

General Software Engineer Interview Questions: Fundamentals, Coding and Design

Software Engineer | Intern to senior level

General software engineer interview questions and answers: programming fundamentals, object-oriented design, coding problems, system design, Git and behavioral questions.

112 questions Read online: <https://www.fredevtemplates.com/interview-questions/software-engineer>

JavaScript

1. What is the difference between var, let and const in JavaScript?

JavaScript Easy Intern, Junior

var is function-scoped and can be redeclared; let and const are block-scoped, and const cannot be reassigned after it is set.

var is function-scoped, is hoisted and initialised to undefined, and can be redeclared in the same scope. let and const are block-scoped: they only exist inside the nearest pair of curly braces.

let and const are also hoisted, but they stay in the temporal dead zone until the line where they are declared, so using them earlier throws a ReferenceError instead of silently giving undefined.

const prevents reassigning the variable, not mutating the value. A const array or object can still be changed by pushing to it or setting a property. Use const by default, let when you need to reassign, and avoid var in new code.

In the interview

I would say: var is function-scoped and hoisted as undefined, while let and const are block-scoped and sit in the temporal dead zone until their declaration runs. const stops reassignment, not mutation, so a const array can still be pushed to. In practice I use const by default, let when a value must change, and avoid var.

Key points

- var: function scope, hoisted as undefined, can be redeclared
- let: block scope, temporal dead zone, cannot be redeclared in the same scope
- const: like let, but the binding cannot be reassigned

```
if (true) {  
  var a = 1;  
  let b = 2;  
}  
console.log(a); // 1 (var ignores the block)  
console.log(b); // ReferenceError: b is not defined  
  
const list = [1, 2];  
list.push(3); // allowed, the array is mutated  
list = []; // TypeError: Assignment to constant variable
```

2. What is the difference between == and === in JavaScript?

JavaScript Easy Intern, Junior

== compares values after type coercion, while === compares both value and type with no conversion, so it is the safer default.

The loose equality operator == converts the operands to a common type before comparing, which leads to

surprising results such as `0 == ''` and `null == undefined` both being true. The strict equality operator `===` returns false immediately if the types differ.

Most teams enforce `===` with a lint rule. The one commonly accepted use of `==` is `value == null`, which checks for both null and undefined in one expression.

Neither operator compares objects by content. Two separate objects or arrays with identical contents are never equal, because both operators compare references.

In the interview

Triple equals compares type and value with no conversion; double equals coerces both sides to a common type first, which produces surprises like `0 == ''` being true. I use `===` everywhere and only allow `== null` as a short way to check for null or undefined.

Key points

- `===` checks type and value, `==` coerces first
- Prefer `===` everywhere except deliberate null checks
- Objects are compared by reference, not by content

```
0 == ''; // true (both coerced to 0)
0 === ''; // false (number vs string)
null == undefined; // true
null === undefined; // false
NaN === NaN; // false (use Number.isNaN)
[1] === [1]; // false (different references)
```

3. What is hoisting in JavaScript?

JavaScript Easy Intern, Junior

Hoisting is JavaScript moving declarations to the top of their scope during compilation, so functions and var variables can be referenced before the line where they are written.

Before running code, the engine processes declarations in each scope. Function declarations are hoisted with their full body, so you can call them before they appear. var variables are hoisted and set to undefined, but their assignment stays where you wrote it.

let, const and class declarations are hoisted too, but not initialised. Accessing them before the declaration throws a ReferenceError (the temporal dead zone). Function expressions assigned to a variable follow the variable's rules, not the function declaration rules.

In the interview

Hoisting means declarations are registered before the code in a scope runs. Function declarations are available with their body, var variables exist but are undefined until assigned, and let, const and class are in the temporal dead zone until their line runs.

Key points

- Declarations are hoisted, assignments are not
- Function declarations are available before their definition
- let and const exist but cannot be used until declared

```
sayHi(); // "hi" - function declaration is fully hoisted
console.log(x); // undefined - var is hoisted, value is not
console.log(y); // ReferenceError - let is in the temporal dead zone

function sayHi() { console.log("hi"); }
var x = 5;
let y = 10;

notYet(); // TypeError: notYet is not a function
var notYet = function () {};
```

4. What is the difference between null and undefined in JavaScript?

JavaScript Easy Intern, Junior

undefined means a variable has been declared but no value was assigned; null is an explicit value meaning 'no value' that you assign on purpose.

JavaScript uses undefined automatically: for uninitialised variables, missing function arguments, missing object properties and functions with no return statement. null is something a developer sets deliberately to say that a value is intentionally empty.

typeof undefined is 'undefined', while typeof null is 'object', a long-standing bug in the language that cannot be fixed without breaking the web. Use value == null to test for both, or optional chaining and nullish coalescing (user?.name ?? 'Guest') to handle them safely.

```
let a; // undefined
const b = null; // null
typeof a; // "undefined"
typeof b; // "object" (historical bug)
a == b; // true
a === b; // false
const name = user?.name ?? "Guest";
```

5. What is the difference between map, filter and reduce in JavaScript?

JavaScript Easy Intern, Junior

map transforms every item into a new array of the same length, filter keeps only items that pass a test, and reduce folds the whole array into a single value.

All three are array methods that do not mutate the original array (when your callback is pure). map returns an array with the same number of items. filter returns a shorter or equal array. reduce takes an accumulator and returns anything: a number, an object, even another array.

They chain well: filter first to reduce the work, then map, then reduce. Forgetting the initial value in reduce is a common bug: on an empty array it throws a TypeError.

Key points

- map: one output per input
- filter: keeps items where the callback returns truthy
- reduce: always pass an initial value

```
const orders = [{ price: 20, paid: true }, { price: 35, paid: false }, { price: 15, paid: true }];

const prices = orders.map(o => o.price); // [20, 35, 15]
const paid = orders.filter(o => o.paid); // 2 orders
const total = orders
  .filter(o => o.paid)
  .reduce((sum, o) => sum + o.price, 0); // 35
```

6. How do you check if a string is a palindrome in JavaScript?

JavaScript Easy Intern, Junior

Normalise the string (lower case, remove non-alphanumeric characters), then compare characters from both ends moving inward with two pointers. This is O(n) time and O(1) extra space beyond the normalised copy.

A palindrome reads the same forwards and backwards. Interviewers usually mean a "valid palindrome" that ignores case, spaces and punctuation, so ask before coding.

The shortest solution is s === s.split("").reverse().join(""), which is fine to mention but builds two extra strings.

The

two-pointer approach shows you understand the problem: compare the first and last characters, move inward, and stop at the first mismatch.

Problem: Given a string *s*, return true if it is a palindrome after converting all letters to lower case and removing all non-alphanumeric characters.

Input: *s* = "A man, a plan, a canal: Panama" -> Output: true

```
function isPalindrome(s) {
  const clean = s.toLowerCase().replace(/[^a-z0-9]/g, '');
  let left = 0;
  let right = clean.length - 1;

  while (left < right) {
    if (clean[left] !== clean[right]) return false;
    left++;
    right--;
  }
  return true;
}

isPalindrome('A man, a plan, a canal: Panama'); // true
isPalindrome('race a car'); // false
```

Time: $O(n)$: every character is processed a constant number of times.

Space: $O(n)$ for the cleaned copy. Skipping invalid characters in place with the pointers makes it $O(1)$.

7. How do you solve the Two Sum problem in JavaScript?

JavaScript Easy Junior, Mid-level

Walk through the array once while storing each number's index in a hash map; for every number, check whether target minus that number is already in the map. This runs in $O(n)$ time and $O(n)$ space.

The brute-force solution checks every pair with two nested loops. It is correct but takes $O(n^2)$ time, which is too slow for large inputs.

The key observation is that for each number *x* you are looking for one specific partner: *target - x*. A Map lets you check whether that partner has already been seen in constant time, so a single pass is enough.

Check for the complement before inserting the current number, otherwise an element could be paired with itself (for example target 6 and value 3 at one index).

Problem: Given an array of integers *nums* and an integer *target*, return the indices of the two numbers that add up to *target*. Assume exactly one solution exists and you may not use the same element twice.

Input: *nums* = [2, 7, 11, 15], *target* = 9 -> Output: [0, 1]

```
function twoSum(nums, target) {
  const seen = new Map(); // value -> index

  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (seen.has(complement)) {
      return [seen.get(complement), i];
    }
    seen.set(nums[i], i);
  }

  return []; // unreachable when a solution is guaranteed
}

twoSum([2, 7, 11, 15], 9); // [0, 1]
```

Time: $O(n)$: each element is visited once and map operations are $O(1)$ on average.

Space: $O(n)$: in the worst case the map stores every element.

8. What is a closure in JavaScript and why is it useful?

JavaScript Medium Junior, Mid-level

A closure is a function that keeps access to the variables of the scope where it was created, even after that outer function has finished running.

Every function in JavaScript carries a reference to the lexical environment it was defined in. When an inner function is returned or passed elsewhere, that environment stays alive for as long as the function can be called, so the inner function can still read and update the outer variables.

Closures give you private state without classes, power patterns such as memoization, debounce and once-only functions, and make callbacks and React hooks work. The trade-off is memory: anything captured by a closure cannot be garbage collected while the closure is reachable.

In the interview

A closure is a function together with the variables from the scope where it was created. Because the inner function keeps a reference to those variables, they stay alive after the outer function returns. I use closures for private state, factories, memoisation and callbacks that need context, and I watch for the memory they keep alive.

Key points

- A closure = a function plus its surrounding scope
- Used for private state, factories, callbacks and memoization
- Captured variables stay in memory while the closure is reachable

```
function createCounter() {  
  let count = 0; // private, not reachable from outside  
  return function () {  
    count += 1;  
    return count;  
  };  
}  
  
const next = createCounter();  
next(); // 1  
next(); // 2
```

9. How does the this keyword work in JavaScript?

JavaScript Medium Junior, Mid-level

this is decided by how a function is called, not where it is written: as a method it is the object before the dot, with new it is the new object, and arrow functions inherit it from the enclosing scope.

For a normal function there are four rules, applied in this order: new Foo() makes this the newly created object; call, apply or bind set it explicitly; obj.method() makes this the object before the dot; otherwise it is undefined in strict mode (or the global object in sloppy mode).

Arrow functions do not have their own this. They use the this of the scope they were defined in, which is why they are handy for callbacks inside methods. The classic bug is passing obj.method as a callback: the call site changes, so this is lost.

Key points

- this depends on the call site
- call, apply and bind set this explicitly
- Arrow functions capture this lexically and cannot be rebound

```
const user = {
  name: "Asha",
  regular() { return this.name; },
  arrow: () => typeof this, // "this" is NOT user here
};

user.regular(); // "Asha"
const detached = user.regular;
detached(); // undefined (or TypeError) - lost context
detached.call({ name: "Ravi" }) // "Ravi"
user.regular.bind(user)(); // "Asha"
```

10. What is the difference between promises and async/await in JavaScript?

JavaScript Medium Junior, Mid-level

async/await is syntax built on top of promises that lets you write asynchronous code that reads like synchronous code; under the hood every async function returns a promise.

A promise represents a value that will be available later and is in one of three states: pending, fulfilled or rejected. You consume it with `.then()` and `.catch()`. `async/await` lets you pause an async function with `await` until a promise settles, and handle errors with ordinary `try/catch`.

They are equivalent in power, but `await` makes sequential logic and error handling much easier to read. A common mistake is awaiting independent calls one after another. Use `Promise.all` to run them in parallel.

In the interview

`async/await` is syntax on top of promises. An async function always returns a promise, and `await` pauses that function until a promise settles, so asynchronous code reads top to bottom. Errors are handled with `try/catch` instead of `.catch`, and for independent work I still use `Promise.all` so the requests run in parallel.

Key points

- async functions always return a promise
- await only pauses the async function, not the whole thread
- Use `Promise.all` for independent work

```
// promises
fetch("/api/user").then(r => r.json()).then(showUser).catch(handleError);

// async/await
async function load() {
  try {
    const res = await fetch("/api/user");
    showUser(await res.json());
  } catch (err) {
    handleError(err);
  }
}

// run independent calls in parallel
const [user, posts] = await Promise.all([getUser(), getPosts()]);
```

11. What is event delegation in JavaScript?

JavaScript Medium Junior, Mid-level

Event delegation means attaching one listener to a parent element and using event bubbling to handle events from its children, instead of adding a listener to each child.

When you click a child element, the event bubbles up through its ancestors. A single listener on the parent can inspect `event.target` (or use `event.target.closest(selector)`) to find which child was clicked and react accordingly.

This uses less memory for long lists and also works for elements added later, because the listener lives on the

parent. It does not work for events that do not bubble, such as focus and blur (use focusin and focusout instead).

```
document.querySelector("#todo-list").addEventListener("click", (event) => {
  const item = event.target.closest("li[data-id]");
  if (!item) return;
  removeTodo(item.dataset.id);
});
```

12. How do you flatten a nested array in JavaScript without using flat()? JavaScript Medium Junior, Mid-level

JavaScript Medium Junior, Mid-level

Recursively walk the array: when an element is an array, flatten it and append the result; otherwise append the element. An iterative version uses an explicit stack to avoid recursion limits.

In real code `arr.flat(Infinity)` does this, and you should say so. Interviewers ask you to write it yourself to see recursion, `Array.isArray` and how you reason about depth.

The recursive solution mirrors the structure of the data and is easy to explain. For very deep nesting it can hit the call-stack limit, so mentioning an iterative stack-based version is a good follow-up.

Problem: Write a function that takes an array that may contain nested arrays to any depth and returns a new, single-level array with all the values in their original order.

Input: `[1, [2, [3, [4]], 5]]` -> Output: `[1, 2, 3, 4, 5]`

```
function flatten(arr) {
  const result = [];
  for (const item of arr) {
    if (Array.isArray(item)) {
      result.push(...flatten(item));
    } else {
      result.push(item);
    }
  }
  return result;
}

flatten([1, [2, [3, [4]], 5]]); // [1, 2, 3, 4, 5]
```

Time: $O(n)$, where n is the total number of values across all nesting levels.

Space: $O(n)$ for the output plus $O(d)$ call-stack depth, where d is the maximum nesting depth.

13. What is the event loop in JavaScript and how does it work? JavaScript Medium Mid-level, Senior

JavaScript Medium Mid-level, Senior

The event loop is the mechanism that lets single-threaded JavaScript handle asynchronous work: it runs the call stack to empty, then runs all queued microtasks (promises), then takes the next task (timers, events) from the queue.

JavaScript runs one piece of code at a time on the call stack. Async operations such as timers, network requests and DOM events are handled by the browser (or Node.js), which places a callback in a queue when they finish. The event loop waits until the call stack is empty and then moves the next callback onto it.

There are two queues that matter. The macrotask queue holds timers, I/O and UI events. The microtask queue holds promise callbacks and `queueMicrotask`. After every task, the engine drains the entire microtask queue before taking the next macrotask, which is why a resolved promise runs before a `setTimeout` of 0.

A long synchronous task blocks everything, including rendering, because nothing else can run until the stack is empty. Splitting heavy work into chunks, using web workers, or yielding to the event loop keeps the page responsive.

In the interview

JavaScript runs on one call stack. When the stack is empty, the event loop first drains the microtask queue (promise callbacks, queueMicrotask) and then takes the next macrotask, such as a timer or an I/O callback. That is why a resolved promise callback runs before a setTimeout of 0, and why long synchronous work blocks rendering and input.

Key points

- One call stack, many queues
- Microtasks (promises) run before the next macrotask (setTimeout)
- Blocking the stack blocks rendering and input

Synchronous code first, then microtasks, then timers.

```
console.log("1");
setTimeout(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
// Output: 1, 4, 3, 2
```

14. What is prototypal inheritance in JavaScript?

JavaScript Medium Mid-level, Senior

In JavaScript every object has an internal link to another object called its prototype, and property lookups walk up this prototype chain until the property is found or the chain ends.

When you read obj.name, the engine checks obj first, then obj's prototype, then that object's prototype, and so on until it reaches null. Methods shared by all instances live on the prototype instead of being copied onto each object, which saves memory.

The class keyword is syntax over this mechanism. class B extends A sets up B.prototype to inherit from A.prototype. You can create an object with a chosen prototype using Object.create(proto), and inspect it with Object.getPrototypeOf(obj).

Key points

- Lookups follow the prototype chain
- class is syntactic sugar over prototypes
- Shared methods live on the prototype, not on each instance

```
const animal = { speak() { return this.name + " makes a sound"; } };
const dog = Object.create(animal);
dog.name = "Rex";
dog.speak(); // "Rex makes a sound"
Object.getPrototypeOf(dog) === animal; // true

class Cat {
  constructor(name) { this.name = name; }
  speak() { return this.name + " meows"; } // lives on Cat.prototype
}
```

15. What is the difference between debounce and throttle in JavaScript?

JavaScript Medium Mid-level, Senior

Debounce waits until the user stops triggering an event for a set time before running the function; throttle runs the function at most once per time interval no matter how often the event fires.

Use debounce when only the final value matters, such as a search box that should call the API after the user pauses typing. Use throttle when you want regular updates during continuous activity, such as scroll position tracking or window resize handling.

Both are built with closures and timers. In React, make sure the debounced function is created once (for example with useMemo or useRef), otherwise every render creates a new timer and the debounce does nothing.

Key points

- Debounce: run after the burst ends
- Throttle: run at a steady maximum rate
- Both rely on closures to keep the timer between calls

```
function debounce(fn, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}

function throttle(fn, limit) {
  let waiting = false;
  return (...args) => {
    if (waiting) return;
    fn(...args);
    waiting = true;
    setTimeout(() => { waiting = false; }, limit);
  };
}
```

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

JavaScript Hard Senior

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);
  };
}, []);
```

Python

17. What is the difference between a list and a tuple in Python?

Python Easy Intern, Junior

A list is mutable and can be changed after creation, while a tuple is immutable; tuples are slightly faster, use less memory and can be used as dictionary keys when their items are hashable.

Use a list for a collection that grows, shrinks or is reordered, such as the items in a shopping cart. Use a tuple for a fixed group of values that belong together, such as a coordinate (x, y) or a database row. Tuples also signal to the reader that the data should not change.

Immutability applies to the tuple itself: if a tuple contains a list, that list can still be modified. A one-item tuple needs a trailing comma, (5,), because (5) is just the number 5 in parentheses.

```
items = [1, 2, 3]
items.append(4) # fine

point = (10, 20)
point[0] = 5 # TypeError: 'tuple' object does not support item assignment

locations = {(10, 20): "home"} # tuples can be dict keys
```

18. What is the difference between is and == in Python?

Python Easy Intern, Junior

== checks whether two objects have equal values, while is checks whether they are the exact same object in memory.

== calls the object's `__eq__` method. is compares identity (the same as comparing `id(a) == id(b)`). Two separate lists with the same content are == but not is.

Use is for singletons, mainly None (if x is None), True and False. Do not use is to compare numbers or strings: CPython caches small integers and some strings, so the result can look right in a quick test and fail in real code.

```
a = [1, 2]
b = [1, 2]
a == b # True (same value)
a is b # False (different objects)
c = a
c is a # True

if result is None:
    ...
```

19. What are *args and **kwargs in Python?

Python Easy Junior, Mid-level

***args collects extra positional arguments into a tuple and **kwargs collects extra keyword arguments into a dictionary, which lets a function accept a variable number of arguments.**

The names args and kwargs are conventions; the stars are what matter. They are useful for wrappers and decorators that must pass any arguments through to another function, and for APIs that accept optional settings.

The same stars work at the call site to unpack: `print(*numbers)` spreads a list into separate arguments and `func(**options)` spreads a dictionary into keyword arguments. The order in a signature is: normal parameters, *args, keyword-only parameters, **kwargs.

```
def report(title, *args, **kwargs):
    print(title, args, kwargs)

report("Sales", 1, 2, region="EU", year=2026)
# Sales (1, 2) {'region': 'EU', 'year': 2026}

nums = [3, 4]
opts = {"sep": "-"}
print(*nums, **opts) # 3-4
```

20. Why should you not use a mutable default argument in Python?

Python Medium Junior, Mid-level

Default argument values are evaluated once when the function is defined, so a default list or dictionary is shared between every call that does not pass its own, causing data to leak between calls.

This is one of the most common Python gotchas. The fix is to use None as the default and create a new object inside the function.

```
def add_item(item, items=[]): # bug: one shared list
    items.append(item)
    return items

add_item("a") # ['a']
add_item("b") # ['a', 'b'] <- surprise

def add_item(item, items=None): # correct
    if items is None:
        items = []
    items.append(item)
    return items
```

21. What is the difference between a shallow copy and a deep copy in Python?

Python Medium Junior, Mid-level

A shallow copy creates a new container but reuses the same nested objects, while a deep copy recursively copies everything so the new object is fully independent.

With a shallow copy (`copy.copy()`, `list.copy()`, slicing `items[:]`), changing a nested list in the copy also changes it in the original, because both point to the same inner object. `copy.deepcopy()` duplicates the nested objects as well, handling circular references, but it is slower and uses more memory.

Plain assignment `b = a` does not copy at all: both names refer to the same object.

```
import copy

original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)

original[0].append(99)
print(shallow[0]) # [1, 2, 99] - shared inner list
print(deep[0]) # [1, 2] - independent
```

22. What is the difference between @staticmethod and @classmethod in Python?

Python Medium Junior, Mid-level

A classmethod receives the class as its first argument (cls) and can create or configure instances, while a staticmethod receives no automatic argument and is just a function placed inside the class namespace.

Use @classmethod for alternative constructors such as Date.from_string(...), and because it uses cls, it works correctly for subclasses. Use @staticmethod for a utility function that is related to the class but needs neither the instance nor the class. An ordinary method receives self, the instance.

```
class User:
    def __init__(self, name):
        self.name = name

    @classmethod
    def from_email(cls, email):
        return cls(email.split("@")[0])

    @staticmethod
    def is_valid_email(email):
        return "@" in email
```

23. What are decorators in Python and how do they work?

Python Medium Mid-level, Senior

A decorator is a function that takes another function and returns a modified version of it, letting you add behaviour such as logging, caching or access control without changing the original code.

The @decorator syntax is shorthand for func = decorator(func). A typical decorator defines an inner wrapper function that runs code before and after calling the original function, and returns that wrapper. Use functools.wraps so the wrapped function keeps its name and docstring.

Decorators rely on functions being first-class objects and on closures. Common built-in ones include @property, @staticmethod, @classmethod and functools.lru_cache. Frameworks use them heavily, for example @app.route("/") in Flask.

```
import functools, time

def timed(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.perf_counter() - start:.3f}s")
        return result
    return wrapper

@timed
def load_data():
    ...
```

24. What are generators and the yield keyword in Python?

Python Medium Mid-level, Senior

A generator is a function that uses yield to produce values one at a time and remembers its state between calls, so it can process large or infinite sequences without holding everything in memory.

Calling a generator function does not run it; it returns an iterator. Each next() call resumes the function until the next yield. A list of ten million numbers takes hundreds of megabytes, while a generator producing them one by one uses almost none.

A generator expression, (x * x for x in range(10**7)), is the lazy counterpart of a list comprehension. Generators

can only be consumed once. They are ideal for reading big files line by line, streaming API pages and building data pipelines.

Key points

- yield pauses the function and keeps local state
- Lazy evaluation saves memory
- A generator can be iterated only once

```
def read_large_file(path):
    with open(path) as f:
        for line in f:
            yield line.strip()

for row in read_large_file("big.csv"):
    process(row) # one line in memory at a time

total = sum(x * x for x in range(1_000_000)) # no list created
```

25. How does Python manage memory and garbage collection?

Python Medium Mid-level, Senior

CPython frees objects mainly through reference counting, deleting an object as soon as nothing refers to it, and uses a cyclic garbage collector to clean up groups of objects that reference each other.

Every object keeps a count of references to it. When the count reaches zero its memory is released immediately. This makes cleanup predictable, but reference counting cannot free cycles, such as two objects that point at each other. The gc module's generational collector periodically finds and frees those.

Memory leaks in Python are usually objects kept alive by long-lived references: global lists or dictionaries, caches without limits, or closures. Use weakref for caches that should not keep objects alive, `__slots__` to shrink many small objects, generators to stream data, and tools such as tracemalloc to see what is allocating memory.

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

Python Hard Senior

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

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

Python Hard Senior

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))
```

Java

28. What is the difference between JDK, JRE and JVM in Java?

Java Easy Intern, Junior

The JVM runs Java bytecode, the JRE is the JVM plus the standard libraries needed to run Java programs, and the JDK is the JRE plus the development tools such as the compiler (javac).

You write source code in .java files, the compiler javac (part of the JDK) turns it into platform-independent bytecode in .class files, and the JVM on any operating system loads and executes that bytecode. That is how Java achieves "write once, run anywhere".

Since Java 11 Oracle and most vendors no longer ship a separate JRE download. You install a JDK, or build a trimmed runtime with jlink. The JVM itself also contains the class loader, the bytecode verifier, the JIT compiler that turns hot bytecode into native code, and the garbage collector.

Key points

- JDK = JRE + development tools
- JRE = JVM + class libraries
- JVM executes bytecode and is platform specific

29. What is the difference between == and equals() in Java?

Java Easy Intern, Junior

== compares primitives by value and objects by reference (whether they are the exact same object), while equals() compares the logical content of two objects when the class overrides it.

For String this surprises many people: two strings with the same text created with new are different objects, so == is false even though equals() is true. String literals may be equal under == only because of the string pool, which you should never rely on.

The default equals() inherited from Object behaves like ==. If you override equals() in your own class, you must also override hashCode() so that equal objects have equal hash codes, otherwise they will be lost in a HashMap

or HashSet.

Key points

- == compares references for objects
- equals() compares content if overridden
- Override hashCode() whenever you override equals()

```
String a = new String("java");
String b = new String("java");
System.out.println(a == b); // false - different objects
System.out.println(a.equals(b)); // true - same content

Integer x = 127, y = 127;
System.out.println(x == y); // true (cached small values) - do not rely on it
```

30. What is the difference between String, StringBuilder and StringBuffer in Java?

Java Easy Intern, Junior

String is immutable, StringBuilder is a mutable and fast but not thread-safe buffer, and StringBuffer is a mutable buffer with synchronized methods that is thread-safe but slower.

Because String is immutable, every concatenation creates a new object, so building text in a loop with += is wasteful. Immutability also makes strings safe to share between threads, cache their hash code and use as HashMap keys, and it is what allows the string pool.

Use StringBuilder for building text in a single thread, which is almost always. StringBuffer is a legacy class from Java 1.0 and is rarely needed, since sharing a builder across threads is uncommon and you usually need your own higher-level synchronisation anyway.

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5; i++) {
    sb.append(i).append(', ');
}
String result = sb.toString(); // "0,1,2,3,4,"
```

31. What is the difference between final, finally and finalize in Java?

Java Easy Intern, Junior

final is a keyword that prevents reassigning a variable, overriding a method or extending a class; finally is a block that always runs after try/catch for cleanup; finalize() was a method called by the garbage collector and is now deprecated.

final on a variable means it can be assigned once (the object it refers to can still change). On a method it cannot be overridden, and on a class it cannot be subclassed (String is final). finally runs whether or not an exception was thrown, which is why it was used to close resources, but try-with-resources is now the preferred way.

finalize() was unreliable because nobody knows when, or whether, the garbage collector will call it. It has been deprecated for removal since Java 18. Use try-with-resources and AutoCloseable, or Cleaner, instead.

```
try (BufferedReader reader = new BufferedReader(new FileReader("data.txt"))) {
    System.out.println(reader.readLine());
} catch (IOException e) {
    log.error("read failed", e);
} // reader is closed automatically
```

32. What is the difference between an abstract class and an interface in Java?

Java Medium Junior, Mid-level

An abstract class can have instance fields, constructors and a mix of implemented and abstract methods but supports only single inheritance, while an interface defines a contract that a class can implement multiple times and, since Java 8, can include default methods.

Choose an abstract class when subclasses share state and base behaviour (for example a BaseRepository with a connection field). Choose an interface when you want to define a capability that unrelated classes can have, such as Comparable or Runnable, and when you want callers to depend on an abstraction.

Interface fields are implicitly public static final. Since Java 8 an interface can have default and static methods, and since Java 9 private methods, which lets library authors add methods without breaking existing implementations. A class can implement many interfaces but extend only one class.

```
interface Payment {
    void pay(double amount);
    default String currency() { return "USD"; }
}

abstract class BasePayment implements Payment {
    protected final String accountId;
    BasePayment(String accountId) { this.accountId = accountId; }
}
```

33. What is the difference between ArrayList and LinkedList in Java?

Java Medium Junior, Mid-level

ArrayList is backed by a resizable array with $O(1)$ index access and $O(n)$ insertion in the middle, while LinkedList is a doubly linked list with $O(n)$ index access and $O(1)$ insertion once you hold the node; in practice ArrayList is faster for most workloads.

ArrayList stores elements contiguously, which is cache-friendly, so iteration and `get(i)` are very fast. Adding at the end is amortised $O(1)$; inserting or removing in the middle shifts the following elements.

LinkedList stores each element in a node with previous and next pointers, which costs extra memory and scatters nodes around the heap. Finding the element at position `i` requires walking the list. It only wins when you repeatedly add or remove at the ends or through an iterator. For queue and deque behaviour, ArrayDeque is usually a better choice than LinkedList.

Key points

- ArrayList: $O(1)$ get, amortised $O(1)$ add at end
- LinkedList: $O(n)$ get, extra memory per element
- Default to ArrayList; use ArrayDeque for queues

34. What is the difference between checked and unchecked exceptions in Java?

Java Medium Junior, Mid-level

Checked exceptions (subclasses of Exception other than RuntimeException) must be caught or declared with throws, while unchecked exceptions (RuntimeException and Error) do not have to be handled and usually signal programming bugs.

Checked exceptions such as IOException and SQLException represent failures the caller can reasonably be expected to handle or recover from, so the compiler forces you to deal with them. Unchecked exceptions such as NullPointerException, IllegalArgumentException and IndexOutOfBoundsException indicate bugs, and fixing the code is better than catching them. Error (for example OutOfMemoryError) signals serious JVM problems you should not catch.

Good practice: never swallow an exception silently, catch the most specific type, add context when you wrap it,

and keep the original as the cause (new ServiceException("msg", e)). Many modern frameworks, including Spring, prefer unchecked exceptions because checked ones clutter method signatures.

35. What are the main features of Java 8: lambdas, streams and Optional?

Java Medium Junior, Mid-level

Java 8 introduced lambda expressions, functional interfaces, the Stream API for declarative collection processing, Optional to represent missing values, default methods in interfaces, method references and the java.time date and time API.

A lambda is a short way to write an instance of a functional interface (an interface with one abstract method), such as Comparator or Runnable. The Stream API lets you describe a pipeline: a source, zero or more intermediate operations such as filter and map that are lazy, and a terminal operation such as collect that triggers the work.

Use Optional as a return type to make absence explicit instead of returning null, but not for fields or method parameters. Streams do not modify the source collection, and parallel streams should be used with care: they only help for large, CPU-bound, stateless operations.

Key points

- Lambdas implement functional interfaces
- Stream intermediate operations are lazy until a terminal operation runs
- Optional is for return values, not fields or parameters

```
List<String> names = users.stream()
    .filter(u -> u.isActive())
    .map(User::getName)
    .sorted()
    .collect(Collectors.toList());

Optional<User> user = repository.findById(id);
String email = user.map(User::getEmail).orElse("unknown");
```

36. How does HashMap work internally in Java?

Java Medium Mid-level, Senior

A HashMap stores entries in an array of buckets; it computes the key's hash code to choose a bucket, resolves collisions with a linked list that becomes a red-black tree when a bucket grows large, and resizes when the load factor of 0.75 is exceeded.

On put(key, value), HashMap calls key.hashCode(), spreads the bits, and uses hash & (capacity - 1) to pick a bucket index. If the bucket is empty, the entry is stored there. If not, it walks the bucket comparing keys with equals(): a matching key has its value replaced, otherwise the new entry is added to the end. That is why correct hashCode() and equals() implementations are essential for keys.

The default capacity is 16 and the load factor is 0.75, so when more than 12 entries are stored, the table doubles and entries are redistributed. Since Java 8, a bucket with 8 or more entries (when the table has at least 64 buckets) is converted from a linked list to a red-black tree, which keeps worst-case lookup at O(log n) instead of O(n). Average get and put are O(1). HashMap allows one null key, does not guarantee ordering and is not thread-safe.

Key points

- Index = hash of key mapped onto the bucket array
- Collisions: linked list, treeified at 8 entries
- Resizes at load factor 0.75; not thread-safe

37. What is the difference between HashMap, Hashtable and ConcurrentHashMap in Java?

Java Hard Mid-level, Senior

HashMap is not thread-safe, Hashtable is thread-safe by locking the whole map on every method, and ConcurrentHashMap is thread-safe with much finer-grained locking and lock-free reads, so it scales far better.

Using a plain HashMap from several threads can lose updates or, in older versions, even loop forever during a resize. Hashtable and Collections.synchronizedMap() fix this by synchronising every operation on a single lock, so threads queue up behind each other.

ConcurrentHashMap locks only the bucket being modified (using CAS and synchronized on the bucket head since Java 8) and allows reads without locking, so many threads can work in parallel. It does not allow null keys or values. Its compound atomic methods such as computeIfAbsent, merge and putIfAbsent should be used instead of a separate get followed by put, which is a race condition.

```
ConcurrentHashMap<String, Integer> hits = new ConcurrentHashMap<>();
hits.merge("home", 1, Integer::sum); // atomic increment

// racy even on a ConcurrentHashMap:
// if (!map.containsKey(k)) map.put(k, v);
```

38. How does JVM memory management and garbage collection work in Java?

Java Hard Mid-level, Senior

The JVM divides memory into a per-thread stack, a shared heap (split into young and old generations) and Metaspace; the garbage collector frees heap objects that are no longer reachable, collecting short-lived young objects frequently and old objects less often.

Each thread has its own stack holding method frames with local variables and references. All objects live on the heap. New objects are allocated in the young generation (Eden and two survivor spaces); objects that survive several minor collections are promoted to the old generation. Class metadata lives in Metaspace, which replaced PermGen in Java 8.

An object is garbage when it can no longer be reached from a GC root such as a thread stack, static field or JNI reference. G1 is the default collector since Java 9 and balances throughput with pause times; ZGC and Shenandoah target very low pauses on large heaps. Typical production problems are OutOfMemoryError from leaks (static collections, listeners, ThreadLocals not cleared) and long pauses from undersized or oversized heaps. Diagnose them with GC logs, heap dumps and tools such as VisualVM or Eclipse MAT.

Key points

- Stack per thread, heap shared, Metaspace for class data
- Generational collection: young then old
- Leaks are objects still reachable by mistake

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

Java Hard Senior

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
    }
}
```

C#

40. What is the difference between a value type and a reference type in C#?

C# Easy Intern, Junior

A value type variable holds its data directly and is copied on assignment, while a reference type variable holds a reference to an object, so assignment copies the reference and both variables see the same object.

Value types include `int`, `double`, `bool`, `char`, `DateTime`, enums and any struct. Reference types include `class`, `string`, arrays, delegates and `object`. A value type is stored inline where it is declared: on the stack for a local variable, or inside the containing object on the heap. A reference type's object always lives on the managed heap and is cleaned up by the garbage collector.

This changes behaviour when you pass values to methods. Passing a struct hands the method its own copy, so changes inside do not leak out. Passing a class passes a copy of the reference, so the method can modify the same object. Reference types can be null; value types cannot, unless you use `Nullable<T>` (int?).

Key points

- Value types are copied, reference types share one object
- Reference types can be null, value types need `int?` syntax
- Large mutable structs are usually a bad idea

```
struct PointS { public int X; }
class PointC { public int X; }

var s1 = new PointS { X = 1 }; var s2 = s1; s2.X = 99;
Console.WriteLine(s1.X); // 1 (copy)

var c1 = new PointC { X = 1 }; var c2 = c1; c2.X = 99;
Console.WriteLine(c1.X); // 99 (same object)
```

41. What is the difference between string and StringBuilder in C#?

C# Easy Intern, Junior

A string is immutable, so every change creates a new string object, while `StringBuilder` is a mutable buffer that modifies its contents in place, making it much faster for repeated concatenation.

Each time you do `s += "x"` in a loop, the runtime allocates a new string and copies the old content, which is $O(n^2)$ work overall and creates a lot of garbage. `StringBuilder` keeps an internal character buffer that grows as needed, so appending is cheap and you call `ToString()` once at the end.

For a handful of concatenations, a plain string or string interpolation is clearer and just as fast. Use `StringBuilder` when building text in a loop or from many pieces.

```
var sb = new StringBuilder();
for (int i = 0; i < 10_000; i++)
{
    sb.Append(i).Append(' ');
}
string result = sb.ToString();
```

42. What is the difference between `const` and `readonly` in C#?

C# Easy Junior, Mid-level

A `const` is a compile-time constant that must be assigned where it is declared, while a `readonly` field can be assigned at declaration or in the constructor, so its value can be decided at runtime.

`const` values are baked into the calling code at compile time and are implicitly static. They can only be primitive types, strings or null. If you change a `const` in a library, every project using it must be recompiled to see the new value.

`readonly` fields are normal fields that cannot be reassigned after the constructor finishes. They can hold any type, can differ per instance, and are read at runtime. Note that `readonly` makes the reference unchangeable, not the object it points to. For compile-time values that will never change (such as the number of days in a week) use `const`; otherwise prefer `readonly`.

```
public class Config
{
    public const int MaxRetries = 3; // compile-time
    public readonly DateTime CreatedAt; // runtime, per instance

    public Config() { CreatedAt = DateTime.UtcNow; }
}
```

43. What is the difference between `ref` and `out` parameters in C#?

C# Easy Junior, Mid-level

Both pass an argument by reference, but `ref` requires the variable to be initialised before the call, while `out` requires the method to assign a value before it returns.

Use `ref` when the method both reads and modifies the caller's variable. Use `out` when the method only produces a value, such as the `bool TryParse(string, out int result)` pattern, which lets a method return a success flag and a value together. A third modifier, `in`, passes by reference as read-only, useful to avoid copying large structs.

In modern C# you can often return a tuple instead `((bool ok, int value))`, which is clearer than `out` parameters.

```

if (int.TryParse(input, out int number))
{
    Console.WriteLine(number * 2);
}

static void AddTen(ref int x) => x += 10;
int n = 5;
AddTen(ref n); // n is now 15

```

44. What is the difference between an abstract class and an interface in C#?

C# Medium Junior, Mid-level

An abstract class can hold state, constructors and partial implementation but a class can inherit only one, while an interface defines a contract that a class can implement many of.

Use an abstract class when related classes share fields, constructor logic or common behaviour, and you want to force subclasses to fill in the rest. A class can extend only one base class, so this is a strong coupling.

Use an interface to describe a capability (IDisposable, IComparable, IEmailSender) that unrelated types can implement, and to make code testable and replaceable through dependency injection. Since C# 8 interfaces may include default method implementations, but they still cannot hold instance fields. A common guideline: start with an interface for the contract, and add an abstract base class only if there is real shared code.

Key points

- Abstract class: shared state and code, single inheritance
- Interface: contract, multiple implementation
- Interfaces make dependency injection and testing easier

```

public interface INotifier { void Send(string message); }

public abstract class NotifierBase : INotifier
{
    protected readonly ILogger Logger;
    protected NotifierBase(ILogger logger) => Logger = logger;
    public abstract void Send(string message);
}

```

45. What is boxing and unboxing in C# and why does it affect performance?

C# Medium Junior, Mid-level

Boxing converts a value type to object by allocating it on the heap, and unboxing converts it back; both add allocations and casts, so avoid them in hot code paths.

Assigning an int to an object variable boxes it: the runtime allocates a new object on the heap and copies the value in. Casting back with (int)obj is unboxing, which checks the type and copies the value out. Boxing in a loop creates garbage that the collector has to clean.

Generics were introduced to avoid this: List<int> stores ints directly, whereas the old ArrayList boxed every item. Watch for hidden boxing when a struct is used as an interface type, or when you pass a value type to a method taking object, such as older string.Format overloads.

```

int i = 42;
object boxed = i; // boxing: heap allocation
int j = (int)boxed; // unboxing: type check + copy

var list = new List<int>(); // no boxing
list.Add(i);

```

46. What is LINQ in C# and what is deferred execution?

C# Medium Junior, Mid-level

LINQ (Language Integrated Query) lets you query collections, databases and XML with a consistent syntax, and deferred execution means a query does not run when you define it but when you enumerate the results.

Methods such as `Where`, `Select` and `OrderBy` only build up a description of the query. The work happens when you iterate with `foreach` or call an operator that forces evaluation, such as `ToList()`, `ToArray()`, `Count()` or `First()`.

This is efficient, but it has traps: if the source changes between defining and enumerating the query, you see the new data, and enumerating the same query twice runs it twice (a database query would hit the database twice). Call `ToList()` when you want a snapshot or need to reuse the results.

```
var numbers = new List<int> { 1, 2, 3 };
var evens = numbers.Where(n => n % 2 == 0); // nothing runs yet

numbers.Add(4);
Console.WriteLine(string.Join(", ", evens)); // 2,4 (runs now)

var snapshot = numbers.Where(n => n > 1).ToList(); // runs immediately
```

47. What is the difference between IEnumerable and IQueryable in C#?

C# Medium Mid-level, Senior

IEnumerable runs the query in memory using delegates, while IQueryable builds an expression tree that a provider such as Entity Framework translates into SQL, so filtering happens in the database.

If you call `.Where(...)` on an `IQueryable<Customer>` from Entity Framework, the condition becomes part of the SQL WHERE clause and only matching rows come over the network. If you call it on an `IEnumerable<Customer>`, all rows are loaded first and then filtered in your application's memory.

A common bug is converting to `IEnumerable` or calling `ToList()` too early, which silently loads the full table. Keep the query as `IQueryable` until you have applied all filtering, sorting and paging, and only then materialise it. Remember that an `IQueryable` can only contain operations the provider can translate to SQL.

Key points

- `IQueryable`: expression trees translated to SQL
- `IEnumerable`: in-memory iteration with delegates
- Materialise (`ToList`) as late as possible

```
// SQL: SELECT ... WHERE IsActive = 1 (filtered in the database)
var active = dbContext.Customers.Where(c => c.IsActive).ToList();

// loads ALL customers first, then filters in memory
IEnumerable<Customer> all = dbContext.Customers.ToList();
var slow = all.Where(c => c.IsActive).ToList();
```

48. What are transient, scoped and singleton service lifetimes in ASP.NET Core dependency injection?

C# Medium Mid-level, Senior

Transient services are created every time they are requested, scoped services are created once per HTTP request, and singleton services are created once and shared for the life of the application.

Register services in `Program.cs` with `AddTransient`, `AddScoped` or `AddSingleton`. Use transient for lightweight, stateless services, scoped for things tied to a request such as an Entity Framework `DbContext`, and singleton for shared, thread-safe objects such as caches and configuration.

The classic bug is a captive dependency: a singleton that depends on a scoped service keeps that one instance alive forever, so a DbContext ends up shared across requests and threads. ASP.NET Core can detect this in development with scope validation. Singletons must also be thread-safe because many requests use them at once.

Key points

- Transient: new every time. Scoped: per request. Singleton: one for the app
- Never inject a scoped service into a singleton
- Singletons must be thread-safe

```
builder.Services.AddTransient<IEmailBuilder, EmailBuilder>();
builder.Services.AddScoped<AppDbContext>();
builder.Services.AddSingleton<IMemoryCache, MemoryCache>();

public class OrdersController(AppDbContext db, IEmailBuilder email) : ControllerBase { }
```

49. What is the difference between Task and Thread, and how does async/await work in C#?

C# Medium Mid-level, Senior

A Thread is an operating-system execution unit, while a Task represents an asynchronous operation scheduled on the thread pool; await releases the current thread while the operation is pending and resumes the method when it completes.

Creating threads is expensive and you must manage them yourself. A Task is a higher-level abstraction: it can represent CPU work queued to the thread pool (Task.Run) or I/O that does not need a thread at all while waiting, such as an HTTP call or database query.

When the compiler sees await, it turns the method into a state machine. If the awaited task is not finished, the method returns control to its caller and the thread is free to serve other requests. When the task completes, the rest of the method continues, possibly on another thread. This is why async I/O lets a web server handle many more concurrent requests with the same number of threads. async/await is about scalability for I/O-bound work; for CPU-bound work it does not make anything faster by itself.

Key points

- Task = promise of work, not a thread
- await frees the thread during I/O
- Use Task.WhenAll for independent operations

```
public async Task<string> GetPageAsync(HttpClient client, string url)
{
    // thread is released while waiting on the network
    string html = await client.GetStringAsync(url);
    return html.ToUpperInvariant();
}

// run independent calls concurrently
var results = await Task.WhenAll(GetPageAsync(c, a), GetPageAsync(c, b));
```

50. How does garbage collection work in C# and when should you use IDisposable?

C# Medium Mid-level, Senior

The .NET garbage collector automatically frees managed objects that are no longer reachable, organising them into generations 0, 1 and 2 so short-lived objects are collected cheaply, and IDisposable is used to release unmanaged resources such as files and connections promptly.

New objects start in generation 0. Objects that survive a collection are promoted to generation 1 and then to

generation 2. Collecting generation 0 is fast and frequent, while generation 2 collections are expensive. Objects of roughly 85,000 bytes or more go on the large object heap, which is collected with generation 2, so allocating many large temporary arrays can hurt performance.

The GC knows about memory, not about files, sockets or database connections. Types that hold those implement `IDisposable`, and you should wrap them in a using statement so the resource is released deterministically, instead of waiting for a finalizer. Common memory problems are not classic leaks but objects kept reachable by static fields, event handlers that are never unsubscribed, and caches without limits.

Key points

- Generational GC: gen 0, 1, 2 and the large object heap
- Dispose releases unmanaged resources; GC releases memory
- Static references and event handlers keep objects alive

```
await using var connection = new SqlConnection(connectionString);
await connection.OpenAsync();
// connection is disposed (and returned to the pool) at the end of the scope
```

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

C# Hard Senior

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);
}
```

Git

52. What is Git and how is it different from GitHub?

Git Easy Intern, Junior

Git is a distributed version control system that tracks changes to files on your machine; GitHub is a hosting service for Git repositories that adds collaboration features such as pull requests, code review and CI.

Git records snapshots of a project as commits. Every clone holds the full history, so you can commit, branch, view logs and compare versions without a network connection. Branches are cheap pointers to commits, which is what makes feature branches and experiments practical.

GitHub, GitLab and Bitbucket host a copy of the repository on a server so a team can share it. On top of Git they add pull requests, code review, issue tracking, permissions and automation such as GitHub Actions. You can use Git without GitHub, but not the other way round.

In day-to-day work you commit locally with Git, push your branch to the hosted remote, open a pull request for review, and merge it once checks pass.

In the interview

Git is the version control tool that runs locally and stores the full history in every clone. GitHub is a service that hosts Git repositories and adds pull requests, reviews, issues and CI on top. I use Git to commit and branch, and GitHub to share work and get it reviewed.

Key points

- Git: the version control tool, works offline
- GitHub: a hosting and collaboration platform for Git repositories
- Every Git clone contains the full history

```
git init # start tracking a project
git add src/app.js # stage a change
git commit -m "Add login form"
git remote add origin https://github.com/you/project.git
git push -u origin main # share it on GitHub
```

53. What is the difference between git fetch and git pull?

Git Easy Intern, Junior

git fetch downloads new commits from the remote without changing your working branch; git pull runs a fetch and then merges (or rebases) those commits into your current branch.

git fetch updates your remote-tracking branches such as origin/main. Your own branch and working files are untouched, so it is always safe. You can then inspect what changed with git log main..origin/main before deciding how to integrate it.

git pull is git fetch followed by git merge by default. If your branch and the remote have both moved, it creates a merge commit or stops with conflicts. Many teams set git config pull.rebase true so pull replays your local commits on top of the remote instead, which keeps history linear.

A good habit before starting work is to fetch, review, then merge or rebase deliberately, rather than pulling blindly into a branch with uncommitted changes.

Key points

- fetch: download only, never touches your branch
- pull: fetch + merge (or rebase with --rebase)
- Fetch first when you want to review incoming changes

```
git fetch origin
git log --oneline main..origin/main # what is new on the remote?
git merge origin/main # integrate when ready

# or in one step
git pull --rebase origin main
```

54. What is the difference between git merge and git rebase?

Git Medium Junior, Mid-level

Both integrate changes from one branch into another: merge keeps both histories and joins them with a merge commit, while rebase rewrites your commits so they appear on top of the target branch, giving a linear history.

Merging main into a feature branch creates a merge commit with two parents. Nothing is rewritten, so it is safe on shared branches, and the history shows exactly when branches were combined. The cost is a busier history with many merge commits.

Rebasing a feature branch onto main takes each of your commits and replays it on the latest main, creating new commits with new hashes. The result reads as if you started your work from the current main, which makes git log and git bisect easier to follow.

Because rebase rewrites history, the golden rule is: never rebase commits that other people have already based work on. Rebase your private feature branch to tidy it up, then merge it (often with a squash or fast-forward) into the shared branch.

In the interview

Merge combines two branches with a merge commit and never rewrites history, so it is safe on shared branches. Rebase replays my commits on top of the target branch, which gives a clean linear history but changes commit hashes. I rebase my own feature branch to keep it current and tidy, and I never rebase a branch someone else is working on.

Key points

- Merge preserves history and is safe on shared branches
- Rebase rewrites commits to produce a linear history
- Never rebase commits that others already use
- Use --force-with-lease, not --force, after rebasing

```
# merge: keeps history, adds a merge commit
git switch feature
git merge main

# rebase: replays feature commits on top of main
git switch feature
git rebase main
git push --force-with-lease # needed because hashes changed
```

55. How do you resolve a merge conflict in Git?

Git Medium Junior, Mid-level

Open the conflicted files, decide what the final code should be between the conflict markers, remove the markers, run the tests, then stage the files and continue the merge or rebase.

A conflict happens when two branches change the same lines, or one deletes a file the other edited. Git pauses the merge and marks each conflicted region with <<<<<<, ===== and >>>>>>. git status lists the files that need attention.

Resolving is not choosing a side. Read both changes, understand why each was made, and write the code that keeps the intent of both. Editors and git mergetool show the two versions side by side, but the decision is yours.

After editing, build and run the tests, because a conflict-free file can still be logically broken. Then git add the resolved files and run git merge --continue or git rebase --continue. If it goes wrong, git merge --abort returns you to the state before the merge.

Key points

- git status shows conflicted files
- Combine the intent of both changes, then remove the markers
- Run tests before continuing
- git merge --abort or git rebase --abort to start over

After editing: git add config.js && git merge --continue

```

<<<<<< HEAD
const timeout = 5000;
=====
const timeout = Number(process.env.TIMEOUT_MS);
>>>>>> feature/config

# resolved: keep the env variable, fall back to the old default
const timeout = Number(process.env.TIMEOUT_MS ?? 5000);

```

56. What is the difference between git reset and git revert?

Git Medium Mid-level, Senior

git revert creates a new commit that undoes an earlier one and is safe on shared branches; git reset moves the branch pointer back to an earlier commit, effectively removing later commits from that branch's history.

git revert <commit> calculates the inverse of a commit and records it as a new commit. History only grows, so teammates who already pulled are not affected. This is how you undo a bad change that has reached main or production.

git reset moves the current branch to another commit. With --soft the undone changes stay staged, with the default --mixed they stay in your working tree unstaged, and with --hard they are discarded. It is ideal for cleaning up local commits that have not been pushed.

If you reset by mistake, the commits are usually still recoverable for a while through git reflog, which records where your branch pointed before each move.

Key points

- revert: new commit, history preserved, safe when shared
- reset: moves the branch pointer, rewrites history
- --soft keeps changes staged, --mixed unstaged, --hard discards them
- git reflog can recover "lost" commits

```

# undo a pushed commit safely
git revert a1b2c3d

# rewrite local history (not yet pushed)
git reset --soft HEAD~1 # undo commit, keep changes staged
git reset --hard HEAD~1 # undo commit and discard changes

# recover from a bad reset
git reflog
git reset --hard HEAD@{2}

```

Docker

57. What is the difference between a Docker container and a virtual machine?

Docker Easy Intern, Junior

A virtual machine runs a full guest operating system on virtualised hardware, while a container shares the host's kernel and isolates only the process, its filesystem and resources, which makes containers much lighter and faster to start.

A hypervisor gives each virtual machine its own virtual CPU, memory and disk, and each VM boots a complete operating system. That gives strong isolation and lets you run different operating systems side by side, but each VM costs gigabytes and takes time to boot.

A container is an ordinary process on the host that the Linux kernel isolates with namespaces (its own view of processes, network and filesystem) and limits with cgroups (CPU and memory). Because there is no guest

kernel, containers start in milliseconds to seconds and image sizes are often tens of megabytes.

The trade-off is isolation: containers share the kernel, so a kernel vulnerability can affect every container on the host. In practice the two are combined: cloud providers run containers inside VMs.

In the interview

A VM virtualises hardware and runs its own operating system, so it is heavy but strongly isolated. A container is just an isolated process that shares the host kernel, using namespaces and cgroups, so it starts quickly and uses far fewer resources. I use containers to package applications consistently, and they usually run inside VMs in the cloud anyway.

Key points

- VM: full guest OS on a hypervisor
- Container: isolated process sharing the host kernel
- Namespaces isolate, cgroups limit resources
- Containers are lighter; VMs isolate more strongly

58. What is the difference between a Docker image and a container?

Docker Easy Intern, Junior

An image is a read-only template made of layers that contains the application and its dependencies; a container is a running (or stopped) instance of an image with its own writable layer on top.

An image is built from a Dockerfile. Each instruction creates a layer, and layers are cached and shared between images, which makes builds and pulls faster. Images are versioned with tags and stored in registries such as Docker Hub or a private registry.

Running `docker run` creates a container from an image: Docker adds a thin writable layer, sets up networking and starts the process. You can run many containers from the same image, just as you create many objects from one class.

Changes made inside a container live only in its writable layer and disappear when the container is removed. Persistent data belongs in volumes, and changes to the application belong in a new image.

Key points

- Image: read-only, layered template
- Container: running instance with a writable layer
- Many containers can share one image
- Container changes are lost on removal; use volumes for data

```
docker build -t my-api:1.2.0 . # create an image
docker run -d --name api1 my-api:1.2.0 # start a container from it
docker run -d --name api2 my-api:1.2.0 # a second container, same image
docker ps # running containers
docker images # local images
```

59. What are Docker volumes and why do you need them?

Docker Easy Junior, Mid-level

Volumes are storage managed by Docker that lives outside a container's writable layer, so data such as database files survives when the container is removed or replaced.

A container's writable layer is deleted with the container, and it is slower than native storage. Anything that must persist, such as a database, uploaded files or caches you want to keep, needs to live outside it.

There are three options. Named volumes are created and managed by Docker and are the default choice for persistent data. Bind mounts map a host directory into the container, which is handy in development for live-reloading source code. tmpfs mounts keep data in memory only.

Volumes are also how containers share data, and they can be backed by network storage in production. Back

them up like any other data: removing the container does not remove a named volume, but docker volume rm or docker compose down -v does.

Key points

- Container writable layers are temporary
- Named volumes for persistent data, bind mounts for development
- Volumes outlive containers until explicitly removed

```
docker volume create pgdata
docker run -d --name db \
-e POSTGRES_PASSWORD=secret \
-v pgdata:/var/lib/postgresql/data \
postgres:16

# the data survives replacing the container
docker rm -f db
docker run -d --name db -e POSTGRES_PASSWORD=secret \
-v pgdata:/var/lib/postgresql/data postgres:16
```

60. How do you write a good Dockerfile? What are multi-stage builds?

Docker Medium Junior, Mid-level

Use a small, pinned base image, order instructions so dependencies are cached before source code, use multi-stage builds to leave build tools out of the final image, run as a non-root user and keep secrets out of the image.

Layer caching drives build speed. Copy the dependency manifest and install dependencies first, then copy the source code. Changing a source file then reuses the cached dependency layer instead of reinstalling everything.

A multi-stage build uses one stage with compilers and dev dependencies to build the app, then copies only the output into a slim runtime stage. The final image is smaller, starts faster and has a smaller attack surface.

Other habits: pin base image versions, add a .dockerignore (exclude node_modules, .git, local env files), run as a non-root user, use CMD in exec form so signals reach your process, and pass secrets at runtime or with build secrets rather than ENV or COPY.

Key points

- Copy dependency files before source code for caching
- Multi-stage builds keep build tools out of production images
- Pin versions, use .dockerignore, run as non-root
- Never bake secrets into image layers

```
# build stage
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# runtime stage
FROM node:20-alpine
WORKDIR /app
ENV NODE_ENV=production
COPY package*.json ./
RUN npm ci --omit=dev
COPY --from=build /app/dist ./dist
USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

61. What is Docker Compose used for?

Docker Medium Mid-level, Senior

Docker Compose defines a multi-container application (services, networks and volumes) in one YAML so you can start, stop and rebuild the whole stack with a single command, typically for local development and testing.

Most applications need more than one container: an API, a database, a cache, perhaps a worker. Starting each with long docker run commands is error-prone. Compose describes them declaratively in compose.yaml.

Services on the same Compose network reach each other by service name, so the API connects to postgres://db:5432 rather than an IP address. depends_on with health checks controls start-up order, and environment variables can come from an .env file.

Compose is ideal for development, CI and small single-host deployments. For production across many machines, teams move to an orchestrator such as Kubernetes or a managed container service, which adds scheduling, self-healing and rolling updates.

Key points

- One file describes services, networks and volumes
- Services find each other by name
- Great for development, CI and single-host setups
- Kubernetes or managed services for multi-host production

```
services:
  api:
    build: .
    ports: ["3000:3000"]
    environment:
      DATABASE_URL: postgres://app:secret@db:5432/app
    depends_on:
      db:
        condition: service_healthy
    db:
      image: postgres:16
      environment:
        POSTGRES_USER: app
        POSTGRES_PASSWORD: secret
      volumes: [pgdata:/var/lib/postgresql/data]
      healthcheck:
        test: ["CMD-SHELL", "pg_isready -U app"]
        interval: 5s
      volumes:
        pgdata:
```

System Design

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

System Design Easy Junior, Mid-level

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
-

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

System Design Medium Junior, Mid-level

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.

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

System Design Medium Mid-level, Senior

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
-

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

System Design Medium Mid-level, Senior

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;
}
```

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

System Design Medium Mid-level, Senior

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.

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

System Design Medium Mid-level, Senior

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 }
```

68. How do you design a rate limiter?

System Design Medium Mid-level, Senior

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.

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

System Design Hard Senior

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"
```

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

System Design Hard Senior

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).

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

System Design Hard Senior

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
-

72. Monolith vs microservices: which should you choose?

System Design Hard Senior

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.

73. How do you design a notification system?

System Design Hard Senior

Accept notification requests through an API, put them on a message queue, and let channel-specific workers (email, SMS, push, in-app) send them using user preferences, templates, retries with idempotency and rate limits.

A notification system sends messages to users across several channels when something happens in the product: an order ships, a password changes, a comment is posted. The hard parts are reliability, not losing or duplicating messages, and respecting user preferences at scale.

The core idea is to decouple producers from delivery. Product services publish an event; the notification service decides who should be told, through which channels, and hands each delivery to a queue so slow third-party providers never block the caller.

Key points

- Decouple producers from delivery with queues
 - Separate queues per channel and priority
 - Retries with backoff, dead-letter queue and idempotency keys
-

- Check user preferences before sending
-

74. How do you design a chat application like WhatsApp or Slack?

System Design Hard Senior

Clients keep a persistent WebSocket connection to stateless chat servers; messages are stored durably, routed to the recipient's server through a pub/sub layer, and delivered as push notifications when the recipient is offline.

A chat system needs low-latency, ordered delivery between users who may be online on several devices or offline entirely. The main design questions are how clients stay connected, how a message finds the server holding the recipient's connection, and how messages are stored and synced.

Start with one-to-one messages, then extend to group chats, read receipts and presence. State your assumptions on scale early, because they decide the storage choice.

Key points

- WebSockets for real-time delivery
 - Connection registry to route messages between servers
 - Per-conversation sequence numbers for ordering and sync
 - Push notifications for offline users
-

75. How do you design a scalable file storage system like Google Drive or Dropbox?

System Design Hard Senior

Store file contents as chunks in object storage and file metadata in a database; clients upload chunks directly with pre-signed URLs, a sync service tracks versions and changes, and a CDN serves downloads.

A file storage service must store large files reliably, sync changes across devices and let users share files, while keeping storage and bandwidth costs under control.

Separating metadata from content is the central decision: metadata (names, folders, versions, permissions) is small and needs transactions; content is large, immutable once written and fits object storage.

Key points

- Separate metadata (database) from content (object storage)
 - Chunk files and deduplicate by content hash
 - Upload directly to storage with pre-signed URLs
 - CDN for downloads, change events for sync
-

Software Testing

76. What is the difference between unit, integration and end-to-end tests?

Software Testing Easy Intern, Junior

Unit tests check one small piece of code in isolation, integration tests check that several pieces work together (for example code plus a real database), and end-to-end tests drive the whole application the way a user would.

Unit tests are fast and precise: when one fails, you know exactly which function broke. They replace slow or external dependencies with test doubles, so they cannot prove the pieces fit together.

Integration tests exercise real collaborations, such as a repository class against a test database or an API handler with its middleware. They catch wiring, configuration and query mistakes that unit tests miss, at the cost of more setup and slower runs.

End-to-end tests run the deployed application through its real interface, usually a browser with Playwright,

Cypress or Selenium. They give the most confidence that a user journey works, but they are the slowest, most expensive to maintain and most prone to flakiness.

In the interview

A unit test checks one function or class in isolation and runs in milliseconds. An integration test checks that real parts work together, like my API handler and a test database. An end-to-end test drives the whole app through the browser like a user. I rely mostly on unit and integration tests and keep a small set of end-to-end tests for the critical journeys.

Key points

- Unit: small, isolated, fast, precise failures
- Integration: real collaborators such as a database or HTTP layer
- End-to-end: full user journeys through the UI, slowest and most brittle

```
// unit: pure function, no dependencies
test('applies a 10% discount', () => {
  expect(applyDiscount(100, 0.1)).toBe(90);
});

// end-to-end: a real browser against the running app
test('user can check out', async ({ page }) => {
  await page.goto('/cart');
  await page.getByRole('button', { name: 'Checkout' }).click();
  await expect(page.getByText('Order confirmed')).toBeVisible();
});
```

77. What is the test pyramid?

Software Testing Easy Junior, Mid-level

The test pyramid is a guideline to have many fast unit tests at the base, fewer integration tests in the middle and only a small number of slow end-to-end tests at the top, so the suite stays fast and reliable.

The shape reflects cost and speed. Lower-level tests are cheap to write and run and give precise failures; higher-level tests give more realistic confidence but are slower and break for more reasons.

The anti-pattern is the "ice-cream cone": most coverage comes from manual testing and UI tests, with few unit tests. Such suites take hours, fail randomly, and teams stop trusting them.

The pyramid is a heuristic, not a law. Some teams prefer a "testing trophy" with more integration tests, especially for frontend code where testing components with their real children gives better confidence per test. The underlying principle is the same: push each check to the lowest level that can catch the bug.

Key points

- Many unit tests, fewer integration tests, few end-to-end tests
- Avoid the ice-cream cone of mostly UI and manual tests
- Test each behaviour at the lowest level that can catch the bug

78. What is mocking in testing and when should you use it?

Software Testing Medium Junior, Mid-level

Mocking replaces a real dependency with a controllable fake so a test can run in isolation and check how the code interacts with it; use it for slow, non-deterministic or external dependencies, not for everything.

Test doubles come in a few flavours. A stub returns canned answers, a mock also records calls so you can assert on them, a spy wraps a real function to observe it, and a fake is a working lightweight implementation such as an in-memory repository.

Mock at the boundaries of your system: payment providers, email services, the clock, random numbers, network calls. That keeps tests fast and deterministic and lets you simulate failures that are hard to trigger for real, such as a timeout.

Over-mocking is the common mistake. When a test mocks every collaborator and asserts on internal calls, it breaks on every refactor while still passing when the real integration is broken. Prefer asserting on outputs and observable behaviour, and cover real integrations with a smaller set of integration tests.

Key points

- Stub, mock, spy and fake are different kinds of test doubles
- Mock external, slow or non-deterministic dependencies
- Over-mocking couples tests to implementation details

```
test('sends a receipt after payment', async () => {
  const email = { send: jest.fn().mockResolvedValue(undefined) };
  const payments = { charge: jest.fn().mockResolvedValue({ id: 'pay_1' }) };

  await checkout({ email, payments }, { userId: 7, total: 50 });

  expect(payments.charge).toHaveBeenCalledWith(7, 50);
  expect(email.send).toHaveBeenCalledWith(7, expect.stringContaining('pay_1'));
});
```

79. How do you deal with flaky tests?

Software Testing Medium Mid-level, Senior

Treat a flaky test as a bug: quarantine it so it stops blocking the pipeline, reproduce it by running it repeatedly, fix the root cause (usually timing, shared state or external dependencies), then bring it back.

A flaky test passes and fails on the same code. Its real cost is trust: once developers learn to click "re-run", genuine failures get ignored too.

Common causes are fixed sleeps instead of waiting for a condition, tests that depend on execution order or shared data, real network calls, time zones and the current date, randomness without a fixed seed, and animations in UI tests.

The process: track flaky tests (many CI systems report them), quarantine them with an owner and a deadline, reproduce by running the test in a loop or under load, and fix the cause. Use auto-waiting assertions in Playwright or Cypress, isolate test data per test, control the clock, and mock external services. Retries can hide symptoms, so use them sparingly and still track the underlying failures.

Key points

- Flaky tests destroy trust in the suite
- Usual causes: timing, shared state, external services, dates, randomness
- Quarantine with an owner, reproduce in a loop, fix the root cause
- Avoid fixed sleeps; wait for conditions

```
// flaky: assumes the request finishes within 1 second
await page.click('text=Save');
await page.waitForTimeout(1000);
expect(await page.textContent('.toast')).toBe('Saved');

// stable: waits for the condition (auto-retrying assertion)
await page.getByRole('button', { name: 'Save' }).click();
await expect(page.getByRole('status')).toHaveText('Saved');
```

Behavioral

80. How do you answer "Tell me about yourself" in a software engineer interview?

Behavioral Easy Intern, Junior

Give a 60-90 second summary in three parts: what you do now, one or two achievements that show your strengths, and why this role is the logical next step.

This question is an invitation to set the agenda for the interview, not a request for your life story. Interviewers want a short, relevant summary that tells them where to dig deeper.

Use a present, past, future shape. Present: your current role or studies and the kind of work you do. Past: one or two concrete things you built or improved, with a result. Future: why this team and role is the next step for you.

Tailor it to the job description. For a backend role, lead with APIs and data work; for a frontend role, with user-facing features and performance. Practise it out loud until it sounds natural rather than memorised.

Key points

- Keep it to about 60-90 seconds
 - Present, past, future structure
 - Pick achievements relevant to this job
 - End with why you want this role
-

81. How do you answer "Why should we hire you?" as a developer?

Behavioral Easy Intern, Junior

Connect two or three of your strongest, provable skills directly to the needs in the job description, give a short piece of evidence for each, and show genuine interest in the team's work.

This is a closing pitch. The interviewer wants a confident, specific summary of why you fit this role, not a list of adjectives like "hard-working" and "passionate".

Before the interview, pick the two or three requirements in the job description that matter most and match each one with evidence from your experience: a project, a metric, a responsibility you held.

Finish with motivation that is specific to the company: its product, technical challenges or engineering culture. Keep the whole answer under a minute.

Key points

- Match your strengths to their stated needs
 - Back each claim with a concrete example
 - Show specific interest in the company
 - Keep it under a minute
-

82. What are you currently doing to grow as a software engineer?

Behavioral Easy Intern, Junior

Give specific evidence of deliberate learning through projects, feedback, reading, experimentation or mentoring rather than listing technologies.

Interviewers want to know whether your growth is intentional and connected to your current level and future responsibilities.

A strong answer combines a skill you are improving, how you practice it and evidence that your behavior has changed.

For senior candidates, include growth beyond coding: architecture, communication, mentoring, product understanding or leadership.

Key points

- Choose specific growth areas
 - Practice deliberately
 - Use feedback
 - Show evidence of improvement
-

83. Why do you want to work for our company?

Behavioral Easy Intern, Junior

Connect something specific about the company, product, engineering challenges or team to your own experience and career goals.

A good answer demonstrates that you understand what the company actually does and why the role matters to you.

Choose two or three specific reasons and connect each to evidence from your background or interests.

Avoid generic statements that could be copied into any interview.

Key points

- Research the company
 - Be specific
 - Connect to your experience
 - Show realistic motivation
-

84. How do you answer "Tell me about a difficult project you worked on"?

Behavioral Medium Junior, Mid-level

Choose a real project with a technical or organisational challenge, explain your specific role and decisions with the STAR method, and finish with the result and what you learned.

The interviewer is testing how you handle complexity and pressure, and whether you can explain technical work to someone who was not there. The difficulty itself matters less than how you reasoned through it.

Pick a project where you made meaningful decisions. Briefly set the context, then spend most of the answer on your actions: how you broke the problem down, what options you considered, why you chose one, and how you worked with others.

Close with an honest result. If something went wrong, say so and explain what you would do differently. Senior candidates should also show how they reduced risk for the team, for example by phasing a migration or adding monitoring.

Key points

- Use STAR: Situation, Task, Action, Result
 - Spend most of the time on your actions
 - Say "I" for your work and "we" for the team's
 - Include what you learned
-

85. How do you answer "Tell me about a production bug you fixed"?

Behavioral Medium Junior, Mid-level

Walk through how you noticed the issue, limited the impact, found the root cause, fixed it, and what you changed afterwards so the same class of bug could not happen again.

This question tests your debugging process and how you behave under pressure. Interviewers want a calm, methodical story, not a hero story.

A strong answer has a clear order: detection (alert, logs, user report), mitigation first (rollback, feature flag, scaling), investigation (reproducing it, narrowing the cause with logs, metrics or git bisect), the actual fix, and prevention (tests, monitoring, a post-mortem).

Be specific about the technical cause, at a level the interviewer can follow. A memorable detail, such as a timezone edge case or a missing index that only hurt at peak traffic, makes the story credible.

Key points

- Mitigate impact before hunting for the root cause
 - Explain how you narrowed the cause down
 - Finish with prevention: tests, alerts, process changes
 - Stay blameless
-

86. Tell me about a mistake you made and what you learned from it.

Behavioral Medium Junior, Mid-level

Choose a real mistake, take ownership, explain the impact, what you did to recover, and what you changed so it would not happen again.

Interviewers are testing accountability, self-awareness and whether you learn from mistakes rather than whether you have a perfect record.

Choose a mistake that is meaningful but safe to discuss. Spend more time on your response and learning than on the mistake itself.

A strong answer shows prevention: a test, checklist, monitoring, review practice, documentation or process improvement.

Key points

- Take ownership
 - Explain impact honestly
 - Show recovery
 - Explain prevention
-

87. Tell me about a time you received difficult or critical feedback.

Behavioral Medium Junior, Mid-level

Describe feedback that was uncomfortable but useful, explain how you responded, and show the measurable or observable improvement that followed.

The interviewer wants evidence that you can accept feedback without becoming defensive and turn it into better performance.

Choose feedback that was genuinely actionable. Explain what you initially thought, what you understood after reflection, and what you changed.

For senior candidates, show how the experience changed the way you give feedback to others as well.

Key points

- Listen without defensiveness
 - Ask clarifying questions
 - Act on feedback
 - Show improvement
-

88. Tell me about a project or decision that failed.

Behavioral Medium Junior, Mid-level

Choose a genuine failure, explain why it failed, what you owned, how you recovered, and what you changed in future projects.

Good answers demonstrate resilience and learning rather than trying to make the failure sound like a success.

Be precise about the original assumption, the signal that showed it was wrong, and the decision you made once you knew.

Senior candidates should explain how they changed team practices or reduced similar risk afterwards.

Key points

- Be honest
 - Identify the failed assumption
 - Recover quickly
 - Show systemic learning
-

89. Tell me about a time you had to deliver under a very tight deadline.

Behavioral Medium Junior, Mid-level

Show how you prioritized the essential outcome, communicated trade-offs, reduced risk and delivered without pretending everything could be done.

The interviewer is testing prioritization and communication under pressure, not willingness to work unlimited hours.

Explain what you cut, deferred or simplified and why. If quality or risk was temporarily accepted, explain how you contained it.

Strong answers include early communication when the original scope was unrealistic.

Key points

- Define the must-have outcome
 - Prioritize ruthlessly
 - Communicate trade-offs
 - Protect critical quality
-

90. Tell me about a time you disagreed with a code review comment.

Behavioral Medium Junior, Mid-level

Show that you can defend a technical decision with evidence while remaining open to a better argument.

Code review is collaborative engineering, not a contest of authority.

Explain the concern behind the comment before explaining your position. A strong answer may end with you changing your implementation.

If the disagreement remained, describe how you used project conventions, documentation, tests or a short experiment to resolve it.

Key points

- Understand the concern
 - Use evidence
 - Stay respectful
 - Accept better reasoning
-

91. Why are you looking to leave your current company?

Behavioral Easy Mid-level, Senior

Keep the answer professional and future-focused: explain what you are looking for next without attacking your current or former employer.

Interviewers are checking motivation, professionalism and whether your expectations match the role.

Focus on positive reasons such as greater ownership, a different technical challenge, product scale or opportunities to grow.

If there were genuine difficulties, describe them factually and briefly, then return to what you want to build next.

Key points

- Stay positive
 - Be truthful
 - Focus on the future
 - Connect to the role
-

92. How do you answer "Tell me about a time you disagreed with a teammate"?

Describe a real technical or process disagreement, show that you listened and argued with evidence rather than opinion, explain how a decision was reached, and how you supported it afterwards even if it was not your idea.

Interviewers want to know whether you can disagree productively. Teams need people who raise concerns, but also people who commit once a decision is made.

Choose a disagreement about something that mattered, such as an architecture choice, a deadline or code review standards. Avoid personal conflicts. Show that you first understood the other person's reasoning, then brought data: a benchmark, a prototype, an incident report.

The outcome does not need to be that you won. "We went with their approach, and I helped make it work" is a strong answer if you explain what you learned.

Key points

- Pick a professional, not personal, disagreement
 - Show you listened first
 - Use evidence: prototypes, data, constraints
 - Disagree and commit
-

93. Tell me about a time you took ownership of something that was not explicitly assigned to you.

Describe a problem you noticed, why you decided to act, how you coordinated with others, and the impact of taking ownership.

This question separates people who only complete assigned tickets from engineers who notice and solve important problems.

Do not describe work you simply volunteered for without explaining why it mattered to users, reliability, delivery or the team.

Show initiative without suggesting that you ignored ownership boundaries or other people.

Key points

- Identify an unowned problem
 - Explain why it mattered
 - Coordinate before changing shared systems
 - Show impact
-

94. Tell me about a time you had to work with unclear or changing requirements.

Explain how you identified ambiguity, clarified the important decisions, made reasonable assumptions and kept delivery moving.

Real engineering work often starts with incomplete information. Interviewers want to see structured thinking rather than waiting passively for perfect requirements.

Explain which questions you asked, which assumptions you documented and which decisions you intentionally kept reversible.

Strong candidates distinguish between uncertainty that must be resolved now and uncertainty that can be handled later.

Key points

- Clarify goals
 - Document assumptions
 - Prioritize unknowns
-

- Keep decisions reversible where possible
-

95. Tell me about a time you had several important tasks competing for your attention.

Behavioral Medium Mid-level, Senior

Explain how you assessed urgency, impact, dependencies and risk, then communicated the resulting priorities.

Strong engineers do not prioritize only by who asks first. They consider customer impact, deadlines, dependencies, severity and reversibility.

Show how you made the priority decision visible to stakeholders so expectations were aligned.

For senior candidates, explain how you helped the team prioritize rather than managing only your own queue.

Key points

- Assess impact and urgency
 - Consider dependencies
 - Communicate priorities
 - Revisit when conditions change
-

96. Tell me about a time you disagreed with your manager or technical lead.

Behavioral Medium Mid-level, Senior

Show respectful disagreement, evidence-based reasoning, willingness to listen and commitment after the final decision.

Disagreeing professionally is healthy. The key is to challenge the decision rather than the person.

Explain what evidence you brought, what you learned from their perspective and how the final decision was made.

If the decision went against your recommendation, explain how you supported it afterwards.

Key points

- Challenge ideas, not people
 - Use evidence
 - Listen to constraints
 - Disagree and commit
-

97. Tell me about a process you improved for your team.

Behavioral Medium Mid-level, Senior

Identify a recurring source of wasted time or risk, make a focused improvement, and demonstrate its impact.

This question looks for engineers who improve the system around the code, not just the code itself.

Choose a process with repeated cost: deployment, code review, testing, incident response, onboarding or local setup.

Quantify the improvement when possible, but a clear reduction in errors or friction is also useful.

Key points

- Identify recurring pain
 - Make a focused change
 - Get team buy-in
 - Measure impact
-

98. Tell me about a repetitive task you automated.

Behavioral Medium Mid-level, Senior

Explain the manual problem, why automation was worthwhile, how you implemented it safely and what time or error reduction resulted.

Automation is valuable when it reduces recurring human effort or prevents avoidable errors.

Explain why you chose the level of automation you did rather than building a complicated system for a small problem.

Mention maintenance, failure handling and how users or teammates adopted the automation.

Key points

- Identify recurring cost
 - Choose proportional automation
 - Handle failures
 - Measure savings
-

99. Tell me about a time you explained a complex technical concept to a non-technical person.

Behavioral Medium Mid-level, Senior

Show how you adapted your explanation to the audience, focused on the outcome and avoided unnecessary technical jargon.

Senior engineers often communicate with product, support, sales and leadership. The ability to make technical risk understandable is important.

Explain the audience, what they needed to decide and how you translated the technical issue into business impact.

Avoid presenting technical simplification as hiding important information; explain the right level of detail for the decision.

Key points

- Understand the audience
 - Explain impact first
 - Avoid unnecessary jargon
 - Confirm understanding
-

100. Tell me about a problem you prevented before it reached production.

Behavioral Medium Mid-level, Senior

Show how you noticed a risk early, validated it, communicated it and added a safeguard.

Preventative engineering demonstrates maturity because the best incident is often the one that never happens.

Explain the signal that made you suspicious and how you verified the risk instead of relying on intuition alone.

Finish with the lasting safeguard: test, validation, monitoring, review, rollout control or design change.

Key points

- Spot risk early
 - Validate before escalating
 - Communicate clearly
 - Add a lasting safeguard
-

101. Tell me about a time you had to work with a difficult stakeholder.

Describe how you understood the stakeholder's goals, handled disagreement professionally and reached a workable outcome.

The interviewer is evaluating communication and emotional maturity, not whether you can complain about difficult people.

Describe the stakeholder's legitimate concern before explaining the conflict. This makes the story balanced and credible.

Focus on shared goals, explicit trade-offs and communication rather than personality differences.

Key points

- Understand their goals
 - Stay professional
 - Use facts and options
 - Find shared outcomes
-

102. Tell me about a technical debt decision you made.

Explain why you accepted, reduced or paid down technical debt and how you managed the associated risk.

Technical debt is not automatically bad. Interviewers want to see whether you understand its cost and deliberately manage it.

Describe the business constraint, the engineering risk and why the chosen approach was appropriate at that time.

Strong answers include a trigger or plan for addressing the debt instead of leaving it invisible.

Key points

- Identify the debt
 - Understand its cost
 - Make the trade-off explicit
 - Create a follow-up trigger
-

103. Tell me about a time you introduced a new technology or tool to a project.

Start with the problem, evaluate alternatives, validate the technology with a small experiment and explain how you managed adoption and risk.

Interviewers want to know that you choose technology because it solves a problem, not because it is fashionable.

Discuss the existing approach, why it was insufficient, alternatives considered, proof of concept and operational implications.

Senior candidates should include how they helped the team adopt and maintain the new technology.

Key points

- Start with the problem
 - Evaluate alternatives
 - Prototype before committing
 - Plan adoption and maintenance
-

104. Tell me about a time you mentored or helped another developer grow.

Explain how you identified what the developer needed, coached rather than simply solved the problem, and measured their progress.

Mentoring is about increasing another engineer's ability to solve problems independently.

Strong answers describe questions, pairing, feedback, documentation or progressively larger ownership rather than simply giving answers.

Senior candidates should show patience and adaptation to the individual developer's level.

Key points

- Understand their starting point
 - Teach reasoning, not only solutions
 - Give actionable feedback
 - Increase independence
-

105. Tell me about a time you influenced a technical decision without having formal authority.

Behavioral Hard Senior

Show how you built trust, understood competing concerns, used evidence and helped the group reach a decision.

Senior engineers often influence architecture and engineering practices without being the manager or final decision-maker.

The strongest stories show persuasion through clarity and evidence rather than relying on seniority or force of personality.

Explain what happened after the decision and whether you supported the team's final choice.

Key points

- Build alignment
 - Use evidence
 - Understand other constraints
 - Support the final decision
-

106. Tell me about a time you had to choose between two good technical approaches.

Behavioral Hard Senior

Explain the constraints, alternatives, trade-offs, decision criteria and why the chosen option was appropriate for the context.

Senior engineering decisions rarely have a universally correct answer. Interviewers want to see explicit trade-off reasoning.

Compare options using factors such as reliability, complexity, performance, cost, team expertise, delivery time and future change.

A strong answer acknowledges what you gave up by choosing the preferred option.

Key points

- Define decision criteria
 - Compare alternatives
 - Make trade-offs explicit
 - Choose for the context
-

107. Tell me about a major production incident you helped handle.

Behavioral Hard Senior

Cover detection, communication, mitigation, investigation, recovery and the changes made afterwards to reduce recurrence.

Incident questions test calm decision-making and ownership under pressure.

Separate immediate mitigation from root-cause analysis. During an incident, restoring service is usually more important than finding the perfect explanation immediately.

For senior candidates, include how you coordinated people, communicated status and improved the system after the incident.

Key points

- Protect users first
 - Communicate clearly
 - Investigate systematically
 - Prevent recurrence
-

108. Tell me about a time you decided not to use a technology or framework.

Behavioral Hard Senior

Explain why the technology looked attractive, what problem it actually solved, and why its cost or complexity was not justified in your context.

Knowing when not to add technology is an important engineering skill.

A strong answer compares the expected benefit against learning cost, operational complexity, team capability, migration cost and long-term maintenance.

Avoid making the decision sound ideological; show that you would reconsider if the constraints changed.

Key points

- Define the actual problem
 - Compare total cost
 - Avoid unnecessary complexity
 - State when the decision could change
-

109. Tell me about a time you had to balance speed of delivery with technical quality.

Behavioral Hard Senior

Explain the deadline, the quality risks, what you protected, what you simplified and how you managed any remaining debt.

Engineering teams constantly balance delivery and quality. Strong answers show deliberate trade-offs rather than treating one as always correct.

Separate reversible shortcuts from dangerous shortcuts. Never present security, data integrity or critical reliability risks as acceptable just to ship faster.

Explain what you documented or scheduled for follow-up.

Key points

- Identify non-negotiable quality
 - Make trade-offs explicit
 - Contain risk
 - Track follow-up work
-

110. Tell me about a technical decision you made that turned out to be wrong.

Behavioral Hard Senior

Explain the original reasoning, the evidence that changed your view, how you corrected course and what

decision-making practice improved afterwards.

This tests intellectual honesty and the ability to update your beliefs when reality disagrees with your assumptions.

Do not rewrite the story as if the decision was obviously correct. Explain what information was available at the time and what you missed.

The strongest answers show a change in process, such as better experiments, metrics or review criteria.

Key points

- Own the decision
 - Explain original assumptions
 - Recognize evidence
 - Improve the decision process
-

111. Tell me about a project you led from idea to production.

Behavioral Hard Senior

Walk through how you shaped the problem, planned the work, aligned stakeholders, managed risks and measured the outcome.

For senior engineers, leadership means creating clarity and reducing risk, not simply assigning tasks.

Cover discovery, design, execution, testing, rollout, monitoring and follow-up. Make your individual leadership contribution clear.

Include a moment where you made a meaningful trade-off or changed the plan based on new information.

Key points

- Define the problem
 - Align stakeholders
 - Plan and de-risk
 - Lead delivery
 - Measure outcome
-

112. Tell me about a time you worked with a teammate who was struggling.

Behavioral Hard Senior

Show empathy, clear expectations, practical support and appropriate escalation rather than blame.

The interviewer wants to see whether you can help a teammate without becoming their manager or doing their work for them.

Start by understanding the cause: unclear requirements, missing knowledge, workload, personal circumstances or process problems.

For senior candidates, explain how you balanced support with accountability and involved the appropriate lead when necessary.

Key points

- Understand the cause
 - Set clear expectations
 - Offer targeted support
 - Maintain accountability
-

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/software-engineer>