* Top-down - Looking down from a very high level build out a design and flesh out the design as various components get done and here are a handful of components that I could see....
* Bottom-up - Looking from the ground up, here are bits and pieces one could build to try to put together....
* Requirement clarification - Asking questions about the projected scale, size, budget, and team used for this design. You could try to have a person code a very simplified word processor or you could plot to spend hundreds of millions of dollars to make the ultimate document management system that you believe is how you Google Doc taken to an extreme. Also in here is the ability to ask something like, "What do you mean by Google Doc? How much of that functionality are you wanting to duplicate?" questions as well.

Key Principles to remember

1. Asynchronous is good  (use Queues, Topics etc.)  
   2) Parallel is good  (Multi-threading, load balancing etc.)  
   3) Avoid points of contention e.g. synchronization  
   4) Avoid writing to disk until you must - cache like crazy

<https://hackernoon.com/how-not-to-design-netflix-in-your-45-minute-system-design-interview-64953391a054>

<https://hackernoon.com/anatomy-of-a-system-design-interview-4cb57d75a53f>

## 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](https://code.google.com/p/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.

<https://github.com/checkcheckzz/system-design-interview>

<https://www.palantir.com/2011/09/how-to-ace-an-algorithms-interview/>

<https://www.palantir.com/2011/10/the-coding-interview/>

<https://www.hiredintech.com/classrooms/interview-strategies/lesson/88>

<http://www.lecloud.net/post/7295452622/scalability-for-dummies-part-1-clones>

<http://www.lecloud.net/post/7994751381/scalability-for-dummies-part-2-database>

<http://www.lecloud.net/post/9246290032/scalability-for-dummies-part-3-cache>

<http://www.lecloud.net/post/9699762917/scalability-for-dummies-part-4-asynchronism>

<http://tutorials.jenkov.com/software-architecture/load-balancing.html>

<https://www.interviewbit.com/problems>