WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

CodeShare: A Real-Time Collaborative Coding and Technical Interview Platform

Evangeline Aishwarya S[#1], Sabeetha M[#2], Ajay N S[#3], Ariya Prakash P[#4], Bala Subramani M[#5], Bhavana R[#6], Brindha Devi K[#7]

[#1] Mentor, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
[#2] Coordinator, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
[#3] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
[#4] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
[#5] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
[#6] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
[#7] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.

ABSTRACT

Modern remote hiring, pair programming and peer learning depend on several tools at once. Participants usually switch between a video-conferencing application, a shared document, an online compiler and a chat tool to complete a single coding session. CodeShare is a web-based platform that brings these functions into one browser workspace. Users create or join a room, edit code together in a Monaco-based editor with live cursors and selections, execute the code in one of 24 supported languages through the Judge0 execution service, exchange persistent chat messages and hold peer-to-peer audio and video calls.

The system is designed around a React single-page application and a Node.js backend using Express.js and REST APIs, together with Socket.IO for low-latency collaboration events and MongoDB for persistent storage. Account security is supported by e-mail OTP verification, bcrypt password hashing, JWT-based authentication and optional password-protected rooms. The platform also supports scheduled interview rooms, e-mail invitations and a dashboard of past and upcoming sessions.

Preliminary measurements with scripted clients show a mean code-synchronization latency of 1.04 ms, 0.99 ms and 1.37 ms for two, three and five clients (p95 at most 2.62 ms) and 460 of 460 successful deliveries with up to 100 concurrent sockets. However, none of five simultaneous-editing trials ended with identical editor content, which confirms the limitation of the current full-document broadcast strategy, and none of 100 code-execution attempts succeeded, so execution time could not yet be characterized. The session-recording module and socket-level authorization also remain to be completed. CodeShare provides a modular foundation for a centralized, browser-accessible platform for coding interviews and collaborative learning.

Keywords: collaborative coding; real-time collaboration; technical interviews; Socket.IO; WebRTC; Monaco editor; Judge0; MERN stack.

INTRODUCTION

Remote work and online education have made browser-based collaboration a routine part of software development. Hiring teams conduct live coding interviews, instructors review student code in real time, and developers pair-program across locations. Each of these activities needs the same ingredients: a shared editor, a way to run the code, a channel for discussion and, ideally, a record of what happened.

Traditional approaches often require interviewers and participants to use multiple tools. An interviewer may share a screen on a video call, paste a problem into a document, ask the candidate to run code in a separate online compiler and take notes elsewhere. This makes the session harder to manage, interrupts the flow of work and leaves no single history to review afterwards. A centralized platform can simplify this process by presenting the editor, output, conversation and video in one interface.

CodeShare is a real-time collaborative coding platform designed to provide such a workspace. The project uses React and Vite with the Monaco editor to develop the client interface, and Node.js with Express.js to implement the backend services. The client communicates with the backend through REST APIs for account and room management and through Socket.IO for real-time events. Code execution is delegated to the Judge0 online judge system, and audio and video are exchanged directly between browsers using WebRTC through PeerJS.

The main objective of CodeShare is to simplify remote coding sessions through a user-friendly and extensible platform. Its modular architecture allows additional capabilities to be incorporated as the project develops. The project also provides opportunities for future enhancements, such as conflict-free

collaborative editing, session recording and replay, interview question libraries and improved performance analysis. By combining real-time communication, sandboxed code execution and persistent session history, CodeShare seeks to make remote coding interviews and collaborative programming more accessible, organized and effective.

LITERATURE SURVEY

Collaborative editing requires several replicas of a document to remain consistent while users edit concurrently. Operational Transformation (OT), introduced for groupware by Ellis and Gibbs, transforms concurrent operations so that all replicas converge [4]. Conflict-free Replicated Data Types (CRDTs) achieve convergence by designing the data structure so that concurrent updates commute [5]. Both approaches are more robust than broadcasting the whole document, which is the simpler strategy used in the present version of CodeShare.

The Monaco editor provides syntax highlighting, multi-cursor support, decorations and language services in the browser and is widely embedded in web-based development environments [6]. Its decoration API makes it possible to draw remote participants' cursors and selections over the local document.

The WebSocket protocol provides a persistent full-duplex channel between browser and server [7]. Socket.IO builds on it with rooms, acknowledgements and automatic reconnection, which suits applications that broadcast events to groups of users [8].

Running untrusted code requires isolation, resource limits and time limits. Judge0 is an open-source online code execution system exposed through a REST API; it compiles and runs submitted source code and returns standard output, standard error, compiler output, execution time and memory use [9]. Delegating execution to such a service avoids running user code on the application server.

WebRTC enables peer-to-peer media exchange between browsers without plug-ins [10]. Peers still require a signalling mechanism and STUN or TURN servers for network traversal. PeerJS wraps the WebRTC API and offers a simple peer-identifier-based calling model [11].

JSON Web Tokens provide a compact, signed representation of claims for stateless authentication [12], and the bcrypt scheme hashes passwords with an adaptive cost and a per-password salt [13]. REST describes an architectural style in which clients interact with server resources through a uniform interface [1], and Node.js, Express.js, React and MongoDB allow a single language to be used across the complete application [2][3][14].

Based on these concepts, CodeShare is designed as a collaborative coding platform that combines a React interface with Node.js, Express.js, Socket.IO and REST API communication. The project focuses on real-time collaboration and code execution and aims to support future enhancements such as session replay and conflict-free editing, providing a foundation for more integrated and efficient remote coding sessions.

METHODOLOGY

Overview of Proposed Method

CodeShare is organized around the concept of a room. A room has a host, a list of participants, a programming language, a type (practice or interview), a status (scheduled, active or ended) and, optionally, a password. Everything that happens in a room, including edits, runs, messages and calls, is scoped by a six-character room identifier. The React client is built with Vite and Tailwind CSS, while the Express.js backend handles incoming requests and Socket.IO events.

The high-level workflow is:

User → React Client → REST API → Node.js Backend → MongoDB → API Response → Dashboard Display

For real-time collaboration and execution, the workflow is extended to:

User → Editor / Run → Socket.IO → Node.js Backend → Judge0 → Result → Room Broadcast + Database

The project development process consists of requirement analysis, architecture design, frontend development, backend implementation, API and socket integration, execution-service integration, testing and evaluation. Each component is tested independently before end-to-end integration.

Dataset Description

CodeShare does not use a fixed image dataset or a machine-learning training dataset as its primary input. Instead, it works with operational data created as users interact with the system. The data are stored in MongoDB collections, summarized in Table 1: user accounts, rooms, code execution records, chat messages, notifications and, for the planned recording feature, timestamped session events.

Table 1. CodeShare Data Collections

Collection	Key Fields	Description
User	name, email, hashed password, role, isEmailVerified, skills	Account and profile data
Room	roomId, host, participants, language, type, status, scheduledAt, isPrivate	Session definition, last code and output
Execution	roomId, user, code, language, input, output, timestamp	One record per code run
Message	roomId, user, text, timestamp	Persistent chat history
Notification	user, title, type, isRead, relatedRoom	In-app notifications
SessionRecord	roomId, participants, initialCode, events	Replay timeline (partial)

Data Preprocessing

Data processing in CodeShare involves receiving, validating and handling information between the client and the server. When a user performs an action, the React client sends a request to the backend through REST APIs or emits a Socket.IO event. The backend validates the input, performs the necessary operation and returns a response in JSON format, which the client converts into a readable interface.

Validation is applied at several points. Sign-up input is checked with express-validator: the name must be non-empty, the e-mail must be well formed and its domain must have MX records, and the password must contain at least six characters. Room creation requires a title of at most 100 characters, a valid room type, an ISO 8601 date for scheduled sessions and a password for private rooms. Chat messages are trimmed and limited to 1000 characters, and profile pictures are restricted to JPEG, PNG or WebP files of at most 2 MB.

Execution results are normalized before they reach the client. The Judge0 response is reduced to a common structure containing standard output, standard error, compiler output, status description, execution time and memory, so that the interface and the stored history use the same format for every language. Errors such as invalid requests, connection failures and execution timeouts are handled and reported to the user.

System Architecture

The CodeShare architecture follows a client-server approach consisting of a React client, a Node.js and Express.js backend, Socket.IO for real-time events and MongoDB for storage. The backend is organized in layers: routes map URLs to controllers, controllers hold the business logic, models define the database schemas, middleware performs authentication, e-mail verification checks and error handling, and services wrap external systems such as Judge0. This separation makes the application easier to maintain, extend and improve as new features are developed.

REST APIs are used for operations that are naturally request-response, such as sign-up, login, creating and joining rooms and loading history. Socket.IO is used for events that must reach all participants with minimal delay, including code changes, cursor positions, typing indicators, execution results, chat messages and call signalling. Audio and video travel directly between browsers through WebRTC.

Table 2. Principal Interfaces of CodeShare

Interface	Type	Function
/api/auth/*	REST	Sign-up, OTP verification, login, password recovery, profile
/api/rooms/*	REST	Create, join, invite, session lists, run and chat history
/api/dashboard, /api/notifications	REST	Statistics and notifications
code-change, cursor-change, typing	Socket	Live editing and presence
run-code, code-result, new-execution	Socket	Execution and shared results
send-message, new-message	Socket	Chat
video-call-start, peer-id-update	Socket	WebRTC call signalling

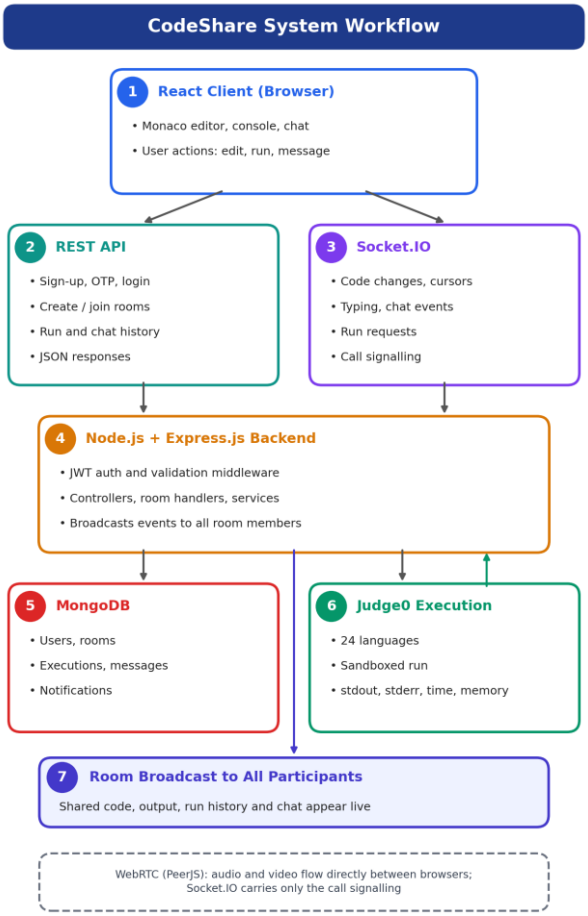


Fig. 1: System Workflow of CodeShare

Development and Testing Process

CodeShare focuses on developing, configuring and testing application components rather than training a machine learning model. The React client displays the room interface, while the Node.js and Express.js backend handles requests and provides responses through REST APIs and Socket.IO. The application is tested by checking server connectivity, API responses, real-time event delivery and communication between the client and backend.

Initially, the backend server is configured and executed together with a MongoDB instance and a Judge0 endpoint. The REST APIs are tested to confirm that they return appropriate JSON responses for valid and invalid input. The React client is then connected to the backend, and the communication is verified. Correct configuration of the API and socket addresses and of the allowed client origin is important for successful communication.

After integration, functional testing is carried out with two or more browser sessions in the same room. Different test cases check authentication, room access control, code synchronization, execution results, chat delivery and call setup. Issues such as connection failures, invalid responses and execution timeouts are analyzed and corrected during development.

Since CodeShare is a software platform rather than a machine learning system, it does not require model training or training datasets. Its development emphasizes implementation, API integration, functional testing, debugging and performance improvement to establish a reliable collaborative coding platform.

Evaluation Metrics

The evaluation metrics of CodeShare measure the performance, reliability and functionality of the application. The system is evaluated based on synchronization latency, execution round-trip time, request success rate, convergence correctness and connectivity.

1. Code Synchronization Latency: This metric measures the time between a keystroke in one client and the corresponding update becoming visible in another client in the same room. Lower values indicate a more natural shared-editing experience.

2. Execution Round-Trip Time: This metric measures the time from pressing Run to receiving the result in all clients, including server processing, Judge0 execution and result broadcast.

3. Request Success Rate: This metric measures the percentage of REST requests and socket operations that complete successfully out of the total made. It helps evaluate the reliability of communication between the client and backend.

4. Convergence and Connectivity: This metric checks that all participants end with identical editor content after simultaneous edits, and evaluates the success of chat delivery and WebRTC call establishment under different network conditions.

Room Dashboard and Workspace:

Fig. 2 shows the dashboard shown after sign-in. It summarizes total rooms, interviews, active rooms and completed sessions, lists recent sessions and upcoming interviews, and provides the Join Room and Create Room actions.

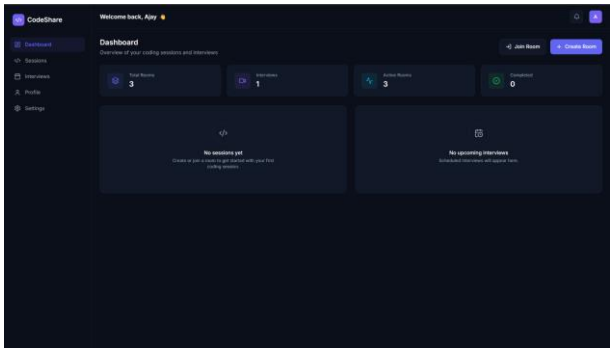


Fig. 2: CodeShare Dashboard

Fig. 3 shows the room workspace, with the Monaco editor on the left and the Console, History, Chat, Participants and Review panels on the right. The toolbar provides the room identifier, language selector, Start Call, Run and Leave controls.

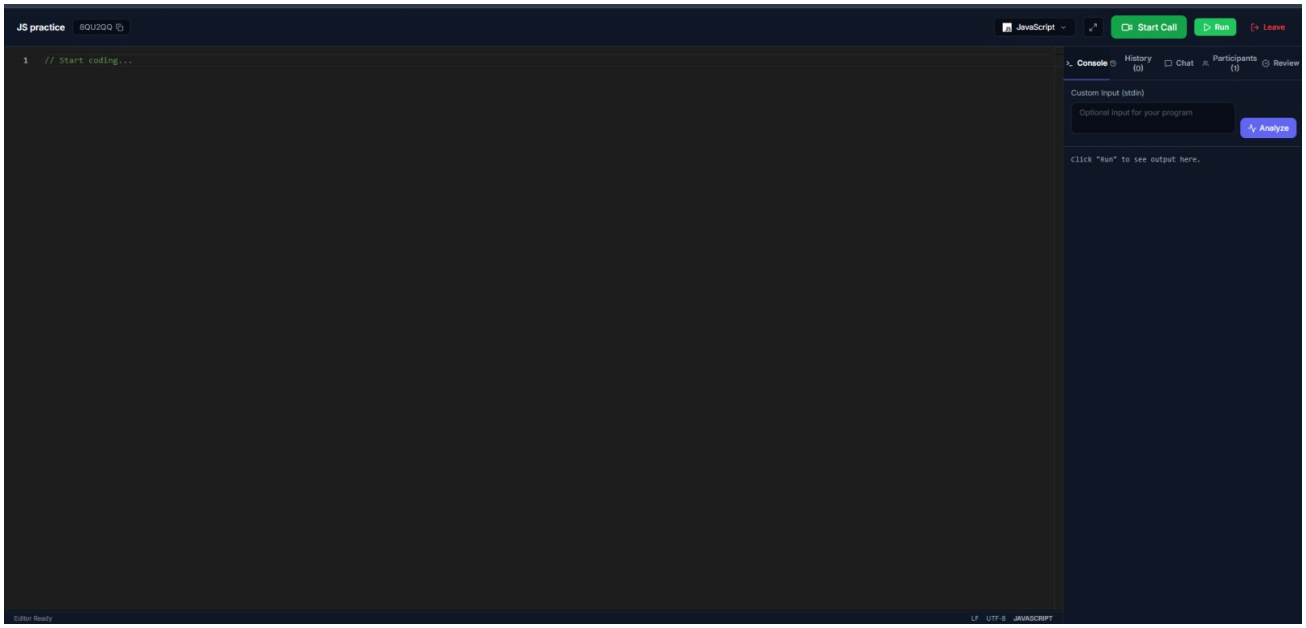


Fig. 3: Room Workspace with Editor, Console and Participants

Advantages of the Proposed Model

The proposed CodeShare system provides a centralized platform for remote coding sessions. It helps interviewers, instructors and developers work in a single interface instead of coordinating several tools. By organizing the editor, output, chat and video in one place, the system aims to make sessions easier to run and review. This centralized approach can improve accessibility and simplify routine collaboration activities.

Another important advantage is its use of modern technologies. React and Monaco provide the interface, while Node.js, Express.js and Socket.IO handle backend operations. Delegating execution to Judge0 keeps untrusted code away from the application server, and the language mapping makes it easy to add further languages. The scheduled interview type, e-mail invitations and verified accounts address practical needs of remote hiring. This separation of components makes the application easier to organize, maintain, test and improve as the project develops.

CodeShare also supports identification and troubleshooting of common problems. Execution history and chat history are stored, so every participant can review what was run and discussed. Because the stack is JavaScript throughout and relies on open components, the platform can be self-hosted and adapted by small teams and educational institutions.

Implementation Process:

The implementation of CodeShare involves developing the web client, configuring backend services and integrating the components through REST APIs and Socket.IO. React and Vite are used to implement the interface, while Node.js and Express.js provide the backend functionality. The client is configured with the backend address through environment variables so that it can communicate with the server.

Authentication. A new user registers with a name, e-mail and password. The password is hashed with bcrypt, and a six-digit OTP is generated, stored only as a SHA-256 hash and e-mailed to the user; it expires after ten minutes. The account becomes usable after OTP verification, when a JWT is issued. Protected routes read the token from the Authorization header, and the Socket.IO handshake performs the same verification. Creating, joining or inviting to a room also requires a verified e-mail address.

Rooms and scheduling. The server generates a six-character room identifier from a 32-character alphabet that omits ambiguous symbols and repeats the generation until the identifier is unused. A room may be a practice session, which starts immediately, or an interview, which may be scheduled for a future time. Private rooms store a bcrypt hash of the room password. The host can e-mail an invitation link that redirects authenticated users into the room.

Collaborative editor. The editor is the Monaco component. Local changes are emitted as code-change events and applied by the other members of the room. Cursor and selection positions are throttled to about five updates per second and drawn on other participants' editors as coloured cursors with name labels. A hash of the user identifier selects the colour from an eight-colour palette, and typing indicators show who is editing.

Code execution. When a participant presses Run, the client sends the code, language and optional input over Socket.IO. The server maps the language to a Judge0 identifier (24 languages are mapped), submits the program synchronously with a 20-second client timeout and receives the outcome. The server broadcasts an execution-started event, then the normalized result, stores an Execution record and saves the last code and output on the room.

Chat and video. Messages are validated, stored and broadcast with the author's name and picture, and the history is loaded when a participant enters the room. Video calling uses PeerJS: a participant who starts a call announces a peer identifier to the room through Socket.IO, other participants call the peer, and a gallery of remote streams is displayed with audio and video controls. Public STUN servers are used for network traversal.

Session recording (under development). A SessionRecord model stores the initial code, participants and a timeline of events for replay, and server handlers append events while a recording is active. The REST endpoints for creating and finalizing recordings and the timeline controls of the replay interface are not yet integrated, so this feature is treated as future work.

Testing is performed to verify that the backend starts correctly, the authentication and room endpoints respond as expected and the client can communicate with the backend. Additional tests must verify real-time behaviour under multiple users and confirm that the client processes responses correctly. Connectivity problems, incorrect server addresses and timeouts can be investigated by checking server logs, network configuration and API responses.

Code Synchronization Latency Graph

The synchronization latency graph represents the time taken for an edit made by one participant to appear in another participant's editor. Lower values indicate faster communication between clients through the server. Latency was recorded by scripting repeated edits between clients on the same machine and measuring the duration of each update. For each of the two-, three- and five-client configurations, 120 edits were measured; the results are summarized in Table 3 and plotted in Fig. 4.

Table 3. Code Synchronization Latency (120 edits per configuration)

Clients	Mean (ms)	Median (ms)	p95 (ms)	Min / Max (ms)	Edits
2	1.0367	0.8671	2.2509	0.2875 / 2.8365	120
3	0.9941	0.7185	2.2696	0.2963 / 7.9224	120
5	1.3671	1.2355	2.6141	0.3749 / 3.3907	120

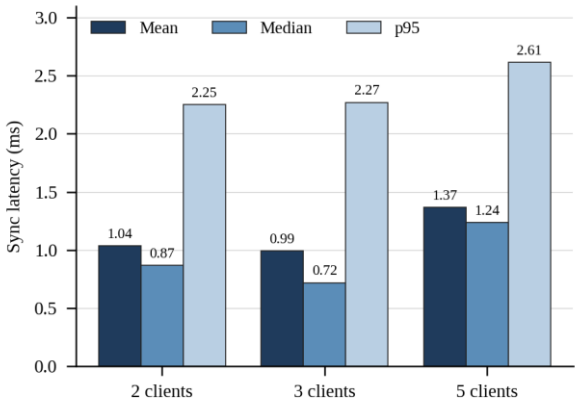


Fig. 4: Code Synchronization Latency by Number of Clients

With two clients the mean latency was 1.0367 ms (median 0.8671 ms, p95 2.2509 ms). Three clients gave a similar mean of 0.9941 ms, whereas five clients raised the mean to 1.3671 ms, about 32% above the two-client value, and the median to 1.2355 ms. The p95 remained below 2.7 ms in every configuration. The largest single observation, 7.9224 ms, occurred in the three-client run; since that run's p95 is only 2.2696 ms, it is an isolated outlier rather than a typical delay. Because the clients ran on the same machine, these values reflect server and event-loop processing and exclude real network delay; they should be regarded as a lower bound for deployed use.

Concurrent Connection Load

To examine scalability, synchronization samples were collected while 10, 25, 50 and 100 sockets remained connected (Table 4, Fig. 5). The samples were sent sequentially by room pair, and every message was delivered: 25 of 25, 60 of 60, 125 of 125 and 250 of 250 at the four levels, or 460 of 460 in total. Mean latency decreased from 1.3899 ms at 10 sockets to 1.1087 ms at 50 sockets and then rose to 1.7946 ms at 100 sockets, about 62% above the 50-socket value, while the p95 latency reached 2.8873 ms. The small sample sizes (25 to 250 deliveries) mean that the lower values at 25 and 50 sockets probably reflect sampling variation rather than a real improvement. Because the samples were sent sequentially rather than as simultaneous bursts, these results demonstrate connection capacity, not throughput under contention.

Table 4. Synchronization Latency under Concurrent Socket Load

Sockets	Mean latency (ms)	p95 latency (ms)	Successful deliveries
10	1.3899	2.1522	25 / 25
25	1.2452	2.5763	60 / 60
50	1.1087	2.0690	125 / 125
100	1.7946	2.8873	250 / 250

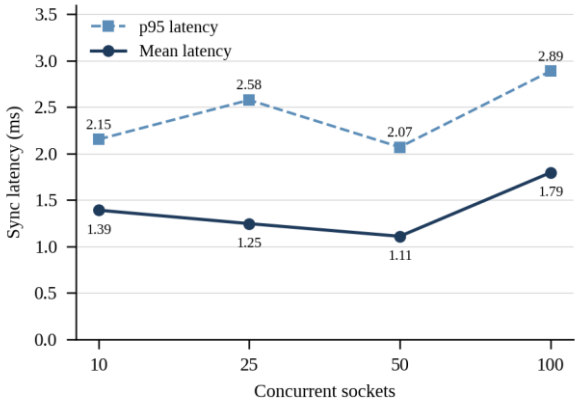


Fig. 5: Synchronization Latency versus Concurrent Sockets

Execution Time Graph

The execution time graph illustrates the round-trip time of code runs for different languages. It includes server processing, Judge0 execution and result broadcast to all participants. Five languages (JavaScript, Python, Java, C++ and C) were tested with 20 attempts each. None of the 100 attempts completed successfully: every attempt returned an error response. The values in Table 5 and Fig. 6 are therefore round-trip times of error responses and must not be read as program execution times.

Table 5. Execution Error-Response Round-Trip Time by Language

Language	Mean RTT (ms)	Sample SD (ms)	Attempts	Successful runs
JavaScript	398.600	501.812	20	0
Python	320.342	442.176	20	0
Java	491.495	698.685	20	0
C++	342.608	454.697	20	0
C	488.321	599.039	20	0

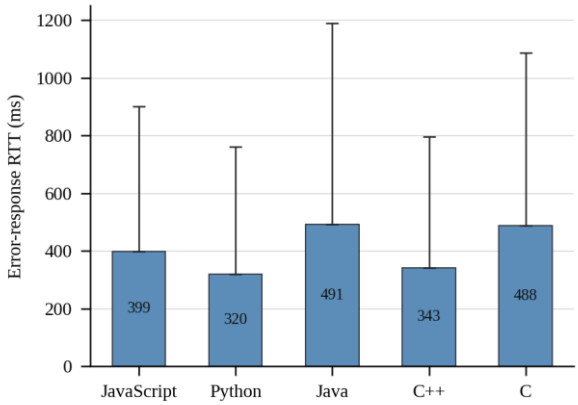


Fig. 6: Error-Response Round-Trip Time by Language (bars: mean; whiskers: +1 sample SD; 0 successful runs)

The mean error-response time ranged from 320.342 ms for Python to 491.495 ms for Java, with sample standard deviations between 442 and 699 ms. Because the spread within each language is larger than the differences between languages, no ranking of languages can be inferred. The data show that a failed execution is reported back to the clients within roughly 0.3 to 0.5 s on average. The cause of the failures was not established in this test (for example, an unreachable or unconfigured Judge0 endpoint in the test environment) and must be investigated before successful-run execution time, memory use and per-language performance can be reported.

Convergence and Request Success

Convergence was tested in five trials of about 30.8 s each, in which every client issued between 942 and 959 edits (Table 6). In none of the five trials were the final editor contents identical (0 of 5). This outcome is expected from the present design, in which each code-change event carries the complete document and the latest broadcast wins, and it provides measured support for adopting OT or CRDT-based merging. Request reliability was higher: 10,758 of 10,858 REST and socket operations succeeded, a success rate of 99.079%. The 100 unsuccessful operations equal in number the 100 failed execution attempts (5 languages × 20 attempts), which suggests that the failures are confined to code execution; the server logs would need to be checked to confirm this.

Table 6. Convergence Trials under Simultaneous Editing

Trial	Duration (s)	Edits from each client	Final contents identical?
1	30.770	959	No
2	30.754	942	No

Trial	Duration (s)	Edits from each client	Final contents identical?
3	30.778	945	No
4	30.767	952	No
5	30.778	959	No

Interpretation

The CodeShare system uses a React client and a Node.js backend connected through REST APIs and Socket.IO. The implemented modules cover authentication, room management, shared editing, multi-language execution, chat, video calling and dashboards. These observations demonstrate that the main frontend and backend components have been implemented. The measurements show low synchronization latency (mean 0.99 to 1.37 ms, p95 below 2.7 ms), a request success rate of 99.079% and complete delivery with up to 100 concurrent sockets. They also reveal two defects: simultaneous edits did not converge in any of five trials, and code execution did not succeed in any of 100 attempts. Further testing is required for successful-run execution time, video-call setup and performance over real networks.

Formula

Request Success Rate (%) = (Successful Requests / Total Requests) × 100

For example, if 98 out of 100 requests succeed, the request success rate is 98%. This is an illustrative example, not a measured CodeShare result. Applied to the measured data, the rate is 10,758 / 10,858 × 100 = 99.079%.

Synchronization Latency = Time of update received at Client B − Time of edit at Client A

Performance Interpretation

The performance of CodeShare can be evaluated by examining synchronization latency, execution round-trip time, request success rate and client-backend connectivity. Successful execution responses indicate that the execution service is reachable. Latency measurements help evaluate the speed of collaboration, while request success rates provide information about reliability. Connectivity testing can reveal problems caused by incorrect configuration, unavailable servers or network restrictions.

These metrics provide a practical basis for evaluating the functionality and reliability of CodeShare. Repeated tests under different conditions can help identify performance limitations and guide future improvements. The measurements in Tables 3 to 6 provide a first quantitative basis for these metrics. They were obtained with scripted clients in a controlled setup and should be repeated on real networks before conclusions are drawn about the overall performance of the system.

Table 7. Performance Evaluation of CodeShare

S. No.	Evaluation Metric	Purpose	Current Observation
1	Authentication and OTP	Verify secure account creation	Implemented; test results to be recorded
2	Room Creation and Joining	Verify session management	Implemented; negative tests required
3	Sync Latency	Measure edit propagation time	Measured: mean 0.99–1.37 ms, p95 ≤ 2.61 ms
4	Execution Round-Trip	Measure run-to-result duration	Not characterized: 0 of 100 runs succeeded
5	Request Success Rate	Evaluate request reliability	Measured: 99.079% (10,758 / 10,858)
6	Video Call Setup	Verify peer connection	Requires testing across networks
7	Concurrent Capacity	Evaluate scalability	460 / 460 deliveries up to 100 sockets
8	Session Recording	Post-session review	Partially implemented
9	Convergence	Verify identical editor content	Failed: 0 of 5 trials identical

Result:

The CodeShare application provides a centralized interface in which users can create or join rooms, edit code together, run it in multiple languages and view shared output and run history. Users can see participants, remote cursors and typing indicators, exchange chat messages and start video calls, helping them understand the current state of the session.

The room module supports practice and interview sessions, private rooms with passwords and scheduled interviews that appear in the participants' dashboards. The execution module returns output, errors, execution time and memory use to every participant at the same time, enabling interviewers to follow exactly what was run.

The results demonstrate that CodeShare provides a unified interface for coding, communication and review. This helps reduce the number of tools needed for remote sessions and supports faster, more organized collaboration. Quantitative measurements (Tables 3 to 6) show a p95 synchronization latency below 2.7 ms, a request success rate of 99.079% and complete message delivery with up to 100 concurrent sockets. They also show that simultaneous editing does not yet converge and that code execution failed in the test environment.

Room Interface and Account Screens:

Fig. 7 shows the Create New Room dialog, in which the host sets the title, room type (practice or interview), programming language, optional schedule, private-room option and optional Anti-Cheat monitoring.

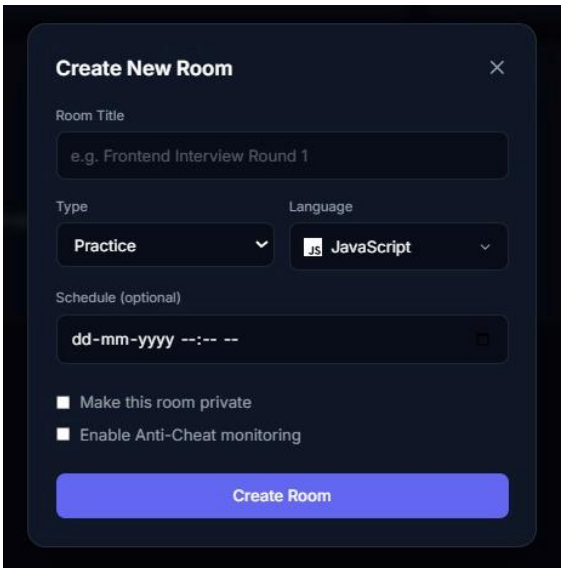


Fig. 7: Create-Room Dialog with Interview Scheduling

Fig. 8 shows the Join Room dialog, which accepts a room link or identifier and a password for private rooms, and Fig. 9 shows the profile page where users manage their name, bio and picture.

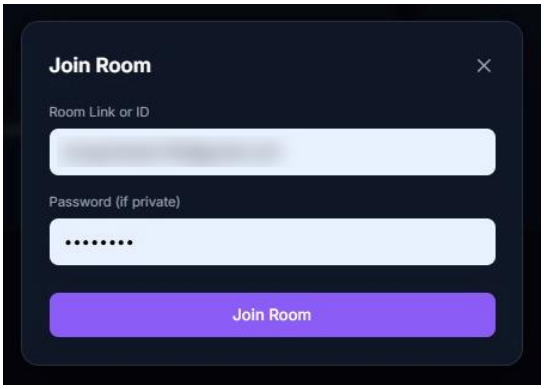


Fig. 8: Join-Room Dialog for Private Rooms

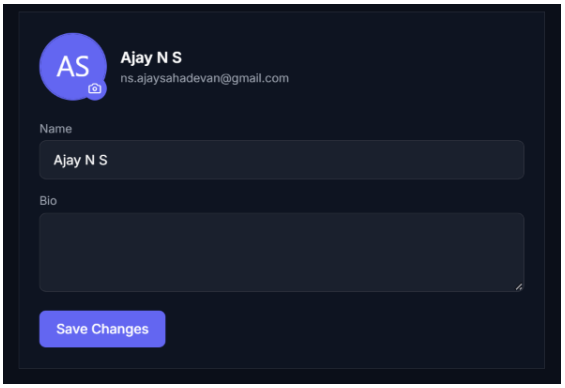


Fig. 9: User Profile Management

Limitations and Future Work

Conflict handling. Each code-change event carries the complete document, so under simultaneous edits the latest broadcast wins and some keystrokes may be overwritten. OT or CRDT-based merging would guarantee convergence. In the convergence test, none of the five trials ended with identical editor contents (0 of 5).

Socket authorization. Socket connections are authenticated with a JWT, but the join-room handler does not check room membership or password; this check, already performed in the REST join call, should be repeated on the socket handler.

Other limitations. Session recording needs to be completed; the password-reset endpoint should deliver the link by e-mail rather than in the response; rate limiting and execution quotas are not yet implemented; only STUN servers are configured, so a TURN relay is needed for restrictive networks; the execution path must be diagnosed because none of the 100 test executions succeeded; and usability evaluation and tests over real networks have not yet been carried out.

Future enhancements include CRDT-based collaborative editing, completed session recording with shareable replay, socket-level authorization, a TURN server, a shared whiteboard, an interview question library with automated test cases, evaluator scorecards, and containerized deployment with horizontally scaled Socket.IO.

CONCLUSION

This project presents CodeShare, a real-time collaborative coding and technical interview platform designed to provide a centralized interface for coding, execution, communication and review. The system combines a React client with a Node.js backend, REST API communication and Socket.IO real-time events, with MongoDB for persistence and Judge0 for sandboxed execution.

Security measures include OTP-verified accounts, hashed credentials, token-based authentication and password-protected rooms. Shared editing with remote cursors, multi-language execution, persistent chat and peer-to-peer video are implemented. However, the current version has known limitations in conflict resolution, socket authorization and session recording, and the measurements show that simultaneous edits do not yet converge (0 of 5 trials) and that code execution did not succeed in the test environment. Synchronization latency was low (mean 0.99 to 1.37 ms) and delivery was reliable with up to 100 concurrent sockets, but further work is necessary to establish reliable behaviour across real network environments.

Future enhancements may include CRDT-based editing, completing session recording, improving authorization and abuse protection, and deploying the platform to a cloud environment. In conclusion, CodeShare aims to simplify remote coding interviews and collaborative programming by combining editing, execution and communication within a unified platform. Continued development and systematic testing can help evolve the project into a more comprehensive solution.

REFERENCES

1. R. T. Fielding (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.

2. React Documentation. Meta Open Source. https://react.dev

3. Node.js Documentation and Express.js Documentation. https://nodejs.org and https://expressjs.com

4. C. A. Ellis and S. J. Gibbs (1989). Concurrency control in groupware systems. Proc. ACM SIGMOD International Conference on Management of Data, pp. 399–407.

5. M. Shapiro, N. Preguiça, C. Baquero and M. Zawirski (2011). Conflict-free replicated data types. Proc. Symposium on Self-Stabilizing Systems (SSS), LNCS vol. 6976, Springer, pp. 386–400.

6. Monaco Editor Documentation. Microsoft. https://microsoft.github.io/monaco-editor

7. I. Fette and A. Melnikov (2011). The WebSocket Protocol. RFC 6455, IETF.

8. Socket.IO Documentation. https://socket.io/docs

9. Judge0 Documentation. https://judge0.com

10. WebRTC 1.0: Real-Time Communication Between Browsers. W3C Recommendation. https://www.w3.org/TR/webrtc/

11. PeerJS Documentation. https://peerjs.com

12. M. Jones, J. Bradley and N. Sakimura (2015). JSON Web Token (JWT). RFC 7519, IETF.

13. N. Provos and D. Mazières (1999). A future-adaptable password scheme. Proc. USENIX Annual Technical Conference, pp. 81–91.

14. MongoDB and Mongoose Documentation. https://www.mongodb.com/docs and https://mongoosejs.com