System Design Interviews: A Step-By-Step Guide

How to efficiently manage a 45 minutes system design interview.

System design interviews (SDIs) have gained a lot of focus in the past few years. Now a standard part of every software engineering interview, these interviews assess a candidate's ability to handle complex systems. **Performance in these interviews determines the position and salary offered by the interviewing company.** Engineers usually struggle with SDIs partly due to their lack of experience developing large-scale systems and partly due to the unstructured nature of SDIs. Engineers who have experience building such systems still find these interviews uncomfortable, mainly because the design problems are open-ended and have no standard answer.

Generally, software engineers have difficulty with system design interviews for three primary reasons:

- SDIs are **unstructured**, where candidates are asked to take on an open-ended design problem that doesn't have a standard solution.

- Candidates **lack experience** in developing complex and large-scale systems.
- Candidates did not **spend enough time** preparing for SDIs.

Like coding interviews, candidates who have not put a deliberate effort to prepare for SDIs, mostly perform poorly, especially at top companies like Google, Facebook, Amazon, Microsoft, etc. In these companies, candidates who do not perform above average have a limited chance to get an offer. **On the other hand, a good performance always results in a better offer (a higher position and salary) since it proves the candidate's ability to handle a complex system.**

During my professional career, I have taken more than 200 system design interviews. My goal in this post is to provide a step-by-step guide on how to approach system design questions during an interview; it will assist with overcoming the problems highlighted above.

> *Check **Grokking the System Design Interview** to learn about important distributed system concepts.*

In my experience, candidates who have performed better tend to follow these seven steps:

1. Requirements clarifications: Always ask questions to find the exact scope of the problem you are solving.

2. Back-of-the-envelope estimation: It's always a good idea to estimate the scale of the system you are going to design. This will also help later, when you will be focusing on scaling, partitioning, load balancing, caching, etc.

3. System interface definition: Define what APIs are expected from the system. This will not only establish the exact contract expected from the system but also ensure that you have not gotten any requirements wrong.

4. Define data model: Defining the system data model early on will clarify how data will flow among different components of the system and later will also guide towards the data partitioning and management.

5. High-level design: Draw a block diagram with 5–6 boxes representing the core components of your system. You should identify enough components that are needed to solve the actual problem from end to end.

6. Detailed design: Dig deeper into 2–3 components; interviewer's feedback should always guide you towards which parts of the system she wants you to explain further. You should be able to provide different options, their pros and cons, and why are you choosing them?

7. Identifying and resolving bottlenecks: Try to discuss as many bottlenecks (and different approaches to mitigate them) as possible.

Let's discuss each step in detail with a real case study:

Step 1: Requirements clarifications

It is always a good idea to ask questions about the exact scope of the problem we are solving. Design questions are mostly open-ended, and they don't have ONE correct answer; that's why clarifying ambiguities early in the interview becomes critical. Candidates who spend enough time defining the end goals of the system always have a better chance to be successful in the interview. Also, since we only have 35–40 minutes to design a (supposedly) large system, we should clarify what parts of the system we will be focusing on.

Let's expand this with an actual example of designing a Twitter-like service. Here are some questions for designing Twitter that should be answered before moving on to the next steps:

- Will users of our service be able to post tweets and follow other people?
- Should we also design to create and display the user's timeline?
- Will tweets contain photos and videos?
- Are we focusing on the backend only, or are we developing the front-end too?
- Will users be able to search tweets?
- Do we need to display hot trending topics?
- Will there be any push notification for new (or important) tweets?

All such questions will determine how our end design will look like.

Photo by Malcolm Lightbody on Unsplash

Step 2: Back-of-the-envelope estimation

It is always a good idea to estimate the scale of the system we're going to design. This will also help later when we will be focusing on scaling, partitioning, load balancing, and caching.

- What scale is expected from the system (e.g., number of new tweets, number of tweet views, number of timeline generations per sec., etc.)?
- How much storage will we need? We will have different storage requirements if users can have photos and videos in their tweets.
- What network bandwidth usage are we expecting? This will be crucial in deciding how we will manage traffic and balance load between servers.

Step 3: System interface definition

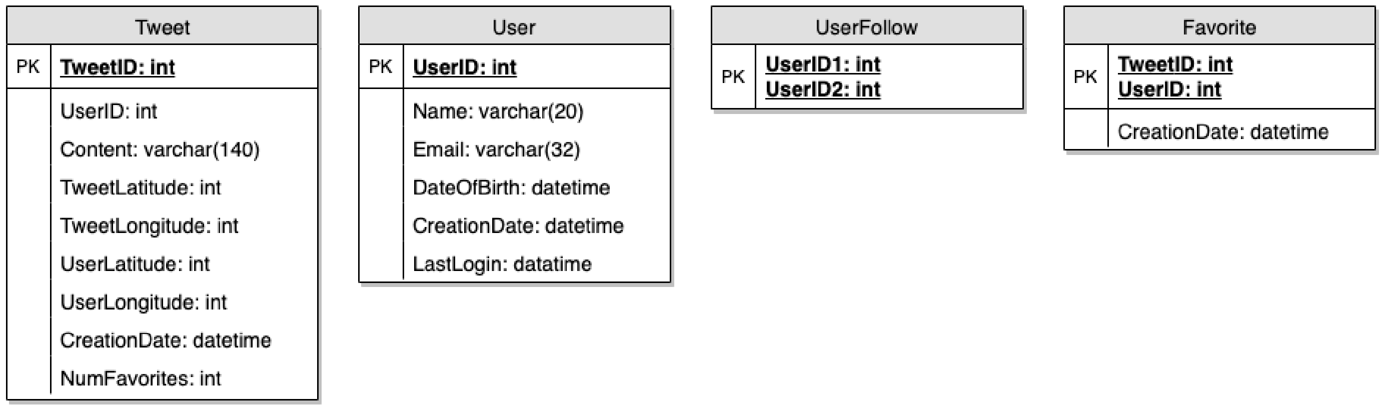
Define what APIs are expected from the system. This will not only establish the exact contract expected from the system but will also ensure that we haven't gotten any requirements wrong. Some examples of APIs for our Twitter-like service will be:

```
postTweet(user_id, tweet_data, tweet_location, timestamp, …)

generateTimeline(user_id, current_time, user_location, …)

markTweetFavorite(user_id, tweet_id, timestamp, …)
```

Step 4: Defining data model

Defining the data model in the early part of the interview will clarify how data will flow between different components of the system. Later, it will guide for data partitioning and management. The candidate should be able to identify various entities of the system, how they will interact with each other, and different aspects of data management like storage, transportation, encryption, etc. Here are some entities for our Twitter-like service:

User: UserID, Name, Email, DoB, CreationData, LastLogin, etc.
Tweet: TweetID, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.
UserFollowo: UserdID1, UserID2
FavoriteTweets: UserID, TweetID, TimeStamp

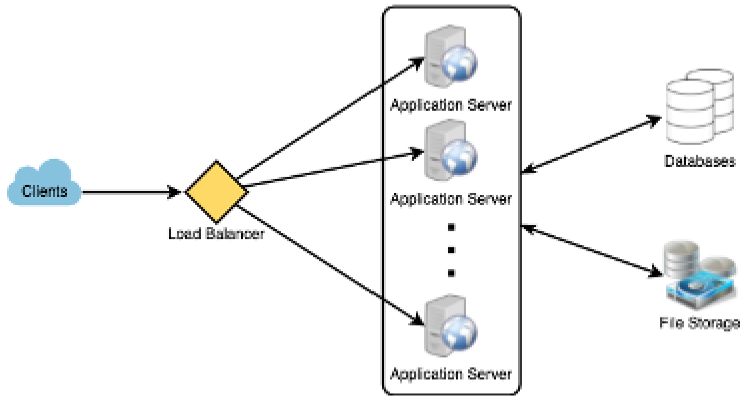


Which database system should we use? Will NoSQL like Cassandra best fit our needs, or should we use a MySQL-like solution? What kind of block storage should we use to store photos and videos?

Step 5: High-level design

Draw a block diagram with 5–6 boxes representing the core components of our system. We should identify enough components that are needed to solve the actual problem from end to end.

For Twitter, at a high level, we will need multiple application servers to serve all the read/write requests with load balancers in front of them for traffic distributions. If we're assuming that we will have a lot more read traffic (as compared to write), we can decide to have separate servers for handling these scenarios. On the backend, we need an efficient database that can store all the tweets and can support a huge number of reads. We will also need a distributed file storage system for storing photos and videos.

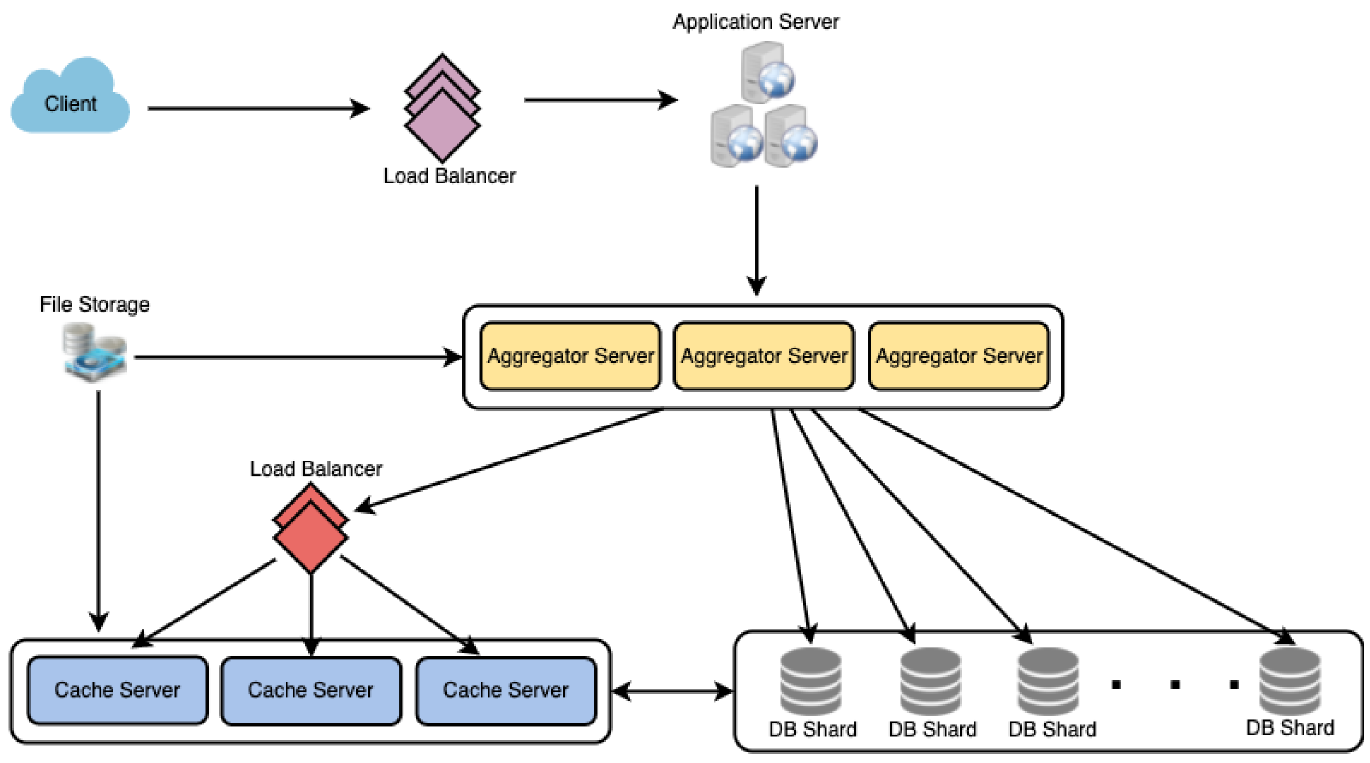


Step 6: Detailed design

Dig deeper into two or three major components; the interviewer's feedback should always guide us to what parts of the system need further discussion. We should be able to present different approaches, their pros and cons, and explain why we will prefer one approach over the other. Remember, there is no single answer; the only important thing is to consider tradeoffs between different options while keeping system constraints in mind.

- Since we will be storing a massive amount of data, how should we partition our data to distribute it to multiple databases? Should we try to store all the data of a user on the same database? What issue could it cause?
- How will we handle hot users who tweet a lot or follow lots of people?
- Since users' timelines will contain the most recent (and relevant) tweets, should we try to store our data in such a way that is optimized for scanning the latest tweets?
- How much and at which layer should we introduce cache to speed things up?

- What components need better load balancing?



Step 7: Identifying and resolving bottlenecks

Try to discuss as many bottlenecks as possible and different approaches to mitigate them.

- Is there any single point of failure in our system? What are we doing to mitigate it?
- Do we have enough replicas of the data so that if we lose a few servers, we can still serve our users?
- Similarly, do we have enough copies of different services running such that a few failures will not cause a total system shutdown?
- How are we monitoring the performance of our service? Do we get alerts whenever critical components fail, or their performance degrades?

Summary

To sum up, preparing well and being organized during the interview is key to success in a system design interview. The steps mentioned above should guide you to stay on track and cover all the different aspects while designing a system.

Please take a look at **Grokking the System Design Interview** and **Grokking the Advanced System Design Interview** for some good examples of system design interview questions and their answers