

👋 The Only 5 Things You  
Need To Do To Land Six  
Figure Tech Offers



Join Now For A FREE In-Depth Interview  
Guide!

Get the latest info on compensation, interview prep, and  
life in tech! 🛎️ 📅

Enter full name

---

Enter email address

---

[Join Us!](#)

*We promise not to spam you. You can unsubscribe at any time.*

## About Me

Hi! Nice to e-meet you. My name's Jeffrey. I'm currently a software engineer at Stripe, previously I was at Uber and Amazon – I have extensive experience interviewing and job-hopping. In the past 2.5 years since I graduated college, I received verbal/written offers from the following places: Snap, LinkedIn, Amazon, Uber, and Stripe. In 2.5 years, my total compensation went from \$145k @ Amazon to \$360k @ Stripe. Here are a few things I'd like to share with you that will help you on your path to landing a six-figure offer from a top tech company!

## 1. LeetCode The Fundamentals



This is obvious, but let me be specific here:

**YOU DO NOT NEED TO LEETCODE 500 PROBLEMS.**

In my experience, 100 problems is enough. The ideal distribution of these 100 problems should be ideally 90/10 or 95/5 medium and hard. If you need to do easys, start off with them, but keep going until you hit 100 problems total in the medium/hard category. In these 100 problems, here are the exact topics you should aim to hit in general:

1. BFS/DFS (know recursive + iterative implementation)
2. Trees/Trie/Graphs/Heaps

3. Array/HashMaps/Sets
4. Linked Lists/Doubly-Linked List/Ghost Nodes
5. Sliding Window
6. Backtracking
  
7. Topological Sort
8. Sorting (know implementation of one of the sorting algorithms: quick sort, merge sort, etc)
9. Recursion
10. Dynamic Programming
11. Inorder/Preorder/Postorder Traversal
12. Binary Search

Here's a list of the exact problems I did in my most recent recruiting cycle. Some of these will require LeetCode Premium, which is a purchase I found very worthwhile. I personally recommend doing at most two a day to not burn yourself out. If you can't solve the question within a 45 minute time frame, cut the loss and look at the solution.

Here's another good list from Blind.

**HOW TO KNOW IF YOU ARE READY: You can solve most LC mediums in about 20 minutes.**

## 2. Do Mock Interviews From Both Sides

Head over to [pramp.com](https://pramp.com). Aim to complete at least 5 interview sessions each as an interviewer and an interviewee. You will be surprised at how much you will learn.



The questions on Pramp are generally a bit easier than what you will likely encounter in a real interview at a top tech company, so if you are able to nail every single Pramp interview, you will be ready.

**HOW TO KNOW YOU ARE READY:** You have completed at least 5 interview sessions each and you were able to ace at least five consecutively.

## 3. Watch/Read System Design Videos/Articles

System design is usually the killer for folks. This one is the hardest to study for – there is usually no one single correct answer in the interview. It is more important to discuss trade-offs and the reasoning for a certain design choice. Below are the resources that I used to learn system design.

**A great place to start:** [System Design Primer](#) (this is not enough in my experience).

Here are the exact playlists/videos that I used to brush up on system design:

1. [Gaurav Sen System Design Fundamentals Playlist](#)
2. [System Design Interviewer](#) (watch all the question walkthroughs)
3. [Tech Dummies Narendra System Design Questions](#) (watch all the question walkthroughs)
4. [What is Apache Kafka?](#)
5. [What is Cassandra?](#)



Gaurav Sen – the best YouTuber for system design IMHO.

Oftentimes, the interviewer will ask you how a specific database/service will work. If the questions is around something like “design Facebook messenger” or “design Twitter news feed”, you can easily see how other companies design their products by reading their engineering blogs. You can simply re-use the design from these engineering blogs.

Here is a list of relevant blogs/readings that I read and you will probably find helpful:

1. [How Discord Stores Billions Of Messages](#)
2. [Designing A Rate Limiter](#)
3. [Designing A News Feed](#)

4. The Infrastructure Behind Twitter Scale

5. How To Show Top K Songs Like Spotify

## HOW TO KNOW YOU ARE READY CHECKLIST:

You know the answers to the following design questions:

1. Design A Rate Limiter
2. Design YouTube Hit Counter
3. Design Facebook Messenger
4. Design Instagram News Feed
5. Design Distributed Cache
6. Design Facebook/Twitter Search
7. Design Autocomplete For Google Search
8. Design Top N Songs (count-min-sketch)
9. Design Facebook Live Commenting
10. Design Nearby Restaurants Like Yelp (quadtree)
11. Design Facebook Privacy Settings
12. Design Web Crawler

You know the following concepts/can answer the following questions:

1. HTTP Polling (Long And Short) vs. Web Sockets
2. Database sharding
3. SQL vs. NoSQL
4. ACID transactions
5. The types of NoSQL databases and their difference (wide-column store, document store, etc)
6. DNS Name Resolution
7. What is Redis? How do in-memory caches work? What are their eviction policies?
8. Publisher/Subscriber model
9. What is a message queue?
10. How do load balancers work? What is a good load balancing policy?
11. What is a push/pull model?
12. What is CDN and how do they work?
13. How do replicas give fault tolerance and how do they work?
14. HTTP POST/GET



## 15. Denormalization

# 4. Write Down Your System Design Interviews

This is an underrated step. In my experience, I found that using Pramp was not a good idea for system design interviews – oftentimes people were in the same boat as me; they were not experienced with system design, so there was no great way to tell if my system design answers during mock interviews were correct.

### Design a Rate Limiter:

#### Functional/Non-Functional Requirements:

1. Limit any number of requests to a system on a per-client basis.
2. Highly available.

#### Algorithms for Rate Limiting:

1. Leaky Bucket - bucket with a capacity N. Add requests to the bucket when they are made, remove them when they are completed. Discard when the request comes in and the bucket is full.

Pros: Easy to implement on a single server.

Cons: Hard to do on a distributed system. Need to coordinate with multiple machines.

2. Fixed Window - have a threshold on a windowed time period. Keep track with counter.

Con: Issue is that on boundary of window, we can process up to 2x the amount the rate limit.

3. **Solution:** Sliding Window Log - For each client, keep track of request and timestamp. When a new request comes in, count how many requests came in in the window prior to the timestamp for the new request. Remove all outdated requests. If count > threshold, remove/drop. Otherwise, add to log and continue. Use a DB/key-value store like Redis to store this information.

#### Handling Scale:

1. Easy way is to have each client's request go to a single node. But this leads to issues with uneven load.

2.: use centralized data store like Redis to store counts for each client. But a single centralized data store is bad. Because everyone is querying it.

3. The solution is to have db node for each rate limiter service. They all keep replicas of the

Example from my doc: Design A Rate Limiter

**Bottlenecks/issues:**

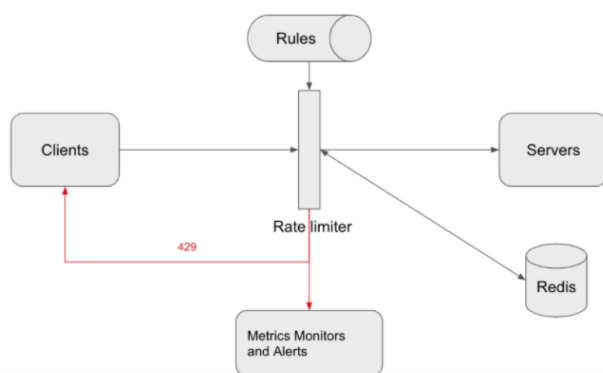
We could use a lock to solve race condition, but that would affect performance with latency.

**Solution:** We have a copy of the current logs for rate limiting in memory for each client locally.

We also have a service that synchronizes this frequently with the database.

Pro: faster, reading in memory, no database query call, all done in memory.

Con: Won't be exactly the threshold that we have, a few requests will leak.



Example from my doc: Design A Rate Limiter

I personally like to use Sketchboard for diagrams.

What I recommend doing is having a Google document that contains every single system design question that you aim to solve. Break it up into the following steps:

1. Functional/Non-Functional Requirements
2. Scale estimation/back-of-envelope-calculations
3. Database schema
4. Simple design (assume traffic can fit on one machine)
5. Scaling up/bottlenecks (how to make things distributed on multiple machines)
6. Optimizing for efficiency (caching, sharding, partition tolerance)

Give yourself a question to solve, and then write it down in the Google document. I personally found that this was helpful as it helped me walk



through the formal train of thought that would be used in an interview and it gave me a point of reference to which I could go back and visit my answers. Compare your answer with a solution available on YouTube.

## HOW TO KNOW YOU'RE READY:

1. You are able to do back-of-the-envelope calculations easily and swiftly.
2. You are able to address bottlenecks and design scaled-up solutions for each of the classic design questions listed in part 3.
3. You are able to address trade-offs for databases (SQL vs. NoSQL) and design a database schema appropriate for your use case.
4. You are able to address partition tolerance and load balancing.
5. Most importantly, you are able to walk through each of the problems listed in part 3 using the 6 step pattern outlined at the beginning of this section.

## 5. Start Applying To Companies You Do Not Care About First

Oftentimes, people rush straight into applying into the companies that they care about first. What I personally recommend is start your actual interview process by applying to companies that you personally have little interest in joining to gain interview experience/confidence.

For example – if you wish to work at Facebook/Google, save their interviews for last. Start by interviewing with companies that mean less to you. I classify companies in three categories: **low, medium, high**. This represents how badly I want to get an offer from said company/their priority to me.

My rule of thumb is as follows:

1. Apply to companies that are of low priority. When you are able to make final rounds consistently and not bomb them, start going for medium priority companies. You should have around 3–4 of these companies.
2. Repeat for medium priority companies. When you are able to make a couple final rounds for the medium priority companies,

make a couple final rounds for the medium priority companies and you feel like you do not bomb them completely, move on to your high priority companies. You should have around 3-4 of these companies.

3. Interview with high priority companies. You should have around 3-4 of these companies.


# Conclusion

Interviewing is a grind – the important thing here is to not burn yourself out. If you follow these steps listed above at a consistent pace (even if it is a slow pace), I can promise you you will see improvement in your interviewing ability.

You can expect the whole process to take about 3-4 months, from initial practicing of LeetCode to signing an offer. With that being said, good luck and I hope that this helped!

 FriendlyNeighborhoodEngineers    December 4, 2021    Software Engineering

---

 [New Grad Pay Is Insane In Software Engineering! →](#)

# Leave a Reply

