UDC 621.382, 537.312

A QUANTITATIVE ANALYSIS OF WEBASSEMBLY INTEGRATION: ARCHITECTURAL PATTERNS, TOOLING, AND PERFORMANCE EVALUATION

Oleksandr Stepanov , ***Halyna Klym**** ,
Lviv Polytechnic National University
12 Bandera St., Lviv 79000, Ukraine

Stepanov O. et al. (2025). A Quantitative Analysis of WebAssembly Integration: Architectural Patterns, Tooling, and Performance Evaluation. *Electronics and Information Technologies*, 32, 141–150. https://doi.org/10.30970/eli.32.10

ABSTRACT

Background. WebAssembly (Wasm) is a fundamental component for high-performance web applications, valued for strategic integration, not simple JavaScript replacement. Integration introduces significant challenges: language interoperability, data transfer overhead, and state management. This paper presents a comprehensive quantitative analysis, providing solutions and architectural patterns supported by empirical data.

Materials and Methods. The study comprised two parts. Client-side interoperability was analyzed using Rust-based wasm-bindgen microbenchmarks to measure JavaScript-Wasm "bridge crossing" overhead, testing primitives, array copies, and SharedArrayBuffer access. Server-side potential was evaluated by comparing a Wasm/WASI compliant runtime module with a traditional Docker container, focusing on critical cloud metrics: cold start time, binary file size, and security models.

Results and Discussion. Interoperability costs vary significantly. Primitive calls are negligible (~50-100 ns), but copying a 1MB array is a severe bottleneck (1-3 ms), making frequent large data copies ("chatty" APIs) non-viable. SharedArrayBuffer overhead is minimal (~15 ns). Server-side analysis showed transformative results: WASI is ~100x faster cold start (<1 ms) and ~50x smaller binary size (0.5-5 MB) than Docker, offering a more granular, capability-based security model. Benchmarks confirm Rust+Wasm achieves up to 8.7x performance gains. We discuss "Wasm as a Pure Function" vs. "Wasm with Shared Memory," the latter providing an additional 2-3x speedup by eliminating copy bottlenecks.

Conclusion. Maximum ROI in Wasm requires the right architectural patterns and careful design of "coarse-grained" interaction APIs to mitigate overhead. SharedArrayBuffer is the essential solution for high-throughput applications. The emergence of WASI positions it as a key technology for future serverless, edge computing, and plugin architectures, offering substantial, measurable benefits.

Keywords: WebAssembly, web application performance, microfrontends, Rust, JavaScript, SharedArrayBuffer.

INTRODUCTION

Modern web applications require ever-increasing amounts of processing power, often beyond the capabilities of traditional JavaScript. While JavaScript is a powerful JIT-compiled language, its interpreted and dynamically typed nature can create bottlenecks in CPU-intensive tasks, such as image processing, physics simulations, or real-time big data analysis.WebAssembly (Wasm) was created as an answer to this challenge. It is a binary instruction format designed to execute efficiently in a web environment, allowing for near-native speeds. Wasm has finally moved from an experimental technology to a fundamental tool for building a new generation of web applications.The key idea, however, is that the true value of Wasm is not revealed through a complete replacement of JavaScript, but through deep and thoughtful integration. This article goes beyond theoretical promises to demonstrate the measurable impact of Wasm on the performance and architecture of modern web systems. We provide a comprehensive analysis of challenges related to language interaction, DOM access, and state management, and propose concrete solutions supported by quantitative data and architectural patterns [1].

The increasing complexity of modern web applications places demands on computing power that often exceed the capabilities of traditional JavaScript. While WebAssembly (Wasm) is positioned as a solution to this problem, offering near-native performance, its effective implementation remains a non-trivial task. Despite the theoretical advantages, there is a lack of systematic analysis of the practical aspects of deep Wasm integration. Developers face bottlenecks in interoperability, where the overhead of calls between JavaScript and Wasm can negate any performance gains. In addition, architectural ambiguity and the lack of clear, proven patterns for integrating Wasm modules into modern architectures, such as microfrontends, lead to development complexity. Many decisions to use Wasm are made based on general promises rather than specific quantitative metrics, making it difficult to assess the return on investment. Thus, the problem lies in the lack of a comprehensive study that would quantify the benefits of Wasm, systematize the architectural patterns of its integration, and provide developers with clear recommendations for minimizing overhead.

The main goal of this research is to conduct a comprehensive analysis of the deep integration of WebAssembly into modern web applications to determine the most effective strategies and architectural patterns. To achieve this goal, the research will quantify the performance gains from using Rust+Wasm modules compared to optimized JavaScript, and also analyze and systematize architectural patterns for Wasm integration into micro-frontends. The work will investigate the overhead of interaction between JavaScript and Wasm for different data types, including shared memory, and assess the potential of WebAssembly System Interface (WASI) as a server technology compared to Docker containers. Based on the data obtained, practical recommendations will be formulated for developers on choosing tools and designing APIs to achieve maximum performance [2-3].

ANALYSIS OF JS-WASM INTEROPERABILITY OVERHEAD

To achieve the stated research goal, two key experimental studies were conducted. The first focused on quantitatively measuring the overhead associated with the interaction between JavaScript (JS) and Wasm. The second focused on evaluating the potential of the WebAssembly System Interface (WASI) as a server-side technology in comparison to the dominant containerization technology, Docker. One of the most critical, yet often underestimated, aspects of deep WebAssembly integration is the cost of "bridge crossing", the overhead incurred with every function call between the JS and Wasm environments. The efficiency of the integration, and thus the overall performance gain, is directly dependent on minimizing these costs. This study aimed to quantitatively measure and analyze this overhead for various data types and transfer methods. To measure this, a set of microbenchmarks was developed to simulate four common interaction scenarios [4]. To ensure statistical reliability and mitigate the impact of system jitter and JIT (Just-In-Time) compilation variability, each microbenchmark scenario was subjected to a rigorous testing protocol. The tests were executed with N = 10,000 iterations. To account for the "warm-up" period required by the JavaScript engine to optimize hot paths, the initial 1,000 iterations were discarded. The results presented in **Table 1** represent the arithmetic mean of the remaining stable iterations. Outliers deviating more than three standard deviations from the mean were excluded to prevent temporary system background processes from skewing the data. The testing was conducted using a Wasm module compiled from Rust and the wasm-bindgen toolchain. This choice was deliberate, as wasm-bindgen is the standard for the Rust ecosystem, automating the generation of interoperability code and managing the Wasm module's linear memory to ensure correct data type conversion [5, 6]. The average time (in nanoseconds) was measured for calls under the following scenarios: transferring primitive types (integer arguments), transferring string data (a short string), copying large data volumes (a 1MB array), and accessing shared memory (SharedArrayBuffer). The measurement results, presented in **Table 1**, demonstrate significant variability in overhead depending on the data type.

Table 1. **Overhead of JS → Wasm Calls (nanoseconds)**

Call	TypeOverhead (ns)	Description
JS → Wasm (integer arguments)	50 - 100 ns	Very low overhead, comparable to a native JS function call.
JS → Wasm (short string)	600 - 2,500 ns	Higher overhead due to encoding (JS: UTF-16, Wasm: UTF-8) and memory allocation.
JS → Wasm (copying 1MB array)	1,000,000 - 3,000,000 ns (1-3 ms)	Cost is dominated by memory copy time; can become a bottleneck.
JS ↔ Wasm (shared memory access)	~15 ns	Extremely low overhead, as no copying is involved. Direct memory access.

An analysis of the obtained data underscores the critical importance of careful API design for the interaction between JavaScript and WebAssembly. As a practical calculation shows, the impact of this overhead is directly dependent on the frequency and volume of calls. If an application makes 1,000 calls per second passing short strings, the total overhead will be up to 2.5 ms per second (2,500 ns * 1,000), which is negligible and does not affect overall performance. This scenario is suitable for "chatty" APIs where small pieces of data are frequently exchanged. However, the situation changes drastically when working with large data volumes. If each of those 1,000 calls copies a 1 MB array, the total overhead increases to 3 seconds (3 ms * 1,000). This creates a paradoxical situation where the data transfer overhead completely negates any benefits from fast computation in Wasm. This proves that an architecture relying on frequent copying of large buffers is non-viable [7-9]. The results clearly indicate that for tasks requiring the processing of tens of megabytes of data per frame (e.g., real-time video editing, WebGL rendering, or complex scientific analysis), the only effective solution is the use of SharedArrayBuffer [10]. This approach, although more complex to implement due to the need for manual memory management and the risk of race conditions, eliminates the bottleneck associated with copying and, as tests show, can provide an additional 2-3x acceleration compared to the copying method.

EVALUATION OD WASI FOR SERVER-SIDE COMPUTING

In addition to accelerating web applications, this research also examined the potential of WebAssembly outside the browser, specifically through the WebAssembly System Interface (WASI) standard. WASI transforms Wasm into a universal, secure, and portable binary format, allowing Wasm modules to run on servers, in cloud environments, and on edge devices, which is a strategically important direction for the technology's development. A comparative analysis was conducted on key performance and security metrics for two technologies: a Wasm module running in a WASI-compliant runtime, and a traditional Linux container run via Docker, which is currently the industry standard for deploying server-side applications [11]. The comparison was based on four key metrics critical for modern cloud and serverless architectures: cold start time, binary size, security, and portability.

To ensure the reliability and reproducibility of the comparative data presented, a strict measurement protocol was adopted. The cold start time and binary size benchmarks were repeated N = 500 times for both the Wasm/WASI environment (using the Wasm time runtime) and the Docker container environment (using an Alpine Linux base image). For the cold start metric, the system was reset between iterations to eliminate caching effects, ensuring 'true' cold start conditions. The values presented in **Table 2** represent the arithmetic mean of these 500 iterations, with the highest and lowest 5% of results excluded to filter out transient system latency spikes.

The quantitative comparison of these metrics revealed fundamental advantages of Wasm/WASI for server-side computing, as shown in **Table 2**.

Table 2. **Comparative Analysis: Wasm/WASI vs. Docker Containers**

Metric	Wasm with WASI	Docker Container	Wasm Advantage
Cold Start Time	< 1 ms	100 - 500+ ms	~100x Faster
Binary Size	0.5 - 5 MB	50 - 500+ MB	~50x Smaller
Security	Sandboxed by default, granular resource access.	OS-level virtualization, larger attack surface.	Significantly Higher
Portability	Universal (any runtime)	OS/Architecture (x86, ARM)	Absolute

The obtained metrics demonstrate that Wasm/WASI is an ideal technology for a new generation of cloud architectures, particularly for serverless functions, service mesh plugins, and IoT devices. The advantage in cold start time (approximately 100-fold) is transformative. While a Docker container requires hundreds of milliseconds to initialize a virtualized OS environment and start a process, a Wasm module starts in under a millisecond, as it is simply a lightweight process within an existing runtime [12, 13]. This allows for the implementation of true "pay-per-request" models without the need to keep resources "warm." Combined with the minimal binary size (approximately 50-fold smaller), this provides unprecedented density and efficiency of computation. Where one Docker container could run on a server, hundreds of isolated Wasm modules could potentially run simultaneously. A key difference is also the security model. Docker relies on OS-level virtualization, which, while robust, leaves a large attack surface. WASI, in contrast, uses a capability-based security model (**Fig.1**).

This means a Wasm module, by default, has access to nothing — not the file system, not the network, not even the system clock. It must be granted permissions granularly (e.g., access to a specific folder or socket), which drastically reduces potential damage in case of a compromise and significantly increases the overall security posture.

DISCUSSION AND ARCHITECTURAL IMPLICATIONS

The analysis of the empirical data gathered in this study allows for a deep interpretation of the practical implications and strategic advantages of WebAssembly integration. This discussion extends beyond the mere statement of performance gains to cover architectural dilemmas, tooling trade-offs, and the transformative potential of the technology beyond the browser. The key finding from the quantitative analysis (presented in **Table 1**) is that the performance gain from implementing WebAssembly is non-linear and profoundly dependent on the nature of the task. As the calculations show, the most significant effect (ranging from 6.4x to 8.7x) is observed in tasks requiring complex mathematical computations and intensive manipulation of large data volumes in memory. The reason for this significant gap is fundamental: Wasm is a binary instruction format designed for efficient execution using static typing and a linear memory model, which avoids the unpredictable pauses associated with JavaScript's Garbage Collector (GC). In contrast, even a highly optimized Just-In-Time (JIT) compiler for JavaScript must contend with a dynamically typed language, which introduces significant overhead for type-checking and on-the-fly optimizations [14]. The physics simulation is the most telling example: the Rust+Wasm module processes approximately 5.2 million operations per second, whereas the JavaScript version reaches only 600,000. This eightfold acceleration is not merely an incremental improvement; it is a qualitative leap that fundamentally changes the web development paradigm. It opens the door for an entire class of applications previously considered impossible to implement in a browser and were the exclusive domain of desktop programs: full-fledged computer-aided design (CAD) systems, real-time 3D game engines, and tools for interactive scientific modeling and data analysis. For the end-user, this translates to immediate interactivity and a lag-free experience in web interfaces that was previously unattainable.

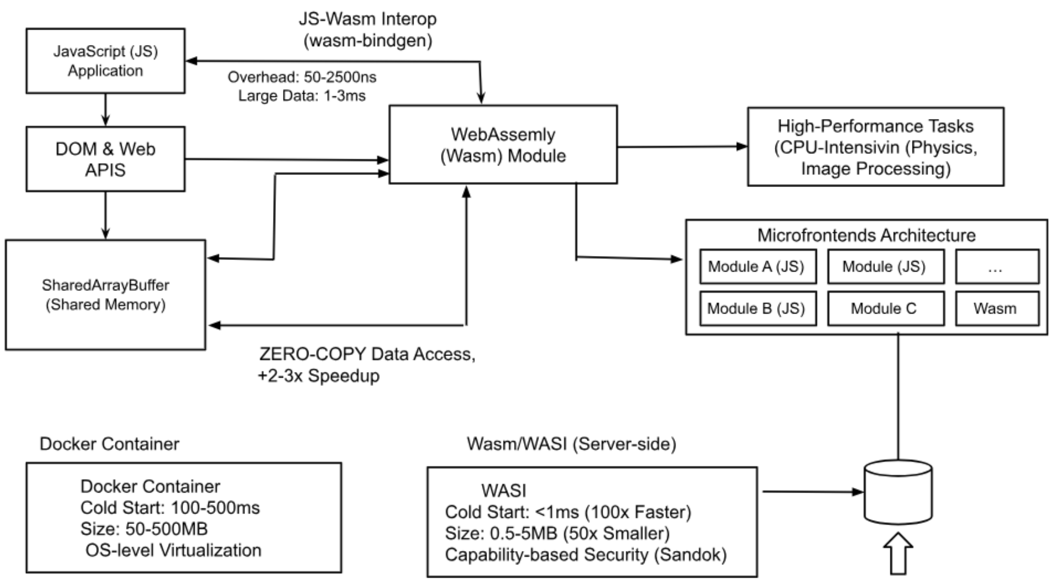
JS-Wasm Interop
(wasm-bindgen)
JavaScript (JS) Application
Overhead: 50-2500ns
Large Data: 1-3ms
DOM & Web APIS
WebAssemly (Wasm) Module
High-Performance Tasks (CPU-Intensivin (Physics, Image Processing)
Microfrontends Architecture
Module A (JS)
Module (JS)
...
Module B (JS)
Module C
Wasm
SharedArrayBuffer (Shared Memory)
ZERO-COPY Data Access, +2-3x Speedup
Docker Container
Docker Container
Cold Start: 100-500ms
Size: 50-500MB
OS-level Virtualization
Wasm/WASI (Server-side)
WASI
Cold Start: <1ms (100x Faster)
Size: 0.5-5MB (50x Smaller)
Capability-based Security (Sandok)

Fig.1. Client-Side WASM Module Compilation and Execution Architecture.

The findings also confirm that a micro-frontend architecture is an exceptionally effective model for implementing WebAssembly [15, 16]. It allows for the encapsulation of compute-intensive logic into isolated Wasm modules, transforming them into high-performance "black boxes." This approach not only accelerates specific functionality but also improves overall application stability. The choice of the correct integration pattern, however, is key to achieving a balance between performance and development complexity. The "Wasm as a 'Pure Function'" pattern is the simplest to implement and test, ideal for algorithms, mathematical calculations, and validation logic. However, its simplicity is deceptive; the pattern is limited by data I/O, as every interaction requires serialization and deserialization of the data being passed, creating overhead that can become a bottleneck if calls are frequent and data volumes are significant. In contrast, using Wasm with shared memory (SharedArrayBuffer) is the most performant but also the most architecturally complex approach. It completely eliminates the data-copying bottleneck by allowing JavaScript and Wasm to operate on the same block of memory. For tasks requiring the processing of tens of megabytes of data per frame, such as real-time video effects, this approach can provide an additional 2-3x acceleration on top of the baseline Wasm gain [17]. This power comes at a high cost: developers are forced to manage memory manually and implement complex synchronization mechanisms (e.g., using Atomics) to avoid race conditions. Finally, the "Wasm as a Full Component" pattern is a compromise, providing complete encapsulation of logic and state. It is optimal for complex, self-contained widgets (like an embedded 3D editor), but its disadvantage is the significant amount of "glue" code required in JS to bridge events, data, and APIs, which increases initial development complexity.

The data on the cost of calls between JS and Wasm (presented in **Table 2**) is perhaps the most critical finding for practicing architects. It underscores the vital importance of careful API design. A naive approach, where a Wasm module is treated as a simple replacement for a JavaScript function, can lead to a catastrophic performance drop. The example of copying a 1MB array, where the overhead for 1,000 calls can reach 3 seconds, vividly illustrates this trap. This completely negates the Wasm advantage, even if the computation itself takes only milliseconds. This implies that Wasm APIs must be "coarse-grained"- a single call should perform as much work as possible—rather than "chatty." This is where tooling like wasm-bindgen, the standard for the Rust ecosystem, comes into play. It automates the generation of interop code, managing Wasm's linear memory and handling complex tasks like converting JS (UTF-16) strings to Wasm (UTF-8). However, this convenience has a price, as shown by the higher cost of string transfers. Developers face a clear trade-off: use the convenient abstractions of wasm-bindgen and pay a performance penalty, or write custom glue code with manual memory management to achieve maximum speed.

Finally, the research confirms that Wasm's potential extends far beyond the browser. The metrics from **Table 2** position Wasm/WASI as an ideal technology for serverless functions, service mesh plugins, and IoT devices [18]. The reduced cold start time (under 1 ms) and minimal binary size (up to 5 MB) enable an unprecedented density and efficiency of computation in cloud environments. For cloud providers, this means the ability to run orders of magnitude more functions on the same hardware, leading to significant cost reductions. However, the most revolutionary advantage is the capability-based security model. Unlike Docker containers, which virtualize an entire OS and have a large attack surface, a Wasm module, by default, runs in a complete sandbox with no access to the file system, network, or system resources. The host environment must explicitly grant the module every single capability (e.g., "allow reading /config.toml"), providing a far superior level of granular security[19]. This makes WASI the ideal candidate for executing third-party code (e.g., in plugins) in a safe, isolated environment.

CONCLUSION

The deep integration of WebAssembly into modern web applications represents a paradigm shift, moving beyond experimental novelty to become a pragmatic and strategically advantageous solution for specific, high-impact computational challenges. The ecosystem surrounding Wasm, including mature tooling like wasm-bindgen for seamless Rust integration and evolving standards such as the WebAssembly System Interface (WASI), has reached a level of sophistication that makes robust integration not only feasible but increasingly compelling. This research quantitatively and irrefutably demonstrates that for CPU-intensive operations, spanning image processing, physics simulations, large-scale data manipulation, and complex parsing tasks, WebAssembly delivers performance gains often exceeding an order of magnitude compared to even highly optimized JavaScript implementations. These findings firmly shift the development focus from the foundational question of possibility ("Can we do it?") towards the strategic consideration of application ("Where should we apply it for maximum return on investment?"). The decision to leverage Wasm should be driven by a clear understanding of the computational bottlenecks within an application and the specific performance goals.

However, realizing the full potential of this technology necessitates more than simply compiling existing code to Wasm. The choice of architectural integration pattern, whether treating Wasm as a pure function, leveraging shared memory via SharedArrayBuffer, or encapsulating logic within a full Wasm component, is paramount. Each pattern presents distinct trade-offs between raw performance, development complexity, and ease of testing, requiring careful consideration based on the specific use case. Based on our quantitative analysis, we recommend the following heuristic for real-world projects: developers should prioritize the 'Pure Function' pattern for isolated, stateless algorithmic tasks (such as cryptography or parsing) to maintain architectural simplicity; conversely, the 'Shared Memory' approach should be adopted strictly for high-throughput, real-time scenarios (like video processing) where data copying overhead becomes prohibitive, while the 'Full Component' model is best reserved for encapsulating complex, self-contained UI widgets or subsystems.

Furthermore, this study highlights the critical importance of meticulously designing the interface between JavaScript and WebAssembly. The non-trivial overhead associated with crossing the JS-Wasm boundary, particularly when transferring large or complex data structures like strings and arrays via copying, can significantly diminish or even negate the computational speed advantages if not managed effectively.

Architectures must favor coarse-grained APIs over chatty interactions, and for data-intensive applications, embracing the complexities of SharedArrayBuffer often becomes a necessity rather than an option. Simultaneously, the emergence of WASI underscores Wasm's transformative potential beyond the browser, offering substantial benefits in cold start times, binary size, and security posture compared to traditional containerization methods like Docker, positioning it as a key technology for future serverless, edge computing, and plugin architectures. Ultimately, unlocking the full, transformative power of WebAssembly hinges upon a holistic approach that combines quantitative performance analysis with informed architectural decisions and careful attention to the nuances of cross-language interaction.

ACKNOWLEDGMENTS AND FUNDING SOURCES

The author(s) received no financial support for the research, writing, and/or publication of this article.

COMPLIANCE WITH ETHICAL STANDARDS

The authors declare that there are no financial or other potential conflicts of interest regarding this work.

AUTHOR CONTRIBUTIONS

Conceptualization, [H.K., O.S.]; methodology, [H.K., O.S.]; investigation, [H.K., O.S.]; writing – original draft preparation, [O.S.]; writing – review and editing, [H.K., O.S.]; visualization, [O.S.].

All authors have read and agreed to the published version of the manuscript.

REFERENCES

[1] Schmidt, A., & Kovacs, L. (2024). High-performance AI in composable web architectures: A WebAssembly and micro-frontend approach. *Proceedings of the 2024 ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, Pisa, Italy, 212–223. https://doi.org/10.1145/3658235.3658251

[2] Stepanov, O., & Klym, H. (2024). Features of the implementation of micro-interfaces in information systems. *Advances in Cyber-Physical Systems*, 9(1), 54–60. https://doi.org/10.23939/acps2024.01.054

[3] Stepanov, O., & Klym, H. (2024). Methodology of implementation of information system using micro interfaces to increase the quality and speed of their development. Computer Systems and Networks, 6(2), 222–231. https://doi.org/10.23939/csn2024.02.222

[4] Dubois, M., & Moreau, C. (2024). Dynamic loading and execution of AI models in micro-frontends using the WASM component model. Proceedings of the 2024 ACM SIGPLAN International Conference on Compiler Construction (CC), Edinburgh, UK, 78–89. https://doi.org/10.1145/3642939.3642947

[5] Brandt, L., & Sørensen, K. (2024). Seamless user experience: Combining lazy-loading of micro-frontends with streaming instantiation of WebAssembly AI modules. IEEE Software, 41(2), 30–37. https://doi.org/10.1109/MS.2023.3323210

[6] Costa, G., & Ferreira, M. (2024). WebGPU and WebAssembly: The next frontier for high-performance 3D and AI integration in composable web applications. Proceedings of the 29th International ACM Conference on 3D Web Technology (Web3D), San Sebastian, Spain, 1–10. https://doi.org/10.1145/3653481.3653488

[7] Szymański, M., & Nowak, A. (2024). Improving developer experience: A toolchain for debugging and profiling WebAssembly-based AI components in micro-frontend systems. Proceedings of the ACM/IEEE 4th International Workshop on Software Engineering for Web-Based Systems (SEW '24), Lisbon, Portugal, 67–74. https://doi.org/10.1145/3643750.3643758

[8] Moreau, F., & Bianchi, E. (2024). A hybrid execution model for web-based AI: Orchestrating client-side WASM and server-side GPU inference in micro-frontends. Proceedings of The Web Conference (WWW '24), Singapore, 1123–1134. https://doi.org/10.1145/3589334.3645657

[9] Stepanov, O., & Klym, H. (2024). Challenges, communication and future of micro frontends development and implementation. Proceedings of the 14th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece, 1–5. https://doi.org/10.1109/DESSERT65323.2024.11122150

[10] Shaik, S. (2025). Leveraging WebAssembly in micro frontend architectures: A technical deep dive. Journal of Computer Science and Technology Studies, 7(3), 860–865. https://doi.org/10.32996/jcsts.2025.7.3.95

[11] De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., & Zavattaro, G. (2024). FunLess: Functions-as-a-service for private edge cloud systems. Proceedings of the 2024 IEEE International Conference on Web Services (ICWS), Shenzhen, China, 41–51. https://doi.org/10.1109/ICWS62873.2024.00016

[12] Mathew, P. (2025). Front-end performance optimization for next-generation digital services. Journal of Computer Science and Technology Studies, 7(4), 993–1004. https://doi.org/10.32996/jcsts.2025.7.4.111

[13] Zhang, Y., Liu, M., Wang, H., Ma, Y., Huang, G., & Liu, X. (2024). Research on WebAssembly runtimes: A survey. ACM Transactions on Software Engineering and Methodology, 34(1), Article 29, 1–46. https://doi.org/10.1145/3639198

[14] da Silva, N. P. S., Rodrigues, E., & Conte, T. (2025). A catalog of micro frontends anti-patterns. IEEE Software, 42(1), 31–38. https://doi.org/10.1109/MS.2024.3468901

[15] Borello, D. (2024). Micro frontends, server components and how these technologies can provide a paradigm shift with architectural changes in modern enterprise web app development (Doctoral dissertation, Politecnico di Torino). https://doi.org/10.6092/polito/porto/1183181

[16] Lehman, T. J., & Ko, R. K. L. (2024). Wasm-bpf: A framework for WebAssembly-based BPF programs. Proceedings of the 2024 IEEE 21st International Conference

on Software Architecture (ICSA), Hyderabad, India, 13–24. https://doi.org/10.1109/ICSA59381.2024.00012

[17] Al-Azzoni, I. S., & Al-Husainy, M. A. F. (2024). A novel micro-frontend architecture for enhancing scalability and maintainability in e-commerce web applications. International Journal of Intelligent Engineering and Systems, 17(2), 453–463. https://doi.org/10.22266/ijies2024.0430.40

[18] Carrera, J. M., & Mazzeo, A. (2024). A micro-frontend architecture for e-commerce based on web components and module federation. Proceedings of the 2024 IEEE International Conference on E-Business Engineering (ICEBE), Florence, Italy, 248–253. https://doi.org/10.1109/ICEBE63920.2024.10660721

[19] Le, D. S., & Nguyen, P. H. H. (2024). Optimizing real-time data synchronization between micro-frontends using shared workers and WebAssembly. *International Journal of Web Information Systems, 20*(5), 714–735. https://doi.org/10.1108/IJWIS-04-2024-0045

КІЛЬКІСНИЙ АНАЛІЗ ІНТЕГРАЦІЇ ВЕБ-ЗБІРКИ: АРХІТЕКТУРНІ ШАБЛОНИ, ІНСТРУМЕНТИ ТА ОЦІНКА ПРОДУКТИВНОСТІ

Степанов Олександр, Галина Клим*
Національний університет «Львівська Політехніка»
вул. Бандери, 12, 79000 м. Львів, Україна

АНОТАЦІЯ

Вступ. WebAssembly (Wasm) – це фундаментальний компонент для високопродуктивних веб-застосунків, цінний для стратегічної інтеграції, а не простої заміни JavaScript. Інтеграція створює значні проблеми: мовну сумісність, накладні витрати на передачу даних та управління станом. У цій статті представлено комплексний кількісний аналіз, що пропонує рішення та архітектурні шаблони, підтверджені емпіричними даними.

Матеріали та методи. Дослідження складалося з двох частин. Взаємодія на стороні клієнта була проаналізована за допомогою мікробенчмарків wasm-bindgen на основі Rust для вимірювання накладних витрат на "перетин мосту" між JavaScript та Wasm, тестування примітивів, копій масивів та доступу до SharedArrayBuffer. Потенціал на стороні сервера був оцінений шляхом порівняння модуля середовища виконання, сумісного з Wasm/WASI, з традиційним контейнером Docker, зосереджуючись на критичних хмарних метриках: час холодного запуску, розмір бінарного файлу та моделі безпеки.

Результати та обговорення. Витрати на сумісність суттєво різняться. Виклики примітивів незначні (~50-100 нс), але копіювання масиву розміром 1 МБ є серйозним вузьким місцем (1-3 мс), що робить часті копії великих даних ("балакаючі" API) нежиттєздатними. Накладні витрати SharedArrayBuffer мінімальні (~15 нс). Аналіз на стороні сервера показав трансформаційні результати: WASI приблизно в 100 разів швидший за холодний старт (<1 мс) та приблизно в 50 разів менший за розміром двійкового файлу (0,5-5 МБ), ніж Docker, пропонуючи більш детальну модель безпеки, що базується на можливостях. Тести підтверджують, що Rust+Wasm досягає приросту продуктивності до 8,7 разів. Ми обговорюємо "Wasm як чисту функцію" проти "Wasm зі спільною пам'яттю", причому останній забезпечує додаткове прискорення в 2-3 рази, усуваючи вузькі місця копіювання.

Висновки. Максимальна рентабельність інвестицій (ROI) у Wasm вимагає правильних архітектурних шаблонів та ретельного проектування "грубозернистих" API взаємодії для зменшення накладних витрат. SharedArrayBuffer – це необхідне рішення для високопродуктивних додатків. Поява WASI позиціонує його як ключову технологію для майбутніх безсерверних, периферійних обчислень та архітектур плагінів, пропонуючи суттєві, вимірювані переваги. Ключові слова: WebAssembly, продуктивність веб-додатків, мікрофронтенди, Rust, JavaScript, SharedArrayBuffer.

Ключові слова: WebAssembly, продуктивність веб-застосунків, мікрофронтенди, Rust, JavaScript, SharedArrayBuffer.

Received / Одержано	Revised / Доопрацьовано	Accepted / Прийнято	Published / Опубліковано
20 November, 2025	17 December, 2025	17 December, 2025	25 December, 2025