System Design Interviews: A step by step guide

We'll cover the following

A lot of software engineers struggle with system design interviews (SDIs) primarily because of three reasons:

- The unstructured nature of SDIs, where the candidates are asked to work on an open-ended design problem that doesn't have a standard answer.
- Candidates lack experience in developing complex and large-scale systems.

- Candidates did not spend enough time to prepare for SDIs.

Like coding interviews, candidates who haven't 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 (higher position and salary) since it shows the candidate's ability to handle a complex system.

In this course, we'll follow a step by step approach to solve multiple design problems. First, let's go through these steps:

Step 1: Requirements clarifications#

It is always a good idea to ask questions about the exact scope of the problem we are trying to solve. 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 to define 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.

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 focus 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 establish the exact contract expected from the system and ensure if 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, user_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 system components. Later, it will guide for data partitioning and management. The candidate should identify various system entities, 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:

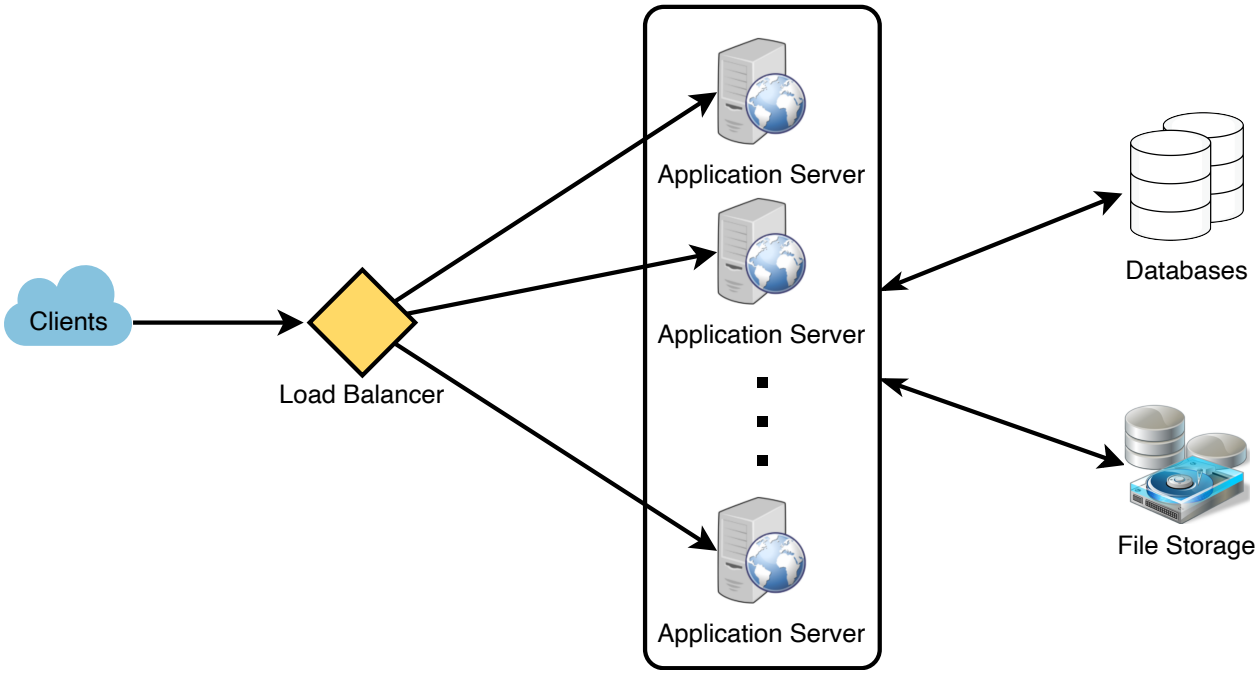
User: UserID, Name, Email, DoB, CreationDate, LastLogin, etc.
Tweet: TweetID, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.
UserFollow: UserID1, UserID2
FavoriteTweets: UserID, TweetID, TimeStamp

Which database system should we use? Will NoSQL like Cassandra (https://en.wikipedia.org/wiki/Apache_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 (compared to write), we can decide to have separate servers to handle these scenarios. On the back-end, we need an efficient database that can store all the tweets and support a large 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 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' timeline will contain the most recent (and relevant)

tweets, should we try to store our data so that it 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 we can still serve our users if we lose a few servers?
- 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#

In short, preparation and being organized during the interview are the keys to success in system design interviews. The steps mentioned above should guide you to remain on track and cover all the different aspects

while designing a system.

Let’s apply the above guidelines to design a few systems that are asked in SDIs.

Happy learning!
Design Guru’s team

Next →

Designing a URL Shortening service li...

Mark as Completed

Report an Issue