Hired_ | educative

ebook

The Complete Software Engineer Technical Interview Guide:

Front-end, Back-end & Full Stack Engineer Edition

A technical interview is an opportunity as much as it is a challenge. Whether you're early in your career or a seasoned engineer, it may be stressful if you feel unprepared. While the most important piece of these interviews is, of course, your technical skills, we've provided some strategies to put your best foot forward.

After all, going in with confidence and preparation is the best way to ease those nerves and let your skills shine through. We're not saying the interview will be easy, but we're sure we can help make it *easier*.

So, what is it you're getting yourself into? Technical interviews put a (fun?) spin on the typical job search process.

In many ways, they let you, as an engineer, do what you do best! They involve assignments and problems to solve, often mimicking what you might encounter in the role itself. Take them as your opportunity to "walk the walk" instead of just "talk the talk" (as you would in a **behavioral interview**). You get to show interviewers what you bring to the table.

In this guide we've collaborated with our partner Educative to bring you a full plan to level up your technical interview game. We'll cover:

1. **Chapter 1: How to prepare for technical interviews**
   Interviewing in the tech industry is a long process. You'll need a plan and a generous amount of time to prepare for a technical interview. Our example interview prep plan in this chapter will really come in handy!
2. **Chapter 2: What employers look for in technical interviews**
   We review some of the major concepts and skills interviewers assess for from these three roles: front-end developers, back-end developers, and full stack developers.
3. **Chapter 3: Common technical interview mistakes to avoid**
   After spending time reviewing what you should do, we warn you on what to avoid. Find the top three technical interview no-nos in this chapter.
4. **Chapter 4: Helpful resources**
   By this time, you're well on your way to nailing your next technical interview. Use our compilation of links to more resources to continue studying with a narrower focus.

Let's get started!

01 How to prepare for technical interviews

Interviewing in the tech industry is a long process. You'll need a plan and a generous amount of time to prepare for a technical interview. We recommend at least three months. That may seem long, but when you've put everything into practice, and your interviews start, you'll be happy you did it right.

Technical interviews take many forms:

- **Onsite interviews:** A series of interviews at a company's office.
- **Take-home assessments:** Assignments delivered virtually and completed with a time limit.
- **Live virtual coding challenges:** Your interviewer monitors you while you answer several technical questions virtually.
- **Whiteboard challenges:** A design-centric interview requiring you to draw on a whiteboard while talking through your design and thought process.
- **Pair programming:** Two programmers, one driver, and one navigator, take turns coding at the same workstation. The driver writes the code, while the navigator reviews it in real time. Typically, the interview candidate takes the role of the driver.

Some forms are harder than others, and your interviewers will likely exert some pressure, but not without reason. Most interviewers want you to succeed, but they must ensure you're the right person for the job.

Making your plan

Choose a programming language

Before you start prepping for a technical interview, you'll need to pick the language you want to use. Most interviewers are flexible, as long as you stick with your choice throughout the interview. It's not a good practice to switch languages halfway through.

Usually, your choice will directly relate to your role.
For example:

- A **front-end developer** would probably use JavaScript.
- A **back-end developer** would use Python, Ruby, Java, PHP, etc.
- For **full stack devs**, it depends on your typical **tech stack**.
  Most stacks include JavaScript and a few other languages.

What and how to study

No matter what kind of developer you are (or are applying to be) it is crucial that you create a **strategic interview prep plan**. There is a lot of material you need to cover so **efficiency is a priority**.

It can be hard knowing what topics you need to cover, and how best to retain all the information. Always start with the basics of your favorite programming language, and then work up to **coding interview patterns**.

Here's an example of an ideal interview prep timeline covering what and how to study.

Example plan

Depending on your experience, target role, and seniority, your plan may look different. This plan aims to be a reference point for most software engineering disciplines.

Week 1

Brush up on the basics of your chosen language.

Many technical interviews start with easy questions to raise the candidate's confidence. Don't let something simple at the beginning trip you up down the line!

Topics to cover:

- Splitting strings
- Parsing CSV or text files
- Declaring and using 2D arrays
- Reading and writing to and from files
- Processing command line arguments

Weeks 2 and 3

Review data structures and algorithms.

The more difficult questions in your interview will likely stem from these topics. Data structures and algorithms are the core of most high-level computer science concepts. It's entirely possible that you haven't thought about them since school, but they are useful in coding interviews.

Pay particular attention to **Big O notation** and other practices for complexity analysis.

Weeks 4 and 5

Practice with data structures and algorithms.

As you're reviewing the basics of data structures and algorithms, start practicing simple problems with the resources listed below. Reviewing the basics will help you internalize these concepts and tackle more difficult problems later.

Topics to cover:

- Arrays
  - Remove even integers from an array
  - Merge two sorted arrays
  - Find the first non-repeating integer in any array
- Linked Lists
  - Find the length of a linked list
  - Search in a singly linked list
  - Reverse a linked list
  - Find the middle value of a linked list

- Stacks/Queues
  - Sort values in a stack
  - Implement two stacks using one array
- Trees
  - Find the minimum value in a binary search tree
  - Find the height of a binary tree
  - Find k^{th} max value in a binary search tree
- Graphs
  - Implement breadth-first search
  - Implement depth-first search
- Tries
  - Find the total number of words in a trie
- Heaps
  - Find *k* smallest elements in a list
  - Find *k* largest elements in an array

With the following hands-on, interactive courses, you can review and practice solving challenges with common data structures and the most important algorithms in a language of your choice.

- **Data Structures for Coding Interviews in Python**
- **Algorithms for Coding Interviews in Python**
- **Data Structures for Coding Interviews in Java**
- **Algorithms for Coding Interviews in Java**
- **Data Structures for Coding Interviews in C++**
- **Algorithms for Coding Interviews in C++**

Weeks 6, 7, and 8

Coding interview practice.

By this point, you should be breezing through easier practice problems. Next, consider the problems interviewers are actually likely to ask.

Best practices:

- **Time yourself.** Try to solve your problem in 20 to 30 minutes, but don't be discouraged if some questions take longer at first.
- **Think about the runtime and memory complexities of your solutions.** Your interviewers will likely want you to articulate these complexities and how to optimize them.
- Work on problems using **coding interview patterns**. Almost all questions for a coding interview are built on patterns that serve as a blueprint for solving related problems.

Weeks 9 and 10

System Design Interviews.

System Design Interviews (SDIs) are increasingly common in many software engineering interviews. These interviews are important because they help determine your engineering level. Many interviewers use the SDI to filter candidates to positions of appropriate seniority.

Most SDIs will ask you to design a large-scale web service. Typically, these systems mimic popular real-world applications so applicants can easily understand the functional and nonfunctional requirements.

SDIs are so important that we'll return to them after this example plan.

Week 11

Concurrency and multithreading.

Interviewees aiming for a senior or staff engineering role especially should consider these concepts.

Example questions:

- What is a BlockingQueue?
- What are some differences between a process and a thread?
- How can you interrupt a running thread in Java?

Week 12

Object-oriented design.

Acknowledging how systems designed with an object-oriented approach work will help make you a better programmer and candidate. It's worth learning the fundamentals of object-oriented design and **applying them to solving real-world design problems**.

Example designs to consider:

- Design an ATM
- Design an elevator
- Design a parking system

Not vying for a senior position?

If you don't anticipate questions about multithreading, concurrency, or object-oriented design, consider preparing for the **behavioral interview**.

Although potentially less overwhelming than a technical interview, the behavioral interview is still critically important in getting that offer letter. Practicing **mock behavioral questions** requires less technical attention than relearning computer science basics. You can also learn **patterns for behavioral questions** and strategies for answering each type. Your preparation will pay off when you sit for the real thing.

The behavioral interview will vary based on the company. It's good to study the company's values and the principles they look for in an employee.

System Design Interviews

SDIs can determine your level, but they are not just for senior developers.

Entry-level engineers: At this level, engineers have a narrow focus on a few different software components and how they interact with each other.

Senior engineers: They have a more holistic view of the software system they're working on, and can describe various scenarios from end-to-end. They can explain how each scenario is executed, give concrete examples, and offer ways to improve the resiliency of a system.

Staff engineers: Engineers at this level are capable of everything mentioned above, but they also monitor the software system over the course of its entire lifetime. By considering the architecture's ability to sustain and support growth, they plan how a system evolves and scales.

What is an interviewer looking for?

Interviewers are looking for **hireable signals**. This may be especially true in an SDI. Unlike in other interviews, the interviewer will expect you to lead the System Design conversation.

Important hireable signals to send:

- **Design ability:** How well can you break a large system down into necessary components?
- **Thought process:** How do you solve problems to satisfy functional and non-functional requirements while bound by real-world constraints?
- **Communication and leadership:** How well can you communicate your ideas and designs?
- **User experience:** Do you understand how certain design choices affect the user? Do you iterate on your design with the intent of improving the UX?

What does a System Design Interview look like?

As mentioned earlier, you will be driving the SDI conversation. Knowing what to expect is imperative, as is going in with a strategy. Below is an 8-step framework for breaking down the interview and solving any SDI question.

educative

Solve Any

System Design Interview Question

The 8-part **RESHADED** method:

1. Requirements
2. Estimation
3. Storage schema (optional)
4. High-level design
5. APIs
6. Detailed design
7. Evaluation
8. Distinctive component/feature

Step 1: Requirements
Gather functional & non-functional requirements

Consider:
- *System goals*
- *Key features*
- *System constraints*
- *User expectations*

Step 2: Estimation
Estimate hardware & infrastructure needed to implement at scale

Consider requirements for:
- *Number of servers*
- *Daily storage*
- *Network*

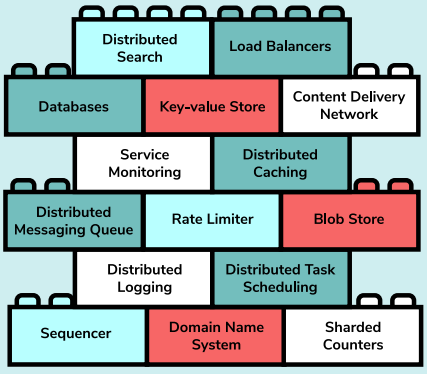
Step 3: Storage schema (optional)*
Articulate data model

Define:
- *Structure of data*
- *Tables to use*
- *Type of fields in tables*
- *Relationship between tables (optional)*

***Relevant when you:**
- *Expect highly normalized data*
- *Will store different parts of data in various formats*
- *Face performance & efficiency concerns around storage*

Step 4: High-level design
- Build high-level design
- Choose building blocks to meet functional requirements

For each, identify:
- ***How** they work*
- ***Why** they're needed*
- ***How** they integrate*



*This layered visual shows dependencies between building blocks. **Blocks in lower layers support those above.***

Step 5: APIs
Translate functional requirements into API calls

E.g.:
- ***Requirement:** Users should be able to access all items*
- ***API call:** GET / items*

Step 6: Detailed design
- *Improve high-level design*
- *Consider all non-functional requirements & complete design*

Step 7: Evaluation
- *Evaluate design against requirements*
- *Explain trade offs & pros/cons of different solutions*
- *Address overlooked design problems*

(8*) Distinctive component/feature
Discuss a distinctive feature that meets requirements
- *E.g. Concurrency control in high-traffic apps*

** Timing varies. Best done after completing design (E.g. Step 6 & 7)*

Building Blocks Glossary:

Domain Name System: *Maps domain names to IP addresses.*

Load Balancers: *Distributes client requests among servers.*

Databases: *Stores, retrieves, modifies, & deletes data.*

Key-Value Store: *Stores data as key-value pairs.*

Content Delivery Network: *Distributes in-demand content to end users.*

Sequencer: *Generates unique IDs for events & database entries.*

Service Monitoring: *Analyzes system for failures & sends alerts.*

Distributed Caching: *Stores frequently accessed data.*

Distributed Messaging Queue: *Decouples messaging producers from consumers.*

Publish-Subscribe System: *Supports asynchronous service-to-service communication.*

Rate Limiter: *Throttles incoming requests for services.*

Blob Store: *Stores unstructured data.*

Distributed Search: *Returns relevant content for user queries.*

Distributed Logging: *Enables services to log events.*

Distributed Task Scheduling: *Allocates resources to tasks.*

Sharded Counters: *Counts concurrent read/write requests.*

02 What employers look for in technical interviews

While we won't cover every possible **topic or interview question** you'll encounter, we will review some of the major concepts and skills that interviewers will want to assess for three roles. These should give you a general idea of what subjects to explore further.

Front-end developers

These developers must be fluent in HTML5, CSS3, and JavaScript. Beyond that, they should expect questions about modern web development frameworks, tooling, and libraries, and how to apply them to different projects.

In addition, front-end developers should have a strong grasp of the best practices for implementing responsive design, accessibility, and cross-device compatibility.

Languages: HTML5, CSS3, JavaScript

Frameworks: React, Bootstrap, AngularJS, Vue.js, jQuery

Tools: Version control using Git and GitHub

Skills: Front-end architecture, responsive design, cross-device compatibility, web optimization

Be prepared to answer questions like:

- What are the most important principles of good front-end design?
- How do you optimize web pages for different devices?
- What are some of the most common front-end development frameworks, and how would you choose one?
- What are some problems you've encountered while developing web pages, and how did you solve them?

Back-end developers

These developers are largely responsible for the functionality of a web application. They are usually interviewed on their ability to set up servers and connect websites to databases using technologies like MySQL or MongoDB.

These developers need to create robust, secure back-ends that can meet the requirements of their clients, so they should be aware of the tradeoffs of using different back-end technologies for performance, scalability, or ease of use.

Back-end developers are also interviewed on APIs, webhooks, and other common tools. APIs are particularly important these days, as they allow websites to connect with various third-party services to provide additional functionality without writing any extra code.

Languages: PHP, Java, Python, Ruby, etc. Understanding HTML, CSS, and JavaScript is a plus, but not necessary.

Frameworks: Django, Laravel, Spring, Ruby on Rails, ASP.NET Core, ExpressJS, etc.

Databases: MySQL, MongoDB, PostgreSQL, etc.

Server handling: Cloud hosting, DNS hosting, Docker, Apache, Nginx, etc.

APIs: SOAP, REST, JSON, etc.

Tools: Version control using Git and GitHub

Computer science: Data structures like trees, queues, and stacks. Algorithms. Object-oriented programming (OOP) principles.

Be prepared to answer questions like:

- When would you use a relational database instead of a NoSQL database?
- What are some of the most important considerations when choosing a back-end development framework?
- What are some of your favorite APIs to work with, and why?
- What is your preferred method for deploying web applications?

Full stack developers

These developers can design, build, test, and deploy web applications from start to finish. As such, developers capable of working on both the client side and server side will need to be familiar with all of the topics mentioned above.

Full stack developers typically choose a technology stack that includes both a front-end framework and a back-end framework like MEAN (Mongo, Express, AngularJS, and Node.js) or Ruby on Rails.

Front-end frameworks: ReactJS, AngularJS, Bootstrap, etc.
Back-end frameworks: Django, Rails, ASP.NET, Spring Boot, etc.

Interview questions for full stack developers include the front-end and back-end in their scope but may focus more on design patterns, best practices, or stack-specific processes.

Here are some examples of questions meant for a candidate who uses the LAMP (Linux, Apache, MySQL, and PHP) stack:

- How do you debug a PHP application?
- What are some tips for optimizing a LAMP deployment?
- What are some challenges you have encountered using Apache and MySQL?
- How would you design the front-end architecture for a LAMP application?

03 Common technical interview mistakes to avoid

Mistake 1: Coming unprepared

One of the biggest mistakes you can make before an interview is not preparing at all. Interviewers want to see candidates who express interest in what their company does.

- Familiarize yourself with the company's culture, mission, and values.
- Test out or use the company's product or service to see what you like or dislike about it.
- **Learn about their interview process**, as not all companies are alike!

Mistake 2: Not giving specific examples

After you've learned about the company's culture and values, practice applying the **STAR** (Situation, Task, Action, Result) method to each bullet point of your resume. While it's unnecessary to produce examples of every bit of code you've written in the past, you should still prepare to connect your answers to specific projects or challenges you've faced. This helps you illustrate your approach to problem-solving without wasting time.

- **Situation:** Describe a specific situation and provide enough context for the interviewer to understand the circumstances.
- **Task:** Describe your responsibilities. What were you asked or expected to do, and why?
- **Action:** Describe the actions you took, and how your contributions were impactful.
- **Result:** Describe the outcome. Go over what happened, what was accomplished, and what you learned from the experience. Be prepared to answer follow-up questions.

Mistake 3: Not asking questions

Interviewers assess your coding and problem-solving skills as well as your ability to work with others. Asking the right questions when you need clarification can show you are both resourceful and communicative when there are obstacles. Not asking questions can raise the risk of making faulty assumptions.

Asking questions is especially important during System Design interviews where you must determine functional and non-functional requirements.

Examples of questions to consider while designing a website or application:

- What are the constraints of this system?
- How much traffic should this system be able to handle?
- What are the availability and latency requirements?

04 Helpful resources

Now that you've read this ebook, you're well on your way to nailing your next technical interview. Compiled below are links to more resources that will help you continue studying with a narrower focus.

Coding Interview Blogs

3 month coding interview preparation bootcamp
Why a strategic coding interview prep plan matters
The coding interview FAQ: preparation, evaluation, and structure
The insider's guide to algorithm interview questions

System Design Interview Blogs

The complete guide to the System Design Interview in 2023
Simplify system design interviews with the RESHADED approach
The top 6 system design interview mistakes to avoid

If you're ready to take your interview prep to the next level, here are some of our **favorite courses** to help you become an interview pro.

Learning Paths

Deep Dive into the System Design Interview
Ace the Front End Interview

Individual Courses

Web Development Interview Handbook
Grokking the Behavioral Interview
Grokking Coding Interview Patterns in Python
Grokking Coding Interview Patterns in JavaScript

After the interview

While you deserve a big pat on the back post-interview, we understand it can still be a stressful time. It's tricky to balance letting go of possible mistakes while also learning to improve. Don't berate yourself over what you didn't know.

If there's a critical gap between what you know and what the job requires, perhaps it wasn't the right fit. If you are concerned about silly mistakes, remember that interviewers are human – and they know you are too.

Take every technical interview as a learning experience. If you encountered a problem that stumped you, work it out on your own time to figure out why it was difficult. Collaborate with others informally or online to learn from their perspectives and experience.

We also encourage you to ask for feedback on your technical evaluation after a potential employer makes a hiring decision. This is another chance to understand strengths and opportunities for improvement (It also highlights your eagerness to learn and ability to absorb feedback.).

Don't forget to send your interviewer a personalized follow-up email to thank them for the opportunity. While a thank you note is not the tipping point to extending an offer (it's the technical skills that really matter here!), it will never hurt your chances. Your interviewer may or may not respond, but rest assured you've left a positive impression.

Feeling more ready for your next technical interview? You should be – you're ahead of the game and a few steps closer to landing that job! Keep practicing and taking advantage of all the resources you have at your hands.

If you're ready to join Hired and have employers search for you instead, sign up now!

Looking for more hands-on learning to get prepared? Take a course with Educative!

Hired_ | educative