A SYNOPSIS ON

MERNCOM

Submitted in partial fulfilment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

In

Computer Science & Engineering

Submitted by:

Kiran (Sec A, CRN: 94, Roll No: 2261322)

Jaya Mehta (Sec A, CRN: 100, Roll No: 2261283)

Jatin Talaniya (Sec C, CRN: 51, Roll No: 2261282)

Ayushi Adhikari (Sec B, CRN: 92, Roll No: 2261136)

Under the Guidance of

Mr Anubhav Bewerwal

Designation

Project Team ID: 41

Department of Computer Science & Engineering

Graphic Era Hill University, Bhimtal, Uttarakhand

March-2025

Graphic Era Hill University
BHIMTAL CAMPUS

CANDIDATE'S DECLARATION

I/We hereby certify that the work which is being presented in the Synopsis entitled **"MERNCOM"** in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science & Engineering of the Graphic Era Hill University, Bhimtal campus and shall be carried out by the undersigned under the supervision of **Guide Name, Designation**, Department of Computer Science & Engineering, Graphic Era Hill University, Bhimtal.

Kiran	2261322
Jaya Mehta	2261283
Jatin Talaniya	2261282
Ayushi Adhikari	2261136

The above mentioned students shall be working under the supervision of the undersigned on the **"MERNCOM"**

Signature   Signature

Supervisor   **Head of the Department**

<u>Internal Evaluation (By DPRC Committee)</u>

Status of the Synopsis: Accepted / Rejected

Any Comments:

Name of the Committee Members: **Signature with Date**

1.

2.

Table of Contents

Chapter 1

Introduction and Problem Statement

1.1 Introduction

In the modern digital era, instant communication is essential for collaboration, social interaction, and productivity. Existing chat applications often introduce unnecessary complexities, high resource consumption, or lack flexibility for customization. Businesses, developers, and small communities need a lightweight, scalable, and easy-to-deploy chat solution that ensures real-time messaging with a seamless user experience.

This project presents a full-stack real-time chat application built using the MERN stack (MongoDB, Express, React, Node.js), designed to provide instant, secure, and efficient communication. The frontend is styled using Tailwind CSS and Daisy UI, ensuring a modern and responsive user interface. Zustand is utilized for state management in React, while Socket.io enables event-driven, bi-directional communication for real-time interactions. Additionally, JWT authentication ensures secure, token-based user access, while Cloudinary is leveraged for scalable, cloud-based image storage with CDN optimization.

Furthermore, the system is designed to be fault-tolerant and scalable, integrating multi-threaded WebSocket handling, asynchronous data pipelines, and load balancing techniques for improved performance. Microservices architecture and containerization using Docker ensure modularity and ease of deployment, making the system highly adaptable to different infrastructures.

1.2 Problem Statement

While numerous chat applications exist, most solutions are either too complex for small-scale use, require extensive backend infrastructure, or lack customization options. Many applications also suffer from high latency and inefficient state management, leading to suboptimal user experiences.

This project aims to build a scalable, lightweight, and customizable chat application that overcomes these challenges by integrating efficient real-time communication (Socket.io), secure authentication (JWT), state management (Zustand), and image storage (Cloudinary). The system ensures horizontal scalability and event-driven architecture, making it an ideal solution for small teams, businesses, and developers.

Chapter 2

Background / Literature Survey

2.1 Introduction

The development of real-time chat applications has evolved significantly with advancements in web technologies. Early messaging systems relied on polling mechanisms, which led to high latency and inefficient resource utilization. Modern solutions leverage WebSocket communication, cloud-based storage, and optimized authentication mechanisms to enhance user experience and security. This chapter provides a comprehensive review of existing systems, technologies, and research studies relevant to real-time chat applications.

2.2 Background

The demand for real-time communication has surged with the increasing reliance on digital collaboration tools. Traditional HTTP request-response models were inadequate for real-time applications due to their high overhead. The advent of WebSockets revolutionized real-time messaging by enabling persistent connections between clients and servers. Additionally, the adoption of **JWT authentication** ensures secure user verification, while **state management libraries like Zustand** optimize frontend performance. This project integrates these technologies to build a scalable and efficient chat application.

2.3 Existing Chat Applications

Several real-time chat applications exist in the market, including WhatsApp, Slack, Telegram, and Discord. While these applications offer robust functionalities, they often have high infrastructure demands, making them unsuitable for small-scale, customizable deployments.

Limitations of Existing Systems:

1. **Centralized Infrastructure:** Most widely used applications rely on centralized servers, creating bottlenecks and potential single points of failure.
2. **High Resource Consumption:** Applications like Discord and Slack use extensive backend services that require significant computational power.
3. **Limited Customization:** Open-source alternatives exist, but many lack flexibility for developers to modify core functionalities.
4. **Security Concerns:** Many chat applications do not provide fully end-to-end encrypted messaging or role-based access control (RBAC).

2.4 WebSocket-Based Communication

Traditional HTTP-based polling methods are inefficient for real-time applications, leading to high server loads. WebSocket-based communication, facilitated by **Socket.io**, allows bidirectional, event-driven interactions between client and server with minimal latency. Studies have shown that WebSockets are **70% more efficient** than long polling for real-time messaging applications.

2.5 Secure Authentication Mechanisms

Many chat applications still rely on **session-based authentication**, which poses security risks. JSON Web Tokens (JWT) provide a **stateless, scalable, and secure authentication mechanism** that ensures user data integrity. This approach allows authentication to be offloaded to authentication servers, enhancing overall system performance.

2.6 State Management in Real-Time Applications

Managing application state efficiently is crucial for real-time applications. Zustand, a lightweight alternative to Redux, ensures minimal re-renders, **reducing state update latency by up to 50%** compared to traditional state management libraries.

2.7 Cloud-Based Media Storage

Handling media files in a chat application requires efficient storage and delivery mechanisms. **Cloudinary** provides a **Content Delivery Network (CDN)-backed** image storage solution, enabling fast media retrieval and optimization. Studies indicate that CDN-based image storage **reduces media load times by 60%**, significantly improving the user experience.

Comparative Analysis of Backend Architectures:

Architecture	Advantages	Limitations
Monolithic	Simple to develop	Difficult to scale
Serverless	Auto-scaling	Higher latency for cold starts

This study concludes that a **microservices-based WebSocket communication model** offers the best trade-off between performance and scalability for a real-time chat system.

Chapter 3

Objectives

The primary objectives of the proposed work are:

1. **Development of a Real-Time Chat Application**
    a. Build a feature-rich real-time chat system using the MERN stack.
    b. Ensure fast, bidirectional communication using Socket.io.
2. **Implementation of Secure Authentication**
    a. Use JWT-based authentication for secure user login and session management.
3. **Efficient State Management**
    a. Integrate Zustand for optimized, lightweight state management.
    b. Ensure seamless user experience with minimal re-renders in React.
4. **Scalable and Cloud-Based Image Storage**
    a. Use Cloudinary for storing and optimizing chat media.
    b. Ensure CDN-based fast image delivery for better performance.
5. **Optimized Database and Query Performance**
    a. Use MongoDB with indexing and replication for faster queries.
    b. Implement efficient data structures for message retrieval and search.
6. **Performance Optimization and Load Balancing**
    a. Deploy WebSockets with a multi-threaded event loop for efficient real-time communication.
7. **User Experience and Interface Design**
    - Develop an intuitive UI using Tailwind CSS and Daisy UI.
    - Ensure mobile responsiveness and accessibility.

Chapter 4

Hardware and Software Requirements

4.1 Hardware Requirements

Sl. No	Name of the Hardware	Specification
1	CPU (Processor)	Minimum : intel core i3 /amd Ryzen 3 Recommended : intel core i5,i7 / amd Ryzen 5,7
2	RAM	8 GB or higher
3	Storage(HDD or SSD)	Minimum: 50 GB SSD or higher
4	Network	Stable Internet Connection

4.2 Software Requirements

Sl. No	Name of the Software	Specification
1	Operating system	Windows 10/11 /Linux /MacOS
2	] Backend Framework	Node.js ,Express.js
3	Frontend Framework	React.js
4	Database	MongoDB with Replication & Indexing
5	Real-Time Library	Socket.io
6	UI Framework	Tailwind CSS, Daisy UI
7	State Management	Zustand
8	Authentication	JWT (JSON Web Token)
9	Image Storage	Cloudinary

Chapter 5

Possible Approach/ Algorithms

5.1 Overall System Architecture

- The application follows a **client-server model** using the **MERN stack**.
- **WebSockets (Socket.io)** facilitate **real-time bidirectional** communication.
- The backend is built with **Node.js and Express**, with **MongoDB as the database**.
- The frontend uses **React with Zustand** for state management.
- Authentication is implemented using **JWT (JSON Web Token)**.
- **Cloudinary** is used for storing and delivering media files efficiently.

5.2 WebSockets Working

- When a user connects to the chat, a **WebSocket connection** is established between the client and server using **Socket.io**.
- The client can send messages via WebSockets, which are immediately received by the server and broadcasted to other connected users.
- Unlike HTTP requests, WebSockets maintain a persistent connection, reducing overhead and ensuring **real-time communication**.
- Example flow:
    1. User sends a message (socket.emit('sendMessage', message)).
    2. Server listens (socket.on('sendMessage', (message) => { io.emit('message', message) })).
    3. All users receive the message instantly.

5.3 Algorithm for Real-Time Messaging

1. **User Authentication:**
    - User logs in using JWT authentication.
    - Upon successful login, a session token is issued.
2. **Message Transmission:**
    - User sends a message.
    - The frontend emits a sendMessage event through Socket.io.
    - The server receives the event, processes it, and stores the message in MongoDB.
    - The server broadcasts the message to all connected users in the chatroom.

3. **Message Retrieval:**
    - When a user joins a chat, the frontend requests message history.
    - The backend fetches and returns the last N messages from MongoDB.
4. **Typing Indicators & Read Receipts:**
    - Typing events are emitted via WebSockets.
    - Read receipts are updated when messages are seen by recipients.

5.4 JWT Authentication Process

1. **User Registration:**
    - User provides email and password.
    - Password is hashed using **bcrypt** before storing in MongoDB.
2. **User Login:**
    - User submits credentials.
    - Server verifies password and generates a **JWT token**.
    - The token is sent to the client and stored in local storage.
3. **Authorization in API Requests:**
    - The token is included in requests as a header (Authorization: Bearer <token>).
    - The server validates the token and allows access if valid.

5.5 Image Storage in Cloud (Cloudinary)

- Images are uploaded to **Cloudinary** for optimized cloud-based storage and fast delivery.
- Steps for image storage:
    1. **User uploads an image** from the frontend.
    2. **Frontend converts the image** into a base64 format and sends it to the backend.
    3. **Backend uploads the image to Cloudinary** via API (cloudinary.uploader.upload(image, { folder: 'chat_images' })).
    4. **Cloudinary returns a URL**, which is stored in MongoDB.
    5. **Users can access the image via the stored URL**, ensuring quick retrieval via Cloudinary's CDN.

5.6 Scalability Enhancements

- **Database Indexing:** Optimizing MongoDB queries for faster retrieval.

References

[1] WebSockets and event-driven architecture: https://socket.io/docs

[2] Scalable authentication patterns with JWT: https://jwt.io/introduction/

[3] Cloudinary image optimization: https://cloudinary.com/documentation

[5] MongoDB indexing and performance optimization: https://www.mongodb.com/docs/manual/indexes/