Frontend Tech Stack

- **React (Web App):**
  - Framework for building a dynamic, responsive user interface.
  - Allows easy integration with REST APIs for communication with the backend.
- **Flutter (Mobile App):**
  - A cross-platform framework for mobile app development.
  - Ensures consistent user experience on both iOS and Android platforms.
- **Key Adjustments for Scalability:**
  - **Lazy Loading:** Load components on demand to improve performance.
  - **Code Splitting:** Divide the application into chunks for faster loading and caching.

Backend Tech Stack

- **Node.js with Express.js:**
  - Highly scalable and event-driven backend framework.
  - Suitable for real-time processing, such as user notifications and feed updates.
- **Key Services:**

1. **API Gateway:** Acts as a single entry point for all requests (e.g., login, search, recipe CRUD).
2. **Authentication Service:** Handles user authentication and role-based access control.
3. **Recipe Service:** Manages recipe creation, updates, and retrieval.
4. **Notification Service:** Sends real-time notifications.
5. **Search Service:** Handles advanced search with filters (e.g., ingredients, categories).
6. **Batch Processing Service (Future Load):**
    - Handles tasks such as:
      - Precomputing trending recipes.
      - Sending bulk notifications (e.g., digest emails or scheduled pushes).
      - Offloading analytics data processing.

- **Scalability Enhancements:**
  - **Horizontal Scaling:** Add more instances of backend services using container orchestration (e.g., Kubernetes).

    - **Asynchronous Processing:** Use **RabbitMQ** or **Kafka** to queue tasks for batch processing (e.g., notifications, feed generation).

Database Tech Stack

- **PostgreSQL (Primary Database):**
    - Relational database for structured data (e.g., users, recipes, comments, likes).
    - Ensures data consistency and supports complex queries.
- **Redis (CacheDB):**
    - In-memory caching layer for frequently accessed data like:
        - Trending recipes.
        - Cached search results.
        - Notifications.
    - Reduces latency for users by avoiding frequent database hits.
- **Key Adjustments for Scalability:**
    - **Database Sharding:** Split the database horizontally to handle large-scale user data.
    - **Read Replicas:** Create replicas of the database for read-heavy workloads (e.g., analytics, feed retrieval).

Third-Party Services

- **AWS S3 (Cloud Storage):**
    - Stores recipe images and serves them through a CDN (e.g., CloudFront) for faster delivery.
- **Firebase Cloud Messaging (Push Notification):**
    - Sends real-time notifications to users.
- **Scalability Enhancements:**
    - Use **CDN** for global content delivery to minimize latency for users in different regions.

Batch Processing Implementation

- **When to Use Batch Processing?**
    - If real-time processing becomes inefficient due to high load (e.g., millions of users interacting simultaneously).

    - Examples:
        - Generating a daily trending feed.
        - Sending bulk notifications (digest emails, weekly recipe trends).
        - Processing analytics (e.g., calculating metrics for trending recipes).
- **Tech Stack for Batch Processing:**
    - **Apache Kafka or RabbitMQ:**
        - Message brokers for queuing large-scale tasks (e.g., feed generation).
        - Kafka is more suitable for high-throughput tasks.
    - **Apache Spark:**
        - Processes large datasets in batches (e.g., trending recipe computations).
        - Integrates well with analytics data stored in PostgreSQL or a data lake.
    - **Cron Jobs or Workflow Orchestration:**
        - Use tools like **Apache Airflow** or Kubernetes Cron Jobs to schedule batch tasks during non-peak hours.
- **Batch Processing Flow:**

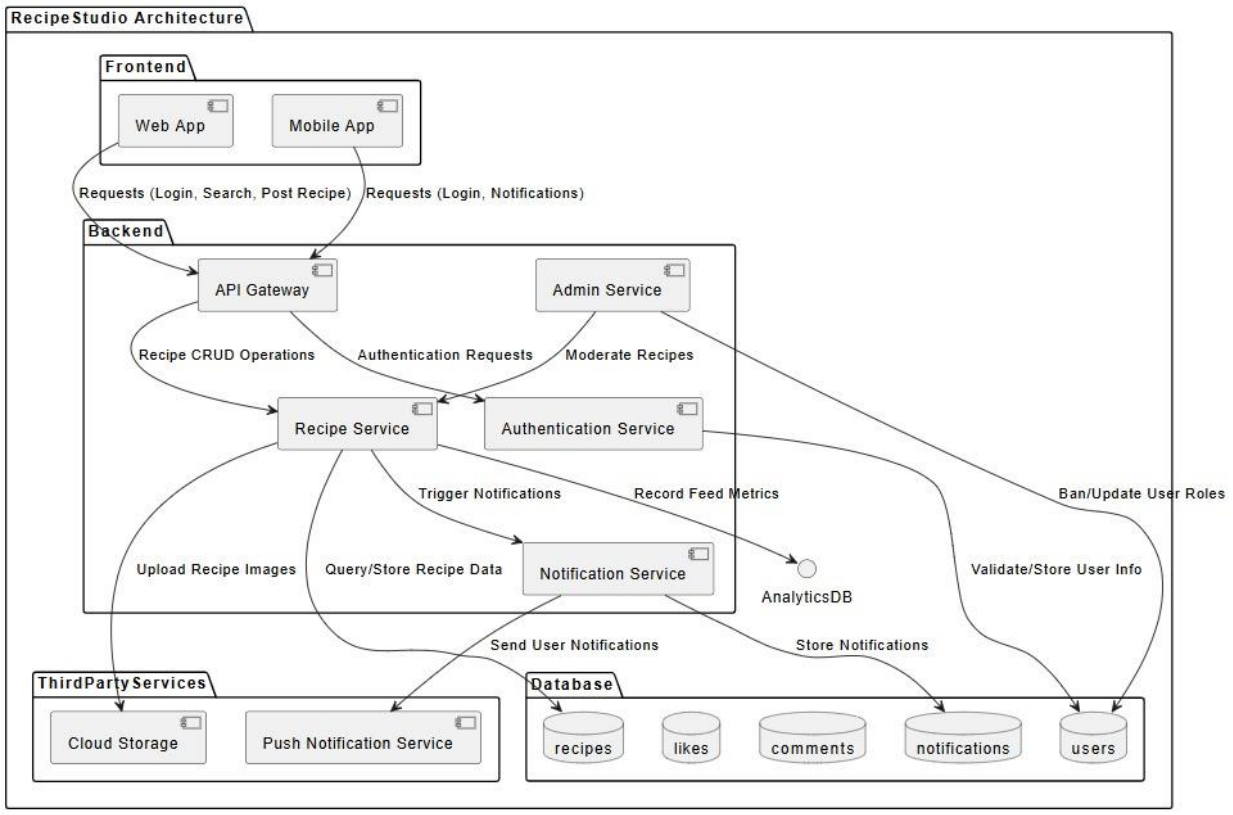
1. Analytics data (e.g., likes, views, comments) is collected in **AnalyticsDB**.
2. The Batch Processing Service fetches this data during scheduled intervals.
3. Precomputes trending recipes, updates cached feeds (Redis), and sends notifications.

Scaling for Increased Load

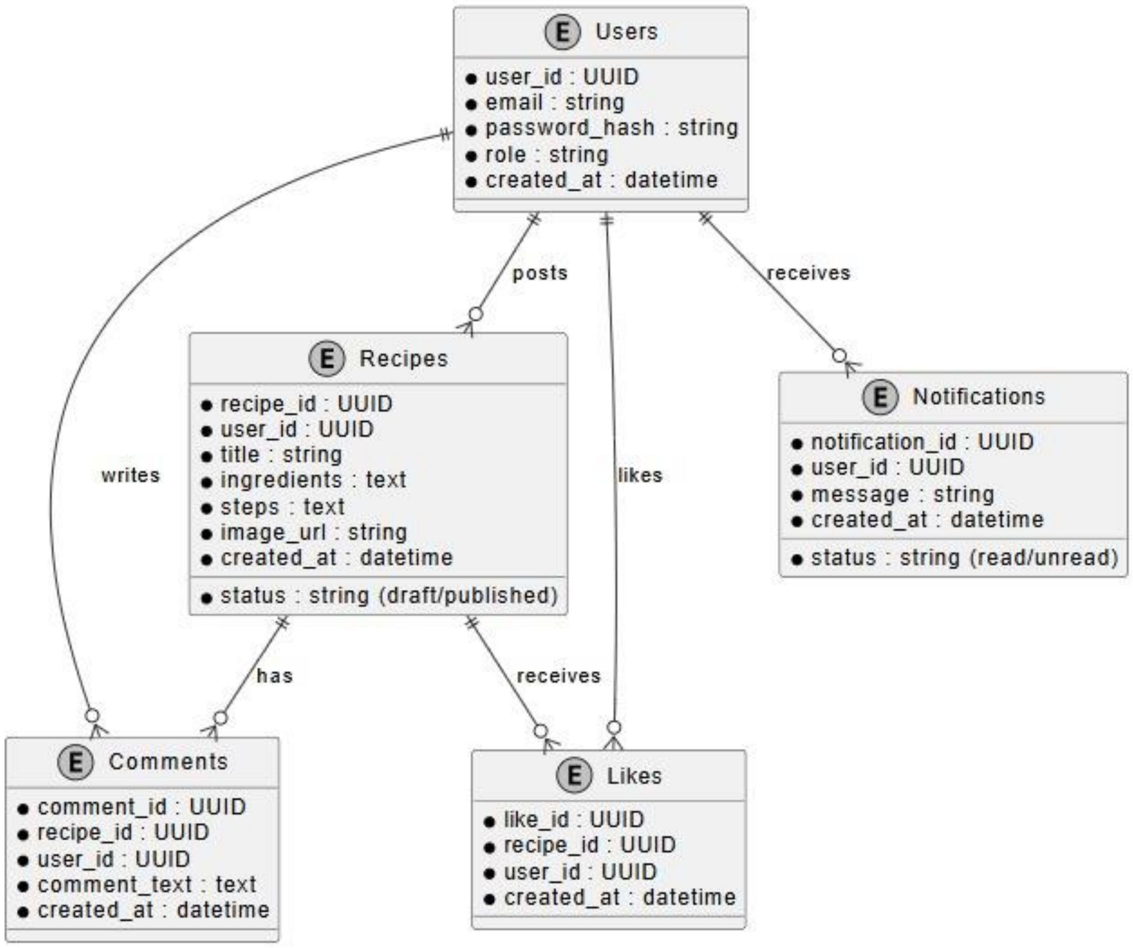
1. **Frontend:**
    - Use a CDN to serve static assets (e.g., images, CSS, JS files) efficiently.
    - Implement service workers for caching and offline support.
2. **Backend:**
    - Adopt **microservices architecture** to decouple components (e.g., Recipe Service, Notification Service).
    - Use **container orchestration** (e.g., Kubernetes) for auto-scaling.
3. **Database:**
    - Use **read replicas** for heavy read operations.
    - Cache frequently accessed queries in Redis.

4. **Batch Processing:**
   - Offload heavy tasks to batch processes (e.g., scheduled feed updates, bulk notifications).
   - Use **event-driven architecture** with Kafka for processing real-time streams if needed.

Application Architecture:



Database Schema:


Users
user_id : UUID
email : string
password_hash : string
role : string
created_at : datetime
posts
receives
writes
likes
Recipes
recipe_id : UUID
user_id : UUID
title : string
ingredients : text
steps : text
image_url : string
created_at : datetime
status : string (draft/published)
Notifications
notification_id : UUID
user_id : UUID
message : string
created_at : datetime
status : string (read/unread)
has
receives
Comments
comment_id : UUID
recipe_id : UUID
user_id : UUID
comment_text : text
created_at : datetime
Likes
like_id : UUID
recipe_id : UUID
user_id : UUID
created_at : datetime