Week 5?

need to see if we can make a project that they can work on

So far, we talked a lot about frontend and backend as seperate pieces. The frontend is what the user interacts with, and the backend is what handles logic, data, and communication behind the scenes.

This week, we are taking a step back from focusing on just one thing. Instead, we are going to look at the whole system and how all the pieces connect.

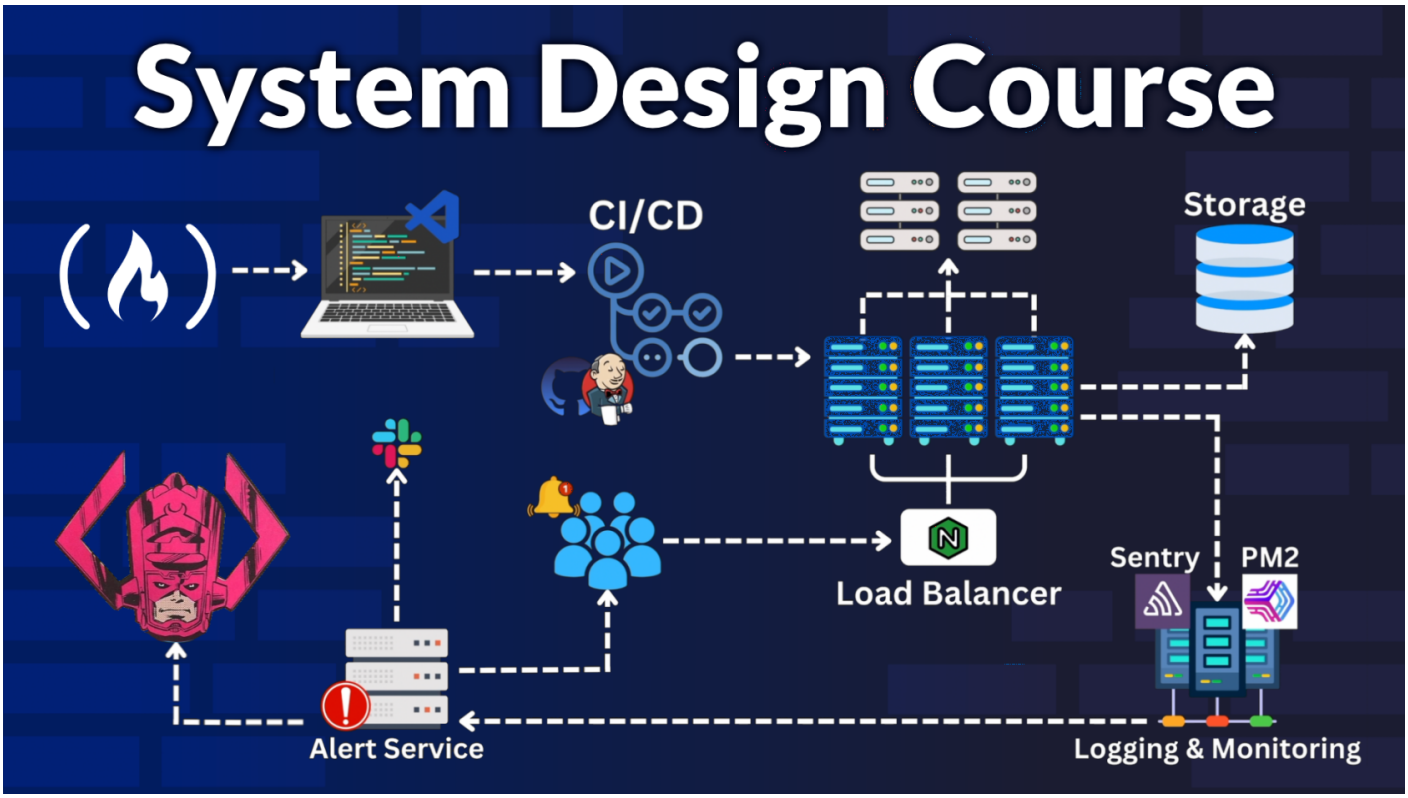
What is system design?

System design is just the process of thinking about how an application works as a whole. When we build an app, there are a bunch of questions beyond "How do I code this?"

You have to answer

1. What parts does this app need?
2. What does the data look like, and where will it go
3. How will the frontend interact with the backend
4. How do we deploy the app
5. What happens when tons of people use it

Answering these questions is pretty much what system design is. It isn't just for big companies or complicated apps. You have probably "done" system design if you've ever worked on a project, because anytime you have a frontend, backend, database, auth, file storage, or deployment, you implemented a system.



Thinking in System Design

The best way to think about system design is to break an app into pieces (and break each piece into smaller pieces if you can)

For most web apps, you usually need

1. Frontend
2. Backend
3. Database
4. Storage
5. Authentication
6. Hosting / Deployment

Not every project needs all of these, but these are the most common things most projects have.

For example, with DuckLink, they have

- A frontend where students browse events
- A backend that handles the requests
- A database that stores the event info
- Authentication so students can log in
- Hosting so that the app exists somewhere people can access

Once you think about apps like this, it becomes much easier to understand what you need to build

Usually, projects go through three primary stages during development: the local development stage, the deployment stage, and the scaling stage.

Local Development

This is where everything runs on your computer. You typically have

1. A frontend running locally (npm run dev)
2. A backend running locally (npm run dev)
3. A database running locally (Docker containers)

At this stage, you just want to make sure the app works properly

Deployment

Once your app runs properly locally, it needs to be hosted somewhere so people can access it (Either on the internet or an App Store). This means you need to think about

1. Where does the frontend live
2. Where does the backend live
3. Where does the database live
4. How can they all connect

Once a project gets deployed, system design matters a lot more because the app isn't just running in one place, its running like a system (system design).

Scaling

Let's say your app had tons of users showing up. Tons of problems will also start showing up. Things that were fine can become bottlenecks. Let's say

1. The backend might get too many requests
    1. Your site gets so much slower, especially if you rely on the backend to get data.
2. The database might slow down under load
    1. Similar to the backend getting too many requests, anything that requires retrieving information will take a while because your database is overloaded
3. File uploads might become expensive
    1. This can be applicable to the frontend and backend too, but let's say you hit the limit on your free plan for your file server. You will no longer be able to upload anything without paying, which, if scaled up to tons of users, can get expensive.
4. Users far away from the server have a slower experience
    1. Let's say you deployed everything to an American server. Users anywhere else in the world will have a slower time on your app because they have to interact with American servers to get everything. This takes a while, and while not as impactful as the other issues, it is still important to consider

This is where ideas like caching, CDNs, load balancing, and horizontal scaling matter. To further show this, let's take a look at an example

The YouTube Example

We asked this question during the Fall Semester recruitment cycle, and very few people could somewhat answer it, and only people who've had internships or tons of projects were able to somewhat answer it, so I figured this is one of the best examples we can do.

"How would you design YouTube?"

More specifically, though, we asked them, "What does your API look like?" How would you deploy this? How would you scale this?

However, for this, I'll walk you through the full thing so you guys can use this as an example if you build anything similar.

First, we need to define the APIs. You generally want to figure out your backend first, as it's harder to make changes to a backend if your frontend is already written. Let's say we are passing parameters from the frontend to the backend, and we need to change the backend; we also need to make changes to the frontend, which is annoying. Now, YouTube has a lot more than these four, but for the video streaming itself, we need a create video, delete video, update video, and get video. Create video takes the account that made the video, any metadata like title, description, thumbnail, and a UUID. A UUID stands for Universally Unique Identifier, and it's a 128-bit number that allows us to create the number before it touches the backend (reducing load) and makes it easier to merge databases as it's always unique (not stored as 1, 2, 3 in the db). Delete video takes the video UUID and the account to make sure whoever is trying to delete a video can actually delete the video. Update video would take similar parameters to create, but instead of creating a new video, you would just update the fields that need to be changed. The get video endpoint retrieves the video metadata that we stored before using the UUID that we provide. We can get the UUID from the frontend, which would request a list of videos from the backend and then map those results into React components. Get video would also return the video, either with a file or a playback URL, which I will explain later.

The frontend can look like whatever, so we don't need to focus on that in terms of system design, for now our working model in our heads can be three buttons, one for each api route, and a bunch of videos using the get video route. In a deployed app, we have our frontend connected to the backend, normally called a full-stack application. However, in this case, it isn't smart to store our video in our backend because the video needs to be streamed. Streaming a video from a database will slow down the database by a lot, and the streaming itself will be slow, so we need something else. For this, we need a file server to store our videos and our thumbnails. So in our mental model, the front end is connected to the backend, and the backend is connected to the file server.

When we store videos with the create video route, we store them in the file server and rename the video to the UUID we generated in the route. This allows us to identify which video corresponds to which entry in our database and retrieve it accordingly. One thing we can also do here to make uploads and downloads better is to have the backend create the UUID + db entry and return a

presigned upload or download URL. Then the frontend can upload or download the video to/from your file storage instead of pushing the whole file through the backend.

Right now, the typical flow of this app is that the user goes to the frontend and interacts with the create video button. They upload the video and fill in whatever they need to, and submit the request. The frontend then gives all the information to the backend, and the backend creates the UUID and stores everything except the video. The backend will return the presigned upload URL so the frontend can upload directly. Whenever the file needs to be streamed, the frontend sends a request to the backend, and the backend gives a presigned download URL to the frontend so it can stream properly, and the backend isn't involved in the actual streaming portion.

Now this is the first development stage done, and everything theoretically works locally, but how would we actually get to deploying this project? For this, we need hosting providers for each thing: the frontend, backend, and file server. For our frontend, we can use Vercel, Cloudflare Pages, AWS Amplify, or an AWS S3 bucket with CloudFront (I think our dev team uses this). For our backend, we would typically use PostgreSQL to store the information, so we can use Supabase, Neon, AWS RDS, or, if you don't want to use PostgreSQL, you can use whatever you can store stuff with (I suggest Convex if you don't want to mess with SQL). For our file server, you can use Vercel Blob, AWS S3, Cloudflare Stream, or Cloudflare R2.

Technically, this config can work. However, a question you may be asked is how would you scale this to a million users in a hypothetical scenario? In our model right now, it is in a straight line. One frontend instance, one backend instance, and one file server instance. We need to scale these horizontally, essentially adding more instances and creating a distributed network. This is where things can get tricky, so I'll try to explain it as simply as possible.

We essentially need multiple instances of each thing, however it is important that the instances for the backend and file server are connected. The frontend is easier to scale because it is mostly static and can be pushed through a CDN, but the backend, database, and file storage need a shared consistent state, otherwise some people will have videos that other people don't have access to.

First, let's address the frontend. We typically want to use a CDN for our multiple frontend instances. A Content Delivery Network allows us to store our frontend on tons of servers instead of just one server. The users will access the server closest to them, making sure they always have the fastest connection.

For our backend and file server, we will use something called a load balancer to distribute the requests effectively. The load balancer is pretty self-explanatory. If a cluster is running slower, the load balancer will reroute the request to a different cluster that has less load. To properly explain everything with examples, we will use Vercel for our frontend, Supabase for our backend, and an S3 bucket + CloudFront for our file server. Vercel has a CDN already, and it is pretty decent, so we don't have to worry too much about that.

For our backend, there is one thing we need to specify. Our backend must be stateless. Pretty much, no user session data (like which account they are on, what video they're watching, etc) can be stored on the database and instead, stored in a session using Redis and Tanstack Query (used to be called React Query). More specifically, Redis would be our shared cache layer on the backend, and React Query would help cache and manage requests on the frontend side. Setting up our backend this way allows our load balancer to switch clusters without the user noticing, and using Redis + React Query allows us to use our cache (locally and from the db) instead of calling to the db all the time, making our performance as fast as possible.

Supabase has a load balancer built into it called the Supavisor that manages connections to different instances (called the Pool Size) so we can use that alongside the load balancer for our clusters. One trick you can do is have a couple of instances of the database, just for writing data, and a lot more for reading data. Supabase calls this Read Replicas. Most people on an app like YouTube don't really post and instead watch, so that's a way you can lessen the load on your system. Supavisor manages the database connections while the Read Replicas distribute read-heavy traffic.

We use S3 and CloudFront for the file server because it takes the load of streaming away from our backend and frontend. If we didn't use the presigned URLs, the frontend would still play a role in streaming. Also, CloudFront is an example of a CDN, so you could also use this method (S3 + CloudFront) to deploy the frontend.

One very minor thing, but usually on a bigger scale, we would also transcode the uploaded video into multiple resolutions so users with different internet speeds can stream smoothly. Other than that, with this setup, this is how you would design YouTube

Tradeoffs

Ok, I know that was a lot, but this is only one answer to this question. With system design, there are different answers to the same question, but as long as you have an answer, that is fine. Because there are multiple answers, every answer has certain tradeoffs. For instance

- Database Type (SQL vs NoSQL)
- Local file storage or cloud storage
- Monolith vs Seperate services
- Simpler setup now vs more scalable setup later

In our version of the YouTube answer. We are

- Locked into the Vercel Edge Network (Their CDN)
- Locked into Supabase and Supabase utils for our database
  - If Supabase goes down, so does our app
- Setting up our file storage is more difficult to set up compared to our frontend and backend, but we have more control over it

A lot of system design is making reasonable choices based on the size and needs of a project.

Maybe there is a service we can use to store our files (uploadThing), so for an early prototype, we can use that instead of using an S3 bucket + CloudFront. Similarly, if we need control over our CDNs and database, we would move away from Vercel and Supabase.

??? Example

Walk me through how you would build ???

Revision #7
Created 8 April 2026 20:19:11 by Nishit Sharma
Updated 9 April 2026 04:46:40 by Nishit Sharma