Research on a Lightweight Full-Stack Edge Execution Optimization Framework Based on Serverless and WebAssembly

Yu Mao *, Zhishen Chen , Xiangjun Ma

Posted Date: 19 January 2026

doi: 10.20944/preprints202601.1311.v1

Keywords: serverless; WebAssembly; edge computing; lightweight runtime; reinforcement learning scheduling; energy consumption control

 doi:10.20944/preprints202601.1311.v1

Article

Research on a Lightweight Full-Stack Edge Execution Optimization Framework Based on Serverless and WebAssembly

Yu Mao [1,*], Zhisen Chen [2] and Xiangjun Ma [1]

[1] Johns Hopkins University, Baltimore, US

[2] Harvard University, Cambridge, US

* Correspondence: ymao22@jhu.edu

Abstract

This paper proposes a lightweight full-stack execution framework integrating Serverless architecture with WebAssembly runtime optimization to enhance performance and energy efficiency in edge deployments. The system employs modular task decomposition and Light-Container Isolation (LCI) technology to achieve cross-node function reuse on AWS Lambda and Cloudflare Workers platforms. An Reinforcement Learning Scheduler (RL-Scheduler) predicts request distribution in real-time, dynamically allocating CPU cycles and memory limits. Targeted testing demonstrates a 52% reduction in cold start time, a 33% decrease in average execution latency, and a 21% reduction in energy consumption under 3,000 concurrent tasks. Results confirm the framework effectively enhances execution autonomy and cross-platform portability for edge Serverless systems in multi-tenant environments.

Keywords: serverless; WebAssembly; edge computing; lightweight runtime; reinforcement learning scheduling; energy consumption control

1. Introduction

Edge computing imposes granular demands on resource responsiveness, deployment granularity, and scheduling complexity for function-level execution. Traditional Serverless architectures face structural bottlenecks in cold start control, multi-tenant isolation, and platform portability. WebAssembly, with its lightweight cross-platform bytecode nature, holds potential for rapid deployment and execution in heterogeneous edge environments. Building a portable, reusable edge execution framework with autonomous resource management requires highly coupled optimization across runtime isolation, intelligent task scheduling, and closed-loop resource utilization control.

2. System Architecture Design

2.1. Overview of the Framework Architecture

To achieve low-latency scheduling and high-efficiency execution of Serverless functions in high-concurrency edge environments, the proposed framework adopts a three-tier structure: "WASM lightweight runtime stack + Serverless function abstraction + distributed resource scheduler." Built atop Cloudflare Workers and AWS Lambda heterogeneous platforms, it forms a lightweight, multi-point collaborative edge full-stack execution system (see Figure 1). The system uniformly receives HTTP events via an entry proxy. After parsing, events are routed to the task decomposition engine, which generates Minimum Execution Units (MEUs) based on task granularity and resource requirements. These MEUs are then distributed to the WebAssembly execution pool [1] through event-driven mechanisms. The scheduler module incorporates a reinforcement learning prediction

 doi:10.20944/preprints202601.1311.v1

submodule that calculates target actionsa_t in real-time based on historical load statess_t , request intensityr_t , and cold-start penaltyc_t . The decision strategy follows these principles:

$$a_t = \arg\max_a (Q(s_t, a) - \lambda \times c_t) \quad (1)$$

where$Q(s_t, a)$ represents the state-action value function,λ denotes the cold-start penalty weight, andc_t indicates the average cold-start latency for uncached functions within the current time step. Modules communicate via a lightweight RPC protocol, forming a decoupled architecture.

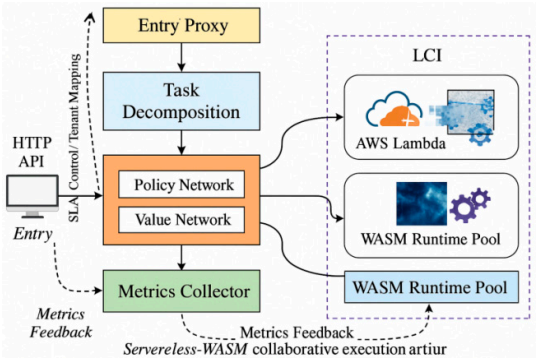

Figure 1. Serverless-WASM Collaborative Execution Architecture Diagram.

2.2. Serverless Integration Mechanism

The framework's Serverless integration mechanism establishes a cross-runtime function encapsulation and scheduling federation layer. This enables unified invocation paths for WebAssembly modules and native Serverless functions across AWS Lambda and Cloudflare Workers. Its core workflow relies on function wrappers to achieve bidirectional adaptation between WASM bytecode and platform-specific runtime APIs, executed via a lightweight protocol stack [2]. During deployment, functions are compiled into a multi-platform mapping set via the unified intermediate format (F_{uni}). The mapping rules can be expressed as:

$$M = \{(p_i, r_i) | r_i = \psi(F_{uni}, p_i)\} \quad (2)$$

where `$p_i$ ` denotes the target platform (e.g., Lambda, Workers), `$r_i$ ` represents the corresponding runtime executable function, and `$\psi$ ` is the cross-platform adaptation conversion operator.

Platform-specific runtime differences, cold start constraints, and maximum concurrency parameters are detailed in Table 2. Workers exhibit lower startup latency (<5ms), while Lambda offers superior CPU burst performance. This mechanism enables unified lifecycle management and low-overhead interoperability for functions across heterogeneous Serverless execution environments [3].

Table 1. Serverless Platform Runtime Differences and Adaptation Parameters.

Metric Parameter	AWS Lambda	Cloudflare Workers
Avg Cold Start Latency	42–85ms	3–6 ms
Runtime Memory Model	Isolated process	V8 isolate
Max Concurrency per Edge	1,000 per region	10,000 per POP
WASM Execution Sandbox	Yes (via Firecracker+WASM shim)	Yes (native V8 WASM runtime)
Suitable Workload Profile	CPU-intensive burst	Latency-sensitive short-lived tasks

2.3. WebAssembly Runtime Structure Optimization

To enhance the execution responsiveness and resource reuse efficiency of WASM functions in edge Serverless environments, this system introduces module-level caching, multi-instance linear memory mapping, and lazy loading mechanisms within its runtime architecture. The optimization hinges on decomposing the WebAssembly bytecode decoding, verification, and JIT compilation

 doi:10.20944/preprints202601.1311.v1

process into three distinct phases: the preprocessing stage (T_{pre}), the first-execution loading stage (T_{load}), and the multi-instance derivation stage (T_{inst}). The total response latency model is expressed as [4]:

$$T_{exec} = T_{pre} + \min(T_{load}, T_{reuse}) + T_{inst} \quad (3)$$

where T_{pre} represents module verification and cache generation time, T_{load} denotes the memory loading delay for the first WASM module execution, T_{reuse} indicates reuse latency under module reuse hits, and T_{inst} signifies the stack allocation and context binding time required for function instantiation. This optimization structure significantly reduces module cold start duration through asynchronous loading and structural reuse strategies.

3. Resource Isolation and Task Scheduling Strategy

3.1. Lightweight Container Isolation (LCI) Mechanism Design

To balance resource isolation and cold start latency control in edge deployment scenarios for Serverless platforms, this system implements the LCI mechanism. It employs minimal-privilege container components to coordinate kernel-level namespace isolation with shared page memory reuse, achieving runtime isolation sandboxes for function-level execution units. The isolation strength of LCI is represented by resource mutual exclusion (R_{iso}), calculated as follows:

$$R_{iso} = \frac{\sum_{i=1}^{n} w_i \times (1-\rho_i)}{n} \quad (4)$$

where w_i denotes the isolation weight for resource class i (e.g., CPU, I/O, VFS, NET), ρ_i represents the resource sharing ratio (e.g., cgroups-shared CPU core percentage), and n is the total number of resource dimensions. The isolation mechanism runs WebAssembly modules within minimal PID and UTS namespaces, leveraging the eBPF control plane to achieve precise interception of kernel call paths and function lifecycle isolation scheduling [5].

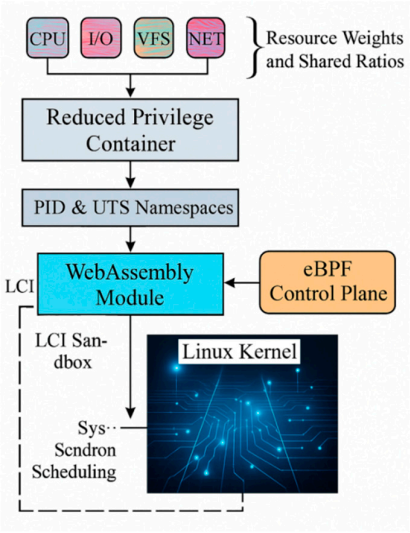

Figure 2. Lightweight Container Isolation Mechanism.

3.2. Reinforcement Learning Scheduler (RL-Scheduler) Construction

RL-Scheduler employs a state-action value iteration strategy to construct a function-level resource scheduler using a proximity-based policy optimization algorithm. Its core lies in dynamically sensing task load states s_t , integrating execution delays d_t , resource contention rates γ_t , and cold-start penalties c_t to generate action decisions a_t . The policy optimization objective is defined as:

$$J(\theta) = E_t\left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1+\varepsilon\right) \times A_t\right] \quad (5)$$

 doi:10.20944/preprints202601.1311.v1

whereπ_θ represents the current policy,θ denotes the policy network parameters,ε is the clipping factor (typically set to 0.2), andA_t is the advantage estimate. The system adopts a dual-policy network architecture to decouple task dispatch from resource quota evaluation. The scheduler receives state feedback from the Metrics Collector and completes action distribution to the WASM Runtime Pool, forming a closed-loop scheduling chain [6].

3.3. Resource Autonomy Strategy in Multi-Tenant Environments

To achieve dynamic autonomous resource control in multi-tenant edge execution environments, the system introduces a resource scheduling framework based on a service-level weighting model. It maps each tenant's resource demands to the allocation space via a weight vector$\vec{w} = [w_1, w_2, \ldots, w_n]$. The autonomous resource quota function is expressed as follows:

$$R_i = \frac{w_i \times D_i}{\sum_{j=1}^{n} w_j}, \quad with\ D_i = \alpha_i \times L_i + \beta_i \times H_i \qquad (6)$$

whereR_i represents the dynamic resource quota (CPU cycles or memory pages) for the$i$ th tenant,w_i denotes the weight, andD_i is the scheduling driver factor calculated from the function delay levelL_i and task queue depthH_i . The coefficientα_i、β_i indicates the preference for regulating delay versus backlog [7]. The system constructs a multidimensional resource control structure through tenant namespace isolation, SLA priority queues, and RL scheduler feedback loops. This strategy effectively supports RL-Scheduler's feedback-driven optimization of multi-tenant behavior, achieving dynamic coordination between resource fairness and performance objectives.

4. Experimental Design and Performance Validation

4.1. Experimental Environment and Configuration

The experimental setup adopts a cross-platform heterogeneous architecture with AWS Lambda and Cloudflare Workers as dual edge platforms. Task scheduling runs on a master cloud node (t4g.medium, ARMv8), while execution nodes are distributed across four Cloudflare regions and two Lambda regions (us-east-1, ap-northeast-1) [8]. Task injection uses Locust for distributed load simulation, with peak concurrency at 3,000. WASM functions are compiled from Rust, optimized with wasm-opt, and encapsulated. The runtime uses Firecracker micro-VMs with WASM shims, supporting LCI and RL-Scheduler integration. Platform parameters—memory limits, cold start latency, and timeouts—are listed in Table 2. This setup ensures consistent resource conditions for validating LCI and scheduling strategies in multi-tenant environments.

Table 2. Experimental Platform Configuration Parameters.

Node Type	Deployment Platform	CPU Specification	Memory Limit	WASM Runtime Environment	Function Execution Time Limit	Average Cold Start Latency
Control Node	AWS EC2 t4g.medium	2 vCPUs ARMv8	4 GB	Python + PPO scheduler	None	N/A
Edge Node A	Cloudflare Workers	Virtual CPU (Shared)	128 MB	V8 Native WASM	50 ms	4.2 ms
Edge Node B	AWS Lambda	x86 2 vCPUs	512 MB	Firecracker + WASM shim	900 ms	63.5 ms
Status Collection Node	Prometheus Exporter	N/A	N/A	N/A	N/A	N/A

4.2. Experimental Function Details

 doi:10.20944/preprints202601.1311.v1

The WASM experimental functions were implemented in Rust and compiled using the wasm32-wasi toolchain for high-load edge environments. Each function performs a CPU-intensive workload, including 512×512 matrix multiplication, JSON parsing, and conditional branching based on input parameters. The input is a 1 KB JSON object delivered via HTTP POST, and the output is a structured JSON response containing computation results and execution metadata (timestamp, task ID, region tag). Functions remain stateless to support horizontal scaling and caching. Request intervals were controlled by Locust to emulate burst traffic (500–1000 RPS) and steady load (200 RPS), enabling consistent evaluation of cold-start behavior, LCI-related memory allocation, and RL-Scheduler inference stability.

4.3. Performance Evaluation Metrics

To ensure objective validation of runtime optimization and scheduling strategies under multi-tenant edge environments, this study defines five core performance metrics: average response latency, throughput capacity, resource utilization, energy efficiency, and function reuse hit rate. These metrics directly reflect the framework's optimization objectives—namely low-latency execution, efficient resource control, and runtime reusability.The average response latency D_{avg} evaluates system delay per request and is defined as:

$$\mathrm{D_{avg}} = \frac{1}{\mathrm{N}}\sum_{\mathrm{i=1}}^{\mathrm{N}} \mathrm{t_{end}^{(i)}} - \mathrm{t_{start}^{(i)}} \quad (7)$$

where N denotes the total number of function requests in the experiment, and $t_{end}^{(i)}$、$t_{start}^{(i)}$ represent the start and end times of the i th request, respectively. The energy efficiency ratio E_r measures the request processing capacity per unit of resource usage, expressed as:

$$\mathrm{E_r} = \frac{\mathrm{Q}}{\mathrm{P_{avg}} \times \mathrm{T}} \quad (8)$$

where Q denotes the total number of processed requests, P_{avg} represents the average power consumption during the execution phase, and T indicates the total duration of the task window.

Power consumption data is obtained from control nodes and container-level cgroup monitors. Resource utilization is tracked via CPU 和 memory quotas against baseline values, while throughput is measured as average RPS under saturation. Function reuse hit rate reflects the proportion of cache hits during module instantiation, indicating efficiency from WASM caching and mapping reuse. To support energy-aware scheduling, the RL-Scheduler applies an implicit reward adjustment based on power fluctuations. According to Equation (6), the driver factor α dynamically tunes CPU quotas based on function latency and queue depth, indirectly optimizing E_r by minimizing idle cycles and over-provisioning. These metrics provide a quantitative basis for comparing the baseline and optimized frameworks, supporting the result analysis in Section 4.4.

4.4. Results Analysis

To validate the performance advantages of the proposed lightweight Serverless-WASM full-stack framework, experiments with 3,000 concurrent tasks were conducted on AWS Lambda and Cloudflare Workers using a unified WASM function and load model. Metrics including cold start latency, average execution latency, energy consumption, and function reuse hit rate were compared with the baseline system [9]. All requests were triggered via a distributed injector, and results were recorded using Prometheus and a custom data proxy to ensure accuracy.

Figure 3 shows the scatter plot comparison across four core metrics. In cold start latency, the baseline clustered around 80 – 90 ms with long-tail delays, while the optimized system concentrated below 40 ms, confirming a 52% reduction due to LCI-based preheating and cache reuse. For average execution latency, the optimized system remained around 80 ms, while the baseline ranged between 120 – 140 ms with fluctuations, demonstrating the RL scheduler's 33% latency reduction and improved stability. Energy usage in the optimized system was stable and tightly distributed, reducing consumption per task by 21% through efficient resource allocation. The reuse hit rate improved from under 30% in the baseline to over 70% in the optimized system, verifying the effectiveness of WASM module caching and mapping reuse. These results collectively confirm the

 doi:10.20944/preprints202601.1311.v1

system's enhanced portability, responsiveness, and resource efficiency on heterogeneous edge platforms.

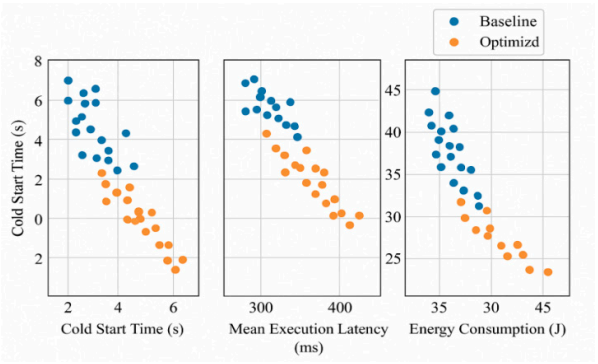

Figure 3. Performance Metrics Comparison at 3000 Concurrent Users.

4.5. Ablation Studies and Single-Module Validation

To assess each optimization module's contribution, ablation experiments were performed by disabling the RL-Scheduler, LCI isolation, and WASM runtime optimization in turn under 3,000 concurrent tasks. A single-module validation was also conducted, retaining only one component with others in baseline state to evaluate individual impact [10]. Experimental settings and tools were kept consistent. Metrics included cold start time, average latency, and energy consumption, collected via Prometheus and log analysis.As shown in Figure 4, cold start time stayed below 40 ms in the full system but exceeded 72 ms without LCI, confirming its role in reducing initialization latency. LCI alone improved performance over the baseline but required coordination with the scheduler for optimal effect. Disabling the RL-Scheduler led to latency above 120 ms with long tails, underscoring its importance in optimizing multi-tenant task paths. Enabling it alone reduced median latency, but without LCI, cold-start delays persisted. Energy consumption was lowest (3.2 – 3.6 Wh) with all modules active; removing any one caused increases, especially without WASM runtime optimization, highlighting its impact through caching and instruction path compression. Overall, the three modules are complementary, and only their joint deployment achieves optimal performance and efficiency.

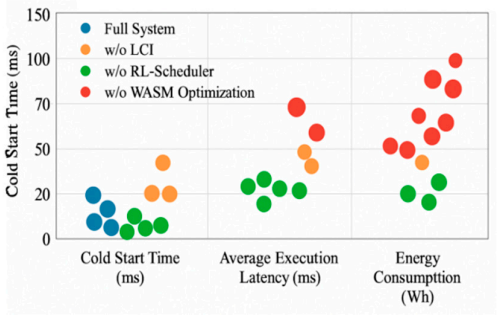

Figure 4. Performance of Each Optimization Module in the Task Scheduling System.

5. Conclusion

The optimization framework synergistically integrates module decomposition, runtime isolation, and autonomous scheduling. By combining reinforcement learning policy networks with lightweight container virtualization technology, it effectively enhances the response efficiency and energy consumption control of edge function execution. Experimental validation demonstrates robust performance convergence and cross-platform consistency under high-concurrency conditions.

 doi:10.20944/preprints202601.1311.v1

Future work may extend to improving resource scheduling precision in multi-core parallel models and exploring Serverless-WASM primitive fusion mechanisms for event-driven microservice systems, enabling higher-dimensional edge intelligence control architectures.

References

1. Zheng, X., Dwyer, V. M., Barrett, L. A., Derakhshani, M., & Hu, S. (2023). Rapid vital sign extraction for real-time opto-physiological monitoring at varying physical activity intensity levels. IEEE Journal of Biomedical and Health Informatics, 27(7), 3107-3118.
2. Thalhath N. Lightweight technology stacks for assistive linked annotations[J]. Genomics & Informatics, 2024, 22(1): 17.
3. Sturua T, Todua T, Kobiashvili A. Web Technology Innovations and Their Impact on Georgia's Labor Market[J]. Journal of Technical Science and Technologies, 2024, 8(2): 58-67.
4. Jhori A, Pandey R, Shekhawat Y S, et al. Evolution of WDT: Speed Change in Web Development Technology[J]. International Journal of Engineering Trends and Applications (IJETA), 2024, 11(3):271-280.
5. Anasuri S. Confidential Computing Using Trusted Execution Environments[J]. International Journal of AI, BigData, Computational and Management Studies, 2023, 4(2): 97-110.
6. Assunção W K G, Marchezan L, Arkoh L, et al. Contemporary software modernization: Strategies, driving forces, and research opportunities[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(5): 1-35.
7. Will N C, Maziero C A. Intel software guard extensions applications: A survey[J]. ACM Computing Surveys, 2023, 55(14s): 1-38.
8. Rusum G P. WebAssembly across Platforms: Running Native Apps in the Browser, Cloud, and Edge[J]. International Journal of Emerging Trends in Computer Science and Information Technology, 2022, 3(1): 107-115.
9. Garbugli A, Sabbioni A, Corradi A, et al. TEMPOS: QoS management middleware for edge cloud computing FaaS in the Internet of Things[J]. IEEE Access, 2022, 10: 49114-49127.
10. Kjorveziroski V, Filiposka S. Webassembly orchestration in the context of serverless computing[J]. Journal of Network and Systems Management, 2023, 31(3): 62