WWW.IJRPR.COM

International Journal of Research Publication and Reviews

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

Project Hub: An AI-Assisted Web Platform for Student Project Collaboration with Peer-To-Peer Screen Sharing

Jayasimha S[1]*, Jeeva A[2], Jeevanantham K[3], Jayasree M[4], Gayathri V[5]

[1-4] Student, [5] Assistant Professor,
Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, India
* E-Mail Id: jayasimha.s@example.com

ABSTRACT

Student project teams work with a handful of tools that were never designed to talk to each other: the plan lives in a spreadsheet, the discussion in a group chat, the code in a shared drive or repository, the design review in a phone call, and the small questions in a search engine. Every move between them costs attention, and the teams that lose the most are those without a laboratory machine or a paid subscription. This paper presents Project Hub, a browser-based collaboration platform built for a single engineering campus that gathers a project board, a media feed, direct messaging and voice calls, topic channels with an AI assistant, a shared code workspace and remote screen sharing into one application. The design keeps the server deliberately small: a 572-line Node.js process with no runtime dependencies serves the static application, streams assistant replies from an upstream model provider so that the API key never reaches the browser, and relays WebRTC session descriptions and ICE candidates between peers. The screen video itself is exchanged directly between browsers over DTLS-SRTP and never passes through the server. The client is a nineteen-module single-page application in plain JavaScript that persists a versioned snapshot to localStorage and keeps open tabs consistent through BroadcastChannel. The system is verified by 115 automated checks executed under jsdom, covering routing, persistence, the feed, projects, messaging, the assistant panel, the remote-sharing state machine and the code workspace. Results show that a dependency-free server, a local-first client and peer-to-peer media together deliver the collaboration features students actually use, at negligible hosting cost, while exposing clear limits: without a TURN relay, peers behind symmetric NAT cannot connect, and browser storage bounds the size of the shared history.

Keywords: Project Collaboration, WebRTC, Peer-to-Peer Screen Sharing, Server-Sent Events, Large Language Models, Single-Page Application, Educational Technology, Local-First Software.

1. INTRODUCTION

Project work has become the principal way engineering students learn to build software. A team of three to five students is given a semester, a problem statement and a guide, and is expected to produce a working system, a report and a demonstration. The technical content of such a project is usually well supported: the department provides compilers, laboratories and, increasingly, free access to a large language model. What is not supported is the coordination around the work. A team of four students in different hostel blocks has to agree on a scope, divide tasks, review each other's code, resolve blockers at eleven at night before a review, and keep a guide informed, and it has to do all of this with whatever consumer tools happen to be at hand.

In practice that means a spreadsheet for the task list, a messaging group for discussion, a shared drive for documents, a screen-share during a call for reviews, and a browser tab for questions. Each of these tools is good on its own; together they impose two costs that this work takes seriously. The first is context switching: the state of a project is spread across four applications and none of them knows about the others, so a student must reconstruct it from memory. The second is inequality of access: collaborative code editing, remote support sessions and AI assistance are typically features of paid products, and a student who has only a borrowed laptop and a metered connection is quietly excluded from the way the rest of the team works.

Three capabilities of the modern browser make a lighter alternative practical. WebRTC, standardised as a W3C Recommendation in 2021, gives any page direct access to camera, microphone and, through the Screen Capture specification, the display surface, with media protected by DTLS-SRTP and negotiated peer to peer [5], [6]. Server-Sent Events give a plain HTTP server a one-way push channel without a WebSocket library or a message broker [7]. Public model providers expose chat completions over HTTPS, so a small server can act as a key-holding proxy and stream tokens to the browser [11], [12]. None of these technologies requires a native application, an install step or a paid licence on the client side.

Project Hub is built on exactly those three capabilities. It is a single-page application in plain JavaScript, served by a Node.js process whose only imports are the standard library modules *http*, *fs* and *path*. A student opens a URL and finds a project board with tasks, roles and comments, a feed with images and video, direct messages and calls, topic channels in which an assistant answers with the project's own context, a shared code workspace that runs in a

sandboxed frame, and a remote support page on which one student shares a screen with up to three classmates by reading out a nine-digit session identifier. Nothing is installed, no account is created outside the campus, and the video never touches the server.

The contributions of this paper are four. First, a design in which the server is reduced to static files, a model proxy and a signalling relay, so that the whole backend is one file of 572 lines with no dependency tree to audit or update. Second, a local-first client in which the complete application state is a versioned snapshot in localStorage that is broadcast to sibling tabs, which makes the platform usable on a single machine and keeps the server free of user data. Third, a screen-sharing flow built on the browser's own capture picker, a nine-digit session identifier and an explicit host approval step, with the session description and ICE exchange carried over Server-Sent Events and the media carried directly between peers. Fourth, an account of what such a system can and cannot do, measured against 115 automated checks, including the cases in which browser security policy prevents capture altogether.

The rest of the paper is organised as follows. Section 2 reviews the related work in computer-supported cooperative work, browser real-time communication, local-first storage, assistants in education and sandboxed execution, and states the gap this work fills. Section 3 states the problem. Section 4 describes the architecture, the data model, each subsystem and the verification method. Section 5 reports the results and discusses them, including the limits of the approach. Section 6 concludes and lists the work that remains.

1.1 Research Objectives

The objectives of this work are:

1. To design a single browser-based environment that carries the coordination of a student project team — tasks, roles, discussion, files, code and reviews — instead of scattering it across unrelated applications.
2. To implement real-time screen sharing between students without installing software on either side, keeping the shared video off the server and off any third-party service.
3. To integrate an AI assistant whose credentials remain on the server and whose replies stream into the interface with the project's own context attached.
4. To keep the deployment small enough for a single campus machine: one process, no runtime dependencies and no database to administer.
5. To verify the system with an automated suite so that the behaviour described in this paper is reproducible rather than anecdotal.

2. LITERATURE REVIEW

Five bodies of work bear on this system: studies of cooperative work and student teamwork, the standards and practice of browser real-time communication, transport choices for signalling, local-first storage and multi-tab consistency, the use of large language models as learning support, and the isolation of untrusted code in the browser. Each is reviewed below, followed by the gap that remains.

2.1 Computer-Supported Cooperative Work and Student Teamwork

Grudin's history of computer-supported cooperative work observes that groupware succeeds or fails on the relation between who does the work and who receives the benefit: systems that ask one participant to maintain structure for the benefit of others are abandoned [1]. Student project teams are precisely such a setting. The task list benefits the guide and the team lead, while the student who updates it receives nothing immediate in return, which is why boards kept in spreadsheets go stale within a fortnight. The lesson adopted here is that structured artefacts must be produced as a by-product of work that already has to happen — a comment on a project, a message in a channel, a shared screen during a review — rather than as a separate administrative chore. Later CSCW literature adds the observation that awareness of who is present and what has changed is often more valuable than the artefacts themselves, which motivates the live feed, presence indicators and cross-tab synchronisation in this design.

2.2 Peer-to-Peer Real-Time Communication in the Browser

WebRTC is the standard that made real-time media a browser primitive rather than a plugin. The W3C specification defines the RTCPeerConnection API, the offer/answer negotiation model inherited from SIP, and the use of DTLS-SRTP for media protection [5]; Johnston and Burnett give the practical account of how those APIs behave in deployment [2], and RFC 8825 sets out the protocol stack as a whole [3]. Two properties matter for an educational platform. First, media is peer to peer wherever the network allows it, so the operator pays for signalling rather than for bandwidth. Second, the capture surface is chosen by the human at the sending end through a browser-controlled picker, defined by the Screen Capture specification [6], which means a viewer can never silently watch a screen: the sharing student must pick a tab, a window or a display, and the browser shows that the surface is being shared. Both properties are used deliberately in Section 4.5.

2.3 Transport for Signalling

WebRTC deliberately does not standardise signalling, because the session description and the ICE candidates are just text that two peers must exchange; RFC 8445 defines how the candidates are gathered and paired [4]. Implementations therefore choose a transport. WebSocket is the common choice, but

it adds a second protocol, a handshake library on the server and, in most deployments, a stateful connection registry that must be replicated when the service is scaled. Server-Sent Events require only a long-lived HTTP response with *text/event-stream* framing and automatic client reconnection, which the HTML standard specifies [7], while the peer-to-server direction can be an ordinary POST, in the REST style described by Fielding [10]. For a campus deployment in which a room holds four participants, that combination removes the dependency entirely: the relay in this system is a plain request handler with no library and no persistent session store.

2.4 Local-First Storage and Multi-Tab Consistency

Kleppmann and colleagues argue for local-first software, in which the copy of the data on the user's device is the primary copy and the network is an optimisation [8]. The argument is usually made about latency and ownership, but it also fits a campus tool: a student should be able to open the board, read the task list and draft a reply on a machine that is momentarily offline, and the operator should not have to run a database for a class of sixty. The browser offers two primitives that make a simple version of this practical: the synchronous localStorage store, whose quota behaviour Grigorik documents alongside the rest of the browser's storage and networking stack [9], and BroadcastChannel, which delivers a message to every other tab of the same origin in the same profile. The design in Section 4.3 uses the first as the durable snapshot and the second as the propagation channel, with a content fingerprint to stop a freshly opened tab from overwriting newer state with its seed data.

2.5 Large Language Models as Learning Support

The arrival of capable chat models in education has produced both enthusiasm and caution. Kasneci and colleagues survey the opportunities — individualised explanation, feedback at scale, help with the mechanics of writing and coding — alongside the risks of hallucination, over-reliance and inequity of access [11]. Two design consequences follow. The assistant in this system is deliberately scoped: it answers inside a project channel with that project's tasks, roles and recent discussion as context, so its replies are about the team's actual work rather than general knowledge, and it is presented as one participant among several rather than as an oracle. The second consequence is architectural: a key held in a browser tab is a key shared with every student who opens the developer tools, so model access is proxied by the server, and the browser receives only the streamed text [12]. A public provider interface such as OpenRouter allows the model to be changed by configuration rather than by code, which keeps the platform neutral between vendors.

2.6 Sandboxed Execution of Untrusted Code

Several platforms now let students run code in the browser. Executing student-written JavaScript on the same origin as the application would be indefensible: the script would share the storage, the session and the DOM of the host page. The HTML standard's sandbox attribute addresses this by placing the framed document in an opaque origin, withholding scripts, forms, popups and top-level navigation unless each is granted explicitly [7]. The workspace in Section 4.7 uses a sandboxed frame with scripts enabled but without same-origin access, and communicates with it only through posted messages, so a runaway loop or a deliberate attempt to read the host's storage fails at the origin boundary.

2.7 Research Gap

The literature describes each ingredient well but rarely assembles them for the setting that matters here. Institutional learning-management systems carry announcements and grades but not a shared editor or a screen share; general messaging tools carry conversation but not tasks or code; collaborative editors are excellent for a document and poor for a project board; commercial remote-support products solve screen sharing but require an installation on the viewer's machine, which on a campus network is often blocked. What is missing is an account of a single, installable-nothing environment in which the ordinary coordination of a student project — planning, discussing, coding, reviewing and asking — happens in one place, with media kept off the server and a cost that a department can carry. This paper reports such a system, its measured behaviour and its limits.

3. PROBLEM STATEMENT

A student team of four, working on a semester project, must coordinate a plan, a discussion, an artefact and a review. In the current practice of most departments each of these lives in a different consumer application, and there is no shared record that links them. Three consequences follow. The task list drifts away from the conversation, so the board says one thing and the group chat another. Code changes are exchanged as archives over messaging, which produces the familiar sequence of final, final-2 and final-with-fix file names. Reviews happen over a phone call in which the guide can hear but not see, or over a screen share that requires the viewer to install a client that the campus network may block.

Existing alternatives do not close the gap. Institutional platforms are built for administration rather than for building; general productivity suites require an account outside the campus and store student work abroad; professional developer platforms assume a repository workflow that first-year teams have not yet been taught; and commercial remote-support tools are designed for a technician assisting a customer, not for four classmates reviewing a report.

At the same time, the browser already contains the primitives needed to close the gap: a capture API, a peer connection, a push channel over ordinary HTTP and a durable local store. The problem this work addresses is therefore one of assembly and restraint rather than of invention: to combine those primitives into a platform that a department can host on one machine, that a student can use in a tab, and that keeps both the media and the model credentials away from the browser and the server respectively. In summary:

- **Fragmentation.** Planning, discussion, artefacts and review are held in unrelated applications, with no shared context and no reliable record of what was decided.
- **Cost and access.** The features that matter most for a review — live screen sharing, simultaneous editing, AI assistance — are usually sold, installed or both, which excludes students on borrowed hardware or a restricted network.
- **Operational weight.** A department that wants to offer such a service is usually asked to run a database, a message broker, a media server and a dependency tree, none of which is proportionate to a class of sixty students.

4. MATERIALS AND METHODS

Project Hub is a three-tier system with an unusually thin middle tier. The client is a single-page application in plain JavaScript; the server is one Node.js file; the external services are a model provider reached through that server and public STUN servers used only for address discovery. Fig. 1 shows the arrangement. The client holds the application state, renders every view and negotiates peer connections. The server does three things and nothing else: it serves static files, it proxies assistant requests so that the API key stays server-side, and it relays signalling messages between browsers. Screen video and audio are represented by the dashed path at the bottom of the figure: they leave one browser and arrive at another, and the server never sees a frame.

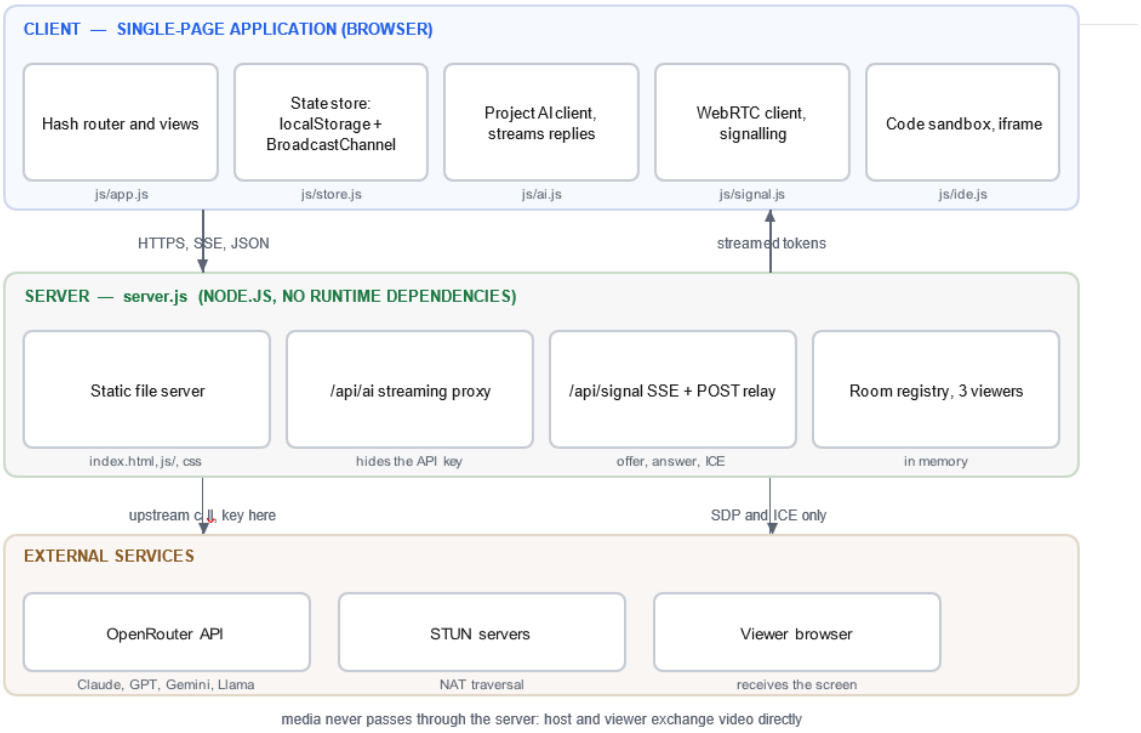


Fig. 1 *Three-tier architecture. The server is limited to static files, the model proxy and the signalling relay; media is exchanged directly between browsers.*

4.1 Technology Stack

The stack was chosen so that a single machine can host the platform and a single student can read the source. Table 1 lists the layers. No layer requires a build step: the browser loads the modules as they are written, which keeps the deployed artefact identical to the readable source and removes a toolchain that would otherwise have to be installed before the project can be run.

Table 1 *Implementation stack.*

Layer	Technology	Role in the system
Presentation	HTML5, CSS custom properties	Layout, theme tokens, light and dark schemes, responsive rules
Interaction	JavaScript (ES2019), hash router	Nineteen view modules; delegated events; no framework
State	localStorage, BroadcastChannel	Versioned snapshot; cross-tab propagation
Real-time media	WebRTC 1.0, DTLS-SRTP, STUN	Screen and voice transport, peer to peer
Signalling	Server-Sent Events, HTTP POST	Offer, answer and ICE exchange

Layer	Technology	Role in the system
Assistance	OpenRouter chat completions	Model access through a server-side proxy
Execution	Sandboxed iframe	Student code with an isolated origin
Server	Node.js http, fs, path	Static files, proxy, relay; no dependencies
Verification	jsdom test harness	115 automated checks over the client

4.2 Client Application and Routing

The client is organised as nineteen modules, each responsible for one view or one cross-cutting service, sharing a single namespace object. Navigation is a hash router: the fragment of the URL is parsed on load and on every change, and the matching view is rendered into the main region while the shell — sidebar, top bar, presence — stays mounted. This choice has three practical consequences for a campus deployment. Deep links survive a reload without server-side rewrite rules, so the static file server needs no route table. The back button behaves as students expect.

nd the whole application works from a single HTML document, which is what allows it to be demonstrated from a USB stick if the network fails during a review.

Interaction is handled by delegation rather than by per-element listeners: one handler per region reads a data attribute from the event target and dispatches to the module that owns that action. Views are re-rendered from state after a change, and the render functions are pure with respect to the model, so the same state always produces the same screen. This discipline is what makes the automated checks in Section 4.10 possible: a test can set a state, dispatch a click, and assert on the resulting document without a running server.

4.3 Data Model and Local-First Persistence

The model is deliberately relational but flat, as Fig. 2 shows. A student record carries identity and presence. A project carries a name, a track, a lead, a member list, an open flag and four child collections: tasks with an assignee and a completion flag, open roles with notes and expressions of interest, comments with an author and a timestamp, and a code workspace holding a script, a stylesheet and a readme. Conversations and topic channels hang off students rather than off projects, because the same two students may talk about several projects in one thread. Feed posts are campus-wide and reference their author, media and replies.

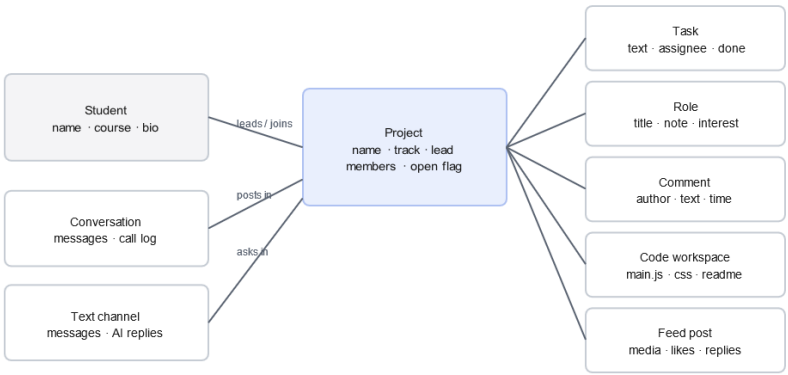


Fig. 2 *Entity model. Projects own tasks, roles, comments and a code workspace; students own conversations and channel memberships.*

Persistence follows the local-first argument of Section 2.4. After any mutation the state is serialised into a snapshot carrying a schema version, and the write is debounced so that a burst of keystrokes produces one write rather than forty. The snapshot is also broadcast on a named channel, which makes a second tab behave as a second signed-in student; a tab that receives a snapshot adopts it without echoing it back. Two details were necessary to make this stable in practice. First, a fresh tab must not overwrite newer saved state with its seed data, so a tab that has just booted stays silent until it has either loaded a snapshot or performed a user action. Second, uploaded photographs and videos are object URLs that die with the document, so they are stripped from the written snapshot while the post and its text remain; a quota guard catches the four-megabyte ceiling that browsers impose per origin and warns once instead of failing silently.

4.4 AI Assistance Through a Server-Side Proxy

The assistant is reached through a single endpoint on the platform's own server. The browser posts the conversation and the project context; the server attaches the provider key from its environment, forwards the request upstream and relays the response as it arrives, so the interface can render tokens as

they are produced rather than waiting for a complete answer. The key exists only in the server process. A student who opens the developer tools finds the proxy endpoint and nothing else, which is the difference between a demo and a service that can be exposed to a class.

Two further decisions are worth recording. The endpoint reports whether a key is configured, so the interface can explain the situation and accept a key pasted by the instructor for a single tab rather than failing with an opaque error. And the model identifier is configuration rather than code, so a department can move between providers without editing the application; a single public provider interface was used in development for that reason.

4.5 Peer-to-Peer Screen Sharing

Remote support is the feature that most clearly demonstrates the architecture. A student who needs help opens the remote page and chooses to share; the browser presents its own picker, the chosen surface becomes a media stream, and the page displays a nine-digit session identifier that can be read aloud in a classroom. A classmate opens the same page, enters the identifier and a display name, and a join request reaches the host through the server's event stream. The host sees who is asking and allows or declines. Only after approval does the host create a peer connection, produce an offer, and send it through the relay; the viewer answers, both sides exchange ICE candidates, and the screen appears in the viewer's page. Fig. 3 shows the exchange.

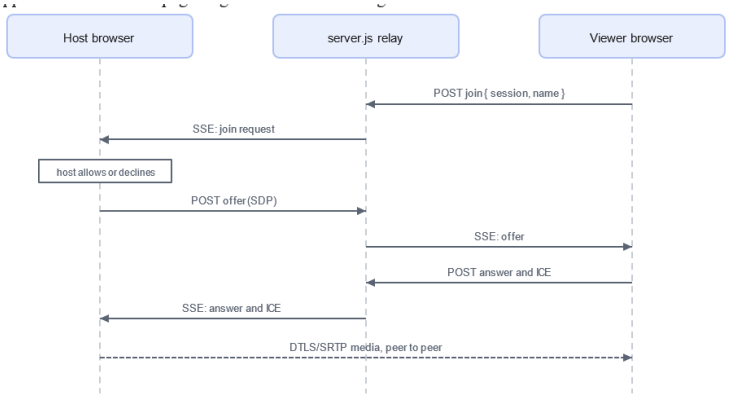


Fig. 3 *Signalling sequence for one sharing session. The relay carries text only; the media path bypasses it.*

The implementation supports up to three viewers from one host, each with its own peer connection, and drops a join request that is not answered within eighteen seconds so that the host cannot be left with a stale dialog. Candidates that arrive before the approval message are queued rather than discarded, which matters on a slow network where the browser's gathering can outrun a human decision. When the host ends the session, every track is stopped and every connection closed, and the viewer is told why the stream stopped. Public STUN servers are used for address discovery; no TURN relay is configured, a limitation taken up in Section 5.2.

Two failure modes deserve explicit handling because students will meet them. Screen capture requires a secure context, so the platform explains the situation when it is opened over plain HTTP on a local network address and points to the secured origin. And a page embedded in another page may be denied the capture permission by policy, so the interface detects the embedding and asks for a top-level tab. Both messages are produced before the browser's picker is invoked, so the student is never left guessing why nothing happened.

4.6 Session Registry and Relay

The relay keeps a registry of open sessions in memory: for each identifier it records the host, the connected viewers, the stream of pending events for each participant, and the time of the last activity. A viewer's event stream is one HTTP response that stays open and receives framed events; a host's events arrive the same way, and both sides post their descriptions and candidates as ordinary requests. Heartbeat comments are written periodically so that an intermediary does not close an idle connection, and a closed response removes the participant from the registry. Because the registry is bounded by the number of sessions rather than by history, the process runs in a few megabytes and needs no database; the consequence, discussed in Section 5.2, is that a restart ends live sessions.

The same relay is used by the voice-call page and by the remote page, which keeps one tested code path for all real-time sessions. When two tabs of the same browser are used — the ordinary way to demonstrate the feature in a laboratory — the client detects the condition and passes signalling messages directly between tabs instead of making a round trip to the server, so a demonstration works even with the network unplugged.

4.7 Shared Code Workspaces Under Sandbox

Each project owns a small workspace with a script, a stylesheet and a readme. The script runs inside a frame whose sandbox attribute grants script execution but withholds same-origin access, so the document inside has an opaque origin, cannot read the host page's storage and cannot navigate the top-level window. Results, errors and console output travel back to the host through posted messages, which is the only channel the sandbox leaves open. A run can therefore be stopped by discarding the frame, and a deliberate attempt to reach the host's session fails at the origin boundary rather than at a

filter that might be bypassed. The workspace is intentionally a scratchpad rather than a full development environment: it exists so that a student can show a classmate a twenty-line idea during a review, not so that a project can be developed inside the browser.

4.8 Security and Privacy Considerations

Four decisions bound the exposure of the system. The model key lives only in the server environment and is never serialised into the page. Media is protected by DTLS-SRTP and carried directly between peers, so the server holds neither video nor audio. Student code executes in an opaque origin behind a sandbox attribute. And user data is not stored on the server at all: the snapshot stays in the browser, which means a compromise of the host machine exposes no student record. The corresponding responsibilities are stated plainly: the platform is a campus tool without authentication of its own and must be placed behind the institution's access control; the session identifier is a capability and should be shared as one would share a meeting link; and the assistant can be wrong, so its output is presented as a suggestion inside the discussion rather than as an authority.

4.9 System Workflow

Fig. 4 shows how the layers of the implementation relate: presentation, interaction, state, media, intelligence, execution, transport and verification. Fig. 5 shows the runtime loop. On load the application restores the snapshot, parses the route and renders the matching view. A user action mutates the model, the view is re-rendered, and the snapshot is written and broadcast after a short debounce. Navigation alone does not touch the model, which is why the loop separates the two paths. The branch on the right is the remote page, where the same loop is extended: capture, publish the identifier, wait for a join request, approve, negotiate, and then stream until either side stops.

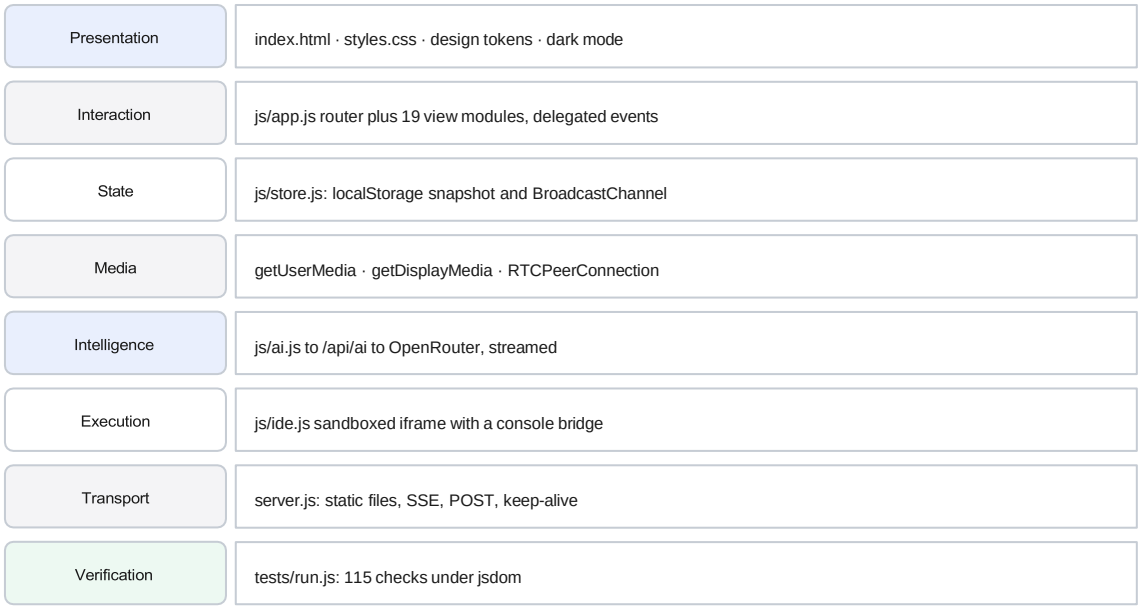


Fig. 4 *Layered view of the implementation, from presentation to verification.*

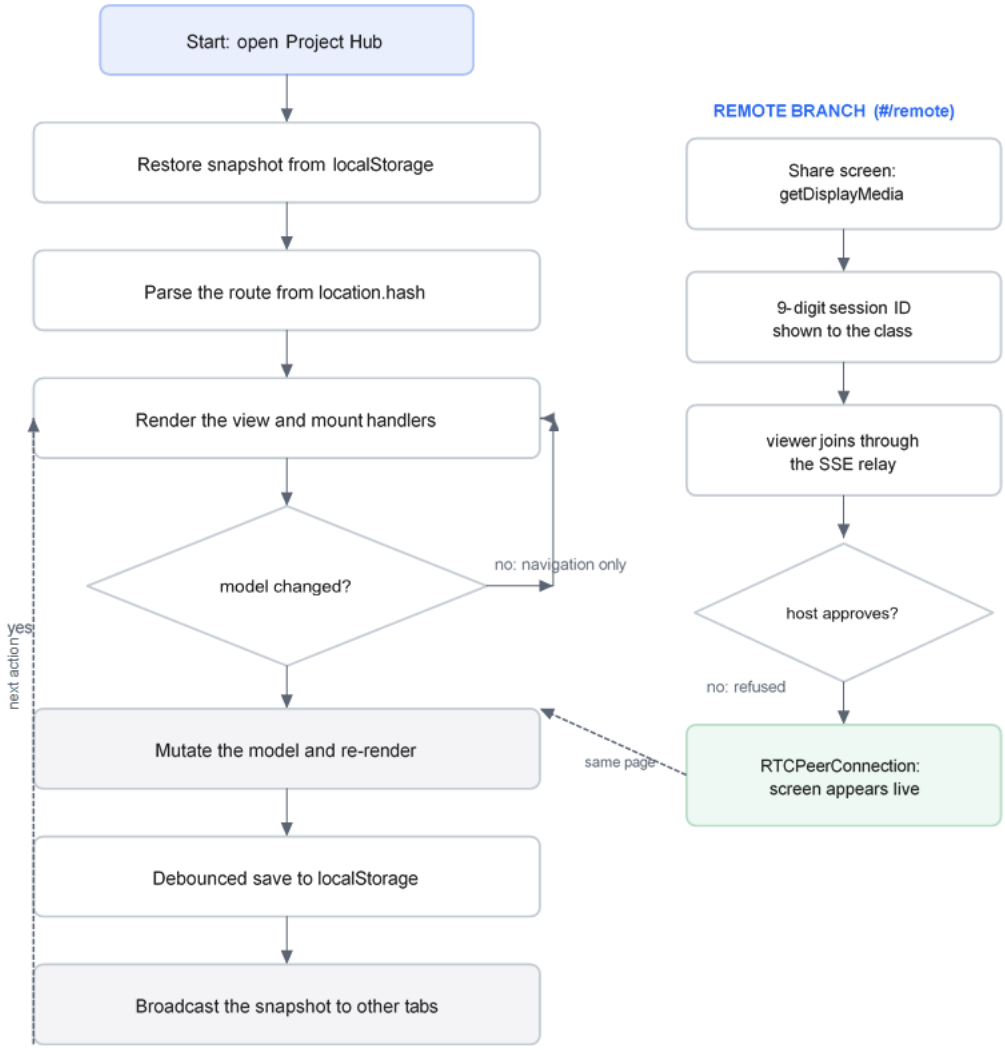


Fig. 5 *Runtime loop, with the remote-sharing branch on the right.*

4.10 Implementation and Verification

The system comprises 23 files and 10,554 lines, as Table 2 records. Roughly two thirds of the client is view code; the largest single module is the remote page at 1,154 lines, which reflects the number of user-visible states a sharing session passes through. The server is 572 lines with no third-party import.

Table 2 *Size of the implementation, measured from the delivered source.*

Component	Files	Lines	Content
Markup	1	127	Document shell, regions and templates
Stylesheet	1	2,306	Design tokens, layouts, themes, responsive rules
Client modules	19	6,759	Views, router, store, signalling, assistant, workspace
Server	1	572	Static files, model proxy, signalling relay, registry
Verification	1	790	Automated checks executed under jsdom
Total	23	10,554	

Verification is a plain Node.js harness that builds a document with jsdom, loads the client modules into it, and asserts on the resulting state and markup. It needs no browser and runs in a few seconds, which is short enough that it was run after every change during development. Table 3 lists the areas covered. The suite reports 115 passing checks.

Table 3 *Areas exercised by the automated verification suite (115 checks in total).*

Area	What the checks assert
Routing	Every route renders its view; an unknown route falls back without an error; deep links survive a reload
State	Mutations persist, a reload restores them, a second tab receives the snapshot, media URLs are stripped before writing
Feed	Posts, replies and likes update the model; empty and over-long input is refused
Projects	Tasks toggle and persist; roles accept and withdraw interest; comments are appended with an author and a time
Messaging	Conversations open, send and reorder; the unread count clears on read
Assistant	The panel reports a missing key clearly; a streamed reply is appended incrementally; the key never appears in markup
Remote	The sharing state machine reaches each state in order; a refused join does not create a connection; capture preconditions produce an explanation
Workspace	Code runs in a sandboxed frame; output and errors return to the host
Events and search	Filters, notifications and query matching behave on seeded data

5. RESULTS AND DISCUSSION

The implementation was completed and exercised on a desktop machine running Node.js 20, with Chrome and Firefox as the reference browsers. This section reports what the system does, what was measured, and what the measurement does not establish.

5.1 Result

Project Hub delivers the collaboration surface described in Section 1 from a single process and a single URL. A student who opens the platform finds a board on which projects are listed with their track, lead, member count and open roles, a project page on which tasks can be toggled, interest in a role expressed and a comment added, a campus feed that accepts text, images and short video with replies and likes, direct conversations that behave as two-party threads with a call button, topic channels in which the assistant answers with the channel's context, a code workspace that runs a script in an isolated frame, an events page and a search across projects and people. Every one of those interactions writes to the same state, so a task completed on the project page is visible in the feed and in the next tab that is opened.

Remote support works end to end. On the host side, the browser's picker is invoked, a nine-digit identifier appears, and each joining classmate is presented as a named request that must be approved. On the viewer side the shared screen appears within a second or two of approval on a campus network, and the viewer can leave at any time without disturbing the other viewers. The host can stop sharing from the interface or from the browser's own sharing indicator; either way every peer connection is closed and each viewer is told that the session ended. Two tabs of one browser can hold a session with no server round trip, which makes the feature demonstrable in a laboratory with the network disabled.

The size and verification figures of Section 4.10 are the quantitative result: 10,554 lines across 23 files, of which the server is 572 lines and imports only the standard library, and 115 automated checks that run in seconds without a browser. Two properties follow, and both were design goals rather than accidents. The deployment has no dependency tree, so there is nothing to audit for vulnerabilities and nothing to update between semesters. And the client is inspectable: a student can read the module that implements the feature being demonstrated, which is a pedagogical benefit that a bundled framework would have removed.

Resource use is proportionate to the design. The server holds a registry of open sessions and nothing else, so its resident memory is a few tens of megabytes and is independent of the number of students, because student data lives in their browsers. Bandwidth on the server is limited to signalling text and the assistant stream; the screen video flows between the peers. A hosting decision therefore reduces to one small virtual machine, which is the practical answer to the operational objection in Section 3.

5.2 Discussion

Three findings are worth generalising beyond this system. The first concerns where the intelligence sits. Keeping the model key on the server and streaming the reply through it costs about forty lines of code and converts an unusable demonstration into a service, because the browser never holds a credential that a student could copy. The same reasoning applies to any metered third-party interface in an educational setting: the proxy is not an optimisation, it is the security boundary. The second concerns where the media sits. Peer-to-peer screen sharing removes the operator's bandwidth bill and the institutional data-protection question in one decision, since a frame of a student's screen is never written to a disk the department controls. The

third concerns where the state sits. A local-first snapshot made the platform demonstrable offline and kept the backend at one process, at the cost described below.

The limitations are as important as the results, and each follows from one of those decisions. No TURN relay is configured, so two peers that are both behind symmetric NAT will fail to connect even though signalling succeeds; on a campus network this is rare, but a student on a mobile hotspot can meet it, and the honest description is that screen sharing works on the networks we tested and not universally. The relay registry is in memory, so restarting the server ends live sessions; a persistent room store would fix this at the cost of a database. The local-first snapshot is bounded by the browser's per-origin quota, so a long-running team will eventually need an export and an import rather than unbounded history, and two devices do not see each other's changes because the broadcast channel is confined to one browser profile. Screen capture requires a secure context, so the platform must be served over HTTPS or on localhost; plain HTTP on a local network address cannot capture, and no amount of interface design changes that. Finally, the platform has no authentication of its own and relies on the institution's access control in front of it, which is acceptable for a campus tool and insufficient for a public one.

A comparison with the alternatives students actually use makes the position clear, and Table 4 records it. Institutional platforms win on administration and identity and lose on building. General messaging wins on familiarity and loses on structure and artefacts. Collaborative editors win on simultaneous text editing, which this system does not attempt. Commercial remote-support tools win on reach through firewalls, because they run relay infrastructure, and lose on the installation and the licence. The contribution here is not superiority on any single axis but the combination: a project board, a feed, messaging, an assistant and a working screen share in one tab, at a hosting cost of one small machine, with no install on either side.

Table 4 *Position against the tools a student team would otherwise combine.*

Capability	Project Hub	Institutional LMS	Messaging app	Collaborative editor	Commercial remote support
Project board with tasks and roles	Yes	Partial	No	No	No
Shared code workspace	Yes	No	No	Partial	No
AI assistance with project context	Yes	Rare	Add-on	Add-on	No
Screen sharing, no install for viewer	Yes	Add-on	No	No	Yes
Media kept off the operator's server	Yes	No	No	No	No
Runs behind institutional access control	Yes	Yes	No	No	No
Hosting cost	One small VM	Licensed	Free tier	Licensed	Per seat

The threats to validity should be stated. The evaluation is functional rather than empirical: 115 automated checks and a set of manual sessions establish that the system behaves as specified, but no controlled study has been run with student teams, so the claim that the platform reduces coordination cost is an argument from design and from the reported experience of development, not a measured effect. The checks run under jsdom, which models the document but not the media stack, so the WebRTC paths were exercised manually rather than automatically; a browser-driven harness is the obvious next step and is listed in Section 6.2. Network conditions were those of one campus, and the no-TURN limitation is the most likely source of failure elsewhere.

6. CONCLUSION AND FUTURE WORKS

6.1 Conclusion

This paper described Project Hub, a browser-based platform that gathers the coordination work of a student project team into one application. The system combines a project board with tasks, roles and comments, a campus feed with media, direct messaging and calls, topic channels with an AI assistant, a sandboxed code workspace and peer-to-peer screen sharing. Its architecture is deliberately austere: a 572-line Node.js server with no runtime dependencies serves the application, proxies assistant requests so the provider key stays server-side, and relays WebRTC signalling, while the screen video itself travels directly between browsers. The client is a nineteen-module single-page application that persists a versioned snapshot in local storage and keeps tabs consistent through a broadcast channel, and it is verified by 115 automated checks that run without a browser.

The result that matters most is not a feature list but a proportion. The features students actually need for a review — seeing each other's screen, reorganising a task list together, asking a question with the project in view — were delivered by assembling browser primitives that already exist, with a backend small enough for one person to read in an afternoon. The limits are equally clear: without a relay for stubborn networks, screen sharing is not universal; without a database, sessions end with the process; without a backend store, history is bounded by a browser quota. Those are the boundaries of the present system and the agenda for the next one.

6.2 Future Works

Five extensions follow directly from the limitations in Section 5.2. A TURN relay, either self-hosted or obtained as a service, would make screen sharing work across symmetric NAT and mobile networks at a bandwidth cost the operator can bound with per-session limits. A persistent room store would let sessions survive a restart and would allow a guide to join a review that is already in progress. Server-side storage of the project state, with the local snapshot retained as a cache, would lift the quota ceiling and let two devices see the same board, while keeping the interface responsive when the network is slow. Authentication through the institution's identity provider would replace the assumption that the platform sits behind an access gateway, and would allow per-project membership and an audit trail of who shared a screen with whom. And a browser-driven test harness, running the suite against a real engine with a synthetic media device, would turn the manual WebRTC verification into an automated one.

Beyond those, three directions are natural. Simultaneous editing of project documents, using a conflict-free replicated data type over the same broadcast channel, would close the one capability in which collaborative editors are clearly ahead. Multilingual and voice-driven interaction with the assistant would widen access for students who are more comfortable speaking than typing. And a lightweight analytics view, computed locally from the snapshot, could show a team where its time actually went without sending any of that data off the device. Each of these preserves the property that makes the present system deployable by a department: the sensitive parts — media, credentials and student work — stay where they belong.

ACKNOWLEDGMENTS

The authors thank the management of Sri Shakthi Institute of Engineering and Technology for the facilities provided for this work, and the Head of the Department of Computer Science and Engineering, Dr. K. E. Kannammal, for her support and encouragement. They record their gratitude to their supervisor, Mrs. Gayathri V, Assistant Professor, whose insistence that every feature be demonstrated rather than described shaped both the system and this paper. They also thank the students who tested early builds and reported what broke, and the maintainers of the open standards and open-source tools on which the implementation depends.

REFERENCES

1. Grudin, J. (1994). Computer-Supported Cooperative Work: History and Focus. *Computer*, 27(5), 19–26.
2. Johnston, A. B., & Burnett, D. C. (2014). *WebRTC: APIs and RTCWEB Protocols of the HTML5 Web*. Digital Codex LLC.
3. Alvestrand, H. (2021). RFC 8825: Overview: Real-Time Protocols for Browser-Based Applications. Internet Engineering Task Force.
4. Keränen, A., Holmberg, C., & Rosenberg, J. (2018). RFC 8445: Interactive Connectivity Establishment (ICE). Internet Engineering Task Force.
5. World Wide Web Consortium (2021). *WebRTC 1.0: Real-Time Communication Between Browsers*. W3C Recommendation.
6. World Wide Web Consortium (2023). *Screen Capture*. W3C Working Draft.
7. WHATWG (2024). *HTML Living Standard*: server-sent events, the sandbox attribute and the BroadcastChannel API. https://html.spec.whatwg.org
8. Kleppmann, M., Wiggins, A., van Hardenberg, P., & McGranaghan, M. (2019). Local-First Software: You Own Your Data, in Spite of the Cloud. *Proceedings of Onward!*, 154–178.
9. Grigorik, I. (2013). *High-Performance Browser Networking*. O'Reilly Media.
10. Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.
11. Kasneci, E., Sessler, K., Küchemann, S., et al. (2023). ChatGPT for Good? On Opportunities and Challenges of Large Language Models for Education. *Learning and Individual Differences*, 103, 102274.
12. OpenRouter (2025). *API Reference: chat completions and streaming*. https://openrouter.ai/docs
13. OpenJS Foundation (2025). *Node.js Documentation*: the http, fs and stream modules. https://nodejs.org/docs

AUTHOR INFORMATION

Jayasimha S, **Jeeva A**, **Jeevanantham K** and **Jayasree M** are final-year students of the Department of Computer Science and Engineering at Sri Shakthi Institute of Engineering and Technology, Coimbatore. As the project team they were responsible for the design, implementation and verification of Project Hub: the client modules and state store, the signalling relay, the remote screen-sharing flow, the assistant integration, the code workspace and the automated test suite. Their individual registers are 714025104101, 714025104103, 714025104104 and 714025104102 respectively.

Gayathri V is Assistant Professor in the Department of Computer Science and Engineering at Sri Shakthi Institute of Engineering and Technology, Coimbatore. She supervised this work, reviewed the architecture and the WebRTC signalling design, and guided the verification strategy. Her research interests include web systems, human-computer interaction and the use of artificial intelligence in engineering education.