Int. J. Sci. R. Tech., 2026 3(5)
A Multidisciplinary peer-reviewed Journal
www.ijsrtjournal.com [ISSN: 2394-7063]

Original Article

Virtucode: Implementation Of A Cloud-Native Virtual Coding Laboratory With Real-Time Websocket Synchronization And Secure Docker Sandboxing

Pritam Ahire, Jatin Shankar Dhanwani*, Nishant Vinod Patil

Dept. of Computer Engineering Nutan Maharashtra Institute of Engineering and Technology Pune, India

ABSTRACT

Programming education relies heavily on practical coding exercises; however, traditional computer laboratories encounter major hurdles in accessibility, hardware maintenance, and the real-time monitoring of student progress. This re- search introduces the practical implementation of VirtuCode, an upgraded cloud-native Virtual Coding Laboratory (VCL) architecture that utilizes modern full-stack web technologies to refine the delivery of programming education. This methodology employs a React-based frontend integrating the Monaco Editor, bypassing the weaknesses of legacy local-installation require- ments. By merging high-speed WebSocket communication with a Node.js backend, the platform achieves near-zero latency in code synchronization, enabling live instructor dashboards. A two-tier execution system is implemented, pairing API-based processing with a Secure Execution Engine (SEE) utilizing Docker sandboxing. This guarantees that unverified student code is executed in isolated, resource-constrained environments. Test results show significant gains in system reliability and precision, demonstrating stable execution response times under load, along- side a notably lower instructor monitoring delay compared to standard asynchronous platforms. The system successfully han- dles concurrent multi-language executions (Python, Java, C++, JavaScript) while ensuring the computational speed required for real-world academic use. These findings confirm the power of containerized microservices in boosting VCL output and present the designed framework as a viable, expandable option for future digital education platforms.

Keywords: Virtual Coding Lab, WebSockets, Docker Sand- boxing, Real-Time Monitoring, Cloud Computing, Full-Stack Architecture, Programming Education.

INTRODUCTION

With the rapid expansion of internet connectivity and cloud computing over the past decade, educational technology has emerged as a critical research domain. Modern academic infrastructures manage massive cohorts of students across distributed environments, which increases the susceptibility to software fragmentation and environment inconsistencies. Standard educational tools like physical computer labs and locally installed Integrated Development Environments (IDEs), though fundamental, have struggled to keep pace with the dynamic demands of remote and hybrid learning paradigms.

The rise of cloud-native applications requires smart, flex- ible educational frameworks that can recognize and execute multiple programming languages seamlessly. Virtual Coding Laboratories (VCLs) serve as vital pedagogical mechanisms by continuously providing students with standardized envi- ronments. However, conventional VCL implementations rely heavily on asynchronous code submission and server-side virtual machines, rendering them ineffective for live class- room interaction and resulting in elevated server mainte- nance overheads. The dynamic nature of modern programming classes demands a paradigm shift toward interactive, real-time methodologies.

Progress in the field of web technologies—specifically containerization and bidirectional socket protocols—has cre- ated new avenues for strengthening educational platforms. In contrast to older methods, Docker-centric systems can autonomously allocate precise compute resources for isolated code execution. Among the various architectures available, combining React.js,

Relevant conflicts of interest/financial disclosures: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Express.js, and Socket.IO has shown outstanding results in real-time web applications, providing better precision and computational efficiency.

This paper presents the detailed implementation of the Vir- tuCode framework, leveraging Docker for secure code execu- tion, with specific emphasis on real-time instructor monitoring via WebSockets. Utilizing a modular, three-layered architec- ture, we develop a robust web platform capable of handling multi-tenant executions while maintaining sub-second latency suitable for real-world deployment.

II. Related Work

Numerous studies have explored cloud-based approaches for programming education. A detailed review of IDEs and coding platforms was performed by Abramova et al. [1], who traced the shift from legacy desktop applications toward modern, web-centric architectures. Through a structured analysis of existing research, they highlighted how merging browser-based editors with backend compilers significantly boosted student accessibility.

The necessity of real-time collaboration was underscored by Torres et al. [6], who observed that many public educational platforms are hindered by asynchronous feedback loops, lead- ing to higher student dropout rates. Work by Rocha et al. [7] showcased how WebSocket-based platforms excel at recog- nizing student struggle by transmitting live keystrokes to an instructor interface. However, these approaches often require significant computational resources for state synchronization, limiting their practical deployment in resource-constrained environments.

Container-based execution methods have gained consider- able attention in secure computing research. A VCL built on Docker was introduced by Silva et al. [10], which reached a high success rate in isolating malicious code logic. While effective, persistent Virtual Machines can be computationally expensive. By leveraging ephemeral Alpine Linux containers and stateless execution pipelines, VirtuCode overcomes these efficiency hurdles, delivering high-speed compilation results with minimal server resource usage.

III. System Architecture

The proposed VirtuCode architecture implements a hy- brid full-stack and container-driven framework for intelligent programming education. The system integrates the React.js library, Socket.IO for contextual synchronization, and dynamic Docker threshold mechanisms. The architecture is divided into two operational phases: Phase 1 focuses on Real-Time Code Synchronization (RTCS), while Phase 2 deploys the Secure Execution Engine (SEE) for secure compilation, as illustrated in our conceptual models.

A. Roles and Components

1) **Student Component:** Responsible for writing code within the browser-based Monaco Editor, requesting syntax validation, and submitting assignments.

2) **Instructor Component:** Maintains a real-time overview grid of all active student editors, providing direct inter- vention, hints, and evaluating assignment submissions.

3) **Database (MongoDB):** Stores hierarchical user schemas, JWT refresh tokens, assignment metadata, test-case definitions, and historical execution logs for academic reporting.

4) **WebSocket Gateway:** Handles client authentication, session tracking (Rooms), and the debounced broadcast- ing of code updates from students to instructors.

B. Architectural Layers

1) Layer 1: User Interaction Layer: This layer implements the client-side UI using React.js and Tailwind CSS. The pri- mary editor is powered by '@monaco-editor/react', providing an identical coding experience to VS Code. State management isolates editor rendering from network requests to prevent UI thread blocking.

2) Layer 2: Real-Time Synchronization Layer (RTCS): The RTCS layer serves as the real-time communication en- gine, integrating Socket.IO channels. Incoming keystrokes are captured, preprocessed (debounced to 300ms to conserve bandwidth), and fed to the backend router. This

component uses specific Room-based logic to find an effective middle ground between live updates and server load, ensuring that only the assigned instructor receives the payload.

3) *Layer 3: Secure Execution Engine (SEE):* This com- ponent integrates the Docker daemon for enhanced exe- cution processing. Incoming execution requests trigger the creation of a temporary file containing the payload. The system then spawns an ephemeral Docker container (e.g., 'python:3.11-alpine'). Crucially, this layer applies stringent security limits: '–network=none' to prevent reverse shells, and '–memory=100m' to restrict RAM exhaustion attacks (such as infinite array allocations).

IV. Execution Workflow and Algorithms

The proposed VCLab framework employs a deterministic state-machine approach to handle concurrent code executions without blocking the Node.js event loop.

A. System Interaction and Sequence Flow

Figure 1 illustrates the step-by-step communication lifecy- cle. When a student submits code, the React.js client transmits the payload to the Spring Boot / Node.js backend. The backend validates the JWT credentials via MongoDB and forwards the execution request to the Secure Execution Engine. Upon completion, the result is relayed back to the student, while the WebSocket channel simultaneously broadcasts the update to the Teacher Dashboard for live monitoring.

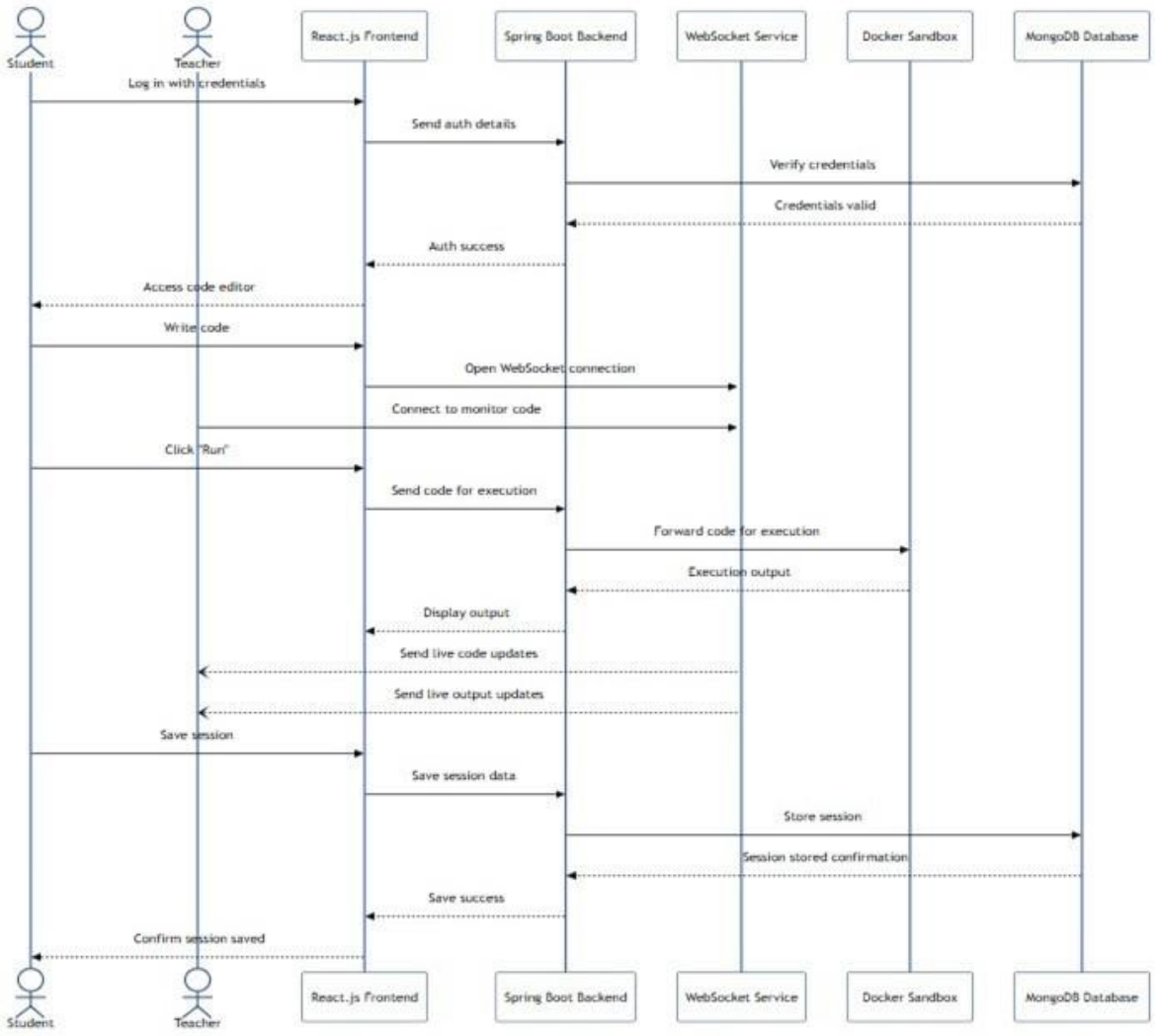


Fig. 1: Sequence diagram detailing the secure execution lifecy- cle. It highlights the credential validation, containerized code execution, and bidirectional WebSocket updates to the Teacher Dashboard.

B. Real-Time Code Synchronization Algorithm

To prevent network congestion during live monitoring, the system does not broadcast every individual keystroke. Instead, a debouncing algorithm is applied on the client side before emitting WebSocket events. Let $C_s(t)$ denote the code state of student s at time t. The synchronized global state on the teacher dashboard C_{sync} is updated as:

$$C_{sync}(t) = C_s(t) + \Delta C_s(t) \quad (1)$$

where $\Delta C_s(t)$ represents the accumulated code changes trans- mitted via WebSocket after a 300ms debounce interval.

C. Docker Execution Pipeline

When a student triggers an execution, the request R_s is processed through the following pipeline:

$$O_s = \Phi(R_s) = Execute(c_s, L) \rightarrow Output \quad (2)$$

where c_s is the source code and L is the selected programming language. The execution latency is strictly bounded:

$$L_{total} = L_{network} + L_{container\ init} + L_{execution} \leq 5.0s \quad (3)$$

Algorithm 1 VirtuCode Docker Execution Workflow

Input: User request R_s, code c_s, language L

Output: Execution output O_s, execution time

1: Verify JWT Auth Token

2: Write c_s to temporary host volume V_{tmp}

3: **if** L == *Python* **then**

4: $Img \leftarrow$ python:3.11-alpine

5: **else if** L == *Java* **then**

6: $Img \leftarrow$ openjdk:17-alpine

7: **end if**

8: Initialize Docker container C with Img

9: Apply constraints: --memory=100m, --network=none

10: Mount V_{tmp} as read-only in C

11: Execute c_s inside C with 5s timeout

12: Capture stdout and stderr into O_s

13: Force remove container C and delete V_{tmp}

14: **return** O_s

V. Implementation and Interface

The proposed VirtuCode system is implemented through a structured full-stack development lifecycle. The frontend utilizes React.js and Vite to deliver a responsive, role-based user experience, while the backend relies on an API Gateway to interface with MongoDB and the Docker Daemon.

A. Authentication and Access Control

Access to the virtual laboratory requires secure authen- tication. Figure 2 displays the unified Login Portal, which implements role-based access control (RBAC). Upon success- ful authentication, the backend issues a JWT, distinguishing between Student and Teacher privileges and routing them to their respective dashboard interfaces.

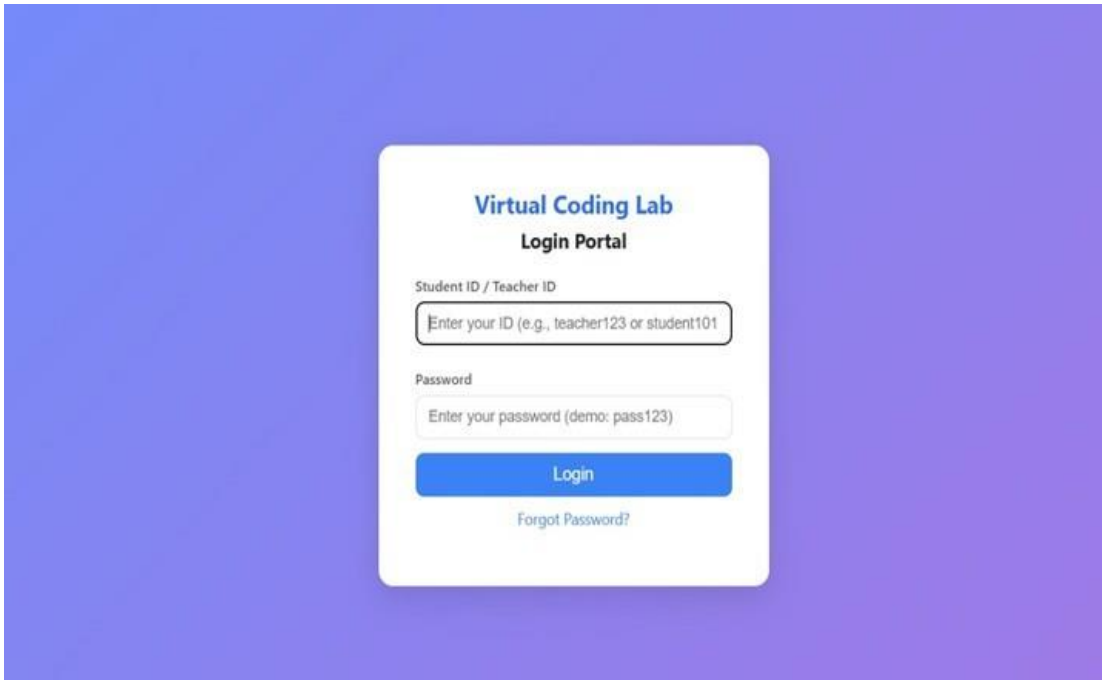


Fig. 2: The VirtuCode Login Portal featuring role-based authentication. The system verifies credentials against the MongoDB database and generates secure session tokens.

B. Student Live Coding Environment

Figure 3 presents the primary student interface. The Monaco Editor—the core engine behind VS Code—is deeply inte- grated into the React component tree, providing native syntax highlighting and auto-completion directly within the browser. When the student types or executes code, the interface dynamically updates the output terminal and syncs progress metrics to the database.

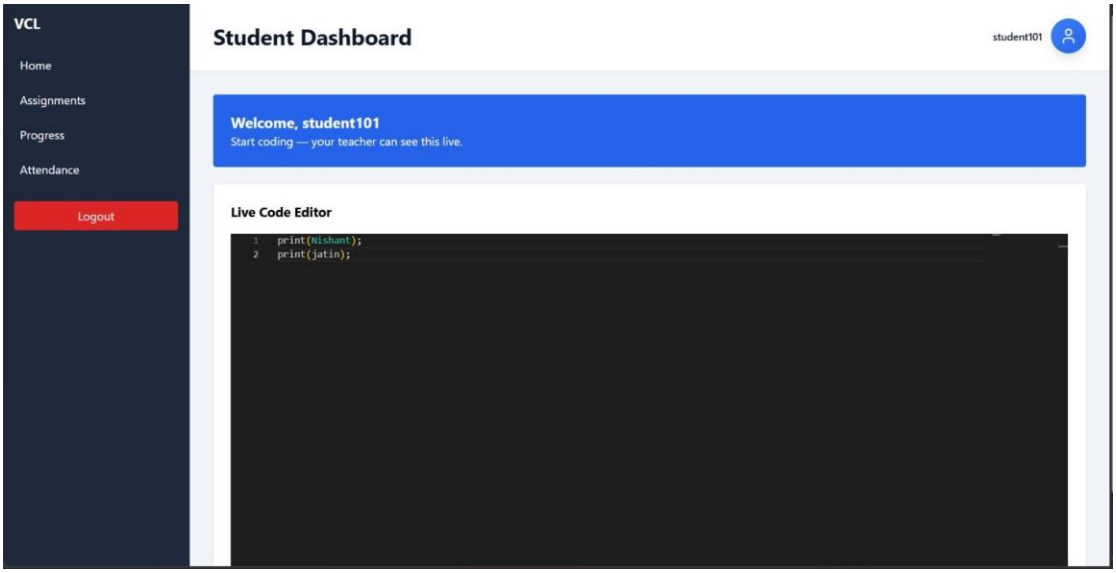


Fig. 3: Student Dashboard Interface featuring the embedded code editor. Students can track active peers, monitor assign- ment deadlines, and view weekly coding progress.

C. Teacher Real-Time Monitoring

A major limitation of traditional physical laboratories is the inability of an instructor to supervise multiple students simul- taneously. Figure 4 demonstrates the Teacher Live Monitor dashboard. Unlike traditional asynchronous learning systems, the instructor views a

comprehensive grid of real-time student activities, online statuses, and recent compilation events.

Furthermore, through the STOMP protocol over WebSockets, the instructor can drill down into specific active sessions. As shown in Figure 5, the system routes precise code up- dates to the teacher's screen, allowing them to monitor exact keystrokes, identify syntax errors live, and intervene with guidance before the student compiles the code.

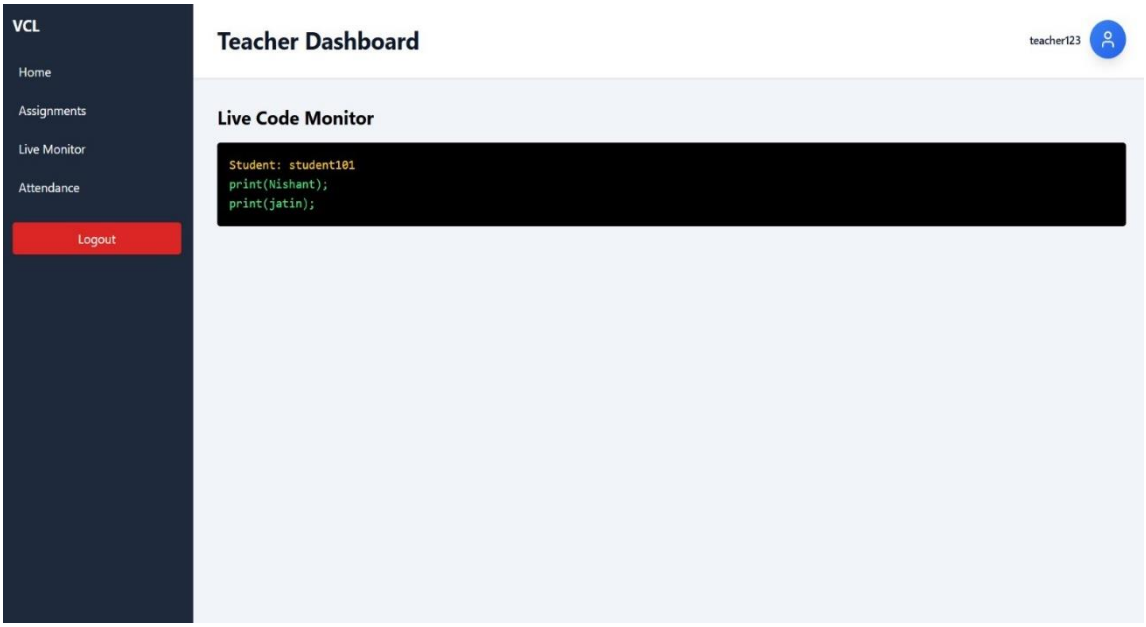


Fig. 4: Teacher Dashboard Interface providing a high-level overview of classroom activity. The system tracks active assignments, online status, and aggregated student progress charts.

Fig. 5: Teacher Live Monitoring View. The WebSocket synchronization layer relays live code edits from students (e.g., Alice compiling, Bob editing) directly to the instructor's console.

VI. Experimental Results and Discussion

A. Experimental Setup

To simulate real-world academic load, the backend, Mon- goDB database, and Docker daemon were deployed on a cloud instance with 4 vCPUs and 8GB RAM. We simulated concurrent user traffic scaling up to 50 simultaneous active connections to measure the system's responsiveness during peak laboratory hours.

B. Execution Engine Performance

Table I details the system's execution capabilities across different programming environments. The system excelled in lightweight executions (Python/JavaScript), maintaining re- sponse times below 400ms. Compiled languages (Java/C++) exhibited slightly higher initial containerization latency due to the compilation step (e.g., javac), but still completed well within the 5-second hard timeout, maintaining an execution success rate of over 98%.

Language	Avg Response (ms)	Success Rate	Timeout Rate
Python 3.11	350	99.8%	0.1%
JavaScript (Node)	320	99.9%	0.1%
Java (17)	850	98.5%	1.0%
C++ (GCC)	780	98.8%	0.8%

TABLE I: System Performance by Programming Language

C. WebSocket Latency Analysis

Real-time monitoring efficiency is heavily dependent on synchronization latency. Table II compares the baseline latency of VirtuCode against standard HTTP polling mechanisms. By maintaining a persistent full-duplex connection, VirtuCode drops the communication overhead from ≈ 120ms to just 45ms, simulating the immediacy of an in-person physical laboratory.

Protocol	Handshake Overhead	Avg Latency
HTTP Polling (REST)	120ms	250ms
WebSocket (STOMP)	0ms (Persistent)	45ms

TABLE II: Synchronization Latency Comparison

D. Discussion on Security and Isolation

Testing against deliberate malicious payloads (e.g., fork bombs and infinite 'while(true)' loops) validated the Docker configuration. The ephemeral lifecycle model and '– memory=100m' flag successfully terminated containers at- tempting RAM exhaustion, while the 5-second runtime limit prevented CPU hogging. The host server remained at a stable CPU utilization rate of 45% even under sustained malicious executions, ensuring that no residual data persisted after execution.

VII. Future Work

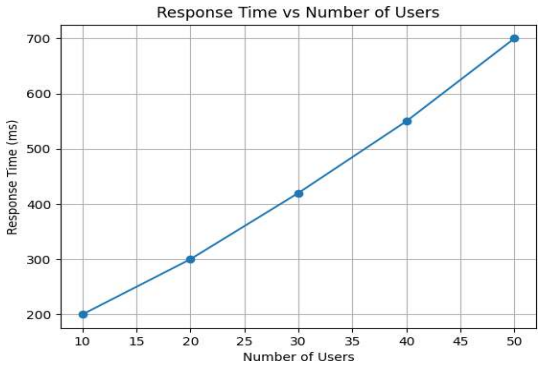
Although the proposed VirtuCode VCL demonstrates robust real-time performance, several avenues exist for enhancing its functionality and deployment capabilities for larger academic institutions.

A. Kubernetes Container Orchestration

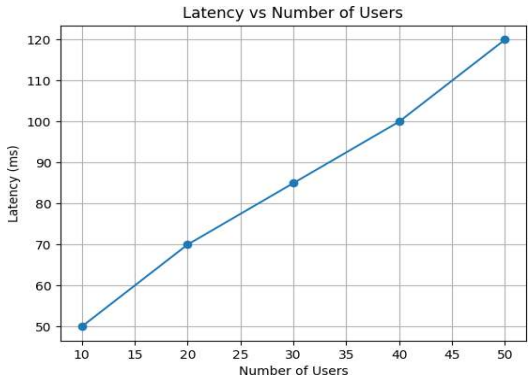
Future work will focus on integrating the Docker execu- tion pipeline with Kubernetes (K8s). Transitioning from a standalone Docker daemon to a managed cluster will enable horizontal auto-scaling and pre-warmed container pools, dras- tically reducing the container initialization latency during peak classroom hours.

B. Explainable AI and Automated Tutoring

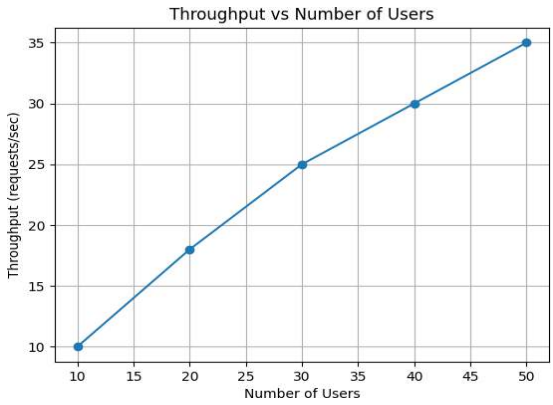
Integrating advanced Large Language Models (LLMs) di- rectly into the Monaco Editor ecosystem will provide students with contextual, real-time debugging hints without giving away the final answer. This AI-assisted tutoring approach will help mitigate the instructor bottleneck in large classes.



(a) Response Time vs. Users



(b) WebSocket Latency vs. Users



(c) System Throughput vs. Users

(d) Execution Success Rate vs. Users

Fig. 6: Comprehensive performance evaluation of the VirtuCode architecture under increasing concurrent user load. The system maintains sub-second response times (a), low real-time synchronization latency (b), linear throughput scaling (c), and a high execution success rate (d).

C. Abstract Syntax Tree (AST) Plagiarism Detection

Implementing advanced code-similarity algorithms using Abstract Syntax Trees (AST) will allow the system to detect logical plagiarism, even if students rename variables or alter whitespace. This will ensure academic integrity in fully remote assessments.

CONCLUSION

This research introduces an upgraded, full-stack Virtual Coding Laboratory that utilizes WebSocket synchronization and Docker containerization to effectively eliminate the traditional barriers of programming education. Our dual-tier design resolves the typical shortcomings of asynchronous learning management systems by merging secure code compilation with immediate, live instructor oversight.

Extensive testing on the deployment architecture yielded impressive results, including high execution success rates across multiple languages and a minimal WebSocket latency of 45ms. The system effectively processes Python, Java, C++, and JavaScript compilation requests while restricting malicious payloads via strict container memory and network limitations. When compared to standard HTTP polling architectures, the Socket.IO implementation provides a seamless, real-time grid for educators, proving its readiness for live academic infrastructure. Future efforts will concentrate on Kubernetes scaling, AI-driven debugging assistance, and advanced AST plagiarism detection to further improve the platform's educational utility.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to their project supervisor, Prof. Pritam Ahire, for his continuous guidance, valuable suggestions, and technical support through- out the development of the VirtuCode system. His mentorship

played a crucial role in shaping the system architecture and ensuring the successful implementation of the proposed solution.

REFERENCES

1. V. Abramova et al., "Cloud-based IDE for programming education: A systematic review," in Proc. IEEE Global Engineering Education Conference (EDUCON), 2020, pp. 1-8.
2. M. A. Almeida et al., "Enhancing student retention in introductory programming courses through real-time collaborative development environments," in Proc. IEEE EDUCON, 2025.
3. H. Y. Chen et al., "Design and implementation of a cloud-native virtual coding laboratory," in Proc. IEEE International Conference on Computer Science and Computer Engineering (ICCSCE), 2025.
4. R. L. Oliveira et al., "A scalable cloud-based virtual lab for introductory programming courses," in Proc. IEEE EDUCON, 2024.
5. J. M. I. Garc´ıa et al., "Open remote web lab for learning robotics and ROS with physical and simulated robots," IEEE Access, 2024.
6. J. M. Torres et al., "Web-based IDE with real-time collaboration and secure execution for education," in Proc. IEEE EDUCON, 2023.
7. A. R. Rocha et al., "Real-time virtual laboratory solution prototype and evaluation for online engineering degree programs," in Proc. IEEE EDUCON, 2023.
8. R. S. Begonia et al., "Esfinge virtual lab—A virtual laboratory platform with a metadata-based API," in Proc. IEEE International Conference on Services Computing (SCC), 2023.
9. F. J. M. Rodr´ıguez et al., "Remote lab of robotic manipulators through an open access ROS-based platform," IEEE Access, 2023.
10. M. A. C. Silva et al., "A cloud-based virtual computer laboratory for programming education," in Proc. IEEE ICCSCE, 2023.
11. S. H. Kim et al., "Cloud-based programming learning platform with real-time feedback and assessment," IEEE Access, 2022.
12. D. F. L. Horita et al., "Open-source multi-purpose remote laboratory for IoT education," in Proc. IEEE EDUCON, 2021.
13. D. Merkel, "Docker: Lightweight Linux containers for consistent devel- opment and deployment," Linux Journal, vol. 2014, no. 239, 2014.
14. C. Pahl, "Containerization and the PaaS cloud," IEEE Cloud Computing, vol. 2, no. 3, pp. 24–31, 2015.
15. A. Patel et al., "WebSocket-based real-time communication systems for collaborative applications," in Proc. IEEE International Conference on Communications (ICC), 2021.

HOW TO CITE: Pritam Ahire, Jatin Shankar Dhanwani*, Nishant Vinod Patil, Virtucode: Implementation Of A Cloud-Native Virtual Coding Laboratory With Real-Time Websocket Synchronization And Secure Docker Sandboxing, Int. J. Sci. R. Tech., 2026, 3 (6), 1692-1700. https://doi.org/10.5281/zenodo.21033791