IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511160

CodeVerse: A Browser-Based Collaborative IDE with Server-Arbitrated File Locking, Hybrid Remote/Local Code Execution, and Schema-Constrained AI Assistance

Mr. J. R. Harshavardhan[1], Shreyas B[2], Yashas A[3], Yashwanth K[4], and Shiva Prakash J[5]

Associate Prof., Dept of CSE, K.S.School of Engineering & Management, Bengaluru, India[1]

Student, Dept of CSE, K.S.School of Engineering & Management, Bengaluru, India[2–5]

Abstract: Setting up compilers, runtimes, and shared repositories remains a recurring obstacle for students and small teams who simply want to write code together. This paper presents CodeVerse, a web-based integrated development environment that runs entirely from a browser tab and combines three capabilities that are usu-ally delivered by separate products: real-time multi-user editing, multi-language program execution, and an AI assistant that edits project files. Rather than adopting operational transformation or conflict-free replicated data types, CodeVerse serializes writers with a server-side, per-file lock that expires after three seconds of inactiv-ity, which keeps the server authoritative and the client logic simple. Code is dispatched first to a remote Judge0 execution service and, on failure, to local toolchains for nine languages, with all workspace files staged together so that multi-file programs build correctly. The AI module forces the language model to answer in a fixed JSON schema of create/update file operations and validates every path before the workspace is touched. Room state, membership, chat, and an edit-level audit trail are persisted in PostgreSQL through Supabase, and member credentials are stored only as salted scrypt or SHA-256 digests. We describe the architecture and protocols, report measurements of the lock service on commodity hardware (about one microsecond per acquire/release pair and no mutual-exclusion violations in 160,000 contended attempts), and discuss the security and scalability limits of the current design together with directions for future work.

Keywords: collaborative editing, web IDE, file locking, code execu-tion sandbox, WebSocket, large language models, software engineering education

I. INTRODUCTION

Programming is increasingly a social activity. Pair programming, remote code review, classroom exercises, and hackathons all re-quire several people to look at and change the same source files, ideally without first agreeing on an operating system, a compiler version, or a version-control workflow. Browser-hosted devel-opment environments lower this barrier: the only prerequisite is a URL. Early research prototypes such as Collabode showed that a shared, web-based editor can support tight collaboration on real programs [1], and the idea has since matured into several commercial services.

Building such a system nevertheless involves three technically distinct problems. The first is consistency: when two people type at the same moment, the system must decide what the shared document becomes. The second is execution: arbitrary, poten-tially hostile programs in many languages must be compiled and run on behalf of anonymous users with bounded time and

- A collaboration protocol in which an authoritative server grants per-file write locks that renew on every edit and ex-pire after three seconds of silence, together with its failure handling (disconnects, room termination, read-only mem-bers).
- A two-tier execution layer that prefers a remote judge service and degrades to local compilers, and that stages the whole workspace so that imports, headers, and multi-file builds resolve.
- An AI gateway that constrains model output to a typed list of file operations and rejects unsafe paths before they reach the editor.
- An audit trail that records, for every edit, the changed line range, line counts, and bounded before/after snippets, com-puted by a linear-time prefix/suffix comparison.
- A measured evaluation of the lock service and credential hashing, and a candid discussion of the limits of the design.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511160

The remainder of the paper is organized as follows. Section II reviews related work. Section III presents the architecture. Sec-tions IV, V, and VI detail the collaboration, execution, and AI subsystems. Section VII reports measurements and a security analysis, Section VIII discusses limitations, and Section IX con-cludes.

II. RELATED WORK

A. Consistency in Shared Editors

Concurrency control for groupware has been studied for more than three decades. Ellis and Gibbs introduced operational trans-formation (OT), in which concurrent operations are rewritten against one another so that all replicas converge [2]; Sun and Ellis later analyzed the algorithmic pitfalls and the achievements of OT in real-time group editors [3]. Conflict-free replicated data types (CRDTs) reach the same goal by making concurrent updates commutative by construction [4], and they underpin much of the local-first software movement [6]. Both families give ex-cellent responsiveness and offline tolerance but demand intricate data structures and, for text, careful handling of tombstones or transformation functions [5].

A more conservative alternative is to prevent concurrency at the granularity of a document region. Greenberg and Marwood examined how locking and other concurrency schemes shape the user interface of real-time groupware and noted that the perceptibility of conflicts matters as much as their resolution [7]. CodeVerse follows this pragmatic line: it trades fine-grained simultaneous typing in one file for a simple, inspectable server-side state machine, while still letting several people work in different files simultaneously.

B. Awareness and Web IDEs

Workspace awareness—knowing who is present, where they are working, and what they have just done—is central to ef-fective synchronous collaboration [8]. CodeVerse therefore ex-poses participant lists, per-user colors, remote cursors, typing indicators, and an activity log. Commercial cloud IDEs and Jupyter-style notebook services provide hosted runtimes and, in-creasingly, shared sessions; CodeVerse differs by being a small, self-hostable system whose entire protocol is visible in a few hundred lines of server code.

C. Online Judges and AI Pair Programmers

Hosted code-execution engines such as Judge0 accept source code, compile and run it in an isolated environment, and return output and resource usage through an HTTP API [9]. On the assistance side, large language models trained on code can syn-thesize programs from natural-language descriptions [10, 11], and controlled studies report measurable productivity gains for developers using such tools [12]. Qualitative work also warns that users struggle to evaluate and debug generated code [13], which motivates CodeVerse's decision to apply model output only as explicit, reviewable file operations.

III. SYSTEM ARCHITECTURE

CodeVerse follows a thin-client, authoritative-server design (Fig. 1). The browser holds presentation state; the server holds room state, locks, and credentials; external services provide execution, persistence, package metadata, and language-model inference.

A. Client

The client is a single-page React application. Monaco provides syntax highlighting, multi-tab editing, and a split view; xterm.js renders a terminal whose commands are forwarded to the server; Zustand stores hold the workspace tree, open tabs, terminal sessions, and settings. HTML and CSS files open a sandboxed live-preview frame, while a command palette, search panel, and resizable layout give the interface a familiar desktop-IDE feel. A dedicated collaboration view adds a participant list, chat panel, file explorer with lock indicators, shared preview, and an AI drawer.

B. Server

A single Express process serves the compiled client, a small REST surface (health, runtime status, package search/install/uninstall, run, terminal exec, AI status and as-sistant), and the Socket.IO endpoint used for collaboration. All operational parameters—ports, origins, runtime commands, lim-its, registry URLs, and model name—are read from environment variables at start-up, and the process refuses to start when a re-quired value is missing. This makes the same artifact deployable on a laptop or in a container.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511160

C. Persistence and Deployment

Durable state resides in five PostgreSQL tables: roons, roon_nenbers, roon_files, roon_nessages, and roon_activity. Each table enables row-level security with a deny-all policy, so only the server's service-role key can read or write them. Writes are coalesced: every mutation re-arms a 250 ms debounce timer, and a promise queue guarantees that persistence passes never overlap. On boot the server reloads all rooms, so sessions survive restarts. The front end is deployed as static assets on Vercel and the back end as a Docker image on Railway, with language runtimes installed in the image.

IV. COLLABORATION SUBSYSTEM

A. Rooms, Identity, and Permissions

A user shares the current workspace by creating a room protected by a password. The server generates a short room identifier and stores a salted scrypt digest of the password [14]. On the first successful join, the server issues a random member token to the browser and stores only its SHA-256 digest; subsequent visits present the token and are recognized without re-entering the password. The host additionally holds a secret host token, compared in constant time. Members carry an access level, either edit or read, that the host may change or revoke at any time. A room admits at most eight participants by default, and each room may hold up to 200 files of at most 1 MiB each.

Real-time messages travel over WebSocket [15] through Socket.IO, whose heartbeat settings (25 s ping interval, 60 s timeout) detect dead peers. A 120 s recovery window lets a participant who lost connectivity rejoin as the same member.

B. Server-Arbitrated File Locking

Instead of merging concurrent edits, CodeVerse prevents them. The lock service keeps a nested map from room to file path to a lock record containing the owner's socket identifier, user name, acquisition time, and an inactivity timer. The protocol has four rules.

1. **Acquire.** A writer obtains a file's lock either explicitly (a lock-request event) or implicitly by sending the first edit. If the file is free, the lock is granted; if the requester already owns it, the timer is simply re-armed; otherwise the request is denied and the current owner is returned.
2. **Renew.** Every accepted edit by the owner restarts a 3 s timer.
3. **Release.** The lock is freed explicitly (for example when the user switches files), when the timer fires, when the owner's socket disconnects, or when the room is terminated.
4. **Announce.** After each change of the lock table, the server broadcasts the complete table to the room, so clients converge on an authoritative snapshot instead of replaying deltas.

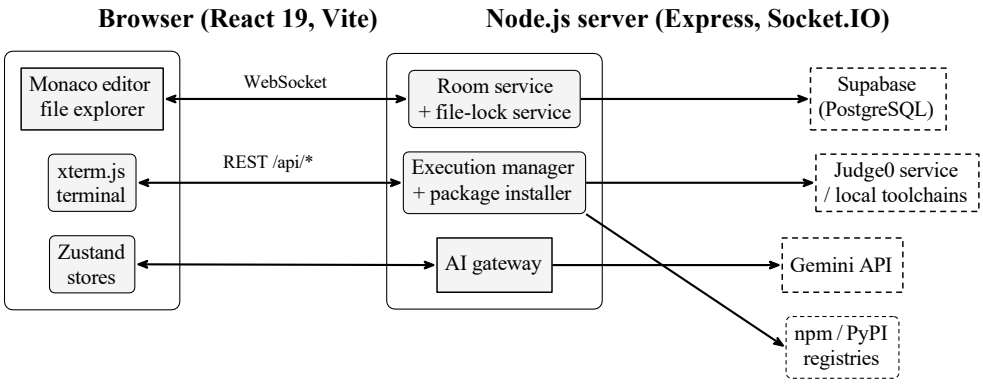

Fig. 1 High-level architecture of CodeVerse. Solid arrows are runtime calls; dashed boxes are external services.

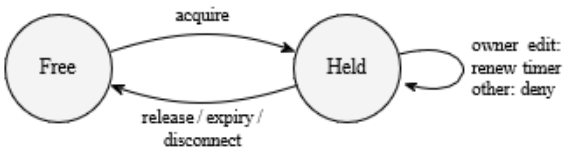

Fig. 2 Lock state machine for one file. The server alone drives transitions.

Fig. 2 shows the resulting state machine for a single file.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511160

Correctness rests on a property of the runtime rather than on explicit mutexes: Node.js processes socket events one at a time, so the "check owner, then assign owner" step inside acquireFileLock is atomic with respect to all other events. Two simultaneous requests are therefore ordered by arrival, and ex-actly one wins. Section VII-B tests this claim empirically.

When an edit arrives, the server validates that the sender is in the room and has edit access, acquires or renews the lock, checks that the new content does not exceed the file-size limit, updates the file and its version counter, appends an activity record, schedules persistence, and broadcasts the new content to the room. Denied writers receive the owner's identity so the interface can display, for instance, "file in use by a collaborator." Because the unit of exchange is the whole file under an exclusive lock, there are no divergence or transformation concerns, at the cost of sending more bytes per keystroke than an operation-based scheme.

C. Edit-Level Audit Trail

Every accepted edit produces a record that answers "who changed which lines, and how." The server compares the old and new text line by line from both ends: it advances a prefix index while lines are equal, retreats a suffix index while trailing lines are equal, and treats the interval in between as the change. From this it derives the first and last affected line, the number of lines added and removed, and a before/after snippet truncated to 500 characters. The procedure is linear in file length and independent of any diff library. File creation, deletion, and renaming are logged similarly, the log is capped at 500 items per room, and each file carries a monotonically increasing version.

V. EXECUTION SUBSYSTEM

A. Language Registry and Dispatch

Supported languages—Python, JavaScript, TypeScript, C, C++, C#, Java, Go, and Rust—are described in a registry that maps a canonical identifier and its aliases (for instance py, c++, golang)to a local executor. The dispatcher first normalizes the requested language and rejects unknown ones with a list of supported choices. If a Judge0 endpoint is configured, which it is by default, the request is submitted there: sources and standard input are Base64-encoded, language identifiers are resolved from the service's language list (cached for ten minutes, with built-in defaults as a fallback), and the result is polled at 500 ms intervals for up to 15 s [9]. Any failure of the remote path is logged and the request falls through to the language's local executor, which writes files into a temporary directory, compiles if necessary, runs the program under a 10 s wall-clock limit, kills it on timeout, and always removes the directory afterwards.

C. Workspace Staging

Real programs span several files. Before running, the server writes every workspace file into the temporary directory and then writes the active file under the requested name. Paths are normalized, empty or .. -containing paths are rejected, and the resolved absolute path must remain inside the temporary root. Consequently Python imports, JavaScript modules, and C/C++ headers resolve as they do on a developer's machine, and the same staged directory is used in a collaborative room so that every participant sees identical output, which is broadcast to the room.

D. Package Management and Terminal

Package search queries the npm and PyPI registries and never installs anything implicitly. Installation is a separate, explicit action: Python packages are installed into a dedicated target directory, Node packages into a dedicated prefix, and both di-rectories are prepended to PYTHDNPATH and NDDE_PATH for subse-quent runs, so installed libraries persist across executions without touching the system interpreter. The terminal emulator is inten-tionally not a general shell: a small command set (python, node, gcc, j ava, pip install, npm install, ls, cat, and a few others) is parsed server-side and mapped to the same execu- tors, which limits the attack surface compared with exposing a pseudo-terminal.

VI. AI ASSISTANT

The assistant is a server-side gateway to a Gemini model [11]. The client sends the user's request, an action hint (chat, explain, fix, refactor, or generate), the active file, the list of workspace paths, optional attachments, and the latest twelve conversation turns. The server composes a structured prompt, rejects attach-ments larger than one mebibyte, and calls the model with a system instruction and a response schema: the model must return a JSON object with a short reply and an array of operations, each consisting of a type (create or update), a relative path, and the complete final file content.

Constraining output to a schema has two practical benefits.First, the client never parses Markdown or guesses where code begins and ends; it simply applies operations. Second, every operation can be validated before use: paths are normalized, leading slashes are stripped, and any path containing a parent-directory segment, an unsupported type, or an empty path aborts the response. The assistant is instructed never to claim that code was run unless execution output was supplied.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511160

API keys remain on the server and the model name is configuration, not code. Within a collaboration room, a separate assistant panel operates on the shared files, so the lock rules above still decide who may write the result.

VII. EVALUATION

A. Method and Scope

We evaluate the components whose behavior can be measured without external services or human participants. All experiments execute the unmodified lock module and Node.js's built-in cryp-tography on a 2-vCPU Intel Xeon (2.1 GHz) virtual machine running Node.js 22.22. Each micro-benchmark was run once in a fresh process. End-to-end network latency, Judge0 turnaround, model latency, and user studies depend on deployment and par-ticipants and are not reported here.

B. Lock Service and Credential Hashing

Table 1 summarizes the results. Uncontended acquire-then-release pairs over 200 distinct file paths cost about one microsec-ond each, so the lock layer is negligible next to network and rendering costs. To test mutual exclusion, eight simulated sock-ets requested the same file in 20,000 independent trials (160,000 requests); in every trial exactly one request was granted, giving zero violations. A lock acquired with no further activity expired after 3001 ms against a configured 3000 ms, confirming the timer behavior. Serializing the lock table for a room holding 200 locks, the largest case permitted by the file limit, takes about 9.5 µs, which bounds the cost of each broadcast snapshot.

Password verification with scrypt (library defaults, 32-byte output) has a median cost of 36.6 ms (min 32.9, max 49.1, n = 20), whereas token digests with SHA-256 take about 1.5 µs. The asymmetry is deliberate: passwords are low-entropy and must resist offline guessing, whereas random member tokens are already high-entropy. However, because the server uses the synchronous scrypt call, each password check blocks the event loop for tens of milliseconds, which becomes visible under bursts of joins (Section VIII).

Table 1: Measured behavior of the lock service and hashing (single run, 2-vCPU VM, Node.js 22.2.2)

Measurement	Result
Acquire + release, uncontended (200,000 pairs)	0.98 µs/pair
Contended trials (8 clients, 20,000 trials)	0 violations
Lock-table snapshot, 200 locks	9.45 µs/call
Idle lock expiry (configured 3000 ms)	3001 ms
scrypt password check (median of 20)	36.6 ms
SHA-256 token digest (20,000 calls)	1.45 µs

Table 2: Consistency Strategies for Shared Code Editing

Criteria	Per-file lock (CodeVerse)	OT	CRDT
Simultaneous typing in one file	No	Yes	Yes
Simultaneous work in different files	Yes	Yes	Yes
Server authority	Complete	Central transformer	Optional
Offline editing	No	Limited	Yes
Implementation effort	Low	High	Medium–high
Per-edit payload	Whole file	Operation	Operation
Conflict visible to user	Yes (lock owner shown)	No (merged)	No (merged)

C. Comparison of Consistency Strategies

Table 2 positions the lock-based approach against OT and CRDTs on the dimensions that matter for a small hosted IDE. The comparison reflects the properties described in the liter-ature [2–4] and the design of CodeVerse, not a head-to-head benchmark.

D. Security Analysis

Credentials. Passwords and tokens are never stored in clear text: scrypt with a per-room random salt protects passwords, SHA-256 protects high-entropy tokens, and comparisons use constant-time primitives. Database tables are locked down by row-level security so that a leaked public key reveals nothing.



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511160

Input handling. File paths from clients and from the language model are normalized and checked for traversal both when stag-ing execution workspaces and when applying AI operations. Package names are validated before being passed as arguments, and processes are launched without a shell by default, which avoids command-string injection.

Execution. Using an external judge for the primary path moves untrusted code off the application host. The local fallback, however, runs compilers and user programs with the privileges of the server process, bounded only by a timeout and a non-root container user. It should therefore be enabled only inside a disposable container; this is the most significant residual risk (Section VIII).

VIII. LIMITATIONS AND FUTURE WORK

We deliberately list the weaknesses of the current design.

- **Coarse concurrency.** Only one person can type in a given file at a time, and the 3 s idle window can feel abrupt when a writer pauses to think. Adaptive timeouts, line-range locks, or a hybrid with a CRDT inside the lock holder's file are natural extensions.
- **Bandwidth.** Whole-file messages scale with file size (up to 1 MiB). Sending diffs, which the audit routine already computes, would reduce traffic.
- **State location.** Locks are held in process memory, so horizontal scaling requires sticky routing or an external store such as a shared cache; persistence currently iterates over all rooms on each pass, which will not scale to many rooms.
- **Ev ent-loop blocking.** Synchronous scrypt calls delay all other sockets for tens of milliseconds; the asynchronous variant or a worker pool would remove this.
- **Sandboxing.** Local execution lacks cgroup, network, and filesystem isolation. Containers with seccomp profiles or microVMs would harden it. In addition, the documentation describes local execution as the main path whereas the implementation tries the remote judge first; aligning the two avoids surprises for operators.
- **Evidence.** The evaluation does not include end-to-end latency under realistic network conditions or a user study of the locking interface. Controlled sessions with students, measuring task completion time and perceived interruption, are the most important next step.
- **AI safety.** Operations are validated for path safety but not for semantic correctness; a diff preview with per-operation approval and automatic test execution would strengthen trust [13].

IX. CONCLUSION

CodeVerse shows that a useful collaborative IDE can be assem-bled from a small number of well-understood parts: an authoritative server that serializes writers with expiring per-file locks, an execution layer that prefers a remote judge and falls back to local toolchains, and an AI gateway whose typed output can be validated mechanically. Measurements confirm that the locking layer adds negligible overhead and preserves mutual exclusion under contention, while the analysis identifies concrete engi-neering steps—diff-based messaging, asynchronous hashing, externalized lock state, and stronger sandboxing—that would carry the design from classroom scale to wider deployment.

REFERENCES

[1]. M. Goldman, G. Little, and R. C. Miller, "Real-time col-laborative coding in a web IDE," in *Proc. 24th Annu. ACM Symp. User Interface Softw. Technol. (UIST)*, 2011, pp. 155–164.
[2]. C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *ACM SIGMOD Rec.*, vol. 18, no. 2, pp. 399–407, 1989.
[3]. C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *Proc. ACM Conf. Comput. Supported Cooperative Work (CSCW)*, 1998, pp. 59–68.
[4]. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proc. Int. Symp. Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS vol. 6976, 2011, pp. 386–400.
[5]. G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for P2P collaborative editing," in *Proc. ACM Conf. Comput. Supported Cooperative Work (CSCW)*, 2006, pp. 259–268.
[6]. M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. Mc-Granaghan, "Local-first software: You own your data, in spite of the cloud," in *Proc. ACM SIGPLAN Int. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2019, pp. 154–178.
[7]. S. Greenberg and D. Marwood, "Real time groupware as a distributed system: Concurrency control and its effect on the interface," in *Proc. ACM Conf. Comput. Supported Cooperative Work (CSCW)*, 1994, pp. 207–217.

 This work is licensed under a Creative Commons Attribution 4.0 International License



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511160

[8]. C. Gutwin and S. Greenberg, "A descriptive framework of workspace awareness for real-time groupware," *Comput. Supported Cooperative Work*, vol. 11, no. 3–4, pp. 411–446, 2002.
[9]. Judge0, "Judge0 CE: Open-source online code execution system," [Online]. Available: https: //j udge0. com. Accessed: Oct. 21, 2026.
[10]. M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.
[11]. Gemini Team, Google, "Gemini: A family of highly capa-ble multimodal models," *arXiv:2312.11805*, 2023.
[12]. S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub Copilot," *arXiv:2302.06590*, 2023.
[13]. P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expecta-tion vs. experience: Evaluating the usability of code gen-eration tools powered by large language models," in *Ext. Abstracts CHI Conf. Human Factors Comput. Syst. (CHI EA)*, 2022, Art. no. 332.
[14]. C. Percival and S. Josefsson, "The scrypt password-based key derivation function," IETF, RFC 7914, Aug. 2016.
[15]. I. Fette and A. Melnikov, "The WebSocket protocol," IETF, RFC 6455, Dec. 2011.

 This work is licensed under a Creative Commons Attribution 4.0 International License