International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 11-No.4 (2025) pp. 7769-7775
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

Serverless 2.0: Unlocking Performance and Portability with WebAssembly

Satya Teja Muddada*

Independent Researcher, USA

*** Corresponding Author Email:** mastergracemuddada@gmail.com - **ORCID:** 0000-0002-0247-7850

Article Info:

DOI: 10.22399/ijcesen.4130
Received : 20 August 2025
Accepted : 11 October 2025

Keywords

WebAssembly Integration,
Serverless Computing Evolution,
Edge-First Architecture,
Polyglot Runtime Support,
Cold Start Optimization

Abstract:

Serverless computing has transformed the development of applications in the cloud since it abstracts the infrastructure operation, enabling it to scale itself automatically. Legacy Function-as-a-Service offerings have been characterized by severe constraints such as high cold start delay, runtime constraints, language bias, and performance flooding, making them inapplicable in latency-sensitive and distributed applications. WebAssembly has become a groundbreaking platform to overcome these issues by creating a lightweight binary instruction, close-to-native speed execution, and a computer-independent compilation target platform. This article presents the idea of Serverless 2.0, which is an enacted paradigm of incorporating WebAssembly modules in serverless computing to achieve enhanced performance aspects without compromising operational simplicity. The architecture makes polyglot development and deployment easy, as it replaces language-specific execution engines with single-execution engines of WebAssembly. Patterns like edge-first deployment use the WebAssembly small footprint when local distributed computing would be effective, making computation closer to the data sources and end users. The extensive benchmarking can be shown to have made significant gains in various key measures, and cold start latency gains, a higher runtime performance, as well as better resource usage are dwelt upon in the above expounded. The combination is supporting new architectural patterns that were once impractical with traditional serverless frameworks, and should not only be limited to straightforward event-driven processing, but can be intensely real-time, application-latency-compromised, and globally distributed applications. Regardless of the remaining room to improve in the areas of tooling maturity and debugging support, WebAssembly-based serverless computing is an important advancement in cloud-native architecture, with promising performance benefits when it comes to cloud-critical and multi-cloud applications.

1. Introduction

The serverless model has revolutionized cloud application development over the last decade, enabling developers to deploy code without having to worry about the underlying infrastructure. Conventional serverless platforms have provided fast development cycles and automatically scaling features with immense operational overhead reduction. Nevertheless, the platforms come with tremendous limitations that limit applicability for some categories of workloads. A recent study of large serverless platforms identifies cold start latency as a main performance bottleneck, with initialization times differing widely depending on runtime choice and resource provisioning [1]. The work by Kelly et al. studied cold start behavior on several platforms, showing that container initialization and runtime bootstrapping add significant overhead to function execution, especially degrading applications that need uniform low-latency responses. Runtime restrictions further constrain the selection of execution environments and programming languages, and vendor-specific interpretations generate lock-in consequences that make multi-cloud strategies troublesome and expensive.

WebAssembly (WASM) presents itself as a promising technical solution to remedy these inherent core problems of present serverless designs. First created to support web browsers to provide near-native performance for

computationally demanding software, WASM has become a portable, secure, and high-performance runtime that can be used in server-side scenarios. A study by Poltavskyi illustrates how the integration of WASM in performance-critical web applications results in large boosts in execution time without compromising security using sandboxing [2]. The small binary format allows for speedier parsing and compilation over standard JavaScript, with the research demonstrating that WASM modules have considerably lower memory overhead when executed. The language-independent compilation target facilitates support for various programming languages such as Rust, C++, Go, and AssemblyScript, solidifying it as the best fit for future serverless platforms where developer agility and performance optimization are of the utmost importance.

This paper presents Serverless 2.0, a new model that uses WASM modules in serverless frameworks to transcend the limitations of conventional Function-as-a-Service platforms while preserving operational advantages. Inclusion of WASM in serverless computing solves several pain points at once by leveraging architectural advances that minimize initialization overhead and enhance execution efficiency. Cold start times are reduced dramatically through the use of WASM's light runtime initialization process that avoids container bootstrapping latency. The polyglot aspect of WASM enables developers to use optimal languages for given tasks and still have one deployment model, such that teams can tap into available expertise without language limits imposed by the platform. In addition, the portability of WASM facilitates genuine multi-cloud and edge deployment strategies without code change, as compiled modules run identically across disparate runtime environments. This work provides a thorough analysis of serverless computing based on WASM, evincing benefits through real-world deployment use cases and empirical benchmarking that confirm the architectural innovations and performance benefits possible with this method.

2. Architectural Framework and Implementation

The Serverless 2.0 architecture critically redefines how functions are packaged, deployed, and run in cloud infrastructure through the adoption of WebAssembly as a primary execution substrate. Fundamentally, the architecture substitutes language-specific runtimes with a common WASM execution engine that runs modules compiled from a different set of source languages. Decoupling development language from execution environment, this design unlocks serverless deployment flexibility at an unprecedented level without sacrificing performance attributes similar to native code execution. Studies by Jangda et al. show that WASM programs attain execution performance within a factor of 2x of natively compiled code on varied workloads, with some computational kernels running at 1.54x behind native versions when optimized suitably [3]. The research illustrates that compilation techniques have a major effect on performance, where ahead-of-time compilation offers lower execution cost than traditional just-in-time methods used in managed runtimes.

The architectural design includes a number of interrelated elements that facilitate effective WASM execution in serverless environments. The compilation pipeline compiles source code in languages like Rust, C++, Go, or AssemblyScript into WASM modules that are optimized for minimal binary formats by LLVM-based toolchains and have smaller deployment footprints compared to conventional serverless packages while retaining execution performance using linear memory models and controlled control flow. The runtime environment of WASM supports sandboxed execution with hardware-level isolation promises, which guarantees security without the latency overhead of container virtualization that normally introduces large latency into function startup. The orchestration layer is responsible for function lifecycle operations such as instantiation, execution monitoring, and termination, while also supporting legacy serverless triggers and event sources via standard interfaces.

Edge-first deployment patterns are an important innovation in Serverless 2.0 architecture that takes advantage of WASM's lightness for distributed computing contexts. Comparison by Fiasco et al. of WebAssembly and unikernels for edge serverless deployments shows that WASM runtimes have better cold startup performance with less than 1 millisecond initialization time vs. 5-10 milliseconds for lightweight unikernel implementations [4]. The study finds that WASM modules have smaller memory profiles of around 4MB for runtime overhead compared to 8-16MB for similar unikernel deployments, making them possible for higher-density function deployment on hardware-constrained edge nodes. The system accommodates both stateless function execution and stateful workflows by integration with distributed storage services and message queues, which enforces consistency in geographically dispersed deployments. Integration with prevailing cloud services calls for judicious handling of interface boundaries and data serialization mechanisms that maintain performance and ensure compatibility.

The WebAssembly System Interface (WASI) gives us standardized system calls that allow WASM modules to access file systems, networks, and other system resources in a platform-agnostic way without compromising the security guarantees of the sandboxed execution model. This standardization makes it easy to integrate with cloud-native services while retaining the portability benefits of WASM deployment on heterogeneous infrastructure environments. The capability-based security model implemented by WASI blocks unauthorized access to resources while allowing fine-grained permission management better than conventional process-based isolation schemes in security as well as in performance attributes.

3. Experimental Methodology and Performance Evaluation

To empirically confirm the benefits of Serverless 2.0, thorough benchmarking experiments contrasted WASM-based serverless implementations with FaaS systems based on traditional architecture through crosscutting evaluation methods. The experimental setup covered several aspects of performance measurement, such as cold start delay, execution time, memory usage, and scalability behavior under different load types that describe actual deployment use cases. The SPEC Cloud Group's research methodology for FaaS and serverless systems offered guiding principles for setting up reproducible benchmarking methodologies that guarantee equitable comparison across varied platforms [5]. Van Eyk et al. highlight that standardized performance analysis necessitates precise attention to workload properties, measurement granularities, and environmental factors affecting serverless function behavior in varied deployment situations. Research vision captures the need for end-to-end metrics that reflect application-level performance and system-level utilization of resources in order to provide comparisons among next-generation serverless technologies.

The experiment used varied cloud providers and platforms in order to provide full coverage of modern serverless ecosystems. Node.js and Python runtimes were used as traditional serverless baselines, representing common deployment options in production as per runtime adoption numbers. WASM-based implementations used modules that were compiled from Rust, C++, and AssemblyScript with optimization flags that optimize for performance while ensuring binary compatibility. Workload choice involved CPU-bound workloads like recursive Fibonacci calculation and cryptography, I/O-bound workloads like API aggregation and file handling, and blended workloads involving JSON parsing with computationally intensive tasks to assess performance across various application classes. All workload categories reflect unique usage patterns seen in production serverless use cases, allowing for measurement of WASM performance attributes under a variety of computational loads. The approach used strict statistical treatment with repeated trial executions to achieve result reliability and statistical significance. Every function went through large invocation sequences to compute average measurements and percentile distributions that reflect performance variability. Cold start measurements recorded the entire initialization process from receipt of request to execution of the first instruction, including all overhead elements contributing to startup delay. In their recent study, Ebrahimi et al. give a thorough taxonomy of cold start latency reduction mechanisms and list initialization phases that consist of container provisioning, bootstrapping at runtime, and loading of the application as the main culprits behind delays in startup [6]. The research establishes that classical mitigation techniques like pre-warming containers and pooling attain partial effectiveness due to erratic invocation patterns, underpinning the significance of architectural methods that inherently minimize initialization overhead as opposed to trying to cover up latency using prophetic methods.

The deployment for each platform utilized best-practice methods established for production environments to provide for experimental validity. Classic serverless functions were optimized for packaging to reduce deployment size via dependency pruning and tree-shaking optimizations. WASM modules were compiled with aggressive optimization options such as link-time optimization and dead code elimination to generate small binaries. Edge deployments stretched across geographically dispersed regions to measure latency properties for large-scale distributed user bases, with measurement infrastructure placed to record end-to-end response times. All experiments were conducted under controlled environments to eliminate external interference, with network baselines set using initial measurements that ensured stable connectivity and low packet loss on the test infrastructure.

4. Results and Comparative Analysis

Experimental results show significant performance gains for WASM-based serverless deployments on all tested metrics, confirming the architectural benefits of the Serverless 2.0 paradigm. Cold start latency improved most dramatically, with WASM functions starting much faster than conventional serverless platforms owing to light runtime

attributes and streamlined module loading strategies. In-depth analysis by Zhang et al. across several WebAssembly runtimes discloses that contemporary WASM execution engines realize instantiation times from microseconds to low milliseconds based on module complexity and runtime performance [7]. The study states that independently running runtimes like Wasmtime and Wasmer show better cold start performance than runtimes embedded into their host processes, with linear memory initialization and table setup adding negligible overhead to the startup paths. These results are in agreement with experimental evidence of WASM-based serverless functions maintaining consistent sub-second response times even at cold start scenarios.

Performance analysis of execution showed that WASM modules provide similar or better runtime performance in spite of running in sandboxed environments that impose strong security boundaries. CPU-bound workloads exhibited cost-efficient execution behavior on WASM platforms, with instruction dispatch and register allocation techniques optimized to reduce interpretation overhead. Memory usage patterns exhibited consistent benefits for WASM deployments, with linear memory models providing predictable allocation behavior and less fragmentation than garbage-collected runtimes. The enhanced memory efficiency allows for increased function density per host, enabling infrastructure providers to attain improved resource utilization and lower operating expenses in multi-tenant deployment environments.

Concurrent load scalability testing brought out WASM's higher throughput-like behavior and predictable degradation patterns. WASM functions supported larger request rates without compromising on acceptable response time distributions, indicating improved resource utilization and more predictable scaling under conditions of stress. Binary size comparisons showed significant decreases in deployment package sizes, and WASM modules achieved small representations by using efficient bytecode encoding and removal of runtime dependencies. Ray's work offers an extensive analysis of WebAssembly usage in IoT environments and illustrates that WASM modules generally obtain 50-80% size savings compared to their equivalent JavaScript counterparts while achieving functional equivalence [8]. The research highlights that binary compactness has a direct effect on quicker transmission times and less storage in resource-limited environments, benefits which directly apply to serverless deployment scenarios where package size has a direct effect on cold start latency and storage expenses.

Geographic distribution testing of edge deployments revealed remarkable performance properties that confirm WASM compatibility with distributed serverless architectures. WASM applications deployed at edge sites had low response times for regional-proximity users that far exceeded centralized, latency-penalized traditional serverless deployments. Universality of performance across multiple edge providers confirms the portability benefits of WASM modules, which run the same across diversified runtime environments without platform-specific translation. Performance measurements on various geographic regions verified that WASM-based edge functions provide reliable latency profiles that make real-time applications that were previously not well-suited for serverless deployment models. Such findings establish WASM as a groundbreaking technology for serverless computing, and especially for applications that need consistent low-latency outputs and global distribution.

5. Discussion and Implications

Experimental results confirm the transformative value of WASM integration in serverless computing to establish Serverless 2.0 as a workable next-generation FaaS evolution, overcoming inherent architectural constraints. The combination of lower cold start latency, better execution performance, and improved portability overcomes key constraints that have traditionally curbed serverless adoption for specific categories of applications. A study by Cisco researchers Eismann et al. on serverless application patterns pinpoints certain situations where current FaaS offerings fall short, viz., applications needing certain levels of predictable latency, function composition with fine granularity, and deployment flexibility across platforms [9]. Developers are found to avoid serverless in workloads sensitive to latency due to the uncertainty of cold start penalties, which is mostly removed by WASM-based systems due to lightweight runtime booting. Such improvements enable new styles and applications of architecture that could previously have been considered impractical with traditional serverless implementations and widen the application of FaaS to event processing beyond simple event-driven processing.

The impact on application architecture is significant; it indeed transforms the way developers design distributed systems and deploy them. The sensational cold start latency decrease eliminates one key obstacle to microservice decomposition on a fine level, allowing developers to build more

modular and manageable systems without the performance overhead traditionally linked to function proliferation. WASM's polyglot nature enables development teams to use the best programming languages for individual components without deployment overhead, supporting best-of-breed development patterns that deliver the most developer productivity and code efficiency. Edge deployment capability fundamentally shifts the economics and performance properties of globally distributed applications, with computation brought closer to data sources and end users while preserving operational ease.

In spite of these benefits, there are a number of challenges for production uptake in the enterprise that need to be given proper consideration. Tooling and debugging support for WASM within serverless platforms is less mature compared to traditional platforms, potentially elevating development complexity and operational burden. Recent benchmarking research by Baek et al. introduces the Wasm-R3 framework for realistic WebAssembly performance evaluation, revealing that current profiling tools lack the granularity necessary for comprehensive performance analysis in production deployments [10]. The study demonstrates that execution characteristics vary significantly based on workload patterns, with memory-intensive applications showing different optimization requirements than compute-bound tasks, necessitating sophisticated tooling for performance tuning. Security issues, though handled by sandboxing technology, are subject to continued monitoring with changing attack surfaces and threat scenarios as adoption of WASM grows.

Economic impacts of Serverless 2.0 need thoughtful examination, both from the provider and consumer sides. Lowered memory usage and enhanced throughput directly equate to cost benefits in consumption-tiered pricing mechanisms, with likely benefits in reduced operational costs for high-volume software. The support for deploying the same WASM modules on a variety of cloud providers allows for advanced cost optimization techniques and minimizes vendor lock-in risks that otherwise limit architectural choices. Arbitrage across a multi-cloud deployment can be realised by organisations routing workloads to the least-cost platforms dynamically, whilst maintaining the same performance properties at the delivery endpoints. The adoption pace in the future is probably more gradual, where first adopters will explore greenfield deployments and more niche applications, which can most directly benefit from WASM advantages and slowly upgrade towards systems that take on progressively greater value from the ecosystem.

Main body of manuscript should be written using

Table 1: *Cold Start Latency Performance Metrics [1, 2]*

Performance Metric	Value/Description
Container initialization overhead	Substantial contributor to the function execution delays
JavaScript vs WASM parsing speed	WASM enables faster parsing and compilation
WASM memory overhead	Substantially less than traditional JavaScript
Supported programming languages	Rust, C++, Go, AssemblyScript
Platform migration complexity	Vendor lock-in increases migration costs
Runtime constraint impact	Limits programming language choices
Deployment model characteristics	Unified model without platform restrictions

Table 2: *WASM Runtime Performance and Resource Utilization Metrics [3,4]*

Architecture Component	Specification/ Performance
WASM vs native code execution speed	Within a factor of 2x
WASM cold start initialization	Under 1 millisecond
Unikernel cold start initialization	5-10 milliseconds
WASM runtime memory overhead	Approximately 4MB
Unikernel memory overhead	8-16MB

Supported source languages	Rust, C++, Go

Table 3: *Benchmarking Protocol and Workload Characteristics for Serverless Evaluation [5,6]*

Experimental Parameter	Specification/Details
Performance dimensions evaluated	Cold start, execution time, memory, scalability
Traditional runtime baselines	Node.js and Python runtimes
WASM compilation sources	Rust, C++, AssemblyScript
Workload categories	CPU-intensive, I/O-bound, mixed workloads
Container provisioning phases	Primary contributor to startup delays
Application loading overhead	Key initialization phase contributor
Pre-warming strategy effectiveness	Limited due to unpredictable invocation
Optimization techniques	Link-time optimization, dead code elimination

Table 4: *Considerations and implications for WebAssembly adoption in production serverless environments*

Implementation Aspect	Characteristic/Impact
Problematic FaaS scenarios	Fine-grained composition, cross-platform deployment
Developer avoidance reason	Unpredictable cold start penalties
WASM mitigation approach	Lightweight runtime initialization
Wasm-R3 framework purpose	Realistic WebAssembly performance evaluation
Security mechanism	Sandboxing with ongoing vigilance required
Cost optimization strategy	Multi-cloud workload routing enabled

[9,10]

4. Conclusions

Introducing WebAssembly in serverless computing frameworks constitutes an epigenetic improvement in the development of cloud-native applications, eliminating persistent constraints that had historically curtailed the use of Function-as-a-Service in workloads requiring high performance. The Serverless 2.0 paradigm showcases that a combination of the lightweight runtime nature of WebAssembly, along with its serverless paradigms of operation, presents strong benefits in a variety of facets. Latency to cold start with hairsplitting. Latency to cold start without performance degradation. Latency Hairsplitting Finer-grained microservice decomposition Fine-grained microservice decomposition best for individual components Polyglot Capabilities Select the optimal language to use in a particular component. Edge deployment patterns change the economics and performance properties of globally distributed applications, enabling real-time interactions, which have not been feasible using serverless architectures, to be feasible. The innovations in architecture go beyond mere enhancements and define additional avenues of application design and deployment strategies. Although tooling sophistication and debugging capabilities continue to be challenging, the move toward more comprehensive ecosystem maturity is evident. WebAssembly-based servers should be considered by organizations working with processes related to WebAssembly, the cases of edge computing, and those where cross-platform portability is needed. The financial aspect of promoting resource use and the versatility of multi-cloud offers further reasons to consider adopting. With the maturity of tool ecosystems and increased standardization movements, WebAssembly-powered serverless computing is poised to be the new favorite of MN, but more cloud-native with high consistency, distributed availability, and operational wholeness.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.

- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

[1] Daniel Kelly et al., "Serverless Computing: Behind the Scenes of Major Platforms", arXiv, 2020. [Online]. Available: https://arxiv.org/pdf/2012.05600

[2] Poltavskyi Dmytro, "Integration of WebAssembly in Performance-critical Web Applications", American Academic Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS), May 2025. [Online]. Available: https://asrjetsjournal.org/American_Scientific_Journal/article/view/11676/2837

[3] Abhinav Jangda et al., "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code", arXiv, 2019. [Online]. Available: https://arxiv.org/pdf/1901.09056

[4] Enrico Fiasco et al., "WebAssembly and Unikernels: A Comparative Study for Serverless at the Edge", arXiv, 11th Sept 2025. [Online]. Available: https://arxiv.org/html/2509.09400v1

[5] Erwin van Eyk et al., "The SPEC cloud group's research vision on FaaS and serverless architectures", ResearchGate, 2017. [Online]. Available: https://www.researchgate.net/publication/321065955_The_SPEC_cloud_group's_research_vision_on_FaaS_and_serverless_architectures

[6] Ana Ebrahimi et al., "Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions", ScienceDirect, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1383762124000523

[7] Yixuan Zhang et al., "Research on WebAssembly Runtimes: A Survey", arXiv, 2024. [Online]. Available: https://arxiv.org/html/2404.12621v1

[8] Partha Pratim Ray, "An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions", MDPI, 2023. [Online]. Available: https://www.mdpi.com/1999-5903/15/8/275

[9] Simon Eismann et al., "Serverless Applications: Why, When, and How?", arXiv, 2020. [Online]. Available: https://arxiv.org/pdf/2009.08173

[10] Doehyun Baek et al., "Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks", arXiv, 2024. [Online]. Available: https://arxiv.org/pdf/2409.00708