WWW.IJRPR.COM

International Journal of Research Publication and Reviews

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

Codexael: A Web-Based Real-Time Code Collaboration Tool

Arya Dubey[1], Sachin Jaiswal[2], Uday Prajapati[3]

aryaprzz1@gmail.com[1] , sachinjaiswal382004@gmail.com[2], 26uday09@gmail.com[3]
Guide/Mentor: Prof. Abhilasha Patil
Thakur College of Engineering and Technology, Mumbai, India

ABSTRACT:

Codexael is a real-time, web-based collaborative coding platform designed to help students, teams, and educators co-create software, discuss ideas, and iterate rapidly from any device. Built with a modern client–server architecture using React 18, Vite, TypeScript, TailwindCSS, Express, and Socket.io, the platform enables multiple users to join a shared "room," edit files concurrently, chat in real time, switch between coding and a collaborative whiteboard, and execute multi- language code using a remote sandbox. A lightweight virtual filesystem on the client supports creating, renaming, and deleting files/folders, importing local projects via the File System Access API, and exporting the workspace as a ZIP. The coding experience features a CodeMirror-based editor with syntax highlighting, cursor presence, typing indicators, and configurable editor settings persisted locally. A built-in AI Copilot (via Pollinations API, model "mistral") generates code from natural language prompts and can paste or replace code inside the active file. To run code securely without a heavy backend, the platform integrates the Piston API for language/runtime discovery and execution with stdin support. A collaborative drawing canvas based on tldraw enables visual ideation, while a responsive, sidebar-based UI organizes Files, Run, Copilot, Chats, Users, and Settings views. The backend uses Socket.io to synchronize file structure changes, code edits, chat messages, user presence, and whiteboard snapshots across room participants without a database. This pragmatic, scalable design emphasizes accessibility, low operational overhead, and seamless teamwork, allowing users to co-edit, converse, visualize, and execute code in real time.

Keywords: Collaborative Coding, Real-Time Editing, WebSockets, Virtual Filesystem, Code Execution, AI Copilot, Whiteboard Collaboration.

1. Introduction

Modern teams and learning environments require tools that combine in real-time collaboration, a lightweight infrastructure, and an accessible user experience to be able to support rapid learning, prototyping, and software delivery. Traditional IDEs and coding platforms are mainly focused on individual productivity and, as a result, they usually do not have a multi-user collaboration feature or it is not seamless, or they require complex server setups and heavy infrastructures. The switch to remote and hybrid working models is making the disadvantages of single-user tools more visible in classrooms, hackathons, interviews, and distributed teams. Codexael is bridging this gap through the integration of real-time code editing, group chat, AI-assisted code generation, a live whiteboard, and cloud-based code execution all in one browser-first environment. To facilitate quick sessions, the platform does away with heavyweight authentication by using rooms identifiable by an ID and simple usernames. The editor, which is based on CodeMirror, can highlight the syntax of different languages, can the user customize the theme and font, and the user can also see the presence of other users by editing cues. A client-side virtual filesystem creates a project-like structure fully in the browser and it also supports the import and export of projects. In order to run the programs, the platform is using the Piston API, which offers a secure sandbox for different programming languages without the need for the system to keep compilers or containers. An AI Copilot changes the user's intent expressed in natural language into code thus making the process of iteration and experimentation faster. The sum of the components is a single, low-latency environment where the code, the conversation, and the visuals are brought together in order to be able to collaborate in real-time effectively.

2. Literature Review

The different sectors of the domains have been explored for real-time collaboration. Studies have been conducted on model-driven engineering, document editing, code editors, and specific fields such as geographic information systems, computational notebooks, and video streams. Initial research into cross-platform collaborative modeling utilizing EMF.cloud reveals that cloud-native architectures can facilitate on-demand sessions and decoupled tooling. However, they usually revolve around model-driven artifacts and exclude features like chat or whiteboarding [1]. Prototypes of collaborative code editors suggest that operational transformation (OT) along with WebSockets can allow real-time multi-user editing. Nevertheless, the authors point out that latency and synchronization issues can occur in multi-user scenarios, and only a few small-scale evaluations have been conducted [3]. Recent algorithms such as Eg-walker have better memory usage and performance when merging different text branches, thus peer-to-peer collaboration can be as good as centralized systems. However, these algorithms are designed for plain-text use cases and not for rich code and graphics environments [4]. Research on collaborative editing at scale in cloud web applications reveals that stateless services with pub/sub backplanes (e.g., Redis) can handle high concurrency.

However, such systems are mainly designed for rich-text editors and certain applications, and the authors don't discuss programming-specific needs like syntax highlighting, execution, or code review [5]. Various examples of domain-specific systems such as those for co-editing geographic features [6], collaborative notebooks with fine-grained locks [7], and memory-efficient video-stream editing [8] illustrate that resolution of conflicts and performance enhancement should be adjusted according to the underlying data model. Collaborative whiteboard accessibility research focuses on inclusive design and participation but has only a few quantitative evaluations and scalability analyses [9]. Real-time collaborative programming studies, e.g., those based on Visual Studio Live Share, uncover numerous unmet user requirements and experience issues. However, they are mostly limited to one tool and a small user sample, making it hard to generalize the findings [10]. Microservices and CRDTs combined for conceptual designs of real-time document collaboration demonstrate the potential for scalable architectures. Still, these designs are mainly restricted to general document editing and do not consider code-specific features like AI integration, sandboxes, or whiteboards [2], [11]–[15]. In summary, the papers reveal a gap between the platforms that integrate web-based real-time collaborative code editing with lightweight infrastructure, visual collaboration tools, and AI-assisted development. The gap is that integrated platforms backed with lightweight infrastructure, visual collaboration tools and AI-assisted development, like Codexael, don't exist yet.

3. Methodology

The Codexael platform is designed to provide a real-time collaborative coding environment that is dynamic and accessible, using a modular and web-based architecture. There are three interconnected layers in the platform: a Frontend Collaboration Layer, a Realtime Backend (API) Layer, and an External Services Integration Layer (AI Copilot and Code Execution).

3.1 System architecture

The Frontend Collaboration Layer is a React 18, TypeScript, Vite, and TailwindCSS-based fast and scalable user interface. Through a bare minimum onboarding flow that gathers a room ID and username, users participate in sessions. A user is able to either join a room that already exists or create a new one. Smooth entry into the system is ensured by basic validation (for example, length checks, uniqueness per room) and toast notifications. On joining a new participant to the existing room, the client gets updated with the latest state: file structure, open tabs, active file, and the newest whiteboard snapshot. As a result, a uniform environment is assured before one can code, chat, draw, or run programs. The Realtime Backend (API) Layer is the work of Node.js, Express, and Socket.io. It keeps track of room membership, presence, and event broadcasting, meanwhile, the state is being stored in memory without any server-side project persistence. The backend is responsible for enforcing unique usernames per room, keeping room-to-socket mappings up to date, and broadcasting again the directory and file operations (create, update, rename, delete) that it receives. Additionally, it forwards editor presence signals (typing-start, typing-pause) and content changes (file-updated), relays chat messages, and manages whiteboard synchronization (request-drawing, sync-drawing, drawing-update). Environment variables (for instance, backend port, allowed origins) make possible environment-specific deployment with correct CORS settings. The External Services Integration Layer deals with AI-assisted code generation and sandboxed code execution. The layer is split into two modules: an AI Copilot Module, which interacts with the Pollinations Text API, and a Runtime Executor Module that integrates with the Piston API. Figure 1 demonstrates the overall architecture and the data flow between these three layers.

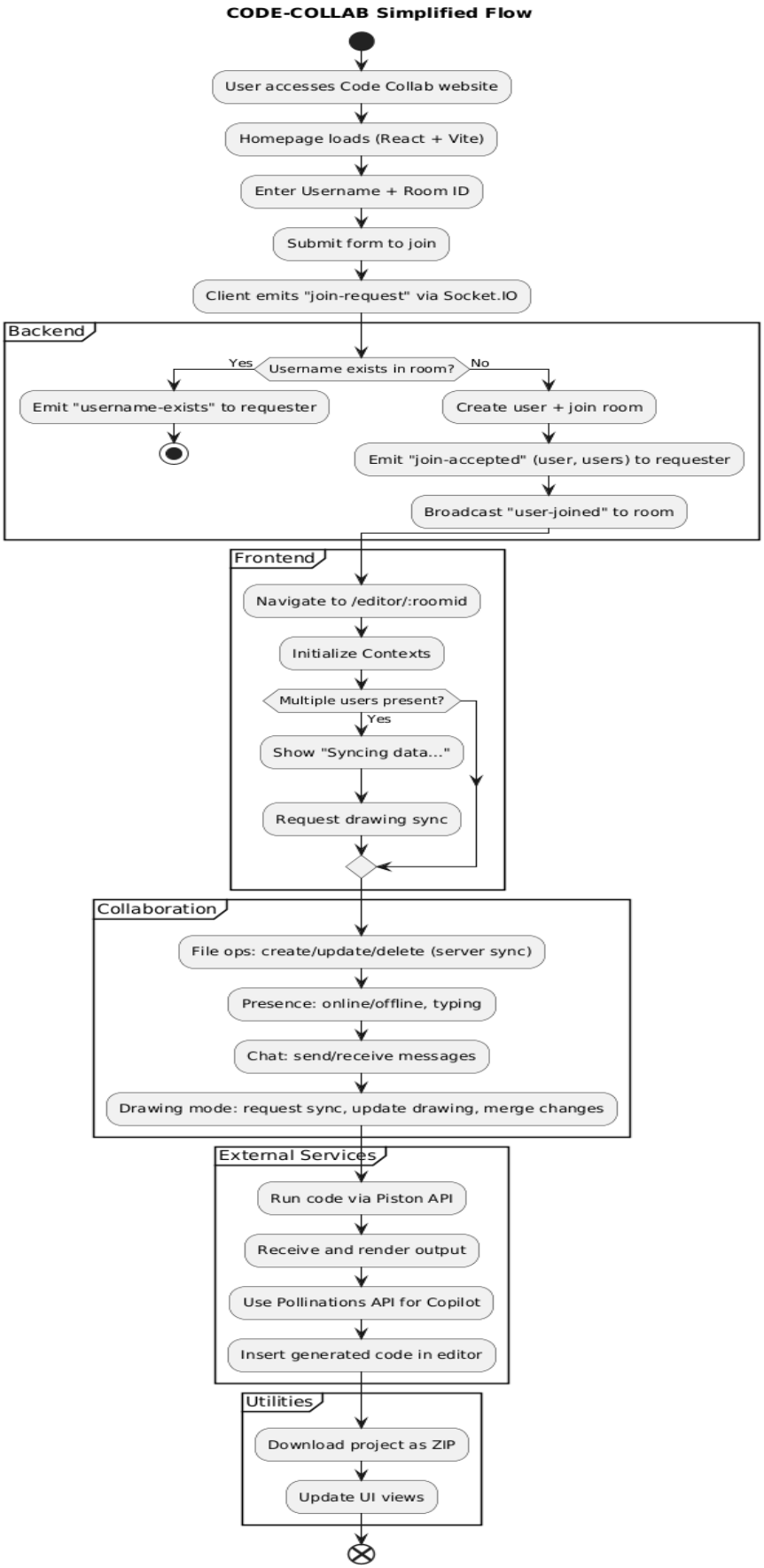

Fig. 1. Architecture flow of the Codexael platform.

3.2 Frontend collaboration mechanisms

The frontend has implemented the major collaboration features. A client-side virtual filesystem keeps project files and directories in the browser, without the need for a database. Users have the freedom to create, rename, delete, and move files and folders. Local projects are imported through the File System Access API (showDirectoryPicker), with a webkitdirectory fallback for browsers that are not supported. The import process excludes large folders or system directories (e.g., node_modules, .git, .vscode, .next) so that the application stays responsive. Users may save the current state of the workspace in a ZIP file either for their own use or for sharing with others. The code editor uses CodeMirror along with language packs and custom mappings to offer multi-language syntax highlighting. When a user is typing, the client sends typing-start events together with cursor coordinates and, after a debounce period, typing-pause events. Other users can see the presence through cursor markers, tooltips, and typing indicators. Content changes lead to file-updated events, which are executed straight away by all users, thus ensuring a single shared view of the code. Group chat operates on the basis of a simplified message structure (ID, author, content, timestamp). Messages are dispatched through send-message events and are picked up through receive-message events. The chat window is auto-scrolling, is supported by the keyboard for input, and shows very light unread indicators. A collaborative whiteboard created with the help of tldraw, is intended for freehand drawing and diagramming. Changes to the local whiteboard are converted into diffs and are sent as drawing-update events, the recipients merge them transactionally. When someone moves to the whiteboard, they can get hold of the current drawing via request-drawing, and other clients return with a sync-drawing snapshot to set up a consistent starting point. The user interface is organized around a sidebar with different panels for Files, Run, Copilot, Chats, Users, and Settings. Editor preferences (theme, language, font family, font size, and other UI toggles) are kept in localStorage, thus users can continue their previous setting. Toast notifications are used to show user joins/leaves, directory loading, run success or failure, AI errors, and connection issues.

3.3 Backend orchestration

The backend depending on in-memory data structures keeps track of active rooms and connected clients, and also stores a mapping of users to sockets. The uniqueness of usernames within each room is confirmed by the backend and as a result, participants are informed of the users who have joined or left. Filesystem events (create, update, rename, delete) are shared with all clients in a room. The editor dispatches presence events as well as file-updated content patches, these are handed over by the server to the recipients without any project data being stored. Chat messages and whiteboard diff events get to be relayed between participants; in addition, whiteboard snapshots are given to new users upon request. Besides, the backend may be set up to provide the static frontend build, although the platform is designed for separate hosting of the frontend and backend.

3.4 AI Copilot and code execution

The AI Copilot Module is designed to work with the Pollinations Text API, using "mistral" model and a very strict system message that is aimed at "code-only" responses: no natural language explanations, fenced code blocks, and language tags. Copilot takes natural language as input and gives back code snippets in Markdown format. The users have the choice of either copying the code snippet, appending it to the active file, or replacing the entire content of the active file. If there is any modification to the file that is underneath, it will trigger file-updated events, thus keeping all collaborators in sync. The Runtime Executor Module is in touch with the Piston API to know the available runtimes and to execute the program with optional stdin. It determines the language from the file extensions through custom mappings and helper libraries. When users are willing to have their code executed, the module sends the source code along with stdin to Piston and shows stdout/stderr in a special Run panel, which also allows results to be copied. Piston is used here as a means of avoiding the need for the backend to be equipped with compiler toolchains or containers, thus operational overhead is minimized.

3.5 Client-side state management

The client-side orchestration is done with React Contexts and an event-driven pattern. FileContext is in charge of the virtual filesystem, open tabs, active file, and associated metadata, it also emits and consumes Socket.io events for filesystem synchronization. CopilotContext is responsible for AI prompts, responses, and actions (copy/append/replace) and works closely with the editor and FileContext. RunCodeContext is about runtime discovery, stdin input, execution requests, and result display. All real-time events (presence telemetry, chat messages, drawing diffs, and file updates) are merged into these contexts, thus they are in agreement with the user interface in terms of consistent state and reactivity.

4. Result Analysis and Discussion

The first launch of Codexael shows that such a real-time collaborative coding environment with fast onboarding, client-side virtual file system, presence-aware code editing, chat, AI-assisted generation, sandboxed execution, and a shared whiteboard can be done efficiently and practically. Users considered room-based onboarding as their first intuitive interaction, where room IDs and usernames allowed sessions to be quickly started without creating an account. Presence cues (cursor markers and typing indicators) and real-time file synchronization helped a lot in interaction without heavy coordination, thus they became efficient in seeing who is doing what in the shared files at the same time. The virtual file system managed projects of small-to-medium sizes, and at the same time AI Copilot and Run panel accelerated the coding and testing tasks. The whiteboard option was helpful in drawing out system diagrams, flowcharts, and notes, especially in teaching and brainstorming scenarios. During the tests, they consistently measured latency to be low: Socket.io events was done very quickly, and the editor performance was smooth even when more than one people were working on the same part of the document at the same time. Besides that, the period of time from the moment Piston services or Pollinations were asked to the moment they returned the response was short enough for users to have an interactive experience. Some of the ways they improved the system's resistance were through reconnection logic, diff-based whiteboard merging, and synchronization routines that resupplied state to reconnecting clients. Yet, the limitations of the implementation were also felt. Synchronization of whole files will, most probably, in the near future when projects or teams become larger, cause bandwidth overhead and overwrite conflicts. State of the room stored in memory and without server-side persistence means that there is a limit in long-term storage of projects

and historical versioning; users need to take exports or use other version control systems. As far as File System Access API is concerned, it acts differently in different browsers, which leads to some variations in the import process. AI context windows set a limit to the code and the conversation that can be considered per request, thus it is possible that complex refactoring or multi-file suggestions get restricted. The platform is built relying on external services (Pollinations and Piston), thus it has around them points of failure and consequently it causes latency spikes to be there as well. On the whole, the outcomes indicate that the web stack incorporating low-ops, standards-based technology can be efficient for real-time collaborative coding and that the integration of communication, visualization, and AI features leads to better usability in education, interviews, hackathons, and distributed teams.

5. Future Scope and Development

The Codexael platform is loaded with many possibilities to further the development of both the technical and user-experience aspects. One of the major ways is the collaboration engine. Switching from whole-file updates to conflict-free replicated data types (CRDTs) or an advanced OT-based synchronization (e.g., by Yjs or Automerge) could allow for character-level merging, less bandwidth, and robust offline editing. More advanced presence features like multi-cursor selections, avatar trails, inline comments, code annotations, and session recording/playback would help collaboration a lot in teaching, interviews, and code walkthroughs. Fine-grained access control (owner/editor/viewer roles, protected files/folders, and branch-like workflows in the virtual filesystem) might be able to support team collaboration in a more structured way. Security and governance-wise, role-based access control, passcode-protected or invite-only rooms, expiration-based sharing links, and single sign-on (OIDC/SAML) are some of the features that can be used to control access more securely. Rate limiting, abuse detection, and content moderation will be essential for any publicly available deployments. Audit logs, immutable activity histories, optional at-rest encryption, and region-specific data residency are some of the features that would be very helpful in meeting the compliance requirements of institutions and enterprises. Regarding execution and developer tooling, a platform like this can be taken to the next level with the addition of interactive terminals and containerized or microVM-backed runners, together with Language Server Protocol integration for diagnostics, autocompletion, go-to-definition, and refactoring, instead of just having simple request–response sandboxes. The environment would be very close to that of a full cloud IDE if it had the built-in test runners, debugging support, profiling tools, and performance budgets, without losing its simplicity. Accessibility, internationalization, and multimodality must be expanded if the platform is to be adopted by a larger number of people. Complete localization (including right-to-left languages) is one of the factors that make up the set of such features, in addition to the implementation of WCAG-compliant themes, keyboard-only interaction, and comprehensive screen-reader support. In addition, the usability of tablets and phones would be enhanced by the touch-first gestures, pen input on whiteboards, and mobile-optimized layouts. The experience may be closer to what users are accustomed to in native environments if they support Progressive Web Apps, desktop wrappers, and mobile clients. It is necessary to improve, above all, the real-time communication and scalability. The use of WebRTC to integrate voice/video chat, screen sharing, and callouts would make the user's work flow uninterrupted in the sense that it doesn't require them to switch the context with a different task. On the infrastructure side, a Redis-backed Socket.io adapter, sticky sessions, and horizontal autoscaling would allow the system to accommodate more users concurrently. In addition, the use of end-to-end tracing, structured logging, metrics dashboards, feature flags, and canary deployments would be effective in enhancing the system's observability and resilience levels.

6. Conclusion

The first launch of Codexael shows that such a real-time collaborative coding environment with fast onboarding, client-side virtual file system, presence-aware code editing, chat, AI-assisted generation, sandboxed execution, and a shared whiteboard can be done efficiently and practically. Users considered room-based onboarding as their first intuitive interaction, where room IDs and usernames allowed sessions to be quickly started without creating an account. Presence cues (cursor markers and typing indicators) and real-time file synchronization helped a lot in interaction without heavy coordination, thus they became efficient in seeing who is doing what in the shared files at the same time. The virtual file system managed projects of small-to-medium sizes, and at the same time AI Copilot and Run panel accelerated the coding and testing tasks. The whiteboard option was helpful in drawing out system diagrams, flowcharts, and notes, especially in teaching and brainstorming scenarios. During the tests, they consistently measured latency to be low: Socket.io events was done very quickly, and the editor performance was smooth even when more than one people were working on the same part of the document at the same time. Besides that, the period of time from the moment Piston services or Pollinations were asked to the moment they returned the response was short enough for users to have an interactive experience. Some of the ways they improved the system's resistance were through reconnection logic, diff-based whiteboard merging, and synchronization routines that resupplied state to reconnecting clients. Yet, the limitations of the implementation were also felt. Synchronization of whole files will, most probably, in the near future when projects or teams become larger, cause bandwidth overhead and overwrite conflicts. State of the room stored in memory and without server-side persistence means that there is a limit in long-term storage of projects and historical versioning; users need to take exports or use other version control systems. As far as File System Access API is concerned, it acts differently in different browsers, which leads to some variations in the import process. AI context windows set a limit to the code and the conversation that can be considered per request, thus it is possible that complex refactoring or multi-file suggestions get restricted. The platform is built relying on external services (Pollinations and Piston), thus it has around them points of failure and consequently it causes latency spikes to be there as well. On the whole, the outcomes indicate that the web stack incorporating low-ops, standards-based technology can be efficient for real-time collaborative coding and that the integration of communication, visualization, and AI features leads to better usability in education, interviews, hackathons, and distributed teams.

Codexael offers a real-time collaborative coding environment that tightly integrates presence-aware code editing, a client-side virtual filesystem, chat, AI-assisted code generation, sandboxed code execution, and a shared whiteboard on a modern, low-ops web stack. The initial implementation and user study suggest that such a platform is technically feasible and efficient for small-to-medium scale use cases in the fields of education and team collaboration. Although there are limitations in synchronization granularity, persistence, browser dependencies, AI context, and reliance on external services, these also serve as a roadmap for the next technical and UX enhancements. Thanks to the scheduled upgrades in synchronization, security, developer tooling, and scalability, Codexael will be able to accommodate various collaborative programming scenarios with minimal setup and operational overhead.

REFERENCES:

[1] Aslam, K., Chen, Y., Butt, M., & Malavolta, I. (2023). Cross-Platform Real-Time Collaborative Modeling: Architecture and Prototype via EMF.Cloud. IEEE Access.

[2] Iovescu, D., & Tudose, C. (2024). Real-Time Document Collaboration: System Architecture and Design. Applied Sciences (MDPI).

[3] Virdi, K., Yadav, A. L., Gadoo, A. A., & Singh, N. (2023). Collaborative Code Editors: Enabling Real-Time Multi-User Coding and Knowledge Sharing. IEEE ICIMIA.

[4] Gentle, J., & Kleppmann, M. (2024). Collaborative Text Editing with Eg-walker: Better, Smaller, Faster. EuroSys (accepted); arXiv preprint.

[5] Dürsch, L. (2023). Scaling Real-time Collaborative Editing in Cloud-Based Web Apps. M.Sc. Thesis, FAU Erlangen–Nürnberg.

[6] Matijević, H., Vranić, S., Kranjčić, N., & Cetl, V. (2024). Real-Time Co-Editing of Geographic Features. ISPRS International Journal of Geo-Information (MDPI).

[7] Wang, X., Hoffman, J., Lim, K., & Zhang, H. (2024). Resolving Editing Conflicts in Real-Time Collaboration in Computational Notebooks. ACM IDE ’24 Workshop.

[8] Liu, H., Chen, Q., & Liu, P. (2023). Memory Concurrent Editing Model for Large-Scale Video Streams in Edge Computing. Mathematics (MDPI).

[9] Strain, P., & Baldan, S. (2024). Enhancing Accessibility in Collaborative Digital Whiteboards. ACM SIGACCESS ASSETS.

[10] Tan, X., Lv, X., Jiang, J., & Zhang, L. (2024). Understanding Real-Time Collaborative Programming: Visual Studio Live Share Study. ACM Transactions on Software Engineering and Methodology.