System Design Guide

We'll follow a step-by-step approach to solve multiple design problems.

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.

This is because system design questions are mostly open-ended, and they don't have just one correct answer.

Thus, clarifying ambiguities early in the interview becomes critical.

Let's expand on this with an actual example of designing a Twitter-like service.

Here are some clarifying questions for designing Twitter that should be asked first:

- 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?
- Can the tweets contain photos and videos?
- Are we focusing on the backend only, or are we developing the frontend too?
- Will users be able to search for tweets?
- Do we need to display hot trending topics?
- Will there be any push notifications for new (or important) tweets?

Step 2: Back-of-the-Envelope Estimation

It is always a good idea to estimate the scale of the system that we are 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 second 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 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, user_location, timestamp, ...)`
- `generateTimeline(user_id, current_time, user_location, ...)`
- `markTweetFavourite(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 us in data partitioning and management.

You should be able to 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
- **FavouriteTweets:** userID, tweetID, timestamp

It is also good to think about what database system we should use.

Will NoSQL like Cassandra best fit our needs? Or should be 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.

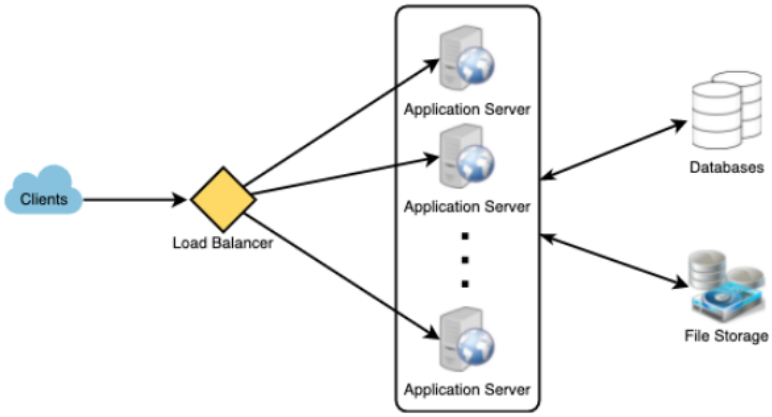
We should identify enough components that are needed to solve the actual problem from end to end.

For Twitter, at a high level, we will need multiple application servers to serve all the read/write requests with load balances 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 backend, 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 as 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.

The most 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 into multiple databases? Should we try to store all the data of a user on the same database? What issues 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 such that it it optimised for scanning the latest tweets?
- How much and at which layer should we introduce a 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 components fail or their performance degrades?