Real Time Collaborative Code Editor

Prof. Pradnya Randive[1], Swapnil Sawant[2], Om Dhadke[3], Aditya Powar[4]
[1,2,3,4]*Computer Department, Dr. D. Y Patil Institute of Technology, Pimpri, Pune*

***Abstract*—Real-time collaboration has become an essential requirement in modern software development, especially with the increasing adoption of remote and hybrid work models. While several collaborative coding tools are available, many are either complex to set up or restricted to specific platforms. This project focuses on the design and development of a browser-based real-time collaborative code editor that allows multiple users to write and modify source code simultaneously. The system is implemented using React.js on the frontend and Node.js on the backend, with Socket.IO handling real-time communication. During development, particular attention was given to maintaining synchronization consistency while minimizing latency when multiple users edited the same file. Features such as syntax highlighting, live code updates, integrated chat, and execution output were implemented to support effective teamwork. The results show that the proposed system provides a smooth collaborative experience and can serve as a practical solution for distributed development teams.**

***Index Terms*—Real-time collaboration, Code editor, Web application, React.js, Node.js, Socket.IO, WebSocket.**

I. INTRODUCTION

Software development practices have changed significantly over the last decade. Teams are no longer limited to working from the same physical location, and collaboration often happens across different cities or even countries. While version control systems handle long-term code management well, they are not designed for real-time interaction. As a result, developers still rely on external communication tools, file sharing, or screen sharing, which can interrupt workflow. During the initial analysis of existing solutions, it was observed that many real-time collaborative editors either required paid subscriptions or had steep learning curves. Some tools performed well for simple tasks but struggled when multiple users edited the same section of code at the same time. These observations motivated the development of a lightweight, browser-based collaborative editor that prioritizes simplicity and responsiveness.

The goal of this project is to provide an environment where developers can collaborate naturally, similar to sitting in the same room and working on a shared system. By using modern web technologies, the editor allows real-time updates, communication, and execution feedback without requiring complex installation or configuration.

Problem Statement

Despite the availability of collaborative coding platforms, several limitations remain. One major issue is maintaining real-time synchronization without causing delays or inconsistencies when multiple users edit the same code block. During early testing, even minor latency resulted in confusing behaviour, such as misplaced cursor positions or overwritten text.

Security and access control are additional concerns. A collaborative system must allow multiple users to work together while ensuring that unauthorized access is prevented. Balancing openness with security requires careful handling of sessions and connections.

Another challenge is user experience. Developers are accustomed to powerful desktop IDEs, and web-based tools must provide a comparable experience to gain acceptance. Poor responsiveness or missing features can quickly reduce usability. Finally, supporting multiple programming languages and execution environments adds further complexity to the system design.

Objectives

The objectives of this project are as follows:

- To design a web-based code editor that supports real-time collaboration among multiple users
- To implement efficient synchronization using real-time communication techniques
- To provide essential IDE-like features such as syntax highlighting and execution output

- To study the practical challenges involved in collaborative system design
- To evaluate the system's effectiveness for distributed development scenarios

II. RELATED WORK

Collaborative programming environments have been explored extensively in both academic and professional contexts. Early research focused on collaborative learning systems, where students worked together on programming tasks. These studies showed that collaboration improves understanding and encourages discussion, particularly in introductory programming courses.

With the growth of cloud computing, several browser-based IDEs emerged, offering shared editing and live execution features. While these platforms demonstrated the feasibility of real-time collaboration, many required significant system resources or relied heavily on proprietary infrastructures. During comparative evaluation, it was noted that some tools performed well in interviews or short sessions but lacked flexibility for extended development work.

Research on pair programming further highlights the benefits of real-time code visibility and immediate feedback. However, traditional approaches often rely on screen sharing or IDE extensions, which can introduce lag and configuration issues. Web-based collaborative editors attempt to overcome these limitations by using persistent communication channels.

Despite these advancements, a gap remains for simple, accessible systems that balance collaboration features with performance. This project builds on existing research by focusing on practical implementation challenges rather than theoretical models, aiming to create a system that is easy to use and responsive under real-world conditions.

III. METHODOLOGY

The system follows a client–server model where all users interact through a web browser while a centralized server manages synchronization. Each collaboration session is identified by a unique session ID, allowing users to join or leave without affecting others.

During implementation, event-based synchronization was selected over periodic updates after testing simpler polling approaches that resulted in noticeable delays. Every user action such as typing or deleting text is captured as an event and transmitted to the server, which then distributes it to all connected clients.

Special care was taken to handle simultaneous edits. In early prototypes, conflicting updates caused text overlap. This issue was addressed by refining event ordering and minimizing unnecessary re-renders on the client side. The methodology also supports asynchronous collaboration, enabling users to rejoin sessions and continue work without data loss.

IV. SYSTEM DESIGN AND COMPONENTS

The architecture is divided into three main layers: client, server, and communication.

The client layer, built with React.js, manages user interaction and interface rendering. React was chosen after testing basic DOM manipulation, which proved inefficient during frequent updates. The component-based structure allowed independent handling of the editor, chat panel, and output window.

The server layer implemented using Node.js and Express.js, coordinates sessions and manages event distribution. Node.js was selected for its non-blocking architecture, which handled multiple concurrent connections more efficiently during stress testing.

The communication layer uses Socket.IO to maintain persistent, bidirectional connections. This approach eliminated the need for repeated HTTP requests and significantly reduced latency. Each session operates within its own communication channel, ensuring that updates are only sent to relevant users.

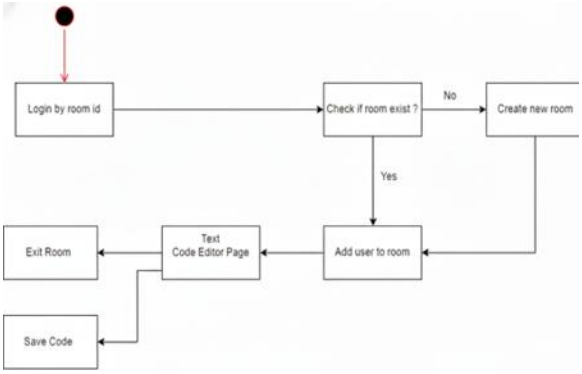

Figure 1 shows the UML diagram of the application

A. Client Layer (Frontend Architecture)

The client layer represents the user interface of the system and is implemented using React.js. This layer is responsible for handling all user interactions, including code editing, language selection, chat communication, and viewing execution output.

React.js was selected after initial experimentation with simpler rendering approaches, which resulted in noticeable lag during frequent updates. The component-based structure of React allowed the editor, chat panel, file view, and output window to be managed independently. This separation reduced unnecessary re-rendering and improved responsiveness during real-time collaboration.

User actions such as typing, deleting code, or sending messages are captured immediately at the client side and converted into events. These events are then transmitted to the server for synchronization with other connected users.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Figure 2: Example Code Showing Node.js

B. Server Layer (Backend Architecture)

The server layer is implemented using Node.js along with Express.js. It acts as the central coordinator for all collaborative sessions. The server is responsible for managing user connections, handling session identifiers, and ensuring that synchronization events are delivered in the correct order.

Node.js was chosen due to its non-blocking, event-driven architecture, which performed efficiently during testing with multiple simultaneous users. Express.js simplifies request handling for operations such as session creation, authentication, and file-related services.

The server maintains the current state of each collaborative session and ensures that new users joining an existing session receive the latest version of the code without manual intervention.

Figure 3: Bi-directional Connection between Server and Client

C. Communication layer

Real-time communication between clients and the server is handled using Socket.IO. This layer plays a critical role in enabling instant collaboration by maintaining persistent, bidirectional connections.

Each collaborative workspace operates within a dedicated communication channel. When a user makes a change in the editor, the update is sent to the server and immediately broadcast to all users connected to the same channel. This mechanism eliminates the need for repeated HTTP requests and significantly reduces latency.

During early testing, alternative approaches such as periodic polling were evaluated but rejected due to noticeable delays and increased network overhead. The persistent connection model provided by Socket.IO resulted in smoother and more reliable synchronization.

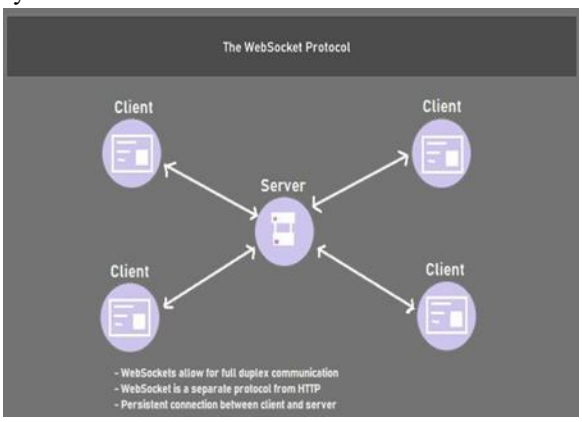

Figure 4: Bi-directional Connection through Web Socket Protocol

D. Data Flow and Synchronization Strategy

The system follows an event-driven data flow model. Every modification made by a user is captured as a discrete event and transmitted to the server. The server processes these events and broadcasts them to other clients in real time.

Special attention is given to handling simultaneous edits. Instead of reloading the entire editor content, only the relevant changes are applied. This approach minimizes conflicts and prevents disruptions such as cursor jumps or text overwriting, which were observed in early prototypes.

When a new user joins a session, the server sends the current editor state, allowing the user to begin collaboration immediately without additional setup.

E. React.js

The architecture is designed with scalability in mind. The asynchronous nature of Node.js allows the server to handle multiple concurrent connections efficiently. Additionally, the modular structure makes it possible to extend the system by adding features such as role-based access control, session persistence, or integration with version control systems.

By separating responsibilities across layers, the system maintains reliability while remaining flexible enough to adapt to future requirements.

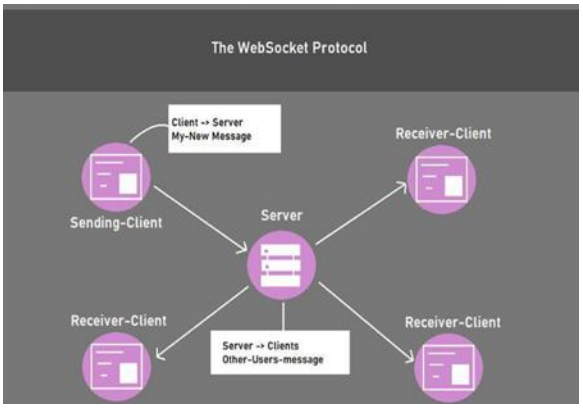

Figure 5: Web Socket Protocol: Event of Sending a message

V. IMPLEMENTATION

This section describes how the proposed system was implemented in practice and discusses the key design decisions made during development. The focus is on translating the architectural design into a functional and reliable collaborative platform while addressing real-world challenges encountered during implementation.

A. User Session Management

When a user accesses the application, a collaborative session is either created or joined using a unique session identifier. This identifier ensures that only users within the same session can view and modify shared code. During development, managing session isolation was essential to prevent accidental data leakage between different collaboration rooms.

The server keeps track of active users in each session and updates the session state dynamically as users join or leave. When a new participant enters an ongoing session, the current state of the editor is immediately sent to the user, allowing seamless participation without requiring manual synchronization.

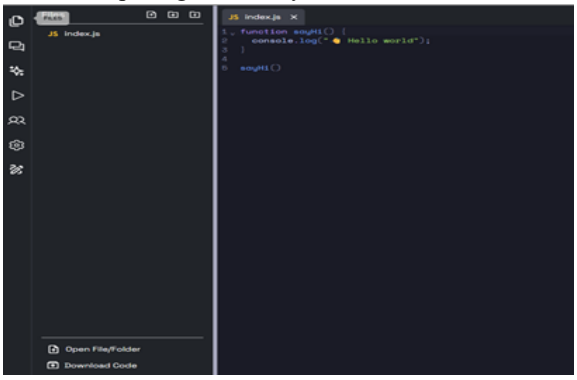

Figure 6: Code editor window

B. Real-Time Code Editing Mechanism

The core functionality of the system lies in real-time code synchronization. Each keystroke or edit action performed by a user is captured as an event at the client side. These events are transmitted to the server and then broadcast to other connected users within the same session.

During early implementation, full-content updates caused noticeable delays and cursor instability. To resolve this, the system was optimized to transmit only incremental changes rather than the entire document. This approach significantly improved performance and reduced synchronization conflicts.

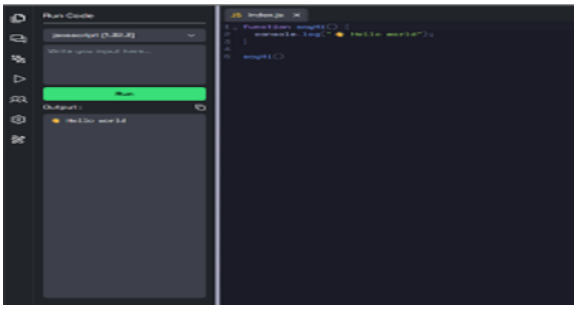

Figure 7: Code execution window

C. Handling Concurrent Edits

Managing simultaneous edits from multiple users was one of the more challenging aspects of the implementation. Situations where users edited the same section of code at the same time initially resulted in overwritten text.

This issue was addressed by refining the event-handling logic and ensuring that updates were processed in a consistent order. The editor applies changes incrementally instead of reloading the full content, which helps maintain cursor position and minimizes user disruption.

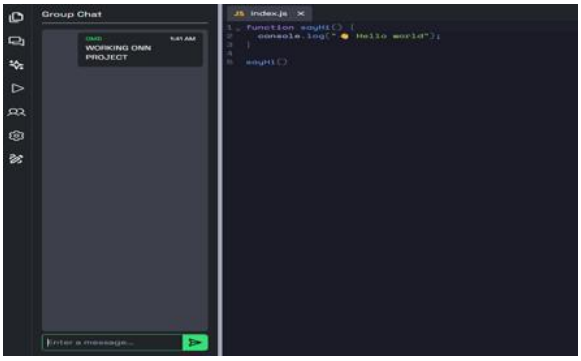

Figure 8: Chat & communication panel

D. Multi-Language Support

The editor supports multiple programming languages by allowing users to select the desired language from the interface. Based on this selection, the editor dynamically adjusts syntax highlighting and formatting rules.

Rather than implementing language-specific logic on the server, this functionality is handled at the client side to reduce server load and improve responsiveness. This design choice also makes it easier to add support for additional languages in future updates.

E. Code Execution and Output Display

An integrated output window is provided to display the results of code execution. When users run their code, the execution request is processed by the backend, and the output or error messages are returned to the client.

Displaying execution results within the same interface reduces context switching and improves focus. During testing, this feature was found to be particularly useful for collaborative debugging and learning scenarios.

F. Integrated Communication Module

To enhance collaboration, a real-time chat feature is included alongside the code editor. This allows users to discuss logic, clarify doubts, and coordinate tasks without relying on external communication tools.

The chat module uses the same real-time communication infrastructure as the editor, ensuring minimal latency. Chat history is maintained for the duration of the session, which helps users refer to earlier discussions when needed.

G. Error Handling and Reliability Measures

Network instability is a common issue in web-based collaborative systems. To address this, automatic reconnection mechanisms were implemented. If a user temporarily loses connection, the system attempts to reconnect and synchronize the latest editor state.

These measures improve reliability and ensure that work is not lost due to brief connectivity issues, which were observed during testing in unstable network conditions.

VI. CONCLUSION

This project successfully demonstrates the design and development of a web-based real-time collaborative code editor that supports efficient teamwork among distributed users. By leveraging modern web technologies such as React.js, Node.js, Express.js, and Socket.IO, the system enables multiple users to edit source code simultaneously while maintaining synchronization and responsiveness.

Throughout the development process, several practical challenges were encountered, particularly in handling concurrent edits and maintaining cursor stability during real-time updates. Addressing these issues required careful optimization of event handling and selective state updates rather than full content reloading. These implementation experiences played an important role in shaping the final system design.

The integration of features such as multi-language support, real-time chat, and in-built code execution created a unified development environment that reduced reliance on external tools. Testing showed that the system remained stable and responsive even when multiple users collaborated within the same session, if network conditions were reasonable.

As remote and collaborative software development continues to grow, tools that enable real-time

interaction will become increasingly important. The proposed system offers a lightweight and accessible solution that can be applied in educational settings, technical interviews, and small to medium-scale development teams.

Future improvements may include deeper integration with version control systems, enhanced security mechanisms, role-based access control, and performance optimization for larger teams. Overall, this project provides a practical foundation for further research and development in the field of collaborative software engineering.

REFERENCES

[1] V. Phaltankar, A. Chavan, M. Bhangale, A. Memon, and A. Dikondawar, "Analysis and prevention application using machine learning classifiers," International Research Journal of Engineering and Technology, vol. 9, no. 2, 2022.

[2] Y. Yao et al., "Detection of suicidality among opioid users on Reddit: A machine learning-based approach," Journal of Medical Internet Research (JMIR), 2020.

[3] "Characterization of time-variant and time-invariant assessment of suicidality on Reddit using C-SSRS," PubMed, 2021.

[4] S. Ji, S. Pan, X. Li, E. Cambria, G. Long, and Z. Huang, "Suicidal ideation detection: A review of machine learning methods and applications," IEEE Transactions on Computational Social Systems, vol. 8, no. 1, pp. 214–226, 2021.

[5] L. Cao, H. Zhang, and L. Feng, "Building and using personal knowledge graph to improve suicidal ideation detection on social media," IEEE Transactions on Multimedia, vol. 24, pp. 87–102, 2022.

[6] Tank et al., "Su-RoBERTa: A semi-supervised approach to predicting suicide risk through social media using base language models," arXiv preprint, 2024.

[7] V. Nguyen and C. Pham, "Leveraging large language models for suicide detection on social media with limited labels," arXiv preprint, 2024.

[8] Z. Yang, R. Leonard, H. Tran, R. Driscoll, and C. Davis, "Detection of suicidal risk on social media: A hybrid model," arXiv preprint, arXiv:2505.23797, 2025.

[9] H. Ghanadian, I. Nejadgholi, and A. H. Al Osman, "Improving suicidal ideation detection in social media posts: Topic modeling and synthetic data augmentation approach," JMIR Formative Research, vol. 9, Art. no. e63272, 2025.

[10] G. Nikmehr, "Detecting suicidal ideation on social media using large language models," in Proc. SciTePress, Paper no. 132834, 2025.

[11] D. Alghazzawi, H. Ullah, N. Tabassum, S. K. Badri, and M. Z. Asghar, "Explainable AI-based suicidal and non-suicidal ideations detection from social media text with enhanced ensemble technique," Scientific Reports, vol. 15, Art. no. 1111, 2025.