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

Vol. 11-No.4 (2025) pp. 8694-8702
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

WebAssembly's Expanding Role in Frontend Development: Transforming Web Application Capabilities

Amey Parab*

Independent Researcher, USA
* **Corresponding Author Email:** reachameyparab@gmail.com - **ORCID:** 0000-0001-0047-7850

Article Info:

DOI: 10.22399/ijcesen.4281
Received : 28 September 2025
Revised : 10 November 2025
Accepted : 12 November 2025

Keywords

Performance Optimization,
Cross-Language Compilation,
Browser Virtualization,
Binary Code Execution,
Modular Architecture

Abstract:

This in-depth technical article traces the development of WebAssembly from a compilation target to a core technology in frontend development. It discusses WebAssembly's fundamental architecture with a focus on its binary instruction format that supports near-native speeds of execution in browser environments, along with security via structured control flow and memory safety features. The article outlines WebAssembly's growing set of features, such as SIMD operations for data parallelism at the data level, thread support for leveraging multi-core processors, and WASI for formalized system interfaces beyond the browser. It examines implementation spaces where WebAssembly has had real impact: gaming with rich rendering features, scientific visualization with real-time data handling, development environments with sophisticated code analysis, and productivity software with advanced document management. Technical details for best-in-class integration patterns are examined alongside new capabilities such as garbage collection support, the Component Model for standardized module interaction, and enhanced security features. Throughout, the article emphasizes how WebAssembly is inherently revolutionizing web application development through the enabling of capabilities heretofore the province of native software.

1. Introduction

WebAssembly (Wasm) has evolved dramatically since its inception, evolving from its initial use as nothing more than a compilation target to now being an underpinning technology in modern frontend development. Its ability to run code at virtually native speeds has unlocked several possibilities that have dramatically altered the way developers tackle web application performance and functionality issues. This technical review assesses WebAssembly's increasing relevance and varied usage within the modern frontend development environment.

The technology was developed out of collective research among prominent browser vendors like Google, Mozilla, Microsoft, and Apple, who united to create a portable binary format for the web. The foundational design and standardization work had in mind key features such as portability, security, and performance, positioning WebAssembly as a complementary tech to JavaScript and not a replacement [1]. This methodology facilitated quick uptake in browser ecosystems without compromising backward compatibility with current web standards and development practices. The standardization process was led by the WebAssembly Community Group, established under the auspices of the World Wide Web Consortium, through a formal and open development process that included input from both browser implementers and the general developer community. WebAssembly technical architecture introduces tremendous performance benefits through its highly efficient compilation and execution model.

The binary format allows for faster transmission over networks than text formats, and the validation process guarantees memory safety and security promises that defend against possible malicious code [1]. The WebAssembly virtual machine has a stack machine with controlled structured control flow that avoids security risks of arbitrary jumps without losing interoperability with a vast array of source languages. This organized control flow not only improves security but also supports more

efficient optimization at both compile time and runtime execution. The performance features of the technology are a huge leap forward for computationally intensive web applications.

WebAssembly modules run in the same security sandbox as JavaScript but are able to take advantage of ahead-of-time compilation strategies that can perform high-level optimizations before execution. The execution model features type checking for memory accesses and function calls, which helps both in safety and performance by decreasing the dynamic type checks needed during execution time [1]. Also, the linear memory model of WebAssembly gives direct access to a continuous block of memory, which avoids the overhead of JavaScript's object-based memory management model for numerical computation and data processing operations. As adoption has spread throughout the industry, WebAssembly has proven its worth in various application areas outside its originally targeted use cases.

The technology has particularly benefited graphics-intensive uses, scientific computing, and media processing, where performance constraints once hindered web-based implementations [2]. Key platforms have adopted WebAssembly to drive important pieces of their web applications, showing faith in the technology's stability, performance, and security features. This expanding ecosystem has led to the establishment of supporting tools, libraries, and frameworks that ease the integration of WebAssembly components within existing web application architecture. In the future, the evolution of WebAssembly is ongoing through an ongoing standardization process that further refines current limitations while increasing capabilities based on real-world implementation experiences.

Proposals for better threading support, garbage collection, and interface types are major contributions that will further broaden WebAssembly's use across various programming paradigms and application areas [2]. The Component Model initiative will make it easier to integrate WebAssembly modules with host environments, solving some of the challenges in the current development process. These continuous developments demonstrate a belief in WebAssembly as a long-term foundation technology for the web platform, with wider applications possibly on servers, edge, and embedded systems in addition to the browser. With continuing development, the influence of WebAssembly on frontend development practice is likely to become even deeper, allowing a new generation of web applications that integrate the ease of access and platform independence of the web with performance traits hitherto only offered by native software. The technology is not an incremental advance but a fundamental extension of what can be done under the limitations of the web platform, and it will create new avenues for innovation across application types.

2. Fundamental Capabilities and Core Technology

WebAssembly is binary code that enables code to be run efficiently in web browsers, written in such languages as C++, Rust, and Go. The innovation provides the creators with the capacity to combine logic that is vital to performance directly into frontend applications, essentially breaking through the majority of the performance bottlenecks that have consistently been linked to JavaScript to execute computationally extreme tasks.

WebAssembly is fundamentally a low-level virtual machine that interprets high-level language-compiled portable bytecode.

The format was specifically created as a target of compilation instead of as a human-written language, although it does have a standardized text form for debugging and teaching purposes [3]. This architecture allows WebAssembly to act as a platform-neutral binary code format that can execute at near-native performance across a variety of operating systems and hardware platforms. The specification avoids platform-specific optimizations intentionally in favor of a homogeneous execution model that allows for predictable performance irrespective of deployment context. WebAssembly modules use a structured memory model in which linear memory appears as a contiguous, resizable typed array that can be shared between JavaScript and WebAssembly [3].

This design supports low-overhead memory management with the security guarantees that web applications are expected to uphold. The memory model accommodates operations typical in systems programming languages, allowing direct ports of legacy codebases without the need for extensive architecture-specific changes. Also, the table construct allows a means of storing function pointers, allowing dynamic dispatch patterns to be followed while type safety is maintained, and misuse by way of invalid function calls is avoided. The execution model follows a validation process before compilation and instantiation, which ensures modules adhere to strict safety properties in terms of control flow, type correctness, and memory access [4].

This validation phase ensures that even maliciously created WebAssembly binaries cannot breach system integrity or make unauthorized accesses. The security architecture goes beyond memory

safety and encompasses structured control flow that blocks exploits based on instruction reuse attacks or arbitrary jumps, greatly mitigating the attack surface over those in traditional native code execution environments. Current studies have shown that this method effectively reduces the most prevalent vulnerability classes without loss of performance traits critical for computation-intensive tasks [4]. WebAssembly's JavaScript interoperability makes use of a well-delineated interface layer that performs type conversions and functional demarcations.

The interface enables bi-directional communication wherein JavaScript may call WebAssembly functions and WebAssembly code might call imported JavaScript functions [3]. The interoperability platform natively supports primitive values but also has mechanisms to enable the sharing of more sophisticated data structures across the boundary. This allows for incremental strategies of adoption where performance-critical modules can be moved to WebAssembly with the ability to continue having integration with current JavaScript codebases and ecosystems. The binary representation of the compact format allows for considerably smaller transmission sizes when compared to the same JavaScript code, alleviating bandwidth limitations that impact application loading efficiency, especially on mobile networks [4].

The encoding of the binary was optimized for rapid decoding and validation, allowing browsers to execute WebAssembly modules more efficiently than equivalent JavaScript source code. This efficiency carries over into compilation, where the typed nature of WebAssembly supports more thorough optimization methods than are normally available with dynamically typed languages, leading to uniform rates of execution across browser implementations.

3. Technical Evolution and Broader Feature Set

The functionality of WebAssembly is now far more extensive than when it was originally standardized, and some important improvements have been made out which make it more likely to be viable in frontend applications.

3.1 SIMD (Single Instruction, Multiple Data)

The SIMD instructions may be able to accommodate parallelism at the data level, i.e., a huge amount of data points may be represented by a single operation executed at a time. The WebAssembly SIMD proposal adds 128-bit wide vector types and associated instructions to support programmers in processing numerous data elements in parallel with a single instruction [6]. This method strongly maps onto the hardware SIMD capabilities found in contemporary CPUs on desktop, mobile, and server platforms, delivering uniform performance advantages independent of the deployment context.

WebAssembly SIMD support adopts a fixed-width vector model with 16 lanes of 8-bit integers, 8 lanes of 16-bit integers, 4 lanes of 32-bit values, or 2 lanes of 64-bit values [6].

The instruction set contains extensive operations involving arithmetic, bit manipulation, comparison, and data reorganization across these different data types. This deployment is a balance between taking advantage of available hardware features and having consistent behavior across various architectures so that improved code provides consistent performance gains across browser deployments. The WebAssembly SIMD V8 JavaScript engine implementation has great performance benefits for algorithms vectorized appropriately. Practical usage illustrates that image processing can be accelerated by 2× to 8× based on the data characteristics and algorithm in question [6]. Such performance has direct implications for increased frontend capabilities with higher real-time processing for multimedia, scientific visualization, and interactive simulations without needing native code components or plugins.

3.2 Thread Support

The addition of shared memory and thread primitives allows WebAssembly applications to efficiently use multi-core processors. This feature is an extension of JavaScript's SharedArrayBuffer, where several instances of WebAssembly can read and write the same regions of memory in parallel [5]. The threading model includes atomic operations and memory barriers that give developers the synchronization primitives needed to create correct concurrent algorithms and support efficient compilation to native threading instructions.

Thread support is a notably important feature for computationally intensive web pages that stand to gain from parallelized execution. Being able to move workloads at the expense of the primary thread to several CPU cores without blocking the primary thread enables more responsiveness of user interfaces even during complex computations [5]. The feature is especially helpful in the context of applications that require real-time data processing, complex simulations, and media decoding/encoding, where sequential processes

would otherwise cause apparent bottlenecks in performance.

3.3 WASI (WebAssembly System Interface)

WASI is a standardized system interface for WebAssembly, whose scope extends beyond browser ecosystems. This specification introduces a capability-based security model that allows fine-grained control over accessing resources with consistent behavior across various execution environments [5]. In contrast to browser-specific APIs, WASI emphasizes platform-agnostic abstractions that facilitate portable implementations across different operating systems and device classes.

The WASI spec realizes a capability-based security principle where access to system resources is based on explicit permission grants in lieu of ambient authority [5]. This fits well with contemporary security best practices while allowing for realistic application development without undue limitation. The modularity decouples core functionality from ancillary capabilities, permitting implementations to implement exactly those features necessary for their particular use cases while having interface compatibility.

For front-end developers, WASI opens up new possibilities for code sharing between client and server environments, possibly easing full-stack development processes [5]. The ability to write components and execute them under different execution environments reduces maintenance costs and ensures consistent behavior by the application stack. The combination of browser and server environments is a radical transformation of web application architecture; the boundary between frontend and backend is increasingly becoming more open.

4. Frontend Applications and Implementation Domains

The performance benefits of WebAssembly are leading to its use in an increasingly wide variety of frontend applications.

4.1 Gaming

WebAssembly has revolutionized browser-based gaming capability and has made the web a viable platform for complex interactive experiences. Performance benchmarks point to WebAssembly-based game implementations delivering execution speeds around 1.3× faster than the equivalent JavaScript code, with certain rendering and physics aspects demonstrating up to 2.4× improvements in performance [7]. This efficiency allows for complex 3D rendering and sophisticated lighting models, and physics simulations that deliver consistent frame rates on even modest hardware configurations.

Principal game engines have introduced WebAssembly compilation routes, where developers are able to publish existing titles to browser platforms with little codebase adjustment. The binary format of the technology decreases initial loading times over JavaScript implementations, with test readings indicating median load time improvements of 23% in commercial game applications [7]. Such performance features allow web-based games to render interaction responsiveness levels close to native applications, especially for computationally intensive actions such as collision detection and particle systems.

4.2 Scientific Visualization

WebAssembly has also emerged as critical in scientific and data visualization applications, with researchers now able to deploy complicated analytical algorithms directly into browser environments. Computational efficiency of the technology supports processing large datasets with thousands or millions of items without diminishing interactive frame rates required for exploratory investigation [8]. Such an ability makes browsers feasible platforms for advanced visualization tools that would otherwise need custom desktop applications.

Research-computing visualization frameworks take advantage of WebAssembly to host domain-specific algorithms such as fluid dynamics simulations, molecular modeling, and statistical function analysis. Performance evaluation shows that such implementations run at 50-80% of the execution speed of corresponding native code, depending on the characteristics of a specific computation and memory access patterns [7]. This performance allows interactive manipulation of intricate datasets in real-time within browser environments, obliterating conventional obstacles to share interactive research visualizations.

4.3 Development Environments

WebAssembly has favorably impacted browser development tools, allowing for sophisticated code editors comparable to desktop versions. Code editors hosted in the web use syntax highlighting, code completion, and refactoring support, which is responsive even when editing large codebases, for performance-critical work leveraging

WebAssembly [8]. It unites the ease of web application access with computational power for pro-level development workflows.

Some of the leading development environments now use WebAssembly to host language servers and compiler toolchains in the browser directly. Such implementations make certain capabilities previously deemed infeasible for web tools possible, such as real-time checking of complex languages and interactive debugging with little performance penalty [8]. The technology's efficiency in running compiled code allows browser-based IDEs to manage projects of hundreds of thousands of lines of code while keeping up with the responsiveness requirements of professional development processes.

4.4 Productivity Applications

WebAssembly has raised the performance bar of web-based productivity applications, supporting complex document processing and data analysis within browser environments. Applications for office suites use the technology to provide performance-critical functionality for text layout, formula computation, and document rendering, and achieve interaction responsiveness on par with native apps [8]. The implementations are used consistently, regardless of document complexity, in solving the limitations that have previously confronted JavaScript-only web applications when dealing with big documents or intricate formatting.

Data analysis platforms make use of WebAssembly to execute computation-heavy operations such as statistical modeling and multi-dimensional visualization. Performance benchmarks show that these implementations are up to 3× faster than comparable JavaScript code for numerical algorithms, with especially notable improvements for operations with large matrices and arrays [7]. This performance allows business intelligence tools to run large datasets client-side, enhancing interactive performance while minimizing server demands.

5. Technical Implementation Considerations

Developers integrating WebAssembly into frontend applications must consider several technical factors to maximize benefits while managing implementation complexity.

5.1 Optimal Use Case Identification

All frontend pieces don't equally benefit from WebAssembly implementation. The technology achieves most of its performance benefits in computationally intensive operations, such as cryptography, image processing, and complex mathematical calculations [9]. A developer must analyze the application thoroughly to find pieces where WebAssembly can deliver worthwhile improvements instead of blindly converting whole codebases.

Performance benchmarks regularly show that WebAssembly performs better in computationally intensive work with little interaction with the DOM. Computation-based work, such as operations in closely looped, intensive calculations, can achieve performance improvements of 30% or higher over their optimized JavaScript counterparts [10]. In contrast, lightweight work or functions requiring repeated access to the DOM can end up being slower when coded in WebAssembly because of the cost of crossing the JavaScript/WebAssembly boundary. This aspect of performance makes thorough component analysis imperative for successful WebAssembly adoption.

Applications needing predictable timing of execution are especially suited for WebAssembly implementation. The technology's predictable performance traits minimize vulnerability to garbage collection pauses affecting JavaScript execution and thus are best for situations demanding smooth animation, tight timing, or consistent frame rates [9]. This predictability allows for more reliable user experiences in interaction-intensive applications where performance variation would detract from usability.

5.2 Patterns of Integration

Some patterns have appeared for the successful incorporation of WebAssembly modules into existing JavaScript codebases. The targeted acceleration approach recognizes performance bottlenecks in current applications and chooses selectively to implement those particular components in WebAssembly while keeping the larger application architecture in JavaScript [10]. The pattern represents a realistic way of incremental adoption in mature projects, enabling teams to begin solving severe performance problems without entire rewrites.

Offloaded processing pattern employs WebAssembly for computation-intensive operations that would otherwise jam the main thread, typically executing these operations within Web Workers to allow proper parallel execution. The pattern keeps user interfaces responsive during intensive computation by separating processing from UI rendering [9]. The pattern is most useful for applications with large sets of data to process or

complex calculations to perform, and still be interactive.

Core functionality implementations put core application logic into WebAssembly and use JavaScript for user interface control and browser API usage. This structure takes advantage of each technology's specialty – WebAssembly for performance-critical code and JavaScript for DOM manipulation and platform integration [10]. Development experience indicates this pattern will most effectively work for new applications built specifically with WebAssembly's capabilities in mind instead of being retrofitted onto existing codebases.

5.3 Development Workflow Considerations

Supporting WebAssembly adds further complexity to frontend development pipelines. Cross-compilation toolchains are a major factor, with frameworks such as Emscripten for C/C++, wasm-pack for Rust, and AssemblyScript for TypeScript programmers offering bridges from source languages into WebAssembly [9]. Each toolchain has varying optimization features, debugging support, and integration options, which makes toolchain choice a significant architectural concern.

Module size optimization is essential for production deployments since unoptimized WebAssembly binaries can have adverse effects on initial loading performance. Aggressive dead code elimination, code splitting to load functionality as needed, and the methods to compress code specifically to the binary format of WebAssembly are some of the methods [10]. They come in handy when it comes to the tradeoff between download size and execution performance, which is of utmost importance to applications that will be deployed on mobile devices or those with constrained bandwidth.Debugging between language interfaces has specific challenges that must be met by development teams through targeted methodologies. Chrome DevTools and Firefox Developer Tools provide debugging capabilities associated with WebAssembly, and source maps are capable of maintaining links between compiled code and sources [9]. Despite these changes, debugging complex WebAssembly applications tends to be more work than native JavaScript code, a workflow factor that must be factored in both development planning and resource allocation.

6. Future Direction and New Capabilities

WebAssembly's technical development goes on in a rapid fashion, with a number of developments set to further cement its place in frontend development.

6.1 Garbage Collection Proposal

Adding garbage collection functionality is one of the biggest upcoming developments in the WebAssembly standard.

This feature adds a group of new instructions and type definitions that make it easier for languages with garbage collection to target WebAssembly [11]. Instead of expecting languages such as Java, C#, and TypeScript to include their own garbage collection libraries or depend on advanced JavaScript interoperability, the proposal introduces native support for managed objects within the WebAssembly virtual machine itself. The proposal on garbage collection specifies a type system extension containing typed object references, arrays, structures, and collection mechanisms that are consistent with typical language implementation patterns [11]. This reduces overhead for compilers to translate high-level language features directly into WebAssembly types without costly abstraction layers or runtime bridges. Merging with the host environment's memory management system reduces boilerplate garbage collection overhead while upholding security assurances fundamental to WebAssembly's design philosophy.

For frontend developers, this development would greatly broaden the language environment for developing with WebAssembly. Existing constraints have privileged languages that have to manage memory manually, such as C, C++, and Rust, hindering developers who are more comfortable with garbage-collected languages [11]. The proposal would simplify more direct compilation pipelines for JavaScript, TypeScript, Java, C#, and other popular languages so that teams could build upon existing experience while taking advantage of WebAssembly's performance benefits.

6.2 Component Model

The WebAssembly Component Model is a proposal to standardize the interaction of WebAssembly modules, which addresses the existing deficiencies in sharing code and defining interfaces. This work introduces a higher-level composition system that makes it possible for WebAssembly modules to interact through properly defined interfaces with uniform semantics regardless of the initial source language [12]. The model puts forward canonical ABI (Application Binary Interface) specifications that mediate among disparate language ecosystems, making it possible for language-agnostic component interaction.

At its core, the Component Model introduces interface types capturing common data structures in

a language-independent manner, facilitating smooth data exchange between components [12]. The types are primitive values, records, variants, lists, and other basic structures that can represent complicated data relationships while still allowing efficient translation to native forms in every language. The type system accommodates synchronous and asynchronous calls to functions, facilitating components to articulate varied patterns of interaction while preserving compatibility.

The real-world implications for frontend development are enormous, especially for large applications that can take advantage of modular architecture. The Component Model allows development teams to write genuine polyglot applications where every module is written in the best language for its particular use case [12]. This allows organizations to make use of specialized language strengths—Rust for performance-critical processing, JavaScript for DOM manipulation, C++ for libraries of existing algorithms—while integrating cohesive system integration through standardized interfaces.

6.3 Advanced Security Features

Security improvement work continues with the goal of broadening the application domains for WebAssembly without diminishing its rigorous safety guarantees. Upcoming development centers on capability-oriented security models where resources must be explicitly allowed before they are accessed, as opposed to ambient authority [11]. The strategy is consistent with the principle of least privilege security designs so that components can only access the particular resources needed to operate.

The security model is based on WebAssembly's current sandbox but goes further with finer-grained control. Instead of running under a single permission context, modules are able to ask for and be granted particular capabilities for file system operations, network access, hardware capabilities, or other system resources [12]. These are described as unforgeable references explicitly handed over to pieces of code that need them, evading unauthorized access via transparent security borders.

For web applications, these security features would allow more advanced behavior without compromising the model of security for the web. Applications would be able to use untrusted third-party libraries or user-submitted code with strict limits on the extent of their possible influence [11]. This feature would be especially useful for extensible programs such as content management systems, data analysis suites, and creative applications whose plugin-based ecosystems augment base functionality while needing rigorous security boundaries.

Table 1: *WebAssembly Performance Characteristics: Comparing Key Execution Metrics with JavaScript [3, 4]*

Feature	WebAssembly	JavaScript
Execution Speed	Near-native performance	Slower for computational tasks
Memory Management	Direct linear memory access	Object-based with garbage collection
File Size	Compact binary format	Larger text-based format
Validation	Pre-execution validation	Runtime type checking
Language Support	C++, Rust, Go, others	JavaScript, TypeScript
DOM Interaction	Requires JS bridge	Native access
Security Model	Structured control flow	Dynamic execution
Optimization	Compile-time optimization	Just-in-time compilation
Type System	Static typing	Dynamic typing
Loading Efficiency	Fast decoding and validation	Requires parsing and interpretation

Table 2: *WebAssembly Advanced Features: SIMD and Threading Performance Metrics [5, 6]*

Feature	Specification	Performance Impact
Vector Width	128-bit	Enables parallel processing
Lane Configuration	16×8-bit, 8×16-bit, 4×32-bit, 2×64-bit	Flexible data type handling
Operation Types	Arithmetic, bitwise, comparison, data rearrangement	Comprehensive instruction set
Image Processing	Vectorized algorithms	2×-8× speedup
Threading Model	SharedArrayBuffer integration	Multi-core utilization
Memory Access	Shared memory regions	Concurrent processing
Security Model	Capability-based permissions	Fine-grained resource control
Deployment Scope	Browser and beyond (via WASI)	Cross-environment execution

Table 3: *WebAssembly Frontend Integration: Technical Implementation Factor [9, 10]*

Implementation	Key Considerations	Performance Impact	Best Practices

Aspect			
Use Case Selection	Computation-intensive tasks (cryptography, image processing)	30%+ improvement for intensive calculations	Analyze performance bottlenecks first
DOM Interaction	Limited DOM access	Performance penalty for frequent DOM operations	Keep DOM manipulation in JavaScript
Integration Pattern	Targeted acceleration, offloaded processing, core functionality	Varies by pattern	Match pattern to application architecture
Boundary Crossing	JavaScript/WebAssembly interface	Overhead for frequent crossings	Minimize boundary transitions
Threading	Worker-based parallel execution	Improved UI responsiveness	Use for background processing
Toolchain Selection	Language-specific (Emscripten, wasm-pack, AssemblyScript)	Affects optimization capabilities	Choose based on team expertise
Bundle Size	Initial download impact	Affects loading performance	Implement dead code elimination, code splitting
Debugging Complexity	Cross-language challenges	Additional development effort	Utilize source maps, specialized tools

Table 4: *WebAssembly's Evolution: Upcoming Features and Technical Capabilities [11, 12]*

Feature	Current Status	Key Benefits	Implementation Impact
Garbage Collection	Proposal phase	Native managed object support	Expands language ecosystem (Java, C#, TypeScript)
Component Model	Under development	Language-neutral interfaces	Enables true polyglot applications
Interface Types	Core specification	Common data structure representation	Seamless cross-component communication
Canonical ABI	Design phase	Standardized calling conventions	Bridges different language ecosystems
Security Capabilities	Proposal phase	Fine-grained permission model	Resource access control
Memory Management	Enhancement	Integration with host GC	Reduced runtime overhead
Module Composition	Architecture design	Standardized component boundaries	Improved code sharing and reuse
Permission Context	Security framework	Explicit capability grants	Isolation of untrusted components

7. Conclusions

WebAssembly has developed to become a promising technology into a vital part of modern frontend development. This has been fundamental in enhancing the functionality of web applications in many areas because of its ability to provide a near-native performance in the browser system. As the technology keeps evolving and adds more features, there is more hope that the developers can expect more chances to develop more advanced, high-performing, and multipurpose web applications, gaining the ability to rival or even surpass the potential that traditional native programs have. The current standardization process and the increased responsibility of ecosystem support indicate that WebAssembly will become more of a core component of frontend development in the future, as web applications continue to gain more complex and performance-intensive tasks in the enterprise and consumer computing settings.

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] Andreas Haas et al., "Bringing the Web Up to Speed with WebAssembly," 2017. https://people.mpi-sws.org/~rossberg/papers/Haas,%20Rossberg,%20Schuff,%20Titzer,%20Gohman,%20Wagner,%20Zakai,%20Bastien,%20Holman%20-%20Bringing%20the%20Web%20up%20to%20Speed%20with%20WebAssembly.pdf

[2] Philipp Spiess and James Swift, "WebAssembly: A New Hope," Nutrient.io, 2024. https://www.nutrient.io/blog/webassembly-a-new-hope/

[3] Mozilla Developer Network, "WebAssembly,". [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly

[4] Gaetano Perrone and Simon Pietro Romano, "WebAssembly and Security: A review," arXiv:2407.12297v1, 2024. https://arxiv.org/html/2407.12297v1

[5] Faang, "WebAssembly Beyond the Browser: Why Your Backend Will Never Be the Same," Medium, 2025. [Online]. Available: https://medium.com/@FAANG/webassembly-beyond-the-browser-why-your-backend-will-never-be-the-same-9ba86d5ad7a4

[6] V8 Developer, "Fast, parallel applications with WebAssembly SIMD," 2022. [Online]. Available: https://v8.dev/features/simd

[7] Yutian Yan et al., "Understanding the Performance of WebAssembly Applications," ACM, 2021. https://weihang-wang.github.io/papers/imc21.pdf

[8] Acodez, "WebAssembly In Modern Web Development: How It Can Revolutionize Web Performance," 2024. [Online]. Available: https://acodez.in/webassembly-in-modern-web-development/

[9] Victor Ogbonna, "The Role of WebAssembly in Frontend Development," Dev. to 2024. [Online]. Available: https://dev.to/outstandingvick/the-role-of-webassembly-in-frontend-development-55pd

[10] Stanley Ulili, "Implementing WebAssembly for High-Performance Web Apps," Better Stack Community, 2025. [Online]. Available: https://betterstack.com/community/guides/scaling-nodejs/webassembly-web-apps/

[11] Enrico Piovesan, "What WebAssembly 2.0 Actually Adds (and Why It Matters)," Medium, 2025. [Online]. Available: https://medium.com/wasm-radar/what-webassembly-2-0-actually-adds-and-why-it-matters-9606dad2faf3

[12] Bytecode Alliance, "WebAssembly Component Model,". [Online]. Available: https://component-model.bytecodealliance.org/