Overview of most popular wasm runtime

Wasm runtimes can be categorized into two types based on their execution environments:

1. **Browser-based Runtime**: Executes Wasm binaries within web browsers using JavaScript engines.
2. **Standalone Runtime**: Executes Wasm binaries directly within operating systems using standalone Wasm runtimes.

Wasm Inside the Web

1. **Compilation**: High-level languages such as C, C++, and Rust are compiled into Wasm binaries using Wasm frontend compilers.
2. **JavaScript Glue Code**: Along with the Wasm binaries, JavaScript (JS) glue code is generated.
3. **Execution in Web Browsers**: The JS glue code and Wasm binaries are executed within JavaScript engines in web browsers.

Wasm Outside the Web

1. **Compilation**: Similar to the web environment, high-level languages are compiled into Wasm binaries using Wasm frontend compilers.
2. **Standalone Wasm Runtimes**: The Wasm binaries are executed in standalone Wasm runtimes, which are directly deployed in operating systems (OSes)

Work-flow.

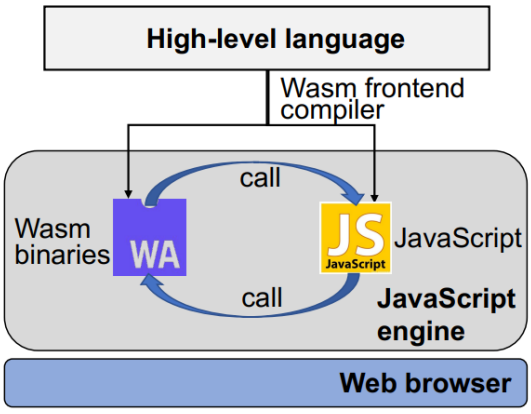


Fig. 3. Wasm workflow inside the Web.

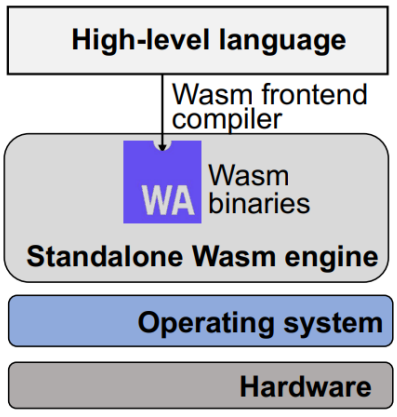
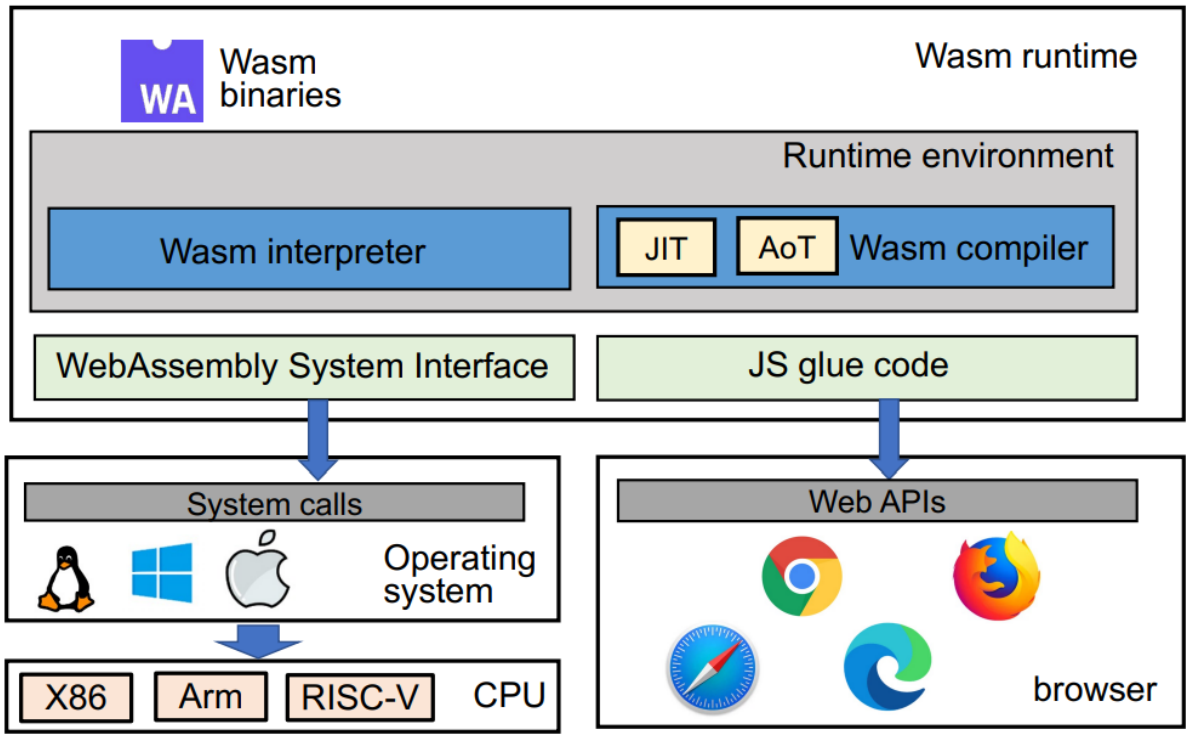


Fig. 4. Wasm workflow outside the Web.

Fronted compiler and source languages :

Wasm frontend compiler	Source language	Commits	Stars
Emscripten	C/C++	26,521	24.8k
Rustc	Rust	243,427	88.9k
Ppci-Mirror	Python	1,118	0
TinyGo	Go	3,614	13.9k
Bytecoder	Java	1,422	836
AssemblyScript	TypeScript	1,564	16.2k

The Architecture of wasm runtimes



The most popular Wasm runtimes

Wasm runtime	standalone	interpreter	JIT	AoT
V8(Chrome/Edge)	○	○	●	○
SpiderMonkey(Firefox)	○	○	●	●
WebKit-JavaScriptCore(Safari)	○	●	○	○
wasmtime	●	○	●	●
wasmer	●	○	●	●
WAMR	●	●	●	●
WasmEdge	●	○	○	●
wasm3	●	●	○	○

*● and ○ refers to the positive and negative value individually.

Standalone Runtime:-

Wasmtime: Wasmtime is a standalone runtime for WebAssembly, WASI, and the Component Model by the Bytecode Alliance.

WebAssembly (abbreviated Wasm) is a binary instruction format that is designed to be a portable compilation target for programming languages. Wasm binaries typically have a `.wasm` file extension. In this documentation, we'll also use the textual representation of the binary files, which have a `.wat` file extension.

Key Features of Wasmtime

- **Fast Execution**: Utilizes the Cranelift code generator for efficient machine code generation, optimized for quick instantiation and low-overhead calls.
- **Secure**: Focuses on security and correctness, leveraging Rust's safety guarantees, undergoing rigorous fuzzing and reviews, and collaborating on formal verification.
- **Configurable**: Offers flexibility with sensible defaults and fine-grained control over CPU and memory usage, suitable for both small environments and large-scale deployments.
- **WASI Support**: Provides a rich set of APIs for interacting with the host environment through the WebAssembly System Interface (WASI).

Pros and Cons of Wasmtime

Pros

- **High Performance**: Fast execution due to efficient code generation with Crane lift, optimizing both runtime performance and instantiation speed.
- **Strong Security Focus**: Built on Rust's safety guarantees, rigorously tested through fuzzing, and incorporates formal verification, enhancing overall security.
- **Flexibility**: Highly configurable to adapt to various environments, from small devices to large-scale servers, offering control over CPU and memory usage.
- **Standards Compliance**: Fully adheres to WebAssembly standards and actively supports future proposals, ensuring compatibility and robustness.

Cons

- **Complex Configuration**: Advanced configuration options might be overwhelming for users who prefer simpler setups or are new to Wasmtime.
- **Limited Ecosystem**: Compared to more established runtimes, Wasmtime may have a smaller ecosystem and fewer integrations, potentially requiring more effort to integrate with existing tools.
- **Evolving Features**: Continuous updates and feature changes can occasionally lead to instability or require adjustments in usage.
- **Resource Overhead**: While optimized, managing fine-grained configurations might still lead to overhead, particularly in resource-constrained environments.

Support language : Rust ,C/C++,GO,Python

Wasmer: Wasmer is an ecosystem based on WebAssembly that allows running applications anywhere: in the browser, embedded in your favorite programming language, or standalone in the server. The Wasmer Runtime is the engine that allows running WebAssembly modules and Wasmer packages anywhere.Wasmer can be used as a library from any programming language, or as a standalone runtime via the Wasmer CLI.

Features of Wasmer

- **Cross-Platform Compatibility**: Run WebAssembly containers on various platforms and environments, including local machines, cloud services, and edge devices.
- **High Performance**: Utilizes a fast, optimizing compiler for efficient execution of WebAssembly code.
- **Extensible and Modular**: Supports plugins and extensions for adding custom functionality and integrations.

- **Comprehensive Ecosystem**: Includes tools like Wasmer Runtime, Wasmer Registry, and Wasmer Edge for managing and deploying WebAssembly containers across different infrastructure.

Language supported by Wasmer: Rust, C/C++, AssemblyScript, Go, Python, Kotlin, TypeScript, and others.

Pros and cons of wasmer

Pros:

- **Portability Across Environments:** Wasmer allows WebAssembly (WASM) code to run on various platforms beyond the browser, ensuring consistent execution across different environments.
- **Lightweight and Secure:** WASM's sandboxed execution environment enhances security by isolating code execution, making it suitable for running untrusted or third-party code.
- **Performance Efficiency:** WASM offers low startup times and efficient execution, which is beneficial for scenarios requiring fast initialization, such as serverless functions or edge computing.
- **Multilingual Support:** Code written in different languages (e.g., Rust, C, Go) can be compiled to WASM and executed in a Wasmer runtime, allowing for the use of diverse programming languages in a single application.

Cons:

- **Limited Direct System Access**: WASMER cannot directly access system resources or OS functions. To perform tasks like file system operations, it relies on APIs provided by the runtime, which may require additional setup.
- **Additional Complexity for Integration:** Integrating WASMER lacks certain high-level features such as garbage collection and native threading support, which can limit its usability for some applications without additional workarounds.
- **Ecosystem and Tooling:** The WASMER ecosystem and tooling are still developing. This can lead to challenges in finding mature tools and comprehensive documentation

Wamr:

WebAssembly Micro Runtime (WAMR) is a lightweight standalone WebAssembly (WASM) runtime with small footprint, high performance and highly configurable features for applications cross from embedded, IoT, edge to Trusted Execution Environment (TEE), smart contract, cloud native and so on.A Bytecode Alliance project.

Features of WAMR (WebAssembly Micro Runtime)

1. **Platform Adaptability:**
    - WAMR supports multiple platforms and can run on various operating systems and architectures. It is developed in C, which contributes to its wide adaptability.
2. **Compilation Modes:**
    - WAMR supports interpretation mode, just-in-time (JIT) compilation, and ahead-of-time (AOT) compilation. These modes offer flexibility in execution depending on the use case and performance requirements.
3. **Low Resource Overhead:**
    - WAMR has a minimal footprint, capable of running a single WebAssembly instance in as little as 100KB of memory, making it suitable for resource-constrained environments.
4. **High Performance:**
    - With features like deeply optimized pre-compilation and efficient foreign function interface (FFI), WAMR achieves performance levels comparable to native binaries.
5. **Security:** WAMR runs WebAssembly code in a strict sandbox environment, ensuring isolated execution with independent memory spaces. This enhances security by preventing buffer overflow and remote code execution attacks.

Pros of WAMR

1. **Excellent Performance:**
    - WAMR provides significant performance improvements through AOT compilation and optimized execution, often resulting in performance comparable to native code.
2. **Low Memory Footprint:**
    - Its ability to run in minimal memory makes WAMR ideal for embedded systems and other resource-constrained environments.
3. **Flexibility:**
    - Support for multiple compilation and execution modes (interpretation, JIT, AOT) allows developers to choose the best approach for their specific needs.
4. **Strong Community and Industry Support:**
    - Backed by the Bytecode Alliance and major industry players, WAMR benefits from robust development and extensive real-world application.

Cons of WAMR

1. **Complex Setup:**
    - Setting up and integrating WAMR can be complex, especially when transitioning from other WebAssembly runtimes like V8. It may require significant adjustments and custom configurations.
2. **Limited Ecosystem:**

- While growing, the ecosystem and tooling around WAMR are not as mature as some other runtimes, which can lead to challenges in finding comprehensive support and documentation.

3. **Specialized Knowledge Required:**
   - Effective use of WAMR, especially leveraging its advanced features like AOT compilation and FFI optimizations, may require specialized knowledge and expertise.
4. **Potential Compatibility Issues:**
   - Transitioning from other WebAssembly runtimes like V8 may present compatibility issues, particularly if relying on specific features or behaviors unique to the previous runtime

Wasmedge

WasmEdge is a lightweight, high-performance, and extensible WebAssembly runtime. It is the fastest runtime today. WasmEdge is an official sandbox project hosted by the CNCF. Its use cases include modern web application architectures (Isomorphic & Jamstack applications), microservices on the edge cloud, serverless SaaS APIs, embedded functions, smart contracts, and smart device.

Features

1. **Multilingual Support and Interoperability**: WasmEdge can run WebAssembly bytecode programs compiled from various languages like C/C++, Rust, Swift, AssemblyScript, and Kotlin. It supports running JavaScript, including ES6, CommonJS, and NPM modules, in a secure and containerized sandbox, enabling seamless integration and interoperability between these languages.
2. **Extensions for Cloud-native and Edge Computing**: WasmEdge includes tailored extensions such as network sockets, PostgreSQL and MySQL database drivers, and AI capabilities. These extensions enhance its functionality for cloud-native and edge computing applications, making it versatile and powerful for distributed environments.
3. **Support for Standard and Proposed WebAssembly Features**: WasmEdge fully supports standard WebAssembly features and many proposed extensions, ensuring compatibility and future-proofing. This allows developers to leverage the latest WebAssembly advancements and capabilities in their applications.
4. **Performance and Portability**: Designed to be fast, lightweight, and portable, WasmEdge provides an efficient runtime for executing WebAssembly programs. Its performance optimizations make it suitable for both server-side rendering (SSR) functions on edge servers and client-side execution, delivering quick and reliable performance across different environments.

Pros of WasmEdge

1. **Multilingual and Interoperable**:
    - Supports a wide range of programming languages (C/C++, Rust, Swift, AssemblyScript, Kotlin).
    - Can run JavaScript, including ES6, CommonJS, and NPM modules, allowing seamless integration and interoperability.
2. **Tailored Extensions for Cloud-native and Edge Computing**:
    - Provides specialized extensions such as network sockets, PostgreSQL, MySQL database drivers, and AI capabilities.
    - Enhances functionality for distributed and edge environments, making it suitable for diverse applications.
3. **High Performance and Portability**:
    - Designed to be fast, lightweight, and portable.
    - Offers efficient runtime performance for both server-side rendering and client-side execution, ensuring quick and reliable application deployment.

Cons of WasmEdge

1. **Complexity for Beginners**:
    - The wide range of features and extensions might be overwhelming for beginners.
    - Requires a good understanding of WebAssembly and its ecosystem to fully utilize its capabilities.
2. **Limited Ecosystem Compared to More Established Runtimes**:
    - Although growing, the WasmEdge ecosystem might not be as mature or extensive as other established runtimes.
    - Fewer community resources, libraries, and third-party integrations compared to more popular platforms.
3. **Dependency on WebAssembly Ecosystem**:
    - The effectiveness and utility of WasmEdge are closely tied to the evolution and adoption of WebAssembly standards and proposals.
    - Potential limitations or delays in the WebAssembly ecosystem's development could impact WasmEdge's functionality and adoption

Wasm3

Wasm3 is a high-performance, universal WebAssembly (WASM) interpreter. It is designed to be the fastest interpreter available, providing a portable and efficient runtime for executing WASM code across various platforms and architectures. It excels in scenarios where low memory usage, quick startup, and ease of integration are prioritized over raw execution speed.

Features of Wasm3

1. **High Performance**: Known for its speed, Wasm3 is considered one of the fastest WebAssembly interpreters.
2. **Universal Compatibility**: Supports a wide range of platforms including Linux, Windows, OS X, FreeBSD, Android, iOS, and embedded systems like Arduino, ESP8266, ESP32, and Raspberry Pi.
3. **Low System Requirements**: Can run on systems with as little as ~64Kb of code space and ~10Kb of RAM.
4. **Wide Architecture Support**: Compatible with multiple architectures such as x86, x86_64, ARM, RISC-V, PowerPC, MIPS, Xtensa, and ARC32.
5. **Extensive Language Support**: Can be used as a library in various programming languages including Python, Rust, C/C++, GoLang, Zig, Perl, Swift, .Net, Nim, and more.
6. **Compliance and Testing**: Passes the WebAssembly specification testsuite and supports many WASI applications.

Pros of Wasm3

1. **Speed and Efficiency**: Optimized for high performance and low memory usage.
2. **Portability**: Runs on numerous platforms and architectures, making it highly versatile.
3. **Easy Integration**: Simple to compile and integrate into existing projects with minimal development overhead.
4. **Quick Startup**: Minimal startup latency, suitable for applications where rapid initialization is crucial.
5. **Security and Predictability**: Provides a sandboxed environment with well-defined execution, enhancing security and predictability.
6. **Flexibility**: Can be used in various scenarios including edge computing, scripting, plugin systems, IoT rules, and smart contracts.

Cons of Wasm3

1. **Interpretation Overhead**: As an interpreter, it may be slower than Just-In-Time (JIT) compiled runtimes in some cases.
2. **Limited Advanced Features**: Lacks support for some advanced WebAssembly features like multiple memories, reference types, tail call optimization, fixed-width SIMD, and exception handling.

3. **Performance Variability**: Performance may vary depending on the specific use case and complexity of the WebAssembly modules.
4. **Smaller Ecosystem**: Compared to more established runtimes, Wasm3 has a smaller user and developer community, which may affect the availability of support and resources

WebAssembly Runtimes Comparison

Feature	Wasmer	WAMR (WebAssembly Micro Runtime)	Wasmtime	WasmEdge	Wasm3
Primary Use Case	General-purpose WebAssembly runtime	IoT and embedded systems	General-purpose WebAssembly runtime	Cloud-native applications and edge computing	High-performance WebAssembly interpreter
Platform Compatibility	Cross-platform (Windows, Linux, macOS, IoT)	Cross-platform with a focus on IoT and embedded systems	Cross-platform (Windows, Linux, macOS)	Cross-platform (Windows, Linux, macOS)	Cross-platform (Windows, Linux, macOS, embedded systems)
Performance	Near-native speed	Optimized for low-memory, resource-constrained environments	Near-native speed	Near-native speed	High-performance with emphasis on speed
Security	Strong sandboxing and isolation	Strong sandboxing with a focus on small footprint	Strong sandboxing and isolation	Strong sandboxing and isolation	Basic sandboxing with emphasis on performance
Embeddability	Embeddable in various programming languages	Highly embeddable in resource-constrained devices	Embeddable in various programming languages	Embeddable in various programming languages	Embeddable, with a focus on minimal dependencies
Containerization	Supports lightweight Wasm containers	Not specifically designed for containerization	Not specifically designed for containerization	Supports lightweight Wasm containers	Not specifically designed for containerization
Edge Computing	Suitable for edge computing	Suitable for edge and constrained devices	Suitable for edge computing	Optimized for edge computing	Suitable for edge computing
Developer Community	Large and active community	Smaller but active community, focused on IoT	Large and active community	Growing community	Smaller, but focused on performance

Feature	Wasmer	WAMR (WebAssembly Micro Runtime)	Wasmtime	WasmEdge	Wasm3
Compilation Framework	Compatible with Cranelift, LLVM	Supports AoT (Ahead-of-Time) compilation, interpreter	Compatible with Cranelift, LLVM	Compatible with Cranelift, LLVM	Interpreter-based, no JIT compilation
Memory Usage	Moderate to high	Low, optimized for minimal memory usage	Moderate to high	Moderate to high	Low memory usage
IoT Suitability	Good	Excellent, specifically designed for IoT	Good	Good	Excellent for IoT devices with limited resources
Smart Contract Execution	Supported, secure environment	Supported, secure environment	Supported, secure environment	Supported, secure environment	Not specifically designed for smart contracts
Language Support	Multiple languages (Rust, C, Python, etc.)	Multiple languages (C, C++, Rust, etc.)	Multiple languages (Rust, C, Python, etc.)	Multiple languages (Rust, C, Python, etc.)	Limited to WebAssembly text-based languages
Standard Compliance	Compliant with WebAssembly standards	Compliant with WebAssembly standards	Compliant with WebAssembly standards	Compliant with WebAssembly standards	Compliant with WebAssembly standards

Use Case Scenarios

1. **General-Purpose WebAssembly Runtime:**
   - **Best Choice:** Wasmer or Wasmtime
   - **Why:** Both Wasmer and Wasmtime offer near-native speed, strong sandboxing, and are cross-platform compatible, making them suitable for a wide range of applications. They also have large and active developer communities.
2. **IoT and Embedded Systems:**
   - **Best Choice:** WAMR (WebAssembly Micro Runtime)
   - **Why:** WAMR is specifically optimized for low-memory and resource-constrained environments, making it ideal for IoT and embedded systems. It has a strong focus on small footprint and high embeddability.
3. **Cloud-Native Applications and Edge Computing:**

- **Best Choice:** WasmEdge
- **Why:** WasmEdge is optimized for cloud-native environments and edge computing, providing near-native speed and strong sandboxing. It supports lightweight Wasm containers, which are beneficial for cloud and edge applications.

4. **High-Performance WebAssembly Interpreter:**
    - **Best Choice:** Wasm3
    - **Why:** Wasm3 focuses on high performance with minimal dependencies, making it an excellent choice for environments where performance is critical but resources are limited. It is suitable for both edge computing and IoT devices.