International Journal of Innovative Research in Engineering
Volume 7, Issue 2 (March - April 2026), PP: 235-238.
https://www.doi.org/10.59256/ijire.20260702028
www.theijire.com
ISSN No: 2582-8746

Cloud-Based Serverless Remote Code Execution System Using AWS Lambda and Docker

Yuvika Singh[1], Dr. Ratnesh Mishra[2]
[1,2]Department of Computer Science and Engineering, Birla Institute of Technology, Patna, Bihar, India.

How to cite this paper:
Yuvika Singh[1], Dr. Ratnesh Mishra[2]" Cloud-Based Serverless Remote Code Execution System Using AWS Lambda and Docker", IJIRE-V7I2-235-238.

Abstract: *Remote Code Execution (RCE) systems are an important factor in the current practice of programming education, online examinations, and competitive coding. Traditional RCE architectures, however, experience scalability limitations, large operational costs and security risks as a result of running untrusted code via persistent servers or virtual machines. This paper presents a cloud-native and serverless Remote Code Execution system constructed on the Amazon API Gateway, AWS Lambda, and Docker container images. The suggested system executes Python, Java, C++ in isolated and short-lived containers to guarantee security and statelessness. A React.js frontend powered by the Monaco Editor provides an interactive IDE-like experience with examination-oriented restrictions, including copy-prevent and paste-prevent features. Experimental analysis proves reliable execution, high scalability, low cold-start latency, and sandbox isolation. The findings confirm that serverless solutions are an economical, secure, and scalable substitute of traditional RCE application.*

Keywords: *Serverless Computing, Online Judge Systems, AWS Lambda, Docker, Remote Code Execution, Cloud Computing.*

I.INTRODUCTION

Remote Code Execution (RCE) systems are used to support the structure of current online programming systems, competitive coding systems, and automated online assessment systems used in academic institutions. Such systems enable the user to provide source code and run it in a remote, controlled environment without the need of local setup. With the proliferation of cloud-based education services and large-scale coding systems, scalability, security and lower-cost execution infrastructures have become increasingly demanded.

Conventional RCE deployment systems were based on unchanging server systems or virtual machines to segregate user programs. Virtualization technologies offer hardware isolation, but with more memory usage and longer startup latency and high cost of operation [6]. The long-standing server-based designs also result in poor resource utilization, especially when there is no workload, as the infrastructure is active even when idle.

With the advent of containerization technologies, specifically Docker, a new way of lightweight virtualization emerged. Docker containers offer isolation similar to operating systems but with increased speed of starting up and reduced memory overhead compared to virtual machines [4], [5]. Serverless computing further removes infrastructure management by enabling developers to create stateless functions triggered by events [1], [2]. Research indicates that serverless systems offer fine-resource allocation and pay-per-execution cost models, making them well-suited for stateless workloads [3].

Despite the popular deployment of container-based RCE systems, a significant portion continues to use managed orchestration frameworks or persistent server clusters. The present paper introduces a cloud-native RCE architecture that integrates Docker container images with AWS Lambda to offer stateless execution, auto-scaling, and high-level isolation.

The key contributions of this work are:

1. Design and development of a serverless RCE based on Docker-based runtime environments on AWS Lambda.
2. Multi-language execution support for Python, Java and C++.
3. Implementation of sandboxed execution with container isolation and resource restrictions.
4. Experimental assessment that proves scalability, reliability and controlled execution behaviour.

II.LITERATURE REVIEW

Secure and scalable RCE systems have been developed in the context of virtualization, containerization, and cloud computing models. Early execution environments were dominated by virtual machines to separate user-submitted programs. Virtualization technologies offer hardware isolation, but with more memory usage and longer startup time compared to lightweight alternatives [6].

With the development of cloud platforms, containerization turned out to be a more effective tool for application deployment and isolation [4]. Docker greatly enhanced portability and deployment consistency by wrapping applications with their dependencies [4]. Container-based environments have lower overheads and can be started quickly compared to

traditional virtualization, making them appropriate for tasks with short lifespan requirements such as remote code execution [5]. Nevertheless, correct security configuration is necessary to ensure no privilege escalation and unauthorized access [15].

Serverless computing removes infrastructure management further by enabling developers to create stateless functions triggered by events [1]. Baldini et al. found that serverless computing ensures simpler scaling and less operational complexity [2]. Empirical studies of serverless services indicate horizontal scaling by default and cost efficiency, especially for intermittent workloads [3]. These features align well with RCE systems wherein every execution submission is an independent unit. Security is a key concern when executing untrusted workloads; research stresses the significance of powerful sandboxing primitives such as namespace isolation and restricted system-call access to address potential vulnerabilities [14], [15].

Based on the literature reviewed, containerization and serverless computing can serve as a powerful foundation of a scalable and secure execution environment. One research gap identified is the combination of Docker container images into a completely serverless architecture specifically aimed at multi-language remote code execution. The proposed system fills this void by integrating lightweight container isolation with event-based, auto-scaling serverless infrastructure.

III. METHODOLOGY

A. System Architecture

The proposed Remote Code Execution system adheres to a three-tier cloud-native design, comprising a presentation layer, API layer, and execution layer. The presentation layer is built on React.js with the Monaco Editor, providing an interactive browser-based interface with syntax highlighting, multi-language selection, and output visualization. Amazon API Gateway manages the API layer, serving as the secure entry point for all execution requests. The gateway authenticates incoming HTTP requests, applies throttling policies, and forwards valid requests to the execution layer via HTTPS and structured JSON messages.

The execution layer is implemented using AWS Lambda with a Docker container image setup. The container image includes Python, Java, and C++ runtime environments and compilers. Whenever the Lambda function is invoked, it generates an isolated execution environment that handles one request and is automatically destroyed after completion, guaranteeing high isolation between submissions.

Figure 1: Overall Architecture of the Proposed Serverless RCE System

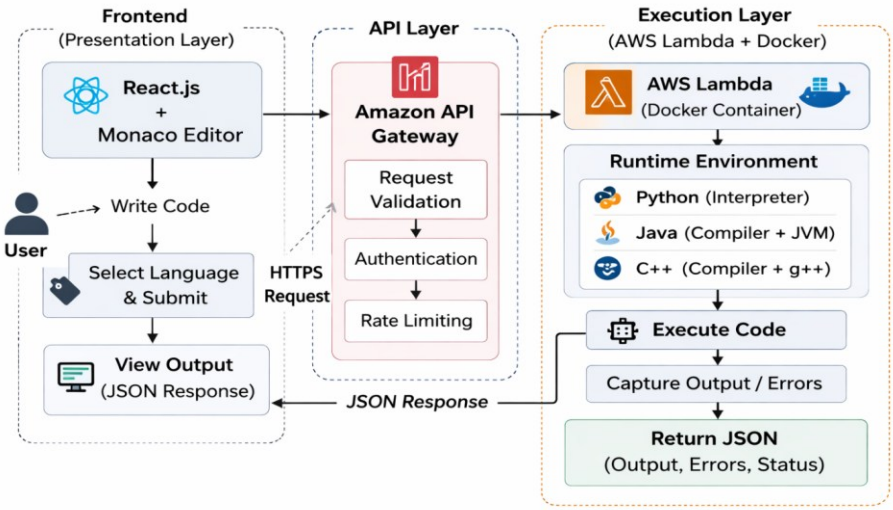

Fig. 1: System Architecture Diagram

B. Execution Workflow

When a user enters code using the web interface, the frontend captures the source code, the selected language identifier, and optional input parameters into a structured JSON object. This payload is transmitted to the API Gateway using HTTPS. After receiving the request, the API Gateway validates the structure and forwards it to the appropriate AWS Lambda function, where each request is executed in a new Lambda execution environment.

The runtime extracts the submitted code and temporarily writes it to the container's ephemeral file system. The system launches the appropriate execution pipeline based on the chosen programming language: Python source is run directly through the interpreter; Java source is compiled using javac and then executed by the JVM; and C++ source is compiled using g++ to generate a binary executable. Standard output, compilation errors, and runtime exceptions are all captured and returned as a structured JSON response through the API Gateway to the frontend. The container instance is terminated upon completion with no residual state.

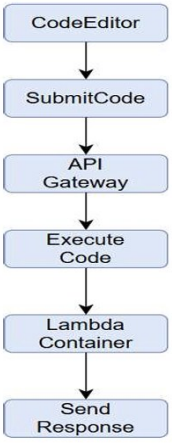

Fig. 2: Execution Workflow Diagram

C. Language-Specific Execution Pipeline

Python Execution: The submitted code is saved to a temporary file and executed by the Python interpreter within the Lambda container. Output streams are programmatically captured and execution is bounded by configured time limits.

Java Execution: The program is stored as Main.java and compiled using javac. Upon successful compilation, the generated bytecode is executed on the Java Virtual Machine. Compilation errors and runtime errors are handled separately to provide precise feedback.

C++ Execution: The code is saved as a .cpp file and compiled using the GNU g++ compiler. Upon successful compilation, the binary is executed under imposed memory and time limits. All language executions occur in isolated invocation contexts within the same Docker-based Lambda environment.

D. Security Model

Security is a central design principle of the system. Each execution request is handled in a new transient Lambda container providing OS-level isolation and preventing cross-user interference. Stateless execution is ensured by destroying temporary file storage upon completion. Resource limits such as memory caps and execution time limits are enforced at the Lambda configuration level. Access to external networks is restricted to prevent unauthorized communication. The frontend also enforces clipboard restrictions to minimize academic dishonesty in examination settings. This combination of server isolation and container sandboxing results in a significantly smaller attack surface compared to persistent server designs.

IV. EXPERIMENTAL RESULTS

Table 1. Performance Comparison of Language Execution in the Serverless RCE System

Language	Average Execution Time	Compilation Time	Cold Start Latency
Python	~120 ms	–	~480 ms
Java	~300 ms	~120 ms	~510 ms
C++	~280 ms	~150 ms	~500 ms

Table 1. Performance Comparison of Language Execution in the Serverless RCE System

Fig. 3: Experimental Results

The system was tested under controlled conditions to evaluate performance, reliability, and scalability. The backend was deployed on AWS Lambda using a Docker image containing the required compilers and interpreters. Test programs were executed in Python, Java, and C++, covering arithmetic operations, loops, recursive functions, syntax bugs, runtime exceptions, and computationally intensive programs. Execution time, compilation overhead, and response latency were observed.

Python code demonstrated low execution time due to the absence of a compilation step. Java and C++ executions carried additional compilation overhead but remained within acceptable bounds. Cold-start latency was observed on initial invocations but was significantly lower in subsequent warm executions. Parallel submission tests confirmed that the serverless architecture automatically scaled to handle multiple concurrent requests without performance degradation.

V. DISCUSSION

The experimental findings confirm the practicability of adopting a Remote Code Execution system built on a completely serverless design. The proposed model eliminates persistent infrastructure and the inefficient resource utilization characteristic of traditional virtual machine-based systems.

AWS Lambda's automatic scaling functionality is responsive to high load variations at any point in the load curve. Security analysis demonstrates that ephemeral container instantiation reduces cross-request data leakage and lowers exposure risk over time. The structured JSON response mechanism enhances frontend usability by clearly differentiating between compilation and runtime errors.

Some restrictions remain. Cold-start latency can influence response time for sporadic workloads. Additionally, multi-file project execution is not currently supported. These aspects represent opportunities for further improvement.

VI. FUTURE SCOPE

Even though the proposed serverless RCE system demonstrates security, scalability, and cost-effectiveness for multi-language execution, several improvements can further enhance its functionality and applicability. Among the extensions are support for multi-file project execution. Subsequent versions can support project-based execution by allowing compressed source bundles to be uploaded, unzipped, and compiled inside the Lambda container, enabling Java projects with multiple classes, C++ programs with header files, and Python modules spread across directories.

Another improvement is automated test-case evaluation. The system could be extended to competitive programming platforms, coursework grading, and recruitment testing by incorporating fixed input-output validation, scoring metrics, and test coverage reporting. Cold-start optimization through provisioned concurrency, workload prediction, and warm container pooling can decrease initial invocation latency and improve user experience for sporadic workloads.

Further evolution may include continuous user authentication and submission tracking backed by a database layer, enabling user dashboards, submission history, performance analytics, and personalized feedback. Advanced sandbox hardening through syscall filtering (seccomp), cgroup-based resource quota controls, and runtime vulnerability scanning would add additional security assurance. AI-aided debugging and code analysis tools could further transform the platform into a comprehensive cloud-native coding and assessment environment.

VII. CONCLUSION

This paper presented the design and implementation of a cloud-based Remote Code Execution (RCE) system built on a serverless architecture using AWS Lambda and Docker container technology. The main goal was to overcome the scalability, cost, and security constraints of traditional virtual machine-based or persistent server-based execution environments.

By leveraging short-lived Lambda containers, the proposed system guarantees that every piece of code is executed in a controlled, isolated, and stateless environment. The architecture demonstrates that serverless computing can successfully deliver multi-language code execution with high sandboxing guarantees. Python, Java, and C++ support was implemented through language-specific execution pipelines within a Docker-based Lambda environment.

Experimental analysis revealed consistent execution behavior, accurate error reporting, and reliable performance under repeated and parallel submissions. Cold-start overhead was tolerable and warm invocations further improved response time. Security was a primary design concern, with isolated container instances, restricted file system access, enforced resource limits, and no persistent state collectively minimizing the attack surface.

In conclusion, a serverless, container-based architecture offers a scalable, cost-efficient, and secure alternative to traditional RCE implementations. The system provides a viable foundation upon which modern online coding platforms, learning management systems, and technical assessment tools can be built using cloud-native technologies.

References

1. M. Roberts, "Serverless Architectures," IEEE Cloud Computing, vol. 4, no. 6, pp. 68–73, 2018.
2. I. Baldini et al., "Serverless Computing: Current Trends and Open Problems," in Research Advances in Cloud Computing, Springer, 2017.
3. E. Jonas et al., "Cloud Programming Simplified: A Berkeley View on Serverless Computing," arXiv preprint arXiv: 1902.03383, 2019.
4. D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux Journal, no. 239, 2014.
5. C. Pahl, "Containerization and the PaaS Cloud," IEEE Cloud Computing, vol. 2, no. 3, pp. 24–31, 2015.
6. R. Dua, A. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in IEEE IC2E, 2014.
7. M. Satyanarayanan, "The Emergence of Edge Computing," Computer, vol. 50, no. 1, pp. 30–39, 2017.
8. Amazon Web Services, "AWS Lambda Developer Guide," 2024.
9. Google Cloud, "Cloud Functions Documentation," 2024.
10. [Microsoft Azure, "Azure Functions Documentation," 2024.
11. S. Hendrickson et al., "Serverless Computation with OpenLambda," in USENIX ATC, 2016.
12. J. Spillner, "Function-as-a-Service: Towards a Platform for Serverless Applications," in IEEE SOCA, 2017.
13. L. Wang et al., "Peeking Behind the Curtains of Serverless Platforms," in USENIX ATC, 2018.
14. A. Shankar et al., "Sandboxing Untrusted Code in the Cloud," IEEE Security & Privacy, 2016.
15. P. Silva et al., "Isolation in Container-Based Cloud Systems," Future Generation Computer Systems, 2018.