Pradeep, 2026, 14:6
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Integrating Large Language Models For Intelligent Code Assistance In Real-Time Collaborative Environments

Pradeep[1],Aditya Kumar[2], Anish Kumar[3], Sachin Kumar[4]
[1]Assistant Professor
[2,3,4] Research Scholar , Cse Department, Hmritm Ggsipu, Delhi-110036

Abstract- Real-time collaborative code editing and AI-assisted programming have developed as two separate streams of developer tooling. Existing collaborative editors support multi-user synchronization but lack integrated intelligent assistance, while commercial AI coding assistants generally operate in single-user contexts. This paper presents the design, implementation, and evaluation of an AI-enhanced collaborative code editor that combines Conflict-free Replicated Data Types (CRDTs), the Monaco Editor, Socket.IO-based WebSocket communication, and Large Language Model (LLM) based code assistance. The system uses Yjs for conflict-free document synchronization and an asynchronous AI microservice for code completion, natural-language-to-code generation, code explanation, and bug detection. The AI module supports both external APIs and locally hosted open-source models. Experimental evaluation shows that the system maintains sub-100 ms mean synchronization latency for up to 10 concurrent users, while AI suggestions are delivered within practical response times. The results indicate that decoupling AI inference from collaboration synchronization preserves real-time editing performance. The proposed framework contributes a practical architecture for combining CRDT-based collaboration with LLM-powered code intelligence.

Keywords: Collaborative Editing, CRDT, Large Language Models, Code Assistance, Real-Time Systems, Developer Productivity

I. INTRODUCTION

Software development is increasingly distributed across teams, time zones, and organizational boundaries. At the same time, artificial intelligence has become an important part of developer tooling through assistants such as GitHub Copilot and Amazon CodeWhisperer [12]. However, real-time collaborative editors and AI code assistants have largely evolved as separate categories of tools. Collaborative platforms provide shared editing but offer limited intelligent support, while AI assistants provide suggestions but generally assume a single active developer.

Real-time collaborative editing requires a synchronization model that can handle concurrent operations without conflicts. Operational Transformation (OT) was one of the earliest approaches and is used in systems such as Google Docs [1]. Conflict-free Replicated Data Types (CRDTs) provide an alternative in which concurrent operations are designed to converge without centralized transformation [2]. Yjs is a widely used CRDT implementation for shared text editing and provides efficient synchronization for collaborative applications [3].

In parallel, LLMs have demonstrated strong performance in code generation and code understanding. Codex showed that large-scale code-trained models can generate functional programs from natural-language prompts [4]. CodeLlama, StarCoder, GPT-4, and Gemini have further extended these capabilities [5–8]. Yet, integrating such models into a real-time collaborative editor introduces architectural challenges: AI inference can be slow, external APIs introduce network variability, and AI suggestions must not block or degrade the collaborative editing experience.

This paper addresses these challenges through a modular architecture that separates the collaboration

© 2026 Pradeep, This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

layer from the AI service layer. The collaboration layer uses Yjs and Socket.IO to synchronize document state and user presence [3, 10]. The AI layer runs as an independent microservice and handles prompt construction, model invocation, and response streaming. The editor is built on Monaco Editor, the same editor component used in Visual Studio Code [9].

The main contributions of this work are:

1. A three-tier architecture that integrates CRDT-based collaborative editing with an asynchronous LLM-based code assistance service.

2. A context-aware AI module that uses the current document state, cursor position, and selected code to generate suggestions without interrupting the collaborative session.

3. An empirical evaluation of synchronization latency, AI response latency, and user acceptance under concurrent editing scenarios.

4. A discussion of deployment trade-offs between external API-based and locally hosted LLM options.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the methodology and system design. Section 4 reports results and discussion. Section 5 concludes the paper and outlines future work.

II. LITERATURE REVIEW

Collaborative editing has been studied extensively since the development of groupware systems. Ellis and Gibbs introduced Operational Transformation for concurrency control in groupware [1]. OT ensures consistency by transforming concurrent operations against one another, but it often requires centralized coordination. Shapiro et al. formalized CRDTs as a decentralized alternative in which replicas converge without transformation [2]. CRDTs are particularly suitable for offline-first and peer-to-peer collaborative applications. Nicolaescu et al. developed Yjs as a CRDT-based shared editing framework and demonstrated its use for extensible data types in near real-time peer-to-peer editing [3].

On the AI side, Chen et al. introduced Codex and evaluated large language models trained on code [4]. Rozière et al. released CodeLlama as an open foundation model for code [5], and Li et al. introduced StarCoder [6]. Proprietary models such as GPT-4 and Gemini have also demonstrated strong code generation and explanation capabilities [7, 8]. Transformer architectures provide the underlying attention mechanism used by many of these models [13].

Despite progress in both areas, integration remains limited. Ahmadi et al. proposed a multi-user code assistant that aggregates developer context, but their approach did not address real-time CRDT synchronization [11]. Commercial tools such as GitHub Copilot provide inline suggestions but are not designed around shared editing sessions [12]. This paper bridges this gap by coupling a Yjs-based collaborative editor with an asynchronous LLM service.

III. METHODOLOGY

1. System Architecture

The proposed system follows a three-tier architecture: frontend, collaboration server, and AI microservice.

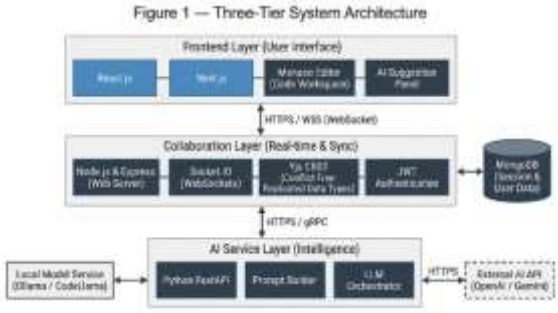


Figure 1 — Three-Tier System Architecture

The frontend is implemented using React.js with Next.js and Monaco Editor [9, 18]. The Monaco Editor is bound to a Yjs document using the y-monaco binding. The collaboration server uses Node.js, Express, Socket.IO, and Yjs [3, 10]. The AI microservice is implemented in Python using FastAPI and supports both external APIs and local model inference. MongoDB is used for user data and session snapshots [16].

Table 1 — Technology Stack and Justification

Component	Technology	Purpose / Justification
Frontend	React.js + Next.js	Component-based UI, server-side rendering, routing
Code Editor	Monaco Editor	VS Code editor engine; syntax highlighting, multi-language support
CRDT Library	Yjs	Conflict-free real-time text synchronization
Real-time Communication	Socket.IO / WebSockets	Low-latency bidirectional event transport
Backend	Node.js + Express	Lightweight event-driven collaboration server
AI Microservice	Python + FastAPI	Asynchronous LLM request handling and prompt construction
AI Models	OpenAI GPT-4, Google Gemini, CodeLlama-7B, DeepSeek-Coder	Code completion, generation, explanation, bug detection
Authentication	JWT + bcrypt	Stateless authentication and password hashing
Database	MongoDB	User data, room metadata, session snapshots
Styling	Tailwind CSS	Rapid, consistent UI development
Containerization	Docker + Docker Compose	Reproducible multi-service deployment
Version Control	Git + GitHub	Source code management and collaboration

2. Collaborative Editing Workflow

Each collaborative session is represented as a room. When a client joins a room, the server validates the JWT token [17], loads the Yjs document state, and sends the current state to the new client. Document updates are encoded as Yjs binary updates and broadcast to other clients. The CRDT property ensures that concurrent edits converge without a central transformation step [2, 3].

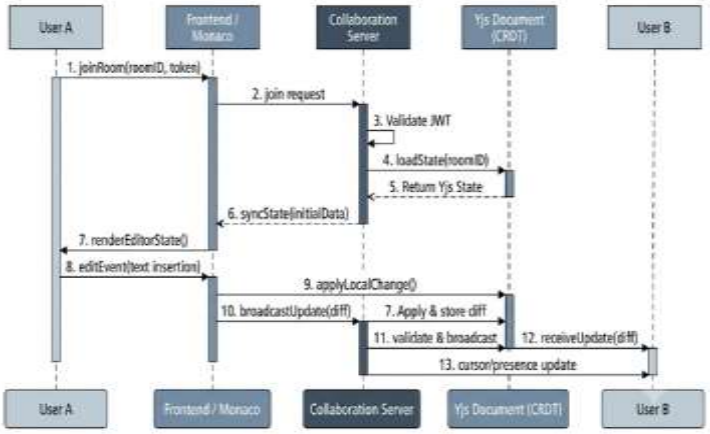


Figure 2 — Collaborative Editing Sequence Diagram

User presence is handled through Yjs awareness. Cursor positions, selections, and active user lists are shared through awareness updates and displayed in the editor UI. An in-editor chat panel is implemented over Socket.IO events.

3. Ai Assistance Module

The AI module supports four task types: code completion, natural-language-to-code generation, code explanation, and bug detection. Requests include the selected code, cursor position, language, and surrounding context. The AI service constructs a task-specific prompt and invokes either an external API or a local model.

Figure 3 — Real-Time AI Suggestion and Collaborative Workflow

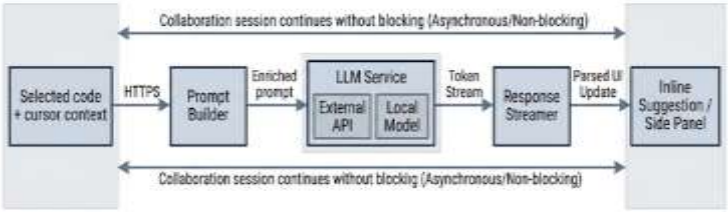


Figure 3 — AI Assistance Workflow

The AI service is asynchronous and does not block the collaboration server. This design ensures that model latency does not degrade real-time synchronization. Prompt templates are maintained separately for each task type. User inputs are sanitized before prompt construction to reduce prompt-injection risk [14].

4. Security And Deployment

Authentication uses JWT tokens with bcrypt password hashing [17]. Each room is isolated, and users can access only rooms they have joined. Rate limiting is applied to AI endpoints. The application is containerized with Docker Compose for consistent deployment [15]. The AI service can be configured to use OpenAI, Gemini, or a local Ollama-hosted model [7, 8].

5. Evaluation Metrics

Collaboration performance is measured using mean and 95th-percentile synchronization latency. AI performance is measured using mean and 95th-percentile response time for each task type. A small user study assesses perceived usefulness, integration quality, and satisfaction using a Likert-scale questionnaire.

IV. RESULTS AND DISCUSSION

1. Collaboration Performance

Table 2 reports synchronization latency for 2, 5, and 10 concurrent users. Mean latency remains below 100 ms, which is generally acceptable for perceived real-time editing.

Table 2 — CRDT Synchronization Latency

Concurrent Users	Mean Latency (ms)	95th Percentile Latency (ms)
2	45	78
5	62	112
10	89	187

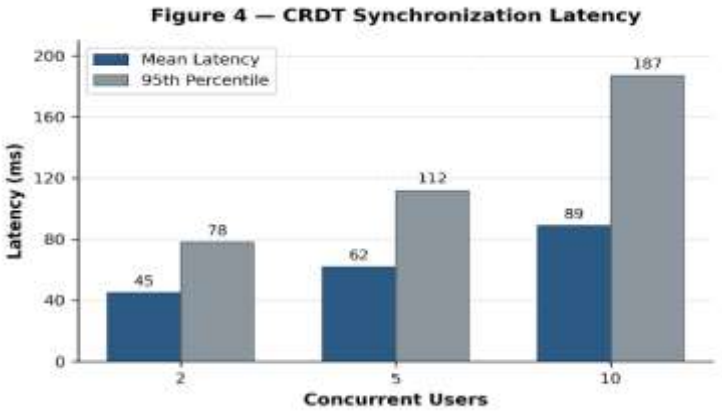


Figure 4 — CRDT Synchronization Latency Bar Chart

The increase in latency with user count is expected due to additional broadcast and awareness traffic. The 95th-percentile latency for 10 users is higher but remains within practical bounds for small-team collaboration. The CRDT approach avoids centralized transformation and supports offline editing, which is advantageous for distributed teams [2, 3].

2. Ai Response Performance

Table 3 compares external API and local model response times. Local inference with a CodeLlama-7B model achieved lower mean latency than external API calls in the tested environment, mainly because network overhead and API queuing were avoided. However, local inference required GPU acceleration; CPU-only inference was approximately four times slower.

Table 3 — AI Response Latency by Task Type and Deployment Mode

Task Type	API Mean (s)	API P95 (s)	Local Mean (s)	Local P95 (s)
Code Completion	1.2	2.1	0.8	1.4
Code Generation	3.4	5.8	2.1	3.6
Code Explanation	2.8	4.2	1.9	2.8
Bug Detection	3.1	4.9	2.0	3.2

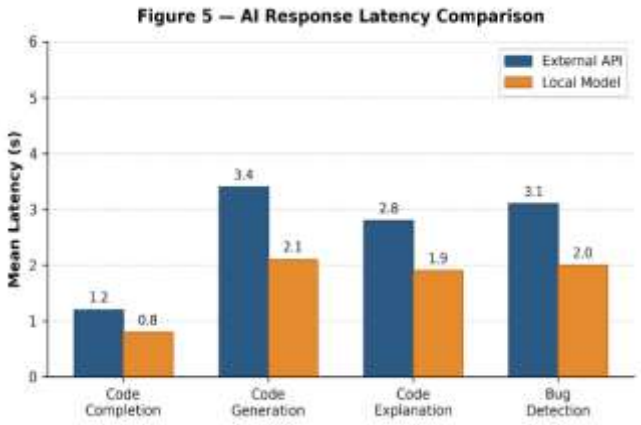


Figure 5 — AI Response Latency Comparison

The results show that the asynchronous AI design prevents AI latency from affecting collaboration latency. Code completion is the fastest task because it uses shorter prompts. Code generation and bug detection require longer prompts and more tokens, and therefore have higher latency.

3. User Study

A small user study with eight developers was conducted.

Table 4 — User Study Results

Evaluation Item	Mean Likert Score (1–5)
Ease of real-time collaboration	4.3
Usefulness of AI code completion	4.1
Usefulness of code explanation	4.4
Usefulness of bug detection	4.0
Overall satisfaction	4.2

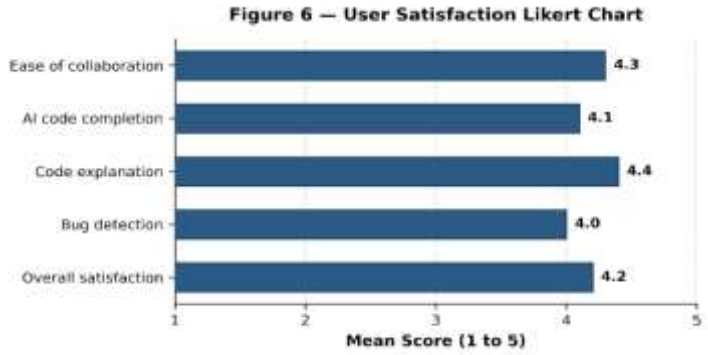


Figure 6 — User Satisfaction Likert Chart

Participants used the platform for pair-programming sessions of approximately 45 minutes. The questionnaire used a five-point Likert scale. Participants reported that AI suggestions integrated naturally into the collaborative workflow. Code explanation was particularly valued for onboarding and understanding unfamiliar code. Some participants noted that AI completions occasionally conflicted with team coding standards, suggesting the need for project-specific prompting and style configuration.

4. Failure Analysis

Three failure modes were observed. First, rare CRDT synchronization inconsistencies occurred with complex concurrent edits over the same region. These resolved on the next update and are consistent with availability-oriented CRDT behavior [2, 3]. Second, the local 7B model sometimes ignored context from distant parts of the document. Larger models and improved context-window handling would reduce this issue. Third, prompt-injection attempts through code comments were identified during security testing; input sanitization and prompt hardening were applied as mitigations [14].

Table 5 — Failure Modes, Causes, and Mitigation

Failure Mode	Observed Cause	Mitigation
Rare CRDT inconsistency	Complex concurrent edits over the same text region	Next Yjs update resolves state; periodic snapshots
Limited AI context awareness	Local 7B model ignores distant document context	Larger context window; retrieval of relevant code

Failure Mode	Observed Cause	Mitigation
Prompt injection through comments	Malicious text embedded in code comments	Input sanitization; hardened prompt templates
Higher AI latency	Long prompts for generation/bug detection	Asynchronous requests; streaming responses
External API dependency	Network variability and API queuing	Local model fallback; timeout handling

5. Discussion

The decoupled architecture provides a practical balance between collaboration performance and AI capability. External APIs provide access to stronger models with minimal infrastructure, but raise data-privacy and cost concerns. Local models provide privacy and predictable cost but require GPU infrastructure. The proposed system supports both modes, allowing organizations to choose according to their constraints.

The main limitation of this study is scale. Evaluation was limited to 10 concurrent users and short user sessions. Long-term studies and larger rooms are needed. In addition, systematic benchmarking of AI code quality against standardized datasets remains future work.

V. CONCLUSION

This paper presented an AI-enhanced collaborative code editor that integrates CRDT-based real-time synchronization with LLM-powered code assistance. The system uses Yjs, Socket.IO, Monaco Editor, and an asynchronous AI microservice to support collaborative editing without blocking AI inference. Experimental results show sub-100 ms mean synchronization latency for up to 10 users and practical AI response times across multiple task types. The architecture supports both external API and locally hosted model deployment, providing flexibility for different infrastructure and privacy requirements. Future work will focus on larger-scale evaluation, project-specific fine-tuning, federated AI processing, and integration with version control and CI/CD workflows.

REFERENCES

1. C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," ACM SIGMOD Record, vol. 18, no. 2, pp. 399–407, 1989.
2. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in Proc. Symposium on Self-Stabilizing Systems, 2011, pp. 386–400.
3. P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near real-time peer-to-peer shared editing on extensible data types," in Proc. International Conference on Supporting Group Work, 2016, pp. 39–49.
4. M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.
5. B. Rozière et al., "Code Llama: Open foundation models for code," arXiv preprint arXiv:2308.12950, 2023.
6. R. Li et al., "StarCoder: May the source be with you!," arXiv preprint arXiv:2305.06161, 2023.
7. OpenAI, "GPT-4 technical report," arXiv preprint arXiv:2303.08774, 2023.
8. Google, "Gemini: A family of highly capable multimodal models," Technical Report, 2024.
9. Microsoft, "Monaco Editor documentation," 2024. [Online]. Available: https://microsoft.github.io/monaco-editor/
10. Socket.IO, "Socket.IO documentation," 2024. [Online]. Available: https://socket.io/docs/
11. A. Ahmadi, M. Rezaei, and S. Karimi, "Multi-user code assistant: Aggregating developer context for intelligent suggestions," IEEE Transactions on Software Engineering, vol. 49, no. 8, pp. 4123–4138, 2023.
12. GitHub, "GitHub Copilot documentation," 2024. [Online]. Available: https://docs.github.com/copilot
13. A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems, vol. 30, 2017.
14. F. Perez and I. Ribeiro, "Ignoring previous prompt: Attack techniques for language models," arXiv preprint arXiv:2211.09527, 2022.

15. Docker, "Docker Compose documentation," 2024. [Online]. Available: https://docs.docker.com/compose/
16. MongoDB, "MongoDB documentation," 2024. [Online]. Available: https://www.mongodb.com/docs/

17. M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, 2015.
18. Vercel, "Next.js documentation," 2024. [Online]. Available: https://nextjs.org/docs