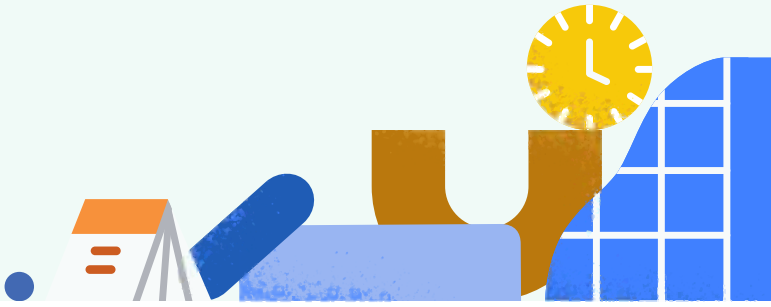
facebook

SOFTWARE ENGINEER

Onsite Interview Guide



What You'll Find in This Guide

If you've completed a Facebook initial technical screen, you've gotten a great start on preparing for your onsite interview. Much of Part 1 of this document will look familiar to you, since preparing for an onsite interview at Facebook involves many of the same steps as preparing for an initial tech screen.

If you haven't completed a Facebook initial technical screen, don't worry! This guide contains everything you'll need to plan, practice, and prepare for your onsite interview at Facebook.

Part 1: Coding

Just like preparing for a tech screen, the most important thing you can do to prepare for your onsite interview is to practice coding. Even the most experienced engineers need to prepare and practice to do well in an interview. For example, if you haven't practiced solving new problems under time constraints, interviewers may think you're unqualified when you're simply unprepared.

The coding portions of the onsite interview will be similar in structure to your initial tech screen, with a few differences:

1. **Format.** If you have an onsite interview at Facebook, you'll likely be coding on a whiteboard. Be sure to practice whiteboarding your solutions so you're comfortable working that way.
2. **Number.** Most interviewees complete two separate coding interview sessions during their onsite interviews instead of one like the initial tech screen.
3. **Difficulty.** Onsite coding interviews frequently feature more challenging problems, and interviewers may ask more or more in-depth follow-up questions to assess your solution and thought process.

Have questions about the format or details of your specific interview? Your recruiter can help!

What We Look For

Your interviewer will be thinking about how your skills and experience might help their teams. Help them understand the value you could bring by focusing on these traits and abilities:

- **Communication.** Are you asking for requirements and clarity when necessary, or are you just diving into the code? Your initial tech screen should be a conversation, so don't forget to ask questions.

- **Problem solving.** We're evaluating how you comprehend and explain complex ideas. Are you providing the reasoning behind a particular solution? Developing and comparing multiple solutions? Using appropriate data structures? Speaking about space and time complexity? Optimizing your solution?
- **Coding.** Can you convert solutions to executable code? Is the code organized and does it capture the right logical structure?
- **Verification.** Are you considering a reasonable number of test cases or coming up with a good argument for why your code is correct? If your solution has bugs, are you able to walk through your own logic to find them and explain what the code is doing?

How to Prepare

Interviewers can only assess your skills and abilities based on what you show them during your interview, so it's important to plan and prepare to best showcase your strengths.

1. Before you practice, plan!

Be honest with yourself—only you know how much prep time you'll need. Make the most of your prep time by following these steps to plan your approach before you start practicing.

Schedule time to study and practice. Block out time every day to write code. Target medium and hard problems.

Prioritize breadth over depth. It's much better to practice solving fewer example problems of many problem types than to become very familiar with one type at the expense of the others.

Set aside time to review what you've practiced. As you solve problems, make cheat sheets or flash cards to review later. Revision and repetition will strengthen your understanding of core concepts.

2. Review core areas

Everyone could use a refresher in at least one area.

Algorithm design/analysis. Consider different algorithms and algorithmic techniques, such as sorting, divide-and-conquer, recursion, etc.

Data structures. Make sure you're using the right tool for the right job. Keep in mind how and when certain data structures should be used and leveraged, and why some are preferred over others.

Other CS fundamentals. Review foundational techniques, including recursion, graph theory, tree traversal, combinatorial problems, and so on.

3. Focus on key practice strategies

Reading through sample questions, recognizing concepts, and having a vague understanding of these concepts won't be enough to help you shine. You need to practice! Make sure you're setting your practice sessions up for success by following these tips from engineers who've been through the process.

Practice coding the way you'll code during your interview. You'll most likely be coding at a whiteboard, but check with your recruiter.

Code in your strongest language. Avoid trying to learn a new language in a few weeks.

Practice talking through the problem space and possible solutions before you dive into code, and practice talking through your decisions out loud as you code. This can be unnatural in an interview setting, especially if you're used to coding alone. But your interviewers will be evaluating your thought process as well as your coding abilities, so explaining your decisions as you code is crucial to helping them understand your choices.

4. Understand the types of problems you may encounter

Practice a variety of different problems—and understand why we ask them—so you're prepared to solve them during your onsite interview.

Don't be surprised if the questions sound contrived. Problems may be different than what you're probably tackling in a day-to-day job. We won't ask a "puzzle" question, but questions may be different than real-world questions because they need to be described and solved in 10-20 minutes.

Problems may assess the depth of your knowledge and your versatility. For example, your interviewer might ask you to solve a problem any way you want. Then, they could add constraints on the running or space characteristics and ask you to solve it again.

Problems may focus on edge cases. You might be asked to parse some data format or mini language. Your answers demonstrate your ability to handle multiple states in your head.

Problems may test how well you know how things work under the hood. For example, you might be asked to implement well-known library functions.

5. Decide what resources you'll use to prepare

It's easy to be overwhelmed by the number of online resources or the detail in an entire theoretical algorithms book. For example, CLRS has excellent info—but it may not be the right resource for your study needs and timeline. Pick two or three resources and focus on those.

Resources from Facebook:

- Facebook Initial Tech Screen Study Guide: Sample Problems & Solutions
- Cracking the Facebook Coding Interview Videos (The Approach and Problem Walk-through. The password is FB_IPS.)
- InterviewBit

Other Websites:

- LeetCode
- HackerRank

Books:

- Cracking the Coding Interview
- Elements of Programming Interviews

Want to know even more? Check out these news articles for firsthand experiences of the technical interview process:

- I Interviewed at Five Top Companies in Silicon Valley in Five Days, and Luckily Got Five Job Offers

How to Approach Problems During Your Interview

Your interview is a conversation, not a test! Your interviewer wants to find out what it would feel like to work with you and solve a problem together, so approach problems like you're working on a team together and collaborating to build the solution.

1. Analyze the problem

Before you jump into building a solution, be sure you've taken the time to fully understand and talk through the problem.

Ask clarifying questions. Talk through the problem and ask follow-up questions to make sure you understand the exact problem you're trying to solve.

Let us know if you've seen the problem previously. That will help us understand your context.

Talk through the problem and your approach, and frame both like you're working with your interviewer. Try using words like "we" and "our" instead of "I" and "my" when you walk through the problem and solution to make the interview feel more collaborative.

2. Code up your solution

Get your thoughts down in code and explain your decisions and actions as you go.

Don't forget to talk while you code! It's crucial to help the interviewer understand your choices and thought process.

Iterate. Don't always try to jump immediately to the most clever solution. Talk through multiple possible solutions, then decide which will work best and explain your decision.

Leave yourself plenty of room. You may need to add code or notes between lines later.

Consider different algorithms and algorithmic techniques, such as sorting, divide-and-conquer, recursion, etc.

Think about data structures, particularly those used most often (array, stack/queue, hashset/hashmap/hashtable/dictionary, tree/binary tree, heap, graph, etc.)

Be prepared to talk about O memory constraints on the complexity of the algorithm you're writing and its running time as expressed by big-O notation.

Avoid solutions with lots of edge cases or huge if/else if/else blocks, in most cases. Deciding between iteration and recursion can be an important step.

Use descriptive variable names. This will take time, but it will prevent you from losing track of what your code is doing.

Simplify. If you can't explain your concept clearly in five minutes, it's probably too complex.

Draw out a sample input. Draw a picture or write out the variable values to see how the variables change as your code executes. You'll see insights and bugs faster and be less likely to lose track when you're thinking through edge cases in your head!

3. Stuck? Try these approaches!

Everyone, even engineers with years of experience, will get stuck at some point. Relax—it doesn't mean you've failed! The interviewer will be assessing your ability to methodically approach the problem from several angles and ask the right questions to get unstuck.

Draw pictures. It's easy to lose track when you're thinking through an edge case or how variables change as your code executes, so use the board! Writing everything out helps you see insights and bugs faster and make fewer mistakes. Draw several different test inputs. Capture how you would get the desired output by hand, then talk through your approach and translate it into code.

Ask questions. For example, if you don't know the exact syntax, you can ask the interviewer for help.

Think out loud as a way to slow yourself down. Use phrases like "Let's try doing it this way" or "We need to consider all the possibilities" to give yourself time to think while still making progress. Talk through what you know.

Talk through what you first thought might work and explain why it won't. You might think of a way to modify your original solution so it works or another question to ask to help you decide on your next steps.

Talk through the bounds on space and runtime. For example, say:

"We have to at least look at all of the items, so we can't do better than X."

"The brute force approach is to test all possibilities, which is X."

"The answer will contain X items, so we must at least spend that amount of time."

Call a helper function and keep moving. If you can't immediately think of how to implement some part of your algorithm, big or small, just skip over it. Write a call to a reasonably named helper function, say what it will do, and keep going. If the helper function is trivial, save it for the end. You can even ask your interviewer if they'd like to help implement it.

Solve a simpler version of the problem. Not sure how to find the 4th-largest item in the set? Think about how to find the largest item and see if you can adapt that approach.

Propose a solution that's naive and potentially inefficient. Consider optimizations later. Use brute force. Do whatever it takes to get some kind of answer.

Remember that it's ok if you don't know! No one who works at Facebook is perfect, and we don't look for perfection in people we interview. If you aren't sure if something is true or if it's the best solution, then say that. Explain what you do know, and your interviewer will ask you follow-up questions.

4. Verify your solution

You don't get more credit for doing this in your head! Testing your solution on the board, out loud will help you find bugs and clear up any confusion your interviewer may have.

Walk through your solution by hand, out loud, with an example input. Write down what values the variables hold as the program is running.

Look for off-by-one errors. Should your for loop use a "<" instead of a "<="?

Test edge cases. These might include empty sets, single item sets, or negative numbers. Don't forget about software verification—formal or otherwise!

Ask yourself if you would approve your solution as part of your codebase. Explain your answer to your interviewer. Make sure your solution is correct and efficient, and that it clearly reflects the ideas you're trying to express.

Part 2: Systems & Product Design

There are two types of design interviews: systems design and product design. Interviewers will evaluate your ability to determine what you should build and to solve large problems. Your interviewer will ask you a very broad design problem and evaluate your solution. We aim to match people interviewing with engineers who have related experience, so your conversation will be based in an area you're at least slightly familiar with.

This portion of the interview will consist of one or two conversations, 45 minutes each. You will only need to practice these skills if you are interviewing for a position which requires these competencies. Your interviewer will let you know whether you need to prepare for this portion of the interview.

If you have deep, specialized knowledge, we may ask something specific to that area. More often, we'll ask you to design something you've never built before. And the scope will be large enough that you won't be able to cover everything in detail.

The Systems Design Interview

1. Possible question topics

Interviewers will focus on your familiarity with complex products or systems and assess your ability to arrive at answers in the face of unusual constraints.

You should be familiar with the areas below—but we're not looking for you to be an expert in all of them. You should know enough to weigh design considerations (and understand when to consult an expert) for:

- Concurrency (threads, deadlock, starvation, consistency, coherence)
- Caching
- Database partitioning, replication, sharding, CAP Theorem
- Networking (IPC, TCP/IP)
- Real-world performance (relative performance RAM, disk, your network, SSD)
- Availability and reliability (types of failures, failure units, how failures may manifest, mitigations, etc.)
- Data storage and data aggregation
- QPS capacity / machine estimation (back of the envelope estimates), byte size estimation

2. Example questions

We might ask you to:

- Architect a worldwide video distribution system
- Build Facebook chat
- Design a mobile image search client

How to Practice

Review the complex systems you've already designed. What would you change about your approach if you did it all over again? What worked well?

Think about how you'd design one of Facebook's (or any other large tech company's) existing systems. It's a good practice to think through the complicated, high-scale systems you already use every day. How would you design one from the ground up?

Weigh multiple approaches and reflect on the tradeoffs in your design:

- Performance vs. scalability
- Latency vs. throughput
- Availability vs. consistency

Take any well-known app and imagine you're going to build the primary feature. For example, imagine you're going to build video distribution for Facebook Video, or group chat for WhatsApp. Now figure out how you would build the various pieces out:

- How would you build your backend storage? How does that scale to Facebook's size?
- How would you lay out the application server layer? What are the responsibilities of the various services?
- How would you design your mobile API? What are the hard problems in representing the data being sent from server to client?
- How would you structure your mobile client? How do low-end devices and poor network conditions affect your design?
- As you're designing these systems, run through the list of things we're looking for and make sure you're able to answer them all for each piece of each app.

The Product Design Interview

1. Possible Question topics

Interviewers will focus on building a product or API at scale that supports an end user product or service.

You should be familiar with the areas below—but we're not looking for you to be an expert in all of them. You should know enough to weigh design considerations (and understand when to consult an expert) for:

- Storage data models
- Scalability
- Design patterns

- Data ownership
- Protocols
- Data formats
- Client-server design
- Designing for long term vs. complexity
- Accommodating possible product changes

2. Example Questions

We might ask you to:

- Design a service or product API
- Design a chat service or a feed API
- Design an email server

3. How to practice

Reflect on your projects. Think about the projects you've built. What was easy, what was difficult?

Weigh multiple approaches and reflect on the tradeoffs in your design:

- Easy-to-build APIs vs. long-term APIs
- UI complexity vs. server complexity
- Payload size vs. performance

Articulate any assumptions you make, such as number of users, frequency of use, etc.

Think through specific questions and answer them as you build. For example, if your interviewer asks you to describe how you'd design an email server, you might think through:

- How do you store mail, especially as the system gets large enough that it won't fit on one machine?
- How do you handle mailing lists with large numbers of recipients?
- How do you handle people abusing the system for spam?
- How do you guarantee the reliability of the system in the face of potential system failures?

How to Approach Problems During Your Interview

Overall, imagine you're designing a system and sharing it with a colleague for their input. Just like in the coding interview, think of this as a conversation—not a test! As you're designing, your interviewer will be looking to see if you can follow the steps outlined below and how well you can adjust the design when requirements or constraints change.

1. Analyze the problem

Spend the first three minutes asking clarifying questions. Keep asking until everything the interviewer is looking for is clear to you.

Start with requirements. Your interviewer may ask a vague question like, "How would you architect the backend for a messaging system?" Start by asking about the requirements:

- How many users are we talking about?
- How many messages sent?
- How many messages read?
- What are the latency requirements for sender -- receiver message delivery?
- How are you going to store messages?
- What operations does this data store need to support?
- What operations is it optimized for?

Determine which parts of the problem are important. Consider and explain which will affect the overall design.

2. Design your solution

After gathering requirements, begin at the highest level possible. Outline your approach, then think about how it can be broken down into subparts. Discuss the system pros and cons before you focus on trade-offs and alternatives.

But don't stay high level. Once you get the high-level portion out of the way, pick a component you can do a deep dive into.

Draw boxes-and-arrows whiteboard diagrams that clearly describe the relationships among different system or product components and illustrate the entire problem and solution space.

Narrate your thoughts as you go. Talk through various solutions and the pros and cons of each approach. Explain your choices, discuss tradeoffs, and describe how you make a decision around them.

Think at scale. Identify and discuss any bottlenecks and limitations of your design.

Explain how your solution handles success and failure cases.

3. Keep in mind

Ask questions as you're designing. If there's anything you're unsure about, ask! Ask your interviewer if you're on the right track, and listen to what they have to say.

Avoid using verbiage or jargon you don't know very well. Be prepared to discuss in detail anything you've listed on your resume.

Be honest about what you don't know. Interviewers understand that one person cannot possibly be an expert on all aspects of systems and product design and architecture.

Where to Learn More

- Grokking the System Design Interview (Video)
- System Design Primer (GitHub)
- How do I prepare to answer design questions in a technical interview? (Quora)
- HiredInTech System Design for Tech Interviews course

Part 3: Getting to Know You

This portion of your onsite interview will consist of a single 45-minute session. Your interviewer will want to learn about your background and interests, what you're passionate about in tech, and what kind of impact you want to make. We also want to know what it's like to work with you every day, what kind of a colleague and/or leader you are, and how you handle challenges.

What We Look For

Facebook values self starters who identify and tackle hard problems. Almost everything we do is done within larger groups, so we also value collaboration and partnership. We encourage you to read more about our company values on our Facebook Careers website.

What We Ask

We may ask you to:

- Discuss anything that's on your resume, including current projects and details.
- Provide specific examples about what you did and the resulting impact.
- Critique yourself and share what you learned from a past situation.
- Talk about what you like about your current role and/or being a developer.
- Discuss why you'd like to make a change.

How to Prepare

Just like with coding and design interviews, it's important to prepare ahead of time for interviews designed to get to know you better.

1. Determine what stories you want to share

Start by identifying examples of situations where your behaviors or actions have resulted in positive outcomes or illustrate your skills in leadership, teamwork, planning, taking initiative, and customer service.

Think beyond your work experience. You can share examples of coursework, internships, hobby projects, volunteer work, etc. If possible, include different types of examples to showcase the breadth of your skills in different situations.

Think about challenges and what you've learned. Think through the tough situations you've faced and explain what you've learned from each. Keep in mind that sharing some examples with negative results can effectively highlight your strengths in the face of adversity and showcase your openness to learn and grow.

Be specific. Instead of generalizing about several events, give a detailed account of one event.

Be honest. Don't embellish or omit any part of the story. The interviewer will be able to see if your answer is built on a weak foundation.

2. Use the STAR format

Write out five to seven examples of your successes using the STAR format outlined below. Writing these out before your interview will help you recall all the details and stay on track when you tell your story in the interview, and the STAR approach will help you frame each story you share with a beginning, middle, and end. You can use these stories to answer almost any question in this portion of your interview.

Situation: Describe the situation you were in or the task you needed to accomplish. this situation can be from a previous job, volunteer experience, or any relevant event.

Task: What was your specific goal? How did your goal impact any larger project goals or outcomes?

Action: What specific steps did you take to achieve your goal? What obstacles did you encounter, and what steps did you take to overcome them? (Focus on your individual contribution and not the team's accomplishments. Say 'I' not 'we.')

Result: Describe the outcome of your actions, and don't be shy about taking credit. What happened? What did you accomplish and learn? Help the interviewer understand why these results were important.

3. Get familiar with Facebook's technical environment

What do you want to work on? What features would you like to improve? What about our work excites you? Know about Facebook's teams, projects, products, and challenges so you can ask the interviewer relevant questions. Check out our:

- Facebook Engineering Facebook page
- Facebook Code videos
- Facebook Open Source website
- Engineering Leadership at Facebook blog