Received: 4 December 2024 | Revised: 21 February 2025 | Accepted: 1 April 2025 | Published online: 10 April 2025

RESEARCH ARTICLE

Archives of Advanced Engineering Science
2026, Vol. 4(3) 275–287
DOI: 10.47852/bonviewAAES52024965

BON VIEW PUBLISHING

A Comparative Study of WebAssembly Runtimes: Performance Metrics, Integration Challenges, Application Domains, and Security Features

Mircea Ţălu[1,2,*]

[1]*Faculty of Automation and Computer Science, The Technical University of Cluj-Napoca, Romania*

[2]*SC ACCESA IT SYSTEMS SRL, Romania*

Abstract: WebAssembly (Wasm), originally developed as a web-focused technology, has evolved into a versatile runtime environment capable of executing code efficiently across web-based and standalone systems. This survey provides a comprehensive analysis of Wasm runtimes, categorizing them into two primary groups: integrated runtimes, which function within web browsers, and standalone runtimes, which operate independently of browsers. Integrated runtimes, such as those found in Chrome, Firefox, Safari, and Edge, execute Wasm binaries through JavaScript engines, allowing interaction with web application programming interfaces while maintaining a secure execution model. Standalone runtimes like Wasmtime, Wasmer, WebAssembly Micro Runtime, WasmEdge, and Wasm3 operate independently of browsers, serving various applications, including edge computing, Internet of Things, and embedded systems. We explore each runtime's performance characteristics, highlighting their execution modes, such as interpretation, Just-In-Time compilation, and Ahead-of-Time compilation, and their ability to handle resource-constrained environments. Our findings provide valuable insights for developers, researchers, and industry professionals seeking to leverage Wasm for optimized performance, scalability, and security across diverse technological landscapes.

Keywords: edge computing, embedded systems, IoT devices, standalone runtimes, WebAssembly, web-based execution

1. Introduction

WebAssembly (Wasm) was launched in 2015 as a revolutionary technology aimed at improving the efficiency and security of web applications. This initiative addressed the challenges of asm.js by providing a more streamlined, assembly-like language that is specifically optimized for web use [1–3]. Wasm's scope has expanded far beyond its original web focus, finding applications across diverse domains such as edge computing, Internet of Things (IoT) devices, and embedded systems. The development of Wasm involved collaboration among major tech giants such as Mozilla, Microsoft, Apple, and Google, focusing on overcoming the limitations of existing web technologies. Wasm empowers developers to compile high-level programming languages into a compact, platform-independent format that executes natively across various environments, delivering near-native performance, increased security, and cross-platform portability. Since its introduction, Wasm has rapidly gained traction, becoming an official W3C standard in 2019 and establishing itself as a crucial component of modern web and cloud ecosystems [4–7]. With the swift expansion of cloud-edge computing, IoT, and embedded systems, ensuring data integrity, confidentiality, and privacy has become a top priority.

Wasm introduces a binary instruction format designed to execute code from various programming languages within a controlled, sandboxed setting. This format allows developers to compile high-level languages such as C, C++, and Rust into Wasm, which can then be executed efficiently in web environments. The architecture of Wasm is engineered to prioritize both performance and security, providing a mechanism that isolates the execution environment. This isolation helps protect the host system from potentially harmful code and ensures that applications run swiftly and reliably. In addition to the primary languages, Wasm supports an ever-expanding list of programming languages, enabling greater flexibility and accessibility for developers. Some of the notable languages that can be compiled into Wasm include AssemblyScript, which is specifically designed for use with Wasm, and popular languages like C#, Go, F#, Dart, Kotlin, Swift, D, Pascal, Zig, and Grain [8–10]. This diverse language support empowers a wide range of applications, from web development to server-side programming and beyond, broadening the scope of what can be achieved using Wasm technology. Wasm binaries are designed with a modular architecture that enhances efficiency and execution. This architecture comprises functions, which are reusable code blocks that improve organization and facilitate reusability, and global variables that provide persistent data

***Corresponding author:** Mircea Ţălu, Faculty of Automation and Computer Science, The Technical University of Cluj-Napoca, Romania and SC ACCESA IT SYSTEMS SRL, Romania. Emails: mircea.talu@accesa.eu; talu.s.mircea@gmail.com

© The Author(s) 2025. Published by BON VIEW PUBLISHING PTE. LTD. This is an open access article under the CC BY License (https://creativecommons.org/licenses/by/4.0/).

storage accessible throughout the module. Additionally, it utilizes linear memory, a dynamically allocated memory model similar to C/C++, allowing flexible management of data sizes during execution. The stack-based execution model employs a last-in-first-out mechanism for managing function calls and variables. Within this framework, an embedder – typically a JavaScript engine – plays a crucial role by loading and executing Wasm modules, thereby bridging Wasm with the host environment. This embedder also facilitates host interaction, enabling input/output operations, managing network requests, and handling timers, all while ensuring robust error management. Collectively, this architecture delivers high performance and security, making Wasm suitable for a wide range of applications across various platforms [11–13].

Wasm runtimes are specialized software environments designed to execute Wasm binaries, providing a crucial foundation for running Wasm code across various platforms. These runtimes can be implemented in multiple forms, including integration within web browser engines or as standalone solutions that can be embedded into diverse applications. Their primary function is to create a secure and efficient execution environment tailored for Wasm code, which includes several critical features such as robust memory management, enhanced security protocols, and performance optimizations. Memory management in Wasm runtimes is particularly important, as it ensures efficient allocation and deallocation of resources while minimizing memory overhead. This is especially relevant in environments with limited resources, where the ability to effectively manage memory can significantly impact overall system performance. Moreover, security features are integral to these runtimes, providing a sandboxed execution model that isolates Wasm code from the host environment. Performance optimization is another key aspect of Wasm runtimes. By leveraging techniques such as Just-In-Time (JIT) compilation or Ahead-of-Time (AoT) compilation, these runtimes can enhance the execution speed of Wasm binaries, bringing performance closer to that of native code. This efficiency is vital for applications that demand quick processing times, particularly in real-time scenarios. Embedded Wasm runtimes are often deployed in IoT devices, microcontrollers, and other hardware platforms that necessitate low memory footprints and rapid execution capabilities. Given the increasing demand for versatile computing environments, Wasm runtimes have emerged as key enablers of efficient execution across a wide spectrum of platforms, from web and mobile applications to edge computing and IoT devices. The selection of an appropriate runtime environment is crucial, as each offers distinct trade-offs in terms of performance, memory utilization, and integration capabilities. Notably, certain runtimes are optimized for specific workloads, such as high-performance computing or resource-constrained devices, while others excel in areas like security, scalability, and compatibility with cloud infrastructures. Furthermore, as Wasm continues to mature, its role in facilitating secure and portable execution of code has garnered significant attention in both academic and industrial contexts. However, the choice of a Wasm runtime is not solely dependent on raw performance metrics but must also account for other critical factors such as ease of integration with existing systems, compatibility with various programming languages, and the runtime's ability to handle complex application demands [14–17].

Mendki [6] investigated the evolution of the edge computing ecosystem, emphasizing serverless architectures and evaluating the role of Wasm in this domain. Hoque and Harras [7] explore the challenges of portability and migratability in edge-offloading, assessing existing technologies and examining Wasm's potential to address code compatibility across heterogeneous edge devices. Ray [8] provides a comprehensive analysis of Wasm's role in IoT, examining its performance, tools, integration challenges, and future directions, emphasizing its potential for secure, efficient, and scalable edge-IoT ecosystems. Lehmann et al. [9] investigated the security of Wasm binaries, revealing that classic vulnerabilities mitigated in native code remain exploitable and introducing new attack primitives that expose significant security risks. Yan et al. [10] conducted a study on Wasm's performance, revealing its dependency on low-level virtual machine (LLVM) optimizations, higher memory usage, and varying performance compared to JavaScript across different execution environments. Dejaeghere et al. [11] compared the security models of eBPF and Wasm, highlighting eBPF's performance-first approach versus Wasm's security-focused design, and identifying future directions for enhancing eBPF's security. Ţălu [12, 13] reviewed vulnerability discovery in Wasm binaries through static, dynamic, and hybrid analysis, and explored advanced techniques for data protection in Wasm, highlighting security challenges and mitigation strategies. Kakati and Brorsson [14] reviewed the role of Wasm in the edge-cloud continuum, highlighting its potential for cross-platform interoperability, performance optimization, and security in heterogeneous computing environments. Zhang et al. [15, 16] provided a comprehensive survey of Wasm runtimes, analyzing their design and challenges, and investigated Wasm runtime bugs, characterizing their impact and proposing detection techniques. Watt [17] studied a mechanized Isabelle specification of Wasm, verifying its type system and developing an executable interpreter, influencing the official specification through formal proofs and differential fuzzing. Górski [18] proposed the 1+5 architectural views model for designing integration solutions of collaborating software systems, extending UML with profiles and introducing an Integration Flow diagram to organize mediation mechanisms.

Ménétrey et al. [19] studied Twine, a trusted runtime for Wasm applications within Intel SGX-based trusted execution environments, offering performance comparable to state-of-the-art solutions while ensuring memory safety, attestation, and controlled OS services. De Macedo et al. [20] compared the runtime and energy performance of Wasm and JavaScript, finding that while Wasm is still developing, it already presents a promising challenge to JavaScript with significant potential for future improvement. Wang [21] conducted a comprehensive characterization study of standalone Wasm runtimes, revealing performance slowdowns compared to native executions and highlighting the need for dynamic optimizations and addressing architectural challenges in non-web domains. Gackstatter et al. [22] studied a Wasm-based serverless container runtime, WOW, for edge computing. Jiang et al. [23] introduced WarpDiff, a differential testing approach to identify performance issues in server-side Wasm runtimes, discovering and analyzing seven performance problems in five popular runtimes. Zhou et al. [24] introduced WADIFF, a differential testing framework for Wasm runtimes, which identified 417 inconsistent instructions and 21 bugs across 7 popular runtimes, with 8 confirmed by developers. Gottschalk et al. [25] explored adaptive Wasm applications leveraging execution environment capabilities, while Kim et al. [26] studied several Wasm security solutions. Johnson et al. [27] studied WaVe, a verifiably secure Wasm sandboxing runtime, while Protzenko et al. [28] discussed formally verified cryptographic web applications in Wasm. Legoupil et al. [29] explored Iris-MSWasm, a mechanism for elucidating and mechanizing the security invariants of memory-safe Wasm, while Tsoupidi et al. [30] discussed Vivienne, a relational verification method for cryptographic implementations in Wasm. Kakati and Brorsson [31] evaluated Wasm across architectures in the cloud-edge continuum, while Li et al. [32] investigated using Wasm for seamless device-cloud integration on resource-constrained IoT devices.

Marcelino and Nastic [33] studied CWASI, a Wasm runtime shim for inter-function communication in the serverless edge-cloud continuum, while Jain [34] explored Wasm's applications in the cloud. Wagner et al. [35] analyzed the energy consumption and performance of Wasm binaries across programming languages and runtimes in IoT, while Long et al. [36] introduced WACP, a performance profiling tool for Wasm-Python interoperability. Kyriakou and Tselikas [37] explored how Rust and Wasm complement JavaScript in high-performance Node.js and web applications, while Szewczyk et al. [38] analyzed Wasm's performance, focusing on bounds checking. Spies and Mock [39] evaluated Wasm's use in non-web environments, while Mohan [40] compared the performance of JavaScript and Wasm in browser environments.

Our study provides a comprehensive evaluation of Wasm runtimes, analyzing their architecture, execution models, performance metrics, integration capabilities, security features, and application domains. We also show a comparative analysis of performance across different workloads and environments, offering valuable insights for developers selecting the most suitable runtime for their needs.

The paper is organized as follows: Section 2 outlines the research methodology, detailing the approach used to analyze and compare Wasm runtimes. Section 3 explores both the architecture of Wasm runtimes and highlights the most popular runtimes, providing a comprehensive overview. Section 4 evaluates the performance, integration capabilities, and application domains of Wasm runtimes, supported by benchmark comparisons across various workloads and environments. Section 5 summarizes the key findings and discusses future research directions.

2. Research Methodology

A thorough survey evaluated recent developments in Wasm runtimes, spanning the years 2018–2024, through three phases: 1) formulating review questions and collecting data, 2) analyzing and extracting information, and 3) synthesizing and drawing conclusions.

1) Formulating review questions and collecting data: The first step involved formulating review questions based on critical areas such as performance, integration, security, and application domains. These questions helped to guide the systematic review process. Following this, data were collected by performing a comprehensive literature review of Wasm runtimes from reputable sources such as academic journals, technical reports, and documentation. The collection process also included web sources, runtime community discussions, and performance benchmarks. Inclusion criteria focused on recent publications, peer-reviewed sources, and those directly related to Wasm runtimes. Citation chaining was also utilized to further extend the breadth of our data.
2) Analyzing and extracting information: The second step of the methodology was focused on thematic analysis of the collected data. A qualitative approach was employed to extract relevant metrics, technical specifications, and use cases. Data from various sources were categorized based on key attributes such as performance, integration capabilities, and security features. Thematic subgroups were developed to align with our review questions. This analysis was done and reviewed for inter-coder consistency to minimize bias and ensure reliable coding.
3) Synthesizing and drawing conclusions: The third step utilized a systematic review process, following a structured framework to ensure transparency and rigor. We applied quality assessment criteria for evaluating the selected studies based on methodological soundness, relevance, and data quality. Once the data were extracted, we synthesized the findings using tools like Excel and NVivo software. These tools helped organize the data into a comprehensive matrix, allowing us to compare and contrast runtimes based on performance and integration features. Cross-verification was conducted to ensure accuracy and consistency in the results. Finally, the synthesized results were interpreted in relation to the research questions, offering clear insights into how each runtime fits various use cases, its integration potential, and its performance under different workloads.

Throughout this process, special care was taken to minimize publication bias and ensure that the review was balanced and structured.

3. The Architecture and Popular WebAssembly Runtimes

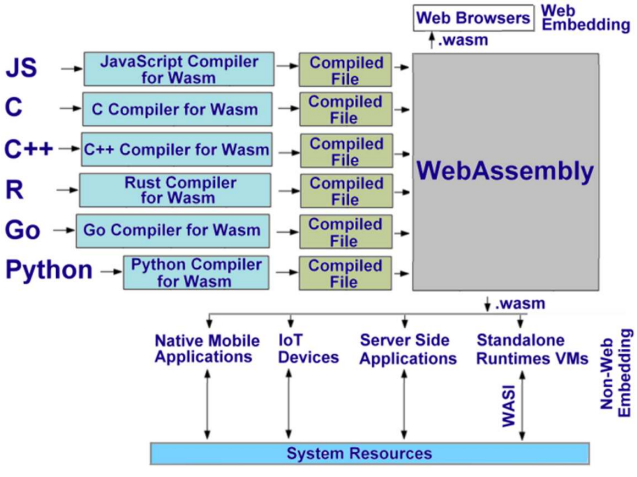
3.1. The architecture of WebAssembly runtimes

The architecture of Wasm runtimes provides a dynamic and efficient environment for executing Wasm binaries across diverse platforms, optimizing performance and security. Figure 1 shows the data flow architecture of Wasm (adapted from Ray [8] with permission of MDPI AG publisher).

Wasm runtimes are built upon several key components that collaboratively manage the loading, execution, interaction, and lifecycle of Wasm modules [15, 16].

Module loading begins the process, where the runtime is tasked with loading Wasm binaries into memory. This stage involves parsing the binary format, validating the module, and preparing it for execution. Efficient memory management is integral, as Wasm modules require a specific allocation of linear memory, with runtimes optimizing this allocation to enhance performance and minimize overhead. At the heart of the architecture is the execution engine, which supports various execution modes. The interpreter executes bytecode sequentially, offering simplicity but at the cost of slower performance. In contrast, JIT compilation converts Wasm code into native machine code during runtime, significantly boosting execution speed. Additionally, AoT compilation pre-compiles the code to native format before execution, further improving startup times

Figure 1
The data flow architecture of WebAssembly



and runtime efficiency. A critical aspect of Wasm runtimes is their security model, centered around sandboxing. By isolating the Wasm module from the host system, runtimes prevent untrusted code from accessing sensitive resources, ensuring a secure execution environment. Memory safety mechanisms are also implemented, preventing out-of-bounds memory access and mitigating potential vulnerabilities. The architecture allows for smooth host environment interaction, as Wasm runtimes can be embedded into a wide range of environments, including web browsers, servers, and IoT devices. Runtimes manage imports and exports, enabling Wasm modules to access host system application programming interfaces (APIs) and share functions between the module and the host. Furthermore, the API and runtime interface provide developers with tools to interact with Wasm modules, including invoking exported functions, managing memory, and handling errors. Some runtimes also offer performance monitoring capabilities, enabling developers to profile and optimize their applications. The typical execution flow within a Wasm runtime begins with the loading of the Wasm module, followed by parsing, validation, and memory allocation. The runtime then executes the module and manages its interaction with the host environment, overseeing the module's lifecycle until it is unloaded. This interconnected structure ensures the seamless and efficient execution of Wasm binaries across platforms.

3.2. The most popular WebAssembly runtimes

The diversity of Wasm runtimes, including browser-based, standalone, embedded, and specialized runtime, is shown in Table 1 and covers the key features, execution modes, and primary use cases for each runtime [8, 19–23].

Table 1 categorizes popular Wasm runtimes into six groups, highlighting their key features, execution modes, and primary use cases:

1) Browser-based runtimes: These runtimes are embedded within popular web browsers and support both client-side and server-side execution. They focus on optimizing performance for web applications and games

V8 (Chrome/Node.js) supports JIT compilation and is designed for web and hybrid apps, as well as server-side Node.js applications. SpiderMonkey (Firefox) is optimized for security and sandboxing, serving web apps and games. JavaScriptCore (Safari) focuses on browser-based applications and mobile web apps with a strong emphasis on performance. ChakraCore (Edge), used in older versions of Microsoft Edge, supports fast JIT compilation for web apps and server-side processing.

2) Standalone runtimes: These runtimes are independent of the browser and are ideal for server-side applications, containerized services, and cloud computing

Wasmtime is a universal runtime designed for WASI (WebAssembly System Interface) and optimized for server-side processing and cloud services. Wasmer is highly versatile, supporting multiple engines and cross-platform execution for IoT, desktop, cloud, and blockchain applications. WebAssembly virtual machine (WAVM) is optimized for compute-intensive workloads like scientific computing and server tasks. Lucet is designed for fast startup times in cloud environments, perfect for serverless computing scenarios.

3) Embedded and IoT runtimes: These runtimes are lightweight and suitable for embedded systems or IoT devices that require minimal resources

WebAssembly Micro Runtime (WAMR) focuses on constrained environments and microcontrollers, offering a small memory footprint. Wasm3 is an ultra-lightweight interpreter designed for IoT and embedded devices. WasmEdge is a high-performance runtime optimized for edge computing and artificial intelligence (AI) workloads in IoT applications.

4) Specialized runtimes: These runtimes are tailored for specific ecosystems such as blockchain and research purposes

Node.js (with Wasm) integrates Wasm support in server-side JavaScript applications.

Blazor WebAssembly enables .NET applications to run in the browser, facilitating cross-platform solutions. AssemblyScript

Table 1
Classification and overview of the most popular WebAssembly runtimes

Category	Runtime	Key features	Execution modes	Primary use cases
Browser-Based Runtimes	V8 (Chrome/ Node.js)	Integrated with Chrome and Node.js, optimized for both client-side and server-side execution, strong JIT support	Just-In-Time (JIT), Baseline Compiler	Web apps, hybrid apps, server-side Node.js applications
	SpiderMonkey (Firefox)	Firefox's JavaScript engine, supporting both Wasm and asm.js, high security and sandboxing	JIT, Baseline Interpreter	Web applications, games, interactive websites
	JavaScriptCore (Safari)	Safari's Wasm runtime, designed with WebKit, secure and optimized for browser-based performance	JIT	Web-based apps, mobile web apps, performance-sensitive web platforms
	ChakraCore (Edge)	Microsoft's JavaScript engine for Wasm in older Edge versions, fast JIT and memory management	JIT, Interpreter	Web apps (Edge legacy), server-side processing on Windows
Standalone Runtimes	Wasmtime	Designed for standalone Wasm execution, optimized for Wasm as a universal runtime, strong support for WASI	JIT, AoT (Ahead-of-Time) Compilation	Server-side processing, containerized applications, cloud services

(*Continued*)

Table 1
(*Continued*)

Category	Runtime	Key features	Execution modes	Primary use cases
	Wasmer	Universal Wasm runtime, supports multiple engines, includes native bindings, built for cross-platform execution	JIT, AoT	Server-side, IoT, desktop, cloud, and blockchain applications
	WAVM	High-performance Wasm runtime built on LLVM, extensive support for WASI and threading	AoT, JIT	Compute-intensive workloads, scientific computing, server workloads
	Lucet	Fast startup Wasm runtime designed for cloud environments, aimed at enabling serverless functions	AoT, JIT	Cloud computing, serverless, fast-execution scenarios
Embedded and IoT Runtimes	WAMR (WebAssembly Micro Runtime)	Lightweight runtime optimized for IoT, embedded systems, and constrained environments, minimal memory footprint	Interpreter, AoT	IoT devices, microcontrollers, embedded systems, smart devices
	Wasm3	Ultra-lightweight Wasm interpreter, fast initialization, very small binary size	Interpreter	IoT, embedded devices, resource-constrained environments
	WasmEdge	High-performance Wasm runtime optimized for edge computing and cloud-native services, supports AI workloads	JIT, AoT	Edge computing, IoT, AI applications, Kubernetes-based services
Specialized Runtimes	Node.js (with Wasm)	Node.js runtime with built-in support for WebAssembly via V8 engine, good for server-side Wasm applications	JIT, Interpreter	Server-side JavaScript applications, microservices, back-end processing
	Blazor WebAssembly	Microsoft's .NET runtime for Wasm, integrates with C# and .NET to run in the browser	Interpreter	Web apps, cross-platform .NET applications, hybrid client-server solutions
	AssemblyScript	TypeScript-based language that compiles to WebAssembly, optimized for high compatibility with JavaScript engines	JIT, AoT	Web applications, performance-critical TypeScript code
Blockchain-Specific Runtimes	EOS VM	High-performance Wasm runtime designed for EOS blockchain, supports smart contracts and decentralized apps (dApps)	AoT, JIT	Blockchain, smart contracts, decentralized apps (dApps)
	Parity Wasm (Substrate)	Specialized runtime for Substrate blockchain framework, supports high-speed execution of smart contracts	AoT, JIT	Blockchain ecosystems, smart contract execution, decentralized applications
Experimental/Research Runtimes	Life	Research-based Wasm runtime, focuses on deterministic execution and reproducibility for scientific computing	JIT, AoT	Scientific computing, research projects, reproducible computing environments
	SSVM (Second State VM)	Wasm runtime optimized for AI and machine learning workloads, with strong support for cloud and edge applications	JIT, AoT	AI workloads, machine learning, edge computing, cloud-native AI services

compiles TypeScript into Wasm, making it highly compatible with JavaScript.

5) Blockchain-specific runtimes: These runtimes are designed for blockchain environments, optimizing Wasm for decentralized applications and smart contracts

EOS VM: Tailored for the EOS blockchain, supporting high-performance smart contract execution and decentralized apps (dApps).

Parity Wasm (Substrate): Specialized for the Substrate blockchain framework, offering fast execution for smart contracts and decentralized applications.

6) Experimental/research runtimes: These runtimes are focused on research or specific high-performance tasks, often used in scientific or AI contexts

Life: A research-based runtime designed for reproducible and deterministic scientific computing. SSVM (Second State VM):

Optimized for AI and machine learning (ML) workloads, supporting cloud and edge computing environments.

4. Performance, Integration, and Application Domains of WebAssembly Runtimes

Wasm runtimes vary significantly in performance, optimization strategies, and suitability for different application domains. An overview of the performance characteristics of several popular Wasm runtimes categorized by their operational contexts, along with performance and integration metrics, is shown below [8, 19–24].

4.1. Performance metrics

Performance is a key criterion in evaluating Wasm runtimes, as it directly impacts the responsiveness and efficiency of applications. Performance can vary significantly across different Wasm runtimes, influenced primarily by their execution modes and optimizations. Table 2 shows a comparison of 18 Wasm runtimes based on performance metrics such as execution speed, memory usage, JIT compilation support, and typical use cases [8–14]. V8 shows the highest execution speed (90–95%) and is optimized for web apps, gaming, and interactive data, with a memory usage of 30 MB. SpiderMonkey, with 80–85% execution speed, is ideal for interactive web apps and supports JIT compilation with 28 MB memory usage. JavaScriptCore offers a slightly lower execution speed (75–80%) but is tailored for mobile web apps, using 24 MB memory. ChakraCore also has a similar speed (75–80%) and is used in legacy Edge apps, requiring 26 MB of memory. Wasmtime, with 85–90% execution speed, is optimized for cloud services and server-side apps, utilizing 25 MB of memory. Wasmer offers good performance (80–85%) and is used in blockchain and cross-platform desktop apps, with 18MB memory usage. WAVM, supporting scientific computing and trading platforms, has a performance range of 80–85%, with 20 MB memory. Lucet shows the same execution speed (85–90%) as Wasmtime, focusing on serverless functions and cloud services with 22 MB memory usage. WAMR, primarily for IoT applications, has a lower execution speed (75–80%) and minimal memory usage (15 MB), while Wasm3 performs at 70–75% speed with only 10 MB of memory, optimized for resource-constrained devices. WasmEdge, ideal for AI and edge computing, offers high performance (85–90%) with 20 MB memory. Node.js (with Wasm) maintains 75–80% performance for server-side JavaScript and microservices with 30MB memory usage. Blazor WebAssembly has lower performance (70–75%) but is suitable for web apps and cross-platform .NET applications, requiring 28MB memory. AssemblyScript, designed for performance-critical TypeScript web apps, runs at 80–85% speed and uses 25 MB memory. EOS VM is optimized for blockchain and smart contracts, showing 80–85% execution speed with 20 MB memory. Parity Wasm, also for blockchain ecosystems, operates at 80–85% speed with 22 MB memory. SSVM, tailored for AI workloads, has 80–85% execution speed and 25 MB memory usage. Finally, Life, focused on scientific computing and reproducible research, performs at 70–75% speed, with minimal memory usage (15 MB).

4.2. Integration capabilities

Integration capabilities are a critical factor when selecting Wasm runtimes, as they determine the ease and flexibility with which these runtimes can be incorporated into diverse development environments and application scenarios. Each runtime's integration efficiency is shaped by factors such as language support,

Table 2
Performance benchmarks of WebAssembly runtimes

Runtime	Execution speed (relative performance)	Memory usage (MB)	JIT compilation	Use case examples
V8	90–95%	30	Yes	Web apps, gaming, interactive data
SpiderMonkey	80–85%	28	Yes	Interactive web applications
JavaScriptCore	75–80%	24	Yes	Mobile web apps
ChakraCore	75–80%	26	Yes	Web apps (Edge legacy), server-side processing
Wasmtime	85–90%	25	Yes	Cloud services, server-side applications
Wasmer	80–85%	18	Yes	Blockchain, cross-platform desktop applications
WAVM	80–85%	20	Yes	Scientific computing, trading platforms
Lucet	85–90%	22	Yes	Serverless functions, low-latency cloud services
WAMR	75–80%	15	No	IoT applications
Wasm3	70–75%	10	No	Resource-constrained devices
WasmEdge	85–90%	20	Yes	Edge computing, AI, Kubernetes-based services
Node.js (with Wasm)	75–80%	30	Yes	Server-side JavaScript, microservices
Blazor WebAssembly	70–75%	28	No	Web apps, cross-platform .NET applications
AssemblyScript	80–85%	25	Yes	Performance-critical TypeScript web applications
EOS VM	80–85%	20	Yes	Blockchain, smart contracts, decentralized apps
Parity Wasm	80–85%	22	Yes	Blockchain ecosystems, smart contract execution
SSVM	80–85%	25	Yes	AI workloads, cloud-native AI services
Life	70–75%	15	Yes	Scientific computing, reproducible research

Note: The execution speed values listed in Table 2 represent relative performance metrics, where each runtime's speed is measured against a baseline runtime. The baseline used for comparison is typically the fastest-performing Wasm runtime within the specific test environment or a commonly accepted reference. These baselines serve as a standard to express the relative efficiency of other runtimes, with the percentage values indicating how each runtime compares to the baseline in terms of execution speed.

Table 3
Integration capabilities of WebAssembly runtimes

Runtime	Integration efficiency (%)	Supported languages	Special features	Typical integration scenarios
V8	90	JavaScript, TypeScript, WebAssembly	Native APIs, extensive library support	Web applications, Node.js services
SpiderMonkey	80	JavaScript, WebAssembly	Developer tools, security features	Web development, gaming
JavaScriptCore	75	JavaScript, WebAssembly	Integration with Apple ecosystems	macOS/iOS applications
ChakraCore	70	JavaScript, WebAssembly	Legacy Edge support, Windows integration	Web apps (Edge legacy), Windows server applications
Wasmtime	85	Rust, C, C++, WebAssembly	WASI support	Serverless functions, microservices
Wasmer	80	Rust, Go, C, Python, WebAssembly	Multi-platform support, Wasmer ecosystems	IoT applications, blockchain, cross-platform apps
WAVM	80	C, C++, Rust	LLVM optimizations	Compute-heavy applications, scientific computing
Lucet	85	Rust, C	Fast startup, low latency	Cloud/serverless functions, containerized services
WAMR	70	C, C++	Microcontroller support, small footprint	Embedded systems, IoT
Wasm3	65	C, C++	Minimal memory usage	Microcontrollers, IoT devices
WasmEdge	85	C, C++, Rust, Go	Optimized for AI and edge	Edge computing, AI applications, Kubernetes services
Node.js (with Wasm)	80	JavaScript, WebAssembly	Native Node.js support	Server-side JavaScript, microservices
Blazor WebAssembly	75	C#, .NET	Full .NET runtime support	Web apps, cross-platform .NET applications
AssemblyScript	70	TypeScript, WebAssembly	JavaScript compatibility	High-performance web applications
EOS VM	80	WebAssembly	Blockchain optimizations	Blockchain, smart contracts, decentralized apps
Parity Wasm	75	WebAssembly	Substrate blockchain integration	Blockchain ecosystems, smart contracts
SSVM	80	WebAssembly, Rust, Python	AI/ML optimizations	AI workloads, cloud-based AI services
Life	70	C, C++, WebAssembly	Deterministic execution for research	Scientific computing, reproducible research

compatibility with existing systems, specialized features, and extensibility with third-party libraries. Table 3 provides an overview of integration efficiency, supported languages, special features, and typical integration scenarios for each Wasm runtime [8, 21–26].

V8 is highly efficient, supporting JavaScript, TypeScript, and Wasm, with native APIs and extensive library support for web and Node.js applications. SpiderMonkey, Mozilla's engine, is optimized for JavaScript and Wasm with developer tools and security features for web development and gaming. JavaScriptCore integrates seamlessly with Apple ecosystems, supporting JavaScript and Wasm, making it ideal for macOS/iOS applications. ChakraCore, though primarily used for legacy Edge support, integrates well with Windows for web apps and server-side processing, supporting JavaScript and Wasm. Wasmtime excels with Rust, C, C++, and Wasm, offering WASI support for serverless functions and microservices. Wasmer supports multiple languages like Rust, Go, C, Python, and Wasm, with multi-platform support for IoT, blockchain, and cross-platform applications. WAVM optimizes C, C++, and Rust with LLVM for compute-heavy applications and scientific computing. Lucet, built for fast startup and low latency, supports Rust and C, making it ideal for serverless functions and containerized services. WAMR is a lightweight runtime supporting C and C++ for embedded systems and IoT, with a small footprint. Wasm3 is a minimal runtime supporting C and C++ with ultra-low memory usage for microcontrollers and IoT devices. WasmEdge, optimized for AI and edge computing, supports multiple languages like C, C++, Rust, and Go, making it ideal for edge applications and Kubernetes services. Node.js, with Wasm support, integrates seamlessly into server-side JavaScript applications, while Blazor WebAssembly enables C# and .NET developers to build cross-platform web apps. AssemblyScript, focused on TypeScript, integrates well with JavaScript for performance-critical web apps. EOS VM, designed for the EOS blockchain, supports Wasm for smart contracts and dApps. Parity Wasm is optimized for Substrate blockchain integration, enhancing smart contract execution. SSVM supports Wasm, Rust, and Python, with AI/ML optimizations for

cloud-based AI workloads. Life, with C, C++, and Wasm support, focuses on deterministic execution for scientific computing and reproducible research. Each runtime is tailored to specific use cases, balancing language support, features, and integration capabilities for diverse application domains.

4.3. Application domains

Each Wasm runtime targets specific application domains based on its performance, integration capabilities, and specialized features [8, 21–25]. Table 4 highlights the primary application domains, industry examples, and notable projects for 18 Wasm runtimes. V8 supports web apps, gaming, and data visualization, with industry usage in Google Apps, Figma, and Unity WebGL, and frameworks like Angular and React. Wasmtime excels in cloud services and server-side applications, used by Fastly, Cloudflare, and GitHub Actions, powering serverless functions. WAVM is tailored for scientific computing and financial applications, supporting high-frequency trading and simulations, as well as AI and ML models. SpiderMonkey is optimized for interactive web applications, with Mozilla Firefox and WebGL games as industry examples, and various browser-based projects. JavaScriptCore is widely used for mobile web applications, supporting Safari and iOS apps, and is integrated into WebKit-based projects. WAMR is designed for IoT and edge computing, with applications in smart home devices and industrial IoT, and supports microcontrollers. Wasm3 operates in resource-constrained environments, serving wearables and embedded systems, particularly in low-power devices. WasmEdge focuses on edge computing and AI workloads, including AI inference at the edge and cloud-native apps, with Kubernetes-based AI services. ChakraCore supports legacy web and Windows integration, used by older Microsoft Edge versions and Windows server applications, providing essential support for Edge-based apps. Wasmer is used in blockchain and cross-platform applications, with various blockchain networks and ecosystems for blockchain development. Lucet is ideal for serverless, low-latency functions in fast cloud startup environments and serverless computing, implemented in containerized environments. Node.js (with Wasm) enables server-side JavaScript applications, often used in server backends and microservices, including Node-based Wasm integrations. Blazor WebAssembly supports cross-platform .NET applications, including browser-based .NET apps and hybrid client-server solutions, based on the Blazor framework. AssemblyScript is used for high-performance web apps, specifically in TypeScript-based apps needing optimized Wasm performance. EOS VM is dedicated to blockchain and decentralized apps, powering EOS smart contracts and dApps. Parity Wasm supports blockchain and smart contracts, integrated into Substrate and Polkadot frameworks for

Table 4
Application domains of WebAssembly runtimes

Runtime	Primary application domains	Industry examples	Notable projects/frameworks
V8	Web apps, gaming, data visualization	Google Apps, Figma, Unity WebGL	Angular, React
Wasmtime	Cloud services, server-side applications	Fastly, Cloudflare, GitHub Actions	Wasmtime-based serverless functions
WAVM	Scientific computing, financial applications	High-frequency trading, simulations	WAVM-based AI and ML models
SpiderMonkey	Interactive web applications	Mozilla Firefox, WebGL games	Various browser-based applications
JavaScriptCore	Mobile web applications	Safari, iOS apps	WebKit-based projects
WAMR	IoT, edge computing	Smart home devices, industrial IoT	WAMR in microcontrollers
Wasm3	Resource-constrained environments	Wearables, embedded systems	Low-power devices
WasmEdge	Edge computing, AI workloads	AI inference at the edge, cloud-native apps	Kubernetes-based AI services
ChakraCore	Legacy web and Windows integration	Older Microsoft Edge, Windows server applications	Legacy support for Edge-based apps
Wasmer	Blockchain, cross-platform applications	Various blockchain networks, cross-platform apps	Wasmer ecosystems for blockchain development
Lucet	Serverless, low-latency functions	Fast cloud startup environments, serverless computing	Lucet in containerized environments
Node.js (with Wasm)	Server-side JavaScript applications	Server backends, microservices	Node-based Wasm integrations
Blazor WebAssembly	Cross-platform .NET applications	Browser-based .NET apps, hybrid client-server solutions	Blazor framework
AssemblyScript	High-performance web apps	TypeScript-based apps needing optimized Wasm performance	AssemblyScript in various web apps
EOS VM	Blockchain, decentralized apps	EOS smart contracts, blockchain networks	EOS-based dApps
Parity Wasm	Blockchain, smart contracts	Substrate blockchain, decentralized applications	Substrate and Polkadot integrations
SSVM	AI and machine learning	AI workloads, cloud-based AI services	SSVM for ML model inference
Life	Scientific research, deterministic computing	Reproducible research experiments	Research-oriented projects in scientific computing

decentralized applications. SSVM focuses on AI and machine learning, used for AI workloads and cloud-based AI services, such as ML model inference. Finally, Life is specialized for scientific research and deterministic computing, used in reproducible research experiments and various research-oriented projects in scientific computing.

4.4. Security features of WebAssembly runtimes

Wasm runtimes implement a range of security features to ensure safe execution across different environments, from the web to resource-constrained devices and blockchain platforms. Key security mechanisms include sandboxing, memory safety, code isolation, and specific vulnerability mitigation techniques, all of which contribute to protecting applications from malicious code and unauthorized access [9, 11, 27–30]. Table 5 summarizes the security features across 18 Wasm runtimes, focusing on key aspects like sandboxing, memory safety, code isolation, and vulnerability mitigation techniques. All runtimes in the table implement sandboxing, ensuring isolated execution environments that prevent unauthorized access to host resources. V8 uses JIT compilation and

Table 5
Security features of WebAssembly runtimes

Runtime	Sandboxing	Memory safety	Code isolation	Vulnerability mitigation techniques
V8	Yes	Yes	Yes	JIT compilation, garbage collection
Wasmtime	Yes	Yes	Yes	WASI, security-focused design
WAVM	Yes	Yes	Yes	LLVM-based code verification
SpiderMonkey	Yes	Yes	Yes	Cross-origin policies, CSP
JavaScriptCore	Yes	Yes	Yes	Secure coding practices, sandboxing
ChakraCore	Yes	Yes	Yes	Cross-origin resource sharing
WAMR	Yes	Yes	Yes	Limited system calls, resource caps
Wasm3	Yes	Limited	Yes	Simple interpreter, isolation of execution
WasmEdge	Yes	Yes	Yes	Secure execution for AI, multi-tenant isolation
Lucet	Yes	Yes	Yes	Fast startup, lightweight isolation
Wasmer	Yes	Yes	Yes	Native binding security, sandboxed execution
Node.js (with Wasm)	Yes	Yes	Yes	JIT, memory protection, isolation in Node.js
Blazor WebAssembly	Yes	Yes	Yes	Browser-based sandboxing, secure client-side
AssemblyScript	Yes	Yes	Yes	Integration with JavaScript sandboxing
EOS VM	Yes	Yes	Yes	Blockchain-specific protections, reentrancy guard
Parity Wasm	Yes	Yes	Yes	Blockchain-based isolation, smart contract safety
SSVM	Yes	Yes	Yes	AI workload isolation, cloud-native security
Life	Yes	Yes	Yes	Deterministic execution for reproducibility

Table 6
Integration with cloud services

Runtime	Cloud platform integration	Serverless support	Container support	Deployment simplicity
V8	AWS Lambda, Google Cloud	Yes	Docker	High
Wasmtime	AWS Lambda, Azure Functions	Yes	OCI containers	Medium
WAVM	Limited	No	Limited	Low
SpiderMonkey	AWS Lambda	Yes	Docker	Medium
JavaScriptCore	Limited	Yes	Limited	Low
WAMR	Limited	No	Limited	Low
Wasm3	No	No	No	Low
WasmEdge	AWS Lambda, Google Cloud, Azure	Yes	OCI containers	Medium
Lucet	AWS Lambda, Azure Functions	Yes	Docker	Medium
Wasmer	AWS Lambda, Google Cloud	Yes	OCI containers	Medium
Node.js (with Wasm)	AWS Lambda, Google Cloud	Yes	Docker	High
Blazor WebAssembly	Azure Functions, AWS Lambda	Yes	Limited	Medium
AssemblyScript	Limited	No	Limited	Low
EOS VM	Limited	No	No	Low
Parity Wasm	Limited	No	No	Low
SSVM	AWS Lambda, Azure Functions	Yes	OCI Containers	Medium
Life	Limited	No	No	Low

garbage collection for vulnerability mitigation, while Wasmtime leverages WASI and a security-focused design. WAVM relies on LLVM-based code verification for ensuring safe execution. SpiderMonkey employs cross-origin policies and Content Security Policy (CSP) for secure web applications, while JavaScriptCore emphasizes secure coding practices alongside sandboxing. ChakraCore integrates cross-origin resource sharing as a security feature, and WAMR limits system calls and enforces resource caps for better control. Wasm3 is a simple interpreter with limited memory safety but isolates execution to minimize vulnerabilities. WasmEdge offers secure execution for AI workloads with multi-tenant isolation, while Lucet focuses on lightweight isolation, suitable for cloud and serverless environments. Wasmer integrates native binding security and sandboxed execution for better isolation. Node.js (with Wasm) ensures JIT, memory protection, and isolation within the Node.js environment. Blazor WebAssembly ensures browser-based sandboxing for secure client-side execution. AssemblyScript benefits from integration with JavaScript sandboxing to prevent unsafe operations. EOS VM offers blockchain-specific protection and includes a reentrancy guard for smart contract safety. Parity Wasm provides blockchain-based isolation and security features for smart contract execution. SSVM isolates AI workloads and offers cloud-native security for edge applications. Finally, Life focuses on deterministic execution, providing reproducibility for scientific computing with added security controls.

4.5. Integration of WebAssembly runtimes with cloud services

Wasm runtimes exhibit varying levels of integration with cloud services, enabling developers to leverage their capabilities in serverless architectures and containerized environments. The ability to easily deploy Wasm modules across different cloud services significantly enhances their appeal for developing scalable, efficient applications that can leverage the power of both cloud computing and Wasm [19, 31–34]. Table 6 provides an overview of how each Wasm runtime integrates with cloud platforms, highlighting their support for serverless functions, containerization, and deployment simplicity. V8 offers high integration with both AWS Lambda and Google Cloud, supports serverless functions, and is compatible with Docker, making it highly deployable in cloud environments. Similarly, Node.js (with Wasm) enjoys high integration with AWS Lambda and Google Cloud, with Docker support and seamless deployment, making it very cloud-friendly. Wasmtime supports serverless functions on AWS Lambda and Azure Functions and can run in open container initiative (OCI) containers, though its deployment simplicity is moderate. WasmEdge has strong cloud integration with AWS Lambda, Google Cloud, and Azure, supporting OCI containers, but its deployment complexity is medium. Lucet, Wasmer, and SSVM also support AWS Lambda and Azure Functions and containerization via Docker and OCI containers, with medium deployment complexity. SpiderMonkey integrates with AWS Lambda and Docker, supporting serverless functions with a medium deployment level. Blazor WebAssembly integrates with AWS Lambda and Azure Functions, but its container support is limited, leading to medium deployment simplicity. JavaScriptCore, WAMR, Life, EOS VM, and Parity Wasm all have limited cloud platform integration, lack serverless or container support, and have low deployment simplicity, making them less suitable for cloud-native environments. Wasm3 has no cloud platform integration, serverless support, or containerization, resulting in low deployment flexibility.

4.6. Performance of WebAssembly runtimes under different workloads

The performance of Wasm runtimes varies significantly under different workloads, reflecting their optimization for specific use cases. Understanding these differences is essential for developers when selecting a Wasm runtime for their specific application needs [35–38]. Table 7 provides the performance of Wasm runtimes under different workloads, including details about compute workload, I/O workload, memory allocation, and threading support. V8 stands out for its fast performance in both compute (50 ms) and I/O (30 ms), requiring 40 MB of memory and supporting threading, making it efficient for performance-critical applications like web apps and gaming. Similarly, Wasmtime performs well in compute (60 ms) and I/O (25 ms), with moderate memory usage (30 MB) and threading support, positioning it as a strong option for cloud services and server-side applications. WAVM and SpiderMonkey have similar performance characteristics, with WAVM taking slightly longer on compute (70 ms) and I/O (35 ms) but requiring less memory (25 MB) and supporting threading. SpiderMonkey performs slightly better in I/O (40 ms) but with higher memory usage (35 MB). Both are suitable for web development and gaming. JavaScriptCore shows slightly higher latency in both compute (65 ms) and I/O (45 ms) but with similar memory allocation (30 MB). It supports threading, making it a reasonable choice for mobile web applications. WAMR, Wasm3, and Blazor WebAssembly show slower compute and I/O times, with WAMR taking 80 ms for compute and 50 ms for I/O. However, WAMR has low memory usage (15 MB) but lacks threading support, making it suitable for IoT applications where low memory is a priority. Wasm3 and Blazor WebAssembly both have high compute (90 ms and 75 ms, respectively) and I/O (60 ms and 40 ms) latencies and low memory allocations but lack threading support, limiting their performance in multi-threaded environments. WasmEdge and Lucet both perform well in compute and I/O (55 ms and 25 ms, respectively), with memory usage ranging from 20 MB to 28 MB. WasmEdge is geared toward edge computing and AI workloads, while Lucet excels in low-latency cloud functions. Wasmer, Node.js (with Wasm), and EOS VM show solid performance with moderate latencies (58–72 ms) in compute and I/O. These runtimes also support threading, making them viable for cross-platform and server-side applications. AssemblyScript and Life have the highest compute and I/O latencies (85 ms and 80 ms), making them less ideal for real-time or low-latency applications. However, AssemblyScript targets performance-critical TypeScript applications, and Life focuses on reproducible research.

4.7. Performance benchmarks for WebAssembly runtimes across different environments

Performance benchmarks for Wasm runtimes reveal significant variations across different environments [23, 39, 40]. Table 8 provides performance benchmarks for Wasm runtimes across browser and server environments, focusing on key metrics such as browser and server performance, startup time, and memory usage. V8 stands out for its low browser performance of 30 ms and excellent server performance of 20 ms, along with the fastest startup time of 10 ms, though its memory usage is higher at 40 MB. Wasmtime offers solid server performance (30 ms) and startup time (15 ms) with moderate memory usage (30 MB), making it a good choice for cloud services. WAVM and SpiderMonkey perform well in both environments, with WAVM showing higher latency (50 ms for browser and 40 ms for server) but lower memory usage (25 MB), while SpiderMonkey has 35 ms browser performance

Table 7
Performance of WebAssembly runtimes under different workloads

Runtime	Compute workload (ms)	I/O workload (ms)	Memory allocation (MB)	Threading support (Yes/No)
V8	50	30	40	Yes
Wasmtime	60	25	30	Yes
WAVM	70	35	25	Yes
SpiderMonkey	55	40	35	Yes
JavaScriptCore	65	45	30	Yes
WAMR	80	50	15	No
Wasm3	90	60	10	No
WasmEdge	55	30	28	Yes
Lucet	65	25	20	Yes
Wasmer	58	33	27	Yes
Node.js (with Wasm)	52	28	38	Yes
Blazor WebAssembly	75	40	32	No
AssemblyScript	85	55	18	No
EOS VM	72	42	22	Yes
Parity Wasm	68	38	26	Yes
SSVM	64	35	24	Yes
Life	80	45	15	No

Table 8
Performance benchmarks for WebAssembly runtimes across different environments

Runtime	Browser performance (ms)	Server performance (ms)	Startup time (ms)	Memory usage (MB)
V8	30	20	10	40
Wasmtime	40	30	15	30
WAVM	50	40	20	25
SpiderMonkey	35	25	12	35
JavaScriptCore	45	35	18	30
WAMR	60	50	25	15
Wasm3	70	60	30	10
WasmEdge	32	28	14	28
Lucet	38	32	16	22
Wasmer	34	29	15	26
Node.js (with Wasm)	31	27	11	38
Blazor WebAssembly	42	38	20	32
AssemblyScript	37	34	22	18
EOS VM	49	45	23	23
Parity Wasm	48	46	21	24
SSVM	39	31	17	27

and 25 ms server performance, with a memory consumption of 35 MB. WAMR and Wasm3 are optimized for resource-constrained environments, with WAMR showing 60 ms in the browser and 50 ms on the server but minimal memory usage (15 MB), while Wasm3 is less performant with 70 ms in the browser and 60 ms on the server, though it uses only 10 MB of memory, making it ideal for IoT devices. WasmEdge, Lucet, and Wasmer balance performance and memory usage, with WasmEdge providing 32 ms browser performance and 28 ms server performance while consuming 28 MB of memory. Lucet excels in low startup time (16 ms) and memory efficiency (22 MB), making it suitable for serverless functions. Node.js (with Wasm) offers low startup times (11 ms) and a moderate memory usage (38 MB), making it a strong contender for server-side JavaScript applications. Blazor WebAssembly, while offering a slightly higher browser latency (42 ms) and server latency (38 ms), supports a .NET environment with medium memory consumption (32 MB). AssemblyScript, EOS VM, Parity Wasm, and SSVM demonstrate higher latency in both browser and server environments compared to other runtimes, with AssemblyScript showing 37 ms in the browser and 34 ms on the server, while EOS VM, Parity Wasm, and SSVM are also above the typical latency benchmarks. However, these runtimes are specifically optimized for niche use cases, such as AssemblyScript for high-performance TypeScript web applications, EOS VM and Parity Wasm for blockchain and smart contract execution, and SSVM for AI and machine learning workloads. Their higher latency is a trade-off for the specialized features they provide, making them well-suited for applications in their respective domains, where performance characteristics such as execution precision and ecosystem compatibility are prioritized over raw latency.

5. Future Research and Directions

While the current study contributes valuable knowledge regarding the performance, integration, and security of Wasm runtimes, it also highlights certain limitations that warrant further investigation. The benchmarking conducted here primarily focuses on synthetic workloads, which may not fully capture the complexities inherent in real-world applications. To address this gap, future research should extend the analysis to practical use cases, particularly those involving large-scale distributed systems or AI/ML workloads, where deeper insights into runtime performance are crucial. Additionally, while the current research explores integration capabilities, there remains a significant opportunity to examine the developer experience and ecosystem support for each runtime. Investigating runtime behavior within specific domains – such as edge computing, decentralized applications, and blockchain – could provide valuable insights into the scalability and reliability of each Wasm runtime in specialized environments. Furthermore, future studies should explore the potential synergy between Wasm runtimes and emerging technologies, including serverless computing and quantum computing. Such research could reinforce Wasm's role in the next generation of application development by enhancing interoperability and flexibility across diverse platforms, thus positioning it as a critical component of modern computing ecosystems.

6. Conclusion

This review shows a comparative analysis of various Wasm runtimes, examining their performance, integration capabilities, security features, and suitability for diverse application domains. Through an evaluation of key metrics such as execution speed, memory consumption, integration flexibility, and security mechanisms, the study provides valuable insights to inform the selection of the most appropriate Wasm runtime for specific use cases. The findings indicate significant variability in the performance and suitability of Wasm runtimes across different environments and workloads. Runtimes such as V8 and Lucet are shown to excel in execution speed and memory efficiency, making them ideal candidates for server-side and cloud-based applications. Conversely, runtimes like WAMR and Wasm3 are more suited for constrained environments such as IoT and embedded systems, where minimizing resource consumption is paramount. Additionally, Wasmtime and Wasmer stand out for their strong integration capabilities, particularly within cloud-native services, while Blazor WebAssembly and AssemblyScript are optimal for .NET and TypeScript-based web applications, respectively. Despite these strengths, several runtimes exhibit limitations, particularly in the areas of multi-threading support, JIT compilation, and integration with certain cloud services. These limitations underscore the need for further research to improve the versatility of Wasm runtimes, especially in emerging application domains such as AI, machine learning, and large-scale distributed systems.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by the author.

Conflicts of Interest

The author declares that he has no conflicts of interest to this work.

Data Availability Statement

Data are available from the corresponding author upon reasonable request.

Author Contribution Statement

Mircea Ţălu: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration.

References

[1] Eich, B. (2015). *From ASM.JS to WebAssembly*. https://brendaneich.com/2015/06/from-asm-js-to-WebAssembly

[2] Herman, D., Wagner, L., & Zakai, A. (2014). *asm.js*. http://asmjs.org/

[3] Perrone, G., & Romano, S. P. (2025). WebAssembly and security: A review. *Computer Science Review*, *56*, 100728. https://doi.org/10.1016/j.cosrev.2025.100728

[4] WebAssembly. (n.d.). *WebAssembly official documentation*. https://WebAssembly.org

[5] Rossberg, A., Titzer, B. L., Haas, A., Schuff, D., Gohman, D., Wagner, L., ..., & Holman, M. (2018). Bringing the web up to speed with WebAssembly. *Communications of the ACM*, *61*(12), 107–115. https://doi.org/10.1145/3282510

[6] Mendki, P. (2020). Evaluating WebAssembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*, 161–166. https://doi.org/10.1109/IEEECloudSummit48914.2020.00031

[7] Hoque, M. N., & Harras, K. A. (2022). WebAssembly for edge computing: Potential and challenges. *IEEE Communications Standards Magazine*, *6*(4), 68–73. https://doi.org/10.1109/MCOMSTD.0001.2000068

[8] Ray, P. P. (2023). An overview of WebAssembly for IoT: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, *15*(8), 275. https://doi.org/10.3390/fi15080275

[9] Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything old is new again: Binary security of WebAssembly. In *Proceedings of the 29th USENIX Security Symposium*, 217–234.

[10] Yan, Y., Tu, T., Zhao, L., Zhou, Y., & Wang, W. (2021). Understanding the performance of WebAssembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, 533–549. https://doi.org/10.1145/3487552.3487827

[11] Dejaeghere, J., Gbadamosi, B., Pulls, T., & Rochet, F. (2023). Comparing security in ebpf and WebAssembly. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, 35–41. https://doi.org/10.1145/3609021.3609306

[12] Ţălu, M. (2024). A review of vulnerability discovery in WebAssembly binaries: Insights from static, dynamic, and hybrid analysis. *Acta Technica Corviniensis-Bulletin of Engineering*, *17*(4), 13–22.

[13] Ţălu, M. (2025). A review of advanced techniques for data protection in WebAssembly. *Annals of the Faculty of Engineering Hunedoara-International Journal of Engineering*, *22*(4), 131–136.

[14] Kakati, S., & Brorsson, M. (2023). WebAssembly beyond the web: A review for the edge-cloud continuum. In *2023 3rd International Conference on Intelligent Technologies*, 1–8. https://doi.org/10.1109/CONIT59222.2023.10205816

[15] Zhang, Y., Liu, M., Wang, H., Ma, Y., Huang, G., & Liu, X. (2025). Research on WebAssembly runtimes: A survey. *ACM*

Transactions on Software Engineering and Methodology, *34*(8), 239. https://doi.org/10.1145/3714465

[16] Zhang, Y., Cao, S., Wang, H., Chen, Z., Luo, X., Mu, D., ... & Liu, X. (2023). Characterizing and detecting WebAssembly runtime bugs. *ACM Transactions on Software Engineering and Methodology*, *33*(2), 37. https://doi.org/10.1145/3624743

[17] Watt, C. (2018). Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 53–65. https://doi.org/10.1145/3167082

[18] Górski, T. (2020). Verification of architectural views model 1+5 applicability. In *Computer Aided Systems Theory: EUROCAST 2019*, 499–506. https://doi.org/10.1007/978-3-030-45093-9_60

[19] Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V., Mazzeo, G., Hollum, A., & Vaydia, D. (2024). A comprehensive trusted runtime for WebAssembly with intel SGX. *IEEE Transactions on Dependable and Secure Computing*, *21*(4), 3562–3579. https://doi.org/10.1109/TDSC.2023.3334516

[20] de Macedo, J., Abreu, R., Pereira, R., & Saraiva, J. (2021). On the runtime and energy performance of WebAssembly: Is WebAssembly superior to JavaScript yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 255–262. https://doi.org/10.1109/ASEW52652.2021.00056

[21] Wang, W. (2022). How far we've come: A characterization study of standalone WebAssembly runtimes. In *2022 IEEE International Symposium on Workload Characterization*, 228–241. https://doi.org/10.1109/IISWC55918.2022.00028

[22] Gackstatter, P., Frangoudis, P. A., & Dustdar, S. (2022). Pushing serverless to the edge with WebAssembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*, 140–149. https://doi.org/10.1109/CCGrid54584.2022.00023

[23] Jiang, S., Zeng, R., Rao, Z., Gu, J., Zhou, Y., & Lyu, M. R. (2023). Revealing performance issues in server-side WebAssembly runtimes via differential testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, 661–672. https://doi.org/10.1109/ASE56229.2023.00088

[24] Zhou, S., Jiang, M., Chen, W., Zhou, H., Wang, H., & Luo, X. (2023). WADIFF: A differential testing framework for WebAssembly runtimes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, 939–950. https://doi.org/10.1109/ASE56229.2023.00188

[25] Gottschalk, F., Schulte, S., Hemadasa, N., Ebrahimi, E., Edinger, J., & Kaaser, D. (2025). Towards WebAssembly-based federated learning. In *Service-Oriented and Cloud Computing: 11th IFIP WG 6.12 European Conference*, 40–54. https://doi.org/10.1007/978-3-031-84617-5_4

[26] Kim, M., Jang, H., & Shin, Y. (2022). Avengers, assemble! Survey of WebAssembly security solutions. In *2022 IEEE 15th International Conference on Cloud Computing*, 543–553. https://doi.org/10.1109/CLOUD55607.2022.00077

[27] Johnson, E., Laufer, E., Zhao, Z., Gohman, D., Narayan, S., Savage, S., ..., & Brown, F. (2023). WaVe: A verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy*, 2940–2955. https://doi.org/10.1109/SP46215.2023.10179357

[28] Protzenko, J., Beurdouche, B., Merigoux, D., & Bhargavan, K. (2019). Formally verified cryptographic web applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy*, 1256–1274. https://doi.org/10.1109/SP.2019.00064

[29] Legoupil, M., Rousseau, J., Georges, A. L., Pichon-Pharabod, J., & Birkedal, L. (2024). Iris-MSWasm: Elucidating and mechanising the security invariants of memory-safe WebAssembly. In *Proceedings of the ACM on Programming Languages*, *8*, 304–332. https://doi.org/10.1145/3689722

[30] Tsoupidi, R. M., Balliu, M., & Baudry, B. (2021). Vivienne: Relational verification of cryptographic implementations in WebAssembly. In *2021 IEEE Secure Development Conference*, 94–102. https://doi.org/10.1109/SecDev51306.2021.00029

[31] Kakati, S., & Brorsson, M. (2024). A cross-architecture evaluation of WebAssembly in the cloud-edge continuum. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing*, 337–346. https://doi.org/10.1109/CCGrid59990.2024.00046

[32] Li, B., Fan, H., Gao, Y., & Dong, W. (2022). Bringing WebAssembly to resource-constrained IoT devices for seamless device-cloud integration. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 261–272. https://doi.org/10.1145/3498361.3538922

[33] Marcelino, C., & Nastic, S. (2024). CWASI: A WebAssembly runtime shim for inter-function communication in the serverless edge-cloud continuum. In *Proceedings of the Eighth ACM/IEEE Symposium on Edge Computing*, 158–170. https://doi.org/10.1145/3583740.3626611

[34] Jain, S. M. (2022). *WebAssembly for cloud: A basic guide for wasm-based cloud apps*. USA: Apress. https://doi.org/10.1007/978-1-4842-7496-5

[35] Wagner, L., Mayer, M., Marino, A., Soldani Nezhad, A., Zwaan, H., & Malavolta, I. (2023). On the energy consumption and performance of WebAssembly binaries across programming languages and runtimes in IoT. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, 72–82. https://doi.org/10.1145/3593434.3593454

[36] Long, Y., Su, Y., & Jiang, Z. (2024). WACP: A performance profiling tool for WebAssembly-Python interoperability. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, 495–498. https://doi.org/10.1145/3671016.3671408

[37] Kyriakou, K. I. D., & Tselikas, N. D. (2022). Complementing javascript in high-performance Node.js and web applications with rust and WebAssembly. *Electronics*, *11*(19), 3217. https://doi.org/10.3390/electronics11193217

[38] Szewczyk, R., Stonehouse, K., Barbalace, A., & Spink, T. (2022). Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization*, 256–268. https://doi.org/10.1109/IISWC55918.2022.00030

[39] Spies, B., & Mock, M. (2021). An evaluation of WebAssembly in non-web environments. In *2021 XLVII Latin American Computing Conference*, 1–10. https://doi.org/10.1109/CLEI53233.2021.9640153

[40] Tushar, & Mohan, B. R. (2022). Comparative analysis of JavaScript and WebAssembly in the browser environment. In *2022 IEEE 10th Region 10 Humanitarian Technology Conference*, 232–237. https://doi.org/10.1109/R10-HTC54060.2022.9929829

How to Cite: Țălu, M. (2026). A Comparative Study of WebAssembly Runtimes: Performance Metrics, Integration Challenges, Application Domains, and Security Features. *Archives of Advanced Engineering Science*, *4*(3), 275–287. https://doi.org/10.47852/bonviewAAES52024965