

Senior Software Engineer Interview Questions and Answers

Senior (5+ years) | All technologies

Senior software engineer interview questions and answers: system design, scalability, concurrency, database internals, architecture trade-offs and technical leadership.

16 questions Read online: <https://www.freedevtemplates.com/interview-questions/level/senior>

JavaScript

1. How do you find and fix memory leaks in JavaScript?

Level: senior JavaScript

Take heap snapshots in Chrome DevTools before and after the suspect action, compare what grew, and fix the retaining reference, which is usually a forgotten listener, timer, global variable or unbounded cache.

JavaScript uses garbage collection based on reachability: an object is freed only when nothing reachable still references it. A leak is memory that your app no longer needs but something still references.

The usual causes are event listeners that are never removed, setInterval callbacks that are never cleared, detached DOM nodes held in a variable, accidental globals, closures that capture large objects, and caches or maps that only grow. In React, a missing cleanup function in useEffect is the most common source.

To diagnose, open the DevTools Memory tab, take a snapshot, repeat the action several times, take another and use the comparison view to see which object types keep growing. The retainers panel shows the chain of references keeping an object alive. Fixes are to remove listeners and timers in cleanup code, use WeakMap or WeakRef for caches keyed by objects, and put a size limit or eviction policy on caches.

Key points

Leak = unneeded but still reachable

Compare heap snapshots to see what keeps growing

Always pair listeners and timers with cleanup

```
useEffect(() => {
  const onResize = () => setWidth(window.innerWidth);
  window.addEventListener("resize", onResize);
  const id = setInterval(refresh, 5000);

  return () => { // cleanup prevents the leak
    window.removeEventListener("resize", onResize);
    clearInterval(id);
  };
}, []);
```

React

2. Why does a React component re-render and how do you prevent unnecessary re-renders?

Level: senior React

A component re-renders when its state changes, its parent re-renders, or a context it consumes changes; you prevent wasted renders by moving state down, using React.memo, and keeping props

referentially stable.

By default, when a component renders, all of its children render too, even if their props did not change. React then compares the output and only updates the DOM where something is different, so a re-render is not necessarily a DOM update, but the work of calling your component still costs time.

Start by fixing the structure: keep state close to where it is used, and split a component so a frequently changing part (such as an input) does not drag a heavy sibling along. Passing children as props also helps, because the same element object does not re-render. If that is not enough, wrap pure components in `React.memo`, stabilise object and function props with `useMemo` and `useCallback`, and split large contexts. Use the React DevTools Profiler to confirm what actually rendered and why before optimising.

Key points

Triggers: own state change, parent render, context change

Move state down and compose with children first

`React.memo` only helps if props are stable

3. How do you optimize performance in a React application?

Level: senior React

Measure first with the Profiler and Lighthouse, then reduce unnecessary renders, split code with lazy loading, virtualise long lists, and optimise images and bundle size.

Rendering: avoid wasted re-renders by colocating state, using `React.memo`, `useMemo` and `useCallback` where profiling shows a benefit, and avoiding creating new objects in props for memoised children. For slow updates triggered by typing or filtering, `useTransition` and `useDeferredValue` keep the interface responsive.

Loading: split the bundle by route with `React.lazy` and `Suspense` (or Next.js dynamic imports), remove unused dependencies, and analyse the bundle with a tool such as `webpack-bundle-analyzer`. Serve properly sized, modern-format images and lazy-load below-the-fold content.

Data and lists: virtualise long lists with a library such as `react-window` so only visible rows are in the DOM, cache server data with `React Query` or `SWR`, and debounce expensive handlers. Track real-user metrics such as Largest Contentful Paint and Interaction to Next Paint, not just lab scores.

Key points

Profile before optimising

Reduce renders, reduce bundle size, reduce DOM size

Use real-user metrics to verify improvements

```
const Reports = React.lazy(() => import("./Reports"));

<Suspense fallback={<Spinner />}>
  <Reports />
</Suspense>
```

HTML & CSS

4. What are Core Web Vitals and how do you improve them?

Level: senior HTML & CSS

Core Web Vitals are Google's field metrics for real-user experience: LCP (loading), INP (interactivity) and CLS (visual stability). They influence search ranking and are measured on real devices, not only Lighthouse.

LCP (Largest Contentful Paint) should happen within about 2.5 seconds: compress the hero image, preload it, and cut render-blocking CSS/JS. INP (Interaction to Next Paint) replaced FID: keep event handlers short, break up long tasks, and show immediate feedback. CLS (Cumulative Layout Shift) should stay below 0.1: always set width and height (or aspect-ratio) on images and embeds, and avoid inserting banners above existing content.

Lab tools (Lighthouse, DevTools Performance) are for debugging. Rankings use field data from the Chrome User Experience Report. Fix the slowest template first, measure with web-vitals or Search Console, and do not chase a 100 Lighthouse score at the cost of features users need.

TypeScript

5. What are mapped types and conditional types in TypeScript?

Level: senior TypeScript

Mapped types build a new object type by looping over keys ({ [K in keyof T]: ... }). Conditional types pick result with $T \text{ extends } U ? X : Y$, which is how many utility types are implemented.

a `Partial<T>` is a mapped type that adds `?` to every key. You can also remap keys with `as` (`[K in keyof T as ${K & string}Id]`). Conditional types power `Exclude`, `Extract` and `Nullable`. They become interesting with `infer`, which pulls a piece out of a type (`T extends Promise<infer U> ? U : T`).

Senior interviews look for judgement: these features belong in shared utility types, not in every component file. If a conditional type is hard to read, a few well-named helpers beat one clever one-liner. Recursion has a depth limit; it is the wrong tool for modelling arbitrarily nested JSON unless you must.

```
type ReadonlyFields<T> = { readonly [K in keyof T]: T[K] };  
type Awaited<T> = T extends Promise<infer U> ? Awaited<U> : T;
```

SQL

6. What are transaction isolation levels in SQL and what problems do they prevent?

Level: senior SQL

Isolation levels (Read Uncommitted, Read Committed, Repeatable Read, Serializable) control how much concurrent transactions can see of each other's changes, trading consistency against concurrency to prevent dirty reads, non-repeatable reads and phantom reads.

A dirty read means reading data another transaction has not committed. A non-repeatable read means reading the same row twice in one transaction and getting different values because someone committed in between. A phantom read means running the same query twice and getting a different set of rows because rows were inserted or deleted.

Read Uncommitted allows all three. Read Committed prevents dirty reads (the default in PostgreSQL, SQL Server and Oracle). Repeatable Read also prevents non-repeatable reads (the default in MySQL InnoDB, and it also prevents most phantoms there). Serializable prevents all three by making transactions behave as if they ran one after another, at the highest cost in locking or retries.

Choose the lowest level that is correct for the business rule, and use explicit locking (`SELECT ... FOR UPDATE`) or optimistic concurrency with a version column for specific hot spots such as inventory or balances.

Key points

Higher isolation = fewer anomalies but less concurrency

Defaults differ by database

Use row locks or version columns for critical updates

7. How do you optimize a slow SQL query?

Level: senior SQL

Read the execution plan with `EXPLAIN` to find full scans and expensive joins, add or fix indexes for the filter and join columns, select only the columns and rows you need, and rewrite predicates so indexes can be used.

Start by measuring, not guessing: run EXPLAIN (or EXPLAIN ANALYZE in PostgreSQL, or the actual execution plan in SQL Server) and look for sequential scans on large tables, big gaps between estimated and actual row counts, sorts that spill to disk, and nested loops over huge inputs.

Typical fixes: add an index matching the WHERE and JOIN columns (and ORDER BY where possible); avoid SELECT *; make predicates sargable (compare the raw column, not a function of it); replace OFFSET pagination on large offsets with keyset pagination (WHERE id > last_seen_id ORDER BY id LIMIT 50); avoid N+1 query patterns from the application; update table statistics; and archive or partition very large tables.

If the query is still too slow, consider a read replica, caching, a materialised view or precomputed summary table. Confirm every change with the plan and with timings on production-sized data.

Key points

Measure with the execution plan first

Indexes, sargable predicates, fewer columns and rows

Keyset pagination beats large OFFSET

```
EXPLAIN ANALYZE
SELECT o.id, o.total
FROM orders o
WHERE o.customer_id = 42
AND o.created_at >= '2026-01-01'
ORDER BY o.created_at DESC
LIMIT 50;

-- keyset pagination instead of a large OFFSET
SELECT id, total FROM orders
WHERE id < 98765
ORDER BY id DESC
LIMIT 50;
```

C#

8. How do you avoid deadlocks with async/await in C#?

Level: senior C#

Use async all the way down instead of blocking on tasks with .Result or .Wait(), and in library code use ConfigureAwait(false) so continuations do not need to return to a captured context.

The classic deadlock happens in UI apps and legacy ASP.NET: the calling thread blocks on task.Result, while the awaited method is waiting to resume on that same captured synchronisation context. Neither can proceed. ASP.NET Core has no such context, so it avoids this particular deadlock, but blocking calls still waste threads and can cause thread-pool starvation under load.

Rules to follow: make the whole call chain async and await it; never call .Result, .Wait() or .GetAwaiter().GetResult() on a task that is not already complete; avoid async void except for event handlers; pass a CancellationToken through your calls; and use ConfigureAwait(false) in general-purpose libraries. Lock statements cannot contain await, so use SemaphoreSlim.WaitAsync() for async mutual exclusion.

Key points

Do not block on async code

Avoid async void

ConfigureAwait(false) in libraries, CancellationToken everywhere

```
// risky: blocks the calling thread
var data = GetDataAsync().Result;

// correct: async all the way
public async Task<IActionResult> Index(CancellationToken ct)
{
    var data = await GetDataAsync(ct);
    return Ok(data);
}
```

Java

9. What is the difference between synchronized and volatile in Java, and how do you avoid deadlocks?

Level: senior Java

synchronized gives mutual exclusion and memory visibility for a block of code, while volatile only guarantees that reads and writes of one variable are visible to all threads and does not make compound operations such as count++ atomic.

Without synchronisation a thread may keep a cached copy of a variable and never see another thread's update. volatile forces reads and writes to go through main memory (and establishes a happens-before relationship), so it is suitable for simple flags such as volatile boolean running. It does not protect count++, which is really read, add, write, and can interleave with another thread.

For compound actions use synchronized, a ReentrantLock, or atomic classes like AtomicInteger and LongAdder, which use compare-and-swap. A deadlock needs four conditions: mutual exclusion, hold and wait, no preemption and circular wait. You prevent it by always acquiring locks in a consistent global order, holding locks for as short a time as possible, using tryLock with a timeout, or avoiding shared mutable state altogether with immutable objects, message passing and higher-level utilities from java.util.concurrent. A thread dump (jstack) reports detected deadlocks.

Key points

volatile: visibility, not atomicity

synchronized / locks: mutual exclusion and visibility

Prevent deadlocks with consistent lock ordering and timeouts

```
private volatile boolean running = true; // visibility only
private final AtomicInteger counter = new AtomicInteger();

void work() {
    while (running) {
        counter.incrementAndGet(); // atomic, no lock
    }
}
```

Python

10. What is the GIL in Python and how does it affect multithreading?

Level: senior Python

The Global Interpreter Lock (GIL) in CPython lets only one thread execute Python bytecode at a time, so threads do not speed up CPU-bound code but still help with I/O-bound work.

The GIL exists to keep CPython's memory management (reference counting) simple and safe. A thread releases it while waiting on I/O such as network calls and file reads, so many I/O-bound threads can overlap their waiting. For pure-Python CPU-bound loops, threads take turns and the program is no faster, sometimes slower.

To use multiple cores, use multiprocessing or `concurrent.futures.ProcessPoolExecutor`, which run separate interpreters, or push the heavy work into C extensions such as NumPy that release the GIL. Recent Python versions have an experimental free-threaded build that removes the GIL, but the default interpreter still has it.

Key points

One thread runs Python bytecode at a time in CPython

Threads are fine for I/O-bound work

Use processes for CPU-bound work

11. What is the difference between threading, multiprocessing and asyncio in Python?

Level: senior Python

Threading runs concurrent threads that share memory and suit blocking I/O, multiprocessing runs separate processes to use multiple CPU cores for CPU-bound work, and asyncio runs many non-blocking tasks cooperatively on one thread for high-volume I/O.

Choose by the type of bottleneck. For CPU-bound work such as image processing or number crunching, use multiprocessing so each worker has its own GIL, and accept the cost of starting processes and passing data between them. For I/O-bound work with libraries that block, such as requests, threads are simple and effective.

For thousands of concurrent network connections, asyncio with `async/await` has the lowest overhead because tasks yield control at each `await`, but every library in the path must be async-aware, and one blocking call stalls the whole event loop. Many production systems combine them, for example an async web server that sends CPU-heavy jobs to a process pool or task queue.

Key points

CPU-bound: multiprocessing

Blocking I/O: threading

Massive concurrent I/O: asyncio

```
import asyncio, httpx

async def fetch(client, url):
    r = await client.get(url)
    return r.status_code

async def main(urls):
    async with httpx.AsyncClient() as client:
        return await asyncio.gather(*(fetch(client, u) for u in urls))

asyncio.run(main(["https://example.com"] * 5))
```

Node.js

12. How do you scale a Node.js application and avoid blocking the event loop?

Level: senior Node.js

Keep the main thread free by moving CPU-heavy work to worker threads or a job queue, and scale across cores and machines by running multiple processes (cluster or PM2) behind a load balancer with shared, external state.

One Node.js process uses one core for your JavaScript. Run one process per core with the `cluster` module, `PM2` or by running several containers behind a load balancer. Because the processes do not share memory, session data, caches and rate-limit counters must move to an external store such as Redis.

Blocking the event loop stalls every request in that process. Common causes are synchronous file or crypto calls (`fs.readFileSync`, `bcrypt` sync versions), parsing huge JSON documents, heavy loops, and regular expressions with catastrophic backtracking. Fix them with async APIs, `worker_threads` for CPU work, queues such as `BullMQ` for background jobs, streaming instead of buffering, and input size limits. Measure event loop delay with

perf_hooks.monitorEventLoopDelay() and alert on it, and make the app stateless so instances can be added or removed freely.

Key points

One process per core, behind a load balancer

Move CPU work to workers or queues

Keep state in Redis or a database, not in process memory

```
import { Worker } from "node:worker_threads";

function runInWorker(data) {
  return new Promise((resolve, reject) => {
    const worker = new Worker("./hash-worker.js", { workerData: data });
    worker.once("message", resolve);
    worker.once("error", reject);
  });
}
```

System Design

13. 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"
```

14. 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).

15. 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

16. 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/level/senior>