Khazar Journal of Science and Technology `Volume 9 №1 2025, 32-46
© Khazar University Press 2025 DOI: 10.5782/2520-6133.2025.9.1.32

A survey of Web Assembly Runtime Ecosystem

Vugar Hasanov, Sallar Khan, Tania Malik, Khalid Hasanov
Faculty of Engineering and Applied Sciences, Khazar University, Azerbaijan
School of Informatics and Cybersecurity, Technological University Dublin, Ireland
Corresponding author: vugar.hasanov96@gmail.com

Abstract

WebAssembly is a secure, portable, low-level binary code format that offers efficient execution and compact representation. With the intro- duction of the WebAssembly System Interface, Wasm is expanding beyond its original role as a browser-only technology, finding applications in diverse fields such as generative AI, serverless AI, edge computing, cloud computing, and high-performance computing. Despite the existence of numerous Wasm run- time ecosystems, there is currently no comprehensive overview or taxonomy of these state-of-the-art runtimes and their ecosystems. This paper aims to fill that gap by providing a detailed taxonomy of Wasm runtime ecosystems, ex- amining runtime characteristics and classifying modern Wasm runtimes within this framework.

Keywords: WebAssembly, Wasm, WASI, Wasm Runtimes

Introduction

WebAssembly (Wasm) is a safe, portable, low-level binary-code format designed for efficient execution and compact representation. Introduced in 2015 as an al- ternative to JavaScript for web browser-based applications (WebAssembly Contributors, 2015). Wasm serves as a cross-platform compilation target for languages traditionally not executed on the web, such as C/C++ (WebAssembly Working Group, 2022). It has been developed as an open standard by the World Wide Web Consortium (W3C), and all major browsers support it (McConnell, 2017). In particular, most modern web browsers enable the loading of Wasm modules through JavaScript, facilitating nearly native performance in web application vir- tual machines. Since Wasm was originally designed for web browsers, it lacked a system interface targeting POSIX environments, which would enable the execu- tion of Wasm modules on these environments as part of its original specification (Andreas, et al., 2017). To address this, the WebAssembly System Interface (WASI) specification was designed (WebAssembly Community Group, 2021). With the introduction of WASI, WebAssembly has begun to tran- scend its browser-only limitations, evolving from a technology exclusively used in browsers to one that spans various application areas, including generative AI, serverless computing, edge computing, cloud computing, and high-performance computing. WASI provides a standardized set of system interfaces for WebAssem- bly modules, enabling them to interact with the operating system in a secure and portable manner. Its sandbox execution environment and portability also make Wasm an ideal candidate for non-web use cases. The environment in which We- bAssembly code executes is called Wasm runtime. These runtimes are crucial because they parse and execute the compiled WebAssembly modules, allowing them to run as efficiently and securely as possible.

A large number of Wasm runtimes have been developed over the past couple of years, and Wasm runtimes have started to be incorporated within or extend higher-level system software. While the usage of Wasm and Wasm runtimes within web browsers are still dominant, the focus of this work is to categorize WebAssem- bly runtimes which are in use outside web browsers. It is worth emphasising that we intentionally do not cover Wasm runtimes that are not actively maintained anymore and do not have commit histories in their GitHub repositories within the last one year.

Unlike previous works in this domain, this paper does not focus solely on stan- dalone Wasm runtimes. Instead, it also includes new emerging wrapper runtimes that are built on top of a standalone runtime and extend it with new functionalities for distributed hosts and server-side application development. While it could be argued that wrapper runtimes are not fundamentally WebAssembly runtimes, the wrapper runtimes examined in this work are specifically designed for WebAssem- bly. They extend a single host WebAssembly runtime with additional layers of security, a distributed application platform, an observability layer, RPC (remote procedure call) capabilities, channel-based message passing, massive concurrency control with their own runtime scheduler, and more. Therefore, these wrapper runtimes deserve to be mentioned and included within a broader classification of WebAssembly runtimes.

For the purpose of this work, we define a WebAssembly runtime as follows: A WebAssembly runtime is a software platform that features a virtual machine executing the WebAssembly instruction set according to the WebAssembly specification. Execution can be managed directly by the runtime itself, or, in more complex designs, it may rely on another We- bAssembly runtime underneath while adding additional functionalities on top.

The Wasm ecosystem and standards are evolving rapidly, which naturally forces Wasm runtimes to catch up with the new developments. Not every Wasm run- time implements and support all Wasm or WASI standards. For instance, some runtimes, such as Wasmer (2024) and Wasmedge (2024) implement the proposal (Bytcode Alliance-a, 2024) to add garbage collection (GC) support to WebAssembly, while Wasmtime (Bytcode Alliance-b, 2024) does not have a stable implementation yet. Similarly, APIs and integrations provided by different runtimes are different, some of them provides a wide range of program- ming language SDKs, while others are designed with a specific language in mind.

The landscape of Wasm runtimes, already rich with diversity, is further ex- panded by the emergence of advanced wrapper runtimes, like Lunatic (Lunatic Inc, 2023) and wasmCloud (WasmCloud LLC, 2024). Both of these runtimes rely on Wasmtime as their foundation, offering higher-level abstractions to facilitate the creation of distributed applica- tions utilizing WebAssembly.

There have been some recent efforts to provide a taxonomy of WebAssembly runtimes and Wasm ecosystem. While the research in (Zhang, et al., 2024) does a thorough study on this front, the focus of that work is to study and classify the previous academic research efforts in the Wasm runtime implementations. However, there is a notable absence of an updated and thorough classification of more recent Wasm runtimes and higher level runtimes that are based on standalone Wasm runtimes. This absence leaves researchers and developers interested in WebAssembly runtimes without a clear understanding of the alternatives to traditional non-WebAssembly technologies. In this study, we aim to fill this gap by presenting a taxonomy of standalone runtimes, as well as, new emerging higher level wrapper and inte- grated runtimes; We validate the usefulness of this taxonomy by applying it to categorize several WebAssembly runtimes. While these runtimes vary in maturity and stability, they collectively offer insight into the landscape of WebAssembly runtimes in use. Broadly, we define each WebAssembly runtime as consisting of two main components: a programming interface (API) and a runtime system. The API serves as the interface between the environment and the programmer, while the runtime system comprises the underlying implementation mechanisms. Additionally, the runtime itself may include various subcomponents to fulfill its functionalities. In this study, we concentrate on the runtime systems, reserving the classification of APIs for future research.

1. WebAssembly Runtimes

WebAssembly is a binary instruction format that runs within a stack-based virtual machine (VM). The stack-based VM records operand values and control constructs. In addition, it provides an abstract store for the global state. The stack-based virtual machines are implemented by different WebAssembly run- times.

In this section, we classify existing WebAssembly runtimes into five broad cat- egories: type, architecture, platform, compilation mode, and finally execution environment.

1.1. WebAssembly Runtime Types

The first classification item in this category is the type of the Wasm runtime. Not all the Wasm runtimes are designed the same and there are fundamental differences in the way they have been implemented and the way they can be used to build applications. With this in mind, we classify the state-of-the-art WebAssembly runtimes into standalone, integrated, and wrapper runtimes. It is important to note that this classification is one of the primary distinction between this work and previous research in this field.

(1) Standalone WebAssembly Runtimes

The standalone Wasm runtimes have zero or less dependencies and usually do not assume the existence of other runtimes to execute Wasm binaries. This makes them ideal for environments where a lightweight and self- contained execution environment is necessary.

(2) Integrated Webassembly Runtimes

Unlike the standalone Wasm runtimes, the integrated runtimes are de- signed in a way that they cannot run on their own and need to be ex- ecuted within the context of another runtime. The another runtime in this context is not necessarily a runtime dedicated for WebAssembly. For instance, there are integrated Wasm runtimes based on JDK or Go pro- gramming language runtimes and cannot operate on their own without the corresponding language runtimes.

(3) Webassembly Wrapper Runtimes

A wrapper runtime is based on a standalone Wasm runtime and extends it with different features, such as distributed application development, net- working, process supervision, hot reloading, and so on. This approach helps to abstract Wasm runtimes further and enables their usage beyond a single host. Wrapper runtimes are typically used to simplify the de- velopment and deployment of complex applications that need to operate across multiple environments or require enhanced functionalities and in- teractions.

Unfortunately, the industry terminology for standalone Wasm runtimes is convoluted and inconsistent. For example, the state-of-the-art server- side WebAssembly runtime, Spin (Fermyon Technologies-a, 2024), is variously described as an ”open source developer tool for building and running serverless applications pow- ered by WebAssembly” and as ”a framework for building, deploying, and running fast, secure, and composable cloud microservices with WebAssem- bly.” In scientific research, it is crucial to clearly differentiate between terms like ”developer tool” and ”framework.” A closer examination of Spin’s features reveals that such a complex system merits being classified as a runtime in its own right. Indeed, the core developers of the Spin project refer to it as a runtime for server-side WebAssembly in GitHub is- sue discussions (Spin (Fermyon Technologies-b, 2024). Therefore, we hope that by introducing the term ”We- bAssembly Wrapper Runtime” and classifying corresponding runtimes as such, this work will help standardize terminology in both the industry and the research community.

1.2. Architecture

As the name suggests, this feature covers CPU architectures supported by Wasm runtimes. We broadly classify the supported architectures into 32-bit and 64-bit systems. The rationale to group Wasm runtimes into these two categories is that, if a runtime supports a specific 32-bit architecture, it usually easily be extended to support another 32-bit architecture. 32-bit architectures usually used in embedded-processors and constrained edge devices.

1.3. Platform

This criteria covers platforms supported by different Wasm runtimes. The list of platforms classified are Android, FreeBSD, iOS, Linux, macOS, Windows. In addition, we broadly cover real-time and embedded operating systems platforms under this category.

1.4. Compilation Mode

WebAssembly runtimes can operate in different compilation modes: Ahead-Of- Time (AoT), Just-In-Time (JIT), and Interpreted.

(1) Ahead-of-Time Compilation(AoT)

Whether the runtime supports ahead-of-time compilation of Webassembly programs into a Wasm binary before execution of the program at a build-time. AoT compilation usually reduces the amount of work needed to be performed at run time. The benefits of AoT are faster startup times and predictable performance.

(2) Interpreted

Whether the runtime can simply interpret the Wasm code. The main benefit of this mode is that, it simplifies implementation and allows for immediate execution without the need for compilation. It generally has slower execution speed compared to AoT and JIT.

(3) Just-in-Time compilation (JIT)

Whether the runtime supports just-in-time (JIT) compilation. In the JIT compilation mode the Wasm program is compiled during execution of the program at run time, rather than before its execution. The benefit of JIT is potentially better optimization based on runtime information. However, it may introduce overhead due to the compilation step during execution.

1.5. Execution Environment

The execution environment of a Wasm runtime is a criteria to indicate whether the runtime is designed to run on a single host only or on a distributed set of nodes.

(1) Single host

Whether the Wasm runtime is designed for a single host

(2) Distributed

Whether the Wasm runtime can run on a distributed set of nodes With these definitions in place, in this work, we provide a set of characterizing features for Webassembly runtimes which encompasses all relevant aspects while remaining as compact as possible; Figure 1 presents our Wasm runtime taxonomy.

2. WebAssembly Runtime Classification

In the previous section, we defined categories for Wasm runtimes. However, we have not classified existing runtimes into those categories. That is the scope of this section where we classify each of the state-of-the-art Wasm runtime we studied.

2.1. Runtime Types

(1) Standalone Runtimes

The leading examples of standalone Wasm runtimes include Wasmtime, Wasmer, Wasmi, Wasm3, WasmEdge, and Wazero. While these runtimes differ in the way they have been implemented and the features they all provide, they all have a common graund; these runtimes are implemented

(2) Integrated Runtimes

A prominent example for integrated Wasm runtimes is Chicory (Chicory Runtime, 2024), which is a JVM native WebAssembly runtime. Chicory can run Wasm binaries on any platform where a JVM can run.

GraalWASM (Oracle, 2024) is another example of an integrated Wasm runtime that is built on top of GraalVM. It can interpret and compile Wasm pro- grams in the binary format, or be embedded into other programs.

Wazero (2024) is another actively maintained Wasm runtime and it is built on top of Go runtime. Wazero does not use shared libraries or libc, this means it can also be embedded into applications that do not use an oper- ating system.

(3) Wrapper Runtimes

Lunatic (2023) is a state-of-the-art wrapper runtime. It is written in Rust and uses Wasmtime to create processes as WebAssembly instances. It was inspired by Erlang and can be used to build highly-concurrent and distributed server side applications. The massive concurrency support is provided via the robust work stealing scheduler implementation in Tokio, Rust's asynchronous runtime.

Spin (Fermyon Technologies-a, 2024) is a server-side WebAssembly runtime built to run applications compiled to WebAssembly.

Another actively developed wrapper runtime is WasmCloud (2024) which is based on Wasmtime as well.

Wasmex (2024) is another actively developed wrapper runtime that uses Wasmtime underneath. It enables the execution of Wasm binaries within Elixir programs through the utilization of the Wasmtime Rust library.

Extism (2024) is a lightweight framework for building Wasm applications and running them both on the browsers and servers, including the edge, command line interfaces, the internet of things. While Extism can be seen as a feature full framework providing support for persistent memory variables, secure and host-controlled HTTP, runtime limiters and timers, simpler host function linking, and so forth, it can also be classified as a Wasm wrapper runtime. As a matter of fact, it provides its own runtime implemented as a Rust crate and uses Wasmtime underneath to extend it with the mentioned functionalities. Moreover, Extism can use different standalone Wasm runtimes underneath. Namely, it uses Wasmtime for host SDKs, and Wazero for Go based applications. Also, for the browser runtimes, it can use V8 in Node/Deno, and JSC for Bun. Therefore, it very well belongs to our definition of wrapper runtime.

MPIWasm (Mohak, et al., 2023) is an academic wrapper runtime based on Wasmer and allows to execute high-performance MPI applications compiled to Wasm binaries on distributed memory platforms. The runtime supports the ex- ecution of MPI-2.2 standard applications written in C/C++.

2.2. Supported Architectures and Platforms

WebAssembly runtimes have evolved to support a wide range of architectures and platforms, enabling developers to run Wasm code efficiently across different environments. They offer robust support for various operating systems, such as Linux, Windows, macOS, and more, as well as compatibility with architectures like x86, ARM, and RISC-V. Below section categorizes various Wasm runtimes according to the platforms and architectures they support.

(1) Chicory

Chicory runtime is written purely in Java, supports running on any ar- chitecture and platform that the underlying JVM runtime supports. It does not rely on platform-specific native dependencies, making it highly portable and suitable for various environments.

(2) Extism

Architecture: x64, aarch64, x86 Platform: Windows, Linux, MacOS

(3) Lunatic

Architecture: X86-64, aarch64 Platform: Windows, Linux, MacOS

(4) WAMR

Architecture: X86-64, X86-32, ARM, AArch64, RISCV64, RISCV32

Platform: Windows, Linux, MacOS, Android

(5) Wasmedge

Architecture: X86-64, AArch64, RISCV64, Raspberry Pi Platform:
Windows, Linux, MacOS, Android

(6) Wasmtime

Architecture: X86-64, AArch64, RISCV64, s390x Platform: Windows x86-64, Linux x86-64, MacOS x86-64

Wasmtime uses cranelift code generator to compile wasm binary into na- tive machine code, and cranelift only supports 64 bits platforms. Due to this, no 32 bit platform is supported by wasmtime yet.

(7) Wasm3

Architecture: X86-64, X86-32, ARM, RISCV64, RISCV32, Raspberry Pi Platform: Windows, Linux, MacOS, FreeBSD, Android, IOS

(8) Wasmer

Architecture: X86-64,X86-32, ARM64, RISCV64, M1

Platform: Windows, Linux, MacOS, Android, IOS

(9) wasmCloud

Architecture: x86-64, aarch64 (ARM64)

Platform: Windows, Linux, MacOS

(10) Spin

Architecture: X86-64, ARM64 Platform: Windows, Linux, MacOS

(11) Wazero

Wazero is similar to Chicory runtime as it doesn't have a built-in platform dependency in mind. It is written in Go programming language and can be run in any environment supported by Go.

Based on this information we can summarise this section that, Wasmtime, and the runtimes that are built on top of Wasmtime do not support 32-bit architec- tures. Wasmtime uses Cranelift as its compiler backend. While Cranelift supports various architectures, its primary focus and optimizations are directed towards 64- bit systems. The rest of the runtimes provide a 32-

bit architecture support to some extent. 32-bit architectures are often used in embedded processors or constrained edge devices due to their lower power consumption and cost. The integrated run- times, such as Chicory and Wazero, provide 32-bit architecture support via their underlying non-Wasm runtimes, namely, JDK and the Go runtimes respectively.

2.3. Compilation Mode

In Section 2.4 we reviewed different compilation modes. In this section we classify the existing Wasm runtimes within this taxonomy.

(1) Chicory

Compilation Mode: AOT and Interpreted

(2) GraalWASM

Compilation Mode: AOT and JIT

(3) Lunatic

Compilation Mode: JIT

(4) WAMR

Compilation Mode: Interpreted, AOT, and JIT

(5) Wasmedge

Compilation Mode: Interpreted and AOT

(6) Wasmtime

Compilation Mode: AOT and JIT

(7) Wasm3

Compilation Mode: Interpreted

(8) Wasmer

Compilation Mode: JIT and AOT

(9) Spin

Compilation Mode: same as Wasmtime since this is a wrapper runtime based on Wasmtime

(10) Wazero

Compilation Mode: AOT and Interpreted

2.4. Execution Environment

As previously mentioned, the execution environment of a Wasm runtime refers to whether it is designed to operate solely on a single host or across a distributed network of nodes. Most of the leading-edge runtimes fall into the first category, focusing on single-host operations. However, newer, higher-level wrapper run- times built on top of these foundational runtimes are increasingly designed with distributed node environments in mind.

(1) Single Host Wasm Runtimes

GraalWASM, Extism, Wasmex, WAMR, Wasmedge, Wasmtime, Wasm3,

Wasmer, Wazero, Spin

(2) Distributed Host Wasm Runtimes

Lunatic, MPIWasm, wasmCloud, Spin

3. Related Work

There are several survey papers related to WebAssembly that offer a broad and inclusive perspective on Wasm and provide comprehensive insights into its scope and applications. These surveys primarily focus on Wasm runtimes in general and explore their respective use cases. However, as the field, technology, and APIs are evolving very rapidly, there remains a gap in systematic research that offers an exhaustive categorization of the standard Wasm runtimes and APIs currently available. Our paper addresses these gaps, presenting aspects which are not covered in the existing literature.

Early work on the adaptation of Wasm to enhance the security of IoT systems has been explored in survey (Radovici, et al., 2018). The authors proposed that Wasm can be in- tegrated as part of a solution to improve the security of IoT by running verified bytecode in an isolated framework. This survey highlights Wasm potential as a foundational technology in developing robust security frameworks for IoT devices. In another early study (Jangda, et al., 2019) make an effort to comprehend how effectively Wasm binaries work. The research, however, is restricted to web browsers only and thus does not offer enough information about standalone runtimes.

The work of (Spies and Mock, 2021) is the first one that evaluates Wasm runtimes performance in non-web environments. The study conducts performance tests across five Wasm runtimes, including standalone and those integrated within traditional software systems in three non-web environments: desktop, server, and IoT device. The work of (Wang, et al., 2018) systemically characterizes and thoroughly analyzes major edge Wasm runtimes for the first time, highlighting their suitability based on extensive appli- cation scenarios and performance metrics. The work demonstrates that different runtimes fit different application scenarios. However, the scope of this work is limited to edge Wasm runtimes and doesn't provide a broader categorization of these runtimes across other potential environments.

Wang addresses the gap in previous studies by conducting a thorough characterization study of standalone Wasm runtimes. His work in (Wenwen, 2022) provides a detailed analysis of the standard Wasm runtimes. However, only the five most common independent Wasm runtimes are covered in this research. The study in (Kakati and Brorsson, 2023) provides a extensive review on how Wasm is applied outside traditional web contexts, specifically in edge and cloud computing environments. The authors reviewed different Wasm runtimes, compared their features and suitability for specific use cases, and identified research gaps and future directions for Wasm expansion into more diverse computing environments. However, they mainly cover Wasm runtimes in the edge-cloud continuum.

The study in (Ray, 2023) provides an overview of Wasm evolution, reviews relevant development tools and runtimes, illustrates real-world applications, and outlines both current challenges and future prospects for Wasm within IoT domain. How- ever, similar to other works in this domain, this study does not specifically focus on Wasm runtimes and APIs, but instead provides a broad overview of different Wasm runtimes and Wasm ecosystem in general with respect to IoT. A most re- cent survey on research on WebAssembly runtimes was presented in (Zhang, et al., 2024). This survey provides an extensive overview of Wasm runtimes, discussing research cov- ered in 98 articles on the subject. It explores Wasm runtime research from both internal perspectives (design, testing, analysis) and external applications across various domains. The paper also highlights the rapid evolution of Wasm technologies and proposes future research directions. This survey does a good job surveying different Wasm runtimes, however, their focus is to survey the existing research papers on Wasm runtimes and discuss the research focus of each paper, rather than classifying the state-of-the-art runtimes or their APIs.

4. Conclusion

The expansion of WebAssembly (Wasm) beyond its initial role as a browser-only technology into areas such as generative AI, serverless AI, edge computing, cloud computing, and high-performance computing has created a diverse ecosys- tem of Wasm runtimes. This diversity, combined with the relative isolation of developer communities, has resulted in a lack of comprehensive documentation and common classification, making it difficult for researchers to gain a complete understanding of the field. In this paper, we make an initial effort to establish a common taxonomy and categorize many existing Wasm runtimes. Our taxonomy classifies Wasm runtimes into five broad categories: type, architecture, platform, compilation mode, and execution environment. difficult We believe that this paper provides a valuable framework for describing WebAssembly runtimes and select- ing, examining, and comparing runtime systems based on their capabilities. This foundation allows for the classification of additional runtime systems using our definitions and offers a better overview and comparative basis for newly imple- mented features within the diverse and expanding field of WebAssembly runtime ecosystems.

References

Andreas, H., Andreas R., Derek S., Ben., T.L., Michael, H., et al. (2017). Bringing the web up to speed with WebAssembly. Pro- ceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 185-200.

Bytcode Alliance (a). (2024). WebAssembly GC proposal. Available from: https://github.com/WebAssembly/gc.

Bytcode Alliance (b). (2024). Wasmtime - a fast and secure runtime for WebAssem- bly. Available from: https://wasmtime.dev/.

Chicory Runtime. (2024). Available from: https://github.com/ dylibso/chicory.

Extism. (2024). Extism - a lightweight framework for building with WebAssembly. Available from: https://github.com/tessi/wasmex.

Fermyon Technologies (a). (2024). Spin - a developer tool for building WebAssem- bly microservices. Available from: https://github.com/fermyon/spin.

Fermyon Technologies (b). (2024). Spin, a server-side webassemnly runtime. Avail- able from: https://github.com/fermyon/spin/discussions/ 2534discussioncomment- 9587850.

Jangda, A., Powers, B., Berger, E.D. & Guha, A. (2019). Not so fast: Analyzing the performance of {WebAssembly} vs. native code. USENIX Annual Technical Conference (USENIX ATC 19), 107–120.

Kakati, S. & Brorsson, M. (2023). WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum. 3rd International Conference on In- telligent Technologies (CONIT), 1-8.

Lunatic Inc. (2023). Lunatic - an Erlang-inspired runtime for WebAssembly. Available from: https://github.com/lunatic-solutions/lunatic/.

McConnell, J. (2017). WebAssembly support now shipping in all major browsers. Available from: https://blog.mozilla.org/blog/ 2017/11/13/ webassembly- in-browsers.

Mohak, C., Nils, K., Jophin, J., Anshul, J., Michael, G. & Shajulin, B. (2023). Exploring the Use of WebAssembly in HPC. PPoPP ’23, Association for Computing Machinery, 92–106.

Oracle. (2024). GraalWasm - an open source WebAssembly runtime. Avail- able from: https://www.graalvm.org/latest/reference-manual/wasm/.

Radovici, A., Rusu, C. & Serban, R. (2018). A Sur- vey of IoT Security Threats and Solutions. 17th RoEduNet Conference: Net- working in Education and Research (RoEduNet).

Ray, P.P. (2023). An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions. Future Internet.

Spies, B. & Mock, M. (2021). An Evaluation of WebAssembly in Non- Web Environments, XLVII Latin American Computing Conference (CLEI), 1-10.

Tetrate.io. (2024). Wazero: the zero dependency WebAssembly runtime for Go developers. Available from: https://github.com/tetratelabs/wazero.

Wasmer Inc. (2024). Wasmer, The leading Wasm Runtime supporting WASIX, WASI and Emscripten. Available from: https://github.com/wasmerio/wasmer.

WasmCloud LLC. (2024). wasmCloud - a universal WebAssembly application platform. Available from: https://github.com/ wasmCloud/wasmCloud.

Wang, Z., Wang, J., Wang, Z. & Hu, Y. (2018). Characterization and Implication of Edge WebAssembly Runtimes, IEEE 23rd Int Conf on High Performance Computing Communications; 7th Int Conf on Data Science Systems; 19th Int Conf on Smart City; 7th Int.

Wasmex. (2024). Wasmex - a fast and secure WebAssembly and WASI runtime for Elixir. Available from: https://github.com/tessi/wasmex.

WasmEdge. (2024). WasmEdge, A WebAssembly Runtime. Available from: https://wasmedge.org/.

WebAssembly Contributors. (2015). WebAssembly Launch. GitHub Issue.

WebAssembly Working Group. (2022). WebAssembly Core Speci- fication. World Wide Web Consortium (W3C). Available from: https://www.w3.org/TR/wasm-core-2 .

WebAssembly Community Group. (2021). WebAssembly System Interface. Available from: https://github.com/WebAssembly/WASI.

Wenwen, W. (2022). How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. 2022 IEEE International Symposium on Workload Characterization (IISWC), 228-241.

Zhang, X., Liu, M., Wang, H., Ma, Y., Huang, G. & Liu, X. (2024). Research on WebAssembly Runtimes: A Survey. Available from: https://arxiv.org/abs/2404.12621.