© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715992

Design and Implementation of a MERN-Based AI Code Generation and Browser-Sandboxed Execution Platform Using Large Language Models

ARYAN GAWADE[1], TUSHAR CHAWRE[2], RUSHABH GAUR[3], OM DHANKE[4], DR. NIRANJAN KULKARNI[5], SWATI PATIL[6]

[1, 2, 3, 4] *Department of Computer Science and Design Engineering, New Horizon Institute of Technology and Management, Thane, India*

[5, 6] *Assistant Professor, Department of Computer Science and Design Engineering, New Horizon Institute of Technology and Management, Thane, India*

***Abstract*—The integration of Large Language Models (LLMs) into software development environments has significantly enhanced developer productivity through intelligent code generation. However, securely and efficiently executing AI-generated code remains a challenge due to potential security risks and infrastructure overhead. This paper presents the design and implementation of a MERN-based web application that integrates the Gemini LLM for multi-language code generation and utilizes browser-based Web Containers for secure sandboxed execution. The proposed system eliminates the need for local setup and server-side execution by leveraging client-side isolated runtime environments. Performance evaluation demonstrates reduced backend load and reliable multi-user execution.**

***Index Terms*—Large Language Models, AI Code Generation, MERN Stack, Web Containers, Sandboxed Execution, BrowserBased IDE**

I. INTRODUCTION

The rapid evolution of Large Language Models (LLMs) has significantly transformed modern software development practices. AI-assisted coding tools now enable developers to generate functional code snippets, debug errors, and accelerate development workflows. Platforms such as GitHub Copilot and Cursor have demonstrated the potential of integrating LLMs into development environments. However, these systems often depend on local execution environments or cloud-based infrastructure, introducing challenges related to setup complexity, scalability, cost, and security.

A key limitation in existing AI-assisted coding platforms is the safe execution of generated code. Executing arbitrary code on centralized servers introduces security vulnerabilities and increases infrastructure costs. Additionally, multi-user environments require strict isolation mechanisms to prevent cross-user interference and malicious exploitation.

To address these challenges, this paper presents the design and implementation of a MERN-based web application that integrates the Gemini Large Language Model for multi-language code generation and utilizes browser-based sandboxed execution through StackBlitz WebContainer. The proposed system enables users to generate, edit, and execute code directly within the browser without requiring local installation or server-side execution.

The platform employs a layered architecture consisting of a React-based frontend interface, a Node.js and Express backend API deployed on Vercel, Postgres-based user session management, and client-side Web Container sandboxing. Multi-user isolation is achieved through route-level separation, database-level user mapping, and execution log tracking.

The primary contributions of this paper are as follows:

1) Design of a browser-native AI code generation and execution platform.
2) Integration of Gemini LLM for multi-language code generation.
3) Secure sandboxed execution using Web Containers.
4) Multi-user isolation through database-backed routing mechanisms.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715992

5) Performance evaluation of AI-assisted browser-based development workflows.

The proposed system aims to democratize AI-powered development environments while ensuring security, scalability, and accessibility

.

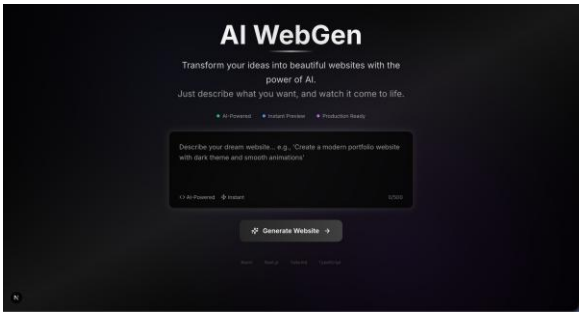

Figure 1: Ai WebGen Landing Page and Hero Section

II. RELATED WORK

The integration of Artificial Intelligence in software development has gained significant momentum in recent years. Large Language Models (LLMs) have enabled automated code generation, intelligent debugging, and contextual assistance within development environments.

One of the most widely adopted AI-assisted development tools is GitHub Copilot, which provides real-time code suggestions within integrated development environments (IDEs). While Copilot enhances productivity, it does not provide built-in execution environments and relies on the local machine configuration for runtime execution.

Similarly, Cursor integrates LLM-based assistance into a code editor interface, allowing conversational code modifications and project-level understanding. However, execution often depends on local or external runtime environments, introducing potential configuration and security challenges.

Browser-based development platforms such as Replit and CodeSandbox provide cloud-hosted execution environments that eliminate the need for local setup. These platforms support multi-language execution but typically rely on centralized server-side containers, which increase infrastructure cost and introduce potential attack surfaces.

More recently, StackBlitz introduced WebContainer technology, enabling Node.js runtime execution directly inside the browser using WebAssembly. This approach improves security by isolating execution within the browser sandbox and reduces backend infrastructure dependency.

However, existing platforms either focus primarily on AI-assisted editing without integrated sandboxed execution or provide execution environments without deep LLM-driven code generation capabilities. Furthermore, few systems combine multi-language AI code generation with fully browser-based secure execution and structured multi-user isolation within a unified MERN-based architecture.

The system proposed in this paper bridges this gap by integrating Gemini-based multi-language code generation, Monaco-based interactive editing, browser-native sandboxed execution using Web Containers, and database-backed user isolation within a scalable web architecture.

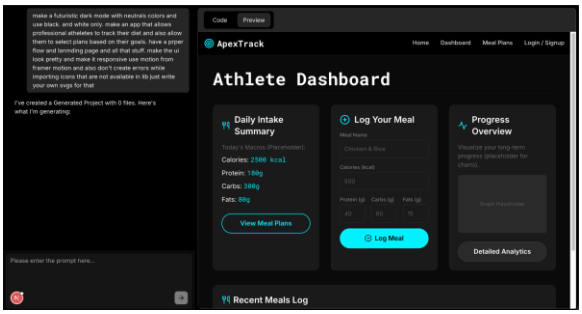

Figure 2: AI WebGen Building Website

III. PROPOSED SYSTEM

The proposed system follows a layered MERN-based architecture integrating AI-driven code generation with secure browser-based execution. The architecture is designed to ensure scalability, security, and multi-user isolation while minimizing server-side execution overhead.

A. Overall Architecture Overview

The system consists of four primary layers:

- Presentation Layer (Frontend).
- Application Layer (Backend API).
- AI Integration Layer
- Execution and Isolation Layer.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715992

The architectural workflow is illustrated as:

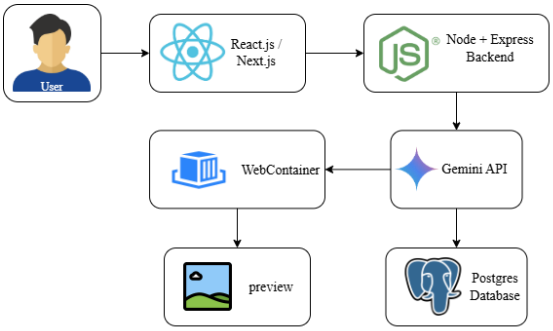

Figure 3: Overall System Architecture

B. Presentation Layer

- Select programming language (JavaScript, React, Next.js, Node.js, Python, Java).
- Generate code using AI prompts.
- Modify generated code.
- Execute code directly within the browser.
- View execution logs and outputs.

The frontend communicates securely with the backend via RESTful API routes.

C. Application Layer

The backend is built using Node.js and Express and is deployed on Vercel. The backend performs the following functions:

- Authentication and user validation.
- Prompt validation and formatting.
- Secure communication with Gemini API.
- Storing user projects in Postgres Database.
- Managing execution logs.
- Enforcing rate-limiting and request control.

API keys for the Gemini model are securely stored in environment variables on the server side, preventing exposure to clients.

D. AI Integration Layer

The system integrates the Gemini Large Language Model for multi-language code generation. The backend sends structured prompts to the Gemini API and receives syntactically formatted code responses. Prompt engineering techniques are used to:

- Ensure language-specific output formatting.
- Avoid unsafe code patterns.
- Reduce hallucinogenic dependencies.
- Maintain consistent code structure.

This layer enables dynamic generation of JavaScript, React, Next.js, Node.js, Python, and Java code

E. Execution and Isolation Layer

Code execution is performed entirely within the browser using StackBlitz WebContainer technology. Web Containers provide a sandboxed Node.js runtime implemented using WebAssembly, ensuring that:

- Code does not access the host operating system.
- Arbitrary system-level commands are not executed.
- File system access remains virtualized.
- Execution remains isolated per browser session.

Unlike traditional cloud-based execution platforms, this approach reduces server resource consumption and improves
Scalability.

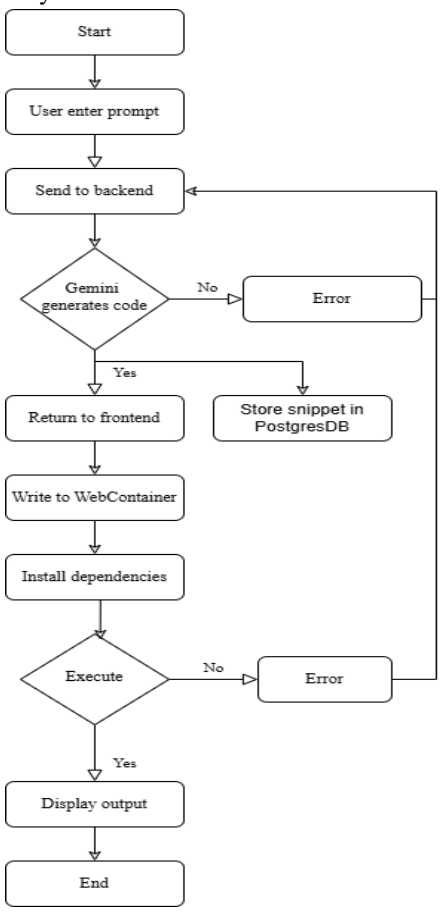

Figure 4: Code Generation and Execution Workflow

F. Multi-User Isolation Mechanism

Multi-user isolation is achieved through:

- Postgres Database-based user project mapping.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715992

- Route-level separation using Express middleware.
- Unique project identifiers.
- Observation of execution logs per user session.
- Authentication-based access control.

The codebase of each user and the execution logs are stored independently in the database, preventing data leakage between users.

G. Deployment Model

The backend is deployed using Vercel, enabling serverless scalability and automatic request handling. The frontend is delivered as a static web application, and execution is handled client-side, significantly reducing backend load.

IV. SECURITY AND ISOLATION MODEL

Security is a critical concern in AI-assisted code generation platforms due to the execution of dynamically generated code and exposure to external APIs. The proposed system incorporates multiple layers of security mechanisms to ensure safe multi-user operation.

A. Authentication and Access Control

The platform implements JSON Web Token (JWT)-based authentication to verify user identity and protect private resources. Upon successful login, a signed JWT is issued to the client and attached to subsequent API requests. Backend middleware validates the token before granting access to protected routes, ensuring that only authenticated users can generate or execute code.

B. Backend API Protection

The Gemini API integration is handled exclusively on the backend server. API keys are stored securely in environment variables and are never exposed to the client. This prevents misuse of the AI service and protects against key leakage.

Rate-limiting mechanisms are applied to API routes to restrict excessive request generation and mitigate denial-of-service (DoS) risks. This ensures fair resource allocation among concurrent users.

C. Browser-Based Sandboxed Execution

Unlike traditional cloud-based execution platforms, the proposed system executes code entirely within the browser using StackBlitz WebContainer. The WebContainer environment provides:

- Virtualized file system.
- Isolated Node.js runtime.
- No direct access to the host operating system.
- No execution of arbitrary system-level commands.

Since execution occurs within the browser sandbox, the backend server is not exposed to arbitrary code execution risks. This significantly reduces infrastructure attack surfaces.

D. Multi-User Data Isolation

User data and project files are stored in a Postgres Database with unique user identifiers. Express route-level middleware ensures that users can access only their own projects and execution logs. Database queries are filtered by authenticated user IDs, preventing cross-user data access.

Execution logs are stored separately per user session, enabling traceability while maintaining isolation.

E. Limitations

Although the platform incorporates authentication and rate limiting mechanisms, input sanitization at the prompt and code submission level is currently limited. Future improvements may include structured input validation and static code analysis before execution to further enhance security robustness.

V. IMPLEMENTATION DETAILS

This section describes the practical implementation of the proposed AI-assisted browser-based development platform, including backend design, database schema, AI integration pipeline, and execution workflow.

A. Technology Stack

The system is implemented using the MERN stack:

- Postgres Database for data storage.
- Express.js for backend API development.
- React for frontend user interface.
- Node.js for server-side execution.

The backend is deployed using Vercel, while browser-based execution is handled using StackBlitz WebContainer.

The Monaco Editor is integrated into the frontend to provide syntax highlighting, multi-language support, and an IDE-like coding experience.

B. Database Design

Postgres Database is used to store user-related information and code snippets. Instead of storing full project file systems, the platform stores:

- User ID.
- Selected programming language.
- AI prompt.
- Generated code snippet.
- Timestamp.
- Execution logs.

Each code snippet is mapped to a specific authenticated user through a unique identifier. This design ensures lightweight storage and simplified retrieval while maintaining multi-user isolation.

C. AI Code Generation Pipeline

The AI generation process follows these steps:

1) The user selects a programming language and enters a prompt.
2) The frontend sends a POST request to the backend API.
3) The Express backend validates the JWT token.
4) The backend constructs a structured prompt tailored to the selected language.
5) The prompt is sent securely to the Gemini API.
6) The generated code response is parsed and formatted.
7) The code snippet is stored in Postgres Database.
8) The response is returned to the frontend editor.

Prompt engineering strategies are applied to:

- Ensure language-specific syntax correctness.
- Avoid unsafe system commands.
- Maintain consistent output structure.

D. Execution Workflow

Execution is performed entirely inside the browser using Web Containers. The workflow is as follows:

1) The generated or edited code is written into a virtual file system inside the WebContainer.
2) If required, supporting files such as package.json are dynamically created.
3) Dependencies are installed within the isolated environment.
4) The runtime executes the code.
5) Final output is captured and displayed in the output console.

Since execution occurs in a sandboxed browser runtime, no server-side execution resources are consumed.

E. Multi-Language Support

The system supports:

- JavaScript
- React
- Next.js
- Node.js
- Python
- Java

Language selection dynamically modifies:

- Prompt formatting
- File structure initialization
- Runtime execution command

This flexible design allows the platform to accommodate different development paradigms within a unified interface.

F. Logging Mechanism

After execution, final output logs are captured and stored in Postgres Database along with the associated code snippet.

This enables:

- Execution traceability
- Debugging history
- User activity tracking

Real-time streaming is not implemented; instead, the system displays the final output after execution completes.

VI. PERFORMANCE EVALUATION

To evaluate the effectiveness of the proposed AI-assisted browser-based development platform, experimental testing was conducted under controlled

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715992

conditions. The evaluation focused on latency, execution performance, memory consumption, concurrency handling, and system reliability.

A. Experimental Setup

The backend services were deployed on Vercel. Code execution was performed entirely within the browser using StackBlitz WebContainer. Testing was conducted using standard desktop configurations with stable internet connectivity.

The system was evaluated using common programming tasks across supported languages, including JavaScript, Python, Java, and Node.js

B. Code Generation Latency

The average response time for AI-based code generation using the Gemini model ranged between:
20–30 seconds per request

The latency primarily depends on:

- Prompt complexity
- Output length
- External API response time
- Network conditions

While higher than simple autocomplete tools, the generated output typically consisted of complete functional code blocks rather than partial suggestions.
The average code generation latency (L_{avg}) is calculated as:

$$L_{avg} = \frac{1}{n}\sum_{i=1}^{n} L_i$$

where Li represents the latency of the i
th request and n represents the total number of test samples.

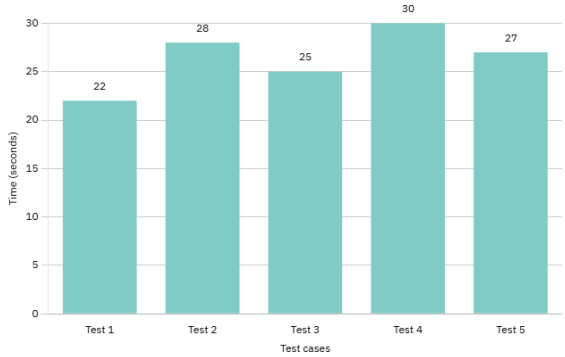

Figure 5: Code Generation Latency

C. Execution Performance

For small to medium-sized programs, the average execution time within the browser-based WebContainer environment ranged between:
40–50 seconds
Execution time includes:

- Virtual file system initialization
- Dependency installation (if required)
- Runtime execution
- Output capture

Since execution is client-side, performance may vary depending on the user's device specifications.

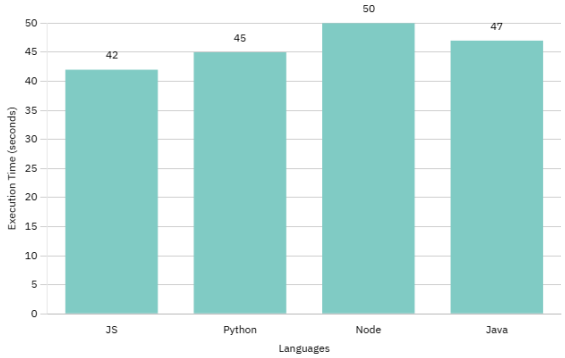

Figure 54: Execution Time Across Languages

D. Memory Utilization

During execution, browser memory usage ranged between:
200–300 MB
Memory consumption was influenced by:

- Language runtime
- Installed dependencies
- Size of generated code

Despite this overhead, the system remained stable, without browser crashes during testing.

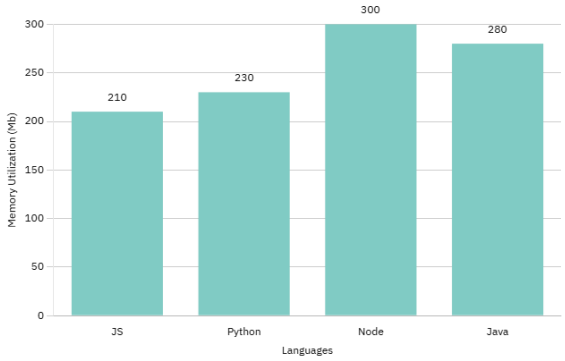

Figure 6: Memory Utilization

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715992

C. Concurrency Testing

The system was tested with up to:

4 concurrent active users Since execution occurs within individual browser sessions, server-side resource consumption remained minimal. The backend handled authentication and API requests without significant degradation in performance.

F. Reliability and Failure Rate

The observed failure rate during testing was approximately 0.5 Failures were primarily attributed to:

- External API timeout
- Network instability
- Dependency resolution errors

No critical system crashes or data leaks were observed during evaluation. The observed failure rate (F_r) is calculated as:

$$F_r = \frac{F}{T} \times 100$$

where F represents the number of failed executions, and T represents the total execution attempts.

G. Discussion

The evaluation demonstrates that browser-based sandboxed execution significantly reduces backend infrastructure load while maintaining secure isolation. Although AI response latency remains relatively high due to external model dependency, the platform provides a fully integrated code generation and execution workflow within a unified web environment.

VII. COMPARATIVE ANALYSIS

Table I compares the proposed system with existing AIassisted development platforms.

TABLE I
COMPARISON WITH EXISTING PLATFORMS

Feature	Copilot	Replit	Proposed
AI Code Generation	Yes	Limited	Yes
AI Code Generation	No	No	Yes
Server-Side Execution Required	Yes	Yes	No
Multi-Language Support	Yes	Yes	Yes

VIII. CONCLUSION

This paper presented the design and implementation of a MERN-based AI-assisted code generation and browsersandboxed execution platform. The system integrates Gemini Large Language Model capabilities with secure backend orchestration and client-side runtime isolation using StackBlitz WebContainer.

Unlike traditional cloud-dependent coding platforms, the proposed architecture enables execution entirely within the browser environment, significantly reducing backend infrastructure load while improving scalability and security. Multi-language support, including JavaScript, React, Next.js, Node.js, Python, and Java, allows the platform to serve diverse development needs within a unified interface.

The implementation demonstrates that AI-driven development environments can be securely deployed using a layered architecture consisting of React frontend, Express backend, Postgres Database-based data isolation, and serverless deployment on Vercel. JWT-based authentication and rate-limiting mechanisms enhance system robustness in multi-user scenarios.

Performance evaluation indicates acceptable response latency for AI-based generation (20–30 seconds) and stable execution performance within browser-based containers. The system achieved reliable operation with a minimal failure rate and controlled memory utilization.

Overall, the proposed platform demonstrates the feasibility of combining LLM-powered code generation with browsernative sandboxed execution to create secure, scalable, and accessible AI-assisted development environments.

FUTURE WORK

While the proposed system demonstrates functional and secure AI-assisted code generation and execution, several enhancements can further improve its performance and robustness.

Future improvements may include:

1) Real-Time Output Streaming Implementing streaming execution output to improve user interaction and debugging efficiency.
2) Input Sanitization and Static Code Analysis Introducing structured input validation and automated code scanning mechanisms to further strengthen security.
3) Fine-Tuned Language Models Utilizing fine-tuned or domain-specific LLMs to reduce response latency and improve code accuracy.
4) Collaborative Development Support Enabling real-time multi-user collaboration within shared projects.
5) Container Orchestration for Advanced Isolation Extending the architecture to support hybrid browsercloud container models for enterprise-grade scalability.
6) Offline LLM Integration Exploring local model integration to reduce external API dependency and latency.

These improvements may enhance both performance efficiency and system reliability, positioning the platform for broader adoption in educational and professional environments.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to our project guide, Dr. Niranjan T. Kulkarni, and our co-guide, Ms. Swati Patil, for their invaluable support and guidance throughout the course of this project. We also extend our heartfelt thanks to our Head of Department, Dr. Niranjan T. Kulkarni, and the management of New Horizon Institute of Technology and Management for providing us with the resources and encouragement necessary to bring this work to successful completion.

REFERENCES

[1] GitHub, “GitHub Copilot,” 2024. [Online]. Available: https://github.com/features/copilot

[2] Cursor AI, “Cursor – AI Code Editor,” 2024. [Online]. Available: https://cursor.sh

[3] Replit Inc., “Replit – Collaborative Browser IDE,” 2024. [Online]. Available: https://replit.com

[4] StackBlitz, “WebContainer Technology,” 2024. [Online]. Available: https://stackblitz.com

[5] Google DeepMind, “Gemini Large Language Models,” 2024. [Online]. Available:https://deepmind.google/research/publications/64816/

[6] J. R. Lorch and A. J. Smith, “The V8 JavaScript Engine,” IEEE Internet Computing, 2010.

[7] A. Haas et al., “Bringing the Web up to Speed with WebAssembly,” ACM SIGPLAN, 2017.