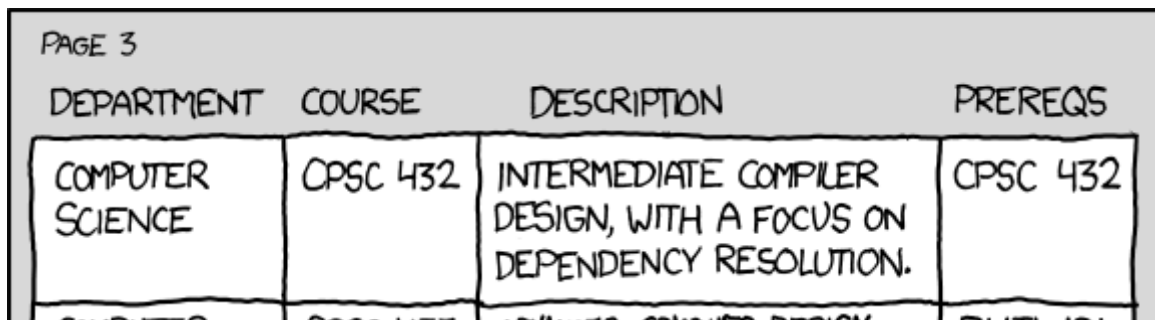


HOW TO ACE A SYSTEMS DESIGN INTERVIEW



PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

Comic courtesy of XKCD, via Creative Commons License

Note: this third installment in our series on doing your best in interviews. Previously: [How to Ace an Algorithms Interview](#) and [The Coding Interview](#).

One interview that candidates often struggle with is the systems design interview. Even if you know your algorithms and write clean code, that code needs to run on a computer somewhere—and then things quickly get complicated. A truly unbelievable amount of complexity lies beneath something as simple as visiting Google in your browser. While most of that complexity is abstracted away from the end user, as a system designer you have to face it head on, and the more you can handle, the better.

At Palantir, many of our teams give a systems design interview along with an algorithms interview and a couple of coding interviews. We don't expect anyone to be an expert at all three disciplines (although some are). We're looking for generalists with depth—people who are good at most things, and great at some. If systems design isn't your strength, that's okay, but you should at least be able to talk and reason competently about a complex system.

Read on to learn about what we're looking for and how you can prepare.

WE'RE MEASURING THREE THINGS

Nominally, this interview appears to require knowledge of systems and a knack for design—and it does. What makes it interesting, though, and sets it apart from a coding or an algorithms interview, is that whatever solution you come up with during the interview is just a side effect. What we actually care about is the process.

In other words, the systems design interview is all about communication.

This reflects what actually working at Palantir is like. As engineers we have a tremendous amount of freedom. We aren't asked to implement fully-specified features. Instead we take ownership of *open-ended problems*, and it's our job to come up with the best solution to each. We need people we can trust to do the right thing without a lot of supervision—people who can own large projects and take them consistently in the right direction. Invariably, this means being able to communicate effectively with the people around you. Working on problems with huge scope isn't something you can do in a vacuum.

IT'S AN OPEN-ENDED CONVERSATION

Usually we'll start by asking you to design a system that performs a given task. The prompt will be simple, but don't be fooled—these problems are wide and bottomless, and the point of the interview is to see how much volume you can cover in 45 minutes.

For the most part, you'll be steering the conversation. It's up to you to understand the problem. That might mean asking questions, sketching diagrams on the board, and bouncing ideas off your interviewer. Do you know the constraints? What kind of inputs does your system need to handle? You have to get a sense for the scope of the problem before you start exploring the space of possible solutions. And remember, there is no single right answer to a real-world problem. Everything is a tradeoff.

TOPICS

Systems are complex, and when you're designing a system you're grappling with its full complexity. Given this, there are many topics you should be familiar with, such as:

- **Concurrency.** Do you understand threads, deadlock, and starvation? Do you know how to parallelize algorithms? Do you understand consistency and coherence?
- **Networking.** Do you roughly understand IPC and TCP/IP? Do you know the difference between throughput and latency, and when each is the relevant factor?
- **Abstraction.** You should understand the systems you're building upon. Do you know roughly how an OS, file system, and database work? Do you know about the various levels of caching in a modern OS?
- **Real-World Performance.** You should be familiar with the speed of everything your computer can do, including the relative performance of RAM, disk, SSD and your network.
- **Estimation.** Estimation, especially in the form of a back-of-the-envelope calculation, is important because it helps you narrow down the list of possible solutions to only the ones that are feasible. Then you have only a few prototypes or micro-benchmarks to write.
- **Availability and Reliability.** Are you thinking about how things can fail, especially in a distributed environment? Do know how to design a system to cope with network failures? Do you understand durability?

Remember, we're not looking for mastery of all these topics. We're looking for *familiarity*. We just want to make sure you have a good lay of the land, so you know which questions to ask and when to consult an expert.

HOW TO PREPARE

How do you get better at something? If your answer isn't along the lines of "practice" or "hard work," then I have a bridge to sell you. Just like you have to write a lot of code to get better at coding and do a lot of drills to get really good at basketball, you'll need practice to get better at design. Here are some activities that can help:

- **Do mock design sessions.** Grab an empty room and a fellow engineer, and ask her to give you a design problem, preferably related to something she's worked on. Don't think of it as an interview—just try to come up with the best solution you can. Design interviews are similar to actual design sessions, so getting better at one will make you better at the other.
- **Work on an actual system.** Contribute to OSS or build something with a friend. Treat your class projects as more than just academic exercises—actually focus on the architecture and the tradeoffs behind each decision. As with most things, the best way to learn is by doing.
- **Do back-of-the-envelope calculations** for something you're building and then write micro-benchmarks to verify them. If your micro-benchmarks don't match your back-of-the-envelope numbers, some part of your mental model will have to give, and you'll learn something in the process.
- **Dig into the performance characteristics of an open source system.** For example, take a look at LevelDB. It's new and clean and small and well-documented. Read about the implementation to understand how it stores its data on disk and how it compacts the data into levels. Ask yourself questions about tradeoffs: which kinds of data and sizes are optimal, and which degrade read/write performance?(*Hint: think about random vs. sequential writes.*)

- Learn how databases and operating systems work under the hood. These technologies are not only tools in your belt, but also a great source of design inspiration. If you can think like a DB or an OS and understand how each solves the problems it was designed to solve, you'll be able to apply that mindset to other systems.

FINAL THOUGHT: RELAX AND BE CREATIVE

The systems design interview can be difficult, but it's also a place to be creative and to take joy in the imagining of systems unbuilt. If you listen carefully, make sure you fully understand the problem, and then take a clear, straightforward approach to communicating your ideas, you should do fine.

Good luck!