

# A Scalable Real-Time Chat System Using MERN Stack with Redis Optimization Sentiment Analysis

Ankush Yeole

*Department of Computer Engineering  
Sinhgad Institute Of Technology  
Pune, India  
ankushby777@gmail.com*

Aditya Panwar

*Department of Computer Engineering  
Sinhgad Institute Of Technology  
Pune, India  
adityapanwar5656@gmail.com*

Ujjwal Singh

*Department of Computer Engineering  
Sinhgad Institute Of Technology  
Pune, India  
ujjwalsingh.sit.comp@gmail.com*

Sakshi Suryawanshi

*Department of Computer Engineering  
Sinhgad Institute Of Technology Pune,  
India  
sakshirs158@gmail.com*

Real-time communication systems are an integral part of modern web applications, requiring low latency, high scalability, and efficient resource utilization. Traditional HTTP-based communication models often fail to meet these requirements due to their request-response nature. This paper presents a scalable real-time chat system developed using the MERN (MongoDB, Express.js, React.js, Node.js) stack integrated with Socket.IO for bidirectional communication.

To enhance system performance, Redis is incorporated as a caching and message brokering layer, supporting session storage, fast data retrieval, and Pub/Sub-based scalability. Additionally, a sentiment analysis module is integrated using an external API to analyze user messages in real time, enabling intelligent insights into user interactions.

The proposed system supports one-to-one and group chat, online/offline user status, message delivery and seen indicators, and a dynamic user interface with dark/light theme support. Experimental evaluation with concurrent users demonstrates improved response time and system efficiency due to Redis-based optimization. The system is designed to be scalable and suitable for handling real-time communication in modern web applications.

**Index Terms**—MERN Stack, Real-Time Chat, Socket.IO, Redis, Sentiment Analysis, WebSockets, Scalable Systems

## I. INTRODUCTION

Real-time communication has become a fundamental requirement in modern web applications, including messaging platforms, collaboration tools, and social networking systems. These applications demand low latency, high scalability, and efficient handling of concurrent users. Traditional HTTP-based architectures rely on a request-response model, which introduces delays and limits real-time interactivity.

With the advancement of WebSocket technologies, frameworks such as Socket.IO enable persistent bidirectional communication between clients and servers, making real-time

messaging systems more efficient. However, challenges such as database bottlenecks, increased latency under high load, and lack of intelligent message processing still persist.

Modern chat systems must also support multiple features such as group communication, user presence tracking, and message status indicators. As the number of users increases, the system must scale efficiently while maintaining performance and reliability. Direct database access for every request can lead to performance degradation, making optimization techniques essential.

To address these challenges, this paper proposes a scalable real-time chat system using the MERN stack integrated with Redis for caching and Pub/Sub-based communication. Redis helps reduce database load, improve response time, and enable horizontal scalability. Additionally, a sentiment analysis module is incorporated using an external API to analyze user messages and provide insights into user interactions.

The proposed system supports one-to-one and group chat, real-time message delivery, online/offline status tracking, and message acknowledgment features. The integration of Redis and Socket.IO ensures efficient communication and improved scalability.

The main contributions of this work are as follows:

- Design and implementation of a scalable real-time chat system using MERN stack.
- Integration of Redis for caching, session management, and Pub/Sub-based scalability.
- Implementation of real-time communication using Socket.IO.
- Incorporation of sentiment analysis for intelligent message processing.
- Development of a feature-rich chat system supporting group chat, user presence, and message tracking.

## II. PROPOSED SYSTEM

The proposed system is a scalable real-time chat application designed using the MERN stack architecture. It focuses on efficient communication, reduced latency, and scalability for handling multiple concurrent users.

The system integrates real-time communication, caching mechanisms, and sentiment analysis into a unified framework. It allows users to communicate seamlessly through one-to-one and group chat while maintaining system performance under load.

The overall workflow of the system is as follows:

- User authentication and session initialization
- Establishment of WebSocket connection using Socket.IO
- Real-time message transmission between clients and server
- Temporary storage and caching of messages using Redis
- Persistent storage of chat data in MongoDB
- Sentiment analysis of messages using external API
- Delivery of messages with status indicators (sent, delivered, seen)

### A. Real-Time Communication Module

Socket.IO is used to enable real-time, bidirectional communication between users. It ensures instant message delivery and supports room-based communication for group chats.

### B. Redis Integration

Redis is used as a high-speed in-memory data store for caching frequently accessed messages, managing user sessions, and enabling Pub/Sub communication for scalability. This reduces database load and improves system performance.

### C. Database Management

MongoDB is used for storing user data, chat history, and message metadata. It ensures persistence and supports flexible schema design for handling dynamic chat data.

### D. Sentiment Analysis Module

The system integrates a sentiment analysis API to classify messages as positive, negative, or neutral. This enhances the system by providing insights into user interactions.

### E. User Interface Features

The frontend is developed using React.js, providing a responsive interface with features such as dark/light theme, real-time updates, and user-friendly interaction.

## III. SYSTEM ARCHITECTURE

The system architecture of the proposed real-time chat application is designed as a scalable and modular client-server model. It integrates multiple components including the frontend interface, backend server, real-time communication layer, caching system, and database to ensure efficient and low-latency communication.

The architecture follows a layered approach, where each component is responsible for a specific functionality, enabling maintainability and scalability.

### A. Architecture Overview

The system consists of the following major components:

- Client Layer (React.js Frontend)
- Application Server (Node.js with Express.js)
- Real-Time Communication Layer (Socket.IO)
- Caching and Messaging Layer (Redis)
- Database Layer (MongoDB)
- Sentiment Analysis Module (External API)

### B. Working of the System

The complete workflow of the system is described as follows:

- 1) The user interacts with the frontend interface developed using React.js.
- 2) HTTP requests such as authentication and data retrieval are handled by the Express.js server.
- 3) A persistent WebSocket connection is established using Socket.IO for real-time communication.
- 4) When a user sends a message, it is transmitted to the server through Socket.IO.
- 5) The server processes the message and forwards it to Redis.
- 6) Redis performs caching and uses Pub/Sub mechanisms to distribute messages across active users.
- 7) The message is stored permanently in MongoDB for future retrieval.
- 8) The sentiment analysis API processes the message and classifies it as positive, negative, or neutral.
- 9) The processed message is delivered instantly to the recipient along with status updates (sent, delivered, seen).

### C. Component Description

**Client Layer (React.js):** The frontend provides an interactive user interface for chat functionality, including one-to-one chat, group chat, user presence indicators, and theme switching. It communicates with the backend through REST APIs and WebSocket connections.

**Backend Server (Node.js and Express.js):** The backend handles API requests, user authentication, session management, and integration with other system components. It acts as the central controller of the application.

**Socket.IO Layer:** Socket.IO enables bidirectional communication between client and server. It supports event-driven messaging and allows the creation of chat rooms for group communication.

**Redis Layer:** Redis is used for caching frequently accessed messages, managing user sessions, and enabling Pub/Sub-based communication. This reduces database load and improves performance under high traffic conditions.

**MongoDB Database:** MongoDB stores user information, chat messages, and metadata. It ensures data persistence and supports scalable storage.

**Sentiment Analysis Module:** An external API is used to analyze the sentiment of messages in real time. This module enhances the system by providing insights into user communication patterns.

#### D. Scalability Considerations

The system is designed to handle multiple concurrent users efficiently. Redis Pub/Sub enables horizontal scaling by distributing messages across multiple server instances. The use of caching reduces database queries, improving overall system performance.

The modular architecture ensures that individual components can be scaled independently, making the system suitable for large-scale real-time applications.

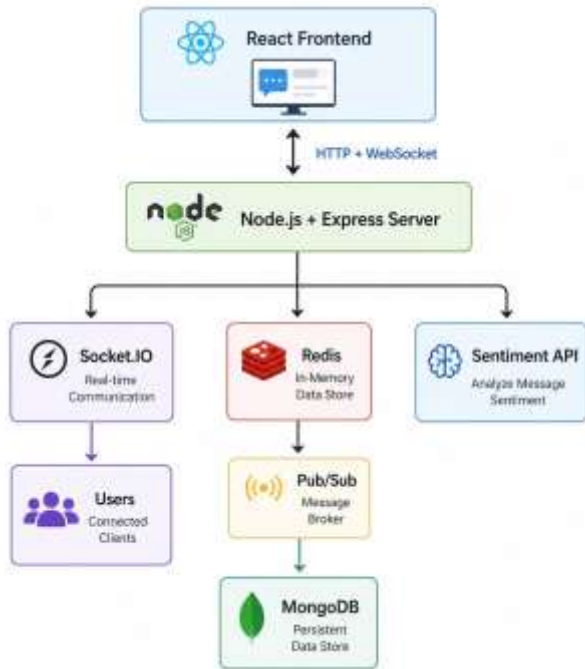


Fig. 1. System Architecture of the Proposed Chat Application

## IV. METHODOLOGY

The proposed system follows a structured methodology to design and implement a scalable real-time chat application. The methodology focuses on efficient data flow, real-time communication, and performance optimization using caching mechanisms.

#### A. System Design Approach

The system is designed using a client-server architecture based on the MERN stack. The frontend handles user interaction, while the backend manages data processing, communication, and storage. Real-time communication is achieved using WebSocket technology through Socket.IO.

#### B. Message Flow Process

The message transmission process follows a sequence of steps:

- 1) The user sends a message through the React-based frontend interface.
- 2) The message is transmitted to the backend server using Socket.IO.

- 3) The backend processes the message and temporarily stores it in Redis cache.
- 4) Redis Pub/Sub mechanism broadcasts the message to connected users.
- 5) The message is then stored permanently in MongoDB.
- 6) A sentiment analysis API is invoked to classify the message.
- 7) The processed message is delivered to the recipient along with status updates.

#### C. Caching Strategy

Redis is used to reduce latency and database load by caching frequently accessed data such as recent messages and active user sessions. This ensures faster retrieval and improves system responsiveness.

#### D. Scalability Strategy

To support multiple concurrent users, the system utilizes Redis Pub/Sub for distributing messages across different instances. This allows horizontal scaling of the backend server without affecting performance.

#### E. Sentiment Analysis Integration

The system integrates an external sentiment analysis API to process chat messages. Each message is classified into predefined categories such as positive, negative, or neutral, enabling intelligent monitoring of user interactions.

#### F. User Interaction Model

The application supports multiple user interaction features, including one-to-one chat, group chat, user presence tracking, and message acknowledgment. These features enhance usability and provide a complete real-time communication experience.

## V. IMPLEMENTATION

The proposed real-time chat system is implemented using the MERN stack, integrating multiple technologies to ensure efficient communication, scalability, and performance optimization.

#### A. Frontend Implementation

The frontend is developed using React.js, providing a dynamic and responsive user interface. It supports features such as one-to-one chat, group chat, online/offline status, message delivery indicators, and theme switching (dark/light mode).

State management is handled efficiently to ensure real-time updates. Socket.IO client is integrated to establish a persistent connection with the backend server for instant message delivery.

### B. Backend Implementation

The backend is implemented using Node.js and Express.js. It handles API requests, user authentication, session management, and real-time communication logic. The server is responsible for managing user connections and routing messages between clients.

Socket.IO is integrated into the backend to enable event-driven communication. Each user is assigned a unique socket ID, and chat rooms are created for group communication.

### C. Database Implementation

MongoDB is used as the primary database for storing user information, chat messages, and metadata. Collections are designed to efficiently handle message storage and retrieval. Indexing techniques are used to optimize query performance.

### D. Redis Integration

Redis is implemented as an in-memory data store to improve system performance. It is used for:

- Caching frequently accessed messages
- Managing user sessions
- Implementing Pub/Sub mechanism for real-time message distribution

The use of Redis significantly reduces database queries and improves response time, especially under high user load.

### E. Real-Time Communication

Socket.IO is used to establish a WebSocket-based communication channel between clients and the server. It supports real-time messaging, event handling, and room-based communication for group chats.

Events such as message send, receive, typing indicators, and message status updates are handled efficiently using Socket.IO.

### F. Sentiment Analysis Integration

The system integrates an external sentiment analysis API to process chat messages in real time. The API returns sentiment labels such as positive, negative, or neutral, which are then associated with each message.

This feature enhances the system by enabling intelligent insights into user communication patterns.

### G. Performance Considerations

The system is tested with concurrent users to evaluate its performance. Redis caching and Pub/Sub mechanisms significantly reduce latency and improve scalability. The architecture ensures efficient handling of multiple users while maintaining real-time communication.

## VI. RESULTS AND ANALYSIS

This section presents the performance evaluation of the proposed real-time chat system. The system is analyzed based on latency, response time, and scalability under concurrent user load. The impact of Redis caching on system performance is also evaluated.

### A. Experimental Setup

The system is tested in a controlled environment with multiple concurrent users connected to the server. The backend is implemented using Node.js and Socket.IO, while MongoDB is used for persistent storage. Redis is integrated for caching and Pub/Sub communication.

The evaluation is conducted by simulating up to 100 concurrent users performing real-time messaging operations.

### B. Latency Analysis

Latency is defined as the time taken for a message to be delivered from sender to receiver. The system is tested under two conditions: with Redis and without Redis.

TABLE I  
LATENCY COMPARISON WITH AND WITHOUT REDIS

| Number of Users | Without Redis (ms) | With Redis (ms) |
|-----------------|--------------------|-----------------|
| 20              | 120                | 60              |
| 50              | 220                | 110             |
| 100             | 400                | 180             |

The results indicate that Redis significantly reduces latency by minimizing direct database access and enabling faster message propagation.

### C. Response Time Analysis

Response time refers to the time taken by the server to process a message request and send acknowledgment to the user.

TABLE II  
RESPONSE TIME ANALYSIS

| Number of Users | Response Time (ms) |
|-----------------|--------------------|
| 20              | 80                 |
| 50              | 150                |
| 100             | 250                |

The response time increases with the number of users, but remains within acceptable limits due to efficient handling using Socket.IO and Redis.

### D. Scalability Analysis

The system demonstrates stable performance with increasing user load. The integration of Redis Pub/Sub allows efficient distribution of messages across users, ensuring that the system scales effectively.

Even at 100 concurrent users, the system maintains real-time communication without significant delays or message loss.

### E. Sentiment Analysis Evaluation

The sentiment analysis module classifies messages into positive, negative, and neutral categories. The integration of the external API provides reliable classification without affecting system performance.

The sentiment processing time is minimal and does not introduce noticeable delay in message delivery.

### F. Graphical Analysis

The performance improvements can be visualized using graphs:

- **Latency vs Number of Users Graph:** This graph shows that latency increases with the number of users, but the increase is significantly lower when Redis is used.
- **Response Time Graph:** This graph illustrates that response time grows gradually with user load, indicating efficient server handling.
- **Performance Comparison Graph:** A comparison graph highlights the difference between systems with and without Redis, clearly showing improved efficiency in the proposed system.

### G. Discussion

The results demonstrate that the integration of Redis improves system performance by reducing latency and optimizing resource usage. Socket.IO ensures efficient real-time communication, while MongoDB provides reliable data storage.

The system successfully handles concurrent users and maintains consistent performance, making it suitable for scalable real-time applications.

## VII. DISCUSSION

The experimental results demonstrate that the proposed real-time chat system achieves significant improvements in performance and scalability through the integration of Redis and Socket.IO within the MERN architecture.

One of the key observations from the results is the reduction in latency when Redis is utilized. The caching mechanism minimizes direct database access, which is typically a bottleneck in real-time systems. By storing frequently accessed data in memory, Redis enables faster retrieval and reduces the response time for message delivery. This becomes increasingly important as the number of concurrent users grows.

The use of Redis Pub/Sub further enhances system scalability by enabling efficient message distribution across multiple active connections. Instead of relying solely on the backend server for message routing, Redis acts as a message broker, ensuring that messages are propagated quickly and efficiently. This design makes the system suitable for horizontal scaling in distributed environments.

Socket.IO plays a crucial role in maintaining real-time communication. Its event-driven architecture ensures instant message delivery and supports features such as group chat and message status updates. The combination of Socket.IO and Redis creates a robust communication layer capable of handling concurrent users without significant degradation in performance.

The integration of sentiment analysis adds an intelligent dimension to the system. Although it does not directly impact system performance, it enhances the functionality by providing insights into user interactions. The use of an external API ensures accurate classification while keeping the system lightweight.

Despite these advantages, certain limitations are observed. The system has been evaluated up to 100 concurrent users, and performance under larger-scale deployments requires further testing. Additionally, reliance on an external API for sentiment analysis introduces dependency on network availability and third-party services.

Overall, the results indicate that the proposed system effectively balances performance, scalability, and functionality. The integration of caching, real-time communication, and intelligent processing makes it a practical solution for modern web-based chat applications.

## VIII. CONCLUSION

This paper presents a scalable real-time chat system developed using the MERN stack, integrating Socket.IO for real-time communication, Redis for caching and message brokering, and an external API for sentiment analysis. The system addresses key challenges in modern communication applications, including latency, scalability, and efficient data handling.

The implementation demonstrates that the use of Redis significantly reduces message delivery latency and minimizes database load by caching frequently accessed data and enabling Pub/Sub-based communication. Socket.IO ensures reliable and low-latency bidirectional communication, supporting essential features such as one-to-one chat, group chat, user presence tracking, and message status indicators.

Experimental evaluation with concurrent users shows that the system maintains stable performance and efficient message delivery under load. The integration of sentiment analysis further enhances the system by providing intelligent insights into user interactions without affecting overall performance.

Overall, the proposed system provides a robust, scalable, and efficient solution for real-time communication, making it suitable for modern web-based applications requiring high responsiveness and reliability.

## IX. FUTURE SCOPE AND SCALABILITY OF PROPOSED SYSTEM

Scalability is a critical requirement for real-time communication systems, especially when handling a large number of concurrent users. The proposed system is designed with scalability in mind by incorporating efficient architectural and technological choices.

The use of Redis Pub/Sub mechanism allows the system to distribute messages across multiple server instances, enabling horizontal scaling. This ensures that as the number of users increases, additional server instances can be added without affecting system performance.

Caching using Redis reduces the dependency on MongoDB for frequent data access, thereby minimizing database load and improving response time. This design ensures that the system can handle increased traffic efficiently.

Socket.IO supports room-based communication and event-driven messaging, which allows efficient management of multiple user connections simultaneously. This ensures that the

system can scale to support group communication and high user activity.

Additionally, the modular architecture of the MERN stack enables independent scaling of frontend and backend components. The backend server can be deployed in a distributed environment, while the frontend can be served through content delivery networks (CDNs) for better performance.

Overall, the proposed system provides a scalable architecture capable of handling increased user load while maintaining low latency and efficient real-time communication .

#### REFERENCES

- [1] Socket.IO Documentation, "Real-time bidirectional event-based communication," [Online]. Available: <https://socket.io/>
- [2] Redis Documentation, "In-memory data structure store," [Online]. Available: <https://redis.io/>
- [3] MongoDB Inc., "MongoDB: The application data platform," [Online]. Available: <https://www.mongodb.com/>
- [4] Meta Platforms, "React – A JavaScript library for building user interfaces," [Online]. Available: <https://reactjs.org/>
- [5] Node.js Foundation, "Node.js JavaScript runtime," [Online]. Available: <https://nodejs.org/>
- [6] I. Fette and A. Melnikov, "The WebSocket Protocol," IETF RFC 6455, 2011.
- [7] J. Carlson, "Redis in Action," Manning Publications, 2013.
- [8] U. Hossain and M. Rahman, "Development of Web Applications using MERN Stack," International Journal of Computer Applications, 2020.
- [9] S. Pimentel and M. Nickerson, "Communicating and Displaying Real-Time Data with WebSocket," IEEE Internet Computing, vol. 16, no. 4, pp. 45–53, 2012.
- [10] B. Liu, "Sentiment Analysis and Opinion Mining," Morgan Claypool Publishers, 2012.
- [11] A. Zhang et al., "Deep Learning for Natural Language Processing: Advances and Applications," IEEE, 2018.
- [12] G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-value Store," ACM Symposium on Operating Systems Principles, 2007.
- [13] C. Strauch, "NoSQL Databases," Stuttgart Media University, 2011.
- [14] M. Kleppmann, "Designing Data-Intensive Applications," O'Reilly Media, 2017.