and data flow
5. Deep dives	The 1–2 hardest parts, with trade-offs
6. Wrap-up	Bottlenecks, failure modes, cost, next steps

The interviewer grades reasoning, not the final diagram. Narrate each choice as a trade-off (“I pick X over Y because the 20:1 read ratio…”) and steer the deep dive toward the part where the requirements bite hardest. A workable split is about 10 minutes for steps 1–3, 10 for step 4 and the rest on 5–6.

```
"Writes are 116/s, so one primary is fine.
 Reads are 7,000/s at peak, 20:1 -> cache.
 Feed staleness of 30 s is acceptable,
 so cache-aside with a 30 s TTL. If the
 cache dies, the DB sees 7,000/s: I'd add
 read replicas and shed load before that."
```

> **Tip:** When stuck, return to the numbers. Most “which component?” questions are settled by a rate, a size or a latency target.

Further reading

- Designing Data-Intensive Applications
- Google SRE book
- AWS Well-Architected Framework
- Microservices for service boundaries and sagas