System Design Document

Introduction

The Insyd notification system is built to keep users connected and engaged by letting them know what's happening around them in real-time. Whether it's a like on their post, a new follower, or a job update, the system ensures that people don't miss out on meaningful interactions.

Right now, we're designing for around 100 daily active users (DAUs), but the architecture is built to smoothly scale all the way to 1 million DAUs as the platform grows.

Goals

- Deliver **real-time notifications** for key user interactions
- Handle different notification types: likes, comments, follows, new posts, etc.
- Scale from a small user base to **1M+ DAUs** without major redesigns
- Keep latency low so notifications feel instant
- Ensure the system is reliable, fault-tolerant, and ready for future features like email or push alerts

---

System Overview

The notification system uses an **event-driven architecture**. Whenever something happens (like a user likes a post), an event is generated, processed, and delivered as a notification.

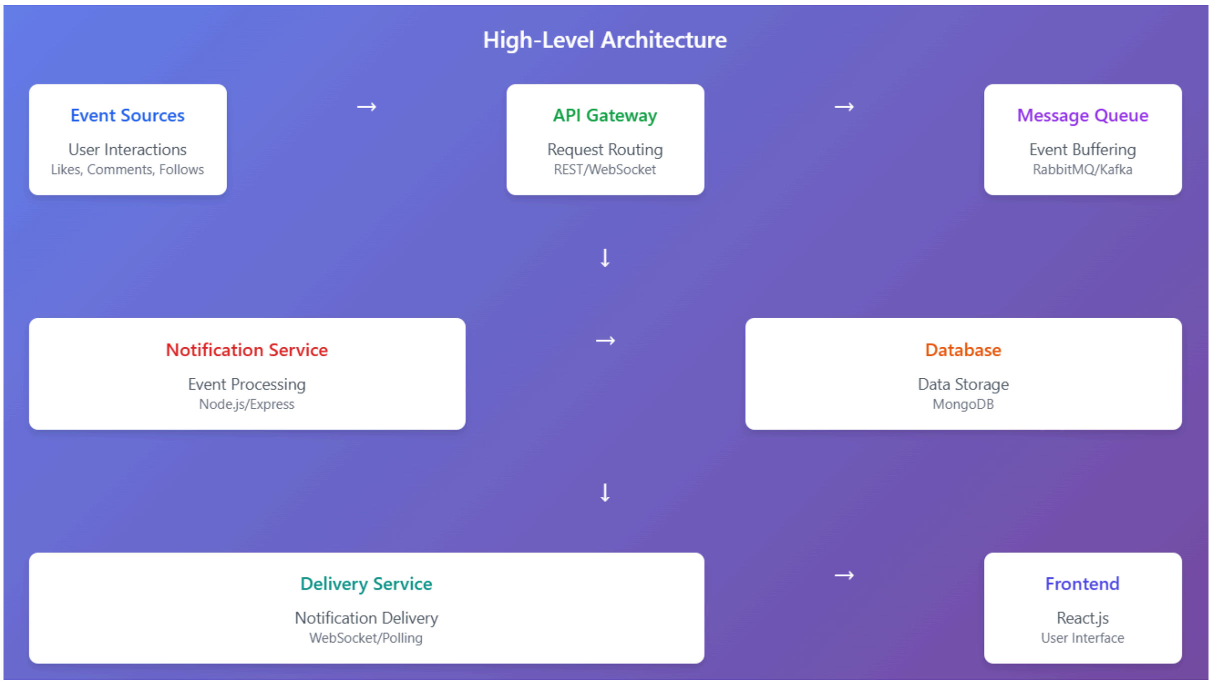
For the initial proof of concept (POC), we'll focus on **in-app notifications**, but the design allows us to add **push and email notifications** later on.

High-Level Flow

1. **User actions** → (Like, Comment, Follow, Post, etc.)

2. **API Gateway** → Receives the event and routes it properly
3. **Message Queue (Kafka/RabbitMQ)** → Buffers the event, helps handle traffic spikes
4. **Notification Service** → Processes events, creates the right notification, applies user preferences
5. **Database (MongoDB)** → Stores notifications, history, and preferences
6. **Delivery Service** → Sends the notification via WebSocket (real-time) or polling (fallback)
7. **Frontend (React.js)** → Displays the notification to the user

---



Components

1. Event Sources

Anything that triggers a notification, such as:

- Post likes or comments

- New followers
- Content shares
- Direct messages
- Job post interactions

2. API Gateway

The central entry point that:

- Routes incoming events to the right services
- Manages authentication and rate limiting
- Handles WebSocket connections
- Balances load across services

3. Message Queue (Kafka/RabbitMQ)

- Acts as a **buffer** between event producers and consumers
- Ensures events aren't lost during traffic spikes
- Makes the system more scalable and fault-tolerant

4. Notification Service

- The **brains** of the system
- Processes events pulled from the queue
- Decides what type of notification to create
- Checks user notification preferences
- Saves notifications in the database

5. Database (MongoDB)

- Stores user profiles and notification preferences

- Maintains notification history for easy retrieval
- Supports fast querying for the frontend

6. Delivery Service

- Sends notifications to users in real time (via WebSockets)
- Uses polling as a fallback for offline or unstable clients
- Can expand to support **email, SMS, or push notifications** later

7. Frontend (React.js)

- Provides the **user-facing experience**
- Displays notifications in a clean, real-time interface
- Connects to the backend via WebSocket