



System Design

A focused reference for interview preparation.

System Design — Complete Interview Cheat Sheet

Recognition framework · key concepts & trade-offs · common components, all in one place

CORE SCALABILITY CONCEPTS

DATA STORAGE & CONSISTENCY

COMMON SYSTEM COMPONENTS

HOW TO APPROACH A SYSTEM DESIGN QUESTION

Ask these questions first — they point at which concepts matter

- Is the bottleneck about read volume, write volume, or storage size?
- Does the system need strong consistency, or is eventual consistency acceptable?
- Would caching or a CDN remove load from the origin/database entirely?
- Is there a single point of failure that replication or sharding would fix?
- Do producers and consumers need to be decoupled in time (a queue)?
- Is uniqueness/ordering needed across multiple machines with no central coordinator?

Bottleneck	Signal	Use	Trade-off
Too much read traffic	Same data read repeatedly	Caching + read replicas	Staleness risk
Too much write traffic / data size	Single DB can't hold or write it all	Sharding	Cross-shard queries get hard
High latency for distant users	Global user base, static content	CDN	Cache invalidation complexity
Single server can't handle load	Traffic exceeds one machine's capacity	Horizontal scaling + load balancer	Requires stateless services
Bursty or spiky traffic	Producers faster than consumers	Message queue	Consumers must handle duplicates/order
Clients overwhelming the system	Abuse or noisy-neighbor traffic	Rate limiter	Legitimate bursts may get throttled
Need unique IDs, no central authority	Multiple machines generating IDs	Snowflake-style ID generator	Clock sync matters for ordering

HOW THIS SHEET DIFFERS FROM THE OTHERS

System design questions are open-ended and trade-off-driven rather than algorithmic — there's no single 'correct' code template. Use this sheet to recognize which concept addresses your bottleneck, then talk through the trade-offs out loud with your interviewer rather than trying to reach one 'optimal' design.

CORE SCALABILITY CONCEPTS

The building blocks behind handling more load

1 Vertical vs. Horizontal Scaling

Capacity strategy

Vertical scaling adds power (CPU/RAM) to a single machine; horizontal scaling adds more machines and splits the load between them.

- **Vertical:** simpler, no distributed-systems complexity, but hits a hardware ceiling (and stays a single point of failure)
- **Horizontal:** near-unlimited scale, but needs load balancing + data partitioning
- **Rule of thumb:** start vertical for simplicity; move horizontal once one box can't keep up
- **Watch for:** horizontal scaling often forces the app to be stateless (session state moves to a shared store)

2 Load Balancing

Traffic distribution

A load balancer spreads incoming requests across multiple servers so no single server is overwhelmed, and can reroute around failures.

- **Round robin:** simplest, ignores server load — fine for uniform requests
- **Least connections:** routes to the server with the fewest active requests
- **Consistent hashing:** routes by key so the same client/key tends to hit the same server (good for caching)
- **Health checks:** automatically stop routing to failed servers

3 Caching Strategies

Reduce read latency

A cache stores frequently accessed data in fast storage (often in-memory) to avoid repeated expensive reads from the source of truth.

- **Cache-aside:** app checks cache first, loads from DB on a miss, then populates the cache
- **Write-through:** writes go to cache and DB together — consistent, but slower writes
- **Write-back:** writes go to cache first, flushed to DB later — fast, but risks data loss on crash
- **Eviction:** LRU is the default choice; set a TTL for data that can go stale

INTERVIEWER FOLLOW-UP

“What about cache stampedes?” — when a hot key expires, many requests hit the DB at once; mitigate with request coalescing (single-flight), jittered TTLs, or serving stale while revalidating.

4 Content Delivery Network (CDN)

Geographic latency

A CDN caches static content (images, video, JS/CSS) at edge servers close to users, cutting latency and offloading the origin server.

- **Push CDN:** you upload content ahead of time — good for less-frequently-changing content
- **Pull CDN:** CDN fetches from origin on first request, then caches it — simpler to manage
- **Use for:** static assets, video streaming, large file downloads
- **Not a fit for:** highly dynamic, per-user, or frequently changing data

DATA STORAGE & CONSISTENCY

Trade-offs in how data is stored, split, and kept in sync

5 SQL vs. NoSQL

Data model choice

SQL databases enforce a strict schema and strong relational guarantees; NoSQL trades some of that structure for horizontal scalability and flexible schemas.

- **SQL:** ACID transactions, joins, strong consistency — best for structured, relational data
- **NoSQL** (key-value, document, wide-column, graph): flexible schema, easier horizontal scaling
- **Pick SQL when:** data integrity and complex queries matter more than raw write throughput
- **Pick NoSQL when:** massive scale, simple access patterns, or a rapidly evolving schema

SAY THIS OUT LOUD

“The divide is blurrier in practice — modern SQL databases shard and replicate too, and some NoSQL stores offer transactions — so I'd decide based on access patterns, not the label.”

6 Database Replication

Availability & read scale

Copy data across multiple database nodes so reads can be spread out and the system survives a node failure.

- **Leader-follower:** writes go to the leader, followers replicate and serve reads
- **Synchronous replication:** stronger consistency, higher write latency
- **Asynchronous replication:** lower write latency, risk of stale reads / data loss on failover
- **Failover:** promote a follower to leader automatically when the leader dies

BREAKS WHEN

Replication scales READS only — every replica still applies every write, so a write-volume bottleneck needs sharding (next), not more replicas.

7 Database Sharding (Partitioning)

Write scale & storage limits

Split a large dataset across multiple database instances (shards), each holding a subset of the data, so no single machine stores or serves all of it.

- **Range-based:** simple, keeps ranges queryable, but can create hot shards if data isn't uniform
- **Hash-based:** even distribution, but range queries become expensive
- **Directory-based:** a lookup service maps keys to shards — flexible, adds a dependency
- **Trade-off:** cross-shard joins/transactions become significantly harder

SAY THIS OUT LOUD

"Shard-key choice is the whole game: I want a key with high cardinality and even access — user_id usually; date is a classic hot-shard mistake for write-heavy loads."

8 CAP Theorem

Distributed systems limit

A distributed system can only fully guarantee two of three properties at once: Consistency, Availability, Partition tolerance — and network partitions WILL happen, so it's really a C-vs-A choice.

- **Consistency (CP):** every read returns the latest write, even if some nodes must reject requests
- **Availability (AP):** every request gets a response, even if it may not be the latest data
- **Partition tolerance:** the system keeps working despite dropped/delayed messages between nodes
- **In practice:** partitions are unavoidable, so real systems choose CP or AP behavior under partition

INTERVIEWER FOLLOW-UP

"And when there's no partition?" — that's PACELC: Else, the trade is Latency vs Consistency. Mentioning it signals you know CAP only constrains behavior during partitions.

9 Consistent Hashing

Even load, minimal reshuffling

Maps both servers and keys onto a hash ring, so adding/removing a server only remaps a small fraction of keys instead of all of them.

- **Problem it solves:** plain hash % N remaps almost every key when N changes
- **Virtual nodes:** each physical server maps to many ring positions for more even load
- **Used in:** distributed caches, load balancers, and distributed databases (e.g. DynamoDB-style)
- **Benefit:** scaling out/in only disrupts $\sim 1/N$ of the key space, not all of it

COMMON SYSTEM COMPONENTS

Recurring pieces that show up across most designs

10 Message Queues

Async decoupling

Producers publish messages to a queue/topic; consumers process them independently, decoupling services and smoothing out traffic spikes.

- **Use for:** decoupling services, buffering bursty traffic, background job processing
- **At-least-once delivery:** the common guarantee — consumers must be idempotent to handle duplicates
- **Ordering:** guaranteed only within a partition/shard key, not globally
- **Examples in the wild:** Kafka (log-based, high throughput), SQS (simple, managed)

11 Rate Limiter

Protect against overload

Caps how many requests a client (or the system) can make in a given time window, protecting downstream services from being overwhelmed.

- **Token bucket:** tokens refill at a fixed rate; a request consumes a token — allows controlled bursts
- **Sliding window log:** precise, but memory cost grows with request volume
- **Fixed window counter:** simple, but allows up to 2× bursts at window boundaries
- **Where enforced:** at the API gateway/load balancer, before hitting application servers

INTERVIEWER FOLLOW-UP

"What does the client see?" — respond 429 Too Many Requests with a Retry-After header, and make sure limits are per-key (user/IP/API key), tracked in a shared store like Redis if there are multiple gateways.

12 Designing a URL Shortener

End-to-end walkthrough

A canonical system design interview: generate a short, unique code for a long URL and redirect efficiently at scale.

- **ID generation:** base62-encode an auto-incrementing ID, or hash + collision check
- **Storage:** a simple key-value store (short code → long URL) is usually enough
- **Redirect:** choose 301 (permanent, cacheable — but browsers stop hitting you) vs. 302 (temporary — every hit reaches you, so clicks are trackable) deliberately
- **Scale reads:** cache hot short codes; reads vastly outnumber writes in this system

13 Designing a Unique ID Generator

Distributed ID generation

Generate unique, roughly time-sortable IDs across many machines without a single point of contention.

- **Snowflake-style ID:** timestamp bits + machine ID bits + sequence number bits, packed into 64 bits
- **Benefit:** IDs are unique across machines with no central coordinator needed
- **Roughly sortable:** the timestamp occupies the high-order bits, so IDs increase over time
- **Alternative:** UUID v4 is simpler (no coordination) but not sortable and larger (128 bits)

BREAKS WHEN

Snowflake assumes clocks never go backwards — NTP corrections can violate that, so generators must refuse or wait when they detect regression; that's the 'clock sync matters' trade-off from the table.

QUICK-REFERENCE SUMMARY

One-line reminder for every concept on this sheet

Concept	What it's for
Vertical vs. Horizontal Scaling	Add power to one machine, or add more machines
Load Balancing	Spread requests across servers; consistent hashing aids caching
Caching Strategies	Cache-aside is the default; write-through/back trade consistency vs. speed
CDN	Cache static content at the edge, close to users
SQL vs. NoSQL	Structured + strong consistency vs. flexible + horizontal scale
Database Replication	Leader-follower for availability and read scaling
Database Sharding	Split data across machines by range, hash, or directory
CAP Theorem	Pick Consistency or Availability once a network partition happens
Consistent Hashing	Ring-based mapping minimizes reshuffling when servers change
Message Queues	Decouple producers and consumers; expect at-least-once delivery
Rate Limiter	Token bucket is the default; enforce at the gateway

Concept	What it's for
URL Shortener Design	Base62 ID + key-value store + cached reads
Unique ID Generator	Snowflake-style: timestamp + machine ID + sequence bits

Verification & Practice

System Design

WHEN NOT TO USE THIS PATTERN

Do not introduce distributed components before requirements, scale, failure tolerance, and operational ownership justify them.

EDGE-CASE CHECKLIST

- Single-region and regional failure
- Hot keys and burst traffic
- Partial failure, retries, and duplicate delivery

HIGH-VALUE TEST CASES

- Estimate peak QPS and storage for 10x growth
- Trace one request through a dependency outage
- Prove idempotency under a repeated write

AUTHORITATIVE REFERENCES

<https://sre.google/sre-book/service-level-objectives/>

<https://owasp.org/API-Security/editions/2023/en/0x11-t10/>

<https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html>

SLOs & Error Budgets

Define a small set of user-centered SLIs, attach measurable targets and windows, then use the remaining error budget to guide release risk.

WRITE THE SLO

- Availability: 99.9% of valid requests succeed over 28 days.
- Latency: 99% of successful reads complete within 250 ms over 28 days.
- Correctness/durability: define the exact event, population, exclusions, and measurement point.

CALCULATE THE BUDGET

- Error budget = $1 - \text{SLO}$. A 99.9% availability SLO permits 0.1% unsuccessful valid requests.
- Track burn rate across short and long windows; alert on sustained consumption, not isolated noise.
- State what changes when the budget is exhausted: pause risky releases, prioritize reliability, or narrow scope.

INTERVIEW CHECK

- Is the metric measured where users experience it?
- Are percentiles used for tail latency rather than averages alone?
- Are targets justified by user and business needs rather than current performance?

SOURCES

Google SRE - Service Level Objectives

<https://sre.google/sre-book/service-level-objectives/>

Security & Architecture Decision Review

Test every proposed component against API abuse paths and the six Well-Architected dimensions before calling the design complete.

API-SECURITY CHECKLIST

- Authorize every object and function; never trust an identifier supplied by the client.
- Harden authentication, minimize exposed properties, and validate outbound/third-party API data.
- Limit resource consumption and sensitive business flows; inventory versions and retire stale endpoints.
- Prevent SSRF, secure configuration and secrets, log security events, and test abuse cases.

SIX-PILLAR REVIEW

- Operational excellence: ownership, observability, deployment, rollback, and incident response.
- Security: identity, least privilege, data protection, detection, and recovery.
- Reliability: failure domains, redundancy, backpressure, recovery objectives, and tested restoration.
- Performance efficiency: workload shape, bottlenecks, latency percentiles, and capacity limits.
- Cost optimization: dominant cost drivers, scaling policy, retention, and unit economics.
- Sustainability: utilization, data movement, retention, and unnecessary always-on capacity.

SOURCES

OWASP API Security Top 10 (2023): <https://owasp.org/API-Security/editions/2023/en/0x11-t10/>

AWS Well-Architected Framework: <https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html>