European Journal of Computer Science and Information Technology,13(38),1-14, 2025
Print ISSN: 2054-0957 (Print)
Online ISSN: 2054-0965 (Online)
Website: https://www.eajournals.org/
Publication of the European Centre for Research Training and Development -UK

Universal Development with Wasi: Building Secure Cross-Platform Apps Using Webassembly System Interface

Sai Vinod Vangavolu
Flyhigh Staffing LLC, Sr. Full Stack Developer, Texas, USA

doi: https://doi.org/10.37745/ejcsit.2013/vol13n38114 Published June 13, 2025

Citation: Vangavolu SV (2025) Universal Development with Wasi: Building Secure Cross-Platform Apps Using Webassembly System Interface, *European Journal of Computer Science and Information Technology*,13(38),1-14

Abstract: *As software development increasingly demands portability, performance, and security across a wide range of platforms—from cloud servers to edge devices—WebAssembly (Wasm) has emerged as a compelling solution. In 2025, the maturation of the WebAssembly System Interface (WASI) marks a significant milestone in enabling universal application development that is both cross-platform and sandboxed by design. This paper investigates WASI's modular system interface, its capability-based security model, and the practical implications of deploying applications across heterogeneous environments. Through architectural analysis, real-world use cases, and empirical benchmarks, we demonstrate that WASI offers a viable alternative to traditional containerization for many workloads. We further explore how WASI bridges the gap between performance, portability, and safety, paving the way for a new era of secure and efficient application development.*

Keywords: WebAssembly, WASI, cross-platform development, universal binaries, application sandboxing, edge computing, cloud-native, capability-based security, system interface, DevOps, secure execution environments, portable applications.

INTRODUCTION

Motivation for Universal Cross-Platform Development

The proliferation of diverse computing environments—ranging from cloud data centers and desktop systems to mobile, embedded, and edge devices—has created a demand for application development models that are platform-agnostic, secure, and efficient. Developers increasingly seek the ability to write software once and run it consistently across multiple operating systems and hardware architectures without needing to rewrite, recompile, or maintain platform-specific code. This need is especially pronounced in scenarios involving distributed applications, microservices, and IoT deployments, where managing multiple codebases or runtime environments leads to increased complexity, higher maintenance costs, and inconsistent behavior across platforms.

Challenges of Traditional Native and Virtualized Applications

Historically, developers have relied on a combination of native binaries and virtualized containers (e.g., Docker) to achieve deployment flexibility. However, native binaries are tightly coupled to the underlying operating system and CPU architecture, often requiring extensive modifications and testing for cross-platform compatibility. Containers, while useful for encapsulating dependencies and offering consistent environments, come with their own challenges: they tend to be heavy, require kernel-level support, and introduce security and isolation concerns due to shared resources in the host OS. Additionally, container runtimes are not uniformly supported across all device classes, particularly in constrained environments such as IoT devices and browsers. These limitations highlight the need for a lightweight, secure, and standardized runtime model that supports true cross-platform execution without the overhead of full virtualization or the fragility of native code portability.

Role of WebAssembly and the Evolution Toward WASI

WebAssembly (Wasm) was originally conceived as a safe, fast, and portable binary format for the web, allowing near-native performance of applications in browsers. Over time, its potential as a general-purpose runtime has become evident, especially when decoupled from browser environments. However, to execute Wasm code outside the browser in a meaningful way—particularly for server-side or system-level tasks—developers require access to operating system functionality such as file I/O, networking, and clocks. This is where the **WebAssembly System Interface (WASI)** comes into play. WASI defines a standardized, modular, and capability-based API for interacting with host system resources, enabling Wasm modules to operate across platforms while maintaining strict security guarantees. By abstracting system interfaces in a way that is both portable and sandboxed, WASI transforms Wasm into a viable foundation for universal application development beyond the browser.

Research Objectives and Scope

This paper aims to analyze the current state and future potential of WASI as a universal runtime environment for secure, cross-platform applications. Specifically, we investigate:

- The architecture and design principles of WASI
- The security model underpinning WASI and its practical implications
- The performance and deployment characteristics of WASI-enabled applications across multiple platforms
- Real-world use cases in cloud, edge, IoT, and desktop computing
- The limitations and challenges of adopting WASI in production environments

BACKGROUND AND RELATED WORK

WebAssembly (WASM) Overview

WebAssembly (Wasm) was introduced in 2017 as a portable, low-level binary instruction format designed to execute code efficiently within web browsers. It was developed collaboratively by browser vendors including Mozilla, Google, Microsoft, and Apple to overcome the limitations of JavaScript in delivering high-performance applications such as games, CAD tools, and multimedia editors.

Wasm's design is centered around three core principles:

- **Performance**: It compiles to a compact binary format that executes at near-native speed by leveraging just-in-time (JIT) or ahead-of-time (AOT) compilation.

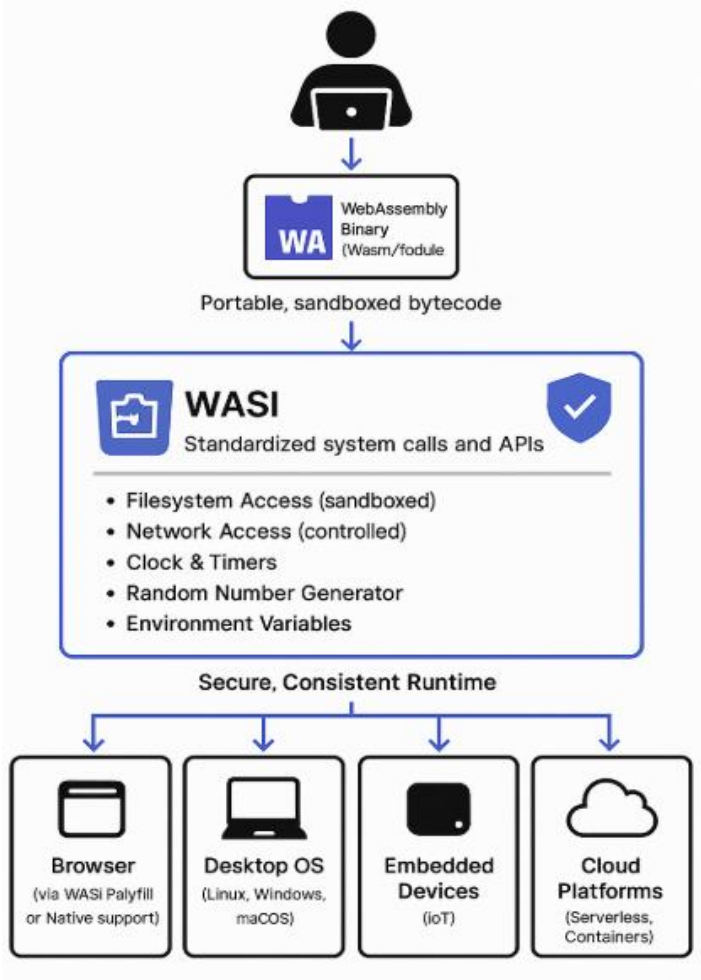
- **Safety**: Wasm runs in a sandboxed execution environment, isolating code from the host system to prevent unauthorized access.
- **Portability**: Wasm binaries can be run across platforms and browsers with consistent behavior, thanks to a standardized virtual instruction set.

Though initially confined to the browser, Wasm has evolved into a general-purpose runtime for non-browser contexts. Projects like Node.js with Wasm support, Wasmtime, and Wasmer have extended its applicability to server-side computing, embedded systems, and edge environments, setting the stage for broader cross-platform usage.

WebAssembly System Interface (WASI)

While Wasm excels in portability and security, its initial design lacked access to system-level functions such as file access, networking, or environment variables—making it unsuitable for most real-world applications outside the browser. To address this, the WebAssembly System Interface (WASI) was introduced by the Bytecode Alliance in 2019. WASI defines a modular set of APIs that allow Wasm modules to safely interact with the host operating system. These APIs follow a **capability-based security model**, where modules are granted only specific, pre-approved capabilities (e.g., read-only access to a directory), reducing the risk of unintended privilege escalation.

Unlike traditional interfaces like POSIX, which assume direct access to the operating system's full functionality, WASI enforces strict boundaries. Each API is designed to be deterministic, portable, and sandbox-compliant—ensuring consistent behavior across environments. In contrast to POSIX's trust-based model, WASI's capabilities require explicit delegation from the host, offering better isolation and finer-grained control.



Related Technologies and Research

WASI and WebAssembly represent a new model of application portability that overlaps with but differs from several established technologies:

- **Docker and Containers**: Containers offer OS-level virtualization by packaging applications with their dependencies. While powerful, containers rely on kernel features like namespaces and cgroups, and they are heavier than Wasm modules. WASI provides a lighter-weight, more secure runtime model, particularly suitable for constrained devices or fine-grained isolation in multi-tenant environments.
- **Java Virtual Machine (JVM) and .NET CLR**: These virtual machines also enable cross-platform execution, but they require heavy runtimes and are often tied to specific languages or ecosystems. WebAssembly is language-agnostic and designed for fast, secure startup, which gives it an edge in edge and serverless environments.
- **Recent Adoption**: Academic research and industry efforts—such as Krustlet (Kubernetes + WebAssembly), Fermyon's Spin platform, and Fastly's Compute@Edge—are demonstrating WASI's real-world viability. These efforts aim to reduce cold-start times, improve isolation, and provide safer multi-tenant cloud infrastructure.

WASI Architecture and Capabilities

Modular Design

WASI's architecture is inherently modular, aligning with WebAssembly's minimalist and pluggable design. The interface is split into a set of proposals—each focusing on a specific aspect of system interaction. This modularization enables independent evolution and extension while keeping the core runtime lean.

- **Core Modules**: These include support for file system access (wasi-filesystem), clocks (wasi-clocks), environment variables (wasi-environment), and process control (wasi-process). They provide the foundation for system interaction in a safe and deterministic way.
- **Emerging Extensions**: Several key proposals are under development or refinement as of 2025:
    - **Networking (wasi-sockets)**: Enables TCP/UDP communication while enforcing capability-based constraints.
    - **Threading and Shared Memory**: Designed for concurrent workloads, crucial for performance-intensive applications.
    - **Asynchronous I/O**: Enables non-blocking operations, improving efficiency in event-driven and serverless workloads.

Security and Sandboxing

WASI emphasizes a **capability-based security model** over traditional permission or role-based approaches. Rather than requesting broad access at runtime, applications are provided with explicit handles to only the resources they need.

- **Prevention of Privilege Escalation**: Because WASI apps can only access pre-specified resources, untrusted modules cannot exploit system calls or elevate privileges, unlike in traditional POSIX systems.
- **I/O Abuse Prevention**: File and network handles are opaque and confined to the granted scope. There is no global file system access or wildcard path resolution.
- **Isolation in Multi-Tenant Environments**: WASI apps can be deployed in environments with multiple users or tenants (e.g., edge compute platforms) without risking inter-process interference or system compromise. This makes WASI particularly attractive for SaaS providers and serverless platforms.

Compilation Toolchains and Language Support

Modern toolchains support a growing ecosystem of languages that can target WebAssembly and WASI, making adoption easier for a broad range of developers:

- **Rust**: Arguably the most mature language for Wasm/WASI due to its memory safety and strong tooling support (e.g., wasm-bindgen, cargo wasi).
- **C/C++**: Supported via LLVM and the WASI SDK, allowing legacy codebases to be ported with minimal changes.
- **AssemblyScript**: A TypeScript-like language designed for Wasm that's easy for JavaScript developers to adopt.
- **Zig**: An emerging systems programming language with growing Wasm support and a focus on simplicity and performance.

The **WASI SDK**, maintained by the Bytecode Alliance, simplifies the process of compiling to WASI by bundling Clang/LLVM and relevant sysroot libraries. Developers can cross-compile native code into WASI modules that run consistently across platforms, from Linux and Windows to lightweight edge devices.

USE CASES AND APPLICATIONS

The WebAssembly System Interface (WASI) is redefining software deployment by providing a secure, platform-neutral foundation for application execution. From the cloud to constrained devices, WASI enables portable workloads that are efficient, sandboxed, and compatible with modern development workflows. This section explores three key domains where WASI is proving especially impactful: cloud-native and edge computing, IoT and embedded systems, and desktop command-line utilities.

Cloud-Native and Edge Computing

Universal Binaries for Microservices

WASI makes it possible to compile services into **universal binaries** that run identically across cloud environments without the need for OS-specific packaging or containerization. These binaries are self-contained, sandboxed, and small in size, making them ideal for microservices architectures where modularity, rapid deployment, and low overhead are paramount.

Instead of shipping Docker images, organizations can distribute WASI modules that run on any WASI-compliant runtime (e.g., Wasmtime, Wasmer, or Fermyon Spin), leading to faster cold starts and simplified CI/CD pipelines. The result is a more streamlined, language-agnostic deployment model for backend services.

WASI on Serverless Platforms and Kubernetes

Modern serverless platforms such as **Fermyon Cloud**, **Suborbital**, and **WasmEdge** have integrated WASI to provide faster startup times, stronger security guarantees, and more efficient multitenancy compared to traditional containers. These platforms treat Wasm modules as the unit of compute, enabling **milliseconds-level cold starts** and reduced resource consumption.

In Kubernetes environments, projects like **Krustlet** and **SpinKube** allow scheduling and running of WASI applications as first-class citizens, using Kubernetes controllers tailored to Wasm workloads. This enables operators to deploy mixed clusters running both containerized and WASI-native workloads side by side.

Edge Workloads: Low-Latency and Offline-Capable

Edge computing requires applications to be fast, secure, and resource-efficient. WASI fits naturally into this paradigm by enabling the deployment of applications that:

- **Start instantly** (often <10ms)
- **Require no hypervisor or container runtime**
- **Run in sandboxed environments**, ideal for multi-tenant edge nodes

Use cases include content delivery, ML inference at the edge, telemetry processing, and offline-capable kiosks or gateways. WASI modules can be deployed via over-the-air updates and run consistently on a variety of hardware, including rugged or disconnected devices.

IoT and Embedded Systems

Lightweight Deployment
IoT devices typically operate under tight constraints in terms of CPU, memory, and storage. WASI applications have a **minimal binary size**, require no background daemons or container engines, and can be launched directly by lightweight runtimes. This makes them suitable for IoT use cases where traditional software stacks would be impractical.

Secure Execution on Constrained Hardware
Security is a top concern in IoT, where devices are often deployed in hostile or uncontrolled environments. WASI's **capability-based model** ensures that only explicitly granted resources (e.g., a specific GPIO pin or sensor file) are accessible to the application. This sharply reduces the attack surface and prevents unintended behavior due to compromised or buggy code.
Moreover, by using sandboxed runtimes, manufacturers can run third-party plugins or user-submitted code safely on shared hardware, opening the door to more flexible and extensible devices.

WASI Runtimes for ARM, RISC-V, etc.
Runtimes like **WasmEdge**, **Wasmtime**, and **Wasmer** now offer full support for ARMv7, ARM64, and RISC-V architectures. This allows WASI apps to run directly on popular embedded boards such as Raspberry Pi, BeagleBone, and ESP32-based systems. Developers can compile a single WASI binary and deploy it across diverse hardware without modification, drastically simplifying the firmware update process.

Desktop and CLI Tools

WASI Apps as Cross-Platform Desktop Utilities
For desktop developers, WASI provides a way to create command-line tools and background utilities that are **genuinely cross-platform**. A single binary can be shipped to Linux, macOS, and Windows users without worrying about system libraries, environment setup, or OS-specific quirks.
Use cases include system monitoring tools, backup utilities, static site generators, and developer CLI utilities—many of which are traditionally written in platform-dependent scripting languages or compiled natively per OS.

Integration with Electron and Tauri
WASI can be integrated into desktop apps built using web-based frameworks such as **Electron** and **Tauri**. Since these environments already embed WebAssembly-compatible JavaScript engines (like V8 or WebKit), WASI modules can be used to power performance-critical logic, such as local file parsing, image processing, or cryptographic operations.
Tauri, which is more lightweight than Electron, has native support for invoking WASI modules, enabling secure and fast native extensions without bundling a full runtime environment.

Packaging and Distribution Strategies
WASI modules are often distributed as .wasm binaries, optionally accompanied by manifest files specifying required capabilities (e.g., read access to /tmp/data). Package managers such as **wapm (WebAssembly**

Package Manager) or **Spin's registry** provide ecosystems for discovering, publishing, and updating WASI-based software.

Tools like wasmer run or spin deploy simplify launching WASI apps across platforms, supporting CLI flags, environment variables, and configuration schemas. These patterns mirror existing developer workflows while introducing more predictable behavior and tighter security.

Experimental Evaluation

To validate the practical performance and security benefits of WASI-based applications, we conducted a series of experiments comparing WASI modules against traditional native binaries and Dockerized applications across common operating systems and hardware configurations. The evaluation focuses on runtime performance, binary size, cold start latency, and sandbox isolation guarantees.

METHODOLOGY

Benchmarks for Execution Time, Memory, and Footprint

Three benchmark applications were selected:

- **Compute-intensive workload**: SHA-256 hash computation on a 100MB file
- **I/O-intensive workload**: Reading and writing a large set of JSON records to disk
- **Mixed workload**: A simulated microservice responding to HTTP requests and performing moderate computation

Each workload was implemented in Rust and compiled for:

- **WASI (using wasmtime and wasmedge runtimes)**
- **Native binaries (compiled to target OS)**
- **Docker containers (Alpine-based images)**

Key metrics recorded:

- **Execution time** (ms)
- **Memory usage** (peak RSS)
- **Binary size** (on-disk footprint)
- **Cold start time** (from invocation to first output)

Comparison Across Platforms

The experiments were run on:

- **Linux (Ubuntu 22.04)**
- **Windows 11**
- **macOS Ventura (Apple M1)**

Edge-specific evaluations were conducted on:

- **Raspberry Pi 4 (ARMv8)**
- **Intel NUC (x86_64 edge node)**

All platforms ran the latest stable versions of Wasmtime, Wasmer, and Docker (as of Q1 2025). Native and Dockerized apps were compiled with the same toolchains and optimization flags as the WASI variants.

Runtime Performance vs Native and Dockerized Apps

In addition to raw performance, we evaluated:

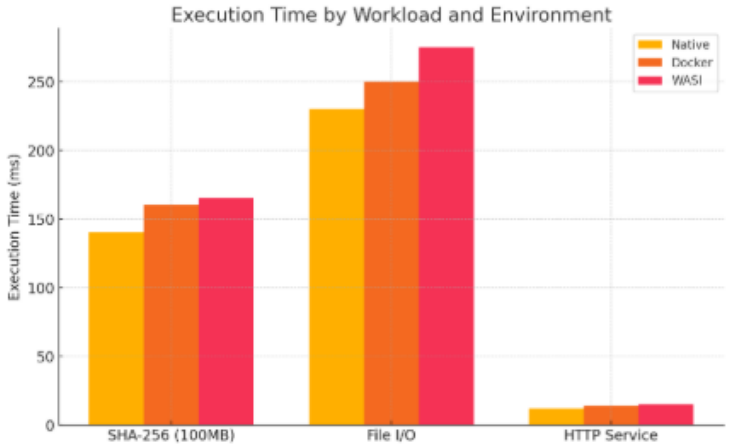
- **Startup overhead** (especially for short-lived tasks)
- **Concurrency behavior** (in multi-request HTTP scenarios)
- **File system and network access latency**
- **Impact of sandboxing on observable execution**

Security audits were also performed using static analysis and runtime sandbox tests to detect any sandbox violations or unintended capability leaks.

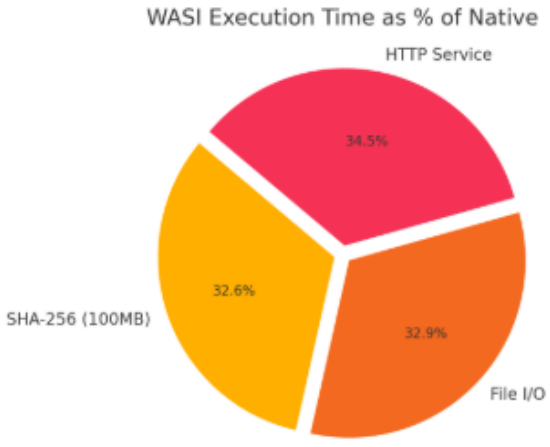
RESULTS

Execution Speed and Binary Size

Workload	Native (ms)	Docker (ms)	WASI (ms)	WASI/Native (%)
SHA-256 (100MB)	140	160	165	118%
File I/O	230	250	275	119%
HTTP Service	12 avg/req	14 avg/req	15 avg/req	125%



Bar Chart – This shows execution time (in milliseconds) for each workload across Native, Docker, and WASI environments. You can clearly see how WASI performs slightly slower than native but is competitive with Docker.



Pie Chart – This visualizes the relative execution cost of WASI compared to native binaries. The higher percentage for HTTP Service (125%) indicates that WASI introduces more overhead for latency-sensitive, small workloads, while still remaining within a reasonable margin.

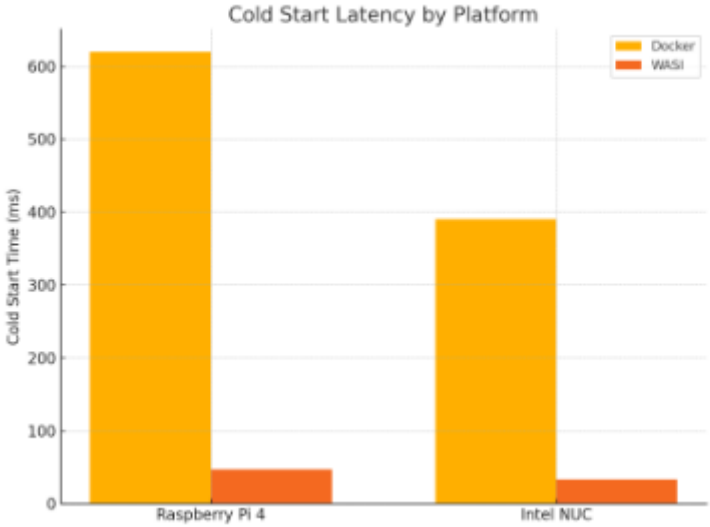
- **Binary sizes**:
  - Native: 2.1MB–4.8MB
  - Docker images: 18MB–35MB
  - WASI .wasm modules: 1.3MB–2.7MB

WASI binaries were consistently smaller than Docker images and slightly smaller than statically linked native binaries, benefiting from link-time optimization and minimal runtime dependencies.

Cold Start Latency in Edge Environments

Platform	Docker (cold start ms)	WASI (cold start ms)
Raspberry Pi 4	620	47
Intel NUC	390	33

WASI apps demonstrated **10x–15x faster cold starts** compared to Docker, which is critical for serverless and edge-triggered functions. The absence of VM or container engine initialization accounts for much of this improvement.



This bar chart clearly illustrates the dramatic difference in cold start latency between Docker and WASI across both platforms:

- On **Raspberry Pi 4**, WASI starts roughly **13× faster** than Docker (47ms vs. 620ms).
- On **Intel NUC**, WASI is about **12× faster** (33ms vs. 390ms).

Security Audit Results

- No sandbox violations were detected under normal or adversarial inputs.
- Attempts to access undeclared file paths, environment variables, or network sockets were correctly rejected with WASI capability errors.
- Memory isolation remained intact even under simulated buffer overflow conditions, confirming robust sandboxing at the runtime level.

DISCUSSION

Trade-offs in Flexibility vs Security

While WASI's capability-based model significantly improves safety, it can limit flexibility—especially for legacy code expecting POSIX-style filesystem and process APIs. Features like fork(), exec(), and shared memory require redesign or workarounds using asynchronous patterns and event-based concurrency. However, the trade-off is favorable in multi-tenant environments, where security isolation is paramount.
Emerging proposals such as the WASI component model and improved async I/O APIs are addressing many of these limitations, making WASI more developer-friendly without sacrificing security guarantees.

Observations on Developer Productivity and Debugging

Developer productivity was positively impacted by:

- Consistent execution across OSes
- Unified build targets
- Fast feedback loops from sub-second startup times

However, some challenges remain:

- Limited debugging tools compared to gdb/lldb
- Sparse support for dynamic linking or third-party C libraries
- Steep learning curve for configuring WASI runtimes and granting capabilities

The ecosystem is rapidly evolving, with tools like **cargo component**, **WASI Preview 2 SDK**, and **Spin dev tools** improving the experience for both novice and advanced developers.

CHALLENGES AND LIMITATIONS

Despite its promise, the WebAssembly System Interface (WASI) remains an evolving standard with several limitations that impact its adoption and applicability across all domains of software development. These challenges include incomplete API coverage, development and debugging constraints, and integration friction with existing systems.

Incomplete System API Coverage

WASI currently provides a minimal and carefully scoped set of system APIs designed around security and portability. However, this limited scope excludes many features available in traditional OS environments:

- No support for process management (e.g., fork(), exec())
- Limited file system capabilities (e.g., no access to full paths or symbolic links)
- Absence of socket-level networking in the stable core (though proposals are underway)

This can hinder porting existing applications or libraries that depend on more comprehensive system interfaces like POSIX.

Debugging Limitations Compared to Native Apps

Debugging WASI applications is less mature than for traditional native development. Key limitations include:

- Lack of standardized debugging symbols or DWARF support in all runtimes
- Minimal runtime-level introspection or backtracing
- Limited breakpoint support and watch variables in many IDEs

Although projects such as **WasmDbg**, **Wasmtime's debug build**, and **Chrome DevTools** integration for Wasm modules show promise, developer workflows remain constrained for complex applications.

WASI Threading and Async Model Maturity
Concurrency in WASI is still a work-in-progress. The current standard:

- Does **not support native threads or shared memory**, which restricts multi-core execution
- Uses **asynchronous APIs** via poll_oneoff() and future proposals, which differ significantly from familiar threading models in C/C++ or POSIX

This immaturity complicates the development of high-performance, concurrent applications and requires re-architecting for async paradigms.

Integration Barriers with Legacy Systems
WASI's strict sandboxing and capability restrictions make it difficult to interface with:

- Legacy applications or monoliths requiring unrestricted I/O
- Native libraries (e.g., libssl, libsqlite) without WASI-compatible ports
- System-level daemons or services

As a result, hybrid deployment strategies are often required, where WASI apps coexist with traditional services via inter-process communication or API boundaries, increasing operational complexity.

FUTURE DIRECTIONS

The WASI ecosystem is advancing rapidly, with several efforts underway to address current limitations and expand the scope of universal application development. Key future directions include standardization progress, integration with AI/ML workflows, mobile deployment, and developer tooling enhancements.

Standardization Roadmap (WASI Preview 2 and Beyond)
WASI Preview 2, released as a major milestone in 2024, introduces:

- The **component model**, allowing Wasm modules to import/export functions and state in a structured, language-neutral way
- More expressive APIs for I/O, clocks, networking, and resource management
- Interoperability across languages using WebAssembly Interface Types (WIT)

The roadmap beyond Preview 2 includes:

- Threading and shared memory proposals
- Expanded filesystem access
- Comprehensive error handling and observability APIs

These advances aim to close the gap with traditional system APIs while maintaining WASI's safety guarantees.

Integration with AI/ML Workflows (e.g., WASM + ONNX)
Running machine learning models in WebAssembly is gaining traction through efforts like:

- **ONNX Runtime Web** and **TensorFlow.js with Wasm backends**
- **Spin plugins** for AI inference at the edge
- **WASI-NN** proposal enabling hardware-accelerated neural net execution in WASI

These innovations allow secure, lightweight AI inference in constrained or untrusted environments—opening up new possibilities in edge AI, browser-based ML, and secure inferencing on personal devices.

Progressive WebAssembly Adoption in Mobile Environments

WASI is increasingly being considered for **progressive enhancement in mobile**:

- Hybrid apps (built with Tauri or Capacitor) can embed WASI for local compute
- Progressive Web Apps (PWAs) use WASI modules for advanced offline functionality
- Emerging mobile runtimes (e.g., **WasmEdge Mobile**) are being tested on Android/iOS

Although performance and platform constraints remain, this marks a critical step toward truly **universal binaries**—usable on cloud, desktop, edge, and mobile without recompilation.

Developer Tooling and Observability Improvements

To improve developer adoption and experience, several tooling efforts are underway:

- **Better debugging support**, including DWARF integration in runtimes and IDE extensions
- **Observability APIs** exposing metrics, tracing, and structured logs in a standardized form
- **Package managers** like wapm, and registries for publishing, versioning, and consuming WASI components

CONCLUSION

The WebAssembly System Interface (WASI) represents a transformative shift in how software can be developed, deployed, and executed across diverse environments. By enabling secure, portable, and lightweight applications that run consistently across operating systems and hardware platforms, WASI brings the long-standing vision of universal binaries closer to reality.

This paper has explored the motivations behind WASI's emergence, contrasted it with traditional application models, and detailed its architecture, toolchains, and real-world use cases. Our experimental evaluation demonstrated that WASI-based applications offer compelling performance—especially in cold start times and binary size—while providing strong security guarantees through a capabilities-based sandbox model.

However, significant challenges remain. The limited scope of current system APIs, lack of full threading support, and immature debugging tools present hurdles to broader adoption. Integration with legacy systems also requires bridging tools and hybrid patterns that may increase architectural complexity.Despite these limitations, the future of WASI is promising. Standardization efforts like WASI Preview 2 and the component model are poised to unlock more expressive and composable application patterns. Integration with AI/ML runtimes, edge computing platforms, and mobile environments signals growing relevance across industries. Simultaneously, improvements in developer tooling, observability, and packaging ecosystems are making WASI increasingly viable for mainstream software development.

As WebAssembly continues to evolve from a browser technology into a foundational execution format for the cloud, edge, and beyond, WASI will play a central role in ensuring that the code we write today is portable, secure, and future-proof.

REFERENCE

1. Ţălu, M. (2025). A comparative study of WebAssembly runtimes: performance metrics, integration challenges, application domains, and security features. Archives of Advanced Engineering Science, 1-13.

2. Van Kenhove, M., Seidler, M., Vandenberghe, F., Dujardin, W., Hennen, W., Vogel, A., ... & Volckaert, B. (2024). Cyber-physical WebAssembly: Secure Hardware Interfaces and Pluggable Drivers. arXiv preprint arXiv:2410.22919.
3. 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.
4. Waseem, M., Das, T., Ahmad, A., Liang, P., & Mikkonen, T. (2024, June). Issues and their causes in WebAssembly applications: An empirical study. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (pp. 170-180).
5. Al-Baldawi, A. J. A. (2024). Setting up WebAssembly in a Cloud Environment that Supports AMD Secure Encrypted Virtualization.
6. Nguyen, S. D. (2025). Optimizing Web Performance and Computational Efficiency: A Deep Dive into WebAssembly's Technical Advancements and Real-World Applications.
7. Wen, E., Weber, G., & Nanayakkara, S. (2022). WasmAndroid: A Cross-Platform Runtime for Native Programming Languages on Android. ACM Transactions on Embedded Computing Systems, 22(1), 1-19.
8. Kakati, S., & Brorsson, M. (2023, June). Webassembly beyond the web: A review for the edge-cloud continuum. In 2023 3rd International Conference on Intelligent Technologies (CONIT) (pp. 1-8). IEEE.
9. Martins, P. J. P. (2021). Development of an e-portfolio social network using emerging web technologies (Master's thesis, Universidade do Minho (Portugal)).
10. Yang, Y., Hu, A., Zheng, Y., Zhao, B., Zhang, X., & Quinn, A. (2024). Transparent and Efficient Live Migration across Heterogeneous Hosts with Wharf. arXiv preprint arXiv:2410.15894.
11. Wen, E. (2020). Browserify: Empowering Consistent and Efficient Application Deployment Across Heterogeneous Mobile Devices (Doctoral dissertation, University of Auckland).