HybridServe: Adaptive WebAssembly-Container Runtime Selection for Edge Serverless Computing

Seokhyeon Kang[1,†], Moohyun Song[1,†], Taeyoon Kim[2,†], Soohyuk Lee[1], Jaeseob Han[1], Hyeokman Kim[1], Kyungyong Lee[2]

[1]College of Computer Science, Kookmin University, Seoul, South Korea
[2]Department of Data Science, Hanyang University, Seoul, South Korea

{kh3654p,mhsong,cg10036,jaeseob,hmkim}@kookmin.ac.kr,{tykim7,kyungyong}@hanyang.ac.kr

Abstract

Edge serverless environments with limited CPU, memory, and heterogeneous hardware demand efficient execution under resource constraints. A key challenge in serverless computing is cold start, where a function instance is created and initialized from scratch instead of reusing an already initialized (warm) instance, which is particularly severe in edge environments. Container-based platforms exhibit cold start latencies exceeding hundreds of milliseconds with tens of megabyte images, while WebAssembly (WASM) provides less than ten millisecond initialization and compact binaries, but slower execution. We comprehensively analyze multiple WASM runtimes (Wasmtime, Wasmer, WasmEdge) and their compiler backends, revealing that WASM reduces cold start latency by up to 88.1% and image size by up to 99.17%, while exhibiting 36.15% slower execution and 9.26% higher power consumption for compute-intensive workloads. Based on these complementary characteristics, we propose HybridServe, a dynamic runtime selection framework that leverages WASM during container cold starts while preparing containers in the background. Evaluation on Azure Function Trace demonstrates 43.11% average response time reduction versus WASM-only and 91.9% cold start latency reduction versus container-only deployments, effectively mitigating the performance-isolation trade-off in edge serverless computing.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

Serverless, WebAssembly, Container, Edge Computing

ACM Reference Format:
Seokhyeon Kang[1,†], Moohyun Song[1,†], Taeyoon Kim[2,†], Soohyuk Lee[1],, Jaeseob Han[1], Hyeokman Kim[1], Kyungyong Lee[2] . 2025. HybridServe: Adaptive WebAssembly-Container Runtime Selection for Edge Serverless Computing. In *11th International Workshop on Serverless Computing (WoSC11 '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3774899.3775011

[†]These authors contributed equally to this research.

1 Introduction

Serverless computing is an execution model where cloud providers manage all infrastructure operations including resource allocation, scaling, and maintenance, allowing developers to focus solely on application logic. Developers write code as individual functions that are executed on-demand in response to events, with billing based only on actual resource consumption, resulting in significant cost efficiencies.

The recent emergence of edge computing has driven interest in extending the serverless paradigm beyond centralized cloud environments. Early systems leveraged cloud serverless models to reduce management overhead and improve cost efficiency. However, as data volumes increased exponentially, with IoT environments alone connecting over 25 billion devices generating exabyte-scale data, fundamental limitations of cloud-centric approaches emerged [6, 18]. For applications requiring low latency, such as autonomous vehicles generating gigabytes of data per second, transmitting all data to the cloud incurs unacceptable latencies of tens to hundreds of milliseconds [6]. Edge computing addresses these limitations by performing computation closer to data sources, significantly reducing latency and network bandwidth consumption [3, 20]. In environments with limited resources, serverless computing's on-demand execution and scale-to-zero characteristics enable efficient utilization of limited edge resources by loading functions only when needed and releasing resource after idle periods [1]. Moreover, the intermittent event generation patterns typical of IoT devices naturally align with serverless computing's event-driven execution model [2]. Therefore, edge serverless architectures must combine the manageability of cloud computing with the low latency of edge processing while operating efficiently within resource constraints.

Current serverless platforms primarily utilize container-based isolation, benefiting from mature ecosystems and proven reliability. However, containers incur cold start latencies ranging from hundreds of milliseconds to several seconds due to OS-level virtualization overhead. To reduce these initialization delays, lightweight runtimes such as WebAssembly (WASM) have been explored, with various implementations (Wasmtime, Wasmer, WasmEdge) achieving cold starts under tens of milliseconds through bytecode execution in sandboxed environments [10]. Nevertheless, WASM runtimes demonstrate slower execution performance compared to native containers, due to runtime compilation and system call translation overhead. Edge computing environments exacerbate this trade-off, as they must handle both latency-sensitive IoT sensor events and computationally intensive tasks. These conflicting requirements indicate that edge serverless systems must transition from single-runtime architectures toward adaptive frameworks capable of selecting appropriate runtimes based on workload characteristics.

In this paper, we propose HybridServe, a hybrid serverless system that combines WASM and containers to address these challenges. While WASM was originally designed for high-performance code execution in web browsers, its small binary size and fast initialization time make it an attractive technology for edge environments. Our experiments demonstrate that WASM reduces cold start latency by up to 88.1% compared to containers, with image sizes up to 99.17% smaller. However, WASM delivers approximately 36% slower execution performance compared to natively-executed containers. HybridServe adopts an approach that leverages the complementary strengths of both technologies, reducing response time by 43.11% compared to WASM-only and cold start latency by 91.9% compared to container-only deployments.

2 Light Runtimes for Edge Serverless

Implementing serverless computing effectively in edge environments requires fundamentally different approaches from cloud deployments. Resource-constrained edge nodes with stringent latency requirements demand lightweight runtime technologies, such as lightweight containers, unikernels, and WebAssembly, as alternatives to traditional heavyweight execution environments.

2.1 Limitations of Container-Based Serverless

Most current serverless platforms rely on container technology for function execution. While this approach provides language independence and isolation guarantees, it is optimized for cloud environments with abundant resources and exhibits several limitations in resource-constrained, latency-sensitive edge deployments.

First, cold start latencies range from hundreds of milliseconds to several seconds [5, 13]. Second, container images typically range from tens of megabytes to gigabytes, which is 70-80 times larger than equivalent WASM modules [14]. Third, containers are architecture specific (x86, ARM, etc.), requiring separate images for each heterogeneous edge device type [4, 9]. These limitations demonstrate that containers fail to meet the edge serverless requirements of fast initialization, small footprint, and architecture independence.

2.2 Alternative Runtime Technologies

To overcome these container limitations, several lightweight runtime technologies have been proposed for serverless environments.

Lightweight Containers. Ongoing efforts to optimize containers for serverless workloads have produced several lightweight alternatives. Linux distribution slim images, Alpine Linux-based images, and Google's Distroless images [7] reduce sizes by removing unnecessary system components and include only application and runtime dependencies. Standard Ubuntu images measure hundreds of megabytes, while Alpine-based ones stay under 5MB. However, despite these optimizations, containers still carry fundamental overhead from OS-level isolation mechanisms including namespace creation, cgroup configuration, and filesystem mounting. They cannot achieve the tens of milliseconds cold start times and architecture-independent deployment required for edge serverless.

Unikernel. Unikernels compile applications together with minimal OS libraries into specialized machine images that execute in a single address space [12]. This approach minimizes attack surface and significantly reduces image size. However, unikernels require recompiling applications with OS libraries, limiting their generality and requiring substantial effort to port existing applications.

WebAssembly. While lightweight containers and unikernels provide strong isolation and good performance, they still suffer from OS kernel overhead and limited portability, respectively. WASM instead offers sandboxed execution, fast initialization, and architecture-independent portability that are attractive in heterogeneous edge environments, albeit with lower execution performance and weaker system-level integration than native or containerized code. Given this trade-off and the importance of fast startup and portability at the edge, we focus on WASM as a promising alternative to containers for edge serverless computing.

3 WebAssembly Runtime Composition

As discussed in Section 2, WASM seems to have emerged as a promising alternative to containers for edge serverless computing due to its lightweight characteristics and fast initialization times. A claim we further support with our analysis in Section 4.2.

While WASM was initially developed to address JavaScript's performance limitations, it has recently expanded beyond browsers into diverse server environments and edge computing domains. WASM offers advantages through its architecture independence and ability to compile code written in various languages such as C/C++ and Rust into a unified bytecode format for execution.

WASM's small binary size and low runtime initialization overhead enable significant reductions in cold start latency compared to containers. WASM modules typically range from tens of kilobytes to a few megabytes, offering substantially smaller footprints than container images.

3.1 Runtime Engines

WASM runtimes fall into two main categories. Browser-embedded runtimes such as Chrome's V8, Firefox's SpiderMonkey, and Safari's JavaScriptCore are integrated as part of JavaScript engines to execute WASM in web environments. In contrast, standalone runtimes operate independently outside browser environments, including servers, edge devices, and cloud platforms. Representative WASM runtimes include Wasmtime, Wasmer and WasmEdge, each with different design philosophies, implementation languages, and compiler backends that result in distinct performance characteristics. WASM runtimes generally support three execution modes:

- Interpreter: Executes bytecode line-by-line with minimal initialization time but very low execution performance.
- Just-In-Time (JIT): Compiles bytecode to machine code during execution, enabling runtime optimization and caching for improved performance at the cost of higher initialization overhead.
- Ahead-Of-Time (AOT): Pre-compiles bytecode to machine code before execution, providing low cold-start time and consistent performance.

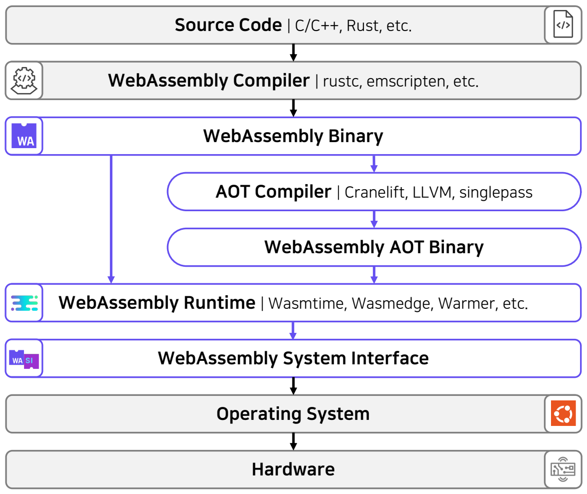

Source Code | C/C++, Rust, etc.
WebAssembly Compiler | rustc, emscripten, etc.
WebAssembly Binary
AOT Compiler | Cranelift, LLVM, singlepass
WebAssembly AOT Binary
WebAssembly Runtime | Wasmtime, Wasmedge, Warmer, etc.
WebAssembly System Interface
Operating System
Hardware

Figure 1: WebAssembly runtime architecture with compilation paths and system interfaces

Table 1 presents the compilers, interpreters, and supported execution modes for Wasmtime, Wasmer, and WasmEdge. Each runtime employs different compiler backends. For instance, Wasmtime uses Cranelift and Winch, Wasmer supports Cranelift, LLVM, and Singlepass, while WasmEdge utilizes an LLVM-based compiler.

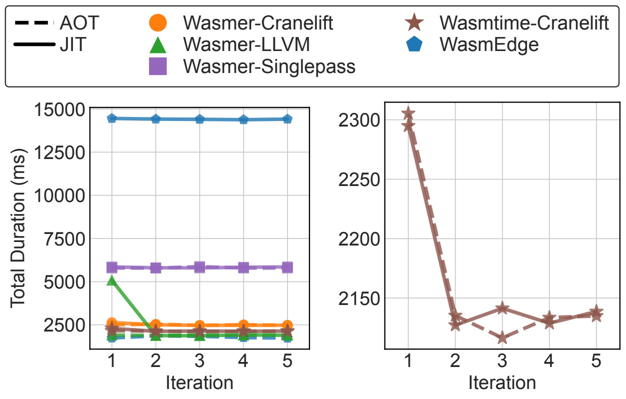
Table 1: Comparison of Wasm Runtimes: Compilers and Supported Execution Modes

Runtime / Properties	Wasmtime	Wasmer	WasmEdge
Compiler(s)	Cranelift Winch	Cranelift LLVM Singlepass	LLVM-based compiler
Interpreter	Pulley	Wasmi, Wamr	WasmEdge Interpreter
JIT	✓	✓	✓
AOT	✓	✓	✓

3.2 WebAssembly System Interface

We briefly discuss WASI because its design both explains some of the performance overheads we observe later and clarifies the components in Figure 1. WASM inherently executes in a sandboxed environment, preventing direct access to system resources such as filesystems, networks, and environment variables. To overcome these limitations, WASI was proposed, enabling WASM runtimes to bind host operating system calls to the WASI interface, allowing WASM modules controlled access to system resources [8]. During WASM module execution, the runtime intercepts WASI calls and executes the corresponding system calls.

Figure 1 illustrates the overall architecture of WASM execution. Code written in C/C++ or Rust can be compiled to WASM using compilers such as rustc or Emscripten. The resulting WASM binaries can be executed directly through the runtime, or alternatively, AOT binaries can be generated using runtime-specific compilers before execution. WASM modules interact with system resources through the WASI interface.



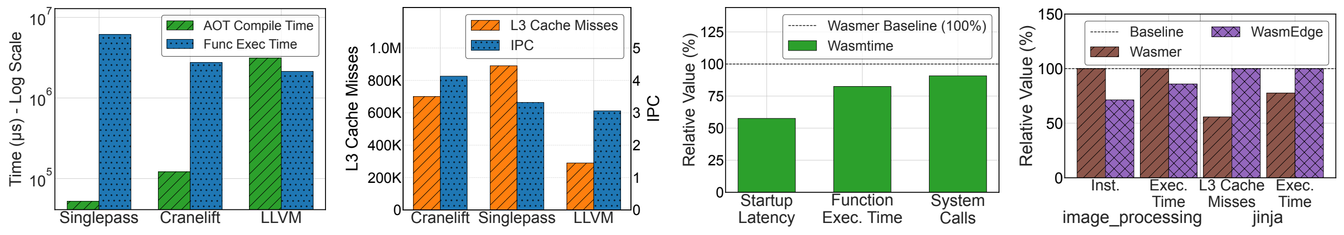
(a) All runtime comparison **(b) Wasmtime Only**

Figure 2: Comparison WebAssembly Runtime Performance

4 Edge Serverless Runtime Analysis

Edge serverless environments differ from cloud environments with limited resources (CPU, memory, storage, power), intermittent and unpredictable traffic patterns, stringent low-latency requirements, and heterogeneous hardware platforms. To identify suitable runtimes for edge serverless, we analyze the performance characteristics of different WASM runtimes and their compiler backends, and evaluate trade-offs between WASM and containers across cold start latency, image size, execution time, CPU/memory utilization, power consumption, Instructions Per Cycle (IPC), and system call frequency.

Experiment environment and workload. We evaluated three major WASM runtimes (Wasmtime, Wasmer, WasmEdge) and their respective compiler backends across multiple edge devices including Intel NUC (i5-1240P) and NVIDIA Jetson Orin Nano. For container baseline measurements, we utilized the Docker, containerd and runc stack. While FunctionBench [11], which is a serverless benchmark suite for evaluating diverse workload characteristics including CPU, memory, disk, and network operations, provides Python implementations, we converted these to Rust for WASM compilation and used the same Rust binaries for native container performance measurements. In our experiments, We excluded interpreter mode due to its significantly lower execution performance, focusing exclusively on JIT and AOT compilation modes. Wasmtime's Winch compiler was excluded as it currently only supports x86 architectures. We collected IPC and system call counts using `perf` and `strace` tools, while power metrics were measured through Inter-Integrated Circuit (I2C) sensors integrated in the edge devices. CPU utilization was calculated by comparing measurements at workload start and end points, while peak memory usage was measured through `/proc/meminfo` metrics. All measurements were collected at one ms intervals. Before benchmark execution, we measured baseline system resource consumption including monitoring overhead for 15 seconds, then extracted only the workload-specific metrics.



(a) Compiler Performance Trade-off **(b) Compiler Efficiency** **(c) Wasmtime vs. Wasmer with Cranelift** **(d) WasmEdge vs. Wasmer with LLVM**

Figure 3: Performance Characterization of WebAssembly Runtimes and Compilers

4.1 Comparing WASM Runtimes and Compilers

Figure 2 shows the execution time distribution of FunctionBench's `image_processing` workload for each WASM runtime on Intel NUC.

Limitation of JIT compile. JIT compilation incurs overhead from real-time transition of bytecode into machine code when caching is unavailable. Figure 2a demonstrates this overhead across multiple WASM runtimes. Unlike other runtimes, WasmEdge JIT does not implement compilation caching, resulting in consistent recompilation penalties across all invocations. This falls far short of the tens-of-milliseconds response times required by edge serverless environments. Even runtimes with JIT caching (Wasmer, Wasmtime) cannot avoid first execution overhead. The scale-to-zero characteristic of serverless leads to frequent function unloading, causing cache invalidation that significantly diminishes JIT advantages.

Compilation strategy analysis. Figure 2a demonstrates the total duration of WASM execution across multiple iterations for different compilation approaches. Once AOT compilation is complete, subsequent executions achieve consistently fast cold starts, while JIT requires slow initial execution for caching before reaching comparable performance. However, AOT compilation's primary drawback is hardware dependency. For example, AOT binaries compiled for x86 cannot execute on ARM-based edge devices. This limitation can be addressed by performing AOT compilation on first execution and caching the results, or by pre-compiling to AOT before deployment. Figure 2b compares JIT and AOT execution times for the `image_processing` workload using Wasmtime Cranelift, showing five executions without prior JIT caching or AOT compilation. Results show minimal difference between JIT execution with caching (2,294ms) and AOT compilation followed by execution (2,305ms) for the initial uncached run.

Performance characterization by compiler. Figure 3a and Figure 3b illustrate the performance characteristics and efficiency metrics of different WASM compiler backends. Even with identical AOT compilation, performance varies slightly depending on the compiler backend used by the WASM runtime. As shown in Figure 3a, which plots compilation time against execution performance, Wasmer's Singlepass compiler exhibited significantly lower performance compared to other runtime compilers. Analysis of the trade-off between compilation speed and execution performance reveals that while Singlepass and Cranelift achieve faster AOT compilation than LLVM, LLVM generates the highest-performing compiled code. This suggests LLVM invests more time in complex optimization passes to generate higher-quality code, achieving the fastest execution times after the compilation completes.

Figure 3b further demonstrates the microarchitectural differences between compilers. At the microarchitecture level, Cranelift achieves the highest IPC, indicating optimal CPU pipeline utilization. LLVM demonstrates superior memory optimization, reducing L3 cache misses by 3.1× compared to Singlepass in the `image_processing` workload. These metrics explain why LLVM-compiled code outperforms other backends despite longer compilation times.

Performance characterization by Runtime. Figure 3c and Figure 3d reveal how runtime architecture impacts performance even when using identical compiler backends. Comparing runtimes using identical compilers reveals the overhead characteristics of runtime architectures themselves. Figure 3c compares Wasmtime and Wasmer, both utilizing the Cranelift backend. The results show that Wasmtime generates fewer system calls and achieves approximately 17.5% faster execution, demonstrating that runtime architecture efficiency impacts performance even when execution code was generated by identical compilers. Figure 3d presents the performance comparison between WasmEdge and Wasmer, both employing LLVM backends, across different workload characteristics. These runtimes exhibited contrasting performance depending on workload type. For the `image_processing` workload, WasmEdge reduced total instruction count by 29% compared to Wasmer, achieving 16.5% faster execution. Wasmer demonstrated superior memory efficiency with 44.2% fewer L3 cache misses, translating to 28.9% faster execution than WasmEdge in the `jinja` workload. This performance variation based on workload characteristics suggests that different runtime architectures optimize for different execution patterns.

4.2 Comparing WASM and Container

We compared Wasmtime Cranelift AOT against containers to understand their trade-offs in edge environments. We selected Wasmtime as the representative WASM runtime because it is a stable project maintained by the Bytecode Alliance, and when using AOT compilation, performance differences between runtimes are negligible except for the Singlepass compiler. Table 2 presents the results using `float_operation` for lightweight operations and `image_processing` for computationally intensive tasks.

Cold Start and Image Size. WASM's primary advantages lie in its rapid initialization and compact image size. For lightweight

Table 2: Performance Comparison between WebAssembly and Container across Different Workloads

	Float Operation		Image Processing	
Metric	**WASM**	**Cont.**	**WASM**	**Cont.**
Cold start (ms)	34.31	270.19	30.97	259.64
Execution (ms)	0.114	0.029	3,875.96	2,846.82
Image size (MB)	0.85	103.51	4.16	104.90
Peak memory (MB)	1.50	31.20	269.06	273.00
Avg. CPU (%)	12.12	20.06	16.36	16.27
Avg. power (W)	0.358	0.958	1.391	1.273
Energy (Wh)	0.000012	0.000225	0.00155	0.00131

workloads, WASM images are approximately 99.18% smaller than containers, while for computationally intensive workloads, they remain 96.03% smaller. This disparity holds significant implications for edge computing environments. When considering limited network bandwidth, including image download time in cold start time would further increase for performance gap. Furthermore, the image sizes shown in the Table 2 represent AOT-compiled files, using non-AOT-compiled images could further reduce download times. This represents a significant advantage for WASM in edge serverless environments that require intermittent deployment of new functions. Even with pre-existing images, WASM demonstrates 87.68% faster cold starts, showing exceptional efficiency for lightweight workloads. Container images used Debian slim to ensure glibc compatibility without specialized build requirements. While Alpine Linux or Distroless images could further reduce size, these alternatives require custom build configurations and complex image construction beyond standard deployment practices.

Execution Performance and Resource Efficiency. As shown in Table 2, containers demonstrate superior execution latency compared to WASM. For `image_operation`, containers complete execution in 2,846ms, approximately 1.36× faster than WASM. Even for lightweight workloads, containers achieve 0.029ms execution time, approximately 3.93× faster than WASM. This performance gap stems from containers executing native code directly, while WASM incurs additional transformation and validation overhead within the runtime. The underlying causes of this performance disparity have been identified through multiple prior studies. For instance, WASM runtimes handle certain operations such as division and modulo inefficiently, and function pointers and virtual function calls incur additional overhead because the runtime must perform table lookups to translate function indices to implementations and validate type conformance between arguments and function signatures [10]. Additionally, accessing system resources through WASI requires extra steps including path validation, permission checks, and descriptor translation [19], ultimately resulting in performance degradation compared to highly optimized code generated by native compilers such as Clang or GCC [17].

Power Consumption. Power efficiency represents an important consideration for battery-powered or power-constrained edge devices. Table 2 demonstrates that WASM's average power consumption for the `image_processing` workload reaches 1.391W, approximately 9.3% higher than containers. This power efficiency gap directly correlates with WASM's slower execution speed. Since identical tasks require longer execution times, WASM consequently consumes more total energy. Therefore, containers may be more suitable for long-running, frequently executed workloads or environments where battery life is important.

Key Findings. These analytical results reveal an intriguing trade-off between the two technologies. While traditional containers excel in execution performance, power efficiency and resource utilization, their cold start latency and image size issues make them unsuitable for edge serverless environments characterized by intermittent traffic. Conversely, WASM provides immediate responsiveness through fast initialization and small footprint but suffers from reduced efficiency for long-running workloads.

These complementary characteristics demonstrate the need for the potential of a hybrid approach. An optimal strategy would process initial requests and intermittent traffic using WASM to ensure user experience, while preparing containers in the background when sustained traffic is detected to handle subsequent requests efficiently.

5 HybridServe Approach

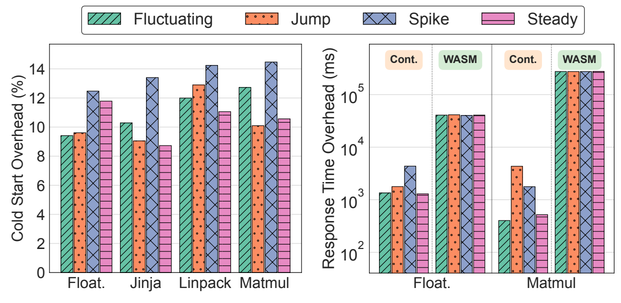
We propose HybridServe, a hybrid serverless system that leverages the complementary characteristics of WASM and containers. HybridServe resolves the trade-off between cold-start latency and execution performance through dynamic runtime selection. For each incoming request, the system first checks for active warm containers and routes the request to a container when available to exploit native execution performance. Otherwise, it executes the request on a WASM runtime to avoid cold-start latency while concurrently preparing warm containers. This adaptive mechanism combines WASM's fast initialization with containers' superior execution performance, achieving low-latency responses and high processing efficiency in edge serverless environments.

5.1 Experiment Scenario

Experiments were conducted on Nvidia Jetson Orin Nano devices to represent edge computing environments. Following the classification scheme proposed by ServiBench [15], traffic scenarios were structured into four representative patterns: (1) Steady – stable traffic maintaining constant request rates, (2) Fluctuating – traffic with periodic variations, (3) Spike – sudden load increase followed by decreases, and (4) Jump – stepwise transitions from low to high levels. Using the Azure Function Trace dataset [16], we selected one function for each pattern that exhibits it and replayed its invocation time series to construct the corresponding traffic scenarios, thus reproducing realistic serverless workloads. We compared against WASM-only using Wasmtime Cranelift AOT and container-only environments.

5.2 Evaluation Result

Figure 4 summarizes the performance comparison, where the vertical axis shows the overhead of each approach relative to HybridServe. In cold-start scenarios (Figure 4a), the container-only approach incurs 8.7%–14.5% higher cold-start latency across the four workloads, whereas WASM-only cold starts are only about 1.5% faster than HybridServe and are therefore omitted. For end-to-end response time (Figure 4b), HybridServe averages 43.34 ms, i.e., 78.5%



(a) Container Cold Start Overhead (b) Response Time Overhead

Figure 4: Performance Comparison Against HybridServe

faster than WASM-only (201.21 ms), while matching container-only performance when warm containers are available and avoiding their cold-start penalties. Overall, these results show that HybridServe mitigates the cold-start versus performance trade-off in edge serverless environments with real-world traffic patterns by combining WASM's fast initialization with containers' superior steady-state execution.

6 Conclusion and Future Work

In this paper, we presented HybridServe, a hybrid serverless system that adaptively selects between WebAssembly and containers based on their complementary strengths: WASM reduces cold-start latency by up to 88.1% and image size by up to 99.17%, but exhibits 36.15% slower execution and 9.3% higher power consumption than containers. HybridServe routes requests to warm containers when available and falls back to WASM otherwise, maintaining responsiveness while avoiding cold starts. On edge devices with traffic derived from the Azure Function Trace dataset, HybridServe reduces average response time by 43.11% compared to WASM-only execution and cold start latency by 91.9% compared to container-only deployment, effectively mitigating the performance-isolation trade-off in edge serverless computing.

Future work will extend HybridServe from a single edge node to distributed edge–cloud cluster environments. Beyond the current runtime selection based solely on warm container availability, we plan to design an intelligent scheduler that also considers cluster-wide CPU, memory, and network conditions, enabling dynamic routing of requests to optimal nodes and runtimes for improved resource efficiency and performance.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (RS-2022-00144309 & RS-2025-25441560)

References

[1] Mohammad S. Aslanpour, Adel N. Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. 2021. Serverless Edge Computing: Vision and Challenges. In *2021 Australasian Computer Science Week Multiconference*. Article 5, 10 pages. https://doi.org/10.1145/3437378.3444367
[2] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (2019), 44–54. https://doi.org/10.1145/3368454
[3] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. 2019. Fog Function: Serverless Fog Computing for Data Intensive IoT Services. In *2019 IEEE International Conference on Services Computing (SCC)*. 28–35. https://doi.org/10.1109/SCC.2019.00018
[4] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. 2022. Pushing Serverless to the Edge with WebAssembly Runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 140–149. https://doi.org/10.1109/CCGrid54584.2022.00023
[5] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. 2019. Challenges and Opportunities for Efficient Serverless Computing at the Edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 261–266. https://doi.org/10.1109/SRDS47363.2019.00036
[6] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. 2015. Edge-centric Computing: Vision and Challenges. *ACM SIGCOMM Computer Communication Review* 45, 5 (2015), 37–42. https://doi.org/10.1145/2831347.2831354
[7] Google. 2017. "Distroless" Container Images. https://github.com/GoogleContainerTools/distroless. [Online; accessed Sep. 2025.].
[8] WebAssembly Community Group. 2020. WebAssembly System Interface (WASI) Specification. https://github.com/WebAssembly/WASI. Accessed: 2025-03-07.
[9] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jean-François Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. 185–200. https://doi.org/10.1145/3062341.3062363
[10] Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R. Lyu. 2023. Revealing Performance Issues in Server-side WebAssembly Runtimes via Differential Testing. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 663–675. https://doi.org/10.1109/ASE56229.2023.00088
[11] J. Kim and K. Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. https://doi.org/10.1109/CLOUD.2019.00091
[12] Nane Kratzke. 2018. A Brief History of Cloud Application Architectures. *Applied Sciences* 8, 8 (2018), 1368. https://doi.org/10.3390/app8081368
[13] Wei Ling, Lin Ma, Chen Tian, and Ziang Hu. 2019. Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 1416–1421. https://doi.org/10.1109/CSCI49370.2019.00265
[14] Mugeng Liu, Haiyang Shen, Yixuan Zhang, Hong Mei, and Yun Ma. 2025. WebAssembly for Container Runtime: Are We There Yet? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 6, Article 174 (2025), 22 pages. https://doi.org/10.1145/3712197
[15] Joel Scheuner, Simon Eismann, Sacheendra Talluri, Erwin van Eyk, Cristina Abad, Philipp Leitner, and Alexandru Iosup. 2022. Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications. arXiv:2205.07696 [cs.DC] https://arxiv.org/abs/2205.07696
[16] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad
[17] Giuseppe Di Stefano, Vincenzo De Maio, Domenico Vita, Francesco Longo, Giuseppe Anastasi, and Giacomo Cabri. 2023. A Cross-Architecture Evaluation of WebAssembly in the Cloud-Edge Continuum. In *2023 IEEE 9th World Forum on Internet of Things (WF-IoT)*. 1–6. https://doi.org/10.1109/WF-IoT57572.2023.10162204
[18] Nan Wang, Blesson Varghese, Michail Matthaiou, and Dimitrios S. Nikolopoulos. 2020. ENORM: A Framework For Edge NOde Resource Management. *IEEE Transactions on Services Computing* 13, 6 (2020), 1086–1099. https://doi.org/10.1109/TSC.2017.2753775
[19] Jiamin Xu, Weitao Du, Runtian Yu, Wei Wu, Shixiong Zhao, Zhaoguo Wang, and Haibo Chen. 2023. eWAPA: An eBPF-based WASI Performance Analysis Framework for WebAssembly Runtimes. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. ACM, 1–17. https://doi.org/10.1145/3552326.3587457
[20] Shuai Yu, Xiaomin Wang, and Rami Langar. 2021. When Serverless Computing Meets Edge Computing. *IEEE Communications Magazine* 59, 12 (2021), 29–35. https://doi.org/10.1109/MCOM.001.2100262