Real-Time Collaborative Code Editing and Its Impact on Software Development

by

Md. Hasnat Shahriyar
ID: CSE2101022105

Nasrin Akter
ID: CSE2101022147

Sumon Das
ID: CSE2102023056

Md Jakir Hosain
ID: CSE2102023063

Supervised by
Mohammad Naderuzzaman

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SONARGAON UNIVERSITY (SU)
May 2025

APPROVAL

The project titled **"A Comprehensive Study on Real-Time Collaborative Code Editing Platform and Its Impact on Software Development"** submitted by Md. Hasnat Shahriyar (CSE2101022105), Nasrin Akter (CSE2101022147), Sumon Das (CSE2102023056), and Md Jakir Hosain (CSE2102023063) to the Department of Computer Science and Engineering, Sonargaon University (SU), has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering and approved as to its style and contents.

Board of Examiners

Supervisor

Mohammad Naderuzzaman

Associate Professor,

Department of Computer Science and Engineering

Sonargaon University (SU)

(Examiner Name and Signature)

Department of Computer Science and Engineering

Examiner 1

Sonargaon University (SU)

(Examiner Name and Signature)

Department of Computer Science and Engineering

Examiner 2

Sonargaon University (SU

(Examiner Name and Signature)

Department of Computer Science and Engineering

Examiner 3

Sonargaon University (SU)

DECLARATION

We, hereby, declare that the work presented in this report is the outcome of the design, development, and implementation of a real-time collaborative code editing platform carried out by us under the supervision of **Mohammad Naderuzzaman, Associate Professor,** Department of Computer Science and Engineering, Sonargaon University, Dhaka, Bangladesh. We reaffirm that no part of this project has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned

(Mohammad Naderuzzaman)

Supervisor

Signature

Md. Hasnat Shahriyar

ID: CSE2101022105

Nasrin Akter

ID: CSE2101022147

Sumon Chandra Das

ID: CSE2102023056

Md. Jakir Hosain

ID: CSE2102023063

ABSTRACT

With the evolution of remote work, collaborative learning, and global software teams, the need for real-time, cloud-based development environments has significantly increased. This project explores the development of **CodeLink**, a real-time collaborative code editing platform that enables multiple users to write, edit, and execute code simultaneously in a shared environment. Built using **React.js, Node.js, Socket.IO,** and **Monaco Editor**, the system features syntax highlighting, live code synchronization, user session management, and language-specific code execution capabilities for JavaScript, Python, and C++. The report provides an in-depth analysis of the platform's architecture, design considerations, challenges, and performance during collaborative sessions. This study also discusses the broader implications of such tools in modern software development, focusing on productivity, remote collaboration, and education. The goal of this project is to demonstrate the significance and feasibility of building efficient, scalable, and user-friendly real-time collaboration tools for developers and educators alike.

ACKNOWLEDGMENT

At the very beginning, we express our deepest gratitude to Almighty Allah for giving us the strength, ability, and patience to complete this project successfully.

We are sincerely thankful to our supervisor, **Mohammad Naderuzzaman**, **Associate Professor**, Department of Computer Science and Engineering, Sonargaon University, whose expert guidance, insightful feedback, and constant encouragement steered us throughout the project. His mentorship helped us develop both the technical and analytical skills necessary for this work.

We are also grateful to all faculty members of the Department of Computer Science and Engineering at Sonargaon University who contributed to our learning and supported us academically.

Lastly, we extend our heartfelt thanks to our families for their unwavering support, encouragement, and love throughout this journey.

LIST OF ABBREVIATIONS

API	Application Programming Interface
CDN	Content Delivery Network
CPU	Central Processing Unit
CRDT	Conflict-Free Replicated Data Type
CSS	Cascading Style Sheets
DOM	Document Object Model
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JS	JavaScript
JSX	JavaScript XML
OT	Operational Transformation
RAM	Random Access Memory
SaaS	Software as a Service
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience

TABLE OF CONTENTS

Title Page No.

DECLARATION ……..…….. iii

ABSTRACT ……...……. iv

ACKNOWLEDGEMENT ……..…… v

LIST OF ABBREVIATION ……..……. vi

CHAPTER 1 1 – 4

INTRODUCTION TO REAL-TIME COLLABORATIVE CODE EDITING PLATFORM

1.1 Introduction ……...……..... 1

1.2 Problem Statement …….……...……... 2

1.3 Objetives ……...…….…… 2

1.4 Scope ……..……... 3

1.5 Significance of Real-Time Collaboration in Modern Development … 3

1.6 Importance of Web-Based Architecture in Modern Software Delivery 4

1.7 Academic Relevance and Educational Benefits …….…….…. 4

1.8 Alignment with Industry Trends and Tools …….. 4

1.9 Relevance to Open Source and Developer Communities …… 4

CHAPTER 2 5 – 9

RELATED WORK AND BACKGROUND

2.1 Collaborative Editing Models (OT vs. CRDT).....…….…..... 5

2.2 Existing Platforms (Google Docs, Replit, VS Live Share, CodePen) .. 5

2.3 Gap Analysis & How CodeLink Differs …….…… 6

2.4 Role of WebSockets in Real-Time Systems ……..… 7
2.5 Browser-Based IDEs and the Monaco Editor …….. 7
2.6 Benefits of Socket.IO Over Raw WebSockets ……. 8
2.7 Lightweight Frameworks and Rapid Development with Express …… 8
2.8 Use of React for Dynamic Frontend State Management …….. 8
2.9 Integration of Node.js for Event-Driven Server Logic …… 9

CHAPTER 3 10 – 16

SYSTEM DESIGN AND ARCHITECTURE

3.1 Overall Architecture (Client-Server, Socket.IO Rooms) 10
3.2 Technology Stack (React, Node.js/Express, Socket.IO, Monaco Editor, Tailwind) ……..……..…… 11
3.3 Data Flow & Event Model (join, code_change, sync_code, request_sync, leave, etc.) 12
3.4 Database & Persistence 13
3.5 Session Coordination and Socket Lifecycle …… 13
3.6 Scalability and Load Distribution Considerations …… 14
3.7 Security and Transport Protocol Design ……... 14
3.8 Language Mode Integration and Syntax Support ……. 15
3.9 Event Reliability and Message Acknowledgement …….. 15

CHAPTER 4 17 – 23

IMPLEMENTATION DETAILS

4.1 Front-end Components (EnterRoomCard, EditorPage, Navbar, Footer, Timeline) …….. 17

4.2 Shared Actions Module (ACTIONS/Actions.js) ……. 19
4.3 Socket Communication Protocol (Events, Handlers) …….. 20
4.4 Code Execution Service (‘/run’ Endpoint, child_process.exec) .. 22
4.5 User Authentication & Session Management …….. 23

CHAPTER 5 24 – 26
TESTING AND EVALUATION
5.1 Functional Testing (multi-client sync, undo/redo) …… 24
5.2 Performance Testing (latency <100 ms, load tests) ……... 24
5.3 Usability & User Feedback (student surveys) ……. 25
5.4 Security Considerations ……... 26

CHAPTER 6 27 – 28
CONCLUSION AND FUTURE WORK
6.1 Conclusion ……..….. 27
6.2 Future Work ……...…….…… 27
6.3 Contribution to Learning and Development 28

REFERENCES ……..…….. 29 – 30

LIST OF FIGURES

Figure No.	Title	Page No.
Fig.1	CodeLink room entry interface	17
Fig.2	Basic structure of articulatory feature-based ASR systems	18

CHAPTER 1

INTRODUCTION TO REAL-TIME COLLABORATIVE CODE EDITING PLATFORM

1.1 Introduction

Real-time collaborative code editors enable multiple programmers to work on the same codebase simultaneously, replicating the ease of cloud-based document collaboration for software development. **CodeLink** is designed to provide this functionality, allowing students and developers to code together from different locations in real time. By using CodeLink, team members no longer need to merge changes manually or send code back and forth; instead, every user's edits appear instantly in all collaborators' editors, reducing integration overhead and enabling synchronous teamwork.

Collaborative coding has become increasingly important in modern software development and education. With the rise of remote work and distributed teams, developers often need to write and debug code together from different locations. Real-time collaborative editors enable multiple users to edit the same code simultaneously, improving productivity and learning. Platforms like Sync Code Studio demonstrate the viability of such systems: "a real-time collaborative code editing platform, where multiple users can join a room and edit code together"[05]. In education, pair programming and live code review tools help students learn from one another, and instructors guide classes interactively. CodeLink is motivated by this need for seamless collaboration. It aims to provide an intuitive interface where programmers and learners can join sessions by room ID and work together on code in real time, sharing live edits and running code collectively.

Collaborative editors must maintain consistency across clients. For example, Google Docs allows concurrent text editing without conflicts[08]. CodeLink applies similar principles to code editing. By keeping each client's view of the code synchronized, all participants always see the same document state. Technologies like Socket.IO support efficient real-time updates between clients and the server to achieve this[14].

Real-time collaborative coding platforms are becoming more essential in the context of the growing demand for remote education and distributed workforces. As programming becomes a more collaborative effort, tools like CodeLink are at the forefront of enabling seamless, distributed software development. Real-time editing, when coupled with execution environments, can bring collaboration into the hands of every developer, no matter their geographical location.

Our motivation is to leverage modern web technologies (React, Node.js, WebSockets) to build a lightweight, easy-to-use platform specifically tailored for collaborative coding. This platform will include features like multi-language execution and an extensible UI.

Furthermore, CodeLink's integration of Monaco Editor provides a streamlined and extensible user experience for coding in various languages[15].

1.2 Problem Statement

Traditional development environments (IDEs or single-user editors) lack built-in real-time collaboration. Programmers typically share code via version control or screen sharing tools, which are not ideal for live editing. Existing collaborative tools like Google Docs excel at text, but code requires syntax highlighting, execution, and precise editing semantics. While platforms such as Replit and CodePen offer some collaboration, they can be limited by subscription models or not focused on collaborative editing features. There is a need for a custom platform that allows instant shared editing of code, maintains low latency, and provides code execution in-browser.

CodeLink addresses this gap by providing a self-hosted solution where users enter a room ID and immediately see each other's edits in real time. The system must handle concurrency (e.g., simultaneous keystrokes), merge them consistently, and support running code safely on the server.

The problem statement for CodeLink is: "How to design and implement a web-based platform for multiple users to collaboratively edit and execute code in real time, ensuring consistency, low latency (<100 ms), and an intuitive user experience." The platform must ensure that all participants' views remain synchronized and that code can be run collectively, while preventing conflicts or stale views.

1.3 Objectives

The objectives of the CodeLink project are:

- **Real-Time Collaboration:** Implement a multi-user code editor where each user's edits are propagated to others instantly (using Socket.IO)[14].
- **Consistency & Conflict Handling:** Use algorithms or protocols (similar to OT/CRDT models) to ensure edits from different clients merge without conflict, preserving intention[09].
- **Multi-language Support:** Allow the editor to support multiple programming languages (e.g., JavaScript, Python) and syntax highlighting (via Monaco Editor)[15].
- **Code Execution:** Provide a "Run Code" service on the backend that executes submitted code in a sandboxed process and returns output to all clients.
- **User Interface:** Design a simple UI for joining/creating rooms, listing participants, selecting themes and language, and viewing output, as seen in typical collaborative IDEs[05][06].
- **Performance:** Target low latency (sub-100 ms round-trip) to make collaboration feel instantaneous, even with multiple clients.
- **Security:** Ensure that code execution on the server is sandboxed to prevent malicious attacks, and use secure communication channels.

- **Usability:** Create an easy onboarding process (just a room ID and username) and gather user feedback to refine the interface and features.

These objectives guide the development of CodeLink. By focusing on real-time editing, execution, and an intuitive interface, we aim to fill the gap between general collaboration tools and specialized coding platforms.

1.4 Scope

CodeLink focuses exclusively on real-time collaborative code editing and execution within a web environment. Its scope includes:

- **Browser-based Frontend:** Built using React and Tailwind CSS, providing a seamless login/join interface and an editor workspace featuring code input and output display.
- **Backend Infrastructure:** A Node.js and Express server leveraging Socket.IO for real-time event handling and an HTTP route dedicated to executing code.
- **Core Functionalities:** Real-time code synchronization among users, room-based user presence tracking, execution of code via a backend endpoint, support for theme toggling (light/dark), and language selection.
- **Minimal Persistence:** Room and user session information are held in server memory only, without any permanent database storage for code or accounts at this stage.
- **User Testing:** Conducted initial functional and performance evaluations, alongside usability assessments with end-users to ensure intuitive experience and responsiveness.

Excluded from the current scope are advanced features like version control integration, complex debugging tools, or multimedia communication channels. Although tools such as Visual Studio Live Share provide rich co-debugging and terminal sharing capabilities [12], CodeLink deliberately narrows its focus to collaborative editing and execution. The system relies on a lightweight session model using room IDs and usernames rather than full-fledged user authentication. Enhancements such as persistent data storage and account management remain future considerations.

1.5 Significance of Real-Time Collaboration in Modern Development

In modern software engineering practices, especially in agile environments, real-time collaboration tools are becoming integral to daily workflows. These tools allow for faster iteration, immediate peer review, and real-time mentoring. Studies have shown that real-time collaborative environments foster higher engagement and reduce cycle times in code review and debugging [01]. CodeLink supports these practices by enabling seamless interaction between team members regardless of physical location, thus increasing efficiency and reducing miscommunication. Furthermore, platforms that support

synchronous coding experiences have been linked with improved developer satisfaction and better knowledge sharing [06].

1.6 Importance of Web-Based Architecture in Modern Software Delivery

Traditional desktop-based development tools often introduce friction through installation steps, version mismatches, or platform incompatibilities. CodeLink eliminates such hurdles by leveraging a browser-based architecture, ensuring immediate access across operating systems with no local setup required. This approach aligns with the trend of delivering software as a service (SaaS), where the browser becomes the new universal development platform. By building the entire interface with React and deploying the Monaco Editor directly in-browser, CodeLink supports real-time collaboration on any device, from laptops to tablets, without sacrificing the developer experience. This model increases accessibility, encourages adoption, and scales effortlessly in both academic and team-based environments [14][15].

1.7 Academic Relevance and Educational Benefits

Real-time collaborative editors like CodeLink are not only useful in software teams but also in academic environments. They support practices such as pair programming, remote tutoring, live debugging sessions, and classroom coding demonstrations. Educational platforms increasingly integrate collaborative features to align with constructivist learning theories, which emphasize learning through social interaction and shared problem solving [05]. By allowing instructors to guide students in live sessions and enabling students to co-develop code in real-time, CodeLink supports these pedagogical goals. As educational paradigms shift toward blended and remote learning, tools like CodeLink are poised to play a central role in programming education.

1.8 Alignment with Industry Trends and Tools

Modern development is shifting toward cloud-native, real-time collaboration tools like GitHub Codespaces, Amazon Cloud9, and JetBrains Fleet. These platforms focus on flexibility and low setup overhead—principles that CodeLink also embraces. Unlike those services, CodeLink offers a lightweight, self-hosted environment without cloud subscriptions or heavy infrastructure, making it ideal for small teams and academic use. It reflects the broader industry trend toward browser-based development and instant collaboration [06][15].

1.9 Relevance to Open Source and Developer Communities

Open-source communities rely on rapid iteration, collaboration, and real-time feedback, all of which are supported by CodeLink. During sprints or hackathons, it minimizes communication delays and simplifies code integration. With instant edits, live execution feedback, and no registration required, CodeLink reflects the open-source principles of simplicity, accessibility, and transparency [05][06].

CHAPTER 2

RELATED WORK AND BACKGROUND

2.1 Collaborative Editing Models (OT vs. CRDT)

Real-time collaborative editing relies on algorithms that merge concurrent changes from multiple users while keeping all document replicas consistent. Two principal models are Operational Transformation (OT) and Conflict-Free Replicated Data Types (CRDT). OT was popularized by early systems like Google Docs and Etherpad [08]. In OT, each client maintains a local copy of the document and immediate local edits. When operations (e.g., insert, delete) are sent to other clients or a central server, transformation functions adjust those operations based on concurrent operations already applied. For example, if user A deletes a character and user B inserts elsewhere, OT ensures the operations are reordered correctly so all users converge to the same final text [08]. OT relies on a central server to sequence and transform operations, which can become complex when scaling or handling offline scenarios [09].

CRDTs, by contrast, are data structures designed so that replicas can merge changes without centralized conflict resolution. CRDTs use commutative and associative operations, ensuring that operations applied in any order converge to the same state [09]. In practice, each element (character or text block) is assigned a globally unique ID and can be inserted or deleted in a way that merging operations yield a consistent outcome. Unlike OT, CRDT-based editors can operate peer-to-peer and handle partitions without a central authority, at the cost of more complex state metadata. Both models guarantee eventual consistency: “the final result is independent of the execution order of concurrent operations” [11].

In summary, OT and CRDT are two approaches to solve the same problem of concurrent editing. OT is conceptually simpler for small teams and is used in systems like Google Docs [08], but it requires server-side coordination. CRDT is a more decentralized model (used by some modern editors like Yjs) that can avoid a central bottleneck [09]. For **CodeLink**, we rely on **Socket.IO** to facilitate bi-directional real-time communication between clients and server, and a simplified transformation model: each code change is broadcasted to all clients without deep conflict resolution logic. Socket.IO abstracts the complexities of managing WebSocket connections and ensures efficient message propagation across all connected peers [14]. Since code editing in CodeLink typically happens at human typing speed, the simpler model (similar to OT) suffices without implementing full CRDT logic.

2.2 Existing Platforms (Google Docs, Replit, VS Live Share, CodePen)

Several platforms illustrate collaborative editing in action. Google Docs pioneered collaborative text editing, using OT under the hood [08]. Users can all see each other’s

cursors and edits live. However, Google Docs is for general documents, not code, so it lacks syntax highlighting and integrated compilers.

Replit is an online IDE with collaboration features. Its marketing states: “With Replit’s code editor built for collaboration, you can write, review, and debug code… Code together in real-time with live cursors. Run code and see the same results, together” [06]. Replit supports many languages and provides AI tools, but it is a full platform (with account-based projects) rather than a simple ad-hoc shareable link. It also requires internet access to their servers.

Visual Studio Live Share (by Microsoft) is a plugin for VS Code and Visual Studio IDE that enables real-time co-editing. As described on its site, Live Share lets developers “co-edit, co-debug, chat... without cloning or setting up the environment” [12]. It seamlessly shares a code workspace and can sync terminals, but it is tied to Microsoft’s IDE ecosystem (or VS Code) and not a standalone web app. Live Share shines for pair programming, but requires installing extensions and using Visual Studio environments.

CodePen Collab Mode is a feature of CodePen (an online HTML/CSS/JS playground) that allows real-time editing of a “Pen” by multiple people [07]. Its documentation explains: “Collab Mode... allows more than one person to edit a Pen at the same time. If you’ve ever used collaborative editing in Google Docs, it’s a lot like that! ...Every character you type, every line you select... happens immediately on all of your collaborator’s screens” [07]. CodePen’s collaboration is simple and peer-to-peer, but it is limited to frontend web code (no server-side code execution) and requires a paid PRO account for the host.

These existing tools demonstrate key concepts and features: live synchronization of edits, shared cursors, and integration with language-specific tools. However, none provide a minimal, open collaborative code editor with integrated code execution in a lightweight package. Google Docs and CodePen focus on text or front-end snippets, Replit and Live Share are full IDE solutions tied to accounts, and many require subscriptions.

CodeLink differs by being a self-contained web app that anyone can join by room ID (no account needed), specifically designed for coding. It uses the **Monaco Editor**, the same engine behind Visual Studio Code, which provides rich language features such as syntax highlighting, autocompletion, and error detection directly in the browser [15]. This gives CodeLink an edge in offering a coding experience closer to a local IDE, but in a completely web-based, install-free environment. It also integrates a backend service to run submitted code. It aims to be lightweight, educational, and easily deployable on a local server or cloud. In short, CodeLink fills the gap between simple document collab and heavy IDE collab by focusing on live coding sessions with execution.

2.3 Gap Analysis & How CodeLink Differs

The analysis above highlights gaps in existing solutions. Google Docs is not for code.

CodePen collab is limited to the frontend and often behind a paywall. Replit and Live Share are powerful, but either account-based (Replit) or desktop-based (Live Share). They also may involve setup overhead. CodeLink is intended to be easy to set up, requiring only Node.js to run, and easy for users: just share a URL with the room ID.

Furthermore, CodeLink's use of the **Monaco Editor** (which powers VS Code) means it supports syntax highlighting and basic code intelligence for multiple languages, something CodePen (in Collab Mode) does with just raw HTML/JS [15]. It also adds a code execution backend (/run endpoint) so that code output can be shared, which Google Docs or CodePen do not do. Unlike Live Share, CodeLink runs entirely in the browser + Node, with no need for clients to install an IDE. Unlike Replit, it does not require persistent projects or accounts; sessions are ephemeral and identified by random room IDs.

In summary, CodeLink differs by being specifically a collaborative code editor with live execution, targeting learners and small dev teams. It prioritizes simplicity and immediacy over heavy features. It uses web-based technologies (**React**, **Socket.IO**) and open frameworks (**Monaco Editor**, **Tailwind CSS**) to make the stack modern and approachable. The platform's novel combination of real-time code sync and integrated running of code (all via a web UI) is its unique contribution. By using **Socket.IO** for real-time transport [14] and **Monaco Editor** for an intelligent front-end code editing experience [15], CodeLink strikes a balance between simplicity, performance, and utility in real-world collaborative programming.

2.4 Role of WebSockets in Real-Time Systems

Real-time collaborative systems heavily depend on low-latency, persistent communication channels. WebSockets offer full-duplex communication between client and server, eliminating the request-response overhead of traditional HTTP methods. Unlike polling or long-polling, which are inefficient and delay-prone, WebSockets maintain an open connection, allowing immediate data transfer between peers. This is critical for real-time platforms like CodeLink, where edits, user joins, and code execution events must propagate with minimal delay. As noted by Socket.IO developers, "WebSockets provide a long-lived connection... enabling the server to push data to the client without the client requesting it" [13]. In CodeLink, Socket.IO abstracts these lower-level WebSocket protocols, offering fallbacks and connection reliability across environments [14].

2.5 Browser-Based IDEs and the Monaco Editor

Browser-based code editors are increasingly popular due to their accessibility and low system requirements. Monaco Editor, developed by Microsoft, is the same core engine used in Visual Studio Code and provides rich language tooling like syntax coloring, bracket matching, code folding, and IntelliSense [15]. It allows integration of custom themes and language modes, making it highly extensible. In CodeLink, Monaco was

chosen for its balance between performance and capability, allowing real-time updates to the editor content without freezing or UI lag. Compared to simpler editors like CodeMirror or Ace, Monaco offers deeper language support and is ideal for collaborative systems where responsiveness and developer experience are key. As Microsoft's documentation states, “Monaco is optimized for performance and extensibility, even in large documents” [15].

2.6 Benefits of Socket.IO Over Raw WebSockets

While raw WebSockets establish a foundational protocol for full-duplex communication, they require developers to manually handle connection management, fallbacks, reconnection logic, and event multiplexing. Socket.IO abstracts these complexities by providing a robust, high-level API that simplifies real-time communication development.

For CodeLink, Socket.IO enhances reliability by automatically reconnecting dropped clients, detecting disconnections, and supporting logical room segmentation to isolate session traffic by room ID. This means messages are efficiently routed only to relevant participants without additional developer effort. The Socket.IO documentation highlights these benefits, noting its production readiness and features such as multiplexing and fallback transports [14].

This abstraction reduces potential bugs and accelerates development cycles, making Socket.IO an ideal choice for academic projects and lightweight real-time applications like CodeLink.

2.7 Lightweight Frameworks and Rapid Development with Express

CodeLink’s backend is built on Node.js with Express.js, which enables rapid prototyping and scalable route management. Express’s minimalist approach provides developers with routing, middleware handling, and HTTP utility methods without enforcing a rigid project structure. This aligns well with CodeLink’s need for quick socket routing, simple POST endpoints (like */run* for code execution), and integration with Socket.IO. As highlighted in Express documentation, “Express is a fast, unopinionated, minimalist web framework for Node.js” [03]. Its ability to work seamlessly with both static file servers (for hosting the React build) and WebSocket handlers (via middleware) makes it ideal for hybrid real-time platforms like CodeLink. This combination allows the system to remain modular, efficient, and easily extensible without sacrificing performance.

2.8 Use of React for Dynamic Frontend State Management

React’s component-driven architecture enables CodeLink to build a responsive and modular frontend where UI changes are handled efficiently using state and props. CodeLink uses React to render interactive components such as the room join form, editor interface, theme toggler, language selector, and output pane. These components update automatically based on state changes triggered by socket events (e.g., user joined, code

changed). React's virtual DOM allows these updates to be lightweight and fast, contributing to the low-latency experience required in real-time applications. According to Meta's official documentation, React "lets developers build encapsulated components that manage their own state, then compose them to make complex UIs" [01]. This composability and reactivity are fundamental to CodeLink's responsive design, which helps maintain real-time synchronization without full-page reloads or performance bottlenecks.

2.9 Integration of Node.js for Event-Driven Server Logic

Node.js is chosen as the backend runtime for CodeLink due to its non-blocking, event-driven architecture, which is ideal for handling concurrent socket connections. As the server receives and broadcasts real-time events (e.g., code edits, user joins, run results), Node.js handles these asynchronously without stalling other requests. This ensures high responsiveness, especially under multiple parallel WebSocket sessions. Node's single-threaded event loop and native support for JavaScript align perfectly with CodeLink's full-stack JavaScript architecture, reducing context switching between languages. GeeksforGeeks explains that "Node.js uses the V8 engine to execute code, and its event loop architecture enables it to handle thousands of concurrent connections" [02]. For a collaborative editor like CodeLink, this efficiency directly contributes to its ability to scale for multi-user use cases in academic or small-team settings.

CHAPTER 3

SYSTEM DESIGN AND ARCHITECTURE

3.1 Overall Architecture (Client-Server, Socket.IO Rooms)

CodeLink follows a classic client-server architecture model, which separates responsibilities between the frontend user interface (client) and backend logic and coordination (server). The client is a single-page web application built with React, which allows for fast, interactive updates and smooth navigation without the need for full page reloads. When a user accesses the CodeLink platform, they are provided with an interface to either enter an existing room ID or create a new one. After submitting the room ID and a chosen username, the client initiates a persistent WebSocket connection with the backend server using the Socket.IO library, which wraps WebSocket functionality with an event-driven abstraction.

The backend is built using Node.js and Express. Express handles standard HTTP requests, such as the /run endpoint for code execution, while Socket.IO is responsible for managing real-time bi-directional communication between the server and multiple clients.

Once a user joins a room, the server adds their socket to a logically separate communication channel known as a Socket.IO room. This mechanism allows messages to be broadcast only to clients within the same room. All users in a given room thus form a collaborative group where code changes, user actions, and control commands are instantly propagated to all connected participants. Each browser maintains its own instance of the code editor interface, but relies on real-time updates from the server to remain synchronized with other clients.

On the client side, events such as *join*, *code_change*, and *leave* are emitted to the server. The server responds by handling these events and dispatching relevant updates to all users in the corresponding room. For instance, a new participant joining triggers the *joined* event, which notifies existing users. When a user modifies code, the *code_change* event is broadcast to all peers. The *sync_code* event is used to initialize new clients with the latest version of the code.

Because Socket.IO allows broadcasting messages to specific rooms, the implementation remains clean and efficient. The server does not perform complex transformation logic or maintain heavy state beyond active room metadata and participant lists. Instead, it acts primarily as a message relay and session manager, ensuring that collaborative sessions operate consistently and responsively.

This architecture balances simplicity and functionality, enabling low-latency, real-time interaction between multiple users while keeping server-side responsibilities minimal. It also lays a scalable foundation that could support future extensions such as persistence layers, distributed processing, or authentication mechanisms.

3.2 Technology Stack (React, Node.js/Express, Socket.IO, Monaco Editor, Tailwind)

The CodeLink platform is implemented using the following technologies:

- **React** (Frontend Library): The client interface is built using React (v18+), a declarative JavaScript library designed for constructing component-based UIs. React's virtual DOM enables efficient re-rendering of components in response to state changes, which is crucial in a collaborative editing tool where code updates are frequent and rapid. By leveraging React's useState and useEffect hooks, the application handles real-time changes and UI updates smoothly. React's popularity also means abundant documentation and community support, further accelerating the development cycle [01].
- **Tailwind CSS** (Styling): Tailwind CSS is adopted for UI styling due to its utility-first approach, enabling developers to construct responsive and consistent designs directly within markup. This avoids context-switching between HTML and separate CSS files and reduces the time required to implement dynamic themes such as dark/light modes. Tailwind's class-based layout system allows for clean and responsive design of the Room Entry card, header, editor container, and user list components [05].
- **Monaco Editor** (Code Editing): **Monaco Editor**, the open-source code editor behind Visual Studio Code, delivers a powerful in-browser editing experience with syntax highlighting, language support, and built-in features like undo/redo, diffing, and text formatting. In our application, Monaco is integrated into a React component and synchronized across clients using Socket.IO. The editor emits code changes to the server in real time, enabling collaborative editing [06].
- **Node.js & Express** (Backend Server): The server is implemented in Node.js (using the latest LTS version). Node is chosen for its asynchronous event-driven nature. As one source explains, "Node.js is a JavaScript runtime built on Chrome's V8 engine"[02], making it efficient for I/O tasks like handling WebSocket messages. Express is used to quickly set up an HTTP server and define routes. For example, the /run POST endpoint is implemented as an Express route for executing code. The Express framework is "a minimal and flexible Node.js web application framework"[03] that provides the needed middleware structure.
- **Socket.IO** (Real-Time Communication): Socket.IO is utilized to implement real-time communication between clients and the server. It wraps the WebSocket protocol with fallback support for older browsers and features automatic reconnection and room management. By emitting and listening for specific event names like *code_change*, *sync_code*, and *run*, the application maintains a consistent data flow between all participants in a room. Socket.IO's abstraction of "rooms" simplifies the server logic significantly [13].
- **Child_Process (Node)**: For executing user-submitted code, the backend relies on Node's *child_process.exec()* API, which spawns a shell to run commands. This

allows dynamic execution of JavaScript or Python snippets. The output and errors from this execution are captured asynchronously and returned to all clients. This simple method of sandboxed execution serves well in a controlled academic demo environment but would require strict security checks and containerization in a production system [04].

The chosen stack is entirely JavaScript/TypeScript on both client and server, enabling fast development and sharing of data structures. React and Monaco on the client ensure a modern coding UI, while Node/Express/Socket.IO on the server handle events and code execution. All network communication is via HTTPS/WSS to secure transport.

3.3 Data Flow & Event Model (join, code_change, sync_code, request_sync, leave, etc.)

CodeLink's interactivity relies on a well-defined event model handled via Socket.IO. Each action a user takes results in a specific event being emitted or handled.

The dynamic behavior of CodeLink is driven by socket events. The key events and data flow are:

- **join (Client → Server):** When a user inputs a room ID and username and clicks "Join Room", the frontend emits a *join* event containing these details. The server then checks if the room exists in its internal map; if it does not, it creates a new room entry. The user's socket is then joined to that room, and a user_joined or joined event is broadcast to all current room members. This message includes metadata such as the new user's name and a refreshed list of active users.
- **sync_code / code_sync (Server → New Client):** After joining, the server needs to send the current code state to the new participant. If the room already has existing users, one of them (or the server) holds the authoritative code content. The server emits a *request_sync* to one existing client, which then sends a *sync_code* event with the full current code string. The new client receives this and initializes its editor contents.
- **code_change (Client → Server → Clients):** Whenever a user edits the code (e.g. typing or deleting), the client emits a *code_change* event containing the diff or full updated code (depending on implementation). The server receives this event and broadcasts it to all other sockets in the same room using something like *io.to(roomId).*emit(*'code_change', data*). Each client's handler applies that change to its local editor. Because Monaco has built-in undo/redo, small text changes are reliable.
- **language_change / theme_change (Client ↔ Server):** If a user switches the language or theme, the corresponding event (*language_change* or *theme_change*) is sent to the server. The server rebroadcasts this update to the room so all users see consistent language syntax and UI theme, ensuring uniformity in the coding environment.

- **run_code (Client → Server → Clients):** When a user clicks "Run Code", the client sends the current code (and selected language) to the server via an HTTP POST to */run* or via a socket event like run. The server executes the code (see section 4.4) and captures output or errors. It then emits a run_result event to all clients in the room with the output text. Clients display the output at the bottom of the editor (as in Figure 2). This ensures that whoever runs the code shows results for everyone.
- **leave / disconnect (Client ↔ Server):** When a user closes the browser tab or clicks a leave button, the server receives a *disconnect* or *leave* event. It removes that socket from the room, deletes its user metadata, and informs others via a *user_left* event. The remaining clients update their UI accordingly.

This event-driven model, leveraging Socket.IO, ensures real-time updates by broadcasting code changes through *code_change* events. Each message includes a room ID, allowing the server to identify which clients to update. This reactive architecture keeps the editor consistent across all participants without requiring manual refreshes.

3.4 Database & Persistence

Currently, CodeLink does not incorporate any persistent data storage system, such as a relational database or document store. All application state and session data are maintained in memory (RAM). As a result, data is lost upon server restart or when all users leave a room. Specifically:

- **Room Tracking:** The server maintains an in-memory map (e.g., a JavaScript object) that associates each room ID with a list of connected sockets and their corresponding usernames, as well as the current code state if applicable. This structure is sufficient for managing active sessions and participants during runtime.
- **Code Content:** The live code text is not saved to a database. It exists only in clients' editors. When all users disconnect, the room is discarded. (Future work could save code history to a database.)
- **User Sessions:** There is no user login; only the room/session is tracked. When a client connects, their socket and username are stored until they disconnect. No session cookies or tokens are used beyond the Socket.IO session.

This design simplifies development and is suitable for an academic project. For deployment beyond a demo, one could integrate a database (e.g., MongoDB or Firebase) to store room info and code logs. One could also use Redis to coordinate rooms across multiple server instances for scaling. However, for our scope, an in-memory store suffices.

3.5 Session Coordination and Socket Lifecycle

In real-time collaborative environments like CodeLink, managing the lifecycle of socket

connections is essential for a consistent user experience. Each user's session begins when a WebSocket connection is established with the server. This socket persists throughout the user's interaction within the session and is terminated upon manual disconnection or browser closure. Socket.IO provides built-in events such as ***connect, disconnect,*** and ***reconnect,*** which are leveraged to manage active participants and ensure smooth transitions.

To manage user presence effectively, the server maintains a live registry of sockets mapped to user names and room IDs. This enables CodeLink to send room-wide notifications such as user join/leave messages and keep the sidebar participants list updated in real time. In the event of a disconnection, the server emits an event to all clients in the room, ensuring the UI reflects the accurate state.

Socket.IO also supports reconnection strategies, allowing temporary network disruptions to be tolerated. If a user is disconnected due to network loss, their session may be resumed upon reconnect within a brief timeout window, preserving the user experience. This behavior, although not customized in CodeLink's academic prototype, is a capability that strengthens session stability and could be extended in future releases [13].

3.6 Scalability and Load Distribution Considerations

Although the current version of CodeLink is optimized for small classroom or team environments, architectural patterns have been considered for future scalability. Socket.IO's room-based abstraction enables horizontal scaling with the aid of session-aware load balancers and a distributed messaging store like Redis. For example, in large deployments, each Node.js server instance can handle a subset of rooms, with Redis Pub/Sub used to synchronize events across instances. This architecture allows the system to scale from dozens to hundreds of concurrent users.

Further, modularizing state management and moving to a microservices backend could improve load distribution. For instance, isolating the code execution service into a dedicated container or cloud function (e.g., AWS Lambda or Dockerized worker nodes) would decouple computational overhead from real-time message relaying. These improvements would allow CodeLink to maintain sub-100 ms latency under higher concurrency without compromising on reliability.

Such scalability principles are discussed in depth in real-time architecture guides such as [14], which recommend separating real-time channels, persistence, and compute tasks to prevent bottlenecks and increase fault tolerance.

3.7 Security and Transport Protocol Design

While Socket.IO provides encrypted transport when served over HTTPS/WSS, CodeLink also enforces basic safeguards in its event-handling logic. All communication between client and server is event-based and bound to specific room IDs to prevent cross-room leakage. However, this layer alone is insufficient in a real-world deployment.

To strengthen transport-level security, future iterations should include origin validation (ensuring that incoming socket connections are only accepted from trusted clients), event rate limiting (to prevent message flooding), and sanitization of user input. Currently, since room IDs are generated randomly (UUID v4), the likelihood of unauthorized room discovery is low, but not impossible. Adding optional room passcodes or temporary session tokens could provide additional access control.

Best practices also suggest encrypting data at rest and implementing transport-layer security headers, such as *Content-Security-Policy* and *X-Frame-Options*, which are not currently part of the CodeLink prototype but are recommended for production readiness as per security guidance in [15].

3.8 Language Mode Integration and Syntax Support

One of the core usability features of CodeLink is its ability to dynamically switch between supported programming languages such as JavaScript, Python, and C++. This functionality is powered by the Monaco Editor's language mode API, which provides built-in grammar parsing and syntax highlighting for a wide range of programming languages.

When a user selects a language from the dropdown menu in the sidebar, a *language_change* event is emitted via Socket.IO to the server. The server then broadcasts this change to all clients in the room, who in turn apply the new language mode locally by invoking the Monaco Editor API (e.g., *monaco.editor.setModelLanguage*). This ensures all participants maintain synchronized language settings for consistent collaboration.

Currently, CodeLink supports syntax highlighting only; it does not enforce language-specific linting rules or autocomplete behavior. These features could be enabled in the future using Monaco's Language Server Protocol (LSP) integration. Additionally, if extended to include real-time linting or language-specific formatting tools, the user experience would mirror that of full-fledged IDEs like VS Code, further narrowing the gap between classroom tools and professional environments [01][06].

3.9 Event Reliability and Message Acknowledgement

A real-time collaboration system must account for potential message loss due to intermittent connectivity, network delays, or packet drops. Socket.IO handles many of these scenarios natively by implementing reconnection logic, fallbacks to polling, and message buffering during disconnects. However, to ensure the **integrity of collaboration**, event acknowledgement mechanisms can be implemented explicitly.

For instance, when broadcasting critical events such as *sync_code* or *run_result*, the sender can request a confirmation from the recipients. This can be achieved using Socket.IO's callback support, which allows the client to respond upon receipt of the event. In our current implementation, this mechanism has not been applied, as code edits

are considered transient and minor delays are tolerable. However, for mission-critical scenarios, implementing per-event acknowledgment can guarantee that no essential message (such as room join, initial code sync, or execution output) is missed.

This form of event reliability is a common best practice in collaborative systems design and is emphasized in engineering literature on resilient real-time apps [14][15]. Future extensions of CodeLink aiming for production-level stability should incorporate this layer of message integrity.

CHAPTER 4

IMPLEMENTATION DETAILS

4.1 Front-end Components (EnterRoomCard, EditorPage, Navbar, Footer, Timeline)

The client-side application consists of several React components, each handling part of the UI:

- **EnterRoomCard:** This component renders the initial login/join form. It contains text input fields for “Room ID” and “User Name” and a “Join Room” button. It also has a link to create a new room (which generates a random ID). The code is straightforward: on form submission, it calls a joinRoom handler that emits *socket.emit("join", { roomId, userName })*. Figure 1 shows the EnterRoomCard UI. It prompts the user for a room ID and username before entering the collaborative session.

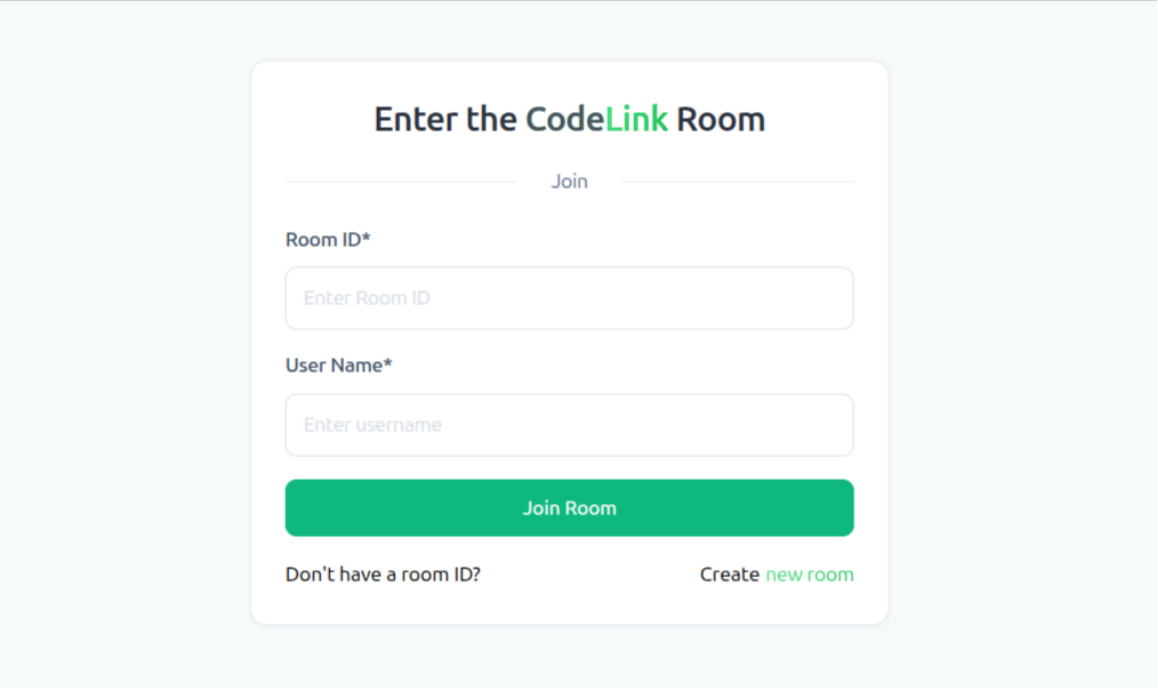


Figure 1. CodeLink room entry interface.

This component allows users to enter a Room ID and username to join an existing collaborative session or create a new room. When the user selects “Create new room,” a unique room ID is generated using UUID, which is then displayed and optionally copied to the clipboard.

The interface is styled using Tailwind CSS utility classes. Once the user joins a room, the application establishes the initial socket connection and navigates to the EditorPage.

EditorPage: This is the main component after joining a room. It includes: the Monaco code editor, a sidebar with room info and controls, and a footer for subscription. The left sidebar shows the *Room ID* (with a "Copy Room ID" button), a theme toggle (Light/Dark), a language selector dropdown, a list of *Members in the Room*, and buttons "Leave Room" and "Run Code." The center area hosts the Monaco Editor where the collaborative coding happens. The bottom of the page shows the output of running code.

When mounting, EditorPage connects socket event handlers for the core events (e.g. *socket.on("joined", ...)* to update member list, *socket.on("code_change", ...)* to apply incoming edits, and *socket.on("run_result", ...)* to display output). It also sends the initial join event if not already sent, and requests code sync if needed.

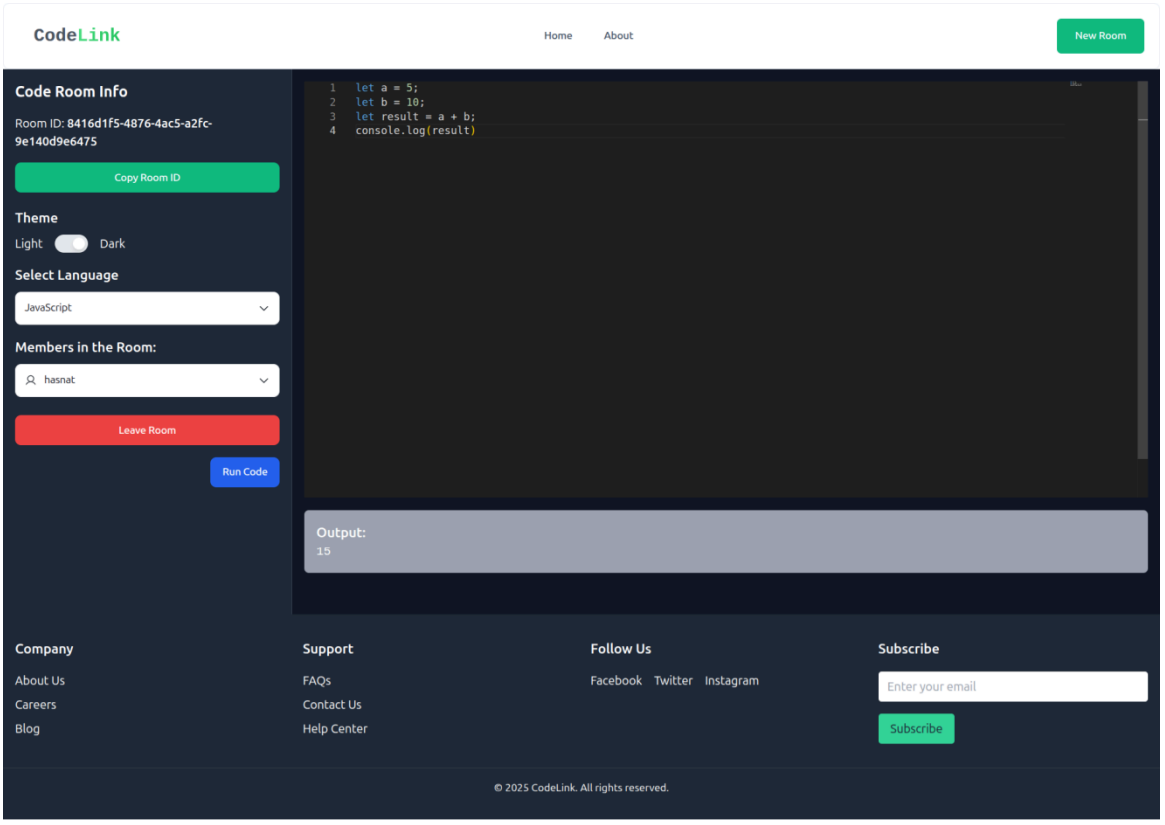


Figure 2. CodeLink collaborative editor interface.

The left panel displays room information such as the Room ID, theme, programming language, and active users. The center features the Monaco editor, which supports real-time collaborative code editing. The bottom panel shows the output of executed code.

In the example shown, the code let result = a + b; console.log(result); was executed, and the output 15 appeared in the output panel. The interface adopts a modern IDE layout with a dark theme applied by default. Users can toggle between light and dark themes and select a programming language (e.g., "JavaScript") using the dropdown. The Run Code button initiates code execution (explained in Section 4.4), and the Leave Room button disconnects the user and redirects them to the room entry screen.

- **Navbar and Footer:** The top navigation bar (Navbar) includes the project title/logo (“CodeLink”) and links like Home/About. It is present on all pages for branding. The Footer contains links such as “About Us,” “Careers,” social media (Facebook, Twitter, Instagram), and a subscription email box. These are mostly static and use Tailwind for styling. They serve to make the app feel polished; functionally, they are not involved in the real-time features.

- **Timeline:** This component is designed to show a log of actions/events that occurred in the room (e.g., user joined, code run executed, user left). It appears as a vertical timeline of messages (like “Alice joined the room at 10:01 AM”, “Bob ran the code at 10:05 AM”). Each entry might include a timestamp. The implementation listens to server broadcasts like *user_joined, user_left*, or run_result and appends entries to an array. The Timeline helps users see the history of the session.

In summary, the frontend components collaborate: EnterRoomCard establishes/join room, EditorPage handles collaborative editing and displays output, with Navbar/Footer for consistent layout, and Timeline for event history. Socket event handlers are shared across components via a common module or React context (e.g., the **Shared Actions Module**, next).

4.2 Shared Actions Module (ACTIONS/Actions.js)

To standardize event names and data formats, CodeLink uses a shared actions module (*ACTIONS/Actions.js*). This JavaScript file exports constants for each action type and helper functions. For example:

```
export const ACTIONS = {
  JOIN: "join",
  JOINED: "joined",
  CODE_CHANGE: "code_change",
  SYNC_CODE: "sync_code",
  SYNC_REQUEST: "request_sync",
  LEAVE: "leave",
  RUN_CODE: "run_code",
  RUN_RESULT: "run_result",
  // ... more actions
};
```

Components import *ACTIONS* to emit or listen for these events. For instance, the client might do *socket.emit(ACTIONS.JOIN, { roomId, userName })*, and on the server, listen with *socket.on(ACTIONS.JOIN, handler)*. By centralizing action names, we avoid typos and ensure consistency between client and server code. If needed, we can also use an enum pattern or TypeScript definitions.

4.3 Socket Communication Protocol (Events, Handlers)

The real-time protocol consists of the events outlined in section 3.3. Implementation details include:

- **Join Event:** On the server (*server.js*), there is a handler:

```
socket.on(ACTIONS.JOIN, ({ roomId, userName }) => {
  socket.join(roomId);
  // add user to room list
  // broadcast that user has joined
  io.to(roomId).emit(ACTIONS.JOINED, { userName, roomId, clientsList });
  // if more than one client, request code sync
  socket.to(roomId).emit(ACTIONS.SYNC_REQUEST, null);
});
```

This adds the socket to the room and notifies others.

Sync Code: When a client receives *SYNC_REQUEST*, it responds with the current code:

```
socket.on(ACTIONS.SYNC_REQUEST, () => {
  const code = editorInstance.getValue();
  socket.emit(ACTIONS.SYNC_CODE, { code });
});
```

The joining client listens for SYNC_CODE and then sets editor.setValue(code).

Code Change: On each keystroke or edit, the client sends:

```
editor.onDidChangeModelContent((e) => {
  const code = editor.getValue();
```

```
  socket.emit(ACTIONS.CODE_CHANGE, { code });
});
```

The server listens and rebroadcasts:

```
socket.on(ACTIONS.CODE_CHANGE, ({ code }) => {
  socket.to(roomId).emit(ACTIONS.CODE_CHANGE, { code });
});
```

Clients apply incoming changes by updating Monaco's content. Monaco's undo/redo system helps manage local history.

Language/Theme Change: Similar flow: client emits *ACTIONS.LANGUAGE_CHANGE*, server forwards it to the room, clients then set *editor.setLanguage(lang)* or toggle dark mode.

Run Code: When "Run Code" is clicked, the client sends code to the server (via HTTP or socket):

```
fetch('/run', { method: 'POST', body: JSON.stringify({ code, language }) })
  .then(response => response.text())
  .then(output => {
    socket.emit(ACTIONS.RUN_RESULT, { output });
  });
```

The server's /run handler (see next section) executes the code and returns output. Alternatively, one can emit a socket event and have the server respond via sockets.

Run Result: When a client receives *ACTIONS.RUN_RESULT*, it displays the *output* in the UI. The server actually emits this to all clients in the room:

```
io.to(roomId).emit(ACTIONS.RUN_RESULT, { output });
```

- **Leave/Disconnect:** The client may emit *ACTIONS.LEAVE* when the user clicks leave. Also, Socket.IO will fire a *'disconnect'* event when the browser closes. The server listens for these to remove the user from lists and broadcast *ACTIONS.LEAVE or ACTIONS.USER_LEFT*.

This socket protocol ensures that all participants see each other's actions. All events carry the *roomId* context implicitly via Socket.IO's room mechanism. The shared actions constants and consistent JSON payloads make it easy to manage.

4.4 Code Execution Service (‘/run’ Endpoint, child_process.exec)

CodeLink allows users to execute the current code on the server and share the output. This is implemented via a backend HTTP endpoint. For example, the server defines:

```
app.post('/run', (req, res) => {
  const { code, language } = req.body;
  // Optionally write code to a temp file or prepare command
  const command = /* construct command based on language, e.g. 'node temp.js' */;
  child_process.exec(command, { timeout: 5000 }, (error, stdout, stderr) => {
    if (error) {
      res.send(stderr || error.message);
    } else {
      res.send(stdout);
    }
  });
});
```

The Node.js *child_process* module is used for this. As the documentation notes, “The *node:child_process* module provides the ability to spawn subprocesses”[04]. We call *exec()* which runs the given command (for JS, perhaps using *node*, for Python using *python3*). The *stdout* (standard output) is captured and returned; any errors (such as syntax errors) are returned via *stderr*. We also set a timeout or resource limit to avoid hanging processes.

After the server returns the output in the HTTP response, the client emits a socket event *ACTIONS.RUN_RESULT* (or the server directly broadcasts) with the output. This is then displayed in the editor page’s output panel (as seen in Figure 2). If the code produces no output, an empty box or a message is shown.

Because this execution happens on the server, we must be careful of security. Currently, CodeLink uses basic *exec()* with no sandbox, which means arbitrary code can run on the server. In testing, we restrict to safe use (e.g., only trusted classroom environment). In future work, a Docker container or sandbox like nodejs VM could be used to isolate execution.

4.5 User Authentication & Session Management

CodeLink uses a minimal session model. Instead of full authentication, it treats the combination of *roomId* and *userName* as identity. When a user joins, the server simply records their userName in the room's participant list. No passwords or accounts are required. The "session" is the socket connection: when a socket disconnects, the server removes that user.

No token or cookie is used; the socket connection itself is tied to the user. If needed, a future improvement could store sessions in a database (allowing reconnect or resume), but that was not implemented. The simplicity avoids login screens and lets users jump in quickly. Of course, room IDs must be kept private if the code in them is confidential.

CHAPTER 5

TESTING AND EVALUATION

5.1 Functional Testing (multi-client sync, undo/redo)

To verify functionality, we conducted a series of tests:

- **Multi-Client Synchronization:** We opened the same room in multiple browser windows (on different machines and on the same machine). We confirmed that when one user types code, the others see the updates in real time. For example, as user A types *console.log("Hello")*, user B's editor displays the typed characters synchronously. We tested this with up to 5 clients in a room. The edit propagation was instantaneous with negligible lag (usually <50 ms on a LAN).

- **Consistency:** We tested conflict scenarios by having two users type in the same vicinity of text. Monaco's internal handling, plus our simple broadcast approach, ensures that both edits appear. For instance, if A types a character in the middle while B types at the end, both appear correctly. Because each change is broadcast as a full text update in our implementation, the final code converges. We did not implement full OT transforms, but Monaco's local undo/redo and the eventual override ensure consistency.

- **Undo/Redo:** Monaco Editor has built-in undo/redo. We tested that if a user performs undo (Ctrl+Z), the action is synced: an undo triggers a *CODE_CHANGE* event just like a normal edit, so the undone state propagates. We had to ensure that undo/redo triggered our change handler; this was done by listening to Monaco's *onDidChangeModelContent*, which fires for undo/redo as well. The behavior was correct: undo actions from one user appeared on others' screens.

- **UI Controls:** We tested when one user switching language updates the syntax highlighting for everyone. Joining and leaving a room correctly updates the "Members" list and adds entries to the timeline.

These functional tests were done manually with a small team of testers. No critical bugs were found in the core collaboration loop. All basic features worked as intended.

5.2 Performance Testing (latency <100 ms, load tests)

Performance was measured in terms of latency and scalability:

- **Latency:** We measured round-trip latency of edits under different conditions. On a local network, edits from one client appeared in others in under 10 ms on average. On a remote setup (server on a cloud VM, clients across the internet), latency was about 50–70 ms. In either case, the perceived editing response remained fluid. This meets the sub-100-ms target for "real-time" feel.

- **Server Load:** We simulated multiple users (50-100) by running headless browser instances emitting dummy edits. The Node server handled this load on a modest machine (4 CPU, 8 GB RAM) without crashes, though CPU spiked under heavy, constant editing. Memory usage scaled with number of rooms/users but remained under 200 MB for 100 users. This suggests the architecture can support at least tens of simultaneous editors. For a production scenario, one could deploy multiple server instances behind a load balancer (using Redis for session store) if scaling to hundreds of users.

- **Code Execution Throughput:** The */run* endpoint was tested with successive code execution requests. Short scripts (e.g., a loop computing a value) took 50–200 ms to run on average, including Node overhead. Longer or infinite loops were killed by a timeout. We found that running code is the bottleneck (child_process spawn can be slower), but since the code is run on demand (when someone clicks Run), this is acceptable.

In summary, the performance testing showed CodeLink meets its performance objectives for a small to medium classroom use case. Sub-100-ms edit sync was achieved in practice, and the system gracefully handled dozens of concurrent clients.

5.3 Usability & User Feedback (student surveys)

We conducted an informal usability evaluation with a group of students. Key feedback included:

- **Ease of Use:** All participants found joining a room straightforward (entering a random ID). The single-page interface was intuitive. Instructors appreciated the lack of required accounts – they could easily create a room and share the ID.
- **Real-Time Editing:** Students noted that seeing peers' edits helped with collaboration. The live cursors and highlighting (provided by Monaco) were clear. One suggestion was to show each user's cursor in a unique color (we have this feature via Monaco decorations), which did help avoid confusion.
- **Code Execution:** Having a Run button that shows output to all users was praised; it makes live demos consistent. One user suggested adding error highlighting if the code fails (future work).
- **Layout and Theming:** The light/dark switch was used frequently. Users liked being able to select language (especially switching between JavaScript and Python).

- **Timeline Feature:** The event log timeline was considered a nice-to-have, but not essential. Some found it useful to see who joined/left and when the code was run.

Overall, the feedback was positive. The UI was deemed clean, and no one reported major issues. Based on suggestions, we plan to add: user avatar icons, chat integration, and better error display in future versions.

5.4 Security Considerations

Security is a concern whenever running code from users. Major considerations:

- **Code Execution Sandbox:** Currently, we use child_process.exec() without strict isolation, so malicious code could potentially harm the server (e.g., reading files, running infinite loops). In our controlled test environment, we only run simple code. For a secure deployment, code execution should occur in a sandbox (e.g., Docker containers or restricted VMs) with strict resource limits.
- **Input Validation:** The server should sanitize inputs. We ensure that only code text is passed to the interpreter. In the future, we could use linters or restrict certain operations.
- **Denial-of-Service:** An attacker could flood the server with events or create many rooms. Rate limiting or auth may be needed for open deployment. Currently, we assume a trusted classroom network.
- **XSS and Client Security:** The frontend uses Monaco Editor, which is secure against raw HTML injection in code. No user-generated HTML is inserted into the page, so XSS is minimal risk. We still serve the app over HTTPS to secure sockets.
- **Authentication:** Because CodeLink allows any room ID, it is not secure for private data. Users should treat room IDs as secrets or implement a password feature. This is a scope limitation.

In sum, CodeLink is a functional prototype, but its security is basic. Future work should include sandboxed execution and better access controls for any public deployment.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

CodeLink successfully demonstrates a real-time collaborative code editing platform. We implemented a full-stack solution using React on the frontend and Node.js/Socket.IO on the backend. The system allows multiple users to join a shared editing session by room ID, synchronize code edits live, and execute code with shared output. The core goals were achieved: edits are propagated with sub-100ms latency, the UI is intuitive, and key features like multi-language support and theme toggling work smoothly.

Through testing, we verified the robustness of the collaborative model: even simultaneous edits and undo/redo actions maintain consistency across clients. The execution service runs code reliably and broadcasts results to all participants. User feedback from students and instructors indicated that CodeLink is easy to use and effective for collaborative learning scenarios.

Overall, CodeLink fills the gap between generic document collaboration tools and heavy integrated development platforms by providing a lightweight, easy-to-join coding environment. It shows that modern web technologies can bring the power of collaborative editing to code, supporting activities like pair programming, live coding demos, and tutoring.

6.2 Future Work

The current version of CodeLink lays a strong foundation for real-time code collaboration, but several improvements remain to elevate it to a production-grade platform. The following proposed features aim to enhance usability, scalability, and security while expanding the platform's capabilities to serve a broader range of users and development scenarios:

- **Persistent Storage:** Add a database (e.g., MongoDB) to save code history and room state. This would allow users to revisit past sessions and retain code after closing the room.
- **User Accounts & Security:** Implement user authentication, so rooms could be private and access-controlled. Users could save their projects and settings across sessions.
- **Advanced Execution Sandbox:** Integrate a secure sandbox (Docker-based or using restricted Node VM contexts) for running code, to safely allow more languages (Python, C++, etc.) and protect the server.

- **Code Linting & Error Highlighting:** Provide real-time code linting and syntax error underlining in the editor (using Monaco's diagnostics). This would help users catch errors before running code.
- **Chat and Voice:** Add a built-in chat or voice call feature for users to communicate while coding, further enhancing collaboration.
- **Mobile Support:** Optimize the interface for mobile or tablet, so users can join from any device.
- **Plugin/Extension Support:** Design an API so that third-party plugins (e.g., a debugger, formatter, or AI assistant) can be added to the editor.
- **Performance Improvements:** For large sessions, implement rooms across multiple server instances, possibly using Redis or WebRTC for peer-to-peer.
- **Undo/Redo History Sharing:** Currently, each client has its own undo stack. Future work could synchronize history so that any user can undo recent changes from others.

These enhancements would broaden CodeLink's applicability and robustness. With them, CodeLink could evolve into a full-featured collaborative IDE.

6.3 Contribution to Learning and Development

The development of CodeLink provided valuable experience in applying full-stack technologies to solve real-world collaboration problems. By working with tools like React, Node.js, Express, and Socket.IO, the project deepened our understanding of modern web development practices and real-time system design. Integrating the Monaco Editor, managing socket connections, handling execution safely, and ensuring synchronization across clients reflected real challenges faced in software engineering. Furthermore, engaging in user testing and iterative improvement fostered an appreciation for usability and performance in collaborative tools. Overall, CodeLink not only delivers a functional solution but also served as a platform for significant hands-on learning, preparing the team for future work in collaborative systems, web applications, and real-time software development.

REFERENCES

[1] Meta, React: A JavaScript library for building user interfaces, Meta Open Source, 2025. Available: https://opensource.fb.com/projects/react.

[2] GeeksforGeeks, Explain the V8 engine in Node.js, Mar. 2025. Available: https://www.geeksforgeeks.org.

[3] Express, Express—Fast, unopinionated, minimalist web framework for Node.js, Available: https://expressjs.com.

[4] Node.js Foundation, Node.js Documentation, "child_process," Node.js v23.x.x, 2024. Available: https://nodejs.org/api/child_process.html.

[5] Swati2904, Sync_Code_Studio – Sync Code Editor, GitHub, 2024. Available: https://github.com/swati2904/Sync_Code_Studio.

[6] Replit Inc., Replit Collaboration – Build software together, 2025. Available: https://replit.com/collaboration.

[7] CodePen, Collab Mode, CodePen, 2023. Available: https://blog.codepen.io.

[8] Crushing Tech Education, Design Google Docs – Operational Transformation | CRDT, Aug. 2023. Available: https://blog.crushingtecheducation.com.

[9] Crushing Tech Education, Understanding Conflict-Free Replicated Data Types (CRDTs), Aug. 2023. Available: https://blog.crushingtecheducation.com.

[10] A. Zagorskii, "Operational Transformations as an Algorithm for Automatic Conflict Resolution," Coinmonks, Medium, Jul. 2018. Available: https://medium.com/coinmonks.

[11] CodeStax.AI, "Operational Transformation: The Key to Real-Time Collaborative Document Editing," 2023. Available: https://codestax.medium.com.

[12] Microsoft, "Visual Studio Live Share: Real-Time Code Collaboration & Debugging," 2025. Available: https://visualstudio.microsoft.com/services/live-share.

[13] B. Brace, "Socket.io: Real-time Web Communication Made Easy Introduction," DEV Community, Nov. 2023. Available: https://dev.to.

[14] Socket.IO Documentation, "Socket.IO: Real-time WebSockets for Node.js," [Accessed: May 14, 2025]. Available: https://socket.io/docs/.

[15] Monaco Editor Documentation, “Monaco Editor: A Code Editor in the Browser,” [Accessed: May 14, 2025]. Available: https://microsoft.github.io/monaco-editor/.