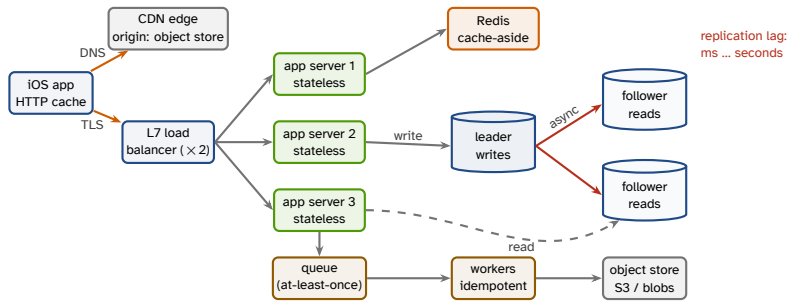
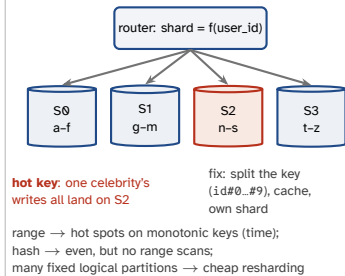


In one line: Scale by making servers **stateless** and adding them behind a **load balancer**; take read load off the database with **caches** and **read replicas**; take write and storage load with **sharding**; move slow work onto **queues**. Each step buys scale with a named cost — staleness, replication lag, hot keys, duplicates — and a senior answer names the cost and the **number** that justifies it.

The canonical read-heavy architecture — and what each hop costs



Sharding by key



Load balancers

	sees	can
L4	IP + port, TCP/UDP flows	fast, protocol-agnostic, TLS passthrough; no per-request routing
L7	HTTP: path, headers, cookies	terminate TLS, route / api vs / img, retries, rate limits, sticky sessions

Algorithms: round-robin (weighted), **least connections**, least response time, IP/consistent hash (affinity, cache locality), **power of two choices** (pick 2 at random, send to the less loaded — near-optimal, no global state). Health checks eject bad nodes. The LB itself is redundant (active-passive pair + floating IP, or DNS/anycast). Session state in a server's memory breaks all of it → **stateless** servers, state in Redis/DB or a signed token.

Caching — layers and patterns

Layers: app/HTTP cache (Cache-Control, ETag) → CDN → reverse proxy → in-process → distributed (Redis/Memcached) → DB buffer pool.

cache-aside	app: read cache; miss → DB → fill; write DB, then delete key	default; stale window small
read-through	cache loads from DB itself	simpler app code
write-through	write cache + DB synchronously	fresh; slower writes
write-back	write cache, flush to DB later	fastest; loses data on crash
write-around	write DB only; cache on read	no churn for write-once data

Invalidate: TTL (bound staleness) + delete-on-write (not update: two racing writers can leave the old value) or versioned keys. **Stampede** (hot key expires, 1000 misses hit the DB): single-flight / request coalescing, a lock + serve stale, early probabilistic refresh, TTL jitter. Effective latency = $h \cdot t_{cache} + (1 - h) \cdot t_{db}$: 90% → 99% hit rate cuts DB load 10x.

Databases — replication and sharding

- Leader-follower:** leader takes writes, streams its log. **Async:** fast, but followers **lag** and a failover can lose the last writes; **sync/semi-sync:** durable, slower. Read replicas scale **reads** only.
- Lag breaks **read-your-writes** (post, refresh, it's gone): read your own data from the leader for a few seconds, or wait until the replica passed your write's log position. **Monotonic reads:** stick a user to one replica.
- Multi-leader** (multi-region, offline clients): write conflicts → last-write-wins (loses data) or CRDTs/merge. **Leaderless** (Dynamo, Cassandra): N replicas, write W, read R; $R + W > N$ ⇒ read and write sets overlap.
- Sharding** splits writes + storage; costs: cross-shard joins and transactions (denormalise, sagas), resharding, hot keys (panel above). Shard key = the access pattern (`user_id`, `space_id`).
- Indexes (B-tree) make reads $O(\log n)$ and every write slower. EXPLAIN.

CAP and PACELC, in plain words

CAP: when the network **partitions**, a replica cut off from the others must either refuse (Consistent) or answer with possibly stale data (Available). P is not optional, so it is **CP vs AP during a partition** — not "pick 2 of 3". **PACELC:** if P, A or C; else (normal operation) Latency or Consistency — the trade you actually pay every day (sync replication = slower writes).

Consistency models — strongest first

linearizable	behaves like one copy; a read sees every completed write (bank balance, locks, unique usernames)
sequential / causal	everyone sees causally related writes in order (reply after the post)
read-your-writes	you always see your own writes (profile edit)
monotonic reads	time never goes backwards for one user
eventual	replicas converge if writes stop (like counts, feeds)

Queues — async work

Decouple producer from consumer, absorb bursts, fan out. Delivery is **at-least-once** → consumers **idempotent** (dedupe on a message id). Poison messages → retry with backoff → **dead-letter queue**. Ordering only per partition key (Kafka partition). **Log** (Kafka: retained, replayable, many readers) vs **queue** (SQS/RabbitMQ: deleted on ack). DB write + publish atomically → **outbox**. Watch **queue depth / age** — it is your backlog alarm.

Back-of-envelope — numbers to know

L1 cache / main memory reference	1 / 100 ns	RAM ≈ 100 × L1
SSD random read (4 KB)	~100 μs	1000 × RAM
read 1 MB: RAM / SSD	~10s μs / ~1 ms	disk seek 5-10 ms
round trip in one datacenter	0.5 ms	
RTT same continent / cross-Atlantic	20-50 / ~80 ms	mobile LTE ~50 ms
frame budget 60 / 120 Hz	16.7 / 8.3 ms	

Method: 1 day ≈ 86 400 s ≈ 10^5 s → 1M requests/day ≈ 12 QPS. **QPS** = DAU × requests per user ÷ 10^5 ; **peak** = 2-3 × average (more for events). **Storage** = items/day × size × retention × 3 replicas. State assumptions, round hard, keep the order of magnitude.

Example. 10M DAU × 20 requests = 2×10^8 /day → ~2300 QPS avg, ~7000 peak. 10% post one 500 KB photo/day → 500 GB/day → ~180 TB/year, ×3 replicas ≈ 0.5 PB → object store + CDN, never the DB.

Interview traps

- ! CAP as "pick two"; "NoSQL scales, SQL doesn't" — most apps fit one well-indexed Postgres plus replicas.
- ! A read replica right after a write → the user's change vanishes.
- ! Updating (not deleting) the cache on write; write-back without saying it can lose data.
- ! Averages instead of **p99**; a single LB/DB = single point of failure.

Remember

Stateless → **cache** → **replicate** → **shard** → **queue**, and for each: what goes stale, what breaks, what number made you do it.

Likely questions

- L4 vs L7? — connections vs requests; L7 routes by path, terminates TLS.
- Cache stampede? — coalesce misses, lock + stale, jittered TTLs.
- Post disappears after refresh? — replica lag: read-your-writes from the leader.