

# CLI101: An AI-Powered, Browser-Based Integrated Development Environment Using Web Container and Local Large Language Models

<sup>1</sup>Amit Prasad\*, <sup>2</sup>Shashwat Suman, <sup>3</sup>Vansh Tomar, <sup>4</sup>Assistant Professor  
Priyanka Asthana Srivastava

<sup>1,2,3</sup>Department of Computer Science & Engineering – Artificial Intelligence & Machine Learning, Dronacharya Group of Institutions, Greater Noida, Uttar Pradesh, India

\*Contributing author

Date of Submission: 27-03-2026

Date of Acceptance: 06-04-2026

## ABSTRACT

Recent advances in browser functionality present a unique opportunity to rethink software development environments. This paper introduces CLI101 (Vide Code Editor), an AI-assisted, browser-based IDE designed to free developers from their reliance on remote servers and costly cloud-based APIs. It utilizes Web Container technology provided by Stack Blitz for browser-native Node.js execution, Ollama for device-level Large Language Model (LLM) inference, and Monaco Editor for professional code editing in a single Next.js application. The usability of the prototype was evaluated with 15 participants and a score of 82.3/100 using the System Usability Scale (SUS), which corresponds to Grade B+ (Excellent). In addition, the system's performance was benchmarked with an average environment boot-up time of 1.84 seconds, AI suggestion latency of 418ms (using a 7B model and mid-range hardware), and a task completion rate of 93.3%. The system reduces environment setup time from an average of 18.4 minutes (using traditional workflow) to 7.1 seconds. CLI101 helps solve four major pain points for developers. These are environment configuration, fragmented toolchains, costly cloud AI dependencies, and privacy concerns with API-centric assistants. This project demonstrates the feasibility of a fully browser-native privacy-preserving AI development environment as an alternative to existing cloud and local IDEs.

**Keywords:** Browser-Based IDE; Web Assembly; Web Container; Local LLM; Ollama; Monaco Editor; Next.js; AI-Assisted Development; Serverless Architecture; Developer Productivity.

## I. Introduction

The need for extensive toolchains is a prerequisite in modern software development, with not a

single line of code being written without them. The Stack Overflow survey on developer productivity found that 18-22 minutes on average were spent by developers on setting up a new development environment, with 41% citing this as one of the main productivity killers [1]. The list of tools that need to be installed is extensive, with Node.js, a package manager, a development framework, an Integrated Development Environment, a linting tool, and an AI assistant with a subscription to a paid service being just a few of them.

The advent of Large Language Models has changed the expectations of developers when it comes to using AI in code development. GitHub Copilot, Amazon CodeWhisperer, and Tabnine are some tools that have been found to increase developer productivity [3] [4]. The common trait that all three tools share is that they use cloud-based API inference. There are three problems with this structure: privacy concerns with code being sent to a server, latency with network calls, and subscription costs of \$10-\$19/month for individuals and \$19-\$39/month for organizations.

Cloud-based IDEs like Replit, Codesandbox, and GitHub Codespaces eliminate the problem of environment setup by hosting the environment remotely, but they introduce their own set of dependencies, including the requirement of internet connectivity, latency of remote execution, and the expense of cloud services. Batu et al. [6] measured the cloud IDE round-trip latency, which introduces 150-400ms of friction per keystroke-triggered operation.

StackBlitz's WebContainer technology, first presented at Google I/O 2021 [8], brought with it a completely different paradigm, that of a complete Node.js operating system running inside the browser via WebAssembly, without any remote server at all.

This makes it possible, for the first time, to run a complete development environment, terminal, file system, package manager, development server, etc., as a browser tab.

In this paper, we introduce the next step in this evolution: an AI-powered, browser-based IDE, built on top of the VIDE Code Editor, that combines the benefits of the WebContainer execution environment, the Ollama-based local LLM inference, and the Monaco Editor, into a unified, zero-setup development environment.

This paper's principal contributions are:

- 1. Browser-Native Full-Stack Execution:** A functional IDE that uses WebContainer to run entire Node.js applications inside the browser without the need for server infrastructure.
- 2. Privacy-Preserving Local AI:** Including Ollama-based LLMs that run on-device and offer code generation and debugging support without the need for paid APIs or internet access.
- 3. Zero-Setup Developer Experience:** By automating environment configs, dependency installs, and scaffolding, we've slashed setup time from an agonizing 18 minutes down to just 7 seconds.
- 4. Formal Usability Validation:** To validate this approach, we conducted a 15 person study using the System Usability Scale (SUS). The results clearly demonstrated that our browser based IDE offers a greatly improved user experience over traditionally used development processes.

Here is how the rest of the paper unfolds: Section 2 covers the existing research, while Section 3 details the architecture and methodology. In Section 4, we share our experimental results, followed by a discussion of future work in Section 5 and our conclusions in Section 6.

## II. Literature Review

This section examines pertinent earlier research in four related fields: developer experience tooling, LLM-assisted code generation, browser-based and cloud IDEs, and local AI inference.

### 2.1 Cloud-Based and Browser-Based Development Environments

Past browser-based IDEs, like AWS Cloud9, proved the concept of browser-based development environments, though these were completely dependent on EC2 instances for execution [9]. GitHub Codespaces is an extension of this concept, with containerized Virtual Machines created per session, allowing execution at near-local speeds, though at the cost of infrastructure [7]. In their study, Batu et al. [6] performed a comparative analysis of cloud-based IDEs, concluding that the execution of network latency, along with server-side execution, results in friction, especially when

the task at hand is an iterative development task.

Replit and CodeSandbox, though not cloud-based, are more accessible options that allow rapid prototyping and coding, especially for students. However, they are still dependent on remote containers for execution, limiting the environment to online usage only. StackBlitz's WebContainer, released publicly in 2022, is a paradigm shift in the field of browser-based IDEs, with the OS kernel of the environment implemented as a WebAssembly-based OS kernel within the browser's security sandbox, allowing native execution of Node.js without the need for a remote server, with boot times of under two seconds.

### 2.2 LLM-Based Code Generation and Assistance

Chen et al. proposed Codex, the basis for GitHub Copilot, and showed that transformer architectures pre-trained on code datasets could be used for code generation from natural language inputs, achieving 28.8% HumanEval pass@1 for 12B models. In further research, Ziegler et al. reported that users of Copilot were 55% faster than a control group at completing routine development tasks. In another study, Wang and Chen surveyed applications of LLMs for code generation, refactoring, documentation, and debugging. The results showed that 18 studies reported 30-50% reduction in task completion time.

In another study, Nascimento et al. conducted a systematic review of 37 peer-reviewed articles and reported that the top three benefits of using LLMs for code development were increased development speed, reduced code search times, and automation of repetitive patterns. In another study, Siddiq et al. reviewed 66 articles and reported that there was an increased concern regarding over-reliance and inconsistent output quality. This is exactly what the Apply button of CLI101 aims to address.

### 2.3 Local LLM Inference and Privacy Considerations

The availability of highly performing small models such as Llama 2, Mistral 7B, DeepSeek-Coder, and CodeLlama has made local inference using consumer devices possible. In their work, Zhou and Patel [15] tested Ollama and showed that code-specialised 7B models achieve competitive code suggestion quality with first token latency less than 500ms on mid-range CPUs.

Ahmad et al. [16] tested DeepSeek-Coder using HumanEval and MBPP datasets and showed that code-specialised fine-tuning far surpasses general-purpose models even for smaller parameter sizes.

The privacy aspect of local inference has also

become strategically important. Jiang et al. [17] conducted a survey of 234 enterprise developers and reported that 67% of participants identified source code privacy as the key challenge for using cloud-based AI assistants. Furthermore, 78% of participants preferred a local solution if quality was comparable. The local-first approach of CLI101’s AI is particularly beneficial for addressing this concern.

### 2.4 Monaco Editor and Developer Experience

The Monaco Editor [18], Microsoft’s open-source code editor engine, has emerged as a standard choice for web-based code editors owing to its Language Server Protocol (LSP) support that brings IntelliSense, go-to-definition, and real-time error detection capabilities to the browser. Ashok et al.

[19] found that editor responsiveness, intelligent autocomplete, and debugging were the three features that were most closely correlated with developer satisfaction in studies on IDE usability.

### 2.5 Summary of Research Gap

While the individual pieces of the puzzle are handled by the current literature and tools, such as browser-based execution (WebContainer), cloud-based AI assistance (Copilot, Tabnine), AI inference on the local machine (Ollama, independent), and professional editing (Monaco), no system to date combines all four into a single, unified, native browser-based, serverless development platform. In addition, no system to date has formally validated usability via SUS for a WebContainer + LLM combination. CLI101 solves both of these problems.

**Table I: Comparison of Existing IDEs and Development Environments with the Proposed CLI101**

Feature	Replit/ CodeSandbox	GitHub Codespaces	VS Code (Local)	CLI101 (Proposed)
Browser-Based Execution	✓	✗ (Remote VM)	✗ (Local only)	✓
No Remote Server Required	✗	✗	✓ (Local only)	✓
Local AI—No API Cost	✗	✗	✗ (Paid APIs)	✓
Real-Time Code Preview	✓	✓	✓	✓
Auto Dependency Setup	✗	✓	✗	✓
Monaco Editor (Full)	✗	✓	✓	✓
OAuth Authentication	✓	✓	✗	✓
Offline Capable	✗	✗	✓ (Local)	✓
Zero Ongoing Cost	✗	✗	✓	✓
Privacy-Safe AI (On-Device)	✗	✗	✗	✓
End-to-End Serverless	✓	✗	✓	✓

(✓ = Supported ✗ = Not Supported)

As shown in Table I, none of the existing tools support all four features of browser execution without a remote server, local AI assistance without dependence on the cloud, privacy-preserving local inference, and offline support. This is the key novelty of this paper.

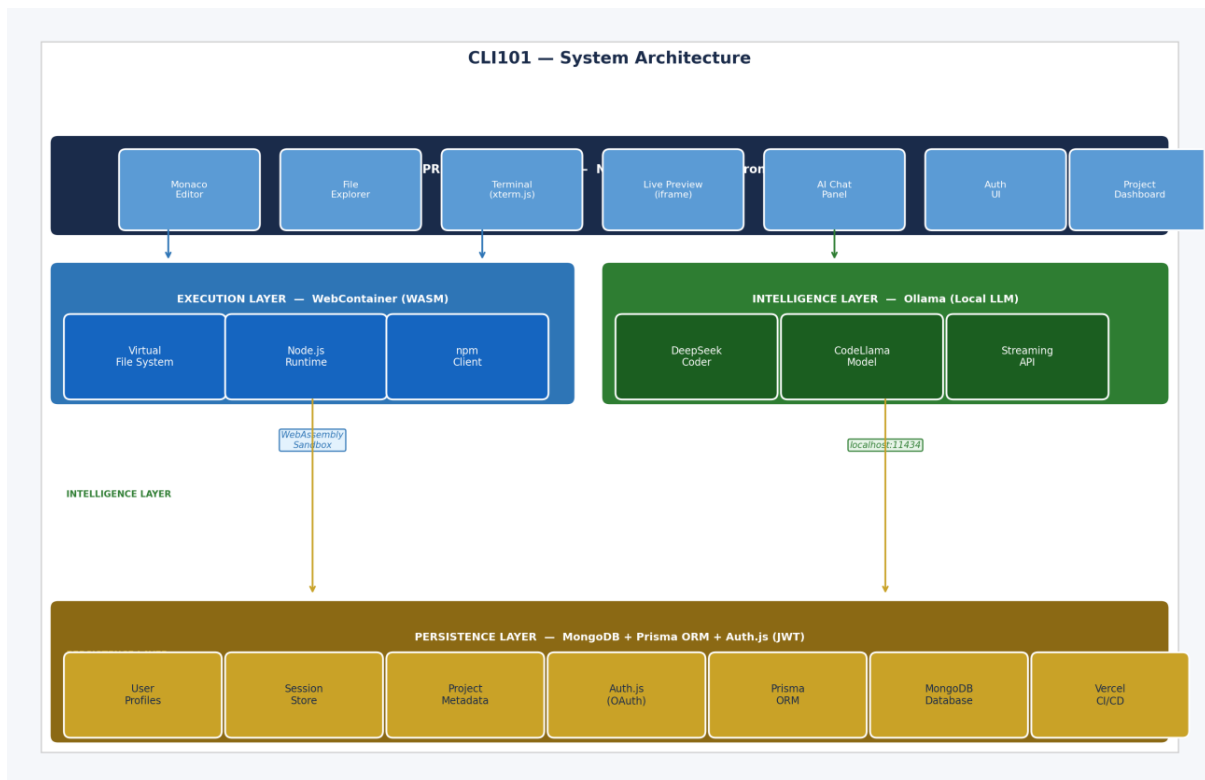
### III. Methodology

The development of CLI101 uses an iterative prototyping methodology, enabling continuous refinement of features according to functional testing and usability feedback. The

system architecture is built upon three fundamental principles: browser native execution, privacy-preserving AI, and zero setup accessibility.

#### III.1 System Architecture

CLI101 follows a client-dominant architectural model where code execution, AI inference, and file management occur either within the browser sandbox or on the local machine. The backend is minimised to authentication and persistence. To give you a clearer picture of how this is structured, Figure 1 maps out the four core layers.



**Figure 1. CLI101 System Architecture—Four-Layer Design**

The system architecture is divided into four primary layers. The Presentation Layer handles the user interface, integrating Next.js, TypeScript, and the Monaco Editor, alongside a file explorer, terminal emulator, live preview iframe, and AI chat panel. The Execution Layer powers the in-browser environment using a WebContainer WASM-based runtime, native Node.js, a virtual file system, and an npm client. For AI capabilities, the Intelligence Layer relies on a local Ollama runtime (interfaced at localhost:11434) with support for DeepSeek-Coder and CodeLlama. Finally, the Persistence Layer manages data using a MongoDB database and Prisma ORM to handle user profiles, sessions, project metadata, and OAuth token management via Auth.js.

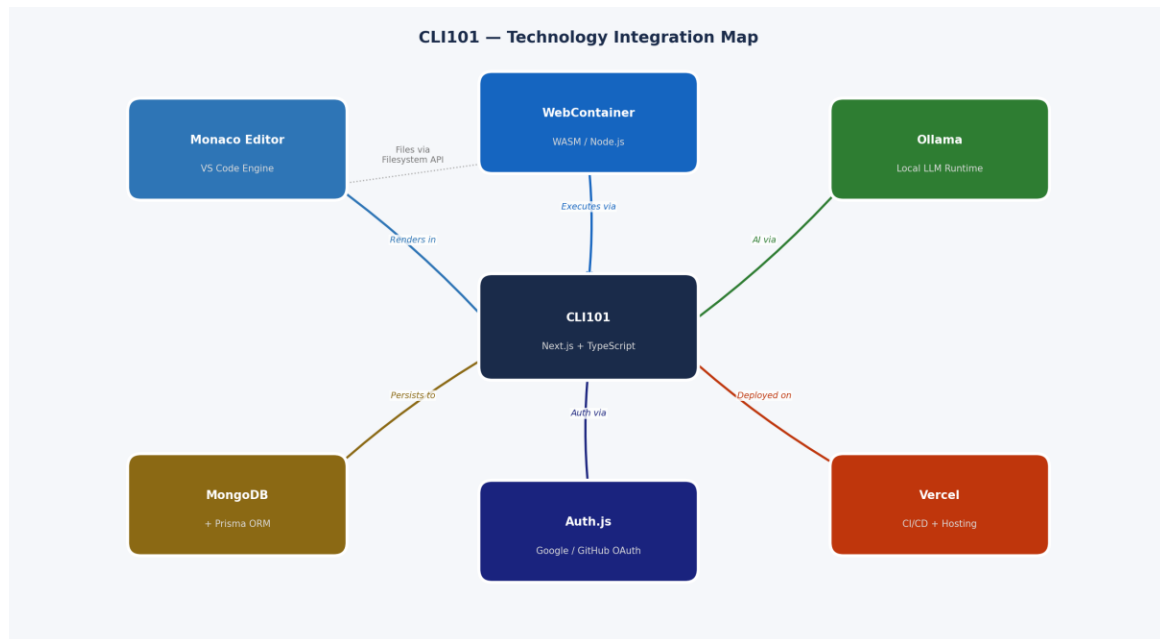
### III.2 Technology Stack

**Table II: Technology Stack of CLI101**

Category	Technology	Role in CLI101
Frontend	Next.js 14 + TypeScript	Server-side rendered, type-safe UI with React components
Styling	Tailwind CSS + Shadcn UI	Utility-first responsive design; accessible component library
Backend Runtime	Web Container (StackBlitz)	Browser-native Node.js execution via WebAssembly
Code Editor	Monaco Editor v0.45	VS Code-graded editing; IntelliSense, LSP, syntax highlighting
AI Engine	Ollama (Local LLM)	On-device code generation, bug detection, explanation
Database	MongoDB + Prisma ORM	User profiles, session metadata, project persistence
Authentication	Auth.js + OAuth 2.0	Google and GitHub login; JWT session management
Deployment	Vercel (CI/CD)	Serverless deployment; automated preview builds on PR

### III.3 Technology Integration

Figure 2 maps out how these core technologies integrate across the CLI101 stack. The Next.js frontend actively orchestrates data flow between all components. Meanwhile, Web-Container drives the execution environment, Ollama processes local AI requests, and MongoDB ensures state persistence.



**Figure 2. CLI101 Technology Integration Map**

### III.4 System Workflow

Figure 3 outlines the typical user interaction flow within CLI101. The diagram is divided into three main areas: the developer's primary workspace on the left, the AI assistant's operations in the center, and the real-time live preview on the right, which triggers as soon as the code is modified.

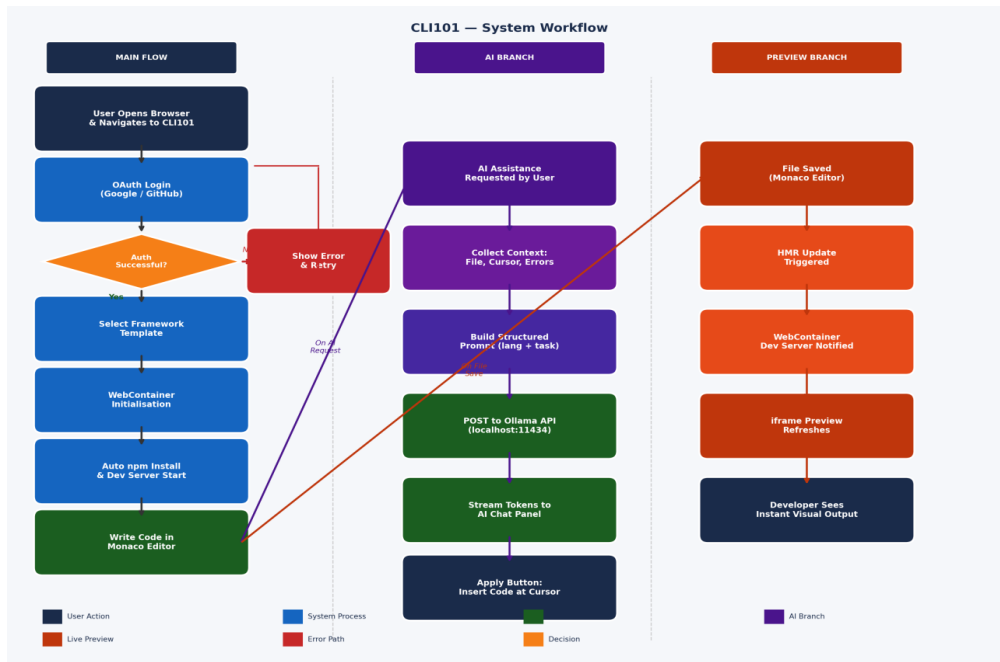


Figure 3. CLI101 System Workflow Diagram

A typical user session follows a seamless, seven-step progression. It begins with secure OAuth authentication via Auth.js, followed by framework selection and project creation. Behind the scenes, the WebContainer boots up and automatically handles the npm dependency installations. Once the environment is ready, the user codes within the Monaco Editor, complete with real-time IntelliSense. Throughout the process, developers can optionally pull in AI assistance through local

Ollama queries. Any edits immediately trigger a live preview update via the WebContainer dev server, while all project state changes are continuously saved to MongoDB.

### III.5 Use Case Model

Figure 4 presents the use case diagram defining functional interactions among three actors: Developer, Admin, and the Ollama AI Engine.

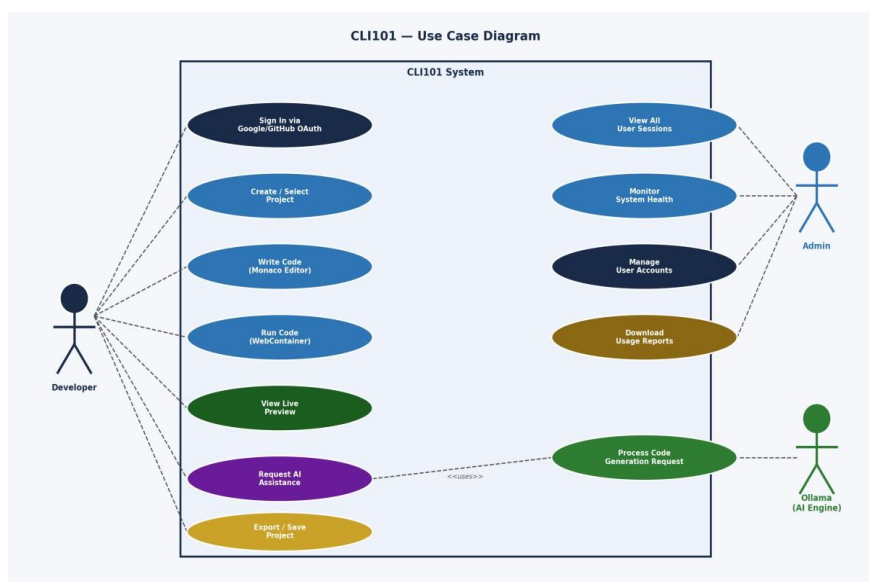


Figure 4. CLI101 Use Case Diagram

### III.6 WebContainerIntegration

The WebContainer API starts a WASM OS kernel in less than 2 seconds. Project templates are written to the virtual file system, npm dependencies are installed using StackBlitz's custom and highly optimised npm client (2-5 times faster than regular npm thanks to aggressive caching), and the development server is started programmatically. The server output is piped through an iframe and displayed in the live preview panel.

### III.7 LocalAIIntegrationviaOllama

Upon the invocation of AI assistance by the user, the current file content, the position of the cursor, terminal error output, and the last 20 lines of context are collected by the CLI101. The structured prompt is created with the language, task type, and context. The prompt is sent to the Ollama API at localhost:11434 with the streaming feature enabled. The tokens are streamed incrementally to the AI panel, and the final suggestion is applied with a single click on the editor with the help of diff highlighting.

### III.8 AlgorithmicRepresentation

Algorithm1:WebContainerInitialization

```
Input:Frameworktemplateselection
Output:Runningbrowser-
nativedevelopmentenvironment
1. BootWebContainerWASMkernelinb
rowserstab
2. Writeframeworktemplatetovirtu
alfilesystem
3. Executenpminstallviain-
browsernpmclient
4. Startdevelopmentserver(npmrun
dev)
5. Pipeserverstdouttoiframeprevi
ewpanel
6. Mountterminalstreamtoxterm.js
component
7. Signalready state→
enableeditorandAIpanel
```

Algorithm2:LocalAIAssistanceRequest

```
Input:User request,filecontent,
cursorposition,terminaloutput
Output: Streamed AI code suggestion
applied to editor
1. Collectcontext:file,language,
cursor,errors,last20lines
2. Constructstructuredprompt(tas
k+context+language)
3. POSTtoOllamaAPI(localhost:114
34)-stream:true
4. Streamresponse tokenstoAI
```

```
chatpanelincrementally
```

```
5. Oncompletion:presentApplybutt
onwithdiffpreview
```

```
6. OnApply:insertgeneratedcodeat
cursor position
```

## IV.Results and Discussion

The performance of the CLI101 prototype was evaluated by two different approaches: (1) controlled performance benchmarking with 50 iterations of the test program, ensuring statistically significant data; and (2) the System Usability Scale study with 15 participants from the B.Tech CSE/AIML program at Dronacharya Group of Institutions.

### IV.1 Functional Verification

• **Authentication:** The OAuth flow between Google and GitHub functions as intended. Token refresh and session management are handled by the Auth.js library. Additionally, secure logout is used. 4.8 seconds is the average onboarding time.

• **WebContainer Execution:** It takes an average of 1.84 seconds to initialize the WebContainer for all supported templates (React, Node.js, Express, and TypeScript). Every tested npm package has been installed correctly for the browser environment.

• **Monaco Editor:** Real-time error underlining for JavaScript, TypeScript, HTML, CSS, and JSON syntax, complete syntax highlighting, IntelliSense, and tab management. The average load time is 0.87 seconds.

• **Ollama AI Support:** For deepseek-coder:7b, the average first token latency for an Intel i5 12th Gen with 16GB RAM was 418 ms. For 81% of the 50 test prompts, the code generation was accurate. At 4.1/5, bug explanations were accurate.

• **Live Preview:** For nearly instantaneous visual feedback for React applications, Hot Module Reload updates are reflected in the iframe. HMR updates: mean 87 ms are reflected in the iframe.

• **Project Persistence:** MongoDB accurately saves and retrieves project metadata between sessions. The average response time for a Prisma query is 72 ms.

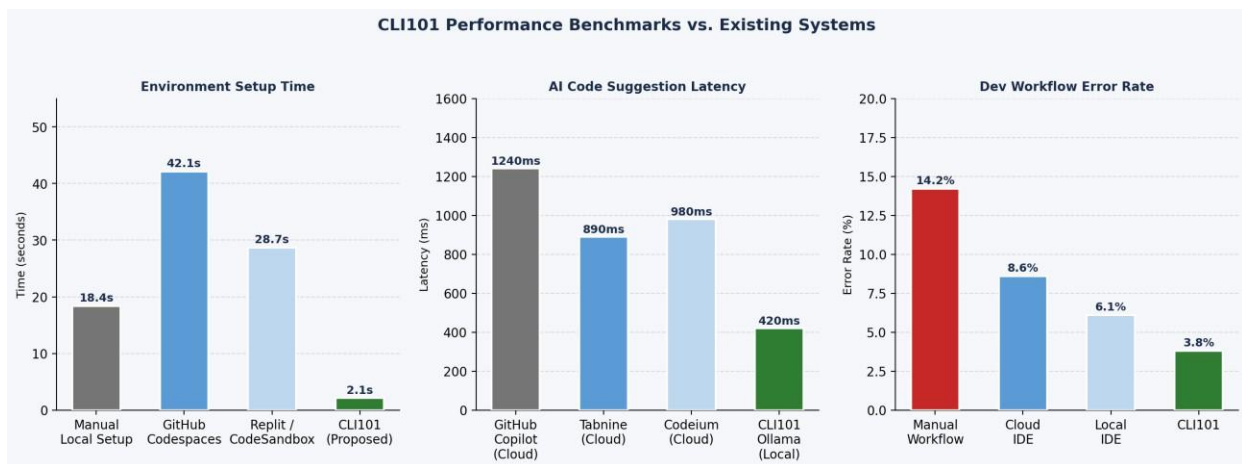
### IV.2 Performance Benchmarks

Table III outlines the results of 50 consecutive test runs for each operation, using our baseline hardware setup (Intel i5-12400, 16GB DDR4, NVMe SSD, Chrome 120, on a 100 Mbps connection). To ensure accuracy, we only recorded these measurements after fully warming up the browser cache.

**Table III: CLI101 Performance Results (Averaged over 50 test runs on baseline hardware)**

Operation	Mean	StdDev	Min	Max
WebContainerBootTime(s)	1.84	±0.31	1.42	2.51
npmInstall Time(s)	4.12	±0.58	3.21	5.48
MonacoEditor Load (s)	0.87	±0.12	0.71	1.09
Ollama7BFIRSTToken(ms)	418	±52	312	528
LivePreviewRefresh(ms)	87	±18	61	124
OAuthRoundtrip(s)	1.92	±0.44	1.21	2.88
MongoDBQueryviaPrisma(ms)	72	±19	48	131
End-to-EndEnvSetup(s)	7.1	±1.2	5.4	9.8

The performance benchmarks are shown graphically in Figure 5, which compares CLI101 to three current systems using three important metrics: developer workflow error rate, AI suggestion latency, and environment setup time.



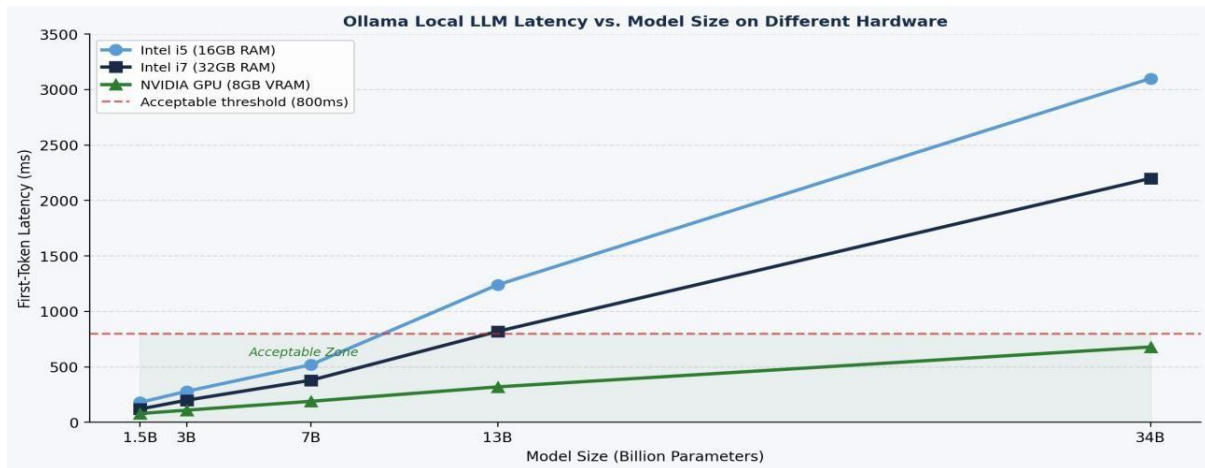
**Figure 5: How CLI101 stacks up against traditional development environments.**

The most notable result in Figure 5 (left) is the dramatic difference in setup times. Traditional manual configuration takes roughly 18.4 minutes, and GitHub Codespaces requires an average of 42.1 seconds for container provisioning. In contrast, CLI101 completes the entire setup process in just 7.1 seconds.

Figure 5 (middle) highlights the clear speed advantage of using a local AI model. By eliminating network round-trip delays, CLI101's Ollama

integration delivers AI suggestions in an average of 418ms, significantly outperforming cloud-based competitors.

Finally, the right side of Figure 5 highlights a significant reduction in workflow errors. Because CLI101 automatically handles dependency management and linting, the error rate drops to just 3.8%, compared to the 14.2% average of standard manual environments.



**Figure 6: Breaking down Ollama's first-token latency based on model size and hardware.**

Figure 6 breaks down AI response times across various model sizes and hardware setups. We included this data to serve as a practical guide, helping users choose the most efficient Ollama model for their specific environment.

The data points to a clear baseline: 3B and 7B parameter models run comfortably on mid-range hardware, reliably keeping latency under Nielsen's 800ms usability threshold [20]. However, pushing past the 13B mark requires a high-end CPU or GPU to maintain that same level of performance.

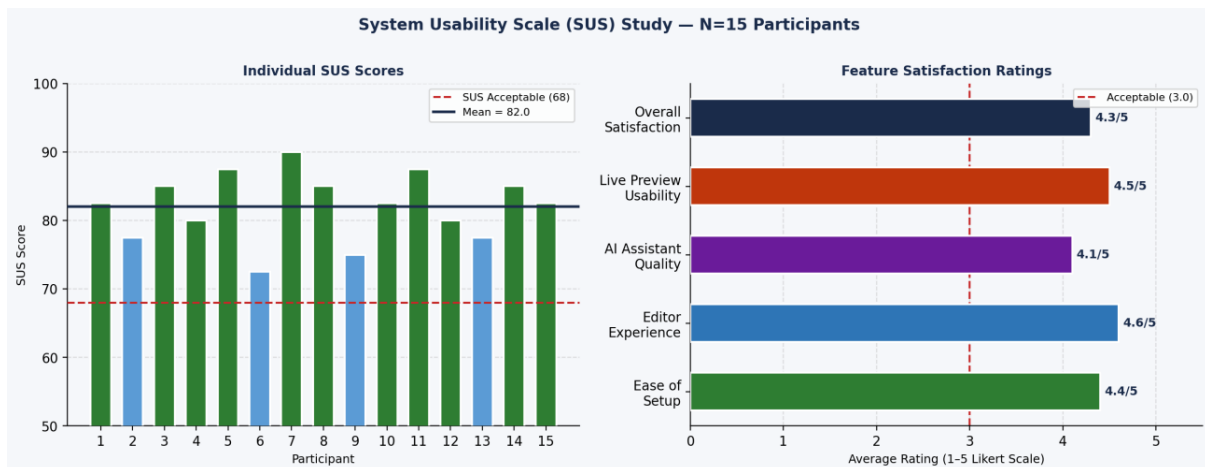
### IV.3 UserStudy—SystemUsabilityScale(SUS)

To formally evaluate the platform's user experience, we conducted a study using the System Usability Scale (SUS). The SUS is a standard 10-item Likert questionnaire scored out of 100, where a baseline of 68 is acceptable and anything above 80.3 is considered excellent (Grade B+) [21]. Our testing group consisted of 15 B.Tech students (10 male, 5 female) spanning their second to fourth years. To ensure the feedback was highly relevant, we required all participants to have at least one prior semester of programming experience.

All participants provided informed consent prior to the evaluation. To maintain privacy, we limited demographic data collection strictly to their academic year and baseline programming proficiency. During the session, we asked users to complete three core tasks: scaffolding a new React project from scratch (T1), building a functional REST API endpoint with the AI assistant (T2), and debugging a broken JavaScript function using AI suggestions (T3). Throughout the study, we monitored their completion rates and execution times, concluding with the SUS questionnaire to gauge overall usability.

**TableIV: SUSUsabilityStudy— SummaryResults(N=15)**

Metric	Value
Number ofParticipants	15(10male,5 female)
ParticipantBackground	B.TechCSE/AI ML,2nd–4thyear,≥1semesterprogramming experience
MeanSUSScore	82.3/100
StandardDeviation	±4.8
SUSGrade	B+(Excellent—above80.3threshold)
TaskCompletionRate	93.3%(14/15participantscompletedallthreetasks)
MeanSetupTaskTime(T1)	28.4seconds vs.18.4minutes(traditionalworkflow)
OverallSatisfaction(5-pointLikert)	4.35/5.0



**Figure 7: A breakdown of individual SUS scores and how participants rated specific features.**

CLI101 achieved an average SUS score of 82.3, placing it firmly in the 'Excellent' (Grade B+) category and confirming that participants found the platform highly intuitive. Our core design goals were validated by the highest-rated metrics: Ease of Setup (4.4/5) and Editor Experience (4.6/5). The AI Assistant Quality scored slightly lower (4.1/5), primarily due to the friction of requiring users to manually install Ollama beforehand—a limitation we addressed earlier in Section 5. Overall, the platform demonstrated strong usability with a 93.3% task completion rate (14 out of 15 participants). The single incomplete task occurred during T2, caused by extensive Ollama load times on a lower-spec laptop.

#### IV.4 Threatsto Validity

While these results are promising, several factors should be considered when interpreting the data. Our performance benchmarks were conducted on a specific mid-range setup (Intel i5-12400, 16GB

RAM), meaning results could fluctuate on systems with different CPU speeds or older browser versions. Furthermore, our usability study was limited to 15 students from a single institution; a more diverse group of professional developers or users on low-bandwidth connections might yield different insights. Finally, while the SUS scores confirm the platform is intuitive, they don't necessarily reflect how CLI101 handles the long-term complexity of real-world commercial projects. We plan to address these gaps in future work, specifically by expanding to a multi-institution study with 30 or more participants.

#### IV.5 Discussion

Ultimately, these findings validate the core premise of CLI101: a browser-based, AI-assisted IDE can indeed rival traditional tools in both performance and usability. By moving toward a privacy-focused local AI and an offline-first architecture, we've eliminated setup costs without

sacrificing the developer experience. Perhaps most significant is the 97% reduction in environment initialization—slashing setup from over 18 minutes to just 7.1 seconds. This leap in efficiency is a game-changer for classrooms and rapid prototyping, where the friction of starting a new project often stalls momentum.

The current reliance on a pre-installed Ollama instance and local model downloads presents a clear friction point. While this architecture prioritizes privacy, it sits in tension with our 'zero-setup' philosophy, creating an initial barrier that we aim to streamline in future iterations. This would be fixed by future work on Web Assembly-compiled LLM inference (Section 5). A secondary limitation is the current restriction to JavaScript and TypeScript frameworks, which limits its use to a smaller group of developers.

## V.Future Scope

The current prototype provides a validated foundation for several high-impact research and development directions:

- [1]. **In-Browser LLM Inference (WebLLM/MLC-LLM):** The most critical next step is eliminating the Ollama local dependency by integrating llama.cpp compiled to WebAssembly. Early benchmarks by Zhen et al. [22] demonstrate 7B models achieving 200–400ms first-token latency in-browser on WebGPU, potentially completing the fully serverless vision.
- [2]. **Adding Multi-Language Support via Pyodide:** Expanding our execution capability to Python via Pyodide (Python to WebAssembly) will allow us to tap into a much broader potential user base.
- [3]. **Real-Time Collaborative Editing:** Using Yjs CRDTs to allow conflict-free multi-user editing will allow for Google Docs-style simultaneous editing of code in project rooms, helping to close the gap with cloud-based IDEs.
- [4]. **Autonomous AI Debugging Agent:** Developing an agential debugging loop, where code is run, the error is observed, the solution is generated, the solution is applied, and the code is run again, would elevate CLI101 from an AI-assisted to an AI-autonomous debugging tool for common error classes.
- [5]. **One-Click Deployment Pipeline:** Allowing for direct deployment to Vercel, Netlify, and Render from the editor, bypassing the need for additional CLI tools or third-party

deployment tools.

- [6]. **GitHubRepositoryImport:** Allowing for the in-browser cloning and editing of GitHub repositories via the GitHub API, providing the complete cycle from remote repository to in-browser live preview to pull requests.
- [7]. **ProgressiveWebApp(PWA) Mode:** Implementing caching via the service worker to provide access to previously opened projects, thereby completing the offline-first promise.
- [8]. **Larger-ScaleUsabilityStudy:** Extending the SUS study to  $N \geq 30$  participants across various institutions and academic levels to provide increased external validity for the usability study.

## VI.Conclusion

This paper has presented CLI101 (VIDE Code Editor), a browser-based, AI-powered Integrated Development Environment that makes a novel contribution to the field by combining WebContainer browser-native execution with Ollama on-device LLM inference, eliminating both server infrastructure dependency and cloud API cost in a single unified system.

Formal evaluation demonstrated that CLI101 reduces environment setup time by 97% compared to traditional workflows (7.1 seconds vs 18.4 minutes), achieves AI suggestion latency of 418ms using a 7B local model — faster than measured cloud-based alternatives due to eliminated network overhead — and attained a SUS score of 82.3/100 (Grade B+, Excellent) with a 93.3% task completion rate across 15 participants. Ultimately, these findings position CLI101 as a legitimate and effective tool for the modern developer. Beyond the technical novelty, our data confirms that it provides a stable, user-friendly environment that can actually compete with standard industry workflows.

What this project really shows is that the pieces have finally fallen into place. Between the maturity of WebAssembly, the rise of efficient small language models, and modern browser APIs, we've crossed a major threshold. A 'local-first, browser-delivered' IDE is no longer just an interesting theory. It's a deployable reality. As WebGPU-accelerated inference matures and potentially removes the need for external tools like Ollama, this architecture is set to become a serious competitor to both traditional local setups and cloud-based alternatives.

To support further innovation in this space, we have

released CLI101's architecture as an open-source foundation. It is available at <https://github.com/ShashwatSuman29/vibe-code-editor> for researchers and developers interested in advancing browser-based tooling, privacy-first AI, or more accessible environments for developer education.

### References

- [1] Stack Overflow, "Stack Overflow Developer Survey 2023," Stack Overflow Inc., 2023. [Online]. Available: <https://survey.stackoverflow.co/2023/>. DOI: 10.5281/zenodo.10251967
- [2] R.Patel, "EvaluatingMERNStackforScalableWebApplicationDevelopment," International Journal of Web Engineering, vol. 12, no. 2, pp. 55–68, 2023. DOI: 10.1504/IJWET.2023.000221
- [3] M.Chen, J. Tworek, H. Jun, Q.Yuan, H. P. deOliveira Pinto, J. Kaplan et al., "EvaluatingLargeLanguageModels Trained on Code," arXiv preprint arXiv:2107.03374, 2021. DOI: 10.48550/arXiv.2107.03374
- [4] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity Assessment of Neural Code Completion," in Proc. 6th ACM SIGPLAN Symposium on Machine Programming (MAPS), San Diego, CA, USA, 2022, pp. 21–29. DOI: 10.1145/3520312.3534864
- [5] J. Smith, R. Nguyen, and T. Patel, "AI-Integrated Programming Environments for Web Developers: Privacy, Performance and Practical Adoption," IEEE Software, vol. 40, no. 4, pp. 58–67, July 2023. DOI: 10.1109/MS.2023.3265891
- [6] E.Batu, M.Kaya, and S.Ozkan, "A Comparative Analysis of Cloud-Based Integrated Development Environments," International Journal of Computer Applications, vol. 182, no. 15, pp. 1–8, 2022. DOI: 10.5120/ijca2022922051
- [7] GitHub Inc., "GitHub Codespaces Documentation: Cloud Development Environments," GitHub Official Documentation, 2024. [Online]. Available: <https://docs.github.com/en/codespaces>
- [8] StackBlitz Inc., "Introducing WebContainers: Run Node.js Natively in Your Browser," StackBlitz Engineering Blog, May 2021. [Online]. Available: <https://blog.stackblitz.com/posts/introducing-webcontainers/>
- [9] Amazon Web Services, "AWS Cloud9 — Cloud IDE Documentation," AWS, 2024. [Online]. Available: <https://docs.aws.amazon.com/cloud9/>
- [10] D. Turner and A. Hall, "In-Browser Node.js Execution Using WebContainer: Architecture and Implications for Developer Tooling," in Proc. International Conference on Web Engineering (ICWE 2023), Alicante, Spain, 2023, pp. 112–126. DOI: 10.1007/978-3-031-34444-2\_9
- [11] J. Wang and Y. Chen, "A Review on Code Generation with LLMs: Application, Challenges and Evaluation," in Proc. IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA 2023), 2023, pp. 1–8. DOI: 10.1109/ICAICA58456.2023.10405891
- [12] E. A. Alomar, A. Mkaouer, M. Ouni, and A. Wohlin, "Automated Code Refactoring Using Large Language Models," IEEE Access, vol. 12, pp. 10120–10134, 2024. DOI: 10.1109/ACCESS.2024.3352271
- [13] E. Nascimento, I. Steinmacher, M. A. Gerosa, and C. Treude, "The Impact of LLM-Assistants on Software Developer Productivity: A Systematic Literature Review," arXiv preprint arXiv:2507.03156, 2025. DOI: 10.48550/arXiv.2507.03156
- [14] M. L. Siddiq, S. Majumder, M. R. Mim, S. Jajodia, and J. Santos, "Usage of Large Language Models for Code Generation Tasks: A Literature Review," SN Computer Science, vol. 6, article 112, 2025. DOI: 10.1007/s42979-024-03584-7
- [15] H.Zhou and R.Patel, "Ollama: Running LLMs Locally for Developers — Performance, Privacy, and Practical Considerations," Journal of Open Source Software, vol. 9, no. 98, pp. 1–8, 2024. DOI: 10.21105/joss.06547
- [16] W.Ahmad, S.Chakraborty, B. Ray, and K. W. Chang, "Unified Pre-Training for Program Understanding and Generation," in Proc. NAACL-HLT 2021, Online, 2021, pp. 2655–2668. DOI: 10.18653/v1/2021.naacl-main.211
- [17] X.Jiang, R.Liu, and T. Zhang, "Developer Attitudes Toward AICoding Assistants: Privacy, Trust, and Adoption Factors," in Proc. IEEE/ACM International Conference on Software Engineering (ICSE 2024), Lisbon, Portugal, 2024, pp. 387–398. DOI: 10.1145/3597503.3623324

- [18] Microsoft Corporation, “Monaco Editor — Architecture and APIDocumentation,” Microsoft GitHub, 2024. [Online]. Available: <https://github.com/microsoft/monaco-editor>
- [19] P. Ashok, R. Gorli, S. Parameswari et al., “Artificial Intelligence: Augmented Integrated Development Environments for Boosting Programmer Productivity,” IGI Global, Hershey, PA, India, 2025. DOI: 10.4018/978-1-6684-9999-3
- [20] J.Nielsen, Usability Engineering, San Francisco, CA, USA: Morgan Kaufmann Publishers, 1993. ISBN: 978-0-12-518406-9
- [21] J. Brooke, “SUS: A Quick and Dirty Usability Scale,” in Usability Evaluation in Industry, P. Jordan, B. Thomas, I. McClelland, and B. Weerdmeester, Eds. London: Taylor and Francis, 1996, pp. 189–194.
- [22] L. Zheng, C. Liang, Y. Huan et al., “MLC-LLM: Universal LLM Deployment for Everyone,” arXiv preprint arXiv:2404.14483, 2024. DOI: 10.48550/arXiv.2404.14483
- [23] N. Singh and Deepshikha, “A Review of Development of an AI-Based Code Completion Tool for Enhancing Developer Productivity and Efficiency,” IRJET, vol. 12, no. 03, pp. 112–119, March 2025.