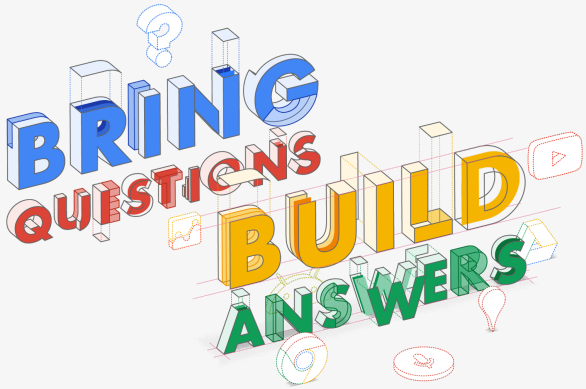
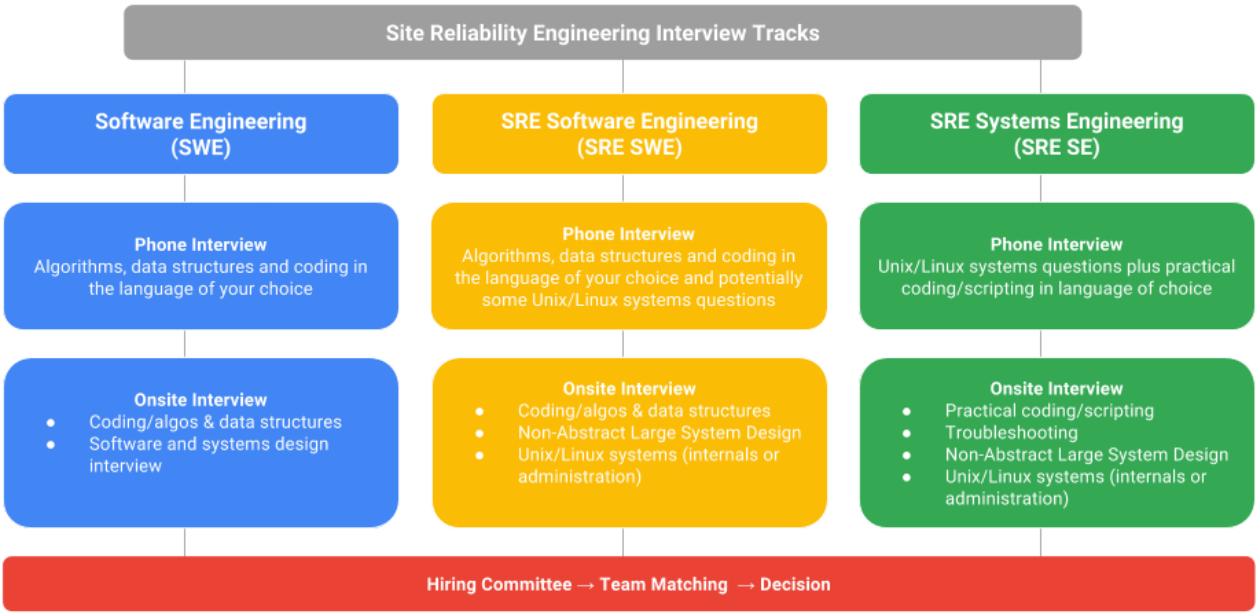


Google Interview Prep Guide
Site Reliability Engineer

What's a Site Reliability Engineer (SRE)?
Everyday, Site Reliability Engineers tackle some of the most complex software and systems issues on the planet. As a SRE, you'll be working with Google code that runs in infrastructure, frontend and backend technology. You'll be responsible for the design, build and development of massively distributed, fault-tolerant software systems and infrastructure, encompassing all of our products and services. This can range from distributed change propagation on live serving systems, to designing distributed hash tables and implementing automation and monitoring solutions for the largest user-facing service in the world. Check out our website and book to learn more about our team and the work that we do. You can also listen to a podcast that one of our SRE directors did with Software Engineering Daily.

Google's engineering organization carries a flat structure in which Managers and Individual Contributors are weighted equally within Google. You can grow your entire career here as an individual contributor or if you have a passion for staying technical, while wanting to grow your people management skills, Google affords this opportunity. The general SRE organizational structure consists of a Site Director, managing varying levels of Site Reliability Managers, who lead teams of 6-10 (a mix of Software and Systems Engineers). SRE team focuses typically fall into two technical areas: Application SRE's (Gmail, Maps, Search, Youtube, Chrome, etc.) and Infrastructure SRE's (Spanner, Bigtable, Dremel, Ganeti, Borg).

The SRE Interview Tracks



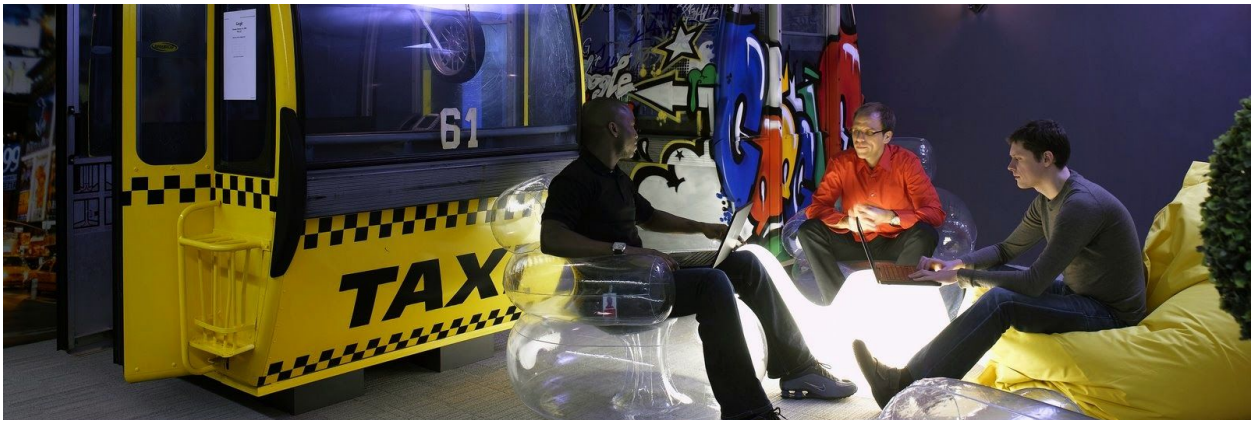
General Interview Tips

Explain: We want to understand how you think, so explain your thought process and decision making throughout the interview. Remember we're not only evaluating your technical ability, but also how you approach problems and try to solve them. Explicitly state and check assumptions with your interviewer to ensure they are reasonable.

Clarify: Many of the questions will be deliberately open-ended to provide insight into what categories and information you value within the technological puzzle. We're looking to see how you engage with the problem and your primary method for solving it. Be sure to talk through your thought process and feel free to ask specific questions if you need clarification.

Improve: Think about ways to improve the solution you present.It's worthwhile to think out loud about your initial thoughts to a question. In many cases, your first answer may need some refining and further explanation. If necessary, start with the brute force solution and improve on it — just let the interviewer know that's what you're doing and why.

Practice: You won't have access to an IDE or compiler during the interview so practice writing code on paper or a whiteboard. Be sure to test your code and ensure it's easily readable without bugs. Don't stress about small syntactical errors like which substring to use for a given method (e.g. start, end or start, length) — just pick one and let your interviewer know.



The Technical Phone Interviews

Your phone interview will cover data structures and algorithms. Be prepared to write around 20-30 lines of code in your strongest language. Approach all scripting as a coding exercise — this should be clean, rich, robust code.

1. You will be asked an open ended question. Ask clarifying questions, devise requirements.
2. You will be asked to explain it in an algorithm.
3. Convert it to a workable code.(Hint: Don't worry about getting it perfect because time is limited. Write what comes but then refine it later. Also make sure you consider corner cases and edge cases, production ready.)
4. Optimize the code, follow it with test cases and find any bugs.

Technical Prep - Software Engineering

Coding: You should know at least one programming language really well, preferably C++, Java, Python, Go or C. You will be expected to know APIs, Object Orientated Design and Programming, how to test your code, as well as come up with corner cases and edge cases for code. Note that we focus on conceptual understanding rather than memorization. Have a look at the Google Style Guide for more info.

Algorithms: Approach the problem with both from the bottom-up and the top-down algorithms. You will be expected to know the complexity of an algorithm and how you can improve/change it. Algorithms that are used to solve Google problems include: sorting (plus searching and binary search), divide-and-conquer, dynamic programming/memoization, greediness, recursion or algorithms linked to a specific data structure. Know Big-O notations (e.g. run time, space) and be ready to discuss standard, well established algorithms. You may wish to discuss or use bullets to outline the algorithm you have in mind before writing code.

Sorting: Be familiar with common sorting functions and on what kind of input data they're efficient on or not. Think about efficiency means in terms of runtime and space used. For example, in exceptional cases insertion-sort or radix-sort are much better than the generic QuickSort/MergeSort/HeapSort answers.

Data structures: Study up on as many other structures and algorithms as possible. Read about the most famous classes of NP-complete problems. You will also need to know about trees, basic tree construction, traversal and manipulation algorithms, hash tables, stacks, arrays, linked lists, priority queues, and when they are appropriate.

Recursion: Many coding problems involve thinking recursively and potentially coding a recursive solution. Prepare for recursion—which can sometimes be tricky if not approached properly. Use recursion to find more elegant solutions to problems that can be solved iteratively.

Mathematics: Some interviewers ask basic discrete math questions. Spend some time before the interview refreshing your memory on (or teaching yourself) the essentials of elementary probability theory and combinatorics. You should be familiar with n-choose-k problems and their ilk.

Technical Prep - Systems Engineering

Practical Coding / Scripting: Can your code tell a computer what to do? This coding exercise will assess simple algorithm/data structure implementation. We are looking for a solution that shows you understand your language usage well with a clean and working implementation that's efficient.

Operating systems: Understand processes, threads, concurrency issues, locks, mutexes, semaphores, monitors and how they work. Understand deadlock, livelock and how to avoid them. Know what resources a process and a thread needs. Understand how context switching works, how it's initiated by the operating system and underlying hardware. Know a little about scheduling and the fundamentals of "modern" concurrency constructs.

Unix / Linux systems: Know what's happening under the hood. Understand kernels, libraries, system calls, memory management, permissions, file systems, client-server protocols and the shell. Check out the online books The Art of Unix Programming and Advanced Programming in the Unix Environment to review system fundamentals.

Troubleshooting: Interviewers are looking for a logical and structured approach to problem solving through network or distributed systems scenarios. Some troubleshooting examples include an App Engine outage and the Life in App Engine Production.

System design: System design questions are used to assess a candidate's ability to combine knowledge, theory, experience and judgement toward solving a real-world engineering problem. Sample topics include distributed systems, designing a system under certain constraints, simplicity, limitations, robustness and tradeoffs. Make sure you also have an understanding of how the internet works and be familiar with the various pieces (routers, domain name servers, load balancers, firewalls, etc.). For information on system design, check out our research on distributed systems and parallel computing.

Mathematics: Estimating large numbers is extremely beneficial when thinking at Google scale. Consider brushing up on your powers of 2 and 10. Also commit to memory "Back-Of-The-Envelope Calculations" that will help you assess system design trade-offs.

Networking (optional): Show off your depth of knowledge and understanding of network theory, like different protocols (TCP/IP, UDP, ICMP, etc), MAC addresses, IP packets, DNS, OSI layers, and load balancing. Check out Computer Networking: A Top-Down Approach.

Resources

Books

Site Reliability Engineering: How Google Runs Production Systems
Betsy Beyer, Chris Jones, Niall Richard Murphy, Jennifer Petoff

Cracking the Coding Interview
Gayle Laakmann McDowell

Programming Interviews Exposed: Secrets to Landing Your Next Job
John Mongan, Eric Giguere, Noah Suojanen, Noah Kindler

Programming Pearls
Jon Bentley

Computer System Engineering
Robert Morris, Samuel Madden

Introduction to Algorithms
Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein

Technical Prep

CodeJam: Practice & Learn
Technical Development Guide
Underneath the covers at Google
Scalable web architecture & distributed systems
How complex systems fail
Building software systems at Google and lessons learned
Putting Google's network to work for you

About Google

Company - Google
The Google story
Life @ Google
Google Developers
Open Source Projects

Google Publications

The Google File System
Bigtable
MapReduce
Google Spanner
Google Chubby

Interview Prep

SRE Hangouts on Air Video
How we hire
Interviewing @ Google
Tech Interviewing @ Google
How to Get a Job at Google
How to Get a Job at Google 2