

System Design Interview Questions and Answers

Intern to senior level

System design interview questions and answers: horizontal vs vertical scaling, load balancing, caching, SQL vs NoSQL, CAP theorem, sharding vs replication, rate limiting, REST idempotency, microservices vs monolith and how to design a URL shortener.

11 questions Read online: <https://www.fredevtemplates.com/interview-questions/system-design>

1. What is the difference between horizontal and vertical scaling?

Level: junior System Design

Vertical scaling means giving one machine more CPU, memory or disk, while horizontal scaling means adding more machines and spreading the load across them.

Vertical scaling is the simplest option and needs no code changes, but it has a hard ceiling, usually costs more per unit of capacity at the top end, and leaves a single point of failure. Horizontal scaling has almost no ceiling and improves availability, but the application must be designed for it: stateless servers, shared sessions and caches, and a load balancer in front.

Databases are the hard part. Stateless web servers scale out easily; a relational database is usually scaled up first, then read replicas are added, and finally the data is partitioned (sharded) when one primary can no longer handle the writes.

Key points

Vertical: bigger machine. Horizontal: more machines

Horizontal scaling needs stateless services

Databases are scaled up, then replicated, then sharded

2. What is a load balancer and what are the common load balancing algorithms?

Level: junior System Design

A load balancer distributes incoming requests across several servers to improve capacity and availability, using algorithms such as round robin, least connections or consistent hashing.

Round robin sends each request to the next server in turn and works when requests cost about the same. Least connections sends the request to the server with the fewest active connections, which is better when request times vary. IP hash or consistent hashing sends the same client or key to the same server, which helps with caches and sticky sessions.

A load balancer also runs health checks and stops sending traffic to failed servers. Layer 4 balancers route by IP and port and are very fast, while layer 7 balancers understand HTTP and can route by path, host or header and terminate TLS. The load balancer itself must not be a single point of failure, so it is deployed redundantly.

3. When should you use a SQL database vs a NoSQL database?

Level: mid System Design

Choose SQL when your data is relational and you need transactions, joins and strong consistency; choose NoSQL when you need flexible schemas, very high write throughput or horizontal scaling for a specific access pattern.

Relational databases such as PostgreSQL and MySQL give you ACID transactions, constraints and ad-hoc queries, which suit orders, payments, users and most business applications. They are a safe default and scale a long way with indexes, replicas and caching.

NoSQL is a family: document stores (MongoDB) for flexible, nested records; key-value stores (Redis, DynamoDB) for very fast lookups by key; wide-column stores (Cassandra) for huge write-heavy workloads; graph databases for relationship queries. Pick by access pattern, not fashion. Many systems use both, for example PostgreSQL as the source of truth and Redis for caching and rate limits.

Key points

SQL: relations, transactions, ad-hoc queries

NoSQL: specific access patterns and horizontal scale

Polyglot persistence is common

4. What is caching and what are the common cache invalidation strategies?

Level: mid System Design

Caching stores frequently read data in fast storage such as memory to reduce latency and database load; the main strategies are cache-aside, write-through, write-back and time-based expiry (TTL).

In cache-aside, the application checks the cache first, loads from the database on a miss and then writes the result to the cache. It is the most common pattern. Write-through writes to the cache and the database together, keeping them in sync at the cost of write latency. Write-back writes to the cache first and flushes to the database later, which is fast but risks data loss if the cache fails.

Invalidation is the hard part. A TTL accepts a bounded amount of stale data; explicit invalidation deletes or updates the key whenever the source changes; versioned keys avoid the problem by creating a new key. When the cache is full, an eviction policy such as LRU decides what to remove. Watch for a cache stampede, where many requests rebuild an expired popular key at once, and mitigate it with request coalescing, locks or slightly randomised TTLs.

Cache-aside with a five-minute TTL.

```
async function getProduct(id) {
  const cached = await redis.get("product:" + id);
  if (cached) return JSON.parse(cached);

  const product = await db.products.findById(id);
  await redis.set("product:" + id, JSON.stringify(product), "EX", 300);
  return product;
}
```

5. What is the difference between REST, GraphQL and gRPC?

Level: mid System Design

REST exposes resources over HTTP URLs and is simple and cache-friendly, GraphQL lets clients ask for exactly the fields they need from a single endpoint, and gRPC uses binary Protocol Buffers over HTTP/2 for fast service-to-service calls.

REST is the default for public APIs: it uses standard HTTP methods and status codes, works with browser and CDN caching and is easy to debug. Its weaknesses are over-fetching and under-fetching, which can force clients to make several round trips.

GraphQL solves that for complex front ends, such as mobile apps needing different shapes of data, but caching is harder and you must guard against expensive queries with depth and cost limits. gRPC gives strongly typed contracts, streaming and low latency, which suits internal microservices, but browsers need a proxy to call it directly.

6. What is idempotency in REST APIs and why does it matter?

Level: mid System Design

An operation is idempotent if repeating it any number of times has the same effect as doing it once; it matters because network retries can otherwise create duplicate orders or payments.

In HTTP, GET, PUT and DELETE are defined as idempotent, while POST is not. Clients and gateways retry requests after timeouts, so a non-idempotent payment endpoint could charge a customer twice.

The standard fix is an idempotency key: the client sends a unique key in a header, the server stores the key with the result of the first request, and any repeat with the same key returns the stored result instead of running again. Store the key and the business change in the same transaction, and expire old keys after a sensible period.

```
POST /payments
Idempotency-Key: 7f3c1a2e-9d41-4b6e-8a53-0c1f6d2b9e11
Content-Type: application/json

{ "orderId": 1042, "amount": 4999 }
```

7. How do you design a rate limiter?

Level: mid System Design

Choose an algorithm such as token bucket or sliding window, store a counter per client (by user ID, API key or IP) in a fast shared store like Redis, and return HTTP 429 with a Retry-After header when the limit is exceeded.

A token bucket gives each client a bucket that refills at a steady rate; each request takes one token, and the bucket size allows short bursts. A fixed window counter is simplest but allows double the limit across a window boundary; a sliding window avoids that at the cost of more storage.

In a distributed system the counters must be shared, so use Redis with atomic operations (INCR with EXPIRE, or a Lua script) so that two servers cannot both let the last request through. Decide what happens when Redis is down (fail open to keep serving, or fail closed to protect the system), apply stricter limits to expensive or sensitive endpoints such as login, and return the limit headers so well-behaved clients can slow down.

8. How do you design a URL shortener like bit.ly?

Level: senior System Design

Generate a short unique code for each long URL (for example by base62-encoding a unique ID), store the mapping in a key-value or relational database, serve redirects through a cache because reads far outnumber writes, and record analytics asynchronously.

Clarify requirements first: how many URLs per day, how long links live, whether custom aliases are required and what analytics are needed. A service with 100 million new links per year and a 100:1 read-to-write ratio is read-heavy, which drives the design toward caching and replicas.

API: POST /links creates a link, and GET /{code} responds with a 301 or 302 redirect. Use 302 if you need to count every click, because browsers cache a 301. Code generation: a counter encoded in base62 gives short, collision-free codes (7 characters give about 3.5 trillion combinations); to avoid a single counter bottleneck, hand each app server a range of IDs. Hashing the URL is an alternative but needs collision handling.

Storage: one table keyed by code holding the long URL, owner and expiry. Put Redis in front for hot links, and add read replicas or partition by code as data grows. Send click events to a queue and aggregate them offline so analytics never slow the redirect. Add rate limiting and URL validation to prevent abuse.

Key points

Clarify scale and read/write ratio first

Base62 of a unique ID gives short collision-free codes

Cache the redirect path, make analytics asynchronous

```
const ALPHABET = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

function toBase62(num) {
  let out = "";
  do {
    out = ALPHABET[num % 62] + out;
    num = Math.floor(num / 62);
  } while (num > 0);
  return out;
}

toBase62(125); // "21"
```

9. What is the CAP theorem and what does it mean in practice?

Level: senior System Design

The CAP theorem says that during a network partition a distributed system must choose between consistency (every read sees the latest write) and availability (every request gets a response); you cannot have both while the partition lasts.

Network partitions are a fact of life in distributed systems, so the real choice is what to give up when one happens. A CP system, such as a strongly consistent store using consensus, may refuse requests on the minority side of a partition to avoid returning stale or conflicting data. An AP system, such as Dynamo-style stores, keeps answering and reconciles differences later, giving eventual consistency.

In practice the choice is per operation, not per company. A bank balance transfer needs consistency, while a social media like count can be stale for a few seconds. Mention also that when there is no partition, systems still trade latency against consistency (the PACELC extension).

10. What is the difference between database sharding and replication?

Level: senior System Design

Replication copies the same data to several servers to improve read capacity and availability, while sharding splits the data across servers so each holds only a part, which increases write capacity and total storage.

With replication, one primary accepts writes and replicas follow it. Reads can be spread across replicas and a replica can be promoted if the primary fails, but replicas can lag, so reads from them may be slightly stale, and write throughput is still limited by the single primary.

Sharding partitions rows by a shard key, such as `user_id`, so each shard handles a fraction of the traffic. Choosing the key is the critical decision: a bad key creates hot shards, and cross-shard queries, joins and transactions become expensive or impossible. Range sharding supports range queries but can be uneven; hash sharding spreads load evenly; consistent hashing makes adding shards cheaper. Shard only when replication, caching, indexing and a bigger primary no longer suffice, because it adds permanent operational complexity.

Key points

Replication: copies, for reads and failover

Sharding: partitions, for writes and storage

Shard key choice decides hot spots and query cost

11. Monolith vs microservices: which should you choose?

Level: senior System Design

Start with a well-structured monolith unless you have several teams and clear service boundaries,

because microservices add distributed-system complexity such as network failures, data consistency and operational overhead.

A monolith is one deployable unit: easy to develop, test, debug and deploy, with in-process calls and straightforward transactions. Its problems appear at scale: long build and release cycles, teams blocking each other, and the inability to scale one hot component independently.

Microservices split the system by business capability, so teams can deploy and scale independently and choose different technologies. The costs are real: network latency and partial failure, distributed transactions and eventual consistency, service discovery, monitoring and tracing across services, and a lot more infrastructure. A pragmatic path is a modular monolith with clear internal boundaries, extracting a service only when there is a concrete reason, such as independent scaling or a separate team owning it.

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/system-design>