AI_CHAT_BOX

SYSTEM DESIGN

- By Karan Kumar

Introduction

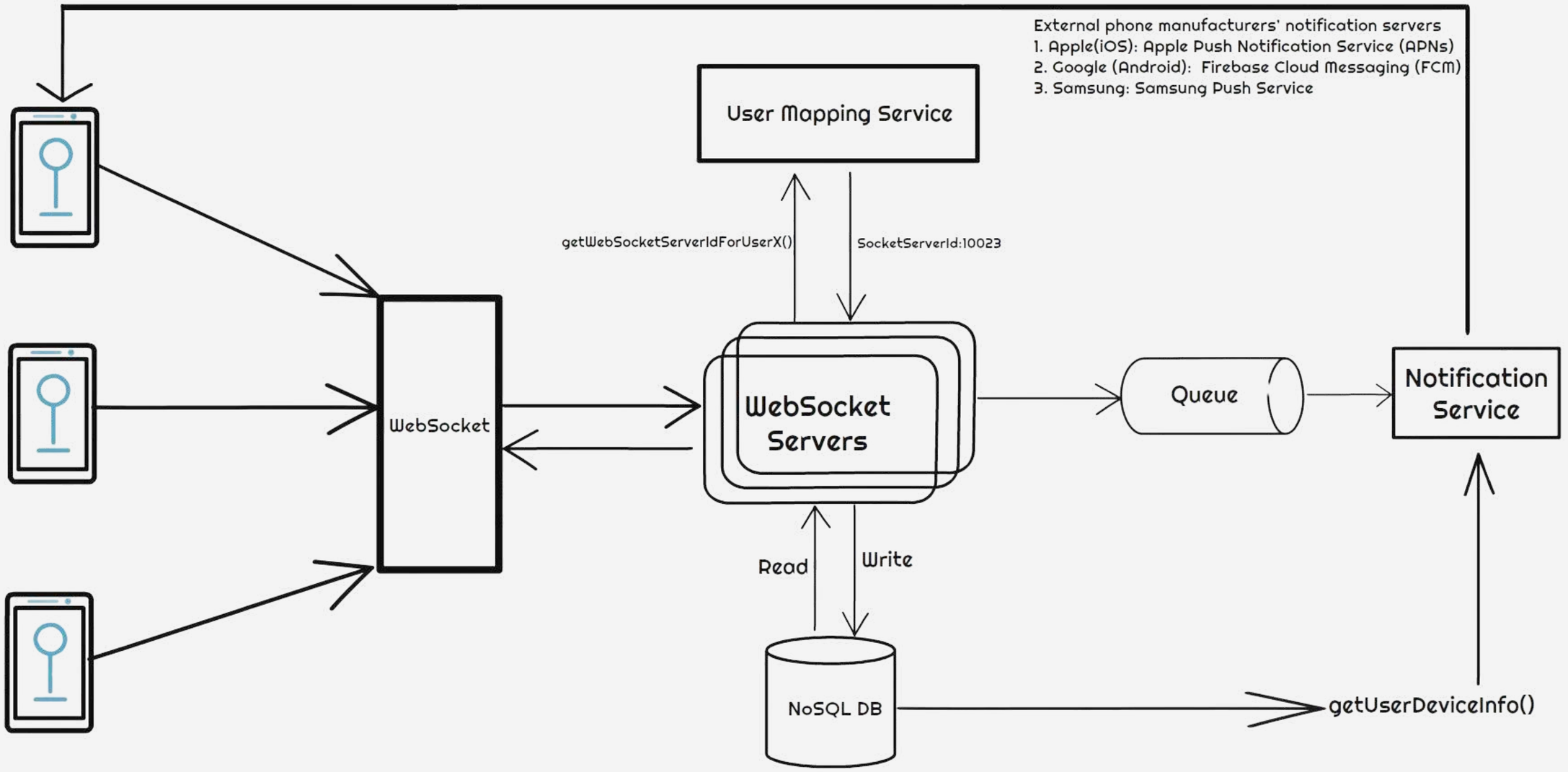
What is a Real-time Chat Application?

A system that allows users to send and receive messages instantly, without noticeable delay.

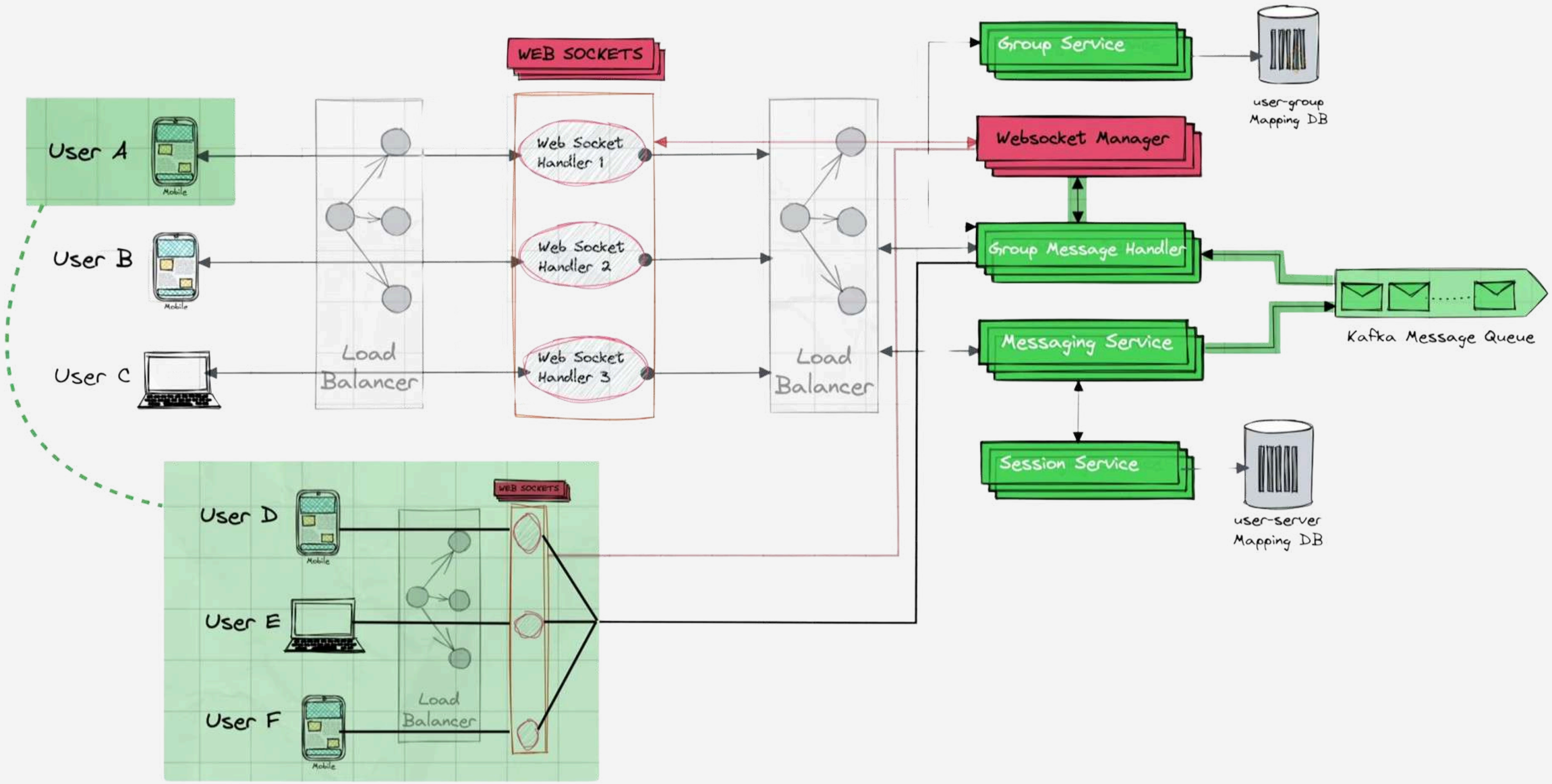
Objective of the Presentation

Explain how to design a scalable system that can handle real-time messaging between users.

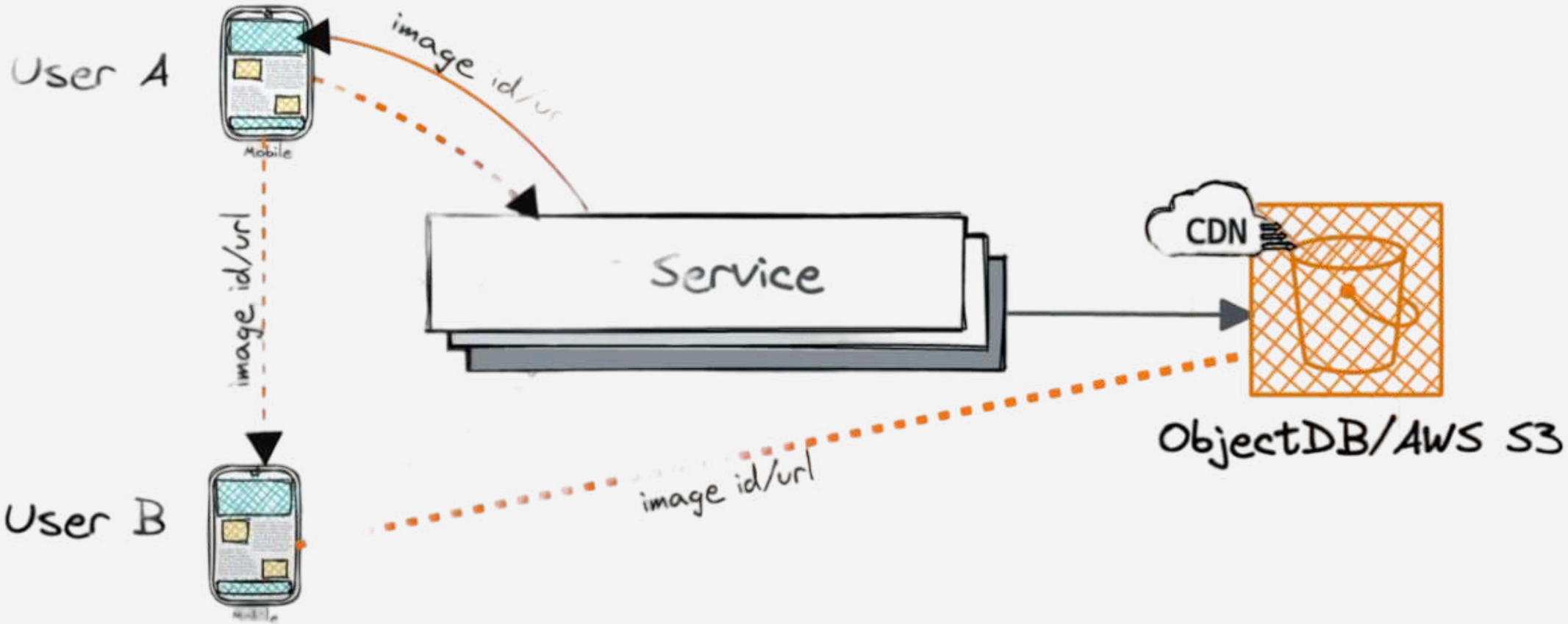
User To User Chats Handling



Group Chats Handling



Media Files Handling



Core Features of a Real-Time Chat Application

Instant Messaging

Messages are delivered in real time, ensuring fast and seamless communication.

User Authentication & Authorization

Ensures only authenticated users can access the chat system, managing permissions securely.

Message Persistence

Stores messages so users can retrieve them later, even after logging out or switching devices.

Presence, Typing, and Read Indicators

Shows user’s online/offline status, notifies when someone is typing, and indicates when a message has been read by the recipient, all in real-time.

System Architecture Overview

Client-Server Model

The chat app uses a client-server structure where clients (users) connect to the server to send and receive messages.

Component Breakdown

- Frontend (Client): Manages the user interface (UI) and interactions.
- Backend (Server): Manages business logic, such as handling messages and user sessions.
- Database: Stores user information and messages.

Real-Time Protocols

WebSockets enable continuous, two-way communication between client and server, unlike HTTP requests, which are one-way.

Message Flow and Data Flow

Sending a Message

Client sends the message via WebSocket, which the server processes and forwards to the intended recipient.

Receiving a Message

The server receives messages and delivers them in real-time to the recipient's client through WebSocket.

Handling Connection Loss

Techniques like retry mechanisms to ensure messages are delivered when the connection is restored.

Scaling for Concurrent Users

Use load balancers and horizontally scale servers to handle thousands of simultaneous connections.

Database Design

- **Messages Table/Collection**
  Stores each message with relevant data like sender, recipient, and timestamp.
- **Users Table**
  Contains user information, including online status and chat preferences.
- **Indexing**
  Indexes help retrieve active conversations quickly by user ID or timestamp for performance.

High Availability and Scalability

- **Horizontal Scaling**
  Add more servers to handle more traffic instead of increasing the capacity of a single server.
- **Load Balancers**
  Distribute incoming requests across multiple servers to avoid overload.
- **Database Replication & Sharding**
  Replication creates multiple copies of data for reliability, while sharding splits the data across multiple databases for performance.

Security Considerations

- **Data Encryption**
  Ensures data (messages) is encrypted during transmission (via TLS) and while stored in the database (at rest).
- **User Authentication**
  Use secure authentication methods like JWT tokens or OAuth to validate users.
- **Rate Limiting**
  Prevents abuse by limiting the number of messages or requests a user can send in a given time.
- **SQL/NoSQL Injection Prevention**
  Input validation to protect against malicious attempts to manipulate the database.

Performance Optimization

- **Message Delivery Latency**

  Minimize delay in message delivery by optimizing WebSocket connections and reducing overhead.

- **Caching**

  Store frequently accessed data (like recent messages or user status) in fast-access memory (Redis/Memcached).

- **Event-Driven Architecture**

  Use message queues like RabbitMQ/Kafka to manage asynchronous tasks like notifications or message processing.

Monitoring and Logging

Monitoring Tools

Use Prometheus or Grafana to track system health, user activity, and message delivery success rates.

Logging

Keep logs for debugging and analyzing errors or performance issues related to user interactions and message failures.

Thank you!