Part 3: Introduction

You've made it to Part 3! So far, you've learned some high-level strategies in Part 1 and some fundamental concepts in Part 2. **Now it's time to get practical.** Part 3 is the first time all of these theories start to fit together into something you can actually do. In other words, it's time to learn a process to start designing systems.

About this 3-step framework

Fundamentals of this framework

This framework is predicated on the following fundamentals:

1. We propose an opinionated method that can be applied to reason over most system design interviews. The less you have to think about the process, the more you can focus on the problems at hand.
2. Our method is broken down into 3 steps. Each of these steps can be seen as functions. They take inputs and have one output. As such, they can be practiced in isolation.
3. We're teaching you a process. This means you'll be ahead of most people who might be very knowledgeable in a few topics but lack a process. You'll come across as a systematic problem solver who's able to decompose a problem and come up with a reasonable architecture in 30 minutes. Again, you only have to understand 20% of the concepts to address 80% of problems you'll encounter.

How to use this framework

In order to use this framework effectively, each step needs to be practiced separately. The three steps are:

1. Requirements
2. Data types, access patterns, and scale
3. Design

Each step has its inputs and outputs at the start of the chapter. There are exercises at the end. Focus on understanding the process. The idea is that once you've learned the process, you can work on improving each of these separately.

Limitations

This is a prescriptive method. As such, the approach might not fit 100% of the system design questions that you encounter. We designed this method to help mid-level / senior engineers get started with system design interviews by using a systematic approach and developing a framework to tackle these questions effectively. You should take this and adapt it so it works best for you.

We intentionally do not tell you how much time to take in each step because we believe that would be an ineffective way to learn a new behavior. If you're told that something takes 5 minutes, and as a beginner you lack 5 minutes of content, you'll end up getting good at "wasting 5 minutes of time." But if you focus on the task instead of the time, you'll get good at doing the thing. This may seem counterintuitive, but it's better for you in the long run. Thank us later.

Overview of the 3 steps

Don't worry, we'll go through each of these one-by-one in detail. And by the end of this chapter, we'll run entire problems through the whole 3-step framework so you can understand how it works holistically. In the meantime, here's a high-level overview of the 3 steps and their inputs and outputs so you can get a rough idea of the process.

Step 1: Requirements

Inputs	Problem statement given by your interviewer.
Outputs	List of functional and non-functional requirements.

Step 2: Data Types, API and Scale

Inputs	- Functional and non-functional requirements. - Problem statement given by your interviewer.
Outputs	- List of Data Types we need to store. - Access patterns for these data types. - Scale of the data and requests the system needs to serve.

Step 3: Design

Inputs	- Functional and non-functional requirements. - Problem statement given by your interviewer. - List of Data Types we need to store. - Access patterns for these data types (API). - Scale of the data and requests the system needs to serve.
Outputs	- Data storage. - Microservices.

Step 1: Requirements

Inputs	Problem statement given by your interviewer.
Outputs	List of functional and non-functional requirements.

System design questions are open-ended and highly underspecified. Inexperienced candidates often dive right into design, without having the full picture of what needs to be built. This is a mistake.

The first step is to specify the requirements. It's up to you to ask the right questions.

Rule of thumb

What makes these questions difficult or complicated is not the fact that they are inherently complicated. It's because the interviewer intentionally chooses to withhold information. Information that you can only get if you ask the right questions.

Rule of thumb

During requirements gathering… if your interviewer says something that seems random, you'll surely need to use it. Remember, interviewers usually withhold information, so if they make the extra effort to share a specific detail, it's because that information is going to be used. For example, if they say that you have "30 days to build this system," the mention of "30 days" is so random/specific that it must be important.

1.1 Functional Requirements

You should start with the functional requirements first—that is, the core product features and use cases that the system needs to support.

Go ahead and type out "Functional requirements" and start asking clarification questions to your interviewer. The number of questions you should ask will vary depending on how underspecified the given problem statement is. Your goal is to ask just enough questions to gather all use cases for the system.

Treat the system as a black box. No thinking about design, implementation, or pretty much anything technical. The sole goal of this first step is to specify what needs to be built. Not how. Not the scale. Focus on the "what."

For example, if you were asked to design Twitter, what clarification questions would you ask in order to scope out functional requirements?

Here are a few steps to guide your requirement gathering:

Identify the main business objects and their relations

Start by identifying the main business objects and their relations. For example, in the case of Twitter there are two main objects of interest: (1) Accounts and (2) Tweets.

Now think about clarifying the relation between these objects. What's the relation between accounts and tweets? What's the relation between two accounts? What's the relation between two tweets? You want to think about the cross product between objects to come up with ideas for use cases and ask relevant questions. For example:

	Account	**Tweet**
Account	Can follow	Can publish Can like Can retweet
Tweet		Can reference (retweet)

- An account can follow other accounts (Account x Account)
- An account can publish a tweet (Account x Tweet)
- A tweet can reference another tweet, i.e., be a "retweet". (Tweet x Tweet)

Then dive deeper into each of the objects of interest. For example, what makes up a tweet? Can it contain media?

Tip

Media is a common part of system design. Always ask yourself whether any of the business objects you identified can hold media.

Think about the possible access patterns for these objects

Access patterns are probably the single most influential part of design because they determine how data will be stored.

Let's think about the cross product of our objects again. This time we want to identify how data will be retrieved from the system.

The general shape of an access pattern requirement is:

- Given [object A], get all related [object B]

So, applying this idea to our Twitter example, we might end up with the following access patterns:

- **Given an account:**
    - Get all of its followers. (Account → Account)
    - Get all the other accounts they follow. (Account → Account)
    - Get all of its tweets. (Account → Tweet)
    - Get a curated feed of tweets for accounts they follow. (Account → Tweet)
- **Given a tweet:**
    - Get all accounts that liked it. (Tweet → Account)
    - Get all accounts that retweeted it. (Tweet → Account)

We're not suggesting you blindly implement all of these, but rather that you consider them as possible access patterns for your clarification questions. For example, should we be able to get all accounts that liked a tweet? Or would the number be enough?

Also, bear in mind that when we say "get all," we don't necessarily mean that there will be a single endpoint that returns all of them in one go. It could be a paged endpoint, but that's an implementation detail and falls out of scope for step 1. We want to focus on identifying the desired access patterns, not how they will be implemented.

For these access patterns, you should also consider ranking. Are there any access patterns that require ranking the object? In this example, "creating a curated feed of tweets" will require further clarification. Strive for simplicity first. Can you return

them sorted by chronological time? Identify these access patterns of interest, like the curated feed, and get a feel for what your interviewer is looking for: do they want you to suggest an algorithm for a feed?

Consider mutability

Finally, as we do throughout this guide, you should always consider mutability. Can the objects the system holds be mutated? Or can they be assumed to be immutable?

For example: Can tweets be edited after they're published?

Another flavor of mutability is deletion. Can these business objects be deleted? What would the consequences be?

For example: Can tweets be deleted? Can accounts be deleted? What happens to tweets when an account is deleted?

It might sound like a small detail at first, but mutability can limit our ability to use caching in our design (more on this in step 3).

Remember: Functional Requirements

1. Identify the main objects and their relations.
2. What information do these objects hold? Are they mutable?
3. Think about access patterns. "Given object X, return all related objects Y." Consider the cross product of all related objects.
4. List all the requirements you've identified and validate with your interviewer.

Example of Functional Requirement gathering

Example 1: Design TikTok

Let's say you are not familiar with TikTok. What do you do next? Probably ask your interviewer, right? Even if you know the platform well, it's wise to start by clarifying rather than assuming. The flow might go something like this:

Interviewer: Design TikTok.

Candidate: I'm not very familiar with the platform. Would you be able to give me a high-level overview of what we are looking for?

Interviewer: Sure. TikTok is a mobile app for video sharing between users. Basically, you can upload a video to TikTok, and you can view a feed of videos. You can follow other users and perform basic actions over videos, such as "like," "favorite," and "comment."

Note:

If this is still not clear to you, or you think your interviewer is still withholding functional requirements from you, by all means go ahead and ask more questions. In this example, it seems like the interviewer has given us a pretty good overview, so we can move into our steps for functional requirements.

1. Identify the main objects and their relations

It appears there will be primarily two objects of interest: accounts and videos. What are their relations?

- Account -> Video
    - Can post
    - Can like
    - Can comment
- Account -> Account
    - Can follow

2. What information do these objects hold?

Check with your interviewer, but we can have some basic information like:

- Account: username, description

- Video: description

Are they mutable? Good question for your interviewer.

Candidate: Can videos be changed after uploading them?

Interviewer: No, let's assume that once a video is uploaded it will stay immutable.

3. Think about the access patterns

Remember, we are looking for statements of the form "Given object X, return all related objects Y." Check with your interviewer to learn what different access patterns are needed here. Example:

- Given a user, get all videos they've posted.
- Given a user, get their feed (videos posted by people they follow).
- Given a video, get likes/comments.

Then add the writes:

- Post a video.
- Follow an account.
- Like/comment on a video.

Example 2: Design a code deployment system aimed for developers at a company. They should be able to tag a release, and our system will package it and deploy it to some servers.

Let's start by identifying the main objects. In this case it looks like the only thing the service will need to store is code artifacts. What properties might we store? Feel free to discuss with your interviewer. For the sake of this example, let's say we want to store:

Artifact: (product name, version, commit hash)

You might also ask: “Can artifacts be mutated after being published?” Your interviewer might say the artifacts will be immutable (as in “not able to be mutated after publishing).

What are the access patterns? In this case we just have one object, so it’s probably going to be pretty straightforward:}

Trigger a release: publishes a code artifact and deploys it to all servers.

This is actually a pretty good example of how some problems might be less intensive on the requirements side than others. The definition the interviewer gave us was pretty clear. Therefore, we can extract requirements with just a few questions.

Extra points: Some interviewers might also be interested in evaluating your user-orientation. Can you think of how a developer might want to use this system beyond just publishing a release?

For example, you might want to ask about the possibility to recall a release. Do they want a page to see all releases and the deployment status? Should we alert them about failures?

This is what we mean when we say that it can be as easy or as hard as you want to make it. Try to get a feel for what the interviewer is looking for, and make sure to check back with them and ensure you both are on the same page.

1.2 Non-Functional Requirements

Once functional requirements have been laid out, you should move onto non-functional requirements (NFRs). These are quality attributes that specify how the system should perform a certain function.

The most common non-functional requirements you should consider in a system design interview are:

- Performance

- Availability
- Security

Note:

You may be wondering why we chose “security” and not “consistency.” We think there are already enough resources out there that will tell you about consistency in NFRs. Again, this is our opinionated approach, and after much debate we decided this would be the most valuable way to teach it. As always in system design interviews: your interviewer cares less about your decisions, and more about whether you are able to talk about the trade-offs (positives and negatives) of your decisions. Therefore, if you wanted to, there are many NFRs that could make a shortlist.

NFRs will strongly influence our design. They define what we should be optimizing for. Bear in mind that you cannot optimize for everything, and you should not overcomplicate your solution. This is a game of trade-offs.

Rule of thumb

Non-functional requirements can feel like platitudes. Who doesn’t want every system they design to be redundant, scalable, available, consistent, and so on and so forth?

But it’s a trap to say, “Non-functional requirements are always the same.”

Good candidates can view non-functional requirements mainly as opportunities to relax one specific requirement, such as “We don’t need to focus on [Insert requirement, such as “consistency”] as much in this case because [Insert reason, such as “it’s okay in this scenario of TikTok if some users get access to certain videos later than the rest of our users”].”

If NFRs are over-specified, the solution may be too expensive to be practical; if they are under-specified, the system will not be suitable for its intended purpose. Use your common sense, and ask the right questions to land on a set of non-functional requirements that make sense for the system you are designing.

Let's explore each of the quality attributes that make up non-functional requirements and see some examples of when to optimize for them.

Performance

Performance is pretty straightforward. It's the system's ability to respond quickly to user requests. While speed is always welcome, it might not be the right thing to optimize for in every system. Better performance may come at the cost of consistency or just an overall more complex solution.

So when does it make sense to optimize for performance?

Rule of thumb

It makes the most sense when we have synchronous user-facing workflows. That is, the user is expecting an immediate response from the system. In addition, we want to optimize for the synchronous workflows that are accessed the most frequently.

Take a look at the access patterns you identified in your functional requirements. Which ones are user-facing, expected to happen synchronously, and accessed frequently? Let's use our Twitter example: Is there any access pattern that we might want to optimize for performance?

The user feed feature stands out as a good candidate for a performant access pattern. It's synchronous, meaning that users expect an immediate response, and it's likely to be a hot path for users because it's the landing page of the app.

Availability

Availability refers to how much downtime the service can tolerate. Just like with performance, we might not always want to optimize for availability. A good question to guide this decision is: What's the cost of downtime? This is as easy as it sounds. If taking downtime will result in financial losses or correctness issues, we might want to put some thought into making the system highly available.

In the case of Twitter, the need for high availability is pretty obvious. The system should ideally not take any downtime. We measure availability by the percentage of the time the system is up and running. A common goal is to aim for five nines, i.e., 99.999% availability—that's less than 6 minutes of downtime a year.

Anecdote from an interviewer

Don't make your interview harder than it has to be.

In an actual interview, if a candidate says, "Do we want 4 nines? 5 nines? 6 nines?" My first follow-up question will be, "What's going to change in your system?" And they don't know what to answer (obviously) because the system they'll design won't change at all whether it's 4 nines or 5 nines or 6 nines.

So what I mean is that talking about availability is good, but talking about "we want 6 nines" in an actual interview can be a signal that this person is behaving like an imposter. This will cause the interviewer to ask harsher follow-up questions than if the candidate hadn't said anything about the "number of nines" in the first place!

An example of a problem where we might be fine with taking a hit on availability is one where consistency is very important. Think, for example, about a banking system. One of the most important mandates of the system would be consistency. Operations need to be transactional. In this case, it might be acceptable if our system is unavailable/stale for small periods of time, as long as it is consistent. However, with Twitter, we'd rather have it be inconsistent than unavailable. When in doubt

about what to prioritize, ask your interviewer whether consistency is preferred over availability.

Question

What's an example of a system you can think of where high availability might not be essential?

Security

Security can be tricky. No one wants to design an insecure system, and there's a baseline level of security that can be expected of any modern architecture (HTTPS, OAuth, password/data encryption). But that's not what we are trying to figure out in this step.

We want to learn if there's some workflow that might require a special design to account for security. For example, imagine you were designing LeetCode, an online judge for coding questions. One security constraint that would come to mind is that user-submitted code should be run in isolation. User submissions should run in some sort of sandbox where they get limited resources and are guaranteed not to affect or see other submissions.

Tip

Whenever there is user-generated code execution involved (aka low trust code), running it in isolation should be a non-functional security requirement.

Remember: Non-Functional Requirements

Consider the three main non-functional requirements: performance, availability, and security.

1. Performance: Which access patterns, if any, require good performance?

2. Availability: What's the cost of downtime for this system?
3. Security: Is there any workflow that requires special security considerations (e.g., code execution)?

Example of Non-Functional Requirement gathering

Example 1: Design TikTok

Discuss with your interviewer, but you probably want to optimize for performance and availability because of reasons that are similar to our Twitter example.

Example 2: Design a code deployment system aimed for developers at a company. They should be able to tag a release, and our system will package it and deploy it to some servers.

Performance: After asking your interviewer, you learn that code artifacts should be deployed within 1 hour after being published.

Availability: The system should be highly available as it would block the company from rolling out new releases. Some downtime here and there might be tolerated (especially if during weekends).

Security: Since there's code involved, it might be worth asking if the code is assumed to be trusted or not. In this case, since it's a code deployment service for a company, let's say that we can assume the code will be high-trust, so there are no special security constraints.

Step 2: Data Types, API and Scale

Inputs	- Functional and non-functional requirements. - Problem statement given by your interviewer.
Outputs	- List of Data Types we need to store. - Access patterns for these data types. - Scale of the data and requests the system needs to serve.

We've gathered functional and non-functional requirements. At this point we understand what the system is supposed to do as a black box. It's now time to take our first steps toward designing it.

However, you should not begin drawing boxes and discussing implementation right away. There's a bit of pre-work needed before we can start thinking about a concrete design. We need to answer the following three questions:

1. What data types does the system need to store?
2. What does the API look like?
3. What volume of requests do we need to support?

These can be answered pretty quickly from your requirements. In fact, you can probably answer these in just a few minutes. Let's walk through how we might answer each of these questions for our Twitter example:

2.1 What data types does the system need to store?

Think about the objects the system needs to hold and their data type. There are largely two types of data we might need to store:

- **Structured data.** Think business objects, like accounts, tweets, likes.
- **Media and blobs.** Think images, videos, or any type of large binary data such as TAR or ZIP files.

For our Twitter example, we will store:

- **Structured data**
    - Accounts
    - Tweets
- **Media**
    - Images or videos in Tweets

2.2 What does the API look like?

Rule of thumb

More than 90% of the time, users will interact with the system through HTTPS, and as such we encourage you to think about the API in terms of HTTPS requests.

Footnote: If you are curious about the rare cases where one might want to use a different protocol (like WebSockets), refer to future iterations of this guide (release dates TBD) where we will dive into these exceptions. But even in these rare cases, it helps to start thinking about the API in terms of HTTPS requests.

Look at the access patterns you defined in the functional requirements to write your API. For example:

```
getTweets:
        GET /{accountId}/tweets?nextPageToken={token}
        returns: Paged list of tweets sorted by creation time desc.

getFeed:
        GET /{accountId}/feed?nextPageToken={token}
        returns: Paged list of tweets for the given user's feed.

putTweet:
        PUT /{accountId}/tweets
        body: content of the tweet.

retweet:
        POST /{accountId}/retweet
        body: id of the tweet that is retweeted.
```

2.3 What volume of requests do we need to support?

Finally, we should consider the volume of requests that the service needs to serve, as that will influence our design.

As a starting point, I recommend that you ask yourself whether this system is read-heavy or write-heavy. Go back to your API and figure out which endpoints are likely to be called more frequently. Do you think our Twitter API would be read-heavy or write-heavy? You guessed it: it's probably read-heavy. Users will be calling getFeed and getTweets far more often than they would call putTweet or retweet.

Normally, it's enough to think about how people will be using the system and apply some common sense to figure out which endpoints get called the most. In case this is not immediately obvious to you (perhaps you are not familiar with these kinds of systems), it's totally fine to just ask your interviewer. For example: "What's the behavior of a typical user using this app?" Or be even more direct: "What does the distribution of requests look like?"

Let's pause for a moment to reassess where we stand. We know how our API looks and which endpoints are going to be hit more frequently (read-heavy vs. write-heavy). This alone should be enough for us to start sketching out a design. However, interviewers might be interested in seeing some back-of-the-envelope math in terms of concretely identifying how many requests we'll be serving.

You should check with your interviewer to see if they want to see you do some math or if they'd rather go into design.

Tip

Tell your interviewer: "It seems like we've identified the main requirements, we have an API in place, and we know how the distribution of requests looks. If I were designing this system for real, I'd probably want to do some back-of-the-envelope

math to estimate the number of requests and average volume of data we need to store. Do you want me to do the math or do you want me to skip it?"

If they agree, you should assign these requests some ballpark numbers in terms of writes/minute and reads/minute. It really does not matter at all if you are right or wrong. In fact, you'll most likely be wrong. Believe me, your interviewer doesn't care. We just want to agree on some numbers so we can do some back-of-the-envelope math.

Outlaw Idea

We've seen online resources that spend so much time showing you how to calculate these numbers down to byte precision. For example: "Remember there are this many bytes in a GB, so if you have 7 GB then you have this many bytes…" But for 90% of problems… who cares? Go for some ballpark numbers, and make the math easy. You know you are probably wrong anyway, and it's irrelevant as long as you are in the ballpark and have something to work with.

Use some nice round numbers to make the math easy. In fact, exclusively use powers of ten to make your life even easier. How far off can you be from the closest power of ten? When in doubt, just guess higher—it's called margin of safety. For our Twitter example we can go for these numbers:

```
Reads/minute: 100k3

Writes/minute: 1k
```

Finally, what's the volume of data we need to store? Go back to your "data types" (section 2.1) and think about how big these can get and multiply that by the number of writes/minute to get how much data you need to store per minute.

Structured data (tweets, accounts): 100 KB each

Media (images, videos): 10MB each

Average size of a write: 1MB (just a guess!)

Again, it does not matter at all if you get these numbers right as long as you are in the ballpark. Please don't spend too much time on this. Just use powers of ten, and when in doubt, pick the higher number. This makes our final math super easy:

1k writes/minute with an average size of 1MB each = 1k * 1MB = 1000MB = 1GB/m

So we'll need to store around a gigabyte of data per minute.

Remember: Data Types, Scale, and Access patterns

Once you know your requirements, it's time to get specific.

1. **Data Types:** Start by identifying the main business objects that you need to store.
2. **API:** How are these going to be accessed?
3. **Scale:** Is the system read-heavy or write-heavy?

Example: Design a code deployment system aimed for developers at a company. They should be able to tag a release, and our system will package it and deploy it to some servers.

Artifact: (product name, version, commit hash)
Trigger a release: publishes a code artifact and deploys it to all servers.

Performance: 1 hour from release triggered to servers.
Availability: Can tolerate some downtime: 99.9% availability.

Data Types:

Ask your interviewer about the type of these artifacts that we are building.

- Code Artifacts. Type: blobs (ZIP, TAR, bz2, etc.)

API

putRelease:

POST release/{productId}/{commitId}

returns: deploymentId # Id to check the status of the deployment

getDeploymentStatus:

GET deployment/{deploymentId}
returns: status # PENDING, DEPLOYED

Scale

Ask your interviewer about the scale of these deployments. Here are examples of some good questions to ask (and an interviewer's possible replies):

Candidate: What's the average size of the artifacts that we need to package?

Interviewer: We'll say 1 to 10GB.

Candidate: How many artifacts do we expect to deploy daily?

Interviewer: In the order of thousands.

Candidate: How many machines do we need to deploy to?

Interviewer: Around hundreds.

Step 3: Design

Inputs	- Functional and non-functional requirements. - Problem statement given by your interviewer. - List of Data Types we need to store. - Access patterns for these data types (API). - Scale of the data and requests the system needs to serve.
Outputs	- Data storage. - Microservices.

The time has come. We've got all the information we need to start drawing boxes and calling this a "system." Yay!

There are several reasons that we spent considerable time in steps 1 and 2. Too often people dive straight into design and fail in spectacular ways. It's easy to make that mistake—isn't this interview called "system design" after all? No one told these candidates that good design is 70%+ requirements and planning.

In fact, we can go as far as saying that if you've executed the last two steps correctly, design should be pretty systematic. This is because system design questions are usually open ended and don't have one single correct answer. Let's use this to our advantage!

Once we know our use cases and what to optimize for, it comes down to knowing a few rules of thumb. Want speed? Use a cache. Want availability? Put in some redundancy. It's really that simple. That's the beauty of systems design. It can be as simple or as complicated as we want to make it.

At the risk of oversimplifying, we suggest that you start small. Just follow some rules of thumb depending on what you identified in steps 1 and 2. We can guarantee you that you'll get a decent design. Then you can use the remaining time to iterate on it. Design is an iterative process.

Tip (Tell your interviewer)

"I’m going to start drawing some boxes. I'm just thinking out loud for now, so don't hold me to any of this. We can come back to it later."

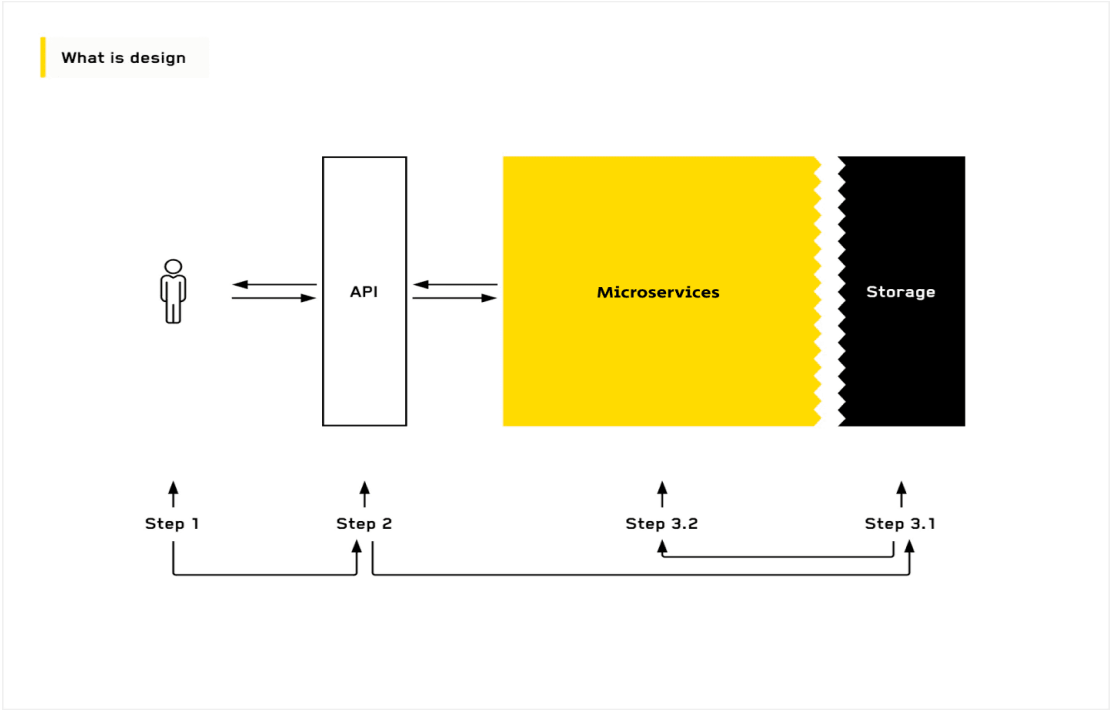
This is basically giving you a free pass to flush your brain and be wrong. Which is exactly what you want when there’s a clean slate in front of you. Again, design is an iterative process. Expecting that you’ll go from clean slate to perfect design in one go is just… foolish.

So what is “design”?

We should first align on our outputs. Design simply means two components:

1. **Data storage.** We already know from previous steps “what” we are storing. Now the question is where are we storing it?
2. **Microservices.** How do we store our data? How do we retrieve it to the API? Think of these as the middlemen between storage and the API.

We know the **what** (steps 1 and 2), so now we focus on the **where** and the **how**. We will start with designing the data storage layer first and then think about the microservices that access this data.



On the far left we have our users and the API, and on the far right we have storage. Microservices are the connective tissue between these. As such, it pays off to think about them last. Otherwise, how will we know what we need to connect?

3.1 Data storage

Blob storage

Let's get some of the more obvious components out of the way first. Did you identify any type of media or blobs in step 2.1? If so, these are great candidates to store in blob storage. A blob (Binary Large Object) is basically just binary data. We store and retrieve these as a single item. For example, ZIP files or other binaries.

Some popular blob stores are Amazon S3 and Azure Blob storage. In general, you don't need to worry too much about the specific brand you'd be using. Just tell your interviewer that these images/blobs you identified are good candidates to store in some blob storage, and then draw a "blob" box.

Going back to our Twitter example, we'll want to store media from tweets in some kind of blob storage.

Rule of thumb

Say the generic name of the component, not the brand name. Unless you are very familiar with a specific brand (like S3), don't say the specific brand. Instead, say "some kind of blob storage." Because if you say, "we should use S3 here," the next question out of your interviewer's mouth will be, "why not Azure blob instead of S3?"

There's a chance you might want to couple the blob storage with a CDN, but that's something we'll look into in step 3.2. This step is all about identifying how to store content, not how to distribute it.

Database

There are a few considerations for this step:

1. Relational vs. Non-Relational
2. Entities to store

3.1.1 Relational vs. Non-Relational

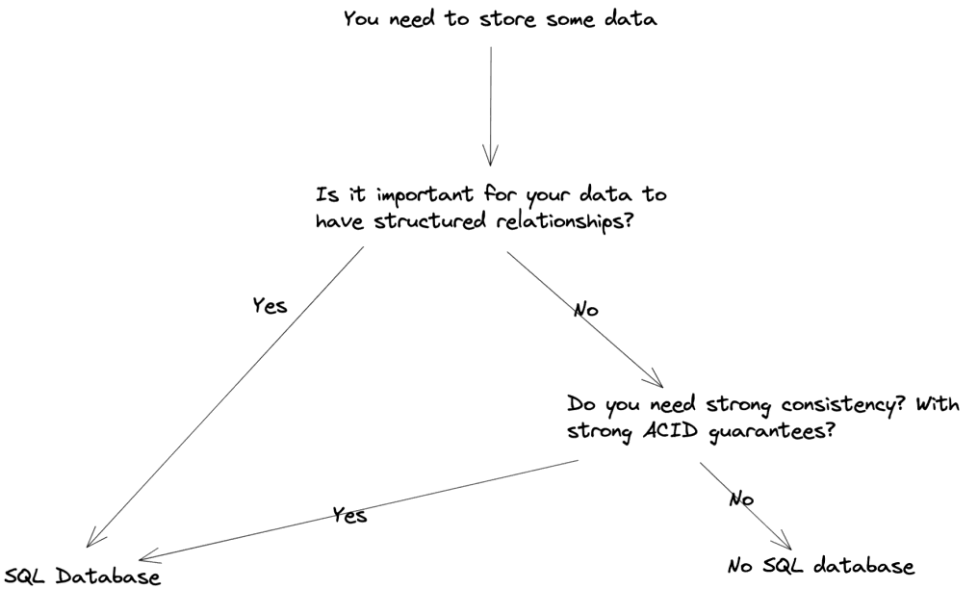
Relational vs. Non-Relational, sometimes referred to as SQL vs. NoSQL, is one of the foundational decisions of database design. There are many trade-offs involved when it comes to picking one or the other. In many interview questions, an argument can be made for any choice you make. It's important that you don't succumb to paralysis through over-analysis. Just pick one, and make your rationale clear for why you chose it. Score extra brownie points if you include a drawback of making the pick you made.

Remember

There's no right or wrong answer—it's all about how to justify your picks.

Don't oversell a solution. Every solution has positive and negative aspects and needs to be approached with a sense of realism. If you're being unrealistic, you probably won't change your mind (even when it benefits you to change your mind!). For example, sometimes the interviewer will give you an out by asking some follow-up questions, giving you a chance to see your own mistake and change your mind. But if you're too fixated on being right, you'll miss the opportunity.

Therefore, we're giving you two very powerful tools: (1) A rule of thumb to pick Relational (SQL) vs. Non-Relational (NoSQL), and (2) A list of trade-offs that you should mention to your interviewer after stating your decision.



At the risk of oversimplifying the decision, we can assert with confidence that if you don't fall into any of the above cases, you are probably fine picking either SQL or NoSQL. However, many interesting system design questions require strong consistency, unstructured data, or both. Actually, using both is also somewhat common, and something we'll touch on.

Tip (Tell your interviewer)

If you picked relational:

"Although I think a relational database better fits this requirement, we should also be mindful of the downsides. For example, our database will have a more rigid structure and schema, so it might be harder for us to incorporate changes. We'll also need to scale up vertically, meaning that as we get more load we'll upscale existing servers rather than dividing the work over more servers."

If you picked non-relational:

"Although I think a non-relational database better fits this requirement, we should also be mindful of the downsides. We'll be able to scale horizontally at the cost of not having ACID guarantees. I'm assuming there will be no need for strong consistency in the future."

So which one is a better fit for our Twitter example? Let's run our requirements through these questions:

Do we need strong consistency?

We probably don't. It's fine if after publishing a tweet, some users can see it before others. Same for likes and followers. We don't need to treat these as atomic, consistent operations. Eventual consistency works fine for our requirements.

Do we have large volumes of unstructured data?

Not necessarily. Our entities, tweets and accounts, will have some well-defined static fields that are unlikely to change in meaningful ways.

Given the answer to these two questions is "no," this is yet another example where we are good with either choice. It comes down to how we justify it. In fact, Twitter started using MySQL and then moved to NoSQL seeking better scalability and availability.

If you ask us, we'd probably go for NoSQL and justify it as: (1) It doesn't look like we need strong consistency, and (2) NoSQL will scale horizontally and likely have better availability.

Examples

1. Design a banking system.

This is a textbook example of strong consistency. Transactions in a banking system need ACID guarantees. As such, we are probably better off picking a relational database that can give us this strong consistency.

2. Design a system to help doctors diagnose potential illnesses given symptoms.

Let's say this is mainly a querying system. Doctors enter a list of symptoms and get back potential illnesses and treatments. The data we will be storing is unstructured in nature, and it will likely be an ever increasing database as we add more illnesses, symptoms, and diagnoses. In this example, it might be wise to pick a non-relational

database where we can store large volumes of unstructured data, scale horizontally, and be fine with just eventual consistency.

3. Design Amazon.

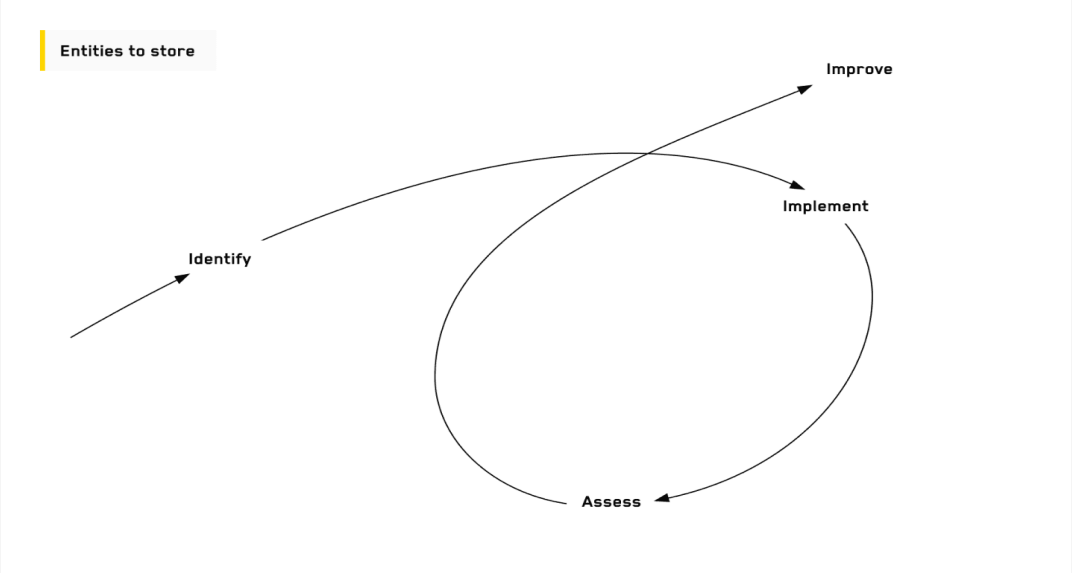
Amazon is a good example of a system where we might want to use both of these. We'd want to have consistency for product transactions, while being flexible about the data in our product catalog. It wouldn't be crazy to suggest using a relational database to keep track of purchases and stock, while using a non-relational database for the product catalog.

Entities to store

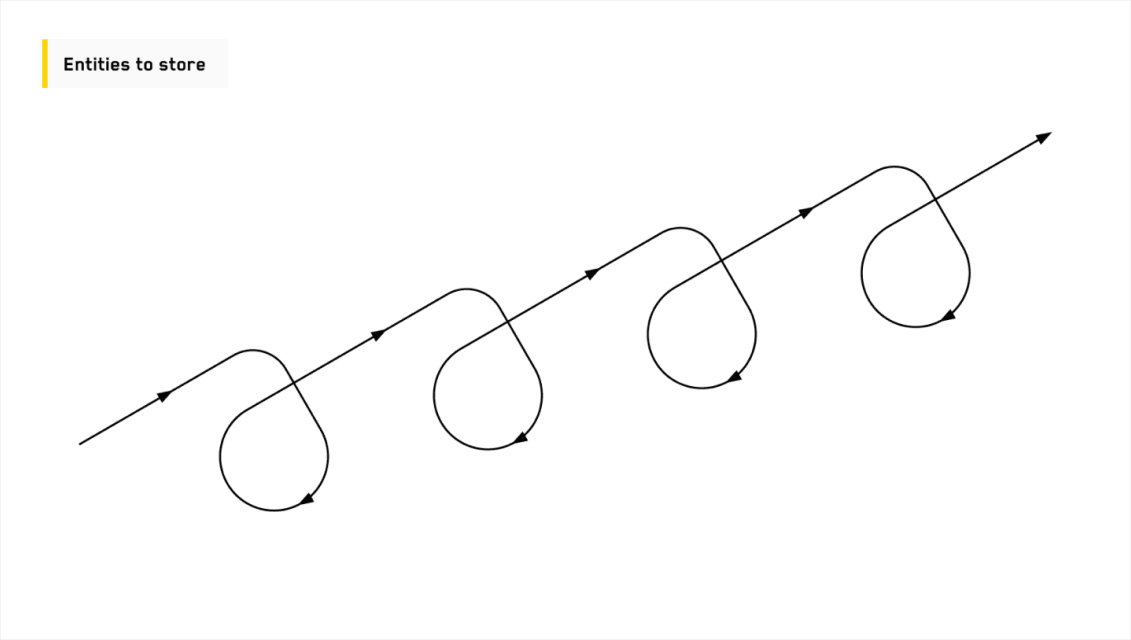
It's time to look at the data and access patterns we defined in step 2 and design our database schema. This will look like a list of tables/documents, and a description of the queries that you'll use to access them.

A good starting point is to sketch out a table for each entity you identified, and then go over the access patterns. Think about how they will be fulfilled. Then you can adapt your tables to better fit these access patterns. **Repeat after me: design is an iterative process.**

Think about these iterations as small cycles where you identify the requirement, implement a solution, assess its limitation, and then improve it.



Rinse and repeat until you have a solution that accommodates your requirements from step 1. When you zoom out, you are just going through several of these cycles:



Let’s see what this might look like through the lens of our Twitter example.

We identified two entities in step 2.1: (1) Accounts, and (2) Tweets. Therefore, we’ll start out nice and simple with two tables:

- Accounts: id, name, surname.
- Tweets: id, content, author_id, media_url.

Note that each of these has an id. This id is immutable and uniquely identifies each object so that we can easily change any of the object’s metadata without the need to update other records. Also, the media_url would point to the address of the blob storage bucket containing the tweet’s media, if any.

Now look at the access patterns and make some adjustments. We identified two access patterns in step 2:

- getTweets gets all the tweets for a given user.
- getFeed gets the feed for a given user.

getTweets should be pretty straightforward given the tweets table. We’d just need to select all tweets with a given author_id. Databases usually support the concept of an index, which provides faster access to entities given a property (called the index).

Indexing tweets by their author seems like a sensible choice to fulfill this access pattern.

Tip (Tell your interviewer)

Be mindful of any “get all” access patterns. These usually need to be guarded by paging. You don’t want a single endpoint returning the entire tweet history of an account. Depending on the account, that might be a very expensive query, and degrade user experience. Usually these will be behind logic that pages the response. That’s why Twitter will load pages of tweets, even if it seems like an “infinite scroll” in the UI.

Now onto getFeed. Let’s define feed to be a list of tweets of all the accounts the given account follows, sorted chronologically. There’s one thing we are missing here already: the information about who follows whom. Let’s say we add that relation in some new table:

- Followers: account_id, follower_id

We can again have indexes to speed up certain access patterns, such as getting all followers for a given account_id. This is totally fine. You can add tables/indexes as you realize they are needed. Remember: Identify → Implement → Assess → Improve.

Computing the feed for a given user would require us to get all the accounts they follow and then get all their recent tweets. We have an implementation—time to assess it. Is this solution acceptable for our requirements?

Given the fact that we identified the system to be read-heavy and getFeed is expected to be called quite frequently, the computation can become prohibitive. Consider that each account might follow thousands of users, and those might have hundreds of tweets. Computing the feed seems like quite a compute-intensive process.

However, don't fall into the common pitfall of prematurely optimizing your system. Your interviewer might not even care about this problem. After you assess the limitations of your solution, check back with your interviewer before continuing to improve the solution.

Tip (Tell your interviewer)

"Although this would work from a functional perspective, I'm afraid it might not fulfill our non-functional requirements. Concretely speaking, we've identified the system to be read-heavy, and this approach would be prone to a slow read performance. I assume we'd like to optimize it—what do you think?"

Assess your current solution, provide your opinion, and then ask your interviewer for their thoughts. This is the best way to iterate on system design. You don't want to rely solely on your interviewer without expressing your thoughts because it may convey a lack of criticality/independence.

You also don't want to move forward without any input from your interviewer because it may be perceived as poor collaboration. We find that **stating your rationale followed by a subtle "what do you think?" or "let me know if you think I'm approaching this the wrong way" is the perfect balance between being independent but also collaborative.**

For the sake of learning, let's say that our interviewer agrees with us and wants to move forward with optimizing this. Whenever you are looking to optimize runtime, trading it off with memory should be your first go-to.

Rule of thumb

When looking to optimize performance for a read-heavy access pattern that requires several queries, consider storing the final result in a separate table and keeping it up to date.

In this example, we might want to store the user feeds in a table and keep that up to date as new tweets come up. That way, we have an instant mapping from user to its

feed, making getFeed fast at the cost of using more memory and the added complexity of having to maintain feeds up to date.

You can also get creative as well. Do you need to keep the feeds for all users up to date? Maybe we can prioritize users who log in frequently. For users who rarely log into the app, we can compute the feed on-demand. These kinds of ideas are worth mentioning. They don't really influence the design that much, but they show thoughtfulness around usage patterns.

3.2 Microservices

Once we have our storage layer somewhat defined, the last step is connecting our API to the storage layer. There are a few decisions that often arise at this stage:

- Caching
- Load balancing
- Queuing systems

Caching

Ask yourself: Are there any access patterns that would benefit from caching the results in-memory? Candidates sometimes add caching to their solution just because. This is often a mistake.

Warning

Not all systems designed in system design interviews require caching.

Remember that every decision you make has some trade-off. There's no such thing as a free lunch in system design. Therefore, we urge you to consider the downsides of your design decisions and mention them during the interview.

Rule of thumb

Consider using caching when all three of these are true:

- Computing the result is costly
- Once computed, the result tends to not change very often (or at all)
- The objects we are caching are read often

A common technology used when caching is needed is Redis. If you are not familiar with it, all you need to know is that it is a way for you to cache parts of your database in memory such that it's faster to access them.

Remember

If you haven't used Redis, don't say, "Let's use Redis here" in the interview; instead, say "Let's add a cache here." Brand names are a riskier bet than generic names of components unless you have thorough experience with a specific brand, because the first follow-up question will likely be, "Why Redis and not Memcached?"

What are some of the downsides of caching, you may ask? To begin with, it introduces two replicas of the same data. Even though our source of truth remains to be persistent storage (our database), it is now possible to get the result from the cache as well. This might introduce inconsistencies if the cache is out of date from the source of truth. This is why it's wiser to cache objects that don't usually change too often. It's also costly to maintain and adds complexity to the system. Now every time we want to update our logic, we'll need to consider the caching layer as well.

Load balancing

Load balancing helps us scale horizontally and maintain high availability. While horizontal scaling is desired in most systems, it is again advisable to consider whether it is strictly necessary given the requirements you've identified.

Load balancing is easier when our API servers are stateless because requests can be routed to any node. In our Twitter example, we'd probably want to load balance incoming traffic into several replicas of our stateless API servers which will just hold the logic to access the database.

As mentioned, this will give us two key benefits:

1. **Horizontal scaling.** We can add more API servers to handle more load.
2. **High Availability.** Whenever we need to upgrade or restart our API servers, we can perform a rolling restart. This means that only one node would go down at a time, while others continue to serve requests. That's how you normally are able to upgrade logic in these systems without taking downtime.

There are different strategies for deciding how to balance load across a set of servers, but most of the time you'll be dealing with round robin.

End-to-end example of the process

Design interviewing.io: A website where people can schedule anonymous technical interviews with other engineers.

Step 1: Requirements

The following simulates a conversation you could have with your interviewer and a final result of the requirements you agree on. While the areas the interviewer wants to focus on might vary, this should give you a good overview of what to consider and how to ask relevant questions.

Let's start with business objects first.

Candidate: I identify two business objects right off the bat: users and interviews. Let me start with interviews—do we need to save the video recording of the interview?

Interviewer: Yes. There is also a showcase feature, where if both parties agree, the interview will be displayed in our "showcase" for others to see.

Candidate: I see. Are there any other properties about interviews that I should be aware of?

Interviewer: Well, generally, interviews are done in one programming language. We want to keep track of that so that users can then filter to only "Java" interviews, for example.

Candidate: Got it. And what about matching interviewers and interviewees? How is that done?

Interviewer: Users set up an availability where they are able to interview or be interviewed, and then they get matched with their counterpart.

So far it seems like we need to track information about:

- User (name, surname, pseudonym, availability)
- Interview (interviewer, interviewee, video recording, programming language)
- Booking (time, interviewer, interviewee)

Looks like we've got a good idea of the business objects involved in this system and a rough sense of the properties they'll hold. Next, we should chat about different access patterns and agree on the functional requirements. Remember, you'll want to think about the cross product of these entities to come up with ideas of possible access patterns that relate these objects.

- **Given a user, get all of the interviews they took part in.**
- **See showcased interviews.**
- **Set availability.**
- **Book interview.**
- **Join interview.**

Non-functional requirements

Availability seems like the obvious thing to optimize for since the platform needs to be up for candidates to be able to interview.

There's nothing notable in terms of performance here that warrants taking a note. Most likely, the thing we'll care the most about will be audio quality during the interview. There's also writing and running code during the interview.

We probably want to note **good audio quality** during the interview as a non-functional requirement. This could as well fall into availability. If the connection is flaky, that's pretty much downtime.

Finally, for code execution, we'd probably want to execute the code in isolation. Candidate submissions are low-trust and should have no side effects on the system. For example, you cannot DOS the system with code submissions.

Step 2: Data Types, API and Scale

Data Types:

```
User:
    - id
    - name
    - surname
    - pseudonym
    - availability
    (Structured data)

Interview:
    - id
    - interviewer_id
    - interviewee_id
    - video_recording_id
    - programming_lang
    (Structured data)

Booking:
    - id
    - interviewer_id
    - interviewee_id
    - time
    (Structured data)

Recording:
    - video (media)
```

REST API:

```
putAvailability:
    POST /users/{userId}/availability

getInterviews:
```

```
    GET /users/{userId}/interviews
    returns: interview_id

getInterview:
    GET /interviews/{interview_id}
    returns: streaming interview

getShowcase:
    GET /showcase
    returns: list of interview ids and metadata.

bookInterview:
    PUT /users/{userId}/bookings
    returns: booking_id

getBookings:
    GET /users/{userId}/bookings
    returns: list of booking_ids

joinInterview:
    POST /interview/{booking_id}
    returns: link to coder pad.
```

Rule of thumb

In an interview, the less you code you write, the more you seem like a senior engineer. And the opposite is true as well: The more code you write in a system design interview, the more you seem like you're below the senior level. Writing this much code for your API would probably be too much if you're aiming for senior or senior plus roles. But if you're a mid-level candidate trying to secure your mid-level position, this is the perfect amount of code to write.

Scale:

Candidate: How many interviews are we expecting?

Interviewer: In the order of thousands per day.

Candidate: How long does one interview last?

Interviewer: About an hour.

Candidate: What do we need to store as part of the recording?

Interviewer: Just the audio and coder pad.

Based on these items, if our interviewer wants us to do some back-of-the-envelope math, we could assume:

100MB per interview x 1000 interviews per day = ~100GB of data per day

Step 3: Design

Let's start with data— where shall we store it and how? If we go back to the data types, we've identified mainly two of them: interview recordings and metadata (users, interviews, bookings).

As for the interview recordings, since they are videos, a blob store would be a decent choice. We can have a table where each interview holds the link back to its recording. Furthermore, we can index them by things like programming language, allowing us to filter down by certain properties. One would imagine that any modern database should be capable of efficiently indexing the volume of records we are expecting (~1000 a day). In order to improve user experience, we would probably make the getShowcase endpoint a paged endpoint, and return paginated results for matching interviews.

When considering which database to use to store metadata, it's not immediately clear that any one technology would be a better fit. We'd probably go for a relational database like MySQL because schemas are quite well defined and entities are tightly related to each other. We'd also probably want to update bookings transactionally so we make sure we are not double booking interviewers. **Having said that, an argument can be made for pretty much any modern database.** We'd mention to the interviewer that our schemas might be a bit more rigid and we'll likely have to prefer vertical scaling.

Part 3: Outro

Part 3 was our breadth-first approach to teach you system design. We gave you a framework that covers a lot of ground, and it can be applied to any system design problem. This is valuable because you learned one systematic approach that you'll be able to use in many different cases.

Part 4 is a depth-first approach to teach you system design. It complements Part 3, yet it's very different in nature. By having different creators contribute to this guide, we avoid bias, and these differing opinions may even contradict each other sometimes, but that can be a great lesson in itself. We engaged in some heated debates about the differing views of Parts 3 and 4, but this reinforced the idea that there is no single "right answer" when it comes to system design. With that in mind, we offer you the best knowledge and insights from the smartest people we could find (even if they weren't always willing to agree with each other!).