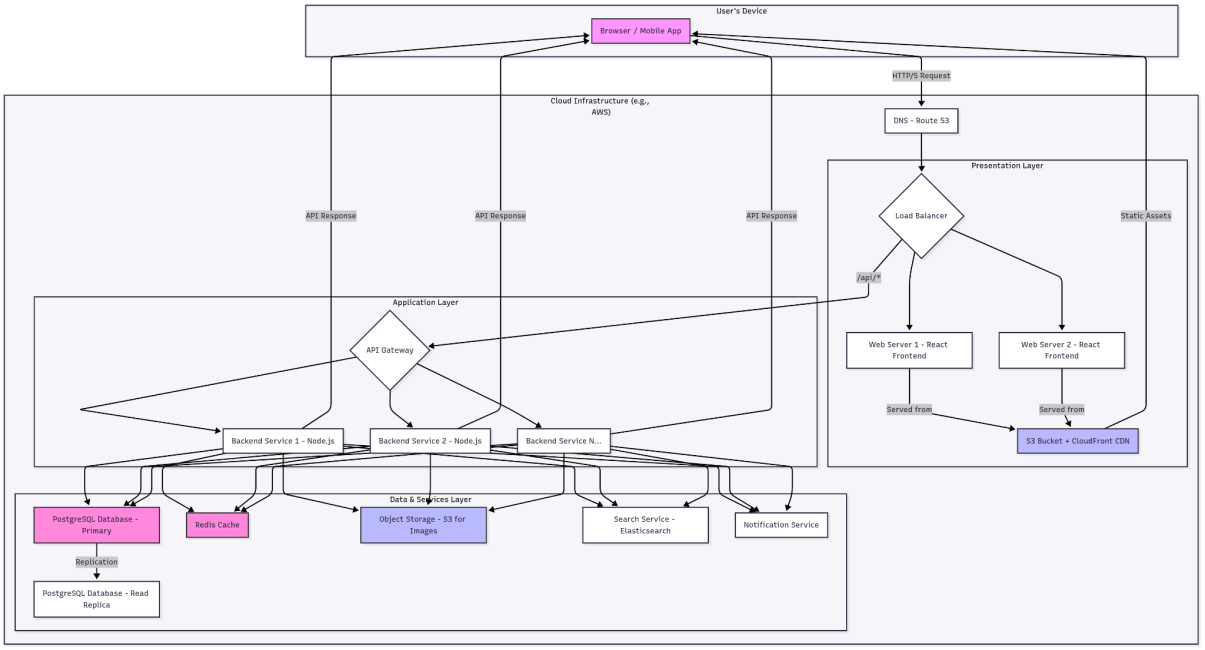
My Architecture Design for RecipeStudio

Here is my proposed application architecture for our new recipe-sharing platform, RecipeStudio. I've designed this with key requirements in mind, focusing on creating a system that is scalable, fast, and secure right from the start.

Deliverable 1: Visual Diagram

To help visualize how everything connects, I've created an architecture diagram. The diagram below (written in Mermaid code) shows the overall flow, from a user opening the app on their phone to how their requests are handled by our backend systems.



How it Works, in Plain English: Essentially, when a user visits RecipeStudio, our CDN (CloudFront) quickly sends them the main application interface. When they do something like post a recipe or leave a comment, that request goes through a Load Balancer, which passes it to one of our backend servers. Those servers then talk to our databases, cache, and image storage to get the job done before sending a response back to the user. (Devops course coming in handy :))

Deliverable 2: Architecture Documentation

Here's a more detailed breakdown of my design choices.

1. My Architecture Plan & Tech Stack

I've decided to go with what's known as a **"microservices-ready monolith."** This approach gives us the best of both worlds. We can build and launch with a single, unified backend application, which is much faster and simpler to start with. However, I've designed it so that as we grow, we can easily break off pieces into separate microservices (like a dedicated Notification service or User service) without a major rewrite.

The microservices approach is something I learned from Twitter when it was bought by Elon Musk. Elon announced on X that they did not need so many engineers to handle, and fired several employees. I later saw a post on X from some user telling others to not logout of their X accounts as the microservice that handled login was not working. I then tried to learn more about microservices.

The Tech Stack I've Chosen:

- **Frontend: React (with Next.js)**
    - **My Reasoning:** React is fantastic for building the kind of snappy, interactive user interface we want. I've specifically chosen Next.js because it gives us server-side rendering (SSR), which means our recipe pages will load super fast and be easily found by search engines like Google.
- **Backend: Node.js with Express.js**
    - **My Reasoning:** Using JavaScript on both the frontend and backend makes development smoother. Node.js is also incredibly efficient at handling many simultaneous users and tasks like database queries and file uploads, which will be common on our platform.
- **Databases:**
    - **Primary DB: PostgreSQL**
        - **My Reasoning:** It's a rock-solid, reliable database that enforces data consistency. I especially like its powerful features like JSONB support, which will be perfect for storing flexible data like ingredient lists and preparation steps. Also, I am biased towards relational databases.
    - **Caching: Redis**
        - **My Reasoning:** For things that need to be lightning-fast, like showing trending recipes or checking if a user is logged in, we can store that data in Redis. It's an in-memory cache that will dramatically speed up response times. I especially have great experience with redis after applying it to bookmystay (bookmystay.com.pk), it made the site super fast. If you had seen what it looked like 3 months ago with the loading times.
- **File Storage: AWS S3**
    - **My Reasoning:** We should never store user-uploaded images on our application servers—it's just not scalable. S3 is the industry standard for this. It's affordable, incredibly reliable, and lets us offload all the file storage heavy lifting.
- **Search: Elasticsearch**
    - **My Reasoning:** At first, basic database search will work. But to give our users a great search experience (like searching by multiple ingredients), we need a real search engine. Elasticsearch is powerful and will allow us to

deliver fast and relevant results as our recipe collection grows. I have never actually used it but have heard about it.

Data Flow Example (Posting a Recipe)

Let's walk through what happens when a user posts a recipe:

1. The user fills out the form in our React app and hits "Submit."
2. Our app securely uploads the recipe image directly to our S3 bucket. It then sends the text data (title, ingredients, etc.) along with the user's authentication token to our backend API.
3. The request first hits our API Gateway, which checks if the user is properly logged in.
4. It then gets routed to one of our Node.js backend servers.
5. The server validates the data (making sure nothing is missing) and saves the new recipe to our PostgreSQL database with a status of 'draft'.
6. Finally, the server sends back a success message, and our frontend redirects the user to their new recipe page.

2. My Database Design

I've designed a relational database schema to keep our data organized and consistent.

Key Tables

- `users`: Stores user account info like username, email, hashed password, and their role (standard or admin).
- `recipes`: The core table for all recipe details, including who posted it (`user_id`), title, description, cooking time, and its current status ('draft' or 'published'). I've used JSONB fields for ingredients and steps to keep them flexible.
- `comments`: Holds all comments, linking a user and a recipe together.
- `likes`: A simple table that tracks which user liked which recipe.
- `saved_recipes`: Tracks which recipes a user has saved to their personal collection.
- `notifications`: Stores a record for every notification (e.g., "User X liked your recipe"), linking to the user who should receive it.

Relationships and Performance

The tables are all linked using foreign keys (e.g., `recipes.user_id` points to the `users` table). To keep things running fast, I plan to add **indexes** on important columns. For example, we'll index the `users.email` column for fast logins and the `recipes.created_at` column so we can quickly fetch the newest recipes for the main feed.

3. My Plan for Scalability

One of my main goals was to ensure RecipeStudio can handle success. Here's how we'll scale as our user base grows:

1. **Stateless Backend:** I've designed the backend so that any server can handle any user's request. We won't store user login info on a single server's memory; instead, we'll use JWTs or Redis. This lets us add more servers seamlessly.
2. **Add More Servers (Horizontal Scaling):** When traffic picks up, we can simply spin up more backend server instances. The load balancer will automatically start distributing the work among them.
3. **Scale the Database:** Reading data is usually the biggest bottleneck. I've planned for this by using **read replicas**. We can have copies of our database that only handle read requests (like fetching recipes), which takes a huge load off our main database that handles the writing.
4. **Use a CDN:** Our CDN (CloudFront) will serve all our images and static files from servers located all over the world. This means the site will load quickly for users no matter where they are.

4. My Plan for Performance Optimization

A slow app is a frustrating app. Here's how I plan to keep RecipeStudio fast and responsive:

1. **Aggressive Caching:** We'll use Redis to cache common database query results. For example, the list of "trending recipes" doesn't need to be recalculated for every single user; we can calculate it once every few minutes and store it in Redis for instant access. I learned this from Aljazeera news website. Even for a news website, it is extremely fast. The main principle is to show the user something before what they want to see actually loads.
2. **Smart Loading (Pagination):** We will never try to load thousands of recipes at once. All our lists will be paginated, meaning we only load the first 20 or so, and then load more as the user scrolls. I'll use cursor-based pagination, which is more efficient than traditional page numbers.
3. **Image Optimization:** This is huge for performance. When a user uploads a large image, we'll automatically create smaller, optimized versions (thumbnails, medium-sized, etc.). We'll serve the smallest possible image size needed for any situation and use modern formats like WebP, which drastically reduces load times.

Bonus: How I'll Handle Security

Security is a top priority in my design. Here's how I plan to protect our users and their data:

1. **Authentication & Authorization:**
    - Passwords will never be stored in plain text. I'll use a strong hashing algorithm like **bcrypt**.
    - We'll use **JSON Web Tokens (JWTs)** for secure, stateless user sessions.
    - Our API will have clear rules. For example, you can only edit or delete your *own* recipes, and special admin functions will be locked down so only users with the 'admin' role can access them.
2. **Data Protection:**
    - We'll enforce **HTTPS** across the entire site to encrypt all traffic.
    - I'll make sure to sanitize all user input to prevent common attacks like **XSS** (displaying malicious code) and **SQL Injection** (database attacks).

3. **Abuse Prevention:**
    - I'll implement **rate limiting** to prevent bots from spamming our site or trying to brute-force user passwords.
    - We'll add **CAPTCHA** on the registration form to make it harder for bots to create fake accounts.