Experiment 5

Student Name: Gourav Sharma **UID: 23BCS10857**
Branch: CSE **Section/Group: KRG 3-A**
Semester: 6th **Date of Performance:26/02/2026**
Subject Name: System Design **Subject Code: 23CSH-314**

1. **Aim**: Design a Messenger Application(Similar to WhatsApp / Facebook Messenger): To design and analyze the architecture of a scalable real-time messenger application that supports one-to-one and group communication with high availability and low latency.

2. **Objective**:

- To implement real-time message delivery using WebSockets.
- To support one-to-one and group messaging.
- To preserve chat history.
- To ensure high availability and reliability.
- To design APIs for user management and chat operations.
- To design the system for large-scale usage (up to 1 Billion users).

3. **Tools Used:**

- **Frontend (ReactJS, HTML, CSS, JavaScript, WebSocket)**
  - Built responsive UI with real-time communication support
- **Backend (Node.js / Java / Spring Boot)**
  - Developed scalable REST APIs & WebSocket server
- **Database (MongoDB / MySQL / PostgreSQL)**
  - Managed structured & unstructured data efficiently
- **Redis**
  - Implemented caching & pub-sub for high performance
- **Deployment (AWS / GCP, Load Balancer, Docker)**
  - Cloud deployment with scalability, containerization & traffic distribution

4. **System Requirements:**

A. Functional Requirements

- **User Registration**
  a. POST /user/register
  b. Fields: userName, email, phoneNumber
- **User Login**
  a. POST /user/login
- **One-to-One Messaging**
  a. WS: /messages/send

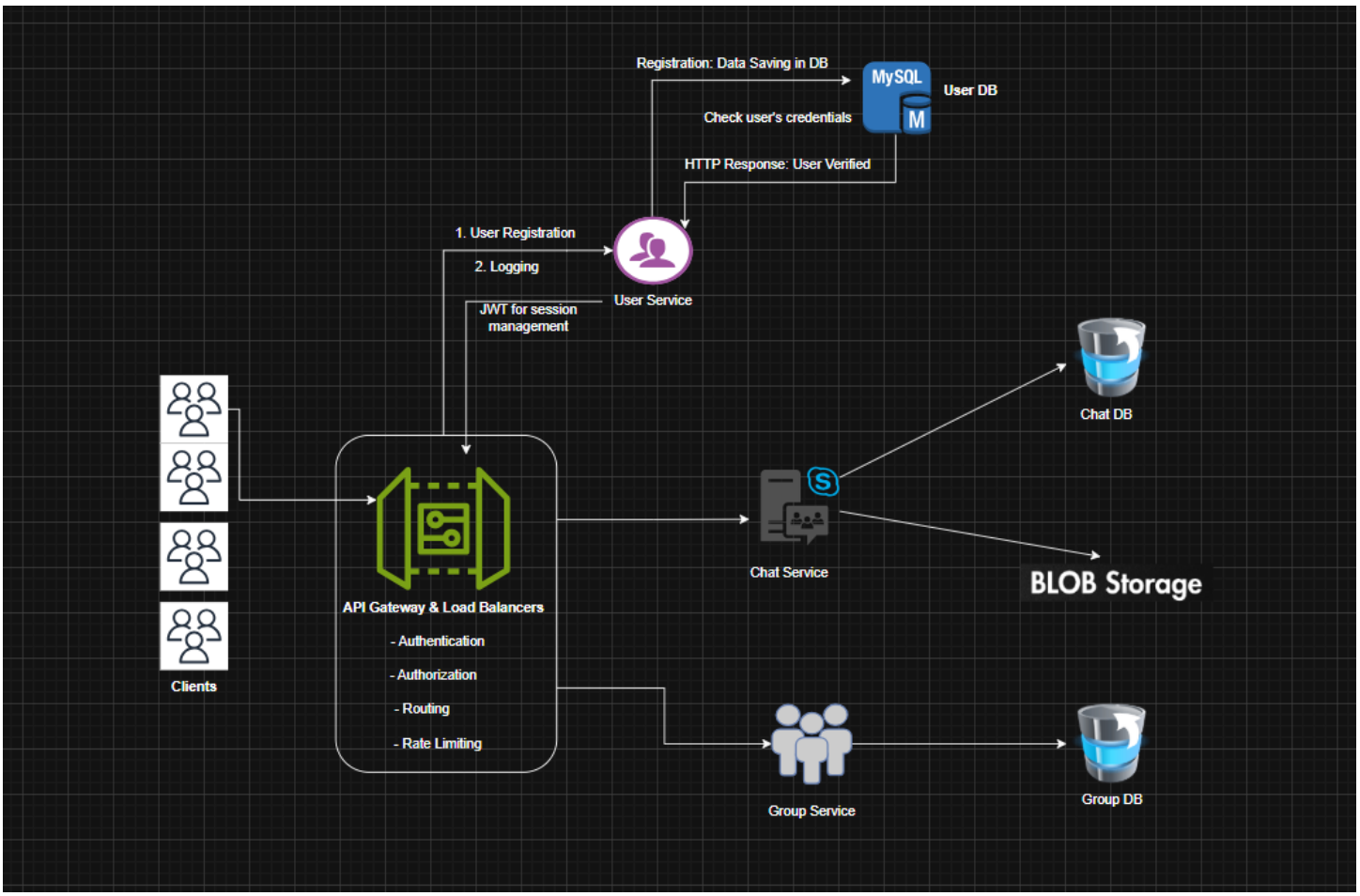
   b. GET /messages/{userId}/{receiverId}
   c. Pagination support
- **Group Messaging**
   a. POST /groups/create
   b. POST /groups/{groupId}/add
   c. DELETE /groups/{groupId}/remove
   d. WS: /messages/send
- **Message Types**
   a. Text messages
   b. Media messages (images, videos, files)
- **Message History**
   a. Persistent storage
   b. Chat list view
- **Read Receipts**
   a. Delivered
   b. Seen

B. Non-Functional Requirements

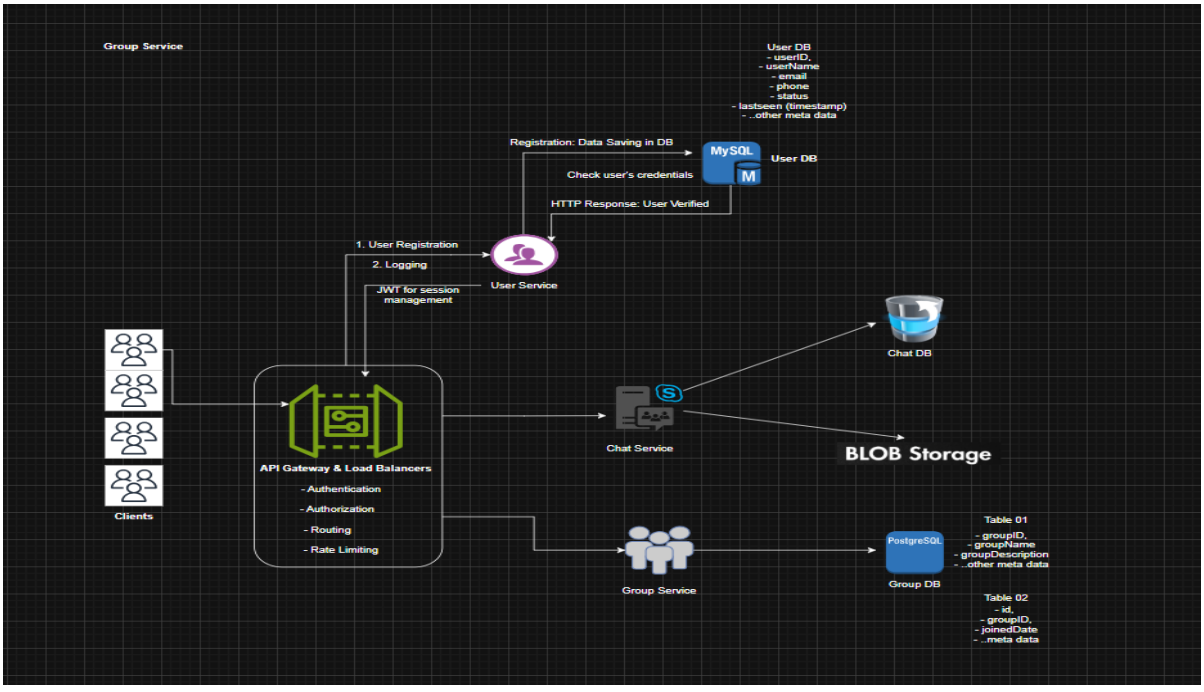
- **Scalability**
   - Target: **1 Billion users**
   - 100 messages per user per day
   - ≈ 100 Billion messages/day
   - Estimated storage: ~100 TB

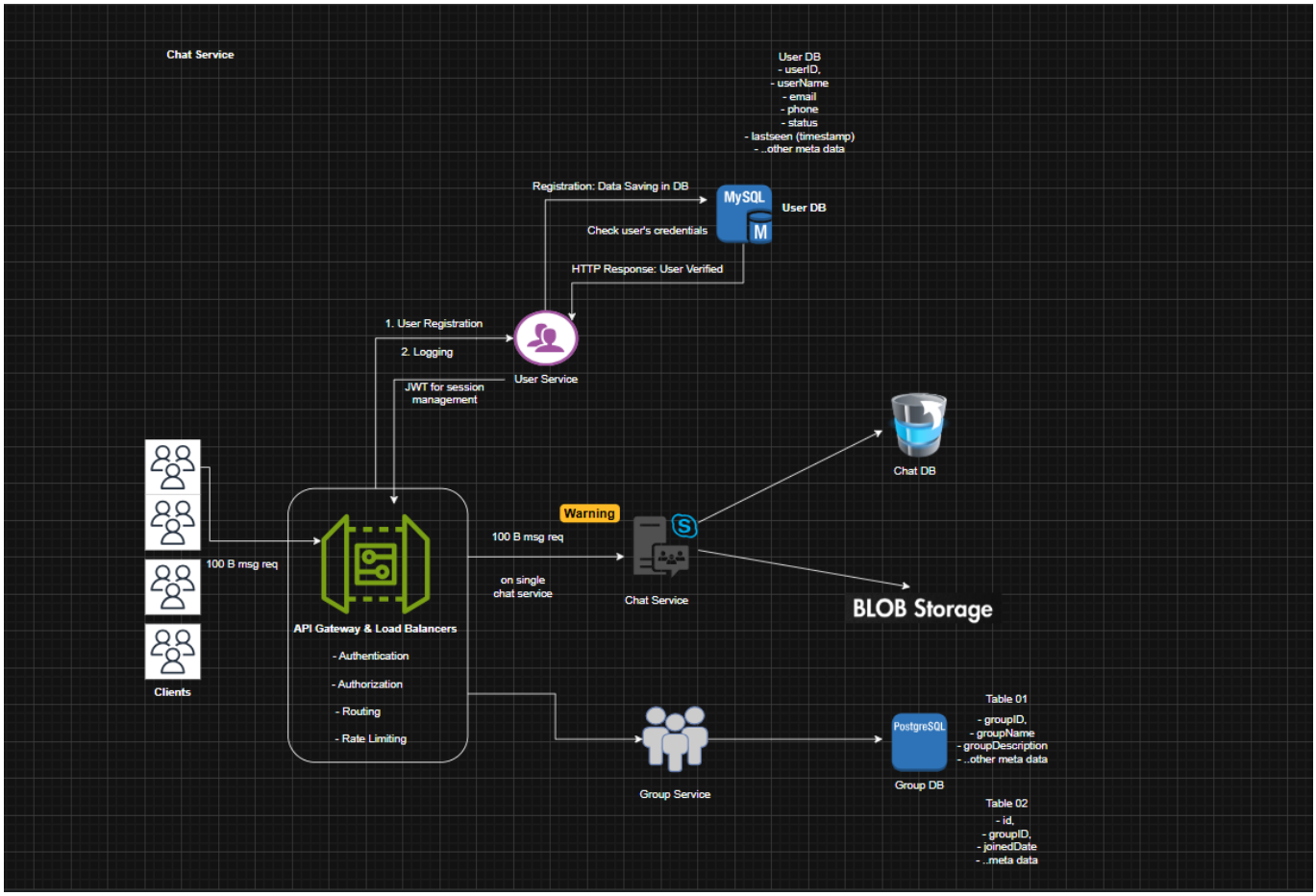
- **CAP Theorem**
  System prioritizes:
   - High Availability
   - Partition Tolerance
   - Eventual Consistency
- **Latency: 200-300ms**
   - Target message delivery time: **200–300 ms**

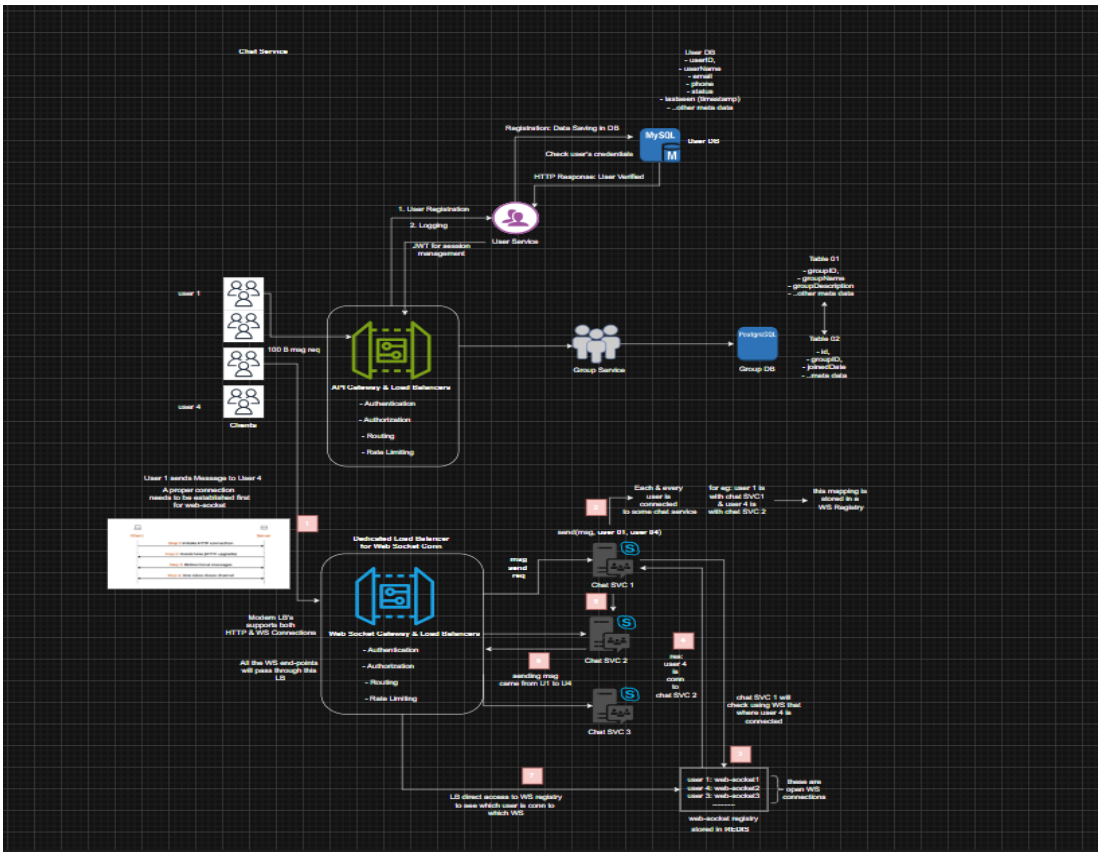
5. High Level Design (HLD)

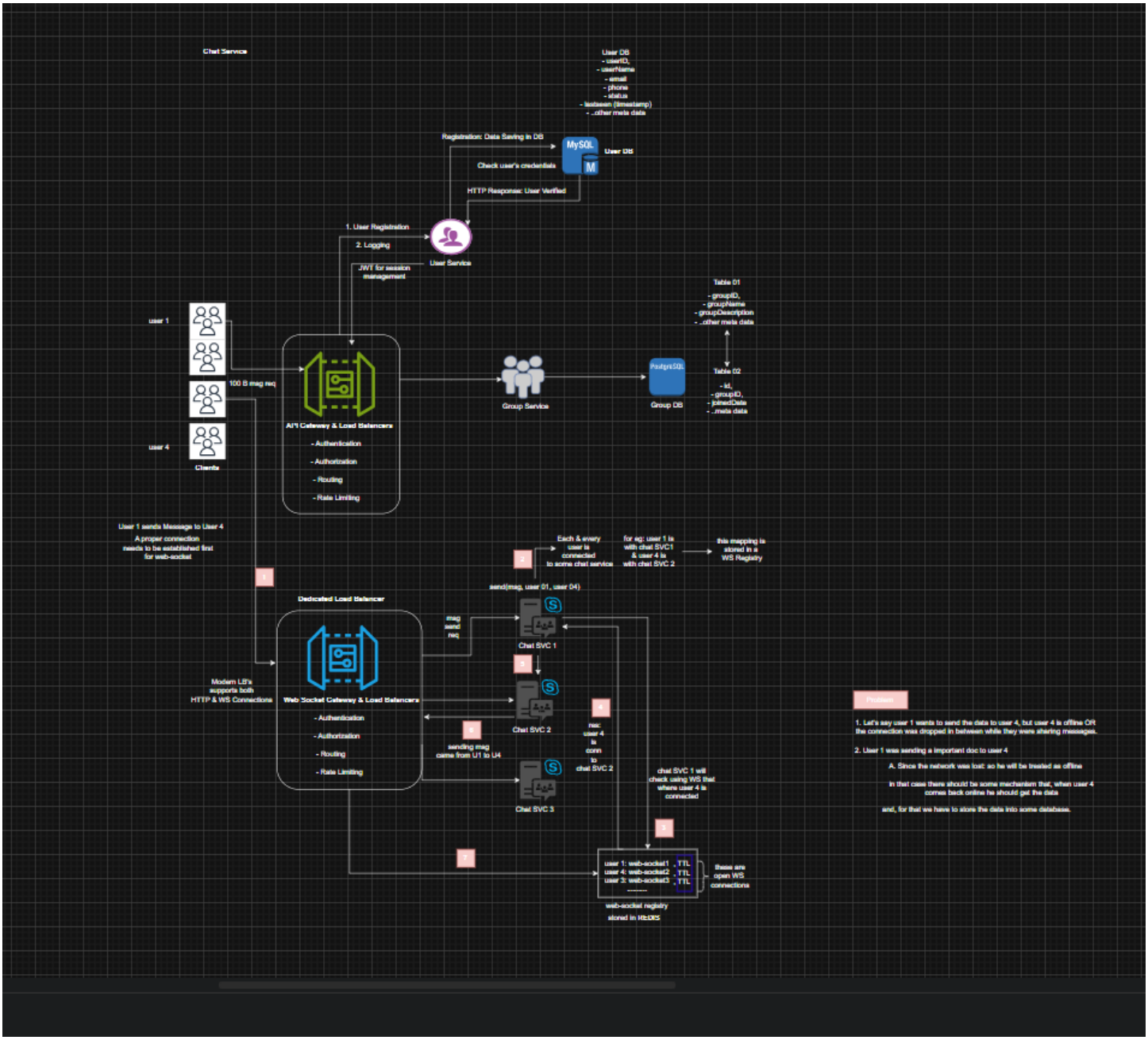


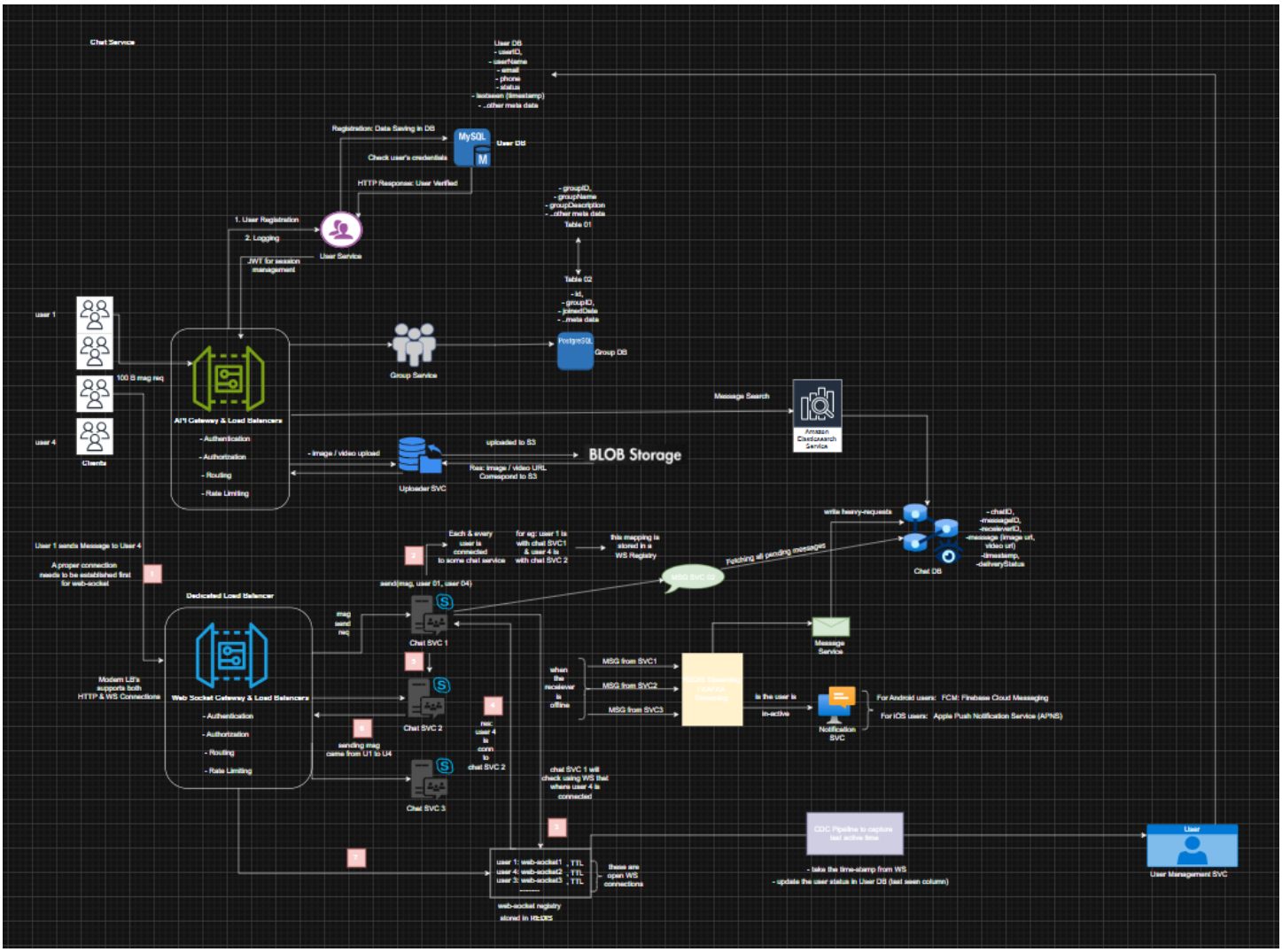
6. Low Level Design (LLD):




Chat Service
User DB
- userID,
- userName
- email
- phone
- status
- lastseen (timestamp)
- ..other meta data
Registration: Data Saving in DB
MySQL
User DB
Check user's credentials
HTTP Response: User Verified
1. User Registration
2. Logging
User Service
JWT for session management
Chat DB
Warning
100 B msg req
100 B msg req
on single chat service
Chat Service
BLOB Storage
Clients
API Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Table 01
- groupID,
- groupName
- groupDescription
- ..other meta data
PostgreSQL
Group DB
Group Service
Table 02
- id,
- groupID,
- joinedDate
- ..meta data


Chat Service
User DB
Registration: Data Saving in DB
MySQL
Check user's credentials
HTTP Response: User Verified
1. User Registration
2. Logging
User Service
JWT for session management
Table 01
Table 02
Group Service
Group DB
API Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Clients
User 1 sends Message to User 4
A proper connection needs to be established first for web-socket
Dedicated Load Balancer for Web Socket Conn
Web Socket Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Modern LB's supports both HTTP & WS Connections
All the WS end-points will pass through this LB
Each & every user is connected to some chat service
for eg: user 1 is with chat SVC1 & user 4 is with chat SVC 2
this mapping is stored in a WS Registry
send(msg, user 01, user 04)
msg send req
Chat SVC 1
Chat SVC 2
Chat SVC 3
sending msg came from U1 to U4
now user 4 is conn to chat SVC 2
chat SVC 1 will check using WS that where user 4 is connected
LB direct access to WS registry to see which user is conn to which WS
user 1: web-socket1
user 4: web-socket2
user 3: web-socket3
these are open WS connections
web-socket registry stored in Rb:DB5


Chat Service
User DB
- userID,
- userName
- email
- phone
- status
- lastseen (timestamp)
- ..other meta data
Registration: Data Saving in DB
MySQL
User DB
Check user's credentials
HTTP Response: User Verified
1. User Registration
2. Logging
User Service
JWT for session management
Table 01
- groupID,
- groupName
- groupDescription
- ..other meta data
Table 02
- id,
- groupID,
- joinedDate
- ..meta data
user 1
user 4
Clients
100 B msg req
API Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Group Service
PostgreSQL
Group DB
User 1 sends Message to User 4
A proper connection needs to be established first for web-socket
Each & every user is connected to some chat service
for eg: user 1 is with chat SVC1 & user 4 is with chat SVC 2
this mapping is stored in a WS Registry
send(msg, user 01, user 04)
Dedicated Load Balancer
msg send req
Chat SVC 1
Modern LB's supports both HTTP & WS Connections
Web Socket Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Chat SVC 2
sending msg came from U1 to U4
rec user 4 is conn to chat SVC 2
Chat SVC 3
chat SVC 1 will check using WS that where user 4 is connected
user 1: web-socket1 , TTL
user 4: web-socket2 , TTL
user 3: web-socket3 , TTL
.....
these are open WS connections
web-socket registry stored in Redis
Problem
1. Let's say user 1 wants to send the data to user 4, but user 4 is offline OR the connection was dropped in between while they were sharing messages.
2. User 1 was sending a important doc to user 4
A. Since the network was lost: so he will be treated as offline
in that case there should be some mechanism that, when user 4 comes back online he should get the data
and, for that we have to store the data into some database.



7. Scalability Solution

- **Horizontal Scaling**
  Add multiple application servers behind a load balancer to distribute traffic and handle billions of requests efficiently.
- **Database Sharding**
  Partition data based on userID or chatID so that message load is distributed across multiple database servers.
- **Caching with Redis**
  Store frequently accessed chats and recent messages in Redis to reduce database load and improve response time.
- **Message Queue System**
  Use Kafka/RabbitMQ for asynchronous message processing and reliable message delivery.
- **CDN for Media Files**
  Use Content Delivery Network to efficiently deliver images, videos, and files globally with low latency.

8. Learning Outcomes (What I Have Learnt)

- Understand real-time communication using WebSockets.
- Learn how to design scalable distributed systems.
- Apply CAP Theorem concepts in real-world system design.
- Design RESTful APIs and WebSocket endpoints.
- Analyze trade-offs between consistency, availability, and latency.

.