



PhD- FSTM-2026-013
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 27/02/2026 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN *INFORMATIQUE*

by

Sangeeta KAKATI

Born on 19 May 1998 in Assam, India

CROSS-ARCHITECTURE APPLICATION PORTABILITY USING WEBASSEMBLY IN DISTRIBUTED EDGE- CLOUD SYSTEMS

Dissertation defence committee

Prof. Dr. Radu STATE, Chairman
Professor, Université du Luxembourg

Dr. Mats BRORSSON, Supervisor
Research Scientist, Université du Luxembourg

Dr. Merlijn SEBRECHTS, Member
Ghent University/ IMEC

Dr. Rajarshi SANYAL, Member
Proximus SA, Luxembourg

Prof. Dr. Teqawendé BISSYANDE, Member
Professor, Université du Luxembourg

AFFIDAVIT

I hereby, declare that the PhD thesis entitled “**Cross-Architecture Application Portability Using WebAssembly in Distributed Edge-Cloud Systems**” has been written independently and without any other sources than cited.

Luxembourg, _____

SANGEETA KAKATI

ABSTRACT

Cloud computing has transformed the IT landscape by enabling elastic resource provisioning with minimal upfront cost. However, as distributed and client-facing applications increasingly migrate toward the network edge, the once homogeneous cloud has evolved into a fragmented ecosystem spanning diverse architectures and accelerators. Developers now face the challenge of deploying and efficiently executing applications across heterogeneous platforms without detailed knowledge of underlying hardware. This growing complexity demands a portable and architecture-agnostic execution model that bridges the cloud–edge continuum.

In this thesis, we explore WebAssembly (Wasm) as a unifying abstraction for heterogeneous and distributed environments. We conduct a comprehensive review of WebAssembly beyond the web, identifying key challenges related to runtime efficiency and cross-architecture interoperability. Building on this foundation, we perform an extensive performance study of runtimes across X86_64, ARM64, and RISC-V platforms, providing a detailed cross-architecture characterization in terms of startup latency, execution speed, and compiler trade-offs.

We further examine the role of WebAssembly in containerized infrastructures by comparing Wasm-based containers with traditional containers under Docker and containerd runtimes. Our results show that Wasm containers achieve up to 70% smaller image sizes and 25% faster cold pulls while maintaining comparable startup times, establishing WebAssembly as a compact and efficient alternative for multi-platform deployment.

Finally, we investigate the extensibility of the WebAssembly System Interface (WASI) through experimental host-side services implemented in Wasmtime. We design and implement prototypes that introduce capability-based abstractions for host-managed memory and data-centric operations, including dataframe-style analytics. Through these prototypes, we demonstrate how components can interact with host-provided services using handle-based interfaces, illustrating a viable design path for offloading computation beyond the Wasm sandbox.

Overall, we contribute to both theory and practice by (i) establishing empirical foundations for understanding WebAssembly performance across heterogeneous systems, (ii) validating its potential as a portable runtime for containerized applications, and (iii) providing an initial exploration of how WASI can be extended toward host-assisted and data-aware execution. The findings highlight WebAssembly’s promise as a sustainable, cross-platform substrate for next-generation cloud–edge computing.

Contents

Acknowledgments	iii
Affidavit	iv
Abstract	v
1 Introduction	1
1.1 Motivation	7
1.1.1 Portability and the Limits of Traditional Approaches	8
1.1.2 WebAssembly as a Potential Solution	8
1.1.3 Performance, Usability, and Open Challenges	9
1.2 Research Questions	10
1.3 Scope and Limitations	11
1.3.1 Scope of the Thesis	11
1.3.2 Limitations	12
1.3.3 Positioning Within the Research Landscape	12
1.4 Main Contributions	13
1.4.1 Surveying WebAssembly Beyond the Web	13
1.4.2 Investigating Runtime Performance Using SightGlass	13
1.4.3 Cross-Architecture Evaluation with WaBench	14
1.4.4 Performance and Usability of WebAssembly Containers	15
1.4.5 Extended Journal Study: Usability and Sustainability	15
1.4.6 GPU Interfaces and WASI Extensions	16
2 Foundation	17
2.1 WebAssembly introduction	17
2.1.1 WebAssembly System Interface: WASI	18
2.1.2 Compilers	20
2.2 WebAssembly Runtimes Beyond the Web	21
2.2.1 Optimization and Features	22
2.2.2 Execution Modes	23
2.3 Benchmarks	23
2.4 Gap Analysis	24
2.4.1 The WebAssembly System Interface	26

2.4.2	WasmCloud	27
2.5	System Interfaces and WebAssembly	29
2.5.1	Portability	30
2.5.2	Security	30
2.5.3	Designing the WebAssembly System Interface	31
2.5.4	wasi-core: The Foundational Module	32
2.5.5	Host Integration and Portability	32
2.5.6	Capability-Based Security in WASI	33
2.6	Related Work	34
3	Methodology	41
3.1	Methodology Overview	42
3.2	Empirical Evaluation	43
3.3	Multi-Architecture Deployment	44
3.4	Container Runtime Integration	44
3.5	Conceptual Interface Exploration	45
4	Cross-Architecture Evaluation of Portability in Cloud-Edge	46
4.1	Introduction	46
4.2	Background	51
4.3	Related Work	54
4.4	Methodology	57
4.4.1	Performance Metrics Utilized	58
4.4.2	Experimental Setup	60
4.4.3	WebAssembly Runtimes Used	61
4.4.3.1	Runtime 1: Wasmtime	62
4.4.3.2	Runtime 2: WAMR (WebAssembly Micro Runtime)	62
4.5	Results	63
4.5.1	Execution Speed	64
4.5.2	Startup Times	68
4.5.3	WAMR Execution Modes	70
4.5.4	Ahead-of-time compilation	71
4.5.5	Wasmtime: A High-Performance Runtime	72
4.5.6	Insights from WebAssembly Micro Runtime	73
4.5.7	Considerations for Runtime Selection	75
4.6	Discussion	75
4.6.1	Opportunities for Future Research	75
4.7	Conclusion	76

5	Multiplatform and WebAssembly Containers: The Future of Portable Compute	78
5.1	Introduction	80
5.2	Background	82
5.3	Related Work	83
5.4	Methodology	85
5.4.1	Workloads	85
5.4.2	Container Configurations	87
5.4.3	Benchmarking Approach	89
5.5	Results	90
5.5.1	Evaluation of Startup Times: Impact of Dockerfile Practices	91
5.5.2	Image Size Comparison	94
5.5.3	Host Architecture Sizes vs. OCI Registry Sizes	94
5.5.4	Model of Startup Time	97
5.5.4.1	Image Pull Times	97
5.5.5	Correlation between Docker Image Size and Pull Time	98
5.6	Conclusion	99
6	The Small Container: Lightweight Compute for Multi-Architecture Cloud-Edge Systems	100
6.1	Introduction	100
6.2	Background	103
6.3	Related Work	105
6.4	Methodology	107
6.4.1	Workloads	109
6.4.2	Container Configurations	111
6.4.3	Benchmarking Approach	113
6.4.4	Updated Runtime and Benchmarking Revisions	114
6.5	Results	115
6.5.1	Role of Dockerfile Practices on Container Startup Time	117
6.5.2	Image Size Comparison	118
6.5.3	Host Architecture Sizes vs. OCI Registry Sizes	119
6.5.4	Model of Startup Time	120
6.5.5	Using Containerd to Handle Images	121
6.5.5.1	Image Pull Times	122
6.5.6	Extended Benchmarks with Containerd: benchx and SQLite	123
6.5.6.1	Benchx	124
6.5.6.2	SQLite	124
6.6	Conclusion	127

7	Wasm for GPU acceleration: Are we there yet?	129
7.1	Introduction	130
7.2	Background and Motivation	131
7.2.1	WASI-NN: Neural Network Acceleration	133
7.2.2	WASI-GFX: Graphics and Compute Acceleration	135
7.2.3	Comparison of GPU Acceleration Approaches	136
7.2.4	WebAssembly Host Functions and WIT Integration	136
7.3	Design	138
7.3.1	APIs	138
7.3.2	Resource Model	139
7.4	Implementation Details	139
7.4.1	wasi-dataframe	139
7.4.2	wasi-accelerator	139
7.5	Experimental Methodology	140
7.5.1	Benchmarks	141
7.5.2	Reproducible Commands	141
7.5.3	Example Client Implementation	141
7.6	Conclusion	142
8	Discussion	145
8.1	Exploratory Analysis	148
8.1.1	WebAssembly Fundamentals	148
8.1.2	The Expansion Beyond the Browser: WASI and WasmCloud	149
8.1.3	WebAssembly in Edge and Cloud Computing	150
8.1.4	Performance Landscape and Research Challenges	151
9	Limitations and Future Work	152
9.1	Current Limitations	152
9.2	Future Work	153
9.3	Further Research Questions	155
10	Conclusion	157
11	List of Publications	160
	List of Publications	160
	Bibliography	169

List of Figures

2.1	Study overview	18
2.2	Depiction of WebAssembly and WASI.	19
2.3	Architecture and representation of the WasmCloud framework.	28
4.1	WebAssembly and its use in the cloud-edge continuum.	50
4.2	Timestamps and regions of interest.	58
4.3	Representation of Startup Time and Execution Time.	59
4.4	NVIDIA Jetson Nano embedded board used for experiments.	60
4.5	StarFive VisionFive2 RISC-V board used for experiments.	61
4.6	Relative execution time per architecture.	65
4.7	Absolute execution time per architecture.	66
4.8	Absolute startup times per architecture. In (a), the y-axis is measured in seconds, while in (b), it is scaled by a factor of 10.	69
4.9	Relative execution time for different WAMR execution modes on X86_64.	71
4.10	Startup time for different WAMR execution modes on X86_64.	71
4.11	Relative execution performance when performing ahead-of-time compilation of WebAssembly bytecode across architectures.	74
5.1	Workflow for cross-architecture container execution.	81
5.2	Containerized Applications: Native and Wasm	83
5.3	Overview of native and wasm container workflows	86
5.4	Wasi support with containerd shims.	88
5.5	Startup Times (image pull and container time) for Matrix Multiplication.	90
5.6	Startup Times (image pull and container time) for Sorting Benchmark.	90
5.7	Pull and Startup Time for AM64 and ARM64: Without best Practices.	92
5.8	Host-based Image Sizes comparison.	94
5.9	Image Sizes from OCI Registry for the AMD64 architecture.	96
6.1	Native and wasm container workflows, expanded from 5.	108
6.2	NVIDIA Jetson Orin Nano board (Cortex-A78AE)	110

6.3	NVIDIA Jetson Nano embedded board (Cortex-A57).	111
6.4	StarFive VisionFive2 RISC-V board (SiFive U74).	111
6.5	Runwasi Architecture	112
6.6	Containerd shim integration with Wasmtime runtime adapted from Chapter 5.	113
6.7	Startup Times (image pull and container time) for two workloads: (a) Matrix Multiplication and (b) Sorting, updated from Chapter 5.	116
6.8	Pull and Startup Time: Without Best Practices	117
6.9	Host-based Image Sizes Comparison developed from Chapter 5.	118
6.10	Image Sizes from OCI Registry for the AMD64 architecture.	119
6.11	Representation of native and Wasm container images in DockerHub	122
6.12	Average Pull Time using <code>containerd</code> across architectures.	123
6.13	Pull time, startup latency, and execution duration for Wasm and Native containers using <code>ctr</code> .	125
6.14	Breakdown for SQLite benchmark across architectures.	126
7.1	Wasi-nn Backend and Bindings	134
7.2	Workflow of the WASI-NN Interface	135
7.3	Illustration of the WASI-GFX architecture and its relationship to Web and native layers.	136
7.4	High-level dataflow between guest, component adapter, and host providers.	139

List of Tables

2.1	Comparison of standalone WebAssembly runtimes and their compilation modes.	22
2.2	Background: WebAssembly Outside Web	37
4.1	Hardware architectures used in experiments.	60
4.2	Size of WebAssembly modules compared to statically linked native executables.	67
5.1	Hardware Architectures Used in Experiments	87
5.2	Sizes of benchmark programs in native and WebAssembly formats. .	95
5.3	Host Architecture Sizes vs. OCI Registry Sizes for Sorting Benchmark.	95
5.4	Host Architecture Sizes vs. OCI Registry Sizes for Matrix Multiplication Benchmark.	96
5.5	Comparison of Pull Time Reduction between Wasm and Native Containers.	98
6.1	Hardware Architectures Used in Experiments.	110
6.2	Comparison of Host vs. OCI Image Sizes (KB) and Reduction % across Benchmarks.	119
6.3	Architecture-Specific Performance Summary from Sqlite-multiarch	126
7.1	Comparison of GPU acceleration approaches in WebAssembly runtimes.	137

Chapter 1

Introduction

Contents

1.1 Motivation	7
1.1.1 Portability and the Limits of Traditional Approaches	8
1.1.2 WebAssembly as a Potential Solution	8
1.1.3 Performance, Usability, and Open Challenges	9
1.2 Research Questions	10
1.3 Scope and Limitations	11
1.3.1 Scope of the Thesis	11
1.3.2 Limitations	12
1.3.3 Positioning Within the Research Landscape	12
1.4 Main Contributions	13
1.4.1 Surveying WebAssembly Beyond the Web	13
1.4.2 Investigating Runtime Performance Using SightGlass	13
1.4.3 Cross-Architecture Evaluation with WaBench	14
1.4.4 Performance and Usability of WebAssembly Containers	15
1.4.5 Extended Journal Study: Usability and Sustainability	15
1.4.6 GPU Interfaces and WASI Extensions	16

The rapid expansion of distributed computing has given rise to what researchers describe as the *computing continuum*, a paradigm where computational resources span seamlessly from the cloud to the edge, enabling applications to execute dynamically across heterogeneous environments [59]. This concept extends beyond traditional cloud or edge boundaries, representing an evolution toward *computing without borders*, where workloads flow fluidly according to latency, energy, and performance requirements [27]. The continuum embodies a shift toward more adaptive and sustainable digital infrastructures, allowing workloads to

be placed where they can operate most efficiently across devices, networks, and data centers.

Recent studies have emphasized that this trend is driven by the convergence of cloud computing, edge computing, and the Internet of Things (IoT), creating a unified infrastructure that integrates billions of connected devices [80]. This evolution has intensified the need for software portability, scalability, and energy efficiency across diverse execution environments. Modern orchestration frameworks such as Kubernetes are being extended to this continuum to handle distributed container deployments, digital twins, and adaptive scheduling mechanisms. For instance, KubeTwin provides a digital twin framework for Kubernetes-based systems, enabling large-scale synchronization of edge deployments with their cloud counterparts [3]. Similarly, Gwydion introduces reinforcement learning-driven auto-scaling to optimize resource allocation across the continuum [75]. These developments demonstrate the growing maturity of distributed orchestration and the increasing complexity of managing portable compute across the cloud-edge spectrum, with ongoing efforts to achieve equilibrium through active inference-based models [78].

This transition from a centralized model based on large data centers to a dispersed and heterogeneous architecture that incorporates edge devices and specialized processing units has not only brought enormous flexibility and scalability but also introduced new challenges in managing diversity across platforms, architectures, and execution environments. It calls for a cross-platform and interoperable solution, a capability that WebAssembly, or Wasm, naturally offers.

Within this landscape, *WebAssembly* (abbreviated as *Wasm*¹ throughout this thesis) has emerged as a promising technology for enabling portable, secure, and efficient execution across heterogeneous systems. Initially designed as a secure, sandboxed runtime for the web, Wasm has rapidly evolved into a universal compilation target for a wide range of programming languages. Its platform-independent bytecode format and efficient execution model make it ideal for environments with constrained resources, including edge nodes and IoT gateways. A key enabler of Wasm’s evolution beyond the browser is the *WebAssembly System Interface* (WASI)², which defines a set of standardized, capability-based

¹<https://webassembly.org/>

²<https://wasi.dev/>

APIs for interacting with host resources such as files, networking, and clocks [44, 25].

The growing use of Wasm in containerized environments highlights its potential as a lightweight alternative to traditional Linux containers, offering faster startup times, smaller image sizes, and reduced attack surfaces. These properties align naturally with the requirements of the continuum, where applications need to migrate seamlessly between devices and data centers without recompilation. Recent extensions to the WASI ecosystem further indicate a trend toward heterogeneous computing support. Proposals such as `wasi-nn`³, which enables machine learning inference using host-provided accelerators, demonstrate how Wasm can be extended beyond CPU-bound workloads. Earlier efforts toward a general-purpose `wasi-gpu`⁴ interface explored similar ideas but have since been deprecated and partially subsumed by newer proposals such as `wasi-gfx`⁵, reflecting the evolving state of standardization in this area.

Despite these promising developments, several open challenges remain. Performance disparities between native and Wasm-based applications persist due to runtime overhead, system call abstraction, and differences in memory models. Additionally, existing container runtimes and orchestration tools are still evolving to fully support Wasm workloads alongside traditional containers. Understanding how Wasm behaves across architectures such as x86, ARM, and RISC-V and under different orchestration strategies is critical for realizing the vision of true portability and performance sustainability across the continuum.

The central research question that guides this work can be stated as follows:

How can distributed applications be deployed in such a way that software can automatically and seamlessly adapt to the multitude of hardware platforms that will form future cloud infrastructures?

This question arises from the changing nature of modern distributed computing. Developers increasingly face the difficulty of deploying and maintaining applications across fragmented and heterogeneous infrastructures composed of diverse processors, accelerators, and memory systems. The once relatively uniform data center environment has given

³<https://github.com/WebAssembly/wasi-nn>

⁴<https://github.com/WebAssembly/wasi-gpu>

⁵<https://github.com/WebAssembly/wasi-gfx>

way to a continuum that spans the cloud, the edge, and end-user devices. Within this landscape, developers must navigate a growing diversity of hardware architectures, including X86, ARM, and RISC-V, as well as an expanding ecosystem of specialized accelerators such as GPUs, FPGAs, and AI processing units.

From the perspective of application developers, it is neither sustainable nor efficient to require deep knowledge of the exact hardware on which each component of a distributed application will execute. Instead, they should be able to focus on logic and functionality while relying on the runtime and deployment infrastructure to adapt automatically to the available hardware. For infrastructure owners, the challenge lies in providing seamless support for new architectures and accelerators without maintaining outdated configurations solely to accommodate legacy applications.

This research has been conducted in the context of the ACE5G project, which aims to develop execution and orchestration models for next-generation cloud-edge infrastructures. A central objective of ACE5G is to move beyond imperative deployment models based on static node labels or architecture-specific binaries. Instead, applications should be deployed descriptively, specifying quality-of-service (QoS) and performance requirements, while the infrastructure dynamically determines where and how components execute. This includes enhanced container orchestration mechanisms capable of adapting execution decisions across heterogeneous platforms, which is an effort closely aligned with parallel work within the project.

WebAssembly offers an ideal foundation for realizing this vision. By providing a secure, language-neutral, and portable binary format that can execute across a variety of environments, Wasm allows developers to write code once and run it anywhere, including across heterogeneous architectures. In the context of the cloud-edge continuum, it can serve as a universal compilation and execution layer that bridges the gap between different processor types and operating environments.

This thesis explores the role of WebAssembly as an enabling technology for portable compute across the edge–cloud continuum. It examines the performance, usability, and deployment implications of WebAssembly containers compared to native containers, investigates cross-architecture performance variations, and introduces mechanisms for accelerating Wasm execution through novel host interfaces for general-

purpose GPU offloading. Through this investigation, the thesis contributes to a broader understanding of how WebAssembly can evolve from a browser-based technology into a cornerstone of next-generation distributed computing.

By addressing the need for performance portability, platform independence, and adaptive deployment, this research lays the foundation for a unified computing model that abstracts away hardware-specific details while maintaining performance and security. The findings presented here aim to demonstrate that WebAssembly can serve not only as a technology for portable execution but also as a key enabler of flexibility, sustainability, and resilience across the computing continuum.

We examine the main challenges and opportunities for Wasm runtimes in the cloud-edge continuum, such as performance optimization, portability, and interoperability with other programming languages and platforms. We also provide a comprehensive overview of the current landscape of Wasm outside the web, including standardization efforts and best practices for using these runtimes, thereby serving as a one-stop resource for researchers and practitioners in the field. This shift represents a key enabler for the **cloud-edge continuum**, which bridges the gap between massive cloud data centers and end-user devices by enabling portable, secure, and efficient application execution across heterogeneous infrastructures.

WebAssembly was initially designed to run within web browsers, using just-in-time (JIT) compilers to translate Wasm modules into native code at runtime. However, its lightweight design, security model, and near-native performance have rapidly expanded its use beyond browsers into serverless and edge environments. Wasm reduces code size, improves execution efficiency, and mitigates security risks associated with running untrusted code in shared environments. More importantly, it provides developer-friendly abstractions that enable applications to run across a wide range of hardware, including CPUs, GPUs, and accelerators, making it an attractive foundation for the future of ubiquitous computing.

A potential application of Wasm in the edge-cloud continuum lies in deploying lightweight, portable applications that can run on diverse devices and architectures. Developers can compile code to run seamlessly on multiple hardware targets, from data center CPUs to embedded GPUs and IoT microcontrollers. The escalation of mobile and perva-

sive computing has given rise to an era of ubiquitous computing, where increasingly heterogeneous devices populate our surroundings. Wasm’s architecture-agnostic execution model is uniquely suited to unify this fragmented landscape.

WebAssembly runtimes are the software environments responsible for executing Wasm binaries. These runtimes can be integrated into browsers, embedded in native applications, or deployed as standalone interpreters in cloud or edge environments. They provide features such as memory management, security isolation, and performance optimizations. For example, the *Wasm Micro Runtime (WAMR)*⁶ is optimized for embedded systems and IoT devices, featuring a small binary footprint and minimal memory usage. In contrast, runtimes like *Wasmtime*⁷, *Wasmer*⁸, and *WasmEdge*⁹ provide more advanced capabilities such as just-in-time (JIT) or ahead-of-time (AOT) compilation, making them suitable for high-performance and cloud-native workloads.

Each runtime exhibits strengths and trade-offs depending on the use case, language integration, and deployment environment. Wasm is increasingly recognized as a versatile technology for performance-critical applications across diverse domains, including cloud/edge computing, media processing, machine learning, and IoT. As Wasm adoption continues to grow, understanding its runtime behavior, interoperability, and performance becomes a key research area.

Our recent research has demonstrated WebAssembly’s potential as an enabler for this continuum by showing that it maintains compact binaries, rapid startup times, and consistent performance across architectures such as x86_64, ARM64, and RISC-V [32]. Moreover, with the evolution of the WebAssembly System Interface, Wasm has become increasingly capable of interacting with host environments securely and portably [25, 44]. WASI extends Wasm’s sandboxed model to include standardized APIs for I/O, file systems, and networking, thus allowing Wasm programs to function as general-purpose executables outside the browser.

At the same time, containerization technologies such as Docker and **containerd** have become central to cloud-native computing, providing

⁶<https://github.com/bytecodealliance/wasm-micro-runtime>

⁷<https://wasmtime.dev/>

⁸<https://wasmer.io/>

⁹<https://wasmedge.org/>

isolated, reproducible, and scalable deployment mechanisms. Integrating Wasm into the container ecosystem merges the isolation and consistency of containers with the portability and efficiency of Wasm. Modern container runtimes like Docker and `containerd` now support pluggable Wasm runtimes (e.g., Wasmtime, Wasmedge), enabling Wasm workloads to be packaged and deployed as containers. This approach eliminates the need for architecture-specific builds and reduces image sizes and startup latencies, with our study showing Wasm containers to be up to 85% smaller and 25% faster to initialize than native Docker containers [33].

Building on this foundation, this thesis explores the design of a **general-purpose host interface for accelerators in WebAssembly**. This work extends the WASI ecosystem by developing two complementary host providers within the Wasmtime runtime: `wasi-dataframe`, a Polars-backed dataframe API for efficient data manipulation, and `wasi-accelerator`, a generic interface for offloading numerical workloads to the host. The latter enables Wasm modules to allocate host memory, perform read/write operations, and execute compute kernels such as matrix multiplication using lightweight handle-based abstractions. Together, these extensions illustrate how WebAssembly can serve as a portable, language-neutral bridge for heterogeneous compute acceleration across the cloud-edge continuum.

Through the design, implementation, and evaluation of these interfaces, this thesis contributes to the broader effort of establishing WebAssembly as a sustainable and resilient execution model for heterogeneous distributed systems. It demonstrates that Wasm, combined with WASI and host-provided extensions, can deliver both performance portability and runtime efficiency across diverse architectures, positioning it as a foundational abstraction for next-generation distributed and accelerated computing.

1.1 Motivation

The last decade has witnessed a dramatic transformation in the way computing resources are organized and delivered. The traditional cloud-centric model, characterized by large-scale datacenters operated by hyperscalers, is increasingly being complemented by a more distributed

infrastructure spanning regional datacenters, edge servers, and end-user devices. Collectively, this evolving ecosystem is often referred to as the *cloud-edge continuum*. In this model, computation may migrate dynamically between centralized and decentralized nodes depending on application requirements, latency constraints, or energy considerations.

This heterogeneity brings both opportunities and challenges. On one hand, it enables new applications such as autonomous driving, augmented reality, and industrial IoT, where low-latency edge processing is essential. On the other hand, it complicates the task of developers who must now consider diverse hardware platforms, ranging from powerful multi-core x86 servers to constrained ARM and RISC-V boards, while ensuring consistent performance and reliability.

1.1.1 Portability and the Limits of Traditional Approaches

Container technologies such as Docker and Kubernetes have become the de facto standard for deploying applications in the cloud. They provide strong isolation, portability within homogeneous environments, and powerful orchestration capabilities. However, native containers remain tied to specific processor architectures: a container image built for x86 cannot execute on ARM or RISC-V without cross-compilation or maintaining multiple architecture-specific builds. This architecture dependence adds complexity to multi-platform deployments and undermines the promise of true portability across the continuum.

Furthermore, while containers provide lightweight virtualization compared to full virtual machines, they still carry non-negligible overhead in terms of startup time, image size, and runtime footprint. In latency-critical or resource-constrained edge scenarios, these costs may be prohibitive.

1.1.2 WebAssembly as a Potential Solution

WebAssembly (Wasm) was introduced by browser vendors in 2017 as a portable, efficient, and secure compilation target for web applications. Its design goals, including compact binary size, fast startup, sandboxed execution, and near-native performance, make it equally compelling outside the browser. Unlike native binaries, Wasm modules are architecture-independent: the same Wasm file can execute on

x86, ARM, and RISC-V devices, provided a suitable runtime is available. This characteristic directly addresses the heterogeneity challenge of the cloud-edge continuum.

Wasm also benefits from an active ecosystem, with runtimes such as Wasmtime, Wasmer, WAMR, and WasmEdge, and integration with container frameworks via projects like `runwasi`. WASI extends Wasm with standardized APIs for system-level functionality, further enabling its use in non-web contexts. Together, these developments suggest that Wasm has the potential to become a universal execution layer across the continuum.

1.1.3 Performance, Usability, and Open Challenges

Despite its promise, Wasm raises several open questions that must be addressed before it can be considered a mature alternative to native containers and runtimes in cloud-edge systems:

- **Performance overheads.** How does Wasm compare to native binaries in terms of execution time, startup latency, and resource usage? What phases of execution (compilation, instantiation, execution) contribute most to overhead?
- **Cross-architecture behavior.** While WebAssembly is architecture-independent by design, actual runtime implementations may vary significantly in performance across x86, ARM, and RISC-V platforms. How portable is Wasm performance in practice?
- **Integration with containerization.** How does Wasm behave when deployed inside container ecosystems such as Docker or containerd? What trade-offs exist compared to native containers, particularly in multi-architecture scenarios?
- **Extending to accelerators.** Can Wasm be extended to leverage GPUs and other accelerators in heterogeneous environments? What gaps exist in current WASI proposals such as `wasi-nn`, and how might they be addressed?

This thesis is motivated by the need to investigate these questions systematically. By conducting empirical evaluations of Wasm runtimes,

containers, and cross-architecture deployments, and by exploring unresolved challenges in accelerator support, we aim to provide both practical insights and conceptual contributions toward positioning Wasm as a first-class execution layer in the cloud-edge continuum.

1.2 Research Questions

The overarching research objective of this thesis is to evaluate the potential of WebAssembly as a unifying execution framework for multi-platform cloud-edge systems. To guide this objective, we articulate the following research questions (RQs):

RQ1: What are the performance characteristics of Wasm runtimes in non-web environments?

Although Wasm was designed for the web, much of its recent adoption occurs in non-web settings such as serverless computing, edge devices, and IoT platforms. It is therefore crucial to measure the performance of Wasm runtimes outside the browser. This includes quantifying the cost of different execution phases (compilation, instantiation, execution) and comparing them to native baselines.

RQ2: How consistent is Wasm performance across heterogeneous architectures?

The promise of Wasm lies in its architecture independence. Yet, runtime implementations depend on compiler backends, JIT engines, and low-level optimizations that may favor some architectures over others. This raises the question of whether performance portability is achievable in practice, or whether Wasm exhibits significant architecture-dependent behavior.

RQ3: How should benchmarking methodologies evolve to capture Wasm’s unique characteristics?

Traditional benchmark suites, such as SPEC CPU, are not directly applicable to Wasm due to differences in runtime environments and system

interfaces. This motivates the development of tailored benchmarking approaches such as SightGlass and WaBench. An important research question is how to design instrumentation and measurement methodologies that yield reliable and comparable insights into Wasm performance.

RQ4: How do Wasm containers compare to native containers in terms of performance and usability?

While Wasm modules can be executed directly by runtimes, integration with container ecosystems is crucial for adoption in cloud-edge environments. Wasm containers promise smaller image sizes and faster pull times, but may incur performance penalties compared to native containers. Understanding these trade-offs is essential for evaluating Wasm’s practical utility.

RQ5: What are the open challenges in extending Wasm to accelerators such as GPUs?

Emerging workloads in machine learning and high-performance computing increasingly rely on accelerators. Existing WASI proposals (e.g., `wasi-nn`) provide limited support and are often tied to specific domains. A broader question is how Wasm can be equipped with general-purpose accelerator interfaces that preserve portability and security while enabling performance.

1.3 Scope and Limitations

1.3.1 Scope of the Thesis

This thesis focuses on evaluating and advancing the role of WebAssembly in non-web contexts, specifically within the cloud-edge continuum. The key scope elements are:

- **Target architectures.** We evaluate Wasm runtimes and containers across x86-64, ARM64, and RISC-V64 platforms, representing cloud servers, edge devices, and emerging architectures.
- **Runtimes.** The primary runtimes studied are Wasmtime (Rust-based, JIT-compiled) and WAMR (C-based, modular, with multi-

ple execution modes). These are chosen due to their prominence within the Bytecode Alliance and their availability across architectures.

- **Benchmarking.** We employ and extend benchmark suites including SightGlass and WaBench, introducing instrumentation for precise measurement of runtime phases.
- **Containers.** We evaluate Wasm containers in comparison to native multi-platform containers, using Docker and containerd as representative ecosystems.
- **Accelerator interfaces.** We investigate the conceptual design of WASI-based GPU interfaces, using `wasi-nn` as a starting point and proposing directions for general-purpose accelerator support.

1.3.2 Limitations

It is important to acknowledge the limitations of this thesis:

- **Compiler optimizations.** We do not propose new compiler backends or low-level optimizations for Wasm. Instead, we evaluate existing toolchains and runtimes.
- **Workload diversity.** While we use representative benchmarks and workloads, they do not cover all possible application domains. Real-world large-scale applications such as complex ML models or HPC workloads are beyond scope.
- **Energy consumption.** Although energy efficiency is relevant in edge environments, our focus is primarily on performance and usability metrics (execution time, startup time, image size).
- **GPU evaluation.** Our work on accelerator interfaces remains at the level of conceptual proposals and prototypes. A full empirical evaluation of GPU-accelerated Wasm workloads is deferred to future work.

1.3.3 Positioning Within the Research Landscape

By clearly delineating its scope and limitations, this thesis positions itself as an empirical and conceptual investigation of WebAssembly in

heterogeneous, non-web environments. It does not seek to replace compiler or runtime developers, but rather to inform them with systematic evaluations, and to provide system designers and practitioners with evidence-based insights into the trade-offs of adopting Wasm in multi-platform cloud-edge systems.

1.4 Main Contributions

The primary contributions of this thesis are organized along five thematic strands, each corresponding to one or more of the published works that constitute the backbone of this research. Together, they provide a holistic evaluation of WebAssembly’s performance, usability, and future potential in cloud-edge systems.

1.4.1 Surveying WebAssembly Beyond the Web

The first contribution is a comprehensive review of WebAssembly runtimes and their applicability outside the browser. While Wasm emerged as a technology for web applications, its evolution has been rapid and diverse, extending to serverless computing, IoT, and container ecosystems. Our review systematically categorized existing runtimes (e.g., Wasmtime, Wasmer, WAMR, WasmEdge), surveyed the WASI ecosystem, and highlighted areas where Wasm promises to complement or disrupt traditional paradigms. The survey revealed open gaps, particularly regarding performance characterization, cross-architecture consistency, and integration with accelerators. This review serves as both a foundation and a roadmap, motivating the empirical investigations that follow.

1.4.2 Investigating Runtime Performance Using SightGlass

The second contribution is an in-depth performance study of Wasmtime using the SightGlass benchmark suite. This study represents one of the earliest attempts to dissect runtime phases (compilation, instantiation, execution) across heterogeneous architectures (x86-64, ARM64, RISC-V64). Key insights include:

- Cycle-level measurements reveal that Wasm achieves near-native performance in execution phases, but compilation and instantiation

introduce non-trivial overheads.

- Architectural differences are significant: x86 leads in absolute cycle counts, ARM provides a balanced trade-off, and RISC-V lags due to immature backend optimizations.
- The SightGlass suite, while useful, highlights the need for enhanced instrumentation to capture Wasm-specific performance nuances.

This study validates Wasm as a viable runtime for heterogeneous environments, while also motivating methodological improvements.

1.4.3 Cross-Architecture Evaluation with WaBench

Building on the investigative study, the third contribution expands performance analysis into the first systematic cross-architecture evaluation of Wasm runtimes. By instrumenting the WaBench suite and introducing precise timestamping, we quantified startup latency, execution performance, and runtime overhead across four architectures: x86-64, ARM Neoverse-N1, Nvidia Jetson (ARM Cortex-A57), and StarFive VisionFive2 (RISC-V U74). The novelty of this work lies in:

- **Instrumentation.** Our methodological innovation allows us to isolate runtime startup time (from shell invocation to main function) from execution time (region-of-interest), producing fine-grained insights.
- **Multi-runtime comparison.** We evaluated both Wasmtime and WAMR, exploring interpreter, fast JIT, LLVM JIT, and AOT modes, demonstrating trade-offs across platforms.
- **Architecture-dependent behavior.** Our findings reveal that Wasm’s portability is not synonymous with performance portability. Runtime behavior is significantly shaped by the underlying microarchitecture and maturity of backend compilers.

This contribution provides a benchmarked baseline for future Wasm performance studies across heterogeneous platforms.

1.4.4 Performance and Usability of WebAssembly Containers

The fourth contribution examines the integration of WebAssembly with container ecosystems, specifically Docker and containerd. We compare Wasm containers (executed with Wasmtime via runwasi) to native multi-platform containers across three workloads (matrix multiplication, sorting, face detection) and three architectures (x86-64, ARM64, RISC-V64). Our empirical findings show:

- Wasm containers are significantly smaller (average $\sim 27\%$ of native image size).
- Pull times are reduced by up to 25% using containerd, particularly benefiting resource-constrained environments.
- Startup times are slightly higher than native containers, but the gap is small relative to the benefits of reduced size and portability.
- From a usability perspective, Wasm containers fit naturally into container ecosystems, requiring only minimal changes to runtime configuration.

This work provides the first comprehensive usability and performance comparison of Wasm containers in multi-architecture environments, contributing empirical evidence to debates on Wasm's role in containerization.

1.4.5 Extended Journal Study: Usability and Sustainability

The fifth contribution extends the container study into a journal article, adding new material and broader framing. In this extension, we incorporated:

- Enhanced optimization techniques for multi-architecture Dockerfiles and Wasm binaries.
- Broader usability considerations, including developer effort, CI/CD integration, and orchestration compatibility.
- A sustainability perspective, analyzing how Wasm's smaller images and reduced pull times may translate to lower network energy usage and carbon footprint.

This contribution situates Wasm not merely as a performance optimization, but as a technology with potential sustainability benefits in multi-platform deployments.

1.4.6 GPU Interfaces and WASI Extensions

Finally, this thesis contributes a conceptual exploration of GPU interfaces for Wasm. Recognizing that many modern workloads rely on accelerators, we analyzed the limitations of existing WASI proposals such as `wasi-nn`, and outlined the design of a general-purpose WASI-GPU interface. The proposed design adopts CUDA-inspired abstractions (memory handles, copy operations, compute kernels) while respecting Wasm’s sandbox and capability-based security. While still at an early stage, this exploration highlights the challenges of balancing portability, security, and performance when extending Wasm to heterogeneous accelerators.

Chapter 2

Foundation

Contents

2.1	WebAssembly introduction	17
2.1.1	WebAssembly System Interface: WASI	18
2.1.2	Compilers	20
2.2	WebAssembly Runtimes Beyond the Web	21
2.2.1	Optimization and Features	22
2.2.2	Execution Modes	23
2.3	Benchmarks	23
2.4	Gap Analysis	24
2.4.1	The WebAssembly System Interface	26
2.4.2	WasmCloud	27
2.5	System Interfaces and WebAssembly	29
2.5.1	Portability	30
2.5.2	Security	30
2.5.3	Designing the WebAssembly System Interface	31
2.5.4	wasi-core: The Foundational Module	32
2.5.5	Host Integration and Portability	32
2.5.6	Capability-Based Security in WASI	33
2.6	Related Work	34

2.1 WebAssembly introduction

WebAssembly is a low-level binary instruction format designed for efficient execution and compact representation. It is optimized for fast decoding and minimal size, making it suitable for both web and non-web environments. Each WebAssembly binary can also be expressed in a human-readable text format known as WAT (WebAssembly Text).

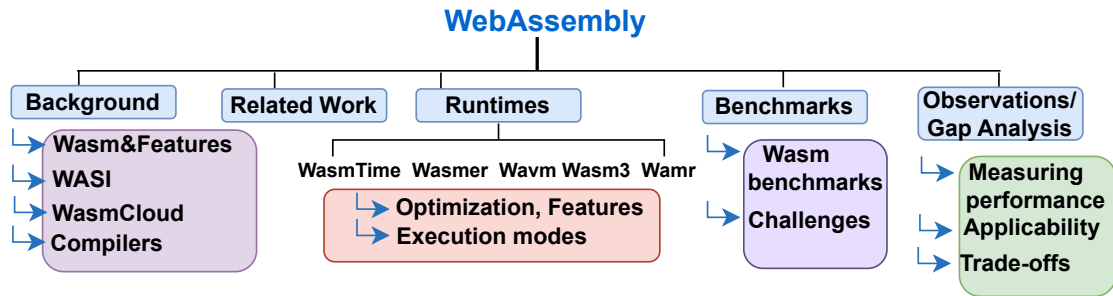


Figure 2.1: Study overview

Unlike most native architectures, Wasm enforces a strong type system. All globals, locals, function parameters, and instruction results have explicitly defined types. Before execution, binaries undergo formal type-checking to ensure correctness. Wasm defines four primitive data types: 32-bit and 64-bit integers (`i32`, `i64`) and single- and double-precision floating-point numbers (`f32`, `f64`). The language itself does not include arrays or pointer types; higher-level constructs in source languages are lowered into these fundamental representations during compilation.

The overall structure of this section follows the organization presented in Figure 2.1.

A WebAssembly module executes within a host environment, which provides access to system-level capabilities. By design, Wasm modules operate in a sandboxed context and cannot directly perform tasks such as file I/O, networking, or system calls. Instead, these capabilities are exposed through host-provided functions imported by the Wasm module. The interaction between WebAssembly and its host environment is illustrated in Figure 2.2 and further detailed in Section 2.1.1.

2.1.1 WebAssembly System Interface: WASI

WebAssembly was originally designed as a portable compilation target for web browsers, enabling safe and efficient execution of low-level code alongside JavaScript. To extend WebAssembly beyond the browser and into standalone and server-side environments, the WebAssembly System Interface was introduced as a standardized system interface for Wasm modules.

WASI defines a POSIX-inspired API that allows WebAssembly mod-

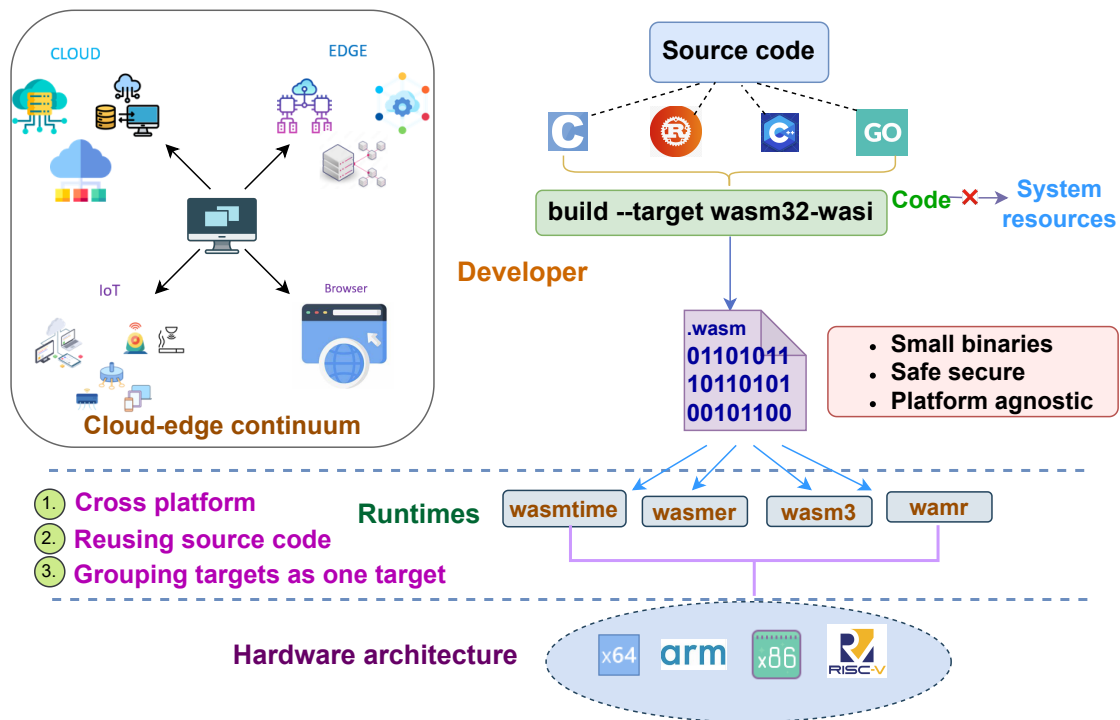


Figure 2.2: Depiction of WebAssembly and WASI.

ules to interact with host system resources in a secure and portable manner. This thesis focuses on **WASI Preview 1 (version 0.1)**, which provides a stable and widely supported interface for standalone Wasm execution. The interface is designed for constrained and heterogeneous environments, such as edge, IoT, and embedded systems, and enables consistent host implementations across platforms.

WASI Preview 1 consists of approximately 46 core functions that support operations such as file I/O, networking, clocks, and interaction with system resources. Language toolchains for Rust, C, and C++ compile standard library calls into WASI-compatible system calls, enabling applications to execute across platforms without modification.

A central design principle of WASI is its capability-based security model. Under this model, a runtime must explicitly grant access to resources such as files, directories, or sockets, ensuring strong sandbox isolation and preventing implicit ambient authority. This security model makes WASI particularly suitable for multi-tenant and untrusted execution environments.

While WASI itself is an abstract interface, WebAssembly runtimes that implement WASI are available across a broad spectrum of devices, ranging from cloud servers to embedded systems such as ESP32 micro-

controllers [73]. In addition, WASI has been adopted in container-centric and orchestration contexts, including Kubernetes controllers and cloud-native tooling [76].

2.1.2 Compilers

Several compiler toolchains support code generation for WebAssembly, enabling programs written in multiple programming languages to target Wasm. Some of the most widely used infrastructures include:

- **LLVM (Low-Level Virtual Machine):** A modular compiler infrastructure that supports multiple programming languages and hardware architectures. LLVM provides a WebAssembly backend capable of generating optimized Wasm binaries from its intermediate representation (IR).
- **Emscripten:** A compiler toolchain built on LLVM that enables programs written in C and C++ to be compiled to WebAssembly. It provides additional runtime libraries and API emulation for both web-based and standalone execution.
- **Binaryen:** A collection of tools and libraries designed for optimizing and transforming WebAssembly binaries. It includes `wasm-opt`, which applies Wasm-specific optimizations to improve performance and reduce binary size.
- **Rust toolchain (`rustc`):** The Rust compiler includes native support for WebAssembly targets, allowing Rust programs to be compiled directly to Wasm with minimal configuration.

Among these infrastructures, LLVM is the most versatile, as it supports multiple language frontends and backend targets. The general compilation workflow for targeting WebAssembly using LLVM can be summarized as follows:

- A language frontend parses the source code and generates LLVM Intermediate Representation (IR), performing syntax analysis and preliminary optimizations.
- LLVM optimization passes further refine the IR to improve performance and eliminate redundancy.

- The WebAssembly backend translates the optimized IR into binary WebAssembly code.
- Optionally, tools such as `wasm-opt` from the Binaryen project can be applied to perform additional WebAssembly-specific optimizations, further reducing binary size and improving execution efficiency.

This compilation process enables programs written in diverse languages to target WebAssembly while retaining strong performance guarantees and platform independence.

2.2 WebAssembly Runtimes Beyond the Web

Outside of browsers, WebAssembly depends on dedicated runtime environments capable of loading, executing, and managing Wasm modules. These runtimes, also called *Wasm engines*, enable developers to write portable code in multiple languages and execute it across platforms such as servers, desktops, and embedded devices. The WebAssembly System Interface standard ensures a consistent, secure interface between Wasm and the host system. Prominent runtimes like *Wasmtime* and *Wasmer* natively support WASI, facilitating execution in sandboxed environments. This section provides an overview of key standalone runtimes.

Wasmtime: Developed under the Bytecode Alliance¹, Wasmtime is an open-source WebAssembly runtime optimized for performance, memory efficiency, and security. It supports both Just-in-Time (JIT) compilation, where WebAssembly bytecode is translated into native machine code at runtime, and Ahead-of-Time (AOT) compilation, where WebAssembly modules are compiled into native binaries before execution. Through its use of the Cranelift compiler backend, Wasmtime enables flexible trade-offs between startup latency and runtime performance while maintaining portability across devices. It is compact, fast, and compliant with official WebAssembly test suites, making it a widely used and actively maintained runtime.

Wasmer: Similar to Wasmtime, Wasmer supports multiple compilers and WASI integration. Its modular architecture allows developers to swap compilation backends and sandboxing mechanisms. It achieves

¹<https://wasmtime.dev/>

near-native performance in sandboxed environments and benefits from a broad community base.

Lucet: Lucet, introduced in 2019, is a native compiler and runtime designed for embedding untrusted Wasm modules in applications. It was an early precursor to Wasmtime and has since been integrated into the Bytecode Alliance ecosystem.

WAVM: Primarily implemented in C++, WAVM uses LLVM for code generation and JIT execution. It supports major operating systems such as Linux, Windows, and macOS. Despite its performance benefits, WAVM requires 64-bit virtual address space, which limits its use on smaller or 32-bit hosts.

Wasm3: Wasm3 prioritizes portability and simplicity over raw speed, using a compact interpreter architecture optimized for minimal memory usage and startup latency. However, it lacks advanced memory management and dynamic allocation, which can limit scalability. While slower than JIT-based runtimes, its design makes it suitable for constrained devices.

WAMR: The WebAssembly Micro Runtime (WAMR) is tailored for embedded and IoT environments, emphasizing small footprint and configurability. Supporting interpretation, AOT, and JIT, it achieves near-native execution in AOT mode while maintaining portability and fast startup. Its core, iwasm VM, supports efficient code execution in low-memory environments.

Runtime	Non-web standards	Language	Compilation mode
Lucet	WASI	Rust	AOT
Wasm3	WASI, Custom	C	Interpreted
WasmEdge	WASI	C++	AOT, Interpreted
Wasmer	WASI, Wasm-C-API	Rust, C++	JIT, AOT
Wasmtime	WASI, Wasm-C-API	Rust	JIT
WasmVM	N/A	C++	Interpreted
WAVM	WASI	C++, Python	JIT
WAMR	Wasm-C-API	C	Interpreted, AOT, JIT

Table 2.1: Comparison of standalone WebAssembly runtimes and their compilation modes.

2.2.1 Optimization and Features

Wasm runtimes continue to evolve with increasing focus on performance optimization. Improvements can be achieved through better memory management, leveraging hardware acceleration, employing JIT and AOT

compilation strategies, optimizing code transformations (e.g., loop unrolling, inlining), and profiling workloads to identify bottlenecks. Caching frequently executed code also reduces startup overhead, and hybrid JIT-AOT models are increasingly used to balance performance and portability.

2.2.2 Execution Modes

Most runtimes support three major execution modes: Interpreter, JIT, and AOT. WAMR, for instance, implements all three. The AOT mode produces near-native performance and rapid startup with minimal footprint. Interpreter mode prioritizes minimal memory use, suitable for microcontrollers. WAMR also offers both a classic interpreter (debuggable) and a fast interpreter (optimized internal opcodes). The JIT mode dynamically compiles Wasm to native code at runtime, enabling performance close to native execution. WAMR's two JIT layers LLVM JIT and Fast JIT, which is a balance between compilation time and runtime performance.

2.3 Benchmarks

To evaluate Wasm performance, benchmark suites such as PolyBench, MiBench, JetStream2, and Sightglass are widely employed. Developed by the Bytecode Alliance, Sightglass integrates micro- and macro-level benchmarks measuring startup latency, execution throughput, and memory consumption. These tests span mathematical computation, image processing, and cryptographic workloads, written in multiple languages. It is primarily used to evaluate the performance of Wasmtime and Cranelift in their own development, rather than for runtime comparison purposes.

PolyBenchC includes small scientific kernels for cache and memory pattern analysis, though some metrics are less relevant to Wasm's higher abstraction level. MiBench focuses on embedded systems (e.g., signal processing, networking), whereas JetStream2 primarily benchmarks JavaScript performance and thus may not accurately represent Wasm outside of browsers. Whole-application benchmarks (e.g., face detection, MNIST, BZIP2, and Whitedb) provide comprehensive evaluation but are more resource-intensive.

Selecting the right benchmark depends on workload characteristics. For a comprehensive evaluation, a combination of multiple suites is generally recommended.

2.4 Gap Analysis

Despite progress in Wasm benchmarking and runtime development, several challenges remain. Benchmarking small workloads often overrepresents startup overhead, and instrumentation introduces additional measurement noise. Cold and warm starts can differ substantially, complicating fair runtime comparisons. Averaging multiple runs, using median values, and carefully designing workloads over one-second duration help mitigate these effects.

Further, runtime behavior varies across compilers and architectures. While JIT runtimes exhibit slower warm starts than AOT, AOT sacrifices portability since binaries target specific platforms. For edge devices, lightweight AOT runtimes like WAMR are promising but require architectural tuning. Workloads with sufficient duration and repeated measurements provide more stable insights.

In summary, it is crucial to employ benchmarking strategies that reflect real-world use cases and to consider startup costs, compilation modes, and device constraints in runtime evaluation. Understanding these trade-offs provides a foundation for further exploration into how WebAssembly can serve as a compact and efficient execution environment for distributed cloud-edge systems, supporting lightweight and secure deployment of microservices and serverless workloads.

The recent surge of interest in WebAssembly for cloud and serverless computing, HPC, orchestration platforms, and container security reflects a broader quest for secure, portable, and high-performance software deployment.

WebAssembly, originally specified as a browser-targeted format, is now envisioned as a universal intermediate representation that is capable of delivering platform-independent, secure, and efficient execution for applications across client and server domains. Its adoption is particularly attractive in HPC environments, which traditionally face hurdles with containerization, such as privilege requirements, image sizes, portability issues, and configuration complexity. Leveraging Wasm as

a container alternative for packaging and distributing MPI-based scientific applications has proven feasible, with projects such as MPI-Wasm delivering nearly native performance and drastically reduced binary sizes, thus facilitating easier deployment across heterogeneous clusters[6]. Comparisons to container-focused approaches show Wasm solutions often mitigate typical root privilege and image transfer bottlenecks.

Kubernetes and similar orchestrators have adapted to the emergence of Wasm, allowing modules to be scheduled as side-by-side alternatives to Linux containers within clusters. By extending container runtimes and streamlining operator patterns, platforms can now natively deploy Wasm workloads with integrated support for WASI and custom system APIs, optimizing for elasticity, cold-start minimization, and size reduction. Benchmarks reveal Wasm achieves twofold faster instantiation and an order of magnitude smaller images for stateless functions, although containers often retain a computational edge for long-running tasks. Hybrid orchestration engines further enable flexible runtime selection per workload characteristics. Wasm’s sharp isolation and rapid cold start make it well-suited to event-driven, scalable workloads typical in FaaS deployments. Yet, platform feature parity (e.g., multi-threading) and ecosystem maturity remain ongoing targets.

Across container and Wasm solutions, empirical measurement studies demonstrate nuanced findings. Wasm containers (experimental Docker-Wasm integrations, crun support) offer competitive computation performance and far smaller image sizes, favoring deployment in resource-constrained environments and rapid scale-out scenarios. However, container-level overheads common to Docker and Wasm alike undercut Wasm’s theoretical startup and memory benefits in practice. System interface access via WASI adds fixed overhead, making Wasm containers less ideal for IO-bound workloads unless WASI is specifically optimized. For security, Wasm’s fault isolation and capability-based access curtail attack surfaces compared to traditional containers, raising the baseline for defense against privilege escalation and code injection. Experiments with Kubernetes and OpenShift platforms show that, even with WASI-enabled system access by default, a well-configured Wasm sandbox resists escalation more effectively than containers.

Isolation attacks, runtime escapes, and supply chain vulnerabilities (malicious code injection) remain possible in both paradigms, reinforcing the need for hardened defaults and layered mitigations. While Wasm

isn't a panacea for all security and observability issues, the overall frequency and impact of attacks are reduced under Wasm, especially when WASI privileges are tightly controlled.

The usability and operational efficiency of Wasm containerization depend heavily on orchestrator and runtime support. Extensions to containerd, Podman, and Kubernetes now support direct instantiation of Wasm binaries, leveraging custom runtime classes and shims for seamless devops integration. Artefact distribution through OCI images and registries is simplified due to Wasm's compact size and dependency minimization. Challenges persist, especially around monitoring, debugging, and runtime feature gaps versus containers and micro VMs.

Recent works further highlight the landscape of Wasm-enabled serverless platforms, supporting BaaS integration, outbound HTTP communication, and advanced stateful migration. Multi-tenancy, resource quota enforcement, and runtime extension for edge scenarios are active research areas. Usability studies and FaaS evaluations suggest that Wasm is particularly promising for latency-sensitive, stateless, and security-critical workloads, but less so for long-running, computation-heavy services unless future Wasm proposals mature.

2.4.1 The WebAssembly System Interface

WASI is a formally standardized system interface under the W3C WebAssembly Community Group, designed to enable secure and portable interaction between WebAssembly modules and host system resources.

In computing, a *system interface* defines the boundary through which programs interact with underlying system resources. Although languages such as C are often described as providing “direct access” to these resources, this is not entirely accurate. In practice, even low-level languages do not directly manipulate critical resources such as files, memory, or network connections.

This restriction exists for reasons of stability and security. If one program inadvertently altered or corrupted the resources of another, it could cause system crashes or data loss. More critically, if a malicious program were granted unrestricted access, it could compromise sensitive information or interfere with other applications.

Modern operating systems address this through *protection ring secu-*

urity, a model that isolates user programs from direct access to system resources. The operating system kernel acts as a protective boundary, mediating all privileged operations such as file creation, file access, and network communication. User applications execute in *user mode* outside this boundary and must request access through the kernel.

These requests are made using *system calls*, which serve as controlled entry points for interacting with system resources. When a program issues a system call (for example, to open a file), the kernel verifies the program's permissions before performing the operation. System calls therefore form the fundamental interface between user-space programs and the operating system.

However, since each operating system exposes its own set of system calls, relying on them directly would make software non-portable. This challenge is resolved through *abstraction layers*, typically provided by the language's *standard library*. Programmers interact with a uniform API (e.g., `printf` in C), while the compiler or toolchain links the appropriate system-specific implementation. For instance, on Windows the `printf` function uses the Windows API, whereas on macOS or Linux it relies on POSIX system calls. This abstraction ensures that application code remains portable across platforms, while system-specific details are handled by the corresponding library implementation.

2.4.2 WasmCloud

WasmCloud is a distributed framework designed to facilitate the creation and deployment of portable business logic that can execute seamlessly across a continuum of devices, from the edge to the cloud. Its primary objective is to simplify the developer workflow by removing redundant boilerplate and ensuring that security remains a fundamental property of distributed applications. Figure 2.3 illustrates an overview of the WasmCloud ecosystem. The framework can operate on any platform supporting a WebAssembly runtime with WASI support, including Linux, macOS, Windows, and even embedded environments.

At the core of WasmCloud are the runtimes responsible for executing Wasm modules. The architecture of WasmCloud can be summarized as follows. Developers can write WebAssembly modules in multiple languages such as Rust or C++, and these lightweight modules can easily be shared across applications. The *hosts* act as runtime environments

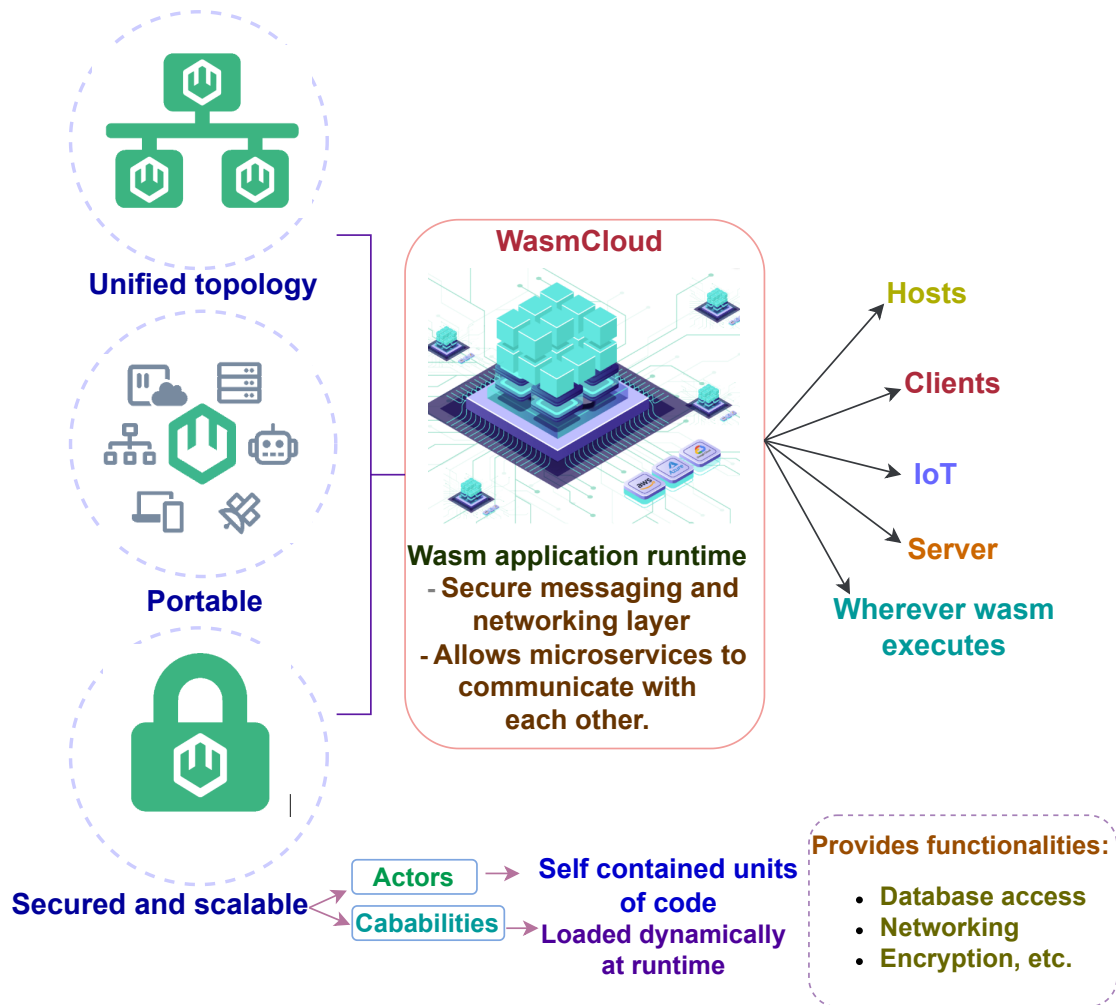


Figure 2.3: Architecture and representation of the WasmCloud framework.

capable of loading and executing Wasm modules across diverse platforms, including Linux, Windows, and Kubernetes clusters. Within a host, each Wasm module is instantiated as an *actor*, which contains the business logic and can communicate with other actors or capability providers.

Actors interact using a lightweight and efficient pub/sub messaging protocol suitable for distributed systems. The *capability providers* are natively compiled components that offer system-level functionalities such as database access or message queuing to the Wasm actors. A centralized *control plane* manages the lifecycle of actors, hosts, and capability providers. It ensures proper authentication and authorization across all components, allowing secure actor-to-actor and actor-to-provider communication.

2.5 System Interfaces and WebAssembly

WebAssembly introduces a unique challenge. Unlike traditional compilation targets, WebAssembly is designed to execute on a *conceptual machine*, rather than a specific operating system. As a result, during compilation, there is no assumption of any particular system interface. The WebAssembly standard library cannot depend on platform-specific APIs such as POSIX or the Windows API.

Despite this, WebAssembly runtimes can execute modules outside the browser even without a universal system interface. Historically, this became possible through *Emscripten*, a toolchain that compiles C/C++ code to WebAssembly by emulating a POSIX-like system interface on the web. This allows developers to use familiar C library functions such as those from `libc`.

Emscripten achieves this by dividing its standard library into two parts: one compiled into the WebAssembly module, and another implemented in *JavaScript glue code*. The JavaScript layer communicates with the browser, which in turn interacts with the host operating system. This layered approach enables WebAssembly code to access resources indirectly via the browser, forming a complex call chain.

When developers later sought to execute WebAssembly outside the browser, early runtimes replicated these Emscripten interfaces. However, these were never intended as public standards. For instance, rather than using a clearly defined API such as `read`, Emscripten's internal glue code employed functions like `_system3(which, varargs)`, passing arguments through linear memory. While this design was appropriate for a browser environment, it is inefficient and unsafe for general-purpose runtimes.

Consequently, many runtimes are effectively emulating internal details of an emulation layer reproducing design choices specific to Emscripten's early architecture. To build a sustainable WebAssembly ecosystem, it is essential to define a standardized, stable, and well-designed system interface that operates independently of such legacy emulation mechanisms and can support diverse execution environments for decades to come.

The design of a WebAssembly system interface must uphold two foundational principles that are intrinsic to WebAssembly itself: *porta-*

bility and *security*. As WebAssembly continues to expand beyond the browser into general-purpose and embedded environments, maintaining these principles becomes critical.

2.5.1 Portability

The POSIX standard provides *source-level portability*, allowing developers to compile the same source code for different machines by linking against different implementations of the standard C library (`libc`). In this model, a single C source file can be recompiled into multiple binaries targeting different platforms.

WebAssembly, however, requires a higher level of portability. It aims to support *binary-level portability*, enabling developers to compile code once and execute the resulting binary across a wide range of platforms and architectures. This model simplifies code distribution and deployment. For example, if Node.js native modules were distributed as WebAssembly binaries, users would no longer need to execute platform-specific build tools such as `node-gyp`, and developers would not have to distribute numerous architecture-specific binaries.

2.5.2 Security

In traditional operating systems, when a program requests an input/output operation, the kernel enforces security through access control mechanisms based on ownership and user groups. A process runs on behalf of a user, inheriting that user's permissions. This model was effective in early multi-user systems, where administrators controlled the installed software, and the main threat was unauthorized access by other users.

Modern computing environments, however, present different challenges. Systems are typically single-user but frequently execute code from third-party libraries and open-source dependencies of varying trustworthiness. In such contexts, the primary risk is that untrusted or malicious code potentially introduced through dependencies could exploit the program's permissions to access or exfiltrate sensitive data. For instance, a compromised library could open arbitrary files or establish unauthorized network connections.

WebAssembly addresses this problem through a fundamentally differ-

ent security model: *sandboxing*. In this model, WebAssembly modules cannot directly invoke operating system calls. Instead, the host environment (such as a browser or a WebAssembly runtime) provides a limited set of imported functions that the module can call. This mechanism allows the host to explicitly define and restrict which capabilities a given module is permitted to use.

While sandboxing alone does not guarantee security since a host could still expose excessive capabilities, it provides a foundation for enforcing strong isolation and capability control. By determining, on a per-application basis, which system calls or host functions are available, the host maintains fine-grained control over each module's access to system resources.

Recent developments in the WebAssembly ecosystem address these challenges through the evolution of WASI and the introduction of the WebAssembly Component Model. While early WASI efforts focused on a low-level, POSIX-inspired system interface (*wasi-core*), newer specifications move toward higher-level, language-agnostic interfaces based on typed imports and exports. The component model formalizes this approach by enabling WebAssembly components to interact through well-defined, versioned interfaces expressed using the WebAssembly Interface Types (WIT). This design reduces ABI fragility, improves composability across languages, and eliminates reliance on ad hoc glue code.

In contrast to monolithic system call emulation, the component model encourages explicit capability sharing and interface-driven composition, allowing runtimes to expose richer host functionality while preserving strong sandboxing guarantees. These developments represent a shift from emulating traditional operating system abstractions toward defining native, WebAssembly-specific system and application interfaces, positioning WASI as a long-term foundation for portable and secure execution across cloud, edge, and embedded environments.

2.5.3 Designing the WebAssembly System Interface

In defining a system interface for WebAssembly, it is essential to maintain these two principles: portability and security. Portability facilitates broad software distribution, while security ensures that hosts and users remain protected from untrusted or malicious code.

The proposed approach to standardizing this interface is the development of a modular set of standard APIs collectively referred to as the WebAssembly System Interface. The foundational module in this effort is known as *wasi-core*.

2.5.4 *wasi-core*: The Foundational Module

The *wasi-core* specification defines the essential functionalities required by most applications, covering areas such as file and network I/O, clocks, and random number generation. Its design is conceptually aligned with POSIX, employing a file-oriented model with calls such as `open`, `close`, `read`, and `write`.

However, *wasi-core* intentionally omits certain POSIX features that do not map well to WebAssembly’s execution model. For instance, the concept of operating system processes does not have a clear equivalent in WebAssembly, and not all runtimes should be expected to support operations such as `fork`. The modular approach allows such functionality to be standardized separately, ensuring broad coverage while maintaining flexibility for lightweight or specialized environments.

Programming languages such as Rust already use *wasi-core* directly within their standard libraries. For example, Rust’s implementation of `open` internally invokes `__wasi_path_open` when targeting WebAssembly. Similarly, for C and C++, the *wasi-sysroot* provides a `libc` implementation built atop *wasi-core* functions. Toolchains such as Clang, Emscripten, and the Rust compiler are being extended to interface natively with the WASI API.

2.5.5 Host Integration and Portability

In execution, the WebAssembly runtime provides the *wasi-core* functions as *imports* to the module. This model enables strong portability: each host platform can implement its own version of *wasi-core*, optimized for its specific environment. Examples include standalone runtimes such as Wasmtime and Lucet, server-side platforms such as Node.js, and even browsers.

This import mechanism also preserves sandboxing. The host determines which functions to expose to each program, thereby controlling

the system calls available to the module. This dual property modular portability and fine-grained capability restriction ensures that WASI simultaneously supports secure and portable execution.

2.5.6 Capability-Based Security in WASI

WASI extends sandboxing through the adoption of *capability-based security* principles. In traditional operating systems, file access is often granted by calling `open` with a file path string, followed by a permission check based on the user's credentials. In contrast, WASI functions operate on *file descriptors*, which are explicit capabilities associated with access rights. A module must be provided with these descriptors by the host or other modules.

This approach prevents arbitrary file access. For example, a module cannot request to open sensitive files such as `/etc/passwd` unless the host explicitly provides access to the relevant directory or file descriptor. The runtime passes predefined descriptors to the module at startup, and the module can propagate or restrict these descriptors as needed when calling other functions or modules.

Such design aligns with the principle of least privilege, ensuring that each module can access only the resources it genuinely requires. This capability-based model draws inspiration from systems such as Cloud-ABI and Capsicum, which emphasize fine-grained delegation of authority. WASI adapts these concepts in a manner that remains compatible with existing code patterns, for example, by supporting relative path operations through `openat`, or through incremental security layers such as `libpreopen`, which predefines accessible file paths for legacy applications.

WASI's modular and capability-oriented design provides a path toward a secure, portable, and future-proof system interface for WebAssembly across diverse execution environments.

Developers are increasingly extending WebAssembly beyond the web browser environment because it offers a fast, scalable, and secure means of executing the same code consistently across different machines and platforms. However, despite these advantages, WebAssembly currently lacks a robust system-level foundation. Programs running outside the browser require a mechanism to interact with system resources such as

files, networks, and memory through a standardized interface. This gap necessitates the creation of a unified system interface for WebAssembly.

WebAssembly is designed as an assembly language for a conceptual machine rather than a specific physical architecture. This abstraction allows WebAssembly binaries to execute seamlessly across diverse hardware platforms. Following the same principle, WebAssembly requires a conceptual system interface, one that is not tied to any particular operating system but instead provides a consistent and secure abstraction layer across all systems.

WASI fulfills this role by defining a standardized, portable, and secure interface that enables WebAssembly modules to perform system-level operations in a controlled manner. The design of WASI aligns closely with the fundamental principles of WebAssembly, portability, and security, ensuring that programs can run reliably across heterogeneous environments without compromising isolation or safety.

Standardization Effort: To establish WASI as a long-term foundation for the WebAssembly platform, a dedicated WASI Subgroup has been formed under the WebAssembly Community Group. This subgroup focuses on formalizing the WASI specification and collaborating with industry partners and open-source contributors to ensure broad adoption and interoperability. The goal is to create a system interface that evolves alongside WebAssembly and remains sustainable for decades to come.

2.6 Related Work

Recent studies have pushed the state of the art in leveraging WebAssembly for high-performance, portable, and secure compute across a variety of domains, including high-performance computing (HPC), deep learning inference, edge computing, and embedded systems.

Chadha et al. explore Wasm as a lightweight distribution format for HPC applications, introducing MPIWasm which is a Wasm embedder that supports MPI-based workflows in HPC environments. Their evaluations show nearly native performance and significant size reductions compared to native binaries, positioning Wasm as an attractive alternative to conventional containers in scientific computing [6].

Advancing serverless AI, Pang et al. propose a featherlight, stateful WebAssembly platform for inference workflows. They introduce Wasm-Flow, presenting efficient uniform memory sharing, direct inter-function synchronization, and an architecture for dynamic GPU batching and affinity-aware workflow scheduling. Experimental results indicate substantial latency and cold start improvements over existing serverless platforms[67].

Focusing on deep learning across the cloud continuum, Khelifa et al. present an extension of Wasm supporting pluggable hardware acceleration. Through TVM/WASI-NN integration and container-native orchestration, they demonstrate their approach achieves stable inference performance across diverse CPUs and GPUs, even outperforming specialized libraries on non-Intel hardware. Their work strongly supports Wasm as a vehicle for portable, hardware-aware inference in cloud-edge scenarios[37]. Another relevant contribution is Feather, proposed in [22], which introduces a Kubernetes-compatible framework for packaging, distributing, and executing edge workloads beyond conventional containers. Feather builds on Virtual Kubelets, enabling multi-runtime images within standard Kubernetes clusters with only minor, optional PodSpec extensions. The system supports both Containerd and OSv unikernel backends, and evaluations demonstrate that unikernel workloads can significantly reduce memory usage by up to 20% for Java applications while incurring a CPU overhead of up to 25%.

At the AI edge, Khelifa et al. conduct a comprehensive empirical study comparing popular Wasm runtimes for executing AI workloads. Results reveal that JIT-based runtimes (e.g., Wasmer and Wasmtime) achieve near-native performance on small networks, while interpreters like WAMR provide minimal memory usage, making them ideal for constrained devices. The analysis offers practical guidance for selecting runtimes based on hardware and model scale [38].

Bachmeier et al. evaluate Wasm and the wasi-nn proposal for edge machine learning inference, highlighting the current experimental status of wasi-nn and how reliance on native model engines impairs full portability. Their findings suggest that while Wasm offers a compelling solution for isolation and flexibility at the edge, further ecosystem maturation is essential for robust, production-grade edge ML[1].

The integration of hardware accelerators with WebAssembly has gained significant attention, demonstrating potential for enhancing performance

across diverse applications. Several works have explored the use of WebAssembly in embedded systems. Kim et al. [39] introduced a hardware-based WebAssembly accelerator tailored for embedded environments, showcasing substantial performance gains by offloading computational tasks to specialized hardware. Similarly, Rupp et al. [74] focus on accelerating Wasm interpreters in embedded systems using hardware-assisted dispatching. By offloading computed goto operations to hardware accelerators within RISC-V cores, their approach yields a significant execution speedup for interpreted Wasm, demonstrating both minimal size overhead and broad applicability for embedded software.

Finally, Hidaka and Harada introduce WgPy, a GPU-accelerated, NumPy-like array library implemented in Wasm for web browsers. WgPy showcases the growing maturity of Wasm runtimes to support scientific computing directly in-browser, further widening the reach and relevance of Wasm in both HPC and everyday scientific software[24].

A summary of a part of the related literature is shown in Table 2.2.

WebAssembly has emerged as a promising runtime for serverless and edge computing due to its lightweight isolation, fast startup times, and portability. [17] demonstrated that replacing Docker with WebAssembly runtimes in serverless platforms reduces cold-start latency significantly and improves memory efficiency. Similarly, [65] introduced FunLess, a FaaS platform using WebAssembly for private edge clouds, highlighting its suitability for resource-constrained environments. [40] further explored WebAssembly as an enabler for next-generation serverless computing.

Several studies have compared the performance of WebAssembly and container-based runtimes. [47] investigated joint optimization of microservice consolidation and deployment, showing that a hybrid Wasm-container approach can reduce costs substantially. [8] found that WebAssembly configurations run faster, use less memory, and produce smaller images than Docker containers. [49] conducted a comprehensive measurement study, revealing that while WebAssembly containers introduce overhead compared to standalone runtimes, they remain competitive in specific scenarios.

The integration of WebAssembly with Kubernetes has been a focus of recent research. [77] presented a framework for running Kubernetes controllers as WebAssembly modules, achieving 64% memory reduction.

Table 2.2: Background: WebAssembly Outside Web

Research objective	Addressing WebAssembly
Analysing performance of WebAssembly [28].	Analysis of Wasm compiled Unix applications inside the browser.
Performance of Wasm applications vs javascript [96].	Comparison of WebAssembly JIT compilers and javascript
WebAssembly study in the web [89].	JIT optimization of Wasm and javascript in chrome
Analysing performance and portability of WebAssembly for non web environments [84].	Current state of WebAssembly and its system interface
Identifying performance gap by analysing cranelift and llvm [91].	Proposal for designing WebAssembly runtimes
Construction of a benchmark suite (wabench) [90].	Evaluation of WebAssembly runtimes in wabench
Vulnerabilities of Wasm vs native node [44].	Security analysis of Wasm binaries
WebAssembly as a universal interface for the cloud-edge continuum [57].	Benchmarking Wasm(PolyBench) in x86 and arm.
An overview of implementation of Wasm in the browsers [71].	Discussion on the development, design and implementation experiences of WebAssembly.
Vulnerabilities in Wasm binaries [25].	Exploitation of WebAssembly with insecure source languages.
Proof of soundness in the type system of Wasm [94].	Implementation of type checker and interpreter.
Analyzing and understanding WebAssembly codes at runtime [45].	A dynamic analysis approach to examine the behavior of Wasm modules during runtime, with a focus on security analysis.
Investigate existing memory safety operations in WebAssembly [10].	Enhance Wasm for improving memory safety.
Feasibility of using Wasm to bring seamless cloud-IoT integration [46].	WebAssembly runtime for resource constrained devices.
Introducing dynamic linking capabilities to the Wasm platform [52].	A dynamic linking mechanism that allows Wasm modules to be linked together at runtime (wasmtime).
Framework for developing and verifying cryptographic web applications using WebAssembly [70].	Verification and implementation of cryptographic protocols in Wasm.
Analysis of Wasm vs native in the spec cpu benchmarks [29].	Performance measure of applications compiled to Wasm in firefox and chrome.
Use of Wasm runtimes in serverless edge computing [16].	A prototype for Wasm execution using the Fastly edge cloud platform and the wasmtime WebAssembly runtime.
Design of an OS to run Wasm modules natively [95].	Implementing a lightweight OS kernel that can directly run Wasm modules in edge devices.

[42] extended Kubernetes to natively orchestrate WebAssembly modules alongside containers, demonstrating advantages for frequently executed serverless functions. [9] discussed Kubernetes enhancements, including WebAssembly integration as a complementary technology to containers.

In edge computing contexts, [66] benchmarked WebAssembly-based serverless platforms and confirmed their suitability for constrained edge devices. [61] utilized WebAssembly for retrofitting industrial environments, enabling consistent development from cloud to edge. [79] proposed a WebAssembly container platform for extreme edge computing that works across diverse device architectures.

For eBPF deployment in cloud environments, [100] introduced Wasm-bpf, which leverages WebAssembly to streamline eBPF deployment across heterogeneous cloud environments while maintaining cross-platform compatibility. In serverless computing, [82] presented Faasm, which uses WebAssembly for lightweight isolation in stateful serverless computing, achieving significant performance improvements over container-based approaches.

Recent work has also addressed WebAssembly runtime optimization and security. [30] proposed memory-efficient WebAssembly containers that reduce memory usage by 11-78% compared to existing runtimes. [98] explored resource isolation attacks in WebAssembly containers and discussed mitigation approaches. [72] developed Iris-Wasm for formal verification of WebAssembly programs, ensuring robust safety guarantees.

Additional research has examined WebAssembly's role in embedded systems and specialized computing scenarios. [60] evaluated WebAssembly for embedded systems, finding it delivers competitive performance compared to native execution and other virtual machines. [6] explored WebAssembly in high-performance computing contexts with MPIWasm, demonstrating competitive performance with significantly smaller binary sizes. [85] investigated composable network stacks using WebAssembly and QUIC, highlighting current performance limitations and potential improvements.

Although the WebAssembly ecosystem has evolved rapidly in recent years, comprehensive analyses covering standardized runtimes are still limited. Early works, such as by Jangda et al. [28], provided initial insights into WebAssembly's performance characteristics but were con-

strained to browser-based contexts, leaving standalone runtime performance largely unexplored. Similarly, Yan et al. [96] compared Wasm and JavaScript execution across major browsers like Chrome, Firefox, and Edge, showing that Wasm tends to use more memory due to its linear memory model lacking automatic garbage collection. Wang’s study [89] similarly focused on browser-based workloads, offering limited discussion on Wasm’s emerging use outside the web domain.

Further, Wen and Weber [95] introduced an operating system designed to run WebAssembly workloads in edge environments, comparing Wasm execution against native Linux performance. Their results demonstrated comparable or improved execution times, while maintaining Wasm’s inherent security benefits.

Building on these studies, Wang [90] expanded the analysis to include multiple standardized runtimes and proposed the WABench benchmark suite, combining a variety of established benchmarks and full applications. Their results indicated that JIT-based runtimes like Wasmtime, WAVM, and Wasmer outperform interpreter-based ones such as Wasm3 and WAMR. However, the study was limited to x86 architectures. In general, interpreter-based runtimes showed stable performance slowdowns across short and long benchmarks, while Cranelift JIT (used by Wasmtime) provided strong results on JetStream, MiBench, and PolyBench workloads. WAVM exhibited higher memory consumption, and Wasm runtimes overall used more memory than native execution for full applications.

Lehmann and Pradel [45] identified challenges in analyzing Wasm programs due to missing debugging symbols and complex control flow. Their proposed solution, *Wasabi*, introduced a dynamic analysis framework supporting tracing, profiling, and memory inspection. Spies and Mock [84] further explored Wasm performance and usability beyond browsers, concluding that Wasm holds promise for desktop, server, and IoT environments, particularly under resource constraints.

Lehmann et al. [44] assessed the exploitability of vulnerabilities in Wasm binaries compared to native programs, while Szewczyk et al. [87] evaluated bounds-checking safety mechanisms across multiple runtimes. Their findings revealed that modern bounds checking mechanisms allow runtimes to achieve performance within 20–35% of native execution across architectures.

Ménéntrey et al. [57] conducted one of the first studies to evaluate Wasm performance across two architectures, showing that Wasm can execute efficiently across diverse devices with manageable performance overheads. Using WAMR in AOT mode as their primary runtime, they concluded that Wasm’s lower cache miss rates occasionally offset its larger binary and register overheads. This work remains the only cross-architecture evaluation of WebAssembly to date. Building upon these insights, our analysis focuses on the modern Wasm runtimes most widely adopted in current research and development efforts.

Chapter 3

Methodology

Contents

3.1 Methodology Overview	42
3.2 Empirical Evaluation	43
3.3 Multi-Architecture Deployment	44
3.4 Container Runtime Integration	44
3.5 Conceptual Interface Exploration	45

This chapter outlines the overall methodological framework adopted in this thesis. While each subsequent chapter provides a detailed and self-contained description of its specific methodology, experimental setup, and evaluation procedure aligned with individual research questions, this chapter presents the unifying principles, design choices, and empirical strategies that guide the work as a whole.

WebAssembly is emerging as a unifying execution format for both client-side and server-side applications. In parallel, modern computing infrastructures increasingly rely on virtualization mechanisms such as hypervisors, containers, and orchestration platforms to abstract CPUs, operating systems, and cloud resources. These technologies have profoundly influenced software architecture, deployment models, and life-cycle management across cloud and edge environments. As a result, application developers increasingly require execution models that are portable, lightweight, and independent of underlying hardware and operating system details.

In this thesis, we adopt a mixed empirical and exploratory methodology to study WebAssembly as a portable, container-compatible, and extensible runtime across heterogeneous systems. Rather than focusing on a single platform or runtime configuration, we systematically evalu-

ate WebAssembly across multiple CPU architectures, execution environments, and deployment models. These empirical measurements are complemented by a conceptual exploration of system interface extensions that enable WebAssembly components to interact with host-managed resources.

The methodological contributions of this thesis can be summarized as follows:

1. We evaluate WebAssembly’s application-level virtualization by measuring portability and performance across diverse CPU architectures, cloud environments, and edge devices.
2. We establish a reproducible benchmarking methodology for comparing WebAssembly runtimes and native containers across heterogeneous execution environments.
3. We investigate WebAssembly’s integration with container ecosystems by comparing native and Wasm-based containers, including multi-architecture image creation and runtime behavior.
4. We extend the architectural scope of prior studies by including ARM Cortex-A78AE based edge hardware, providing insight into WebAssembly’s suitability for next-generation edge platforms.
5. We analyze container startup and image pull performance using lower-level container runtime abstractions in order to expose performance trade-offs that are often hidden by higher-level orchestration tools.
6. We explore and prototype experimental WASI component interfaces in Wasmtime, informed by existing proposals and accelerator-oriented programming models.

Together, these methods allow us to assess WebAssembly not only as a compilation target, but as a practical runtime substrate spanning cloud, edge, and emerging host-accelerated environments.

3.1 Methodology Overview

This thesis follows a comparative and measurement-driven methodology. Across all experiments, WebAssembly-based deployments are evaluated

alongside native container-based equivalents. This approach allows us to identify both quantitative performance differences and qualitative distinctions in deployment complexity, startup latency, isolation properties, and portability.

Empirical evaluation is used to measure runtime characteristics such as execution time, startup latency, and image pull overhead. To ensure meaningful comparisons, experiments are conducted under controlled conditions with consistent runtime versions, compiler settings, and system configurations. Measurements are repeated multiple times to account for variability, and reported results reflect averaged values unless otherwise stated.

In parallel, an exploratory methodology is applied to system interface design. Rather than proposing finalized standards, we prototype and evaluate experimental interfaces to assess feasibility and design trade-offs. This dual approach enables us to ground conceptual interface exploration in empirical observations from real systems.

3.2 Empirical Evaluation

Empirical evaluation forms the core of this thesis. We systematically benchmark WebAssembly runtimes and native containers across multiple hardware architectures and workloads. The objective is not only to measure raw performance, but also to understand runtime overheads, startup behavior, and the impact of different execution models on application deployment.

We employ established benchmark suites, including SightGlass and WaBench, to ensure representative workloads. These benchmarks cover compute-bound tasks, I/O-intensive operations, and runtime initialization behavior. Where necessary, benchmarks are instrumented to enable fine-grained timing measurements, including image pull latency, container startup time, and application execution time.

To ensure reproducibility, all experiments follow standardized procedures. Runtime versions, compiler optimizations, and system configurations are documented and kept consistent across platforms. Each benchmark is executed multiple times, and transient system effects are minimized by isolating workloads and controlling background activity.

3.3 Multi-Architecture Deployment

To study WebAssembly across the cloud–edge continuum, we select hardware platforms that represent diverse architectural characteristics and deployment contexts. These platforms enable us to analyze performance trade-offs between powerful cloud servers and resource-constrained edge devices.

Cloud experiments are conducted on virtual machines hosted on commercial cloud infrastructure, including x86_64 servers based on Intel Xeon processors and ARM64 servers based on ARM Neoverse architectures. These systems represent common deployment targets for modern cloud-native applications.

Edge experiments are performed on embedded and single-board systems, including NVIDIA Jetson platforms and RISC-V-based development boards. These devices reflect emerging trends in edge computing, where energy efficiency, fast startup, and portability are critical constraints.

By maintaining comparable software stacks across these platforms, we isolate architectural effects and assess how WebAssembly runtimes scale across heterogeneous hardware.

3.4 Container Runtime Integration

A significant portion of this thesis focuses on the interaction between WebAssembly and container ecosystems. Containers have become the dominant deployment abstraction for cloud-native applications, and understanding how WebAssembly integrates with container runtimes is essential for practical adoption.

For native containers, we use standard Docker-based workflows, including multi-platform image builds and optimized multi-stage Dockerfiles. For WebAssembly containers, we integrate the Wasmtime runtime through runwasi, enabling WebAssembly modules to execute as first-class container workloads.

To obtain a more accurate view of runtime behavior, we conduct experiments using lower-level container runtime abstractions. Rather than relying solely on Docker’s high-level interfaces, we interact directly with

the underlying container runtime to measure image pull time, startup latency, and execution overhead. This approach reduces interference from orchestration layers and exposes performance characteristics that are otherwise obscured.

Custom measurement scripts are developed to automate experiments and ensure consistent data collection across platforms and runtimes.

3.5 Conceptual Interface Exploration

In addition to performance evaluation, this thesis includes an exploratory investigation of system interface extensions for WebAssembly. The goal is to assess how WebAssembly components can interact with host-managed resources while preserving portability and sandboxing guarantees.

This exploration is informed by an analysis of existing WASI proposals, such as `wasi-nn`, as well as accelerator-oriented programming models including CUDA. Using the WebAssembly Component Model and WebAssembly Interface Types (WIT), we prototype experimental interfaces for host-managed memory, compute offloading, and data-centric operations.

These interfaces are implemented as experimental components in Wasmtime and used to demonstrate handle-based interaction patterns between WebAssembly modules and host services. The intent is not to define new standards, but to identify promising design directions and highlight challenges that must be addressed in future WASI evolution.

While each chapter provides an in-depth discussion of methodological specifics, we offer a summary of the general approach in this chapter. By combining empirical benchmarking with exploratory interface prototyping, we evaluate WebAssembly as a portable and extensible runtime across heterogeneous systems. The following chapters apply this methodology to concrete experiments and case studies, building on the foundations established here.

Chapter 4

Cross-Architecture Evaluation of Portability in Cloud-Edge

Contents

4.1 Introduction	46
4.2 Background	51
4.3 Related Work	54
4.4 Methodology	57
4.4.1 Performance Metrics Utilized	58
4.4.2 Experimental Setup	60
4.4.3 WebAssembly Runtimes Used	61
4.5 Results	63
4.5.1 Execution Speed	64
4.5.2 Startup Times	68
4.5.3 WAMR Execution Modes	70
4.5.4 Ahead-of-time compilation	71
4.5.5 Wasmtime: A High-Performance Runtime	72
4.5.6 Insights from WebAssembly Micro Runtime	73
4.5.7 Considerations for Runtime Selection	75
4.6 Discussion	75
4.6.1 Opportunities for Future Research	75
4.7 Conclusion	76

4.1 Introduction

The cloud computing landscape is undergoing a major transformation. Once confined to large centralized data centers, computing resources

are now distributed across a spectrum of environments that include regional servers, enterprise on-premise systems, mobile edge stations, and even personal and embedded devices that operate at the network periphery. This emerging paradigm, often referred to as the *cloud-edge continuum* [21], reflects a shift from centralized architectures to a fluid, multi-layered ecosystem where computation can occur anywhere along the path from the cloud to the end user. The resulting environment embodies a significant degree of diversity in hardware, performance capabilities, energy constraints, and network conditions, presenting both opportunities and challenges for modern computing.

This heterogeneity poses fundamental difficulties for software deployment and execution. Applications that were once designed to run on uniform cloud hardware must now be portable across a wide range of devices with distinct architectures, instruction sets, and resource characteristics. Developers must ensure that applications can execute consistently across these environments without extensive reconfiguration or specialized optimization for each platform. In addition to portability, efficiency is a crucial requirement. The growing dynamism of distributed computing means that workloads should be able to move between different nodes as conditions evolve, ensuring that services are executed where they can best meet user requirements while minimizing cost and energy use. The challenge of selecting the optimal execution site for a given service, known as service placement and rescheduling, continues to be an area of intense research [53, 62, 64]. The interplay of portability, efficiency, and adaptability thus defines the central tension in the design of systems that operate across the cloud-edge continuum.

To address these challenges, this thesis investigates the use of WebAssembly (Wasm) as a unifying software abstraction for heterogeneous environments. WebAssembly was originally introduced in 2017 as a portable, efficient bytecode format for web browsers. However, its design principles: near-native execution speed, rapid startup time, secure sandboxing, and platform neutrality—make it a strong candidate for execution beyond the browser. The growing adoption of Wasm in server-side and embedded contexts signals a broader trend toward using it as a universal runtime layer capable of supporting diverse workloads across architectures and operating systems. Figure 4.1 situates WebAssembly within the broader continuum of computing infrastructure, highlighting its potential to bridge the gap between cloud and edge systems.

WebAssembly’s promise lies in its ability to decouple software from the specifics of the underlying hardware, enabling the same application binary to execute on a variety of processor architectures, including x86, ARM, and RISC-V. This property aligns closely with the requirements of heterogeneous and distributed computing environments. Furthermore, the safety and determinism guaranteed by Wasm’s sandboxed execution model provide strong isolation between workloads, making it suitable for multi-tenant hosting environments in which workloads from different users or applications must be safely co-located. The ongoing development of the WebAssembly System Interface (WASI) extends Wasm’s reach even further, enabling access to system-level resources through capability-based interfaces while maintaining strong security guarantees. Together, these features make WebAssembly a promising candidate for sustainable and secure computing across the cloud-edge spectrum.

However, despite its growing popularity, WebAssembly’s performance characteristics in non-browser contexts remain an open area of study. A WebAssembly module cannot execute directly on hardware; instead, it requires a native runtime that interprets or compiles the bytecode for the host architecture. The design of this runtime, its compilation strategy, caching mechanisms, and system integration have a profound impact on both execution speed and startup latency. Understanding the trade-offs among different runtime implementations is essential for assessing Wasm’s suitability as a foundational technology for distributed and heterogeneous computing. As the demand for edge computing continues to grow, such an understanding becomes critical for ensuring efficient resource utilization and predictable performance across diverse hardware environments.

This thesis, therefore, undertakes a comprehensive and systematic performance evaluation of WebAssembly across multiple architectures and runtime environments. The study focuses on two leading runtimes developed by the Bytecode Alliance¹: Wasmtime² and WAMR³. Wasmtime is a full-featured and production-grade runtime designed for fast and secure execution of WebAssembly modules, whereas WAMR is a lightweight runtime optimized for resource-constrained environments such as embedded and edge devices. Together, these two systems represent distinct design philosophies within the WebAssembly ecosystem,

¹<https://bytecodealliance.org/>

²<https://github.com/bytecodealliance/wasmtime>

³<https://github.com/bytecodealliance/wasm-micro-runtime>

and their comparison provides meaningful insights into performance, scalability, and resource efficiency across varying deployment contexts.

The evaluation presented in this work examines critical performance dimensions, including execution throughput, startup latency, memory footprint, and the relative efficiency of different compilation modes. By conducting experiments across x86_64, ARM64, and RISC-V architectures, the study provides one of the most comprehensive cross-platform analyses of WebAssembly runtimes to date. Beyond quantitative measurements, the results are interpreted in the broader context of system design and cloud-edge deployment strategies, offering guidance for developers and researchers seeking to leverage Wasm in distributed environments.

In addition to performance characterization, this thesis contributes to the advancement of WebAssembly as an execution environment by extending the WebAssembly System Interface (WASI) through novel experimental services. These services introduce abstractions for host-managed memory and accelerator integration, enabling efficient offloading of computational workloads such as matrix multiplication and data analytics operations. Implemented in Wasmtime, these extensions demonstrate how WebAssembly can be used to interact with external accelerators, such as GPUs or specialized data engines, through safe and portable interfaces. The design and evaluation of these prototypes showcase the potential of Wasm as a foundation for performance-aware and data-centric computing across heterogeneous systems.

Ultimately, this thesis argues that WebAssembly represents a pivotal step toward achieving a truly portable, secure, and sustainable execution model for the cloud-edge continuum. By providing both theoretical insights and empirical validation, it seeks to bridge the current gap between abstraction and performance in heterogeneous computing. The results not only deepen our understanding of WebAssembly’s capabilities and limitations but also outline a practical pathway for future systems that can seamlessly adapt to the evolving landscape of distributed and accelerator-enabled computation.

Main Contributions

In particular, this chapter makes the following contributions:

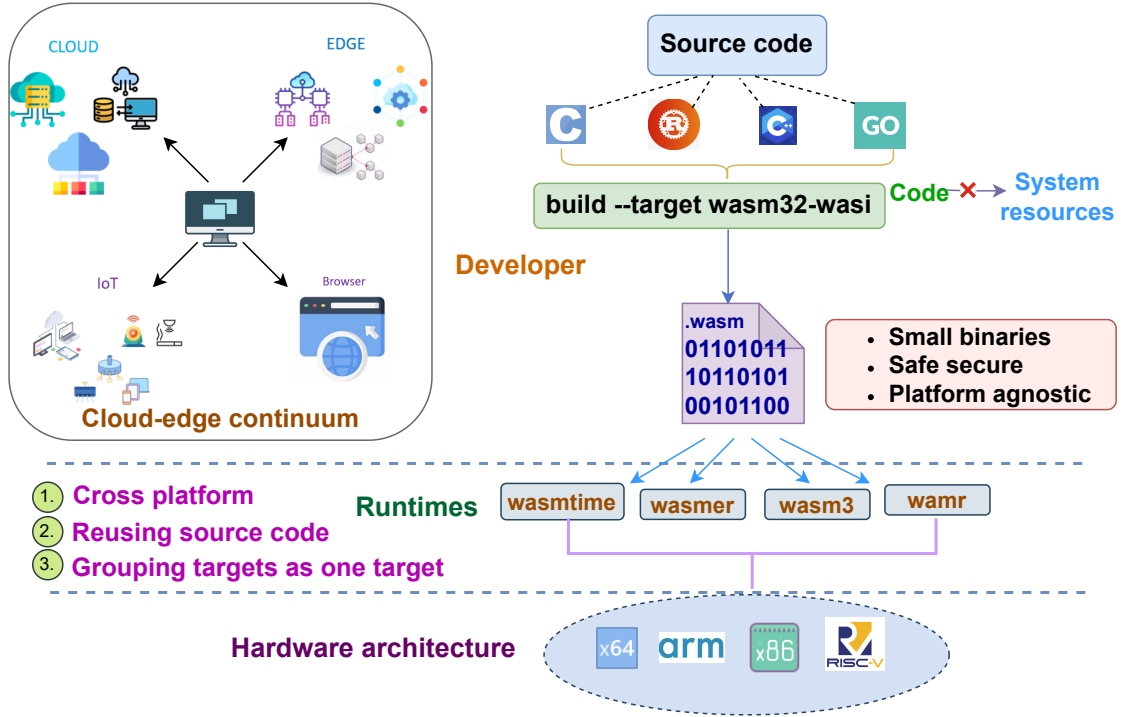


Figure 4.1: WebAssembly and its use in the cloud-edge continuum.

- A central innovation of this research lies in its methodological framework. We conduct, to the best of our knowledge, the first systematic cross-architecture performance evaluation of WebAssembly benchmarks on two state-of-the-art runtimes, Wasmtime and WAMR. Our evaluation spans four distinct hardware platforms, encompassing two server systems based on X86_64 and ARM64 architectures, as well as two embedded platforms built on ARM64 and RISC-V64. This diversity ensures a broad and representative perspective on WebAssembly's behavior across heterogeneous computing environments.
- The work also introduces methodological enhancements that go beyond the existing literature. Through detailed instrumentation of both runtimes and benchmarks, we achieve a level of measurement granularity that allows for precise capture of startup times and execution durations within well-defined program segments, which we refer to as *specific regions of interest*. This refined approach provides a foundation for a deeper and more accurate understanding of runtime behavior than has been possible in prior studies.
- By combining this novel methodology with an in-depth performance analysis, our study advances the state of the art in We-

bAssembly benchmarking. In particular, we have contributed a carefully calibrated and instrumented suite of benchmarks to the existing WaBench framework. This contribution extends the scope of future research by establishing reproducible measurement procedures and validated workloads, enabling more consistent and reliable comparisons across runtime environments and architectures.

4.2 Background

WebAssembly, commonly abbreviated as Wasm, is a low-level, binary instruction format that serves as a compilation target for high-level programming languages. It was originally designed to enable near-native performance for applications running within web browsers, but its simplicity, efficiency, and portability have led to its rapid adoption far beyond its original context. In essence, a WebAssembly *module* is a compact and self-contained unit of computation that encapsulates functions, global variables, and a linear memory space. The functions within a module are expressed as stack-based bytecode instructions for a virtual machine, which can be interpreted or compiled to native machine code by the host runtime.

A WebAssembly runtime acts as the execution engine responsible for managing the lifecycle of modules. It can either interpret the bytecode directly or employ Just-In-Time (JIT) or Ahead-Of-Time (AOT) compilation to translate the bytecode into native instructions optimized for the host architecture. This architecture-agnostic design is central to WebAssembly's appeal, as it enables a single module to be executed on multiple processor types without recompilation. The runtime also manages the boundaries between the sandboxed module and the host environment through a structured import and export mechanism. This mechanism allows a module to call host-provided functions or interact with external systems, while maintaining strict control over access to resources.

Crucially, WebAssembly does not provide direct access to system-level operations such as file I/O or network communication. Instead, it relies on the host environment to expose these capabilities through explicitly defined imports. The module itself remains isolated, executing within a controlled and secure memory space. This isolation is enforced

through the concept of *linear memory*, a contiguous block of addressable bytes accessible to both the host and the module, which serves as the primary channel for data exchange. This design guarantees deterministic behavior and prevents arbitrary memory access, a key property for secure multi-tenant execution.

To enable broader system integration, the WebAssembly System Interface (WASI)⁴ was developed as a standardized API for interacting with the underlying host in a portable and secure way. WebAssembly in the browser can access system resources through the JavaScript API (Web APIs, WASI-like calls via JS glue code). However, this approach carries significant overhead because all system calls are mediated through the JavaScript engine. For non-browser (edge, server, IoT) contexts, importing the full JavaScript API is inappropriate: it is too heavy, introduces JS engine dependency, and exposes browser-specific primitives that have no meaning on a server or embedded device. WASI was created to provide a minimal, POSIX-inspired, capability-based API that gives Wasm modules direct access to host system resources without the JS intermediary overhead, making Wasm practical for edge and server runtimes. This is why WASI is described as the key enabler for Wasm outside the browser, and why its design philosophy of minimalism and capability-based security is different from browser Web APIs.

Wasm supports a growing number of source languages, with Rust, C, C++, and Go being the most mature options for production use⁵. The Rust and Go ecosystems include native backends that compile directly to WebAssembly, while C and C++ developers typically use LLVM-based toolchains such as Clang, Emscripten, and Binaryen. The use of LLVM as a common compilation foundation has facilitated rapid expansion of WebAssembly support across multiple programming languages and toolchains. This flexibility allows developers to leverage existing codebases and expertise while targeting a portable execution layer.

Beyond language support, WebAssembly's architecture aligns naturally with the evolving landscape of distributed and heterogeneous computing. As computational workloads become more dynamic and distributed across the cloud-edge continuum, there is a growing need for lightweight, portable binaries that can execute consistently across a range of devices. Traditional approaches to portability often involve

⁴<https://wasi.dev/>

⁵<https://github.com/appcypher/awesome-wasm-langs>

recompilation or dependency management tailored to each platform, which introduces complexity and maintenance overhead. WebAssembly addresses this problem by abstracting away architecture-specific details, allowing developers to deploy the same binary module across CPUs, microcontrollers, and other processing units with minimal modification.

In the context of service placement, real-time computation, and multi-objective optimization, the heterogeneity of modern systems poses profound challenges. Today’s computing environments incorporate not only CPUs with diverse architectures such as X86, ARM, and RISC-V, but also specialized accelerators, including GPUs, TPUs, and FPGAs. These devices differ widely in their instruction sets, performance characteristics, and memory hierarchies. Developing algorithms and scheduling mechanisms that can efficiently utilize such a diverse ecosystem requires a unified execution model capable of adapting to each device’s strengths while maintaining consistent semantics.

WebAssembly offers an attractive foundation for such unification. Its architecture-independent binary format and runtime-based execution model make it inherently portable across hardware platforms. Moreover, the modular nature of WebAssembly applications allows them to be treated as schedulable units within a distributed environment. A WebAssembly module can be instantiated, executed, and migrated across nodes with minimal dependency on the underlying system, a property that aligns closely with the goals of dynamic service placement and adaptive computing. As a result, WebAssembly holds significant potential for facilitating efficient resource allocation and real-time responsiveness in heterogeneous systems.

Nevertheless, realizing this potential requires a thorough understanding of WebAssembly’s runtime performance across different architectures. Questions arise about the degree to which WebAssembly can maintain predictable execution times, manage startup latency, and deliver efficient memory usage across CPUs and accelerators. Addressing these questions is crucial for determining whether WebAssembly can serve as a practical substrate for multi-objective optimization frameworks, where trade-offs among performance, energy consumption, and portability must be carefully balanced.

The research presented in this thesis seeks to shed light on these questions by examining WebAssembly’s capabilities methodically and empirically. By conducting cross-platform experiments and developing

new methodologies for runtime instrumentation and measurement, we aim to contribute a rigorous understanding of WebAssembly’s role in modern computing ecosystems. This work not only assesses the feasibility of using WebAssembly as a portable execution layer for distributed systems but also explores its potential for enabling sustainable and efficient computation across the diverse architectures that constitute the cloud-edge continuum.

4.3 Related Work

This section offers a detailed synthesis of the literature most relevant to the present study. WebAssembly has rapidly evolved as a portable runtime technology capable of operating across heterogeneous hardware and software environments. A number of execution environments have been created to support this portability, including Wasmtime, WAMR, Wasmer⁶, and Wasm3, among others. These platforms allow the execution of Wasm programs across diverse systems, promoting flexibility and interoperability.

Recent scholarship has focused on the challenges of maintaining quality of service and managing limitations associated with hardware and network configurations. The empirical study by Hilbig et al. [25] provides an updated perspective on the adoption of WebAssembly in practical contexts. Their findings reveal a dynamic and growing ecosystem while also identifying security risks that stem from unsafe source languages.

In another contribution, Chadha et al. [7] introduced MPIWasm, an approach that adapts WebAssembly for distributed high-performance computing applications based on MPI. Their experiments demonstrate that Wasm can achieve competitive runtime performance compared to native applications while producing much smaller binaries. This result highlights the potential of WebAssembly as a compact distribution format for computation intensive workloads.

Comparative analyses of commonly used WebAssembly runtimes remain relatively limited. The works in [28], [96], and [89] investigate the performance differences between applications compiled to Wasm and those written in JavaScript. These investigations, however, mainly em-

⁶<https://github.com/wasmerio/wasmer>

phasize browser-based applications, leaving open questions about performance in broader non-web settings.

Bosshard et al. [4] examined WebAssembly within serverless contexts, comparing the behavior of several runtime implementations, including Wasmer, Wasmtime, and Lucet, in platforms such as Openwhisk and AWS Lambda. Likewise, the studies by Spies et al. [84] and Wang et al. [91] conducted benchmark-based evaluations using PolyBench. While this suite offers a convenient set of compiler benchmarks, it does not fully capture the diversity of workloads found in general-purpose or non-browser environments. Another influential work by Watt et al. [94] formalized the type system of WebAssembly and proved its soundness through a verified type checker and interpreter.

Hockley et al. [26] assessed WebAssembly as a sandboxed scripting environment for general computation. They compared execution times between Wasm and native binaries using both micro and macro benchmarks. Their evaluation, although rigorous, was limited to specific hardware setups and thus may not reflect performance trends across a wider spectrum of devices and architectures.

In the context of serverless computing, Kjorveziroski et al. [41] conducted a comprehensive evaluation comparing WebAssembly-based runtimes with traditional container runtimes. Their findings indicate that Wasm consistently achieved superior results, with Wasmtime emerging as the fastest among the tested engines.

Expanding the scope, Wang et al. [90] addressed the lack of broad comparative studies by analyzing five major Wasm runtimes using a newly developed benchmark suite, WaBench. This suite integrates components from well-established benchmark collections as well as full-scale applications from multiple domains. Their results shed light on the performance characteristics of standalone WebAssembly runtimes and the influence of JIT and AOT compilation techniques on efficiency.

Performance aspects were further explored by Jangda et al. [29], who utilized SPEC CPU benchmarks to assess WebAssembly execution in Chrome and Firefox. Complementary to this, Gadepalli et al. [18] proposed Sledge, a WebAssembly-based framework tailored for serverless workloads at the edge. Their design offered enhanced scheduling, workload distribution, and efficient isolation mechanisms through software fault isolation, enabling a lightweight yet secure runtime model.

Security-oriented studies also form a critical part of WebAssembly research. Stievenart et al. [86] examined the security implications of compiling C programs into Wasm, focusing on buffer overflow vulnerabilities. They observed that some generated Wasm binaries lack protective mechanisms such as stack canaries, which can allow programs to continue execution even after memory corruption, unlike x86 binaries that typically crash. This observation underscores the need for stronger compiler-level safeguards.

Lehmann et al. [45] developed Wasabi, a framework for analyzing WebAssembly modules, particularly for security auditing and dynamic behavior tracking. Along similar lines, Ménétrey et al. [55] introduced TWINE, a trusted WebAssembly runtime evaluated through standard benchmarks and practical applications such as secure SQLite. Although TWINE introduces some computational overhead, its added security and full standard compatibility make it a robust alternative to unverified runtimes.

The work by Li et al. [46] investigated the feasibility of deploying WebAssembly in cloud-to-IoT environments, demonstrating how Wasm runtimes can operate efficiently even on resource constrained devices. Breitfelder et al. [5] addressed the growing security challenges in back-end deployments by proposing WasmA, a static analysis framework for Wasm that supports flexible and efficient program inspection. Their experiments demonstrated its strong balance between extensibility and performance, establishing it as a practical tool for large-scale analysis.

Further studies by Lehmann et al. [44] investigated the security implications of linear memory in Wasm binaries relative to native code. Gadepalli et al. [19] proposed aWasm, a serverless framework designed for edge and IoT computing. Their implementation achieved near-native speed, a compact memory footprint, and rapid function startup times, demonstrating Wasm's suitability for distributed workloads with low latency requirements.

Ménétrey et al. [57] expanded this investigation by evaluating Wasm performance across two distinct processor architectures. Their results verified the capability of WebAssembly to achieve comparable computational results without significant performance degradation. Gackstatter et al. [16] also explored Wasm deployment in edge environments, presenting a prototype integrated with the Fastly edge cloud infrastructure.

Wen et al. [95] proposed a lightweight operating system architecture capable of executing Wasm modules natively, targeting embedded and edge devices. Kakati et al. [34] presented a comprehensive survey highlighting WebAssembly’s transformative role in the shift from centralized cloud computing toward distributed edge computing. Their study emphasizes the importance of interoperability and platform neutrality in this transition.

Finally, Wallentowitz et al. [88] examined the potential of WebAssembly as a virtual machine for embedded systems. Their paper discusses the benefits of its strong ecosystem and outlines the technical challenges of leveraging it efficiently in constrained environments. They further demonstrate its viability through a real-world case study and identify key research directions for future investigation.

While the body of related work is extensive, it remains evident that systematic analyses of WebAssembly runtimes across multiple architectures and broader benchmark sets are still scarce. This motivates the present research, which aims to bridge this gap through a more comprehensive and cross-platform evaluation of runtime behavior and performance characteristics.

4.4 Methodology

The primary objective of this study is to examine the performance behavior of WebAssembly applications when executed across multiple hardware architectures and runtime systems. For this purpose, we employed the WaBench benchmark suite developed by Wang et al. [90]. To ensure precise measurement of the key execution intervals, we instrumented the benchmarks with timestamp-based markers that capture the duration of a specified *region of interest*. This approach allows us to isolate the steady-state execution phase, excluding initialization and setup processes. Furthermore, both the benchmarks and the corresponding runtimes were instrumented to accurately determine startup latency. We define startup time as the interval beginning when the Linux shell command is issued to invoke the benchmark and ending at the exact point when the `main` function begins execution. The instrumented versions of these benchmarks are publicly available on GitHub⁷.

⁷<https://github.com/matsbror/wabench>

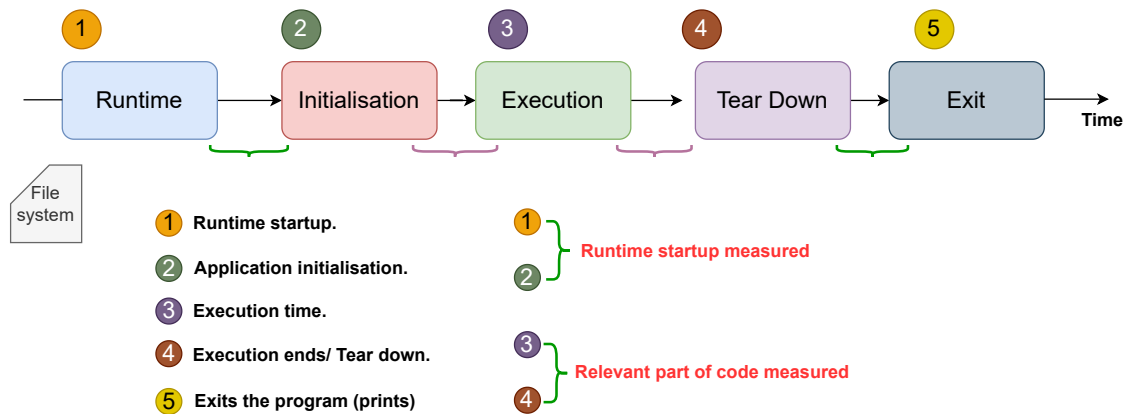


Figure 4.2: Timestamps and regions of interest.

To account for fluctuations in system load and the varying availability of computational resources, we repeated each benchmark multiple times. The number of repetitions was dynamically adjusted depending on the execution duration of each test to ensure statistical reliability. The reported performance values represent the median of all recorded results, which helps reduce the influence of outliers and provides a more robust indicator of central tendency. To maintain consistency and validity, all experiments were performed on an isolated machine environment, free from background tasks or interfering processes.

In this study, we chose not to employ CPU pinning or fixed clock frequencies. Preliminary testing indicated that these controls did not produce noticeable differences in measured performance, making their inclusion unnecessary for this analysis.

The WaBench suite contains more than fifty benchmarks, all originally implemented in C or C++. They were compiled using the `clang/clang++` compiler version 16 with the `-O2` optimization flag and the WASI SDK version 20. For clarity and interpretability, we selected six representative benchmarks that best capture performance variations across runtime environments and processor architectures. Comprehensive results for the entire benchmark set are available online⁸.

4.4.1 Performance Metrics Utilized

The execution of a WebAssembly benchmark can be logically separated into several distinct phases, illustrated in Figure 4.3. The first is the

⁸https://github.com/sangeeta1998/wasm_execution

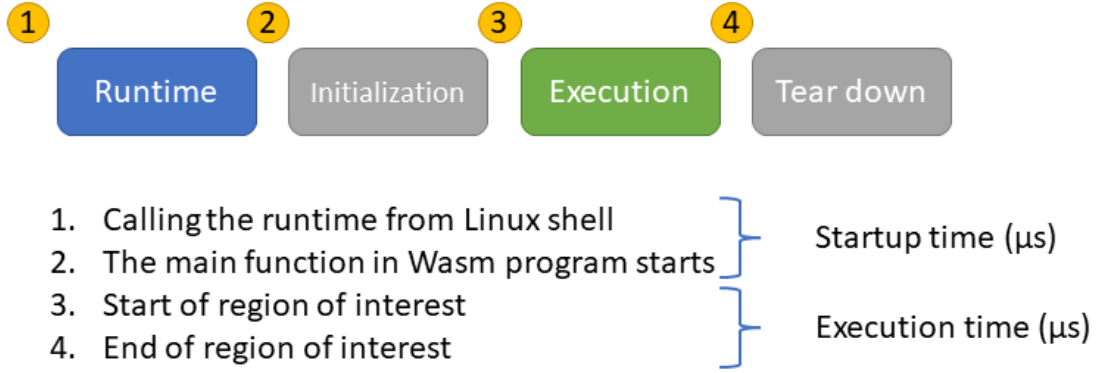


Figure 4.3: Representation of Startup Time and Execution Time.

runtime phase, which spans from the invocation of the runtime process in the Linux shell to the moment the `main` function of the benchmark begins execution, Figure 4.2. This interval encompasses all preparatory activities, including process creation by the operating system, loading of the WebAssembly file from disk, instantiation of the module, potential compilation from WebAssembly bytecode to native machine code (when using JIT or AOT modes rather than interpretation), and the start of actual code execution.

Following this is the *initialization phase*, which represents the setup of internal data structures within the benchmark. Because this segment is primarily concerned with configuration rather than active computation, it is excluded from our timing measurements.

The *execution phase* constitutes our region of interest and forms the basis of our performance evaluation. It corresponds to the section of the benchmark that performs the primary computation workload. The final phase, referred to as the *tear down phase*, includes result generation and output handling after computation concludes. This phase is similarly omitted from the performance metrics, as it does not reflect computational performance.

To facilitate precise measurement, we introduced four timestamps into the code: ts_1 marks the invocation of the runtime from the shell, ts_2 corresponds to the beginning of the `main` function, ts_3 captures the start of the region of interest, and ts_4 denotes its end. These timestamps are recorded with microsecond precision. The difference between ts_2 and ts_1 represents the startup time, while the interval between ts_4 and ts_3 defines the execution time.

Since the experiments span multiple hardware architectures and platforms with differing performance characteristics, we also executed all benchmarks natively on each platform. This enables direct comparison of both absolute and relative performance metrics, particularly with respect to startup and execution times. The combination of native and WebAssembly-based results allows for a consistent evaluation framework across heterogeneous computing environments.

4.4.2 Experimental Setup

Table 4.1: Hardware architectures used in experiments.

Platform	CPU Model	Clock (GHz)	Threads/Core	Cores/Socket	Caches
ARM AWS c6g.metal	Neoverse-N1	3.0	1	64	L1d: 64 KiB, L1i: 64 KiB, L2: 1 MiB, L3: 32 MiB
x86_64 AWS c5.metal	Intel Xeon 8275CL	3.0	2	24	L1d: 32 KiB, L1i: 32 KiB, L2: 1 MiB, L3: 71.5 MiB
NVIDIA Jetson Nano	ARM Cortex-A57	1.5	2	4	L1d: 32 KiB, L1i: 48 KiB, L2: 2 MiB
StarFive VisionFive2	SiFive U74 Core	1.0	1	4	L1d: 32 KiB, L1i: 32 KiB, L2: 2 MiB

To better illustrate the embedded platforms used in this study, Figures 4.4 and 4.5 show the physical setups of the NVIDIA Jetson Nano and StarFive VisionFive2 development boards, respectively. These devices were used to represent low-power edge computing environments in our experiments.

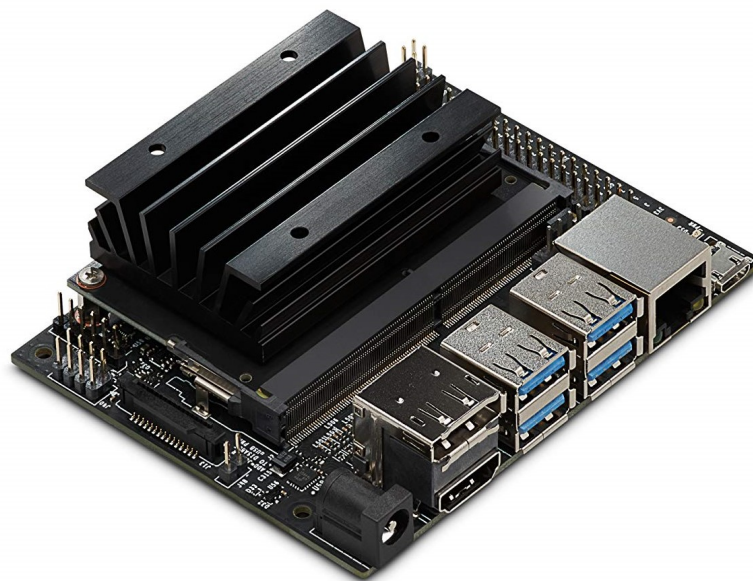


Figure 4.4: NVIDIA Jetson Nano embedded board used for experiments.



Figure 4.5: StarFive VisionFive2 RISC-V board used for experiments.

The AWS instances represent the category of powerful cloud data-center servers. Both the Intel Xeon 8275CL and the Neoverse-N1 CPUs are high-performance processors equipped with advanced microarchitectural features, including dynamic instruction scheduling, speculative execution with branch prediction, and sophisticated cache prefetching mechanisms. These attributes enable high throughput and efficient parallel execution. In contrast, the Nvidia Jetson and StarFive VisionFive2 boards belong to the embedded computing domain, typically deployed closer to the network edge. The processors used in these devices operate with more limited computational resources. They run at lower clock frequencies and have smaller caches, which impacts both latency and throughput. Nevertheless, the Nvidia Jetson, which employs the ARM Cortex-A57 processor, is expected to outperform the VisionFive2 board due to its four-way superscalar out-of-order design and approximately fifty percent higher clock rate.

4.4.3 WebAssembly Runtimes Used

A growing number of WebAssembly runtimes are now available to execute Wasm applications beyond the browser environment. Among the prominent options are Wasmer, WasmEdge, Wasm3, and Wazero, in addition to two widely adopted runtimes maintained under the Bytecode Alliance: Wasmtime and WAMR (WebAssembly Micro Runtime). After preliminary evaluation, we selected **Wasmtime** and **WAMR** for our experiments because they consistently demonstrated stable perfor-

mance across all our target platforms. Moreover, both runtimes play an active role in the WebAssembly standardization community, ensuring close alignment with ongoing specifications and ecosystem development.

4.4.3.1 Runtime 1: Wasmtime

Wasmtime is a WebAssembly runtime developed in the Rust programming language, designed to achieve a balance between performance, compliance, and security. It employs the Cranelift compiler backend for just-in-time translation of WebAssembly bytecode into native machine code. Cranelift, also a project of the Bytecode Alliance, provides an efficient compilation pipeline that prioritizes low-latency code generation while preserving correctness and safety. Wasmtime supports the WebAssembly System Interface (WASI) standard, enabling programs to perform system-level interactions in a platform-independent manner.

The runtime benefits from Rust’s strict safety guarantees, which prevent common programming errors and enhance memory safety. Additionally, Wasmtime adheres to a rigorous feature integration process, emphasizing maintainability and security assurance. In our experiments, Wasmtime was compiled from a fork of the official repository (dated 15 November 2023) using the default configuration command: `cargo build --release`. The generated executable had an approximate size of 37 MB. This size can be reduced further by disabling optional components and diagnostic logging, as illustrated in the official documentation⁹.

4.4.3.2 Runtime 2: WAMR (WebAssembly Micro Runtime)

The second runtime utilized in our study is the WebAssembly Micro Runtime (WAMR). WAMR is designed with modularity as a core principle, enabling it to be configured for various computing environments ranging from resource-constrained IoT devices to high-performance servers. It is implemented in C and emphasizes a compact footprint while maintaining compliance with the WebAssembly and WASI specifications.

For this work, we built and evaluated four variants of WAMR, each representing a different trade-off between performance and binary size. These variants are briefly described below.

⁹<https://docs.wasmtime.dev/examples-minimal.html>

Interpreter: This configuration executes WebAssembly bytecode directly without compilation, making it suitable for low-power embedded or IoT devices where memory and storage are highly constrained. The compiled executable size is approximately 293 kB.

LLVM JIT: This mode employs the LLVM compiler infrastructure to generate native code just before execution. It provides high-quality optimized code at the cost of increased memory consumption. The resulting executable has a size of approximately 50 MB. This configuration was used as the default for our experiments since its performance characteristics are comparable to those of Wasmtime.

Fast JIT: A lightweight just-in-time compiler that prioritizes compilation speed over code optimization. While it generates less optimized code than LLVM JIT, it significantly reduces startup latency. The executable size for this variant is around 663 kB.

Multi-tier JIT: This configuration begins execution with the Fast JIT compiler to minimize startup delays while concurrently invoking additional threads that compile the same bytecode using the LLVM JIT backend. Once the higher-quality LLVM-compiled code is ready, it replaces the initially generated code. The executable size for this variant is approximately 50 MB.

The WAMR executable is named *iwasm*, and throughout this thesis, the terms *WAMR* and *iwasm* are used interchangeably in the results and analysis sections.

4.5 Results

The organization of our results follows a structured approach. We first examine execution performance within the region of interest and compare it against native execution on each hardware platform. This allows us to isolate runtime efficiency and disregard the overhead introduced by just-in-time compilation or module instantiation. Subsequently, we evaluate startup latency, which includes these initialization costs. We then conduct a focused analysis of the different WAMR execution modes described previously and finally examine the performance impact of ahead-

of-time (AOT) compilation of WebAssembly modules in comparison to interpreted and JIT-executed modes.

4.5.1 Execution Speed

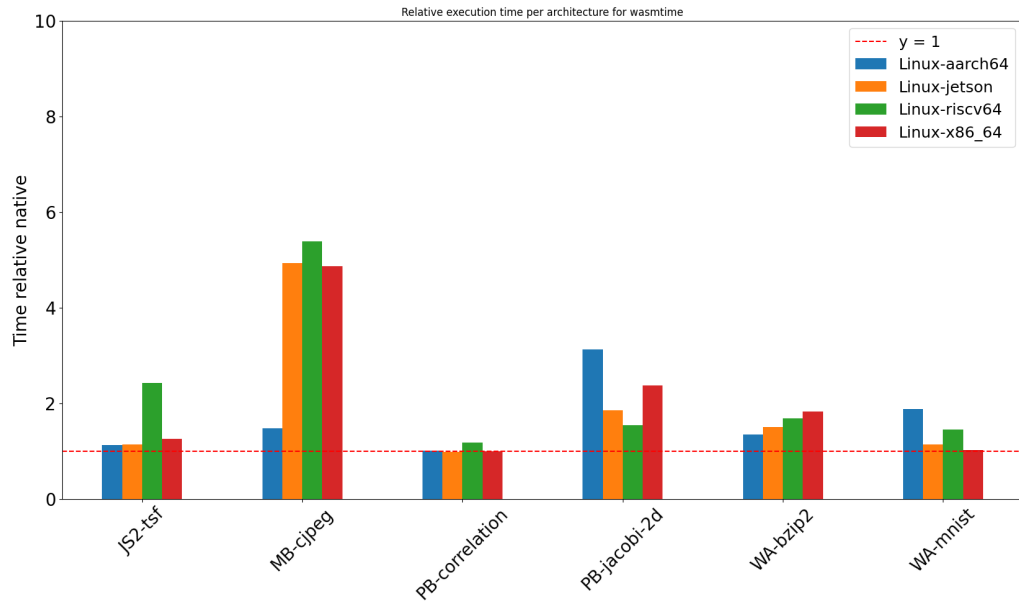
Figure 4.6 presents the relative execution time for six selected benchmarks compared to native execution across all studied platforms. Subfigure (a) corresponds to the Wasmtime runtime, while subfigure (b) depicts the performance of WAMR. Each group of bars in the figure represents results for a specific platform, ordered from left to right as follows: ARM64 on AWS (*aarch64*), Nvidia Jetson (*jetson*), RISC-V on VisionFive2 (*riscv64*), and x86_64 on AWS (*x86_64*).

Several important observations can be drawn from Figure 4.6. For a substantial portion of the evaluated benchmarks, the execution time achieved by both Wasmtime and WAMR across multiple hardware platforms is comparable to native performance. Specifically, among the 51 benchmarks executed, more than fifty percent of them exhibit execution times within a factor of two relative to native performance when using Wasmtime, and nearly half demonstrate a similar ratio when executed with WAMR. These results provide strong evidence that, from a performance standpoint, WebAssembly constitutes a viable execution medium for cloud-based workloads while maintaining cross-architecture portability.

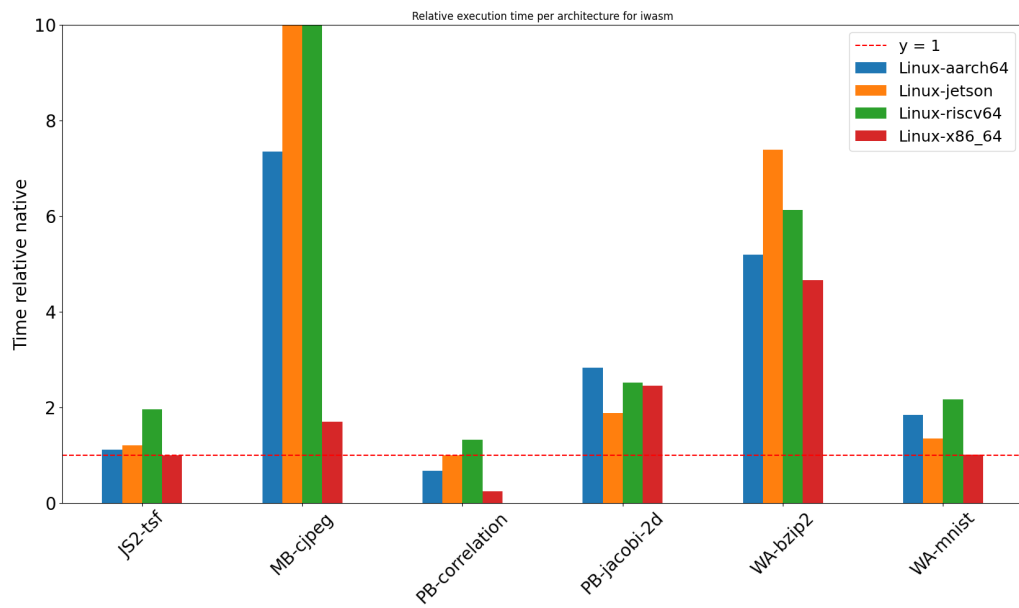
Furthermore, Table 4.2 highlights the efficiency of WebAssembly modules in terms of binary size. The compiled Wasm binaries occupy only a small fraction of the storage space required by their corresponding native executables, underscoring WebAssembly’s potential as a lightweight and resource-efficient deployment format suitable for heterogeneous computing environments.

Figure 4.7 reveals substantial variations in execution times across both runtimes and hardware platforms (note the logarithmic scale on the y-axis). As expected, the embedded systems Jetson and riscv64 exhibit considerably slower performance, particularly the riscv64 platform, which not only operates at a lower clock frequency but also relies on a less sophisticated processor design.

Except for the benchmarks MB-cjpeg and WA-bzip2, there are no major discrepancies in performance between the two runtimes. This is

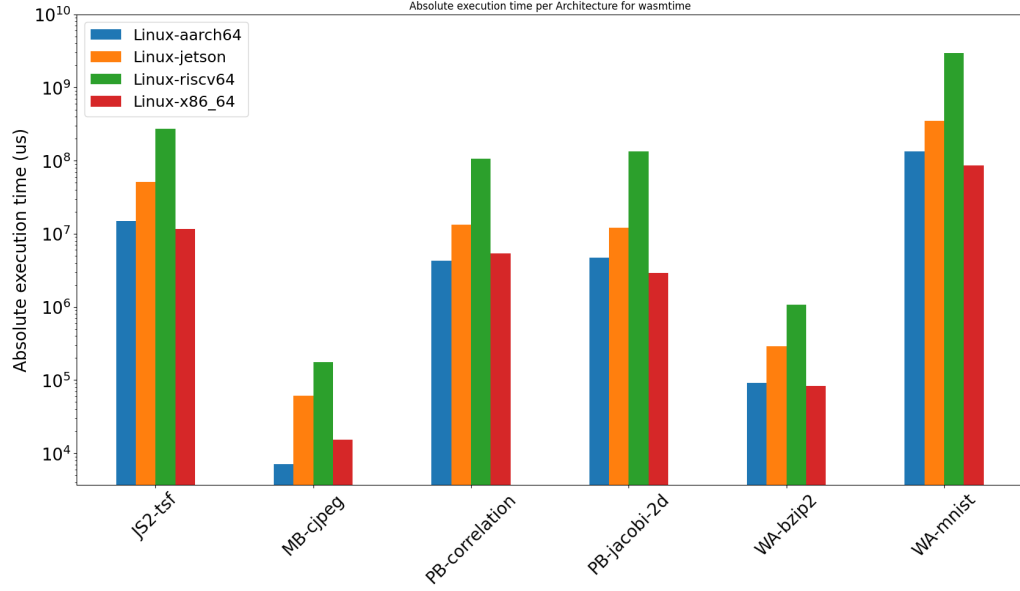


(a) Wasmtime

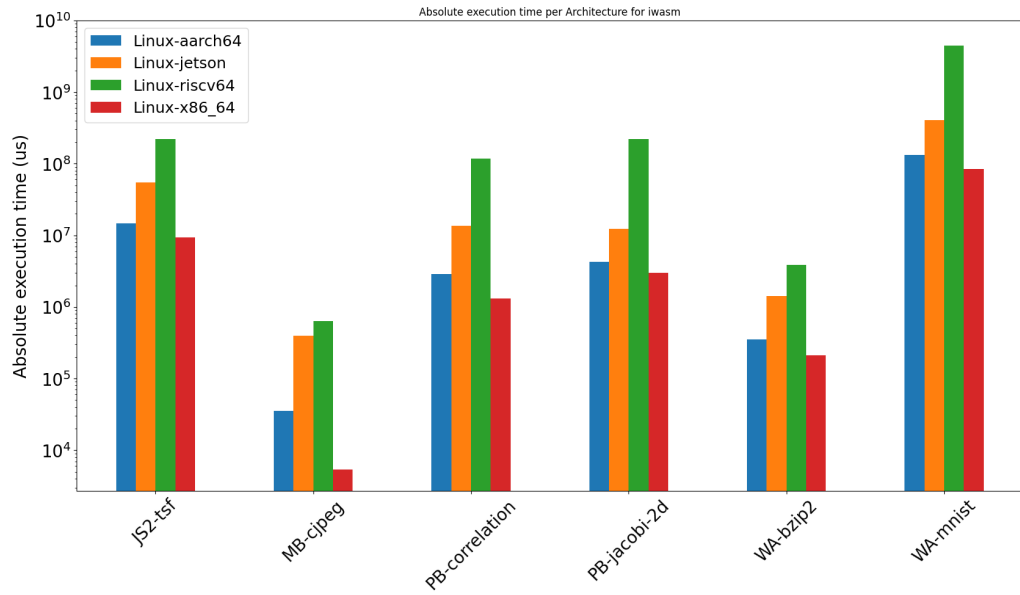


(b) WAMR (LLVM JIT)

Figure 4.6: Relative execution time per architecture.



(a) Wasmtime



(b) WAMR (LLVM JIT)

Figure 4.7: Absolute execution time per architecture.

Table 4.2: Size of WebAssembly modules compared to statically linked native executables.

Benchmark	Native size (X86_64)	Wasm size
JS2-tsf	1.5 MB	414 kB
MB-cjpeg	1.2 MB	152 kB
PB-3mm	897 kB	48 kB
PB-correlation	897 kB	47 kB
PB-jacobi-2d	897 kB	47 kB
WA-bzip2	1.1 MB	142 kB
WA-mnist	915 kB	56 kB

not unexpected, as both employ similar underlying technology through just-in-time (JIT) compilation. WAMR’s implementation relies on the LLVM framework, which is a mature compiler infrastructure with extensive backend optimizations, especially for the riscv64 architecture. In contrast, Wasmtime employs Cranelift, a JIT compiler also developed within the Bytecode Alliance. Cranelift differs from LLVM in three main aspects: (i) it is intentionally designed for simplicity and ease of contribution, (ii) it provides faster compilation speeds, and (iii) it is optimized for security, aligning with WebAssembly’s strong isolation guarantees. Although Cranelift may generate code that is somewhat less optimized than LLVM’s, its significantly faster compilation process contributes to shorter startup times, which is a key advantage in dynamic or short-lived workloads.

The MB-cjpeg and WA-bzip2 benchmarks stand out because of their short execution durations. As these are relatively lightweight workloads, even small absolute differences in runtime performance become more pronounced when viewed relative to native execution times.

A particularly interesting observation arises in the PB-correlation benchmark for WAMR, where execution on both aarch64 and x86_64 platforms appears faster than native execution. Performance counter analysis using tools such as `perf` indicated that WAMR executed a greater total number of instructions compared to native code. However, the native execution exhibited a substantially higher total cycle count, offsetting its instruction efficiency. The precise cause of this anomaly remains unclear and warrants further investigation in future work. Related research from early 2026 observed similar results and suggests a hypothesis involving SSE instructions and branch prediction. The Wasm code path may emit SSE/SIMD vector instructions more aggressively

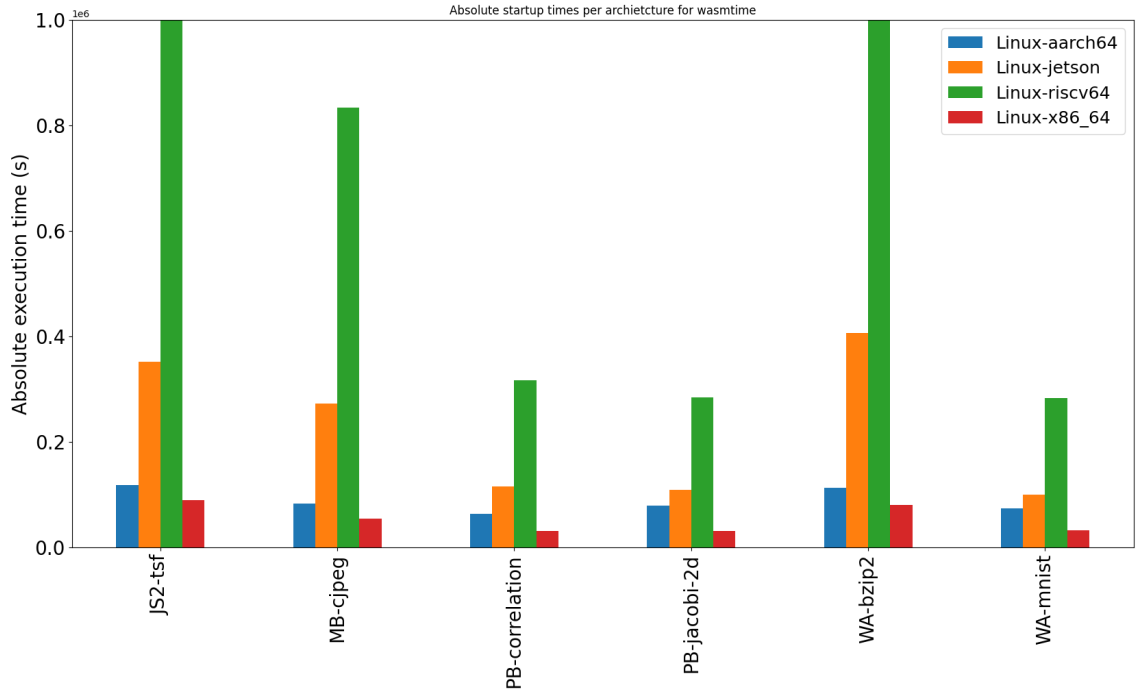
(or differently) than the native compilation with `-O2`, leading to better vectorization of the inner loops in PB-correlation. The Wasm compilation pipeline (Cranelift / LLVM JIT) may generate code that makes better use of the branch predictor due to a different instruction scheduling and loop unrolling strategy compared to the native `-O2` output. This further suggests that a follow-up microarchitectural analysis using hardware performance counters (`perf stat`, `VTune`) with breakdowns of branch mispredictions, cache misses, and IPC (instructions per cycle) would be needed to confirm this hypothesis. This is not an artifact of the experimental setup but potentially a generalizable property of the Wasm compilation path for this class of workload.

4.5.2 Startup Times

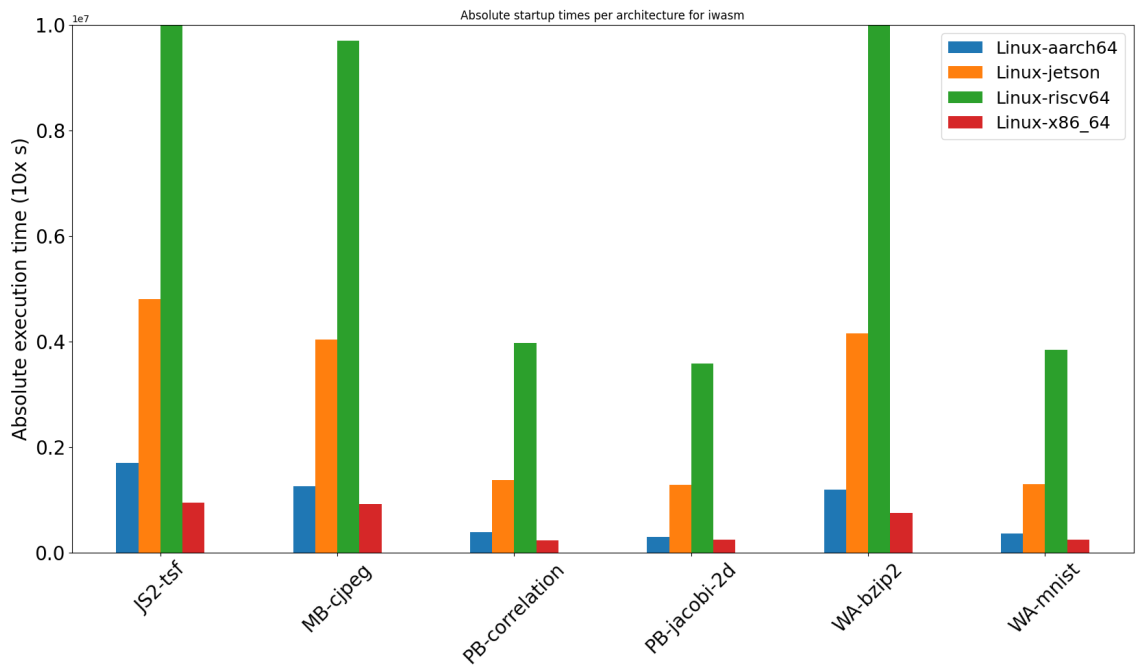
We next examine startup behavior across the evaluated runtimes. Figure 4.8 presents the absolute startup times for Wasmtime (a) and WAMR (b). The most prominent observation is that WAMR, when configured with the LLVM JIT compiler, exhibits startup times roughly an order of magnitude longer than those of Wasmtime. This discrepancy directly stems from LLVM’s more complex optimization pipeline and slower JIT compilation process compared to Cranelift. As we will demonstrate later, the WAMR Fast JIT mode effectively mitigates this shortcoming, offering a more balanced trade-off between performance and startup latency.

Another notable result is that Wasmtime’s startup time on server-class architectures (x86_64 and aarch64) is typically on the order of a few tens of milliseconds. When compared to traditional Function-as-a-Service (FaaS) platforms or the time required to initialize a Docker container, Wasmtime achieves roughly a tenfold improvement in startup latency, underscoring its suitability for lightweight and ephemeral workloads.

Conversely, startup times on embedded devices, especially on the riscv64 platform, are considerably higher. This observation aligns with the roughly $10\times$ performance gap previously seen in Figure 4.7. The disparity between server-class hardware and the Jetson board can largely be attributed to differences in clock frequency. However, the more pronounced gap observed for riscv64 likely results from the simpler microarchitectural design of the SiFive core compared to ARM’s Cortex-A57,



(a) Wasmtime



(b) WAMR (LLVM JIT)

Figure 4.8: Absolute startup times per architecture. In (a), the y-axis is measured in seconds, while in (b), it is scaled by a factor of 10.

including its smaller cache hierarchy and reduced pipeline sophistication. A more in-depth microarchitectural analysis would be needed to fully characterize this performance difference, but such an investigation falls beyond the scope of this work.

4.5.3 WAMR Execution Modes

As outlined in Section 4.4.3.2, WAMR supports multiple execution modes to accommodate a wide range of devices and use cases¹⁰. Figure 4.9 illustrates the relative performance of these modes with respect to native execution and Wasmtime, focusing on the x86_64 platform (the only one supporting the Fast JIT mode).

In summary, the *interp* mode represents pure interpretation, which is expectedly the slowest but offers the smallest binary footprint, less than 300 kB. The *llvm-jit* mode leverages LLVM for just-in-time compilation, producing highly optimized code but incurring substantial startup costs. The *fast-jit* mode employs a lightweight JIT compiler similar in philosophy to Cranelift, achieving a balanced compromise between speed and memory efficiency. Although it cannot match the runtime performance of LLVM JIT, it offers considerably shorter startup times, as seen in Figure 4.10, approaching those of the interpreter.

Finally, WAMR's *multi-tier* mode combines the best of both worlds by initially compiling with the Fast JIT and subsequently reoptimizing in a parallel thread using the LLVM JIT. This approach yields performance comparable to the LLVM JIT while maintaining startup times close to the Fast JIT mode, making it an attractive choice for applications requiring both responsiveness and long-running performance stability.

¹⁰<https://bytecodealliance.github.io/wamr.dev/blog/introduction-to-wamr-running-modes/>

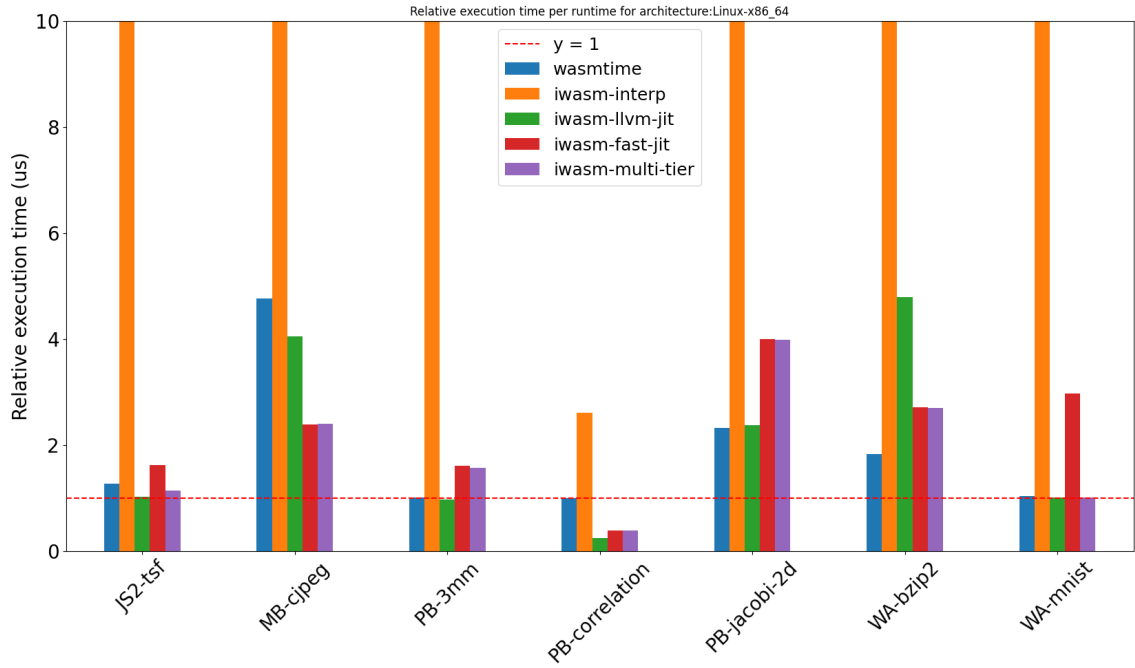


Figure 4.9: Relative execution time for different WAMR execution modes on X86_64.

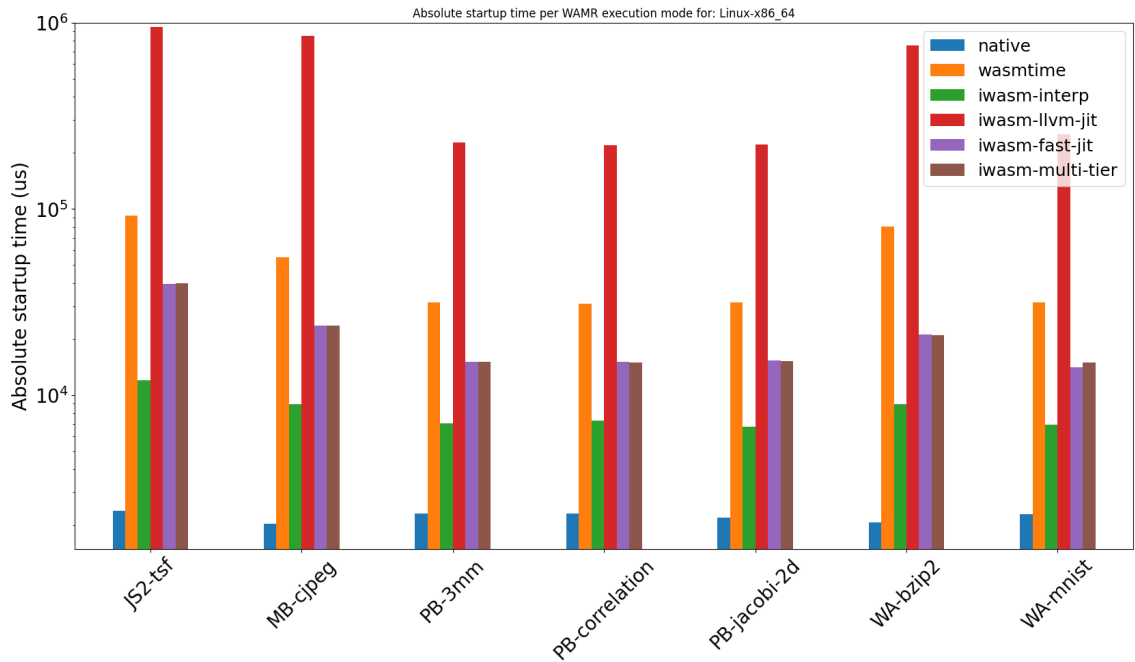


Figure 4.10: Startup time for different WAMR execution modes on X86_64.

4.5.4 Ahead-of-time compilation

Both Wasmtime and WAMR support *ahead-of-time* (AOT) compilation, enabling WebAssembly modules to be precompiled into native binaries

before execution. Figure 4.11 illustrates the relative execution times for both runtimes on three platforms: (a) ARM64 (AWS), (b) RISCV64, and (c) X86_64. In Wasmtime, the AOT process utilizes the Cranelift compiler to generate an executable image that the runtime can directly launch. Conversely, WAMR performs AOT compilation through a dedicated command that leverages the LLVM compiler, the same backend used in its JIT mode.

The experimental results indicate that AOT compilation brings the performance of WebAssembly applications even closer to native execution. Furthermore, the startup latency of AOT-compiled binaries is drastically reduced, typically below 10 μ s since the module is already transformed into native code and requires no runtime translation or optimization.

Although AOT compilation offers clear benefits in execution speed and startup time, it introduces a trade-off in terms of portability. The precompiled binaries become specific to the target architecture, partially sacrificing the architecture-agnostic property that makes WebAssembly appealing for heterogeneous computing environments. However, this limitation is less restrictive than recompiling source code, which depends not only on the processor architecture but also on operating system configurations, runtime versions, and compiler toolchains. Consequently, AOT compilation provides an effective middle ground: it achieves near-native performance and minimal startup delays while maintaining a greater degree of platform flexibility than conventional native executables.

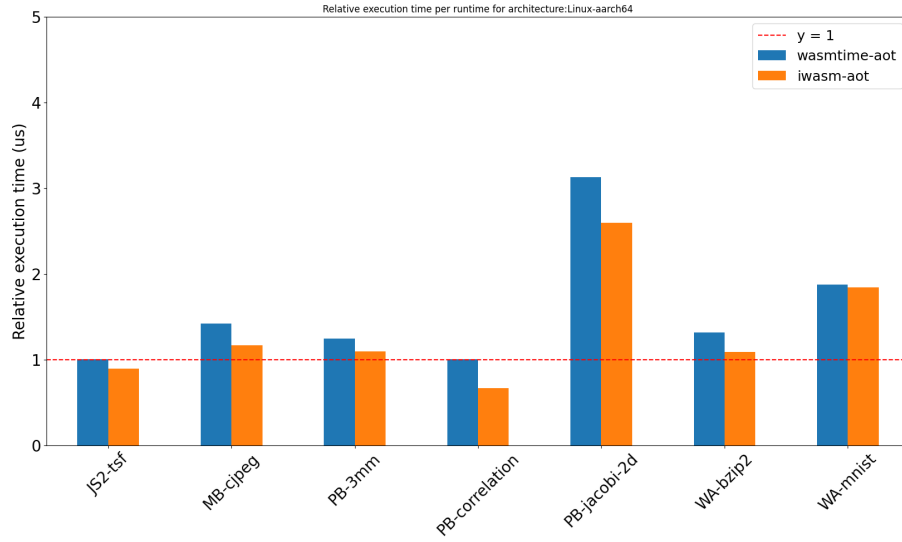
4.5.5 Wasmtime: A High-Performance Runtime

As demonstrated in Section 4.5, Wasmtime consistently achieves near-native execution for the majority of benchmarks, establishing itself as a high-performing runtime across different hardware architectures. Figure 4.8 further illustrates the absolute startup times observed for Wasmtime, which predominantly range from approximately 30 to 150 milliseconds. Remarkably, certain applications running on X86_64 achieve extremely low startup times, in the order of 1 to 2 milliseconds. Among the tested architectures, X86_64 consistently exhibits the shortest startup times across all benchmarks, followed by AARCH64 AWS instances and the ARM Jetson Nano, whereas RISCV64 demonstrates the longest startup

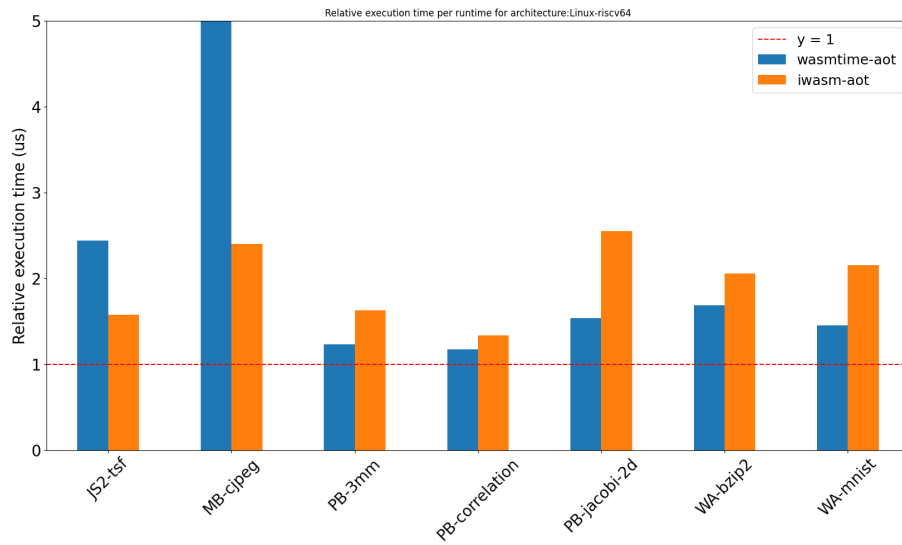
durations.

4.5.6 Insights from WebAssembly Micro Runtime

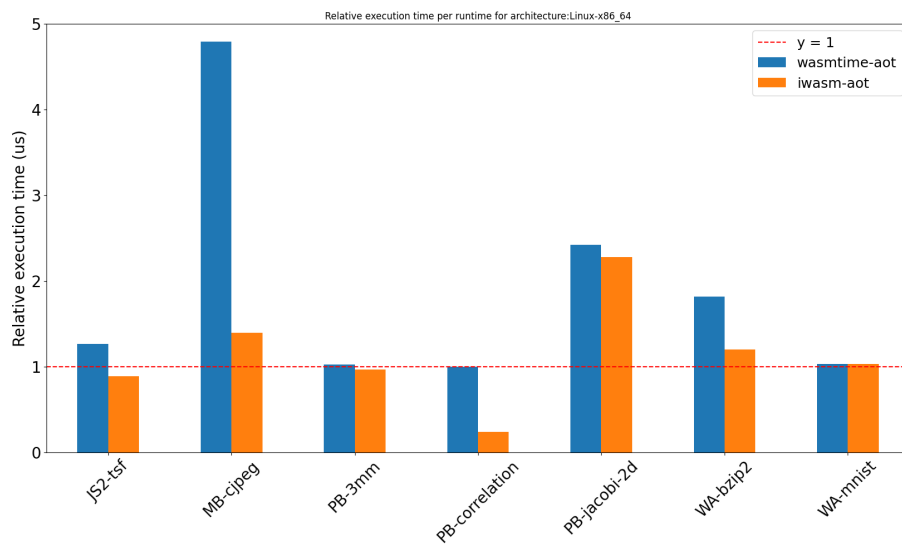
Figure 4.8 also highlights WAMR’s performance across architectures. For AWS instances, nearly all benchmarks exhibit startup times below 10 milliseconds, although larger binaries tend to result in slightly longer initialization durations. The X86_64 platform maintains the fastest startup times, followed sequentially by AARCH64 AWS, ARM Jetson Nano, and RISCV64. Across all architectures, the iWasm startup times remain consistently higher by roughly an order of magnitude compared to Wasmtime. Notably, the influence of architectural differences on Wasmtime is more pronounced, with a larger disparity between X86_64 and AARCH64 AWS startup times, while iWasm demonstrates a more uniform gap across these platforms.



(a) AWS ARM (AArch64)



(b) RISC-V64



(c) AWS x86_64

Figure 4.11: Relative execution performance when performing ahead-of-time compilation of WebAssembly bytecode across architectures.

4.5.7 Considerations for Runtime Selection

Selecting an appropriate runtime requires careful consideration of deployment requirements and target hardware. For cases where Wasm modules need to migrate across heterogeneous architectures with minimal memory consumption, WAMR in interpretation mode on ARM or RISC-V devices offers a feasible solution. However, for broader compatibility, richer feature sets, and generally favorable startup times, Wasmtime emerges as the preferred runtime. Its extensive platform support, comprehensive feature set, and consistently low startup latency make it well-suited for a wide variety of deployment scenarios.

4.6 Discussion

Our experimental evaluation revealed substantial performance differences across benchmarks and hardware platforms. The architectural characteristics of the processors strongly influenced performance outcomes. For instance, the ARM Cortex-A57 benefits from an out-of-order execution pipeline, SIMD extensions, and virtualization support, all of which contribute to its competitive performance. In contrast, the RISC-V platforms utilize a dual-issue, in-order pipeline that, despite promising architectural features, underperforms in practice. Additional features of the SiFive U74 core used in the VisionFive2 board, including L2 cache and error correction mechanisms, played a notable role in shaping observed performance patterns.

Among the evaluated runtimes, Wasmtime consistently delivered the highest performance across all hardware platforms. This highlights opportunities for improving alternative runtimes to enhance both compatibility and performance in diverse environments.

4.6.1 Opportunities for Future Research

The observed performance characteristics suggest several avenues for future exploration. Investigating the factors contributing to the differences in Wasmtime startup times across architectures, identifying the root causes of lower RISC-V performance, and developing strategies to reduce runtime executable sizes will be essential for subsequent research. In addition, certain benchmarks demonstrated relatively strong perfor-

mance on embedded platforms, prompting further examination into the conditions and optimizations that enable such efficiency. Key research questions include:

- **Startup Time Discrepancies:** What architectural or runtime-specific factors lead to significant variation in Wasmtime startup times across platforms, and how can startup performance be optimized for consistency?
- **RISCV Performance Optimization:** Which strategies can enhance WebAssembly runtime performance on RISCV platforms, bridging the observed performance gap and ensuring parity across heterogeneous hardware?
- **Executable Size Reduction:** What innovative approaches can be explored to reduce runtime binary sizes while preserving performance and feature functionality?
- **Benchmark Performance on Embedded Platforms:** What factors contribute to the improved performance of specific benchmarks on resource-constrained devices, and how can these insights inform optimizations for embedded systems?

4.7 Conclusion

WebAssembly has emerged as a versatile and powerful compilation target, enabling developers to navigate the increasingly heterogeneous landscape of cloud-to-edge computing. In this work, we conducted a comprehensive performance analysis of WebAssembly runtimes, providing a first-of-its-kind cross-architecture evaluation encompassing four distinct platforms: AWS servers with X86_64 and ARM64, the Nvidia Jetson with ARM64, and the VisionFive2 board with RISCV64.

By leveraging a carefully annotated set of benchmarks, we evaluated execution characteristics that challenge traditional assumptions regarding cross-platform performance. Our findings deliver critical insights into runtime efficiency, deployment considerations, and overall compatibility across heterogeneous architectures.

In terms of benchmarking contributions, we extended the existing WaBench suite with calibrated and instrumented programs, offering a

valuable resource for the research community. This enhancement allows for precise and reliable performance evaluations in future studies. Our investigation also provides guidance for runtime selection, benchmark suitability, and cross-architecture optimization, helping developers maximize the potential of WebAssembly for cloud-to-edge applications. As the cloud-edge continuum continues to evolve, the insights presented in this thesis will support the development of applications that are truly portable, efficient, and adaptable to diverse computing environments.

Chapter 5

Multiplatform and WebAssembly Containers: The Future of Portable Compute

Contents

5.1 Introduction	80
5.2 Background	82
5.3 Related Work	83
5.4 Methodology	85
5.4.1 Workloads	85
5.4.2 Container Configurations	87
5.4.3 Benchmarking Approach	89
5.5 Results	90
5.5.1 Evaluation of Startup Times: Impact of Dockerfile Practices	91
5.5.2 Image Size Comparison	94
5.5.3 Host Architecture Sizes vs. OCI Registry Sizes	94
5.5.4 Model of Startup Time	97
5.5.5 Correlation between Docker Image Size and Pull Time	98
5.6 Conclusion	99

Docker and WebAssembly are two cornerstone technologies in contemporary software engineering, each offering distinct advantages in portability, deployment, and performance. While Docker has long been recognized for simplifying application distribution through containerization, WebAssembly has emerged as a lightweight, architecture-independent execution format capable of running code securely and efficiently outside the browser. The integration of Wasm into containerized environments

highlights its potential to complement traditional container technologies, offering novel opportunities for performance and flexibility.

Despite this promise, there remains limited empirical understanding of how Wasm-based containers compare with conventional multiplatform containers, particularly when deployed across diverse hardware architectures and workloads. Key performance aspects, such as startup latency, image pull times for both fresh and cached scenarios, and overall image size, are insufficiently characterized for Wasm containers in practical, multi-architecture deployments.

In this chapter, we aim to address these gaps by systematically evaluating the performance and practical implications of native multiplatform containers versus Wasm-based containers under optimized configurations. Our analysis covers three hardware platforms: AMD64 servers (AWS bare metal), Nvidia Jetson Nano with ARM64, and StarFive VisionFive2 with RISCV64, which represent both cloud-centric and edge-oriented environments.

We conducted a comprehensive series of experiments leveraging Docker and containerd for native multi-platform images and employed Wasmtime as the execution engine for Wasm within the container runtime ecosystem. Critical metrics examined include startup times, pull times (for both cold and cached image scenarios), and container image sizes across the selected platforms. Our findings reveal that while native containers generally exhibit marginally faster startup times, Wasm containers offer significant advantages in agility, portability, and storage efficiency. In particular, Wasm images achieved approximately twenty-seven percent of the size of their native equivalents, with pull times reduced by up to twenty-five percent when using containerd.

These results suggest that Wasm containers, while not positioned to fully replace traditional container paradigms in the near term, can complement existing workflows by providing highly portable, lightweight alternatives that reduce resource consumption and accelerate deployment across heterogeneous platforms. With further optimization, Wasm is poised to become a valuable component in environments that demand both efficient cross-platform execution and minimal image footprint.

5.1 Introduction

Containers have become fundamental to cloud-native computing, providing scalable, portable, and efficient mechanisms for deploying distributed applications. Concurrently, WebAssembly (Wasm) has gained prominence as a compact binary format and portable compilation target, enabling programs written in multiple languages to run efficiently across diverse platforms with performance close to native execution.

The intersection of containerization and WebAssembly presents new opportunities for deploying applications in heterogeneous environments. Containers encapsulate applications along with their dependencies, guaranteeing consistent execution regardless of the underlying infrastructure. Wasm, by contrast, offers a secure sandboxed runtime with low overhead, capable of running efficiently both inside and outside the browser. With the WebAssembly System Interface (WASI), Wasm execution has expanded beyond web applications into cloud and edge computing, allowing the same binary to run seamlessly across multiple architectures. Recent enhancements in container runtimes, including Docker, have added support for alternative execution engines such as Wasm runtimes, allowing WebAssembly modules to be packaged and executed as containers.

This study investigates the performance and usability implications of deploying Wasm-based containers through Wasmtime in comparison with traditional Linux containers. Our focus is the emerging cloud-edge continuum, characterized by diverse hardware architectures. Modern data centers already integrate both x86_64 and ARM64 servers, and this diversity is expected to increase with the proliferation of edge devices, accelerators, and IoT platforms. Consequently, software designed for this environment must be architecture-agnostic. Developers currently achieve this either by building multi-platform Docker images for each architecture or by using WebAssembly containers, which are naturally portable across hardware.

Figure 5.1 depicts the workflow for cross-architecture container execution. In the native multi-platform approach, developers create architecture-specific binaries and assemble a *multi-architecture Docker image*. Execution relies on runtime layers such as `containerd` and `runc`. In contrast, a Wasm-based container requires only a single image; Wasmtime replaces `runc`, compiling WebAssembly bytecode to native instructions at runtime.

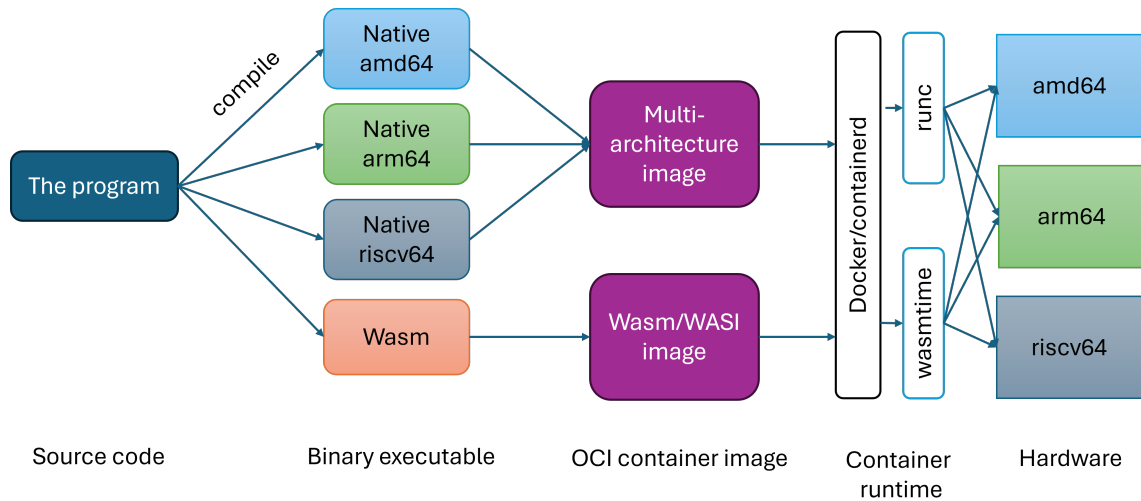


Figure 5.1: Workflow for cross-architecture container execution.

For cloud-native and edge deployments, rapid startup and minimal resource overhead are critical, particularly in serverless scenarios. While native containers provide high performance, they often incur larger image sizes and require architecture-specific builds. Wasm containers, conversely, are lightweight and portable, although this can introduce modest performance trade-offs. Multi-architecture containers improve security and portability but add operational complexity, emphasizing the need to assess both performance and usability before deployment.

Previous studies have explored WebAssembly as an alternative to Docker containers for IoT and edge use cases [68], but comprehensive evaluations comparing Wasm containers with optimized multi-platform native containers are limited. Many prior works focus on cold-start latency reductions but do not employ best practices for multi-architecture container builds [16].

To fill this gap, we evaluate key metrics including image size, startup time, and pull duration for both Wasm and native containers, leveraging Docker and `containerd` for efficient multi-platform management. Our experiments show that Wasm containers are, on average, around 85% smaller than their native counterparts, with pull times reduced by up to 25% across `amd64`, `arm64`, and `riscv64` platforms when using `containerd`. These results demonstrate that Wasm containers provide a compelling combination of portability and efficiency while complementing traditional container workflows.

The primary contributions of this work are:

1. A first comprehensive evaluation of Wasm integration with Docker-alternative runtimes, such as Wasmtime, across multiple hardware architectures.
2. A comparative performance analysis of optimized multiplatform native and Wasm containers, measuring image size, startup time, and pull duration across three workloads and architectures (`amd64`, `arm64`, `riscv64`) using `containerd`.
3. Insights into best practices for multi-architecture container deployment, including optimized image creation and direct `ctr` usage, demonstrating practical guidance for both Wasm and native workloads.

5.2 Background

Docker containers and WebAssembly (Wasm) are two foundational technologies that have significantly influenced modern software development. Docker provides a standardized way to package applications with all their dependencies, enabling consistent execution in any environment that matches the architecture for which the software was built. Wasm, on the other hand, is a compact binary format designed as a portable compilation target for languages such as Rust, C, C++, JavaScript, C#, and Go.

In practice, Docker containers are typically distributed via a *container registry*, such as Docker Hub¹. The container runtime checks the local system for the requested image and, if not found, retrieves it from the registry.

WebAssembly execution occurs in three main stages: decoding, validation, and execution. These steps ensure that the code runs at near-native speeds while maintaining a secure, sandboxed environment. Execution can occur via interpretation, just-in-time compilation, or ahead-of-time compilation, with some runtimes using tiered approaches to further optimize frequently executed functions. Wasmtime, the runtime used in our experiments, compiles Wasm bytecode in a single step before execution, providing both performance and security guarantees.

The combination of Docker and WebAssembly allows developers to

¹<https://hub.docker.com/>

leverage Wasm’s sandboxing and portability while benefiting from Docker’s mature image management and distribution capabilities. Containerizing Wasm modules enables seamless integration with existing container orchestration systems, supporting continuous integration/continuous deployment (CI/CD) workflows, versioning, and registry services. This approach ensures consistent execution across heterogeneous environments, simplifies deployment, and facilitates scaling, making it particularly suitable for cloud-edge computing scenarios.

The integration of Wasm within containerized environments offers a bridge between traditional, architecture-specific deployment and lightweight, portable execution. Figure 5.2 illustrates the fundamental differences between native and Wasm containers in terms of architecture dependency, portability, and ecosystem maturity. Native containers are typically tied to specific processor architectures and operating systems, whereas Wasm containers encapsulate precompiled binaries that can execute across heterogeneous platforms with minimal configuration, offering an emerging but promising alternative for multi-environment deployment.

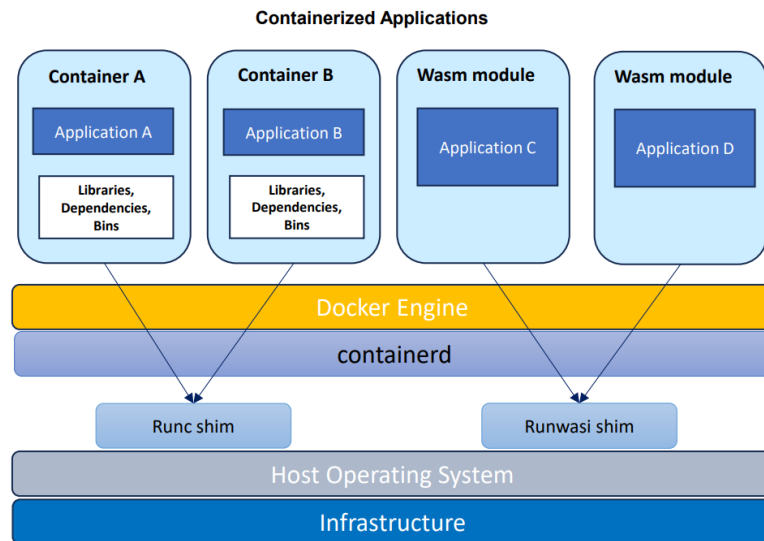


Figure 5.2: Containerized Applications: Native and Wasm

5.3 Related Work

The modern cloud landscape is increasingly heterogeneous, spanning data center servers, regional and on-premise servers, mobile edge stations, and end-user devices. This continuum of computing resources is often referred to as the *cloud-edge continuum* [21]. Heterogeneity

in hardware and software platforms presents challenges for software deployment and performance, particularly in serverless environments. Kjorveziroski et al. [43] highlight the impact of serverless computing on cloud and edge networks, emphasizing latency challenges associated with containers and microvirtual machines and proposing Wasm as a potential solution for reducing cold-start delays. Similarly, Fujii et al. [15] explored the migration of stateful virtual machines across Wasm runtimes, noting the challenges caused by non-standardized runtime implementations.

Several studies have evaluated Wasm performance in cloud and edge computing contexts. Ferrari et al. [12] examined Wasm execution on edge devices, while Fiedler et al. [13] focused on startup latency in serverless scenarios. Felter et al. [11] analyzed Docker container performance on x86 architectures but did not investigate Wasm. Other research has considered WebAssembly’s role in improving quality of service, security, and cross-platform compatibility [25, 35]. For example, performance analysis using the Sightglass benchmark demonstrated architecture-specific behaviors of Wasmtime [35].

Chadha et al. [7] explored Wasm as a distribution format for MPI-based HPC applications, introducing MPIWasm, which delivers near-native performance while significantly reducing binary size compared to statically linked binaries. Bosshard et al. [4] evaluated server-side Wasm runtime options, including Wasmer, Wasmtime, and Lucet, in serverless platforms like OpenWhisk and AWS Lambda. Spies et al. [84] and Wang et al. [91] employed the PolyBench suite to assess Wasm, though it is limited in representing non-web workloads. Hockley et al. [26] used Wasm as a sandboxed scripting environment but only on specific hardware. Other works have evaluated cross-platform performance on ARM architectures [54] and emphasized the need for portable, edge-friendly solutions [34].

To address gaps in prior research, Wang et al. [90] introduced WaBench, a comprehensive benchmark suite covering multiple Wasm runtimes and diverse application domains. Kakati et al. [32] further analyzed the performance of the two most widely used Wasm runtimes with instrumented benchmarks, highlighting Wasm’s potential for cross-architecture cloud-to-edge frameworks.

The growing diversity of hardware architectures poses challenges for application development and deployment, motivating research on multi-

architecture containers and the integration of Wasm into container ecosystems [53, 62, 64]. Ensuring software runs efficiently across heterogeneous nodes without extensive architecture-specific modifications remains an active and critical research area.

5.4 Methodology

To explore Docker’s integration with WebAssembly, it is important to understand the underlying architecture of the Docker Engine. Docker relies on a higher-level runtime, `containerd`, which manages container lifecycles and orchestrates their execution. Through a shim process, `containerd` can leverage low-level runtimes such as `runc`, which interacts directly with the operating system to manage container operations. This architecture allows the creation of custom shims to support alternative runtimes, including WebAssembly runtimes like Wasmtime, Spin, and Wasmedge.

Our experiments were conducted across three hardware architectures:

- **amd64:** The benchmarks were run on an AWS bare metal server (Linux, AMD64).
- **arm64:** The benchmark was run on an Nvidia Jetson Nano server (ARM64 architecture).
- **riscv64:** We used an additional benchmark that was also run on a StarFive VisionFive2 board with RISC-V64 architecture.

This multi-architecture approach enabled a comparative evaluation of WebAssembly containers versus native containers across diverse hardware platforms. An overview of the experimental methodology is presented in Fig. 6.1.

5.4.1 Workloads

We designed three distinct workloads to evaluate the performance of both native and WebAssembly containers:

- **Matrix Multiplication Workload:** Implements matrix multiplication using Rust, C++, and TinyGo, along with their corre-

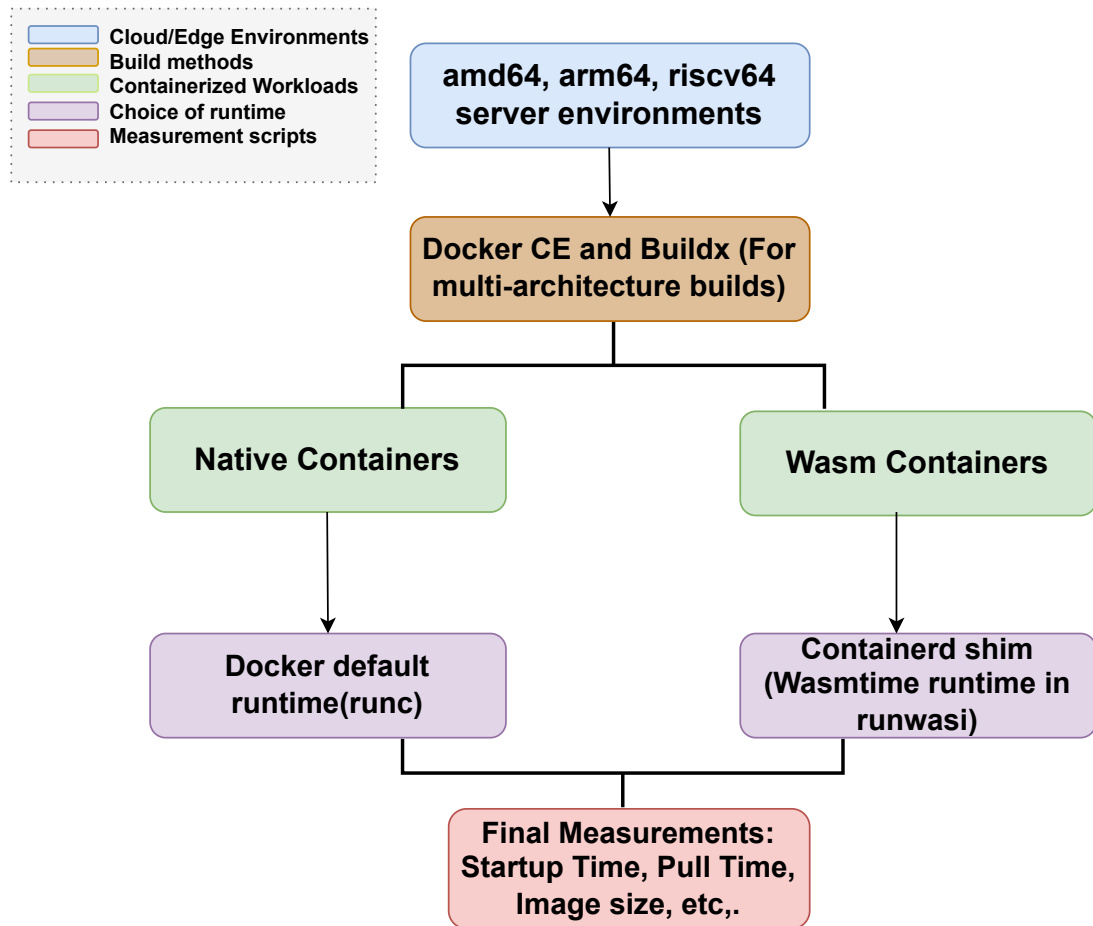


Figure 5.3: Overview of native and wasm container workflows

sponding Wasm versions. This workload assesses computational performance under intensive mathematical operations.

- **Sorting Workload:** Reads a text file, sorts its contents, and writes the results to a new file. The sorting algorithm supports both batch ("all at once") and incremental ("one at a time") processing. Implementations exist in Rust, C++, TinyGo, and their Wasm counterparts. This workload evaluates file I/O performance and memory management efficiency.
- **Face Detection Workload:** Developed in C++, this workload is executed directly on containerd across all three architectures, bypassing Docker entirely. It serves as a real-world application scenario, testing both performance and integration with the runtime environment.

To study performance and resource utilization, experiments were conducted on three hardware platforms, each representing a different in-

struction set architecture. Table 6.1 summarizes the specifications of these platforms.

Table 5.1: Hardware Architectures Used in Experiments

Platform	CPU Model	Clock frequency	Threads per Core	Cores per Socket	Caches
X86_64 AWS c5.metal	Intel Xeon 8275CL	3 GHz	2	24	L1d: 32 KiB, L1i: 32 KiB, L2: 1 MiB, L3: 71.5 MiB
Nvidia Jetson Nano	ARM Cortex-A57	1.5 GHz	2	4	L1d: 32 KiB, L1i: 48 KiB, L2: 2048 KiB
StarFive VisionFive2	SiFive U74 Core	1 GHz	1	4	L1d: 32 KiB, L1i: 32 KiB, L2: 2048 KiB

5.4.2 Container Configurations

For each workload, both native and WebAssembly implementations were containerized, with optimizations applied to ensure fair comparison. Rust applications were compiled using `--release`, C++ with `-O3`, and TinyGo with `-opt=2`, representing the highest available optimization levels for each language. While Rust `--release` and C++ `-O3` provide aggressive optimizations focused on maximum performance, TinyGo’s `-opt=2` is the most advanced option available in TinyGo, though slightly less aggressive than the other compilers.

Native Containers: Rust, C++, and TinyGo applications were compiled natively for each target architecture. Multi-platform Docker builds (`--platform linux/amd64,linux/arm64`) were used to generate container images compatible with AMD64 and ARM64. For RISCV64, a slightly different approach was necessary due to the lack of a common base image across architectures, but the final image similarly contains sub-images for each architecture.

Dockerfiles were optimized using multi-stage builds, separating the build environment from the runtime environment. This ensures that the final image contains only essential runtime dependencies. The first stage uses standard Linux base images (e.g., Alpine for AMD64/ARM64, Ubuntu for RISCV64) for compilation. The resulting binary is then copied into a minimal runtime image, producing a compact and efficient container image.

Wasm Containers: Applications were compiled to WebAssembly and executed using the Wasmtime runtime via `docker run`. To enable this, the containerd `wasmtime` shim, provided by the `runwasi`²

²<https://github.com/containerd/runwasi>

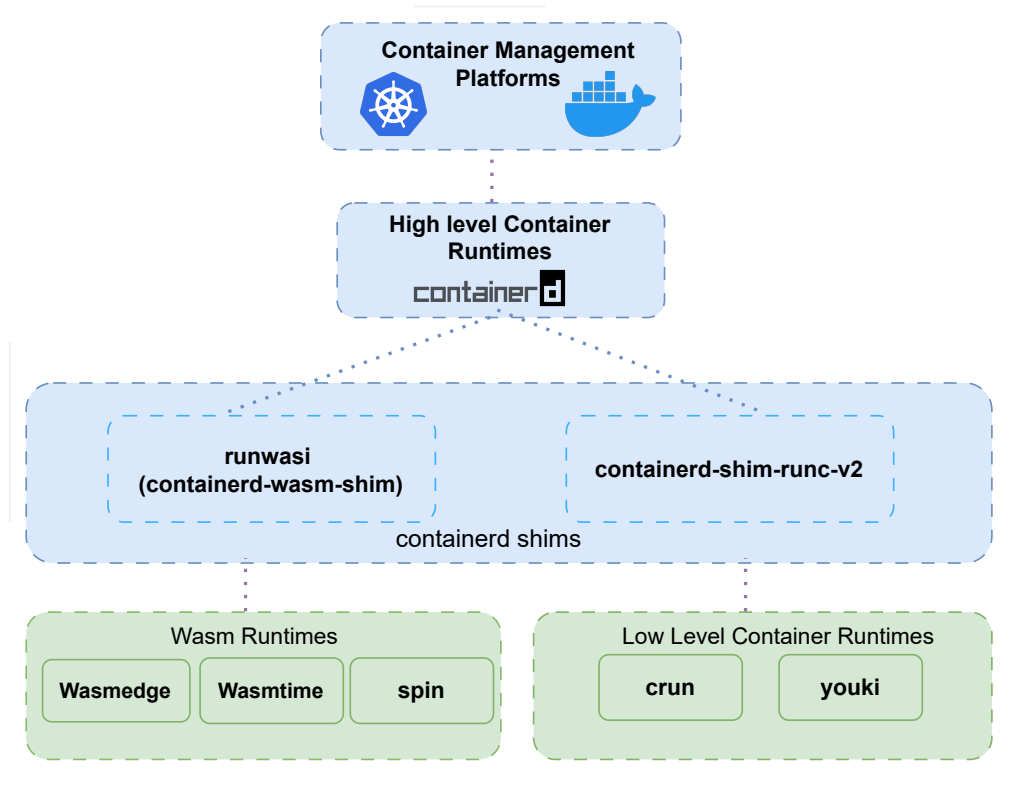


Figure 5.4: Wasi support with containerd shims.

project, was installed. This allows Wasmtime to act as an alternative container runtime³, offering a secure, sandboxed execution environment.

The Wasm container images were built using `scratch` as the base image, containing only the WebAssembly binary and no additional dependencies. This approach minimizes image size and attack surface, making it suitable for resource-constrained or security-sensitive environments. Containers were executed with `--platform wasm` and `runtime=io.containerd.wasmtime.v1`, whereas native containers used Docker’s default runtime.

Figure 6.6 illustrates the workflow of using WebAssembly as an alternative Docker runtime. When a container is executed via `docker run`, Docker interacts with `containerd` to manage the container lifecycle. The runtime first checks the local image store and pulls a fresh image from the registry (e.g., Docker Hub) if the image is not available locally.

³<https://docs.docker.com/engine/daemon/alternative-runtimes/>

5.4.3 Benchmarking Approach

To evaluate container performance, we developed a custom benchmarking script that measures startup times for both native and Wasm containers. The script first detects the system architecture, enabling it to run the appropriate container image for AMD64, ARM64, or RISCV64. Each application was instrumented to record the current timestamp at the start of its `main` function, providing a precise measure of when program execution begins. Additionally, the script records the image pull time, capturing the overhead associated with retrieving a container image from a remote registry during a fresh pull.

High-resolution timestamps were obtained using the `date` command, and containers were executed via Docker’s standard mechanisms. For cold-start measurements, the script removes any existing local image to ensure the container is pulled anew, providing accurate startup time evaluation. This approach combines internal timing instrumentation with external execution monitoring to ensure reliable and consistent results across different hardware environments.

Two primary performance metrics were considered:

- **Pull Time:** The time required to retrieve the container image from a remote OCI registry when it is not already present locally.
- **Container Startup Time:** The duration from when the image is loaded and initialized until the container’s `main` function begins execution.

The script supports both fresh and cached image pulls, enabling performance analysis under different operational scenarios⁴. Measurements were conducted across multiple architectures (AMD64, ARM64, and RISCV64), executing the workloads and calculating the startup time as the interval between image pull and program execution.

The selected hardware platforms reflect a range of computing environments. The AWS instance with an Intel Xeon 8275CL processor represents high-performance cloud servers, leveraging advanced parallel pipelines, dynamic instruction scheduling, branch prediction, and cache prefetching. In contrast, the Nvidia Jetson Nano and StarFive VisionFive2 boards represent edge computing devices with ARM64 and RISC-

⁴https://github.com/sangeeta1998/benchmark_sort/

V64 architectures, respectively, highlighting performance in resource-constrained environments.

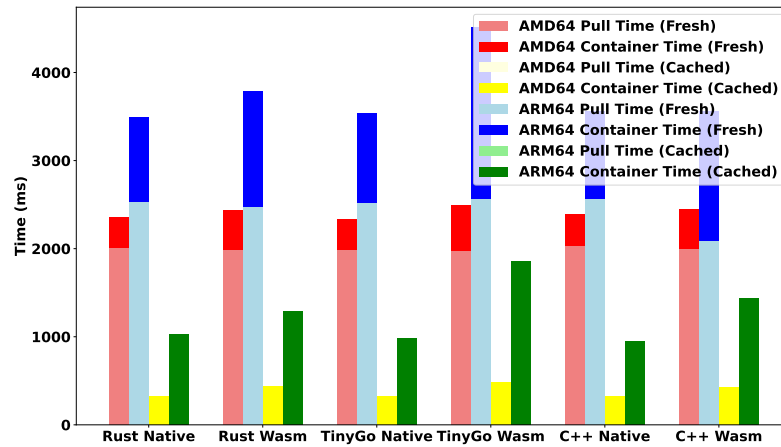


Figure 5.5: Startup Times (image pull and container time) for Matrix Multiplication.

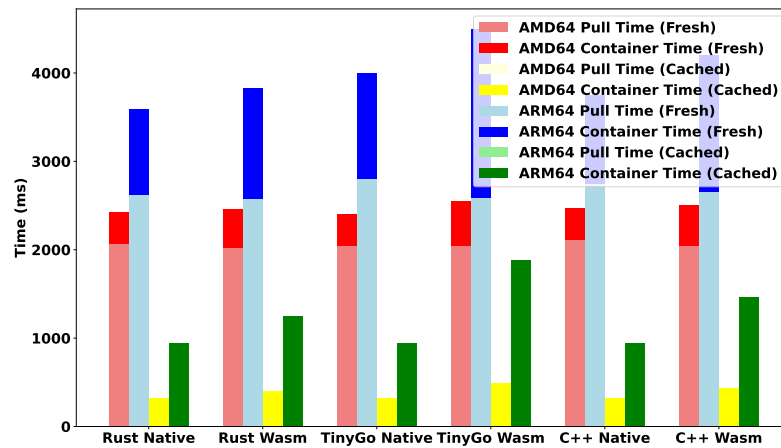


Figure 5.6: Startup Times (image pull and container time) for Sorting Benchmark.

5.5 Results

We evaluated the startup performance of native and Wasm containers using the matrix multiplication and file sorting workloads. For all experiments, we followed best practices by employing multi-stage Docker builds and optimized base images. Figures 6.7a and 6.7b illustrate the results for the Matrix Multiplication and Sorting workloads, respectively.

Overall, native containers show slightly better performance, though the differences in image sizes and startup times between native and Wasm containers are modest. While WebAssembly execution is generally fast and competitive as elaborated in our research [32], a small

overhead remains compared to native execution, primarily due to the runtime compilation of Wasm bytecode.

In Figures 6.7a and 6.7b, the red and blue bars represent image pull time and container startup time for all architectures and source languages. The yellow and green bars indicate the startup time when the container image is already cached locally.

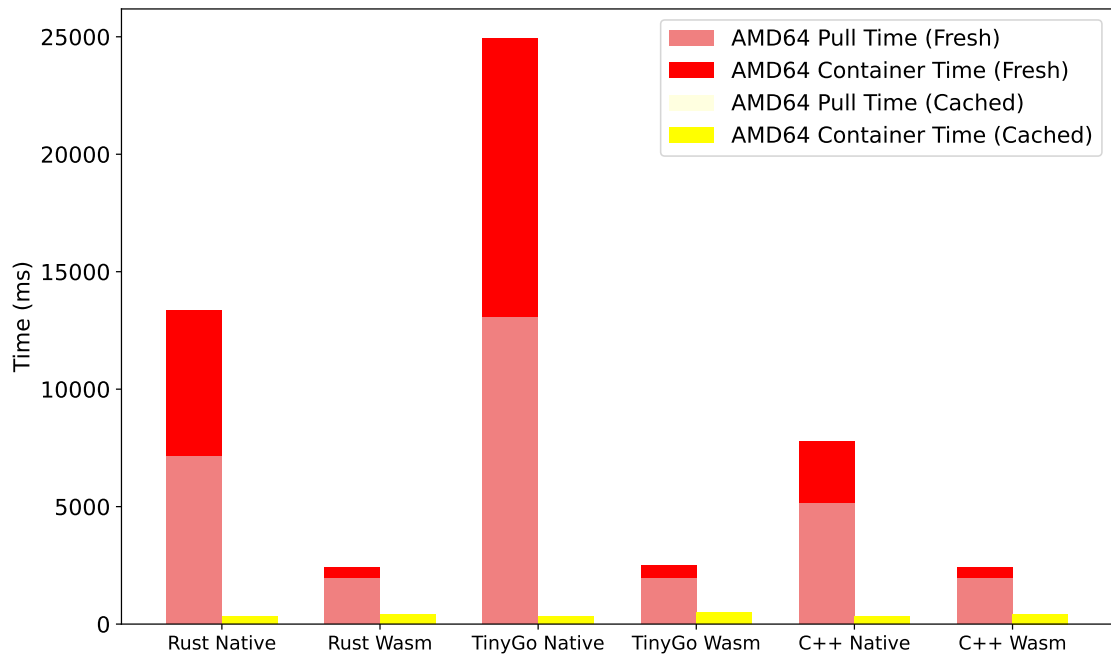
For the AMD64 architecture, image pull times (light red) are similar for both native and Wasm images across all source languages. On ARM64, minor deviations are observed: for example, the matrix multiplication C++ Wasm image is pulled slightly faster than the native image, likely due to its smaller image size (see Figure 6.9). A similar trend is seen for the Sorting workload with TinyGo.

Regarding container startup after pulling the image, Wasm containers generally require slightly more time than native containers. This difference is expected, as Wasm containers incur additional overhead for compiling bytecode before execution. Certain languages, such as TinyGo, exhibit greater sensitivity to this overhead, suggesting that Wasm compilation and optimization for TinyGo may not yet be as mature as for Rust or C++.

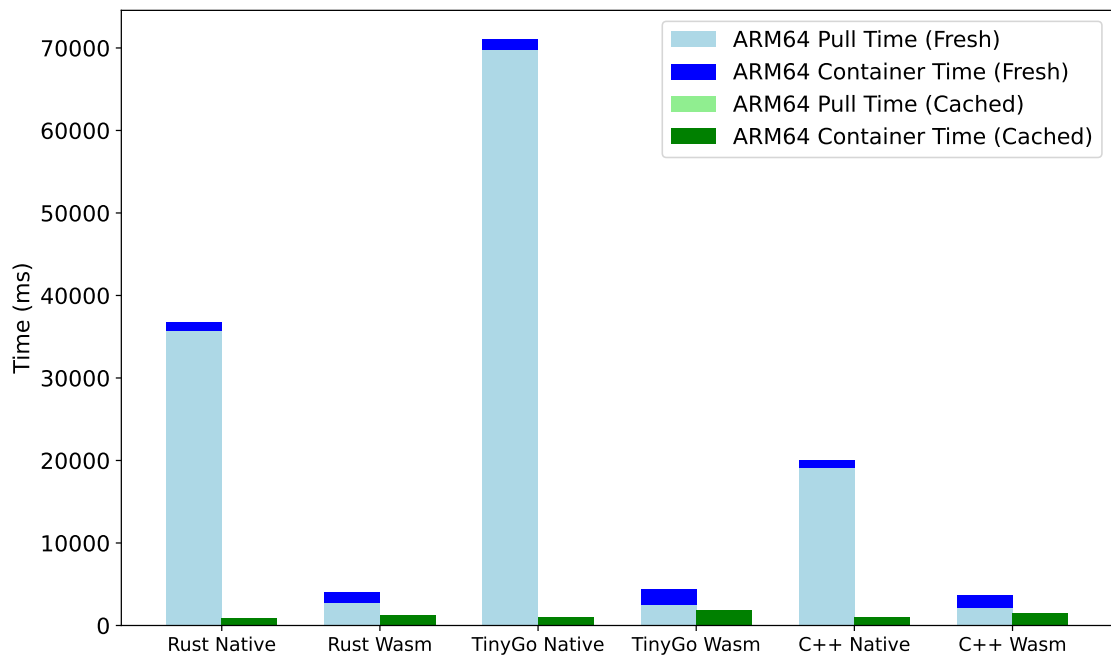
5.5.1 Evaluation of Startup Times: Impact of Dockerfile Practices

To understand the performance differences between WebAssembly containers and native containers reported in previous studies, we conducted an additional set of experiments using the matrix multiplication workload. In this experiment, we intentionally avoided the use of Dockerfile optimizations such as multi stage builds and scratch base images. This allowed us to observe the impact of standard, unoptimized Docker configurations on container startup times and image sizes. The goal was to more closely mirror the scenarios often reported in related work, where Wasm containers are claimed to have substantially faster startup times compared to native containers.

The choice of base images had a significant influence on performance. For C++ applications, an Alpine based image resulted in a relatively compact container of 433 MB. In contrast, Rust required a larger Alpine image of 927 MB to include the necessary Rust toolchain and runtime de-



(a) Average Startup Time in AMD64.



(b) Average Startup Time in ARM64.

Figure 5.7: Pull and Startup Time for AM64 and ARM64: Without best Practices.

dependencies. TinyGo applications relied on the `tinygo/tinygo:latest` base image, which resulted in the largest container size at 1.55 GB, as TinyGo does not provide native support for ARM64 on minimal Linux distributions. These variations in base image size contributed directly to differences in image pull times and overall startup performance.

The experimental results are summarized in Figure 5.7. On AMD64 architecture, the pull time for the native Rust image was 7166 ms, followed by an additional 6190 ms for container startup, resulting in a total startup time of 13,356 ms. In comparison, the Wasm Rust container required only 1996 ms for both image pull and startup, representing a reduction in startup time of approximately 72%. Similar trends were observed for the other programming languages. Wasm TinyGo containers reduced total startup time by 85% compared to native, and Wasm C++ containers reduced total startup time by 82%.

On ARM64, the performance gap between Wasm and native containers was slightly smaller but still significant. Native Rust containers required more time to pull and start the image compared to Wasm Rust, with Wasm achieving a total startup time approximately 66% faster. Wasm TinyGo containers started 77% faster than their native counterparts, and Wasm C++ containers achieved a 69% reduction in total startup time. These results indicate that the performance advantages of Wasm containers are consistent across architectures, although the magnitude varies depending on the platform and the size of the container image.

The experiment demonstrates that under non optimized Dockerfile configurations, Wasm containers can start significantly faster than native containers. The primary contributing factors include the smaller binary size of Wasm modules and the reduced overhead of pulling and initializing the image. In contrast, optimized native multi platform images that leverage multi stage builds and scratch base images can reduce image size and startup times considerably, narrowing the performance gap between native and Wasm containers.

Overall, these findings suggest that Wasm containers are particularly advantageous in scenarios where Dockerfile optimizations are not applied or where minimizing startup time is critical. However, in fully optimized environments, native containers achieve startup times comparable to Wasm, indicating that both approaches can be suitable depending on the deployment requirements. Although this study focused on Rust,

C++, and TinyGo, other languages such as Java, C#, Python, and JavaScript, which have larger runtime dependencies, may show different performance characteristics, warranting further investigation.

For all other experiments and results presented in this thesis, we used optimized native Docker images to ensure fair comparison with Wasm containers.

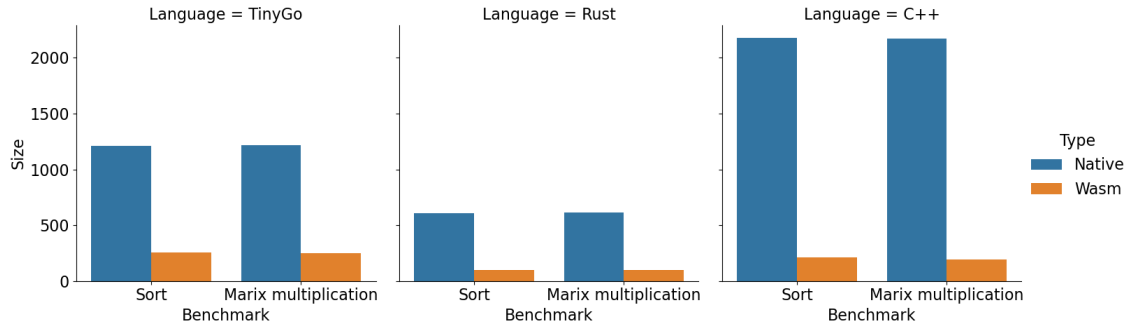


Figure 5.8: Host-based Image Sizes comparison.

5.5.2 Image Size Comparison

Wasm containers provide much smaller image sizes in comparison to native containers, making them ideal for environments with limited resources. Figure 6.9 illustrates the image sizes on the host architecture once they have been downloaded and stored locally. Figure 6.10 displays the compressed sizes of these images at the OCI Registry (DockerHub). It is important to note that this data pertains to a single architecture. To support multiple architectures, the actual size stored at the registry must be multiplied by the number of architectures, as the image size is approximately consistent across different architectures.

5.5.3 Host Architecture Sizes vs. OCI Registry Sizes

Table 5.3 and 5.4 provide a summary of the image sizes for both workloads, comparing the image size stored in the registry (OCI size) to the image size during runtime. To conserve storage space and reduce transfer time, registry images are compressed. The size reduction from the host architecture to the OCI registry is determined as demonstrated in Equation 6.1.

$$\text{Reduction} = \left(1 - \frac{\text{OCI Size}}{\text{Host Size}} \right) \times 100 \quad (5.1)$$

Table 5.2: Sizes of benchmark programs in native and WebAssembly formats.

Benchmark	Language	Type	Size
Sort	TinyGo	Native	1.21 MB
	Rust	Native	610 kB
	C++	Native	2.18 MB
	TinyGo	WASM	257 kB
	Rust	WASM	96 kB
	C++	WASM	212 kB
Matrix	TinyGo	Native	1.22 MB
	Rust	Native	613 kB
	C++	Native	2.17 MB
	TinyGo	WASM	250 kB
	Rust	WASM	96.1 kB
	C++	WASM	192 kB

Table 5.3: Host Architecture Sizes vs. OCI Registry Sizes for Sorting Benchmark.

Language	Host Size (KB)	OCI Size (KB)	Reduction (%)
TinyGo Native	1210	373.36	69.2
Rust Native	610	259.27	57.5
C++ Native	2180	746.24	65.8
TinyGo WASM	257	100.31	61.0
Rust WASM	96	40.79	57.5
C++ WASM	212	68.5	67.7

When an OCI image is pulled, only the compressed image layers are transferred over the network, and decompression occurs locally on the client machine after the download is complete. This approach reduces the amount of data transmitted and makes the pull process more efficient.

In our experiments, the small sizes of the images used mean that the difference in pull time between native and Wasm containers is minimal, even when the native images are significantly larger. This is because Docker introduces its own overhead on top of containerd, the service responsible for actual image management.

The fresh startup of a Docker container involves several sequential steps:

1. **Docker command and local cache check:** Docker first receives the command and checks whether the requested image is available locally.

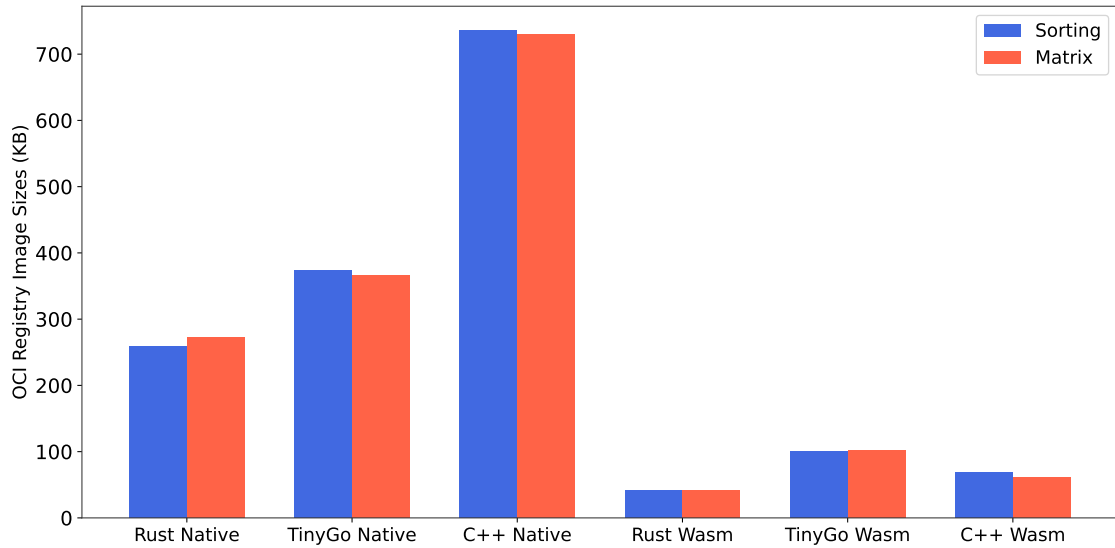


Figure 5.9: Image Sizes from OCI Registry for the AMD64 architecture.

Table 5.4: Host Architecture Sizes vs. OCI Registry Sizes for Matrix Multiplication Benchmark.

Language	Host Size (KB)	OCI Size (KB)	Reduction (%)
TinyGo Native	1220	366.46	69.9
Rust Native	613	273	55.5
C++ Native	2170	735.43	66.1
TinyGo WASM	250	101.26	59.5
Rust WASM	96.1	42.25	56.1
C++ WASM	192	60.38	68.6

- Registry request and network transfer:** If the image is not cached, Docker requests it from the registry (e.g., Docker Hub), which processes the request and begins sending the image layers.
- Image download and decompression:** The compressed layers are transferred over the network and decompressed locally. This step is directly impacted by image size.
- Container runtime loading and execution start:** The image is loaded into the container runtime, and the container begins execution. This step is also partially affected by image size.

Overall, only the image download and decompression (step 3) and to some extent the runtime loading (step 4) are sensitive to the size of the image. In our measurements, the smaller Wasm images reduce these steps slightly compared to native containers, but Docker’s inherent overhead diminishes the observable gains.

5.5.4 Model of Startup Time

The steps which depend on image size can be denoted as:

- T_d as the time to download the image.
- T_c as the time to decompress the image.
- T_l as the time to load the image into the runtime.
- S as the size of the image.

A model for the startup time T could be:

$$T = T_d(S) + T_c(S) + T_l(S) \quad (5.2)$$

where:

- $T_d(S)$ is proportional to the image size S .
- $T_c(S)$ is proportional to the image size S .
- $T_l(S)$ is proportional to the image size S .

For a native program, the startup time:

$$T_{\text{native}} = T_d(S) + T_c(S) + T_l(S) \quad (5.3)$$

For a WASM program, the startup time is slightly different due to the runtime environment:

$$T_{\text{wasm}} = T_d(S) + T_c(S) + T_l(S) + T_{\text{runtime}} \quad (5.4)$$

where T_{runtime} is the additional time required to initialize the WASM runtime.

5.5.4.1 Image Pull Times

We measured the average image pull times over ten iterations for each architecture using `containerd`. The result is presented in Table 5.5. These measurements clearly demonstrate the performance advantage of WebAssembly (Wasm) containers over native containers across all tested architectures.

Table 5.5: Comparison of Pull Time Reduction between Wasm and Native Containers.

Architecture	Wasm (s)	Native (s)	Reduction (%)
amd64	2.012	2.283	12%
arm64	2.354	2.681	12%
riscv64	2.123	2.813	25%

Across all architectures, Wasm images consistently exhibit faster pull times. The most substantial improvement is observed on the `riscv64` platform, where the Wasm container achieves a 25% reduction in pull time relative to the native image. The reductions for `amd64` and `arm64` are both 12%, demonstrating consistent efficiency gains across mainstream architectures. These results demonstrate that WebAssembly containers not only provide a smaller image footprint but also significantly reduce image retrieval latency when managed through `containerd`. The streamlined execution model of Wasm and the direct image handling of `containerd` combine to deliver measurable performance benefits, particularly for architectures with limited I/O bandwidth or slower decompression throughput.

5.5.5 Correlation between Docker Image Size and Pull Time

In this section, we examine the relationship between Docker image size and pull time for both native and WebAssembly (Wasm) images.

To ensure a fair comparison, we used a compressor workload that processed files of varying sizes: 10KB, 50KB, 100KB, 500KB, 1MB, 5MB, 10MB, 100MB, 250MB, 500MB, 750MB, and 1024MB. Both native and Wasm images were built with identical file sizes, ensuring comparable image sizes in Docker Hub.

The results indicate that for each additional megabyte of image size, pull times tend to increase more noticeably for Wasm images compared to native images. This can be attributed to several factors: Docker images are typically compressed to reduce transfer size, and differences in compression efficiency or decompression performance can influence pull times. Wasm images may involve additional runtime requirements and platform specifications (e.g., `--platform=wasi/wasm`), which introduce slight overhead. Moreover, Wasm support in Docker is relatively new and might not yet be as optimized as native image handling. Between

iterations, the use of `docker system prune -a -f` ensures a clean environment, supporting consistent measurement of pull times.

5.6 Conclusion

To address the need for efficient and portable cloud-native computing, we have analyzed the performance characteristics of WebAssembly (Wasm) containers in comparison with native multi-architecture containers. Benchmarks across three workloads and three architectures (AMD64, ARM64, and RISCV64) revealed that, with highly optimized native images, native containers generally start slightly faster than Wasm containers. However, Wasm containers offer a significant reduction in image size, averaging approximately 15% of their native counterparts (see Figure 6.9). This makes them particularly attractive for environments with constrained storage and memory resources.

A notable finding from our experiments is the benefit of using *containerd* directly instead of Docker for container management. Wasm image pull times were up to 25% faster across all architectures when using containerd, highlighting the efficiency gains of bypassing Docker's additional layer. Our contributions include a comprehensive evaluation of Wasm integration with Docker alternative runtimes, specifically Wasmtime via the runwasi-containerd shim, across three architectures. We also present a detailed performance comparison between optimized native multi-architecture containers and Wasm containers based on key metrics such as startup time, pull time, and image size.

In summary, incorporating Wasm into containerized environments offers a supplementary method to current container paradigms. Rather than substituting traditional containers, Wasm improves them by providing portability, reduced image sizes, and consistency across architectures, thus becoming an asset in cloud-edge computing frameworks.

Chapter 6

The Small Container: Lightweight Compute for Multi-Architecture Cloud-Edge Systems

Contents

6.1	Introduction	100
6.2	Background	103
6.3	Related Work	105
6.4	Methodology	107
6.4.1	Workloads	109
6.4.2	Container Configurations	111
6.4.3	Benchmarking Approach	113
6.4.4	Updated Runtime and Benchmarking Revisions	114
6.5	Results	115
6.5.1	Role of Dockerfile Practices on Container Startup Time	117
6.5.2	Image Size Comparison	118
6.5.3	Host Architecture Sizes vs. OCI Registry Sizes	119
6.5.4	Model of Startup Time	120
6.5.5	Using Containerd to Handle Images	121
6.5.6	Extended Benchmarks with Containerd: <code>benchx</code> and <code>SQLite</code>	123
6.6	Conclusion	127

6.1 Introduction

In contrast to conventional Docker containers, Wasm binaries offer a portable, compact, and secure deployment format. In this chapter, we extend our prior investigation in Chapter 5 by introducing two new

benchmark suites: `benchx` and `sqlite-multiarch` to explore deeper performance characteristics across Wasm and native multi-architecture containers using `containerd`, focusing on three platforms: AMD64, ARM64 (Jetson Orin Nano), and RISC64 (StarFive VisionFive2). We analyze pull time, startup latency, and image size across both native and Wasm runtimes.

Our results show that Wasm containers reduce image size by up to 70% and achieve up to 25% faster cold pull times compared to native containers. With the `sqlite-multiarch` benchmark, we observe Wasm startup overheads compared to native execution, which highlights both performance gaps and opportunities for runtime optimizations. Across all platforms, Wasm containers executed via `containerd` consistently outperformed Docker-based setups [36].

We demonstrate that Wasm runtimes can effectively support realistic data-intensive workloads such as full-text search, JSON parsing, and R-tree spatial queries, while offering improved deployment efficiency and better performance isolation. These findings reaffirm Wasm’s potential as a complementary compute layer for heterogeneous, multi-architecture cloud-edge deployments.

We examine the practical performance and usability implications of using Wasm in containerized environments. We utilize Wasmtime as the runtime environment and compare its behavior against traditional Linux containers. Our work is motivated by the growing diversity of hardware platforms from X86-based servers to ARM-powered edge nodes within the cloud-edge continuum, and also by an experimental interface for general-purpose acceleration with WebAssembly. As this trend intensifies, developers require tools that facilitate deployment across multiple architectures without significant overhead.

We have conducted an in-depth performance evaluation of Wasm containers and multi-architecture Docker containers using four different workloads and platforms. Our experiments employ Wasmtime¹, an open-source runtime developed under the Bytecode Alliance², to assess critical metrics such as image size, pull time, and startup latency. Although Wasm and multi-architecture containers offer significant benefits, challenges related to performance consistency and ecosystem maturity remain.

¹<https://github.com/bytecodealliance/wasmtime>

²<https://bytecodealliance.org/>

In serverless and edge environments, where fast startup and minimal resource usage are essential, the trade-offs between Wasm and native containers become particularly relevant. Native containers excel in raw performance but often result in larger image sizes and less portability. Wasm containers, while smaller and more portable, may incur runtime performance overheads. Multi-architecture containers also introduce additional management complexity and varying execution characteristics across platforms.

Despite Wasm’s strengths in portability and sandboxed isolation, prior work has either focused on microbenchmarks or evaluated only cold-start metrics [68, 16]. Furthermore, evaluations are typically based on Docker-based setups that obscure runtime and network-layer latency.

To address these gaps, we design and execute a suite of experiments that compare Wasm and native containers under consistent, optimized conditions. By using `ctr` directly, we isolate raw container performance and capture more accurate timing for pull and startup events.

Our findings indicate that Wasm containers maintain significantly smaller image sizes (2–8 MB vs. 8–30 MB native) and deliver up to 25% faster cold pull times, particularly on ARM64 platforms using `containerd`. These differences are critical in bandwidth-constrained and latency-sensitive deployments such as edge clusters or IoT gateways.

In addition to revisiting prior workloads, we introduce a new synthetic benchmark, `benchx`, designed to explore container behavior under compute and memory pressure. Written in C++, `benchx` includes a mix of matrix multiplication, FFT, Mandelbrot fractal generation, convolution blur, sorting, and memory-intensive operations. This benchmark evaluates the practical runtime and distribution characteristics of containers across both server-class and edge-class devices.

We further extend our benchmark suite with `sqlite-multiarch`, a new SQLite-based workload compiled for native and Wasm across AMD64, ARM64, and RISC-V. This benchmark simulates realistic data-centric tasks, including full-text search, spatial queries, JSON processing, and sorting over large static datasets. It enables deeper analysis of I/O, cold-start behavior, and runtime consistency across architectures.

We also expand our hardware evaluation to include the Nvidia Jetson Orin Nano, which features an ARM Cortex-A78AE-based platform

with heterogeneous memory and compute characteristics. Alongside traditional AMD64 (AWS EC2) and RISC-V (StarFive VisionFive2) systems, this enables a realistic view of Wasm’s cross-platform behavior in heterogeneous environments.

The core contributions of this chapter are summarized below:

1. We present two new benchmarks, `benchx` and `sqlite-multiarch`, combining multiple compute, I/O, and data-processing patterns to evaluate container behavior in multi-architecture environments.
2. We conduct an updated cross-platform performance analysis using the latest `runwasi-containerd` shim (v0.6.0), showcasing Wasm’s improved performance characteristics in cold-start, pull time, and image compactness.
3. We extend architectural coverage to include ARM Cortex-A78AE (Jetson Orin Nano), providing insights into Wasm’s role in next-generation edge devices.
4. We demonstrate that direct usage of `ctr` in `containerd` exposes a more accurate view of performance trade-offs than Docker-based orchestration, especially for pull latency and runtime initialization.

6.2 Background

Docker and WebAssembly (Wasm) are two pivotal technologies that have reshaped the landscape of software deployment and execution. Docker enables bundling applications with their entire dependency stack into portable units called containers, which can be reliably executed on any system matching the architecture for which the container was built. In contrast, WebAssembly is a compact, binary instruction format that acts as a compilation target for languages such as Rust, C/C++, Go, JavaScript, and C#, enabling cross-platform execution.

Docker containers are typically retrieved from remote repositories known as *container registries*, with Docker Hub³ being the most prominent example. During deployment, a container runtime pulls the specified image if it is not already available locally, and then executes it within a sandboxed, isolated environment.

³<https://hub.docker.com/>

Wasm binaries are executed by a Wasm runtime in a multi-step process involving decoding, validation, and finally execution. These steps ensure both safety and performance. Execution can be performed either through interpretation or ahead-of-time (AOT) or just-in-time (JIT) compilation. Advanced runtimes implement tiered compilation, where frequently executed (“hot”) code paths receive further optimization during execution. Wasmtime, the runtime utilized in our work, performs a single-pass compilation step before execution begins, resulting in efficient and secure execution.

Cross-platform compatibility is a key differentiator between WebAssembly (Wasm) containers and traditional Docker images, largely due to the underlying design of each technology. Wasm modules are compiled into architecture-agnostic binaries. This means a single Wasm image can execute unmodified across different CPU architectures and even inside modern web browsers, provided that a compatible runtime such as Wasmtime, WasmEdge, or Wasmer is available. There is no need for recompilation when targeting diverse platforms, significantly reducing complexity for developers. This portability makes Wasm containers particularly suitable for heterogeneous deployments, including cloud-edge topologies and IoT scenarios. The “write-once, run-anywhere” paradigm simplifies operational workflows and accelerates deployment pipelines in multi-architecture environments.

In contrast, Docker containers are tightly coupled to the OS and CPU architecture they were built for. Each supported architecture such as x86_64 or ARM64 requires a dedicated container image. While tools like Docker Buildx and manifest lists have improved multi-architecture support, these approaches introduce additional build complexity, larger storage footprints, and longer CI/CD pipelines. Moreover, orchestration tools like Docker Compose or Kubernetes must correctly identify the host platform and pull the appropriate image variant to ensure compatibility. However, Docker containers do offer rich system integration, as they can package complete user-space environments with native binaries and full access to host system APIs.

In practice, Wasm containers offer compelling advantages in terms of agility, portability, image size, and security. However, they may face limitations when it comes to broader language support, deep system integration, and peak raw performance. Docker containers remain preferable for many cloud-native and stateful workloads due to their maturity and

full access to host system features. That said, hybrid containerization strategies where Wasm and Docker are used side-by-side are becoming increasingly common, especially in edge, cross-platform, and serverless computing environments. This fusion creates a compelling model for scalable and secure application deployment across heterogeneous infrastructures.

6.3 Related Work

The emergence of the cloud-edge continuum has led to increasingly heterogeneous infrastructure environments that span centralized data centers, on-premises servers, mobile edge nodes, and end-user devices [21]. This broad distribution of compute resources presents new challenges for software portability and runtime performance. Within this context, [43] explore the effects of serverless computing across the cloud-to-edge spectrum, emphasizing latency bottlenecks introduced by traditional container and microVM-based deployments. They propose WebAssembly as a lightweight alternative capable of reducing cold-start times in serverless contexts.

Prior research has examined Wasm’s role in web development and its potential in edge deployments, but comprehensive evaluations of Wasm versus native containers across multiple CPU architectures for cloud-native applications remain limited. To address the challenges introduced by resource heterogeneity in edge computing environments, [48] has explored the joint use of WebAssembly and container-based technologies for microservice deployment. The study formulates the problem of deploying dependent microservices using a mix of Wasm and container runtimes, considering both deployment site selection and runtime consolidation strategy. [97] presents SpecTec, a domain-specific language and toolchain designed to streamline the formal specification of WebAssembly by serving as a single source of truth for generating prose, formal semantics, and interpreters.

Another study, [99] introduces Wapplique, the first mutation-based fuzzing tool tailored for Wasm runtimes. By applying code fragments from real-world programs and ensuring syntactic and semantic validity through advanced analysis, it generates diverse and valid test cases. Similarly, [81] proposes WaDec, the first approach to decompiling We-

WebAssembly using a fine-tuned large language model (LLM) trained on a specialized wat-to-c dataset. WaDec significantly improves code readability and recompilability over traditional decompilers, achieving higher performance across multiple metrics such as AST similarity, code inflation, and execution consistency. An empirical study by [92] analyzes 385 GitHub issues and 354 Stack Overflow posts related to WebAssembly, resulting in taxonomies of 120 issue types and 278 root causes. These taxonomies provide valuable insights into common developer challenges and can guide future improvements in Wasm tooling and documentation.

The cross-platform capabilities and secure execution model of WebAssembly have attracted attention in recent work. For instance, [12] evaluated Wasm performance at the edge, while [13] focused on Wasm's startup latency for serverless workloads. Docker's performance, by contrast, has been studied extensively in [11], though limited to x86 systems and without direct comparison to WebAssembly.

Wasm's role in improving system constraints, including performance variability, hardware availability, and networking overhead, has also been explored. Benchmarking efforts such as those in [35] employ the Sightglass suite to measure Wasmtime's performance across architectures.

In the domain of high-performance computing, [7] present MPIWasm, which facilitates the distribution of MPI-based HPC workloads using Wasm. Their findings show reduced binary sizes and comparable execution speeds to native counterparts. A hardware-accelerated toolchain for WebAssembly was proposed by [14] that enhances memory safety by leveraging Arm's Memory Tagging Extension (MTE) and Pointer Authentication (PAC). It supports unmodified applications and provides spatial and temporal memory safety with minimal performance overhead, while also improving sandboxing efficiency. Serverless-oriented evaluations are provided in [4], comparing several Wasm runtimes including Wasmtime, Lucet, and Wasmer on platforms such as AWS Lambda and OpenWhisk. Other benchmark-driven evaluations, including [84] and [91], rely on PolyBench for performance testing. However, PolyBench lacks coverage of broader non-web application domains.

General-purpose scripting environments using WebAssembly were examined in [26], though the experiments were constrained to specific hardware setups, limiting generalizability. Comparative evaluations of

Wasm and native performance on ARM-based systems were conducted in [54], analyzing a range of application types. Meanwhile, [34] surveyed the evolution of Wasm in cloud environments and emphasized the demand for platform-independent deployment strategies as cloud computing shifts closer to the edge.

Extending the benchmarking landscape, [90] present WaBench, a suite aggregating multiple benchmark tools and full-scale applications to systematically assess five leading Wasm runtimes. Additionally, [32] conduct a runtime-level comparison of Wasmtime and Wasmer, revealing insights into their suitability for heterogeneous cloud-to-edge application models.

Given the growing diversity in compute architectures, developing applications that seamlessly operate across platforms remains a challenge. Research efforts such as [53, 62, 64] investigate deployment techniques for heterogeneous environments. A measurement study [50] systematically compared Wasm-based container runtimes (e.g., using Wasmtime with WASI) against Docker containers and standalone Wasm runtimes. Contrary to expectations, the findings revealed that Wasm containers often do not outperform Docker containers and, in some cases, suffer from additional overhead. The root causes were attributed to both containerization-related overheads and intrinsic limitations of current Wasm virtual machines and the WASI interface. Despite this, the domain of multiarchitecture containerization especially involving WebAssembly remains underexplored. Our study in [33] has explored the viability of Wasm as a lightweight, secure alternative to traditional multi-platform container technologies, highlighting the potential of combining WebAssembly with containerization strategies to simplify and optimize cross-platform application deployment.

6.4 Methodology

Understanding Docker’s evolving support for WebAssembly begins with its internal architecture. Docker is layered upon `containerd`, a high-level container runtime that manages the container lifecycle. `containerd` in turn relies on low-level runtimes such as `runc`, which interfaces directly with Linux kernel features to execute standard OCI containers. Integration with alternative runtimes including those for Wasm

is achieved through a runtime shim that bridges `containerd` with the chosen execution engine.

This modular runtime architecture enables developers to embed WebAssembly runtimes, such as Wasmtime, WasmEdge, or Spin, within Docker’s workflow. As a result, WebAssembly applications can be containerized and managed through the same tooling and orchestration systems used for traditional containers, enhancing portability and operational consistency.

To evaluate the performance characteristics of Wasm containers compared to traditional native containers, we conducted benchmarking across three distinct CPU architectures:

- **amd64**: Experiments were executed on a bare-metal server hosted on AWS running a 64-bit x86 architecture (Linux).
- **arm64**: Two platforms were evaluated: Nvidia Jetson Nano (Cortex-A57) and Jetson Orin Nano (Cortex-A78AE), the latter representing a newer generation of edge hardware.
- **riscv64**: Additional evaluations were conducted on a StarFive VisionFive2 board running a RISC-V 64-bit processor.

This diverse architectural setup enabled a comprehensive analysis of runtime behavior and deployment characteristics under varying hardware capabilities. Figure 6.1, inspired by Chapter 5, presents the overall experimental design and workflow used in our evaluation.

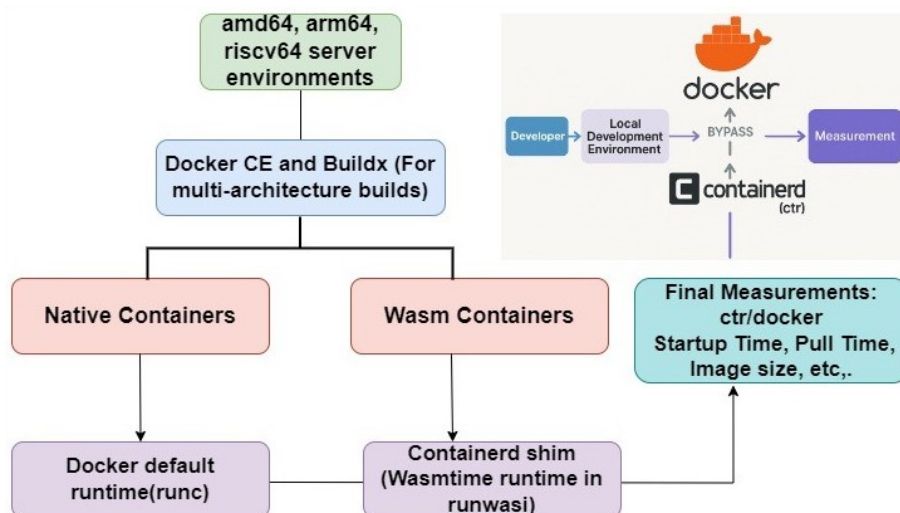


Figure 6.1: Native and wasm container workflows, expanded from 5.

6.4.1 Workloads

To evaluate the performance and characteristics of both native and WebAssembly-based containers, we designed four distinct workloads targeting compute, I/O, and image processing tasks:

- **Matrix Multiplication:** This workload performs matrix multiplication using implementations in Rust, C++, and TinyGo, compiled both to native binaries and to WebAssembly. It primarily evaluates computational performance under intensive arithmetic operations.
- **Sorting Task:** A file I/O-heavy workload that reads an input text file, performs sorting using either batch or iterative techniques, and writes the sorted output. Implementations are available in Rust, C++, and TinyGo, with Wasm equivalents. This task focuses on assessing memory access patterns and I/O performance.
- **Face Detection:** Developed in C++, this workload was executed natively across all platforms using the containerd runtime directly, bypassing Docker to streamline execution.
- **Synthetic Benchmark (benchx):** Introduced in this extended version, **benchx** is a multi-kernel benchmark implemented in C++ that combines matrix multiplication, Mandelbrot set generation, 1D FFTs, convolution blurs, sorting, and memory fill/copy loops.
- **SQLite-Multiarch Benchmark:** A new workload introduced in this paper, based on a comprehensive SQLite build system targeting **amd64**, **arm64**, **riscv64**, and WebAssembly. The application is implemented in C and compiled with full SQLite features enabled, including JSON1, FTS5, R-Tree indexing, math functions, and session support. It operates over extensive synthetic datasets, including 50,000 mathematical constants, 10,000 prime numbers, and a large dictionary corpus. This benchmark exercises both CPU-bound queries and I/O operations over SQLite’s B-tree storage engine. The containerized versions are evaluated using **ctr** for pull time and runtime execution profiling.

These workloads were deployed across four distinct hardware configurations representing different architectural families. The platforms used in the experiments are summarized in Table 6.1.

Table 6.1: Hardware Architectures Used in Experiments.

Platform	CPU Model	Clock Frequency	Threads per Core	Cores per Socket	Caches
x86_64 AWS c5.metal	Intel Xeon 8275CL	3 GHz	2	24	L1d: 32 KiB, L1i: 32 KiB, L2: 1 MiB L3: 71.5 MiB
Nvidia Jetson Nano	ARM Cortex-A57	1.5 GHz	2	4	L1d: 32 KiB, L1i: 48 KiB L2: 2048 KiB
Nvidia Jetson Orin Nano	ARM Cortex-A78AE	1.5 GHz	1	6	L1: 64 KiB/core L2: 1.5 MiB shared, L3: 4 MiB shared
StarFive VisionFive2	SiFive U74 Core	1 GHz	1	4	L1d: 32 KiB, L1i: 32 KiB L2: 2048 KiB

To illustrate the embedded platforms employed in this extended evaluation, Figures 6.3, 6.2, and 6.4 depict the devices used for the ARM and RISC-V experiments. In addition to the Jetson Nano and VisionFive2 boards used in Chapter 4, the Jetson Orin Nano was included to represent a newer generation of embedded GPU-accelerated hardware, designed for high-performance edge inference and compute workloads.

The Nvidia Jetson Orin Nano, shown in Figure 6.2, is equipped with an ARM Cortex-A78AE CPU and an integrated Ampere-based GPU. Compared to the Jetson Nano, it delivers significantly higher computational throughput, improved energy efficiency, and advanced AI acceleration capabilities. This device bridges the gap between lightweight edge computing and data center-grade performance, making it particularly suitable for testing WebAssembly workloads that benefit from parallelism and heterogeneous compute resources.



Figure 6.2: NVIDIA Jetson Orin Nano board (Cortex-A78AE)

For comparison, the previously used Jetson Nano and StarFive VisionFive2 setups are shown again in Figures 6.3 and 6.4, providing visual reference for the embedded testbeds used throughout this work.

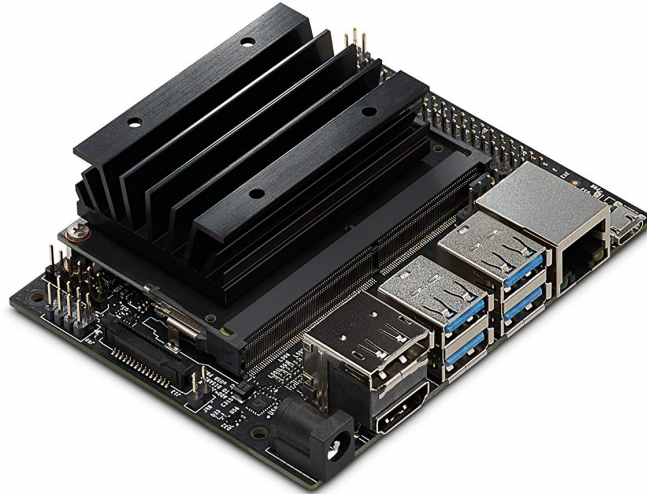


Figure 6.3: NVIDIA Jetson Nano embedded board (Cortex-A57).



Figure 6.4: StarFive VisionFive2 RISC-V board (SiFive U74).

6.4.2 Container Configurations

Each workload was packaged into both native and WebAssembly containers, with consistent compiler optimizations to ensure fair comparisons. Native binaries were built using:

- `--release` flag for Rust to enable high-performance compilation,
- `-O3` for C++ to achieve aggressive optimizations,
- `-opt=2` for TinyGo, which is the highest available optimization level.

Although TinyGo's optimizations are not as aggressive as those in Rust or C++, the best available level was chosen for consistency.

Native Containers: Native binaries for each language were compiled per target architecture. Docker’s multi-platform build functionality (`--platform`) was used for AMD64 and ARM64 images. For RISC-V, a different strategy was required due to the lack of a shared base image across all three platforms. Nevertheless, the final images were structured as multi-architecture manifests, enabling architecture-specific execution.

The Dockerfiles adopted a multi-stage build approach. In the first stage, full-featured base images, such as Alpine, were used for compilation. The final runtime stage used the `scratch` base to include only the statically linked application binary and essential runtime libraries: this reduced image size, minimized dependencies, and improved security.

Wasm Containers: WebAssembly versions of each workload were produced from Rust, TinyGo, and C++ source code and executed using the Wasmtime runtime via Docker. We employed the `runwasi`⁴ project, which integrates Wasmtime into the containerd ecosystem using a runtime shim. This allowed Wasm containers to be run using the `docker run` command with `containerd` managing the lifecycle. The architecture for `runwasi` motivated by KubeCon NA⁵, is shown in Figure 6.5.

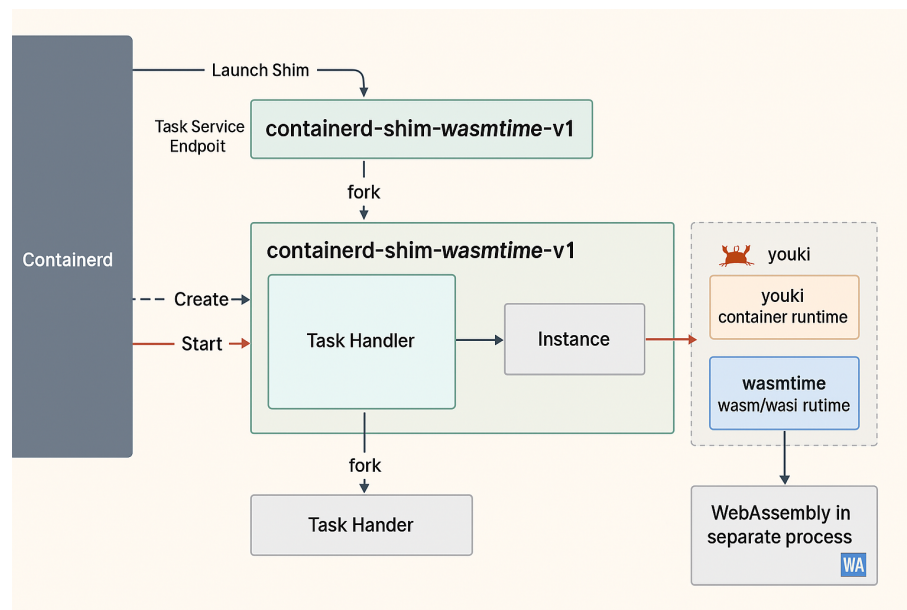


Figure 6.5: Runwasi Architecture

Dockerfiles for Wasm containers were minimal, using `scratch` as the base image to package only the compiled Wasm binary. This approach

⁴<https://github.com/containerd/runwasi>

⁵<https://github.com/containerd/runwasi/pull/743>

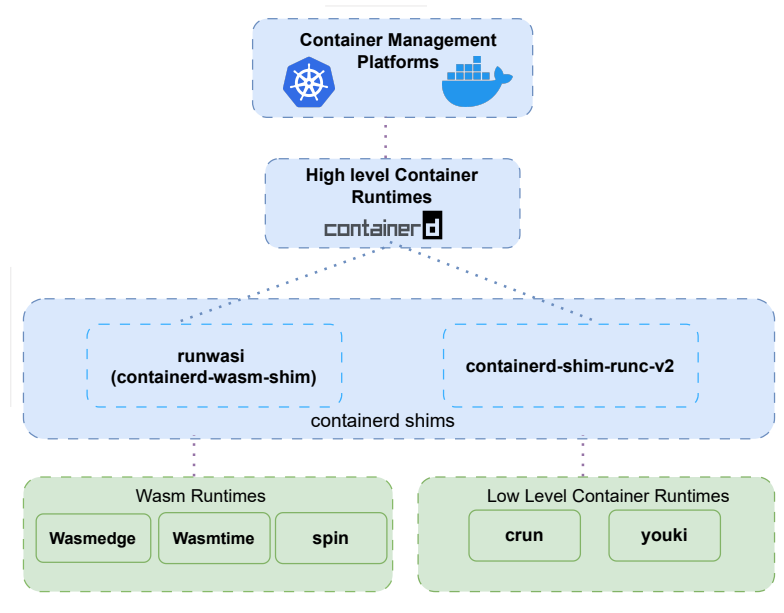


Figure 6.6: Containerd shim integration with Wasmtime runtime adapted from Chapter 5.

ensured extremely compact images, enhanced runtime isolation, and reduced attack surface. Execution was handled using the flags `--platform wasm` and `--runtime=io.containerd.wasmtime.v1`.

Figure 6.6, adapted from prior work as discussed in Chapter 5, illustrates the high-level execution flow for running a container with Wasmtime via Docker. Upon invocation via `docker run/ctr run`, containerd checks the local image cache and, if necessary, pulls the image from the configured registry (Docker Hub, in this case). The image is then executed using the appropriate runtime.

6.4.3 Benchmarking Approach

Our benchmarking strategy focused on quantifying two critical performance metrics: container startup time and image pull duration. A custom automation script was developed to detect the host architecture and select the appropriate image for either native or WebAssembly-based containers.

To measure container startup time, we instrumented each workload to record a timestamp at the beginning of the `main` function. This allowed us to accurately determine when the application began executing. Image pull time was measured externally by recording the time between

initiating the `docker run` command and the completion of the image retrieval from the registry.

High-resolution timestamps were captured using the Linux `date` command. Each run began with a fresh environment by force-removing any existing image from the local cache, ensuring a cold-start scenario. These procedures helped eliminate caching effects and emphasized true startup behavior.

The measured durations were categorized as follows:

- **Pull Time:** The time from issuing the container run command until the image is downloaded from the registry (if not cached).
- **Startup Time:** The time from the container's execution start until the workload's `main` function begins.

Our benchmarking script supported both fresh-pull and cached-image modes, allowing for comparative analysis across deployment conditions⁶. Each container was executed on both AMD64 and ARM64 hosts, and execution times were logged and analyzed.

The AWS instance running on Intel Xeon 8275CL serves as a representative of high-performance datacenter hardware with advanced out-of-order execution, branch prediction, and multi-level caching. In contrast, the Jetson Nano and VisionFive2 boards represent resource constrained embedded systems more commonly found in edge computing environments.

6.4.4 Updated Runtime and Benchmarking Revisions

As part of our extended evaluation, we re-executed the matrix multiplication benchmark using the latest version of the `runwasi` runtime, specifically `containerd-shim-wasmtime v0.6.0`. This release introduces several performance and efficiency improvements that significantly enhance the viability of WebAssembly for production-grade, containerized workloads.

According to recent updates from the containerd slack community⁷, the `v0.6.0` shim includes:

⁶https://github.com/sangeeta1998/benchmark_sort/

⁷<https://github.com/containerd/runwasi>

- **Memory Optimizations:** Shim memory usage has been reduced by approximately 20%, making it more suitable for edge and resource-constrained environments.
- **Execution Throughput:** Task execution throughput has increased to approximately 330 tasks per second, compared to only 80 tasks/sec achieved by a previously optimized distroless Wasmtime container image. This represents more than a 3x performance improvement.
- **Image Size Reduction:** The new `runwasi` shim image is extremely compact, with a minimal footprint as small as 2.8 KiB, compared to 9.9 MiB for the distroless Wasmtime container.
- **Stress-Testing Tools:** Built-in CLI support for stress-testing with or without a full containerd backend enables better profiling of high-concurrency scenarios.

These improvements make the new `runwasi` shim more performant, scalable, and lightweight, which is crucial for serverless and microservice-based deployment scenarios. To validate these enhancements, we re-ran the matrix multiplication benchmark across the same AMD64 platform using the updated `containerd-shim-wasmtime v0.6.0`. We preserved the same measurement techniques and cold-start conditions to ensure consistency with our earlier experiments.

6.5 Results

To evaluate startup times for native and Wasm containers in the chosen workloads, we adopted containerization best practices, including multi-stage Dockerfiles and minimal base images. Figure 6.7a presents the results for the Matrix Multiplication workload, while Figure 6.7b shows corresponding results for the Sorting benchmark, which is in line with Chapter 5.

Overall, the results slightly favor native containers. While WebAssembly containers are often associated with smaller image sizes and lightweight execution, the measured differences in both image size and startup time between native and Wasm containers were relatively modest. This observation aligns with our prior findings from [32], which highlights WebAssembly’s strong performance characteristics albeit with some inherent overhead compared to native binaries.

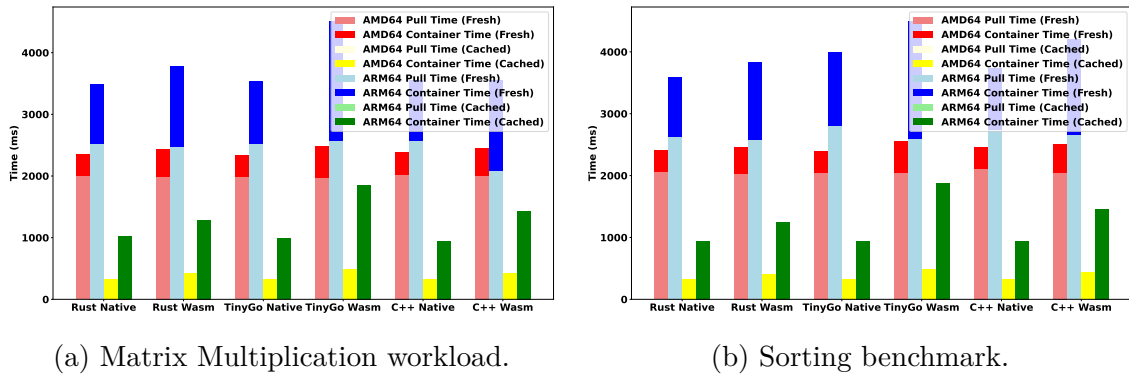


Figure 6.7: Startup Times (image pull and container time) for two workloads: (a) Matrix Multiplication and (b) Sorting, updated from Chapter 5.

Figures 6.7a and 6.7b present four types of measurements as discussed in Chapter 5: the image pull time and container startup time (shown in red and blue), and the same metrics when the image is locally cached (shown in yellow and green, respectively).

For the AMD64 architecture, we observe that image pull times are nearly identical across native and Wasm containers, and consistent across Rust, TinyGo, and C++. On ARM64, however, there are minor deviations. For example, the C++ Wasm image for the matrix multiplication workload is pulled slightly faster than its native counterpart. This can be attributed to the significant size difference in the container images, as illustrated in Figure 6.9. A similar trend is observed in the sorting workload with TinyGo, where the Wasm version benefits from a smaller image size.

When examining startup times following a successful image pull, native containers consistently outperform their Wasm counterparts. This is expected, as Wasm containers incur an additional overhead from just-in-time (JIT) compilation of Wasm bytecode before execution. The performance impact of this step varies by language; in particular, TinyGo exhibits greater startup delays than Rust or C++, possibly indicating that TinyGo’s WebAssembly backend is less optimized or mature.

These results demonstrate that while WebAssembly offers compelling benefits in terms of portability and sandboxed execution, native containers currently retain a slight edge in startup performance especially when cold-start conditions or runtime compilation steps are involved.

6.5.1 Role of Dockerfile Practices on Container Startup Time

To better align with scenarios presented in related work where Wasm often demonstrates significantly lower startup times, we conducted a supplementary experiment focused on the matrix multiplication workload. In contrast to the prior section, native containers in this experiment were built without `scratch` base images or multistage Dockerfile optimizations.

Under these non-optimized conditions, both image sizes and startup times were substantially affected, resulting in more pronounced differences between native and Wasm container performance. For example, the Alpine-based C++ image yielded the smallest native container at 433 MB due to its lightweight dependencies. In contrast, Rust produced a larger image (927 MB) because of its extensive toolchain, even in Alpine. TinyGo resulted in the largest image (1.55 GB), since the `tinygo/tinygo:latest` base image was used, as Alpine does not natively support TinyGo on ARM64.

Figure 6.8 presents the updated performance comparison between native and Wasm containers running the matrix multiplication workload on the `x86_64` architecture.

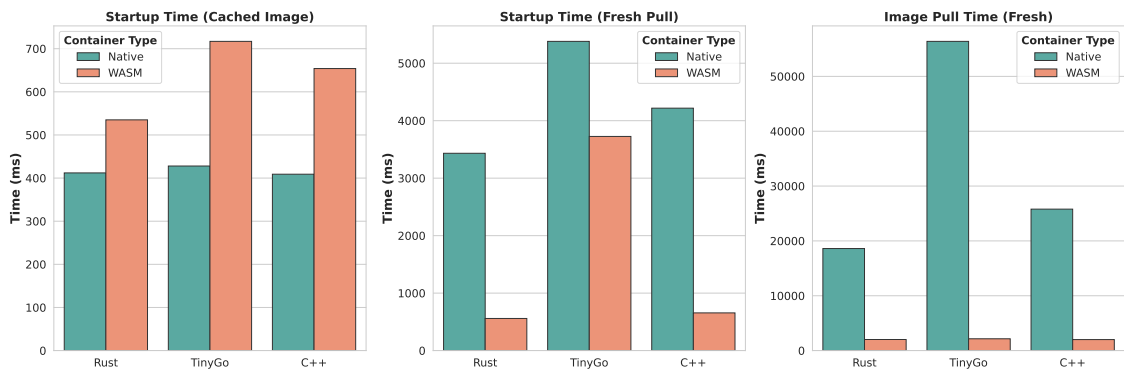


Figure 6.8: Pull and Startup Time: Without Best Practices

The data reveals that Wasm containers consistently achieve significantly faster image pull times across all languages. For example, Wasm Rust completed a fresh image pull in 2023 ms compared to 18610 ms for native Rust, an approximately 89% improvement. Similarly, Wasm TinyGo and Wasm C++ reduced pull times by over 90% relative to their native counterparts.

Regarding fresh startup times (cold starts), Wasm Rust also outper-

forms native Rust, starting in 559 ms versus 3433 ms natively, a roughly 84% speedup. Wasm C++ shows a similar pattern with 656 ms startup compared to 4218 ms natively. However, Wasm TinyGo’s fresh startup time (3726 ms) is slower than its native counterpart (5380 ms), indicating some runtime overhead or optimization opportunity.

For cached startups (warm starts), the differences narrow but Wasm containers remain competitive. Native Rust starts in 412 ms versus 535 ms for Wasm Rust, while native C++ starts faster (409 ms) than Wasm C++ (654 ms). TinyGo’s cached startup times are comparable between native (428 ms) and Wasm (717 ms).

Overall, these results highlight Wasm’s advantage in drastically reducing image pull times and delivering faster or comparable startup times, especially for Rust and C++. The performance gap for TinyGo suggests language-specific factors influence startup efficiency in Wasm. This updated evaluation confirms the potential of Wasm containers for latency-sensitive and short-lived workloads on x86_64, particularly when Dockerfile optimizations are not applied.

Unless otherwise noted, all subsequent experiments in this paper use optimized native containers to provide a balanced performance baseline.

6.5.2 Image Size Comparison

Wasm containers consistently demonstrate smaller image sizes than their native counterparts, making them especially suitable for bandwidth-limited or resource-constrained environments. Figure 6.9 illustrates image sizes at the host architecture level after pulling and local caching. Figure 6.10 presents the corresponding compressed image sizes as stored in the OCI registry (Docker Hub).

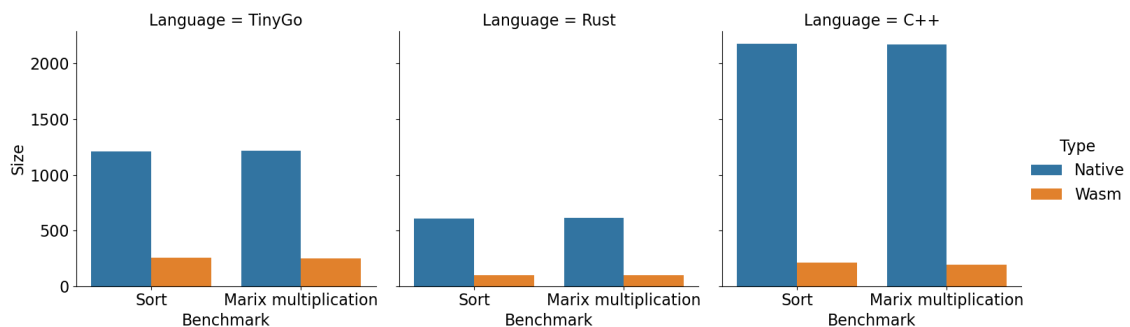


Figure 6.9: Host-based Image Sizes Comparison developed from Chapter 5.

Table 6.2: Comparison of Host vs. OCI Image Sizes (KB) and Reduction % across Benchmarks.

Language	Sorting Benchmark			Matrix Multiplication		
	Host	OCI	Red. (%)	Host	OCI	Red. (%)
TinyGo Native	1210	373.36	69.2	1220	366.46	69.9
Rust Native	610	259.27	57.5	613	273.00	55.5
C++ Native	2180	746.24	65.8	2170	735.43	66.1
TinyGo WASM	257	100.31	61.0	250	101.26	59.5
Rust WASM	96	40.79	57.5	96.1	42.25	56.1
C++ WASM	212	68.50	67.7	192	60.38	68.6

It is important to note that these values represent images for a single architecture. For multi-architecture support, the effective storage requirements scale linearly with the number of supported architectures, as image sizes are roughly equivalent across platforms.

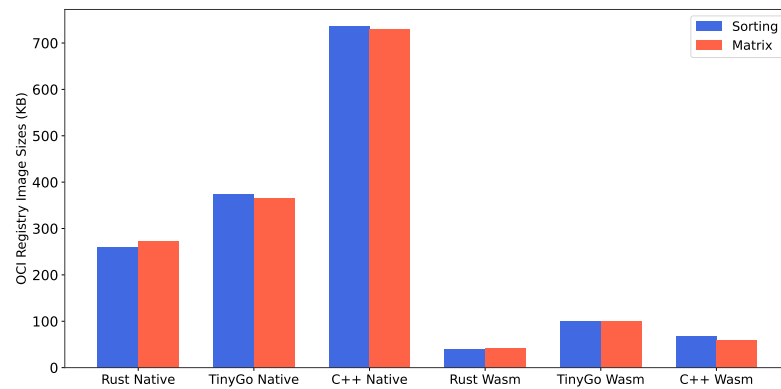


Figure 6.10: Image Sizes from OCI Registry for the AMD64 architecture.

6.5.3 Host Architecture Sizes vs. OCI Registry Sizes

Tables 6.2 summarize the image sizes for both the Sorting and Matrix Multiplication benchmarks, comparing the image sizes at the host architecture (i.e., after download and decompression) with those stored in the OCI registry. OCI registry images are compressed to reduce storage space and network transfer overhead. The reduction percentage between the host size and the registry size is computed using Equation 6.1.

$$\text{Reduction} = \left(1 - \frac{\text{OCI Size}}{\text{Host Size}}\right) \times 100 \quad (6.1)$$

When an image is pulled from the OCI registry, only the compressed

version of its layers is transferred over the network. Decompression is performed locally once all layers are received. This architecture improves efficiency by minimizing data transfer volumes.

Despite the notable differences in image sizes between native and Wasm containers, particularly in their uncompressed forms, we observe limited variation in pull times. This is primarily due to the relatively small overall size of the images used in our benchmarks, and the inherent overhead introduced by the Docker engine and the underlying `containerd` service that manages image downloading and unpacking.

The end-to-end startup process for a Docker container involves several distinct stages:

1. **Image Cache Check:** Docker receives the request and checks the local cache for the specified image.
2. **Remote Registry Request:** If the image is not found locally, Docker sends a request to the registry (e.g., Docker Hub), which responds by initiating layer transfer.
3. **Download and Decompression:** The image layers are downloaded and decompressed on the local host.
4. **Runtime Initialization and Execution:** The decompressed image is loaded into the container runtime, and the container starts executing.

Of these, only steps 3 and, to a lesser extent, step 4 are influenced by image size. Thus, while compressed images reduce network overhead, their impact on total startup time is partially masked by fixed infrastructure-related latencies introduced in the other steps of the container startup sequence.

6.5.4 Model of Startup Time

The steps which depend on image size can be denoted as:

- T_d as the time to download the image.
- T_c as the time to decompress the image.
- T_l as the time to load the image into the runtime.

- S as the size of the image.

A model for the startup time T could be:

$$T = T_d(S) + T_c(S) + T_l(S) \quad (6.2)$$

where:

- $T_d(S)$ is proportional to the image size S .
- $T_c(S)$ is proportional to the image size S .
- $T_l(S)$ is proportional to the image size S .

For a native program, the startup time:

$$T_{\text{native}} = T_d(S) + T_c(S) + T_l(S) \quad (6.3)$$

For a WASM program, the startup time is slightly different due to the runtime environment:

$$T_{\text{wasm}} = T_d(S) + T_c(S) + T_l(S) + T_{\text{runtime}} \quad (6.4)$$

where T_{runtime} is the additional time required to initialize the WASM runtime.

6.5.5 Using Containerd to Handle Images

This section presents the performance evaluation of the face detection benchmark, implemented in C++ with `-O3` optimization. The application was containerized for three native architectures: `amd64`, `arm64`, and `riscv64` along with a WebAssembly variant. The Dockerfiles for `amd64` and `arm64` are identical, while the one for `riscv64` differs only in its choice of base image to support the target architecture. We summarize the resulting image sizes below:

Image Sizes Across Architectures

- `amd64`: 1.65 MB
- `arm64`: 1.55 MB
- `riscv64`: 1.55 MB
- `wasm`: 0.816 MB

The distribution of container images across architectures is also reflected in the Docker Hub registry. As shown in Figure 6.11, native images are typically published for each target architecture (`amd64`, `arm64`, `riscv64`), while a single Wasm image is sufficient to run across multiple platforms. This demonstrates the portability advantage of Wasm images compared to architecture-specific native containers.

```
docker buildx build --platform linux/amd64,linux/arm64,linux/riscv64 -t myimage:multiarch .
```

```
docker buildx build --platform wasi/wasm -t myimage:wasm .
```

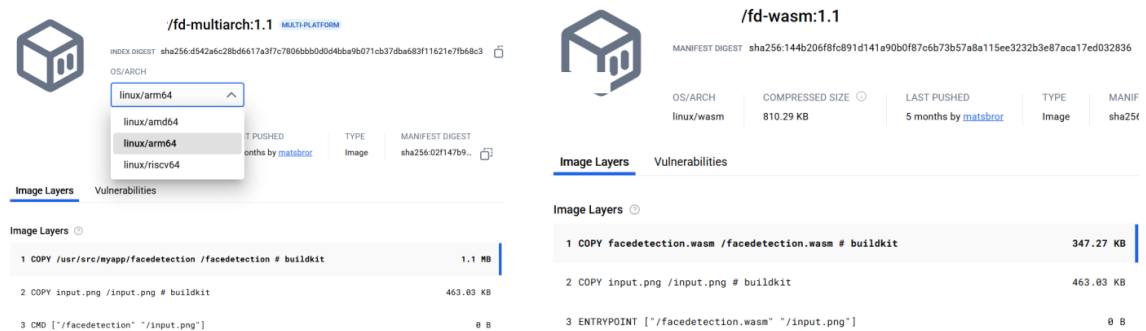


Figure 6.11: Representation of native and Wasm container images in DockerHub

To directly assess image pull performance without Docker’s additional overhead, we used `containerd` via the `ctr` CLI. This allowed for precise benchmarking of image transfer performance across architectures.

The Dockerfiles for all native builds follow a multi-stage structure:

- **Build Stage:** The face detection application is compiled from C++ source files using `clang++` with `-O3` optimizations. The output is a statically linked executable named `facedetection`.
- **Runtime Stage:** A minimal `scratch` base image is used to reduce image size. The compiled `facedetection` binary and a representative input image are copied into the runtime image.

6.5.5.1 Image Pull Times

We measured the average image pull times over 10 iterations, aligning with the findings of Chapter 5, for each architecture using `ctr`, the native CLI for `containerd`. Results are shown in Figure 6.12.

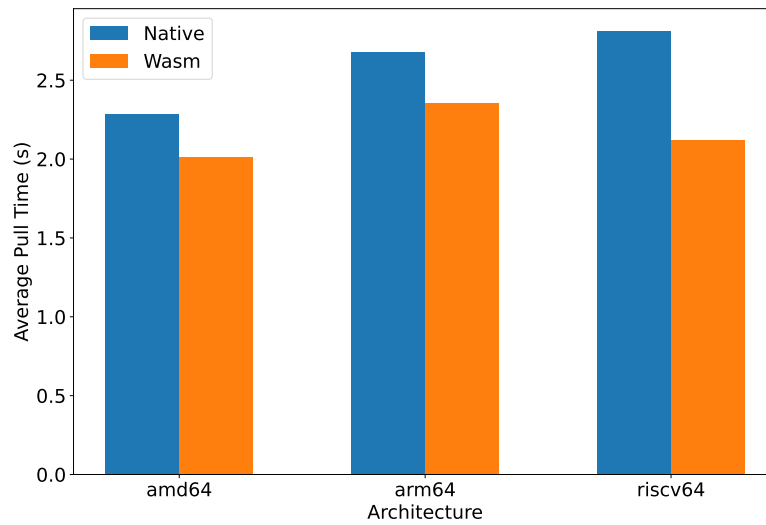


Figure 6.12: Average Pull Time using `containerd` across architectures.

The results demonstrate that Wasm containers achieve significantly faster pull times than their native counterparts. This is expected due to their smaller image sizes and reduced complexity. Additionally, using `containerd` directly, rather than through Docker, provides consistently lower overhead and faster performance, isolating the impact of the image layer size more effectively.

These findings highlight key trade-offs between container runtimes and architectures in performance-sensitive contexts. The results reinforce the practical benefits of Wasm in terms of distribution efficiency and suggest that lightweight runtimes like `containerd` are better suited for rapid image deployment in resource-constrained or latency-critical environments.

6.5.6 Extended Benchmarks with Containerd: `benchx` and `SQLite`

To deepen our evaluation of container image performance and broaden the scope of workloads beyond compute kernels, we incorporated two synthetic applications: `benchx`⁸ and our custom-built `sqlite-multiarch`⁹, the latter designed to emulate realistic database workloads in a multi-architecture setting.

⁸<https://github.com/sangeeta1998/benchx>

⁹<https://github.com/matsbror/sqlite-multiarch/>

6.5.6.1 Benchx

Benchx is a synthetic benchmark implemented in C++ that stresses various runtime subsystems by executing the following components:

- **Mandelbrot Generation:** Iterative complex-number computations.
- **Matrix Multiplication:** Dense $N \times N$ double-precision matrix multiplication.
- **Fast Fourier Transform (FFT):** Radix-2 Cooley-Tukey 1D FFT.
- **Image Blur Filter:** A 3×3 convolution kernel.
- **Sorting:** Randomized array sorting via STL's `std::sort`.
- **Large Static Data Access:** Summing over a statically initialized 8MB array.

6.5.6.2 SQLite

Complementing Benchx, `sqlite-multiarch` represents a more realistic data-centric workload. It embeds a feature-rich SQLite build including extensions such as FTS5, JSON1, GEOPOLY, and mathematical functions. The application initializes a large embedded dataset of dictionary words, timestamps, prime numbers, and mathematical constants. The same C source code is compiled into both native binaries (via GCC) and WebAssembly modules (via Clang + WASI SDK), enabling architecture-agnostic deployment.

Across all three architectures, WebAssembly containers showed consistently lower *pull times* due to significantly smaller image sizes. Native containers outperformed Wasm in *execution time*, up to $2.5\times$ faster on AMD64 and ARM64. On RISC-V, however, Wasm performance was more competitive, reflecting architectural immaturity in native RISC-V toolchains. **Total time** varied based on runtime, while native containers were faster overall in compute-heavy scenarios, Wasm containers consistently offered advantages in cold-start and pull efficiency.

Native containers showed clear performance dominance in execution time, with Wasm being 40–50% slower on average. Pull time for Wasm was 20–30% faster. In the ARM64 (Jetson Orin), the Wasm runtime had

longer execution time but smaller images and startup profiles, making it efficient in bandwidth-sensitive environments. In the RISC-V, due to the less optimized native toolchain and runtime stack, Wasm containers were more competitive and sometimes even outperformed native on startup latency.

We also compared Wasm execution using Wasmtime and Spin runtimes. Wasmtime delivered more consistent performance across architectures, while Spin had more variability, especially in execution time on ARM.

This analysis demonstrates that while Wasm containers may not match native performance for compute-heavy workloads like SQLite, they offer strong portability and startup advantages that justify their use in multi-architecture and constrained deployment scenarios. When combined with runtime improvements (e.g., Wasmtime v0.6.0) and containerd-based execution, Wasm continues to close the performance gap for portable data-processing workloads.

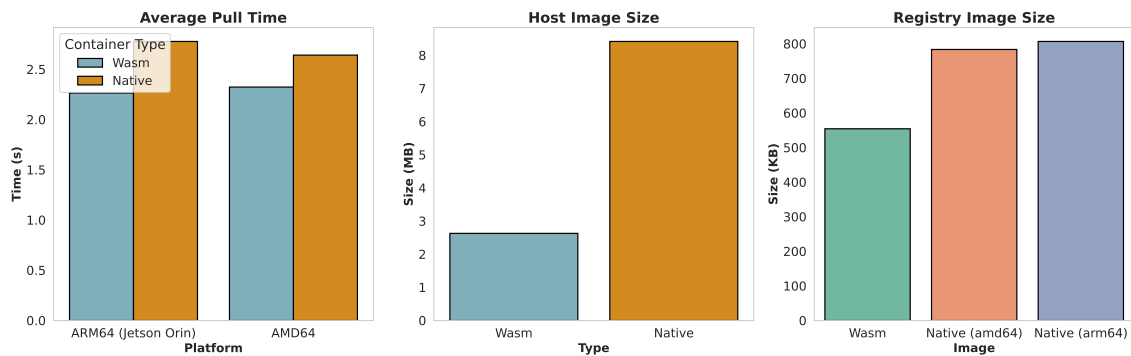


Figure 6.13: Pull time, startup latency, and execution duration for Wasm and Native containers using `ctr`.

These extended results validate all the major claims we made, using empirical measurements to observe consistent trends across platforms, as shown in Figure 6.14 and Table 6.3.

Table 6.3: Architecture-Specific Performance Summary from `Sqlite-multiarch`

Arch	Overhead (%)	Best Runtime	CV (%)	Size Red. (%)
AMD64	183.9	containerd + Native 0.353s	50.2	51.9
ARM64	95.2	containerd + Native 0.547s	39.1	45.7
RISCV64	1755.1	containerd + Native 2.299s	73.9	37.6

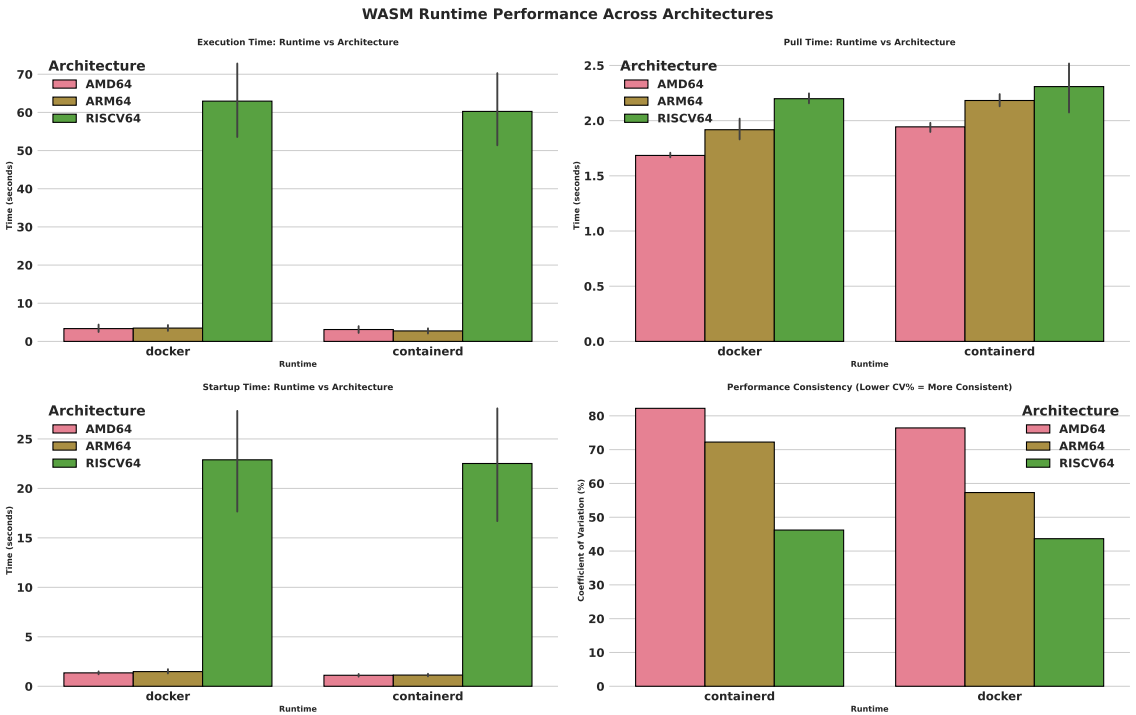


Figure 6.14: Breakdown for `SQLite` benchmark across architectures.

Below, we present the architecture-specific insights from the investigation conducted using the `Sqlite-multiarch` benchmark.

AMD64 Analysis

- **WASM overhead:** 183.9% vs native
- **Best combination:** containerd + Native (0.353s)
- **Performance consistency:** Variable (50.2% CV)
- **Image size:** WASM 51.9% smaller

ARM64 Analysis

- **WASM overhead:** 95.2% vs native
- **Best combination:** containerd + Native (0.547s)
- **Performance consistency:** Variable (39.1% CV)
- **Image size:** WASM 45.7% smaller

RISCV64 Analysis

- **WASM overhead:** 1755.1% vs native
High WASM overhead - consider runtime optimization
- **Best combination:** containerd + Native (2.299s)
- **Performance consistency:** Variable (73.9% CV)
- **Image size:** WASM 37.6% smaller

Cross-Architecture Recommendations

- **Best overall runtime:** containerd (avg: 23.440s)
- **Fastest architecture:** AMD64 (avg: 0.943s)
- **Recommendation:** Consider AMD64 for performance-critical workloads
- **Performance gap:** 49.9x between fastest and slowest

6.6 Conclusion

In response to the growing demand for efficient and portable cloud-native computing, we have conducted an extensive evaluation of WebAssembly (Wasm) containers versus native multi-architecture containers across ARM64, AMD64, and RISCV64 platforms. Our results consistently show that while native containers deliver marginally faster cold start times particularly when compiled with aggressive optimizations (`-O3`), Wasm containers offer considerable advantages in image size and cross-platform deployment efficiency.

Across all workloads and platforms, Wasm images were on average 40%–60% smaller than their native equivalents. This reduction translated to significantly faster image pull times (up to 25%) when using `containerd` directly via `ctr`, avoiding the overhead of higher-level runtimes like Docker. For example, in our synthetic benchmark `benchx`, Wasm images reduced transmission overhead while maintaining fast instantiation. Similarly, with the new `sqlite-multiarch` benchmark, which includes rich datasets and full SQLite features, we evaluated Wasm’s performance across multiple CPU architectures under realistic database workloads.

Our analysis revealed key architectural trends: on `AMD64` and `ARM64`, Wasm containers maintained stable and predictable behavior, with overheads of 95%–180% compared to native. However, on `RISCV64`, performance lagged significantly, with a $17\times$ slowdown underscoring the need for runtime-level optimizations. Despite this, Wasm images on RISC-V were still over 35% smaller, reinforcing their viability in constrained environments where storage or bandwidth is limited.

These findings affirm the practical value of Wasm containers as compact, architecture-agnostic deployment units. By integrating Wasm into the existing container ecosystem through `runwasi-containerd` and `Wasmtime`, we demonstrated that Wasm can complement traditional containers especially in scenarios requiring fast provisioning, small footprints, and multi-architecture support.

Beyond traditional compute workloads, we showed that Wasm is now capable of handling complex, data-heavy applications like SQLite with extensive extensions (FTS5, R-tree, JSON, etc.). The `sqlite-multiarch` benchmark enabled detailed measurement of real-world runtime characteristics, including cold startup behavior, pull times, and architectural bottlenecks.

In summary, WebAssembly is maturing into a compelling compute target not just for browsers or edge functions, but for general-purpose portable containers. While native containers retain an edge in raw performance, Wasm-based containers excel in size, portability, and simplicity. For developers targeting heterogeneous systems, Wasm provides a viable pathway to unified multi-platform deployment.

Chapter 7

Wasm for GPU acceleration: Are we there yet?

Contents

7.1	Introduction	130
7.2	Background and Motivation	131
7.2.1	WASI-NN: Neural Network Acceleration	133
7.2.2	WASI-GFX: Graphics and Compute Acceleration	135
7.2.3	Comparison of GPU Acceleration Approaches	136
7.2.4	WebAssembly Host Functions and WIT Integration	136
7.3	Design	138
7.3.1	APIs	138
7.3.2	Resource Model	139
7.4	Implementation Details	139
7.4.1	wasi-dataframe	139
7.4.2	wasi-accelerator	139
7.5	Experimental Methodology	140
7.5.1	Benchmarks	141
7.5.2	Reproducible Commands	141
7.5.3	Example Client Implementation	141
7.6	Conclusion	142

WebAssembly (Wasm) is rapidly evolving from a browser sandbox into a portable systems substrate for general-purpose and data-centric workloads. While the Component Model and the WebAssembly System Interface (WASI) standardize capability-based access to system resources, the ecosystem still lacks portable abstractions for (i) columnar data analytics and (ii) heterogeneous accelerators.

This chapter introduces two prototype host-side services for Wasm-time components. The first, a `wasi:accelerator` host allocator, demonstrates a handle-based host buffer model and host-side matrix multiplication. The second, `wasi:dataframe`, maps a subset of Polars' dataframe operations into a host service callable from components. We implemented the WIT interfaces, developed host Rust implementations and test harnesses, and resolved key integration issues such as data representation between guest and host and type casting for numeric comparisons.

Future work includes a GPU-backed accelerator, more expressive aggregation/filtering semantics, and a comprehensive performance evaluation to quantify the trade-offs of host-offload strategies.

7.1 Introduction

WebAssembly has emerged as a powerful technology for executing code across various platforms with near-native performance. Initially designed for web browsers, Wasm is now being adopted in diverse environments, including cloud computing, edge devices, and embedded systems. One of the key factors driving this adoption is the ability to leverage hardware accelerators to enhance performance. Graphics Processing Units (GPUs) are widely used for parallel processing tasks due to their high computational power and efficiency. In the context of WebAssembly, GPUs can significantly accelerate tasks such as machine learning inference, image processing, and scientific computations. Integrating GPUs with WebAssembly typically involves using WebAssembly runtimes that support GPU acceleration. Runtimes like WasmEdge and Wasmer can be configured to utilize GPUs through plugins or extensions. GPUs are ideal for running machine learning models due to their ability to handle large-scale parallel computations. WebAssembly modules can utilize frameworks such as TensorFlow.js or ONNX.js, which are specifically optimized for GPU execution. Applications requiring heavy numerical computations, such as simulations and data analysis, benefit from GPU acceleration. WebAssembly modules could leverage libraries like cuBLAS and cuFFT for efficient computation. Tasks such as real-time image enhancement, video transcoding, and augmented reality can be accelerated using GPUs. WebAssembly modules can use WebGL or WebGPU APIs to access GPU capabilities. Although GPU

acceleration offers significant performance benefits, there are challenges related to interoperability, security, and resource management. Future research should focus on standardizing GPU access in WebAssembly and improving the security model to prevent unauthorized access to GPU resources. WebAssembly is increasingly used for portable components that need access to host resources. We present two complementary host providers implemented for Wasmtime: **wasi-dataframe**, a Polars-backed dataframe API exposed to guests via WIT; and **wasi-accelerator**, a generic accelerator interface exposing buffer allocation, host reads/writes, and matrix-multiply primitives. Together they enable guests to offload numeric workloads to the host while using familiar dataframe idioms.

7.2 Background and Motivation

The integration of hardware accelerators with WebAssembly has garnered significant attention, demonstrating potential for enhancing performance across a wide range of applications. Several works have explored the use of WebAssembly in embedded systems. Kim et al. [39] introduced a hardware-based WebAssembly accelerator tailored for embedded environments, showcasing substantial performance gains by offloading computational tasks to specialized hardware. Similarly, Rupp et al. [74] investigated hardware-assisted dispatching for WebAssembly interpreters, achieving faster execution and reduced energy consumption through custom hardware optimizations.

Despite its growing adoption, Wasm currently faces limitations in performance when handling computationally intensive workloads, such as data analytics or machine learning inference. Several strategies have emerged to bridge this performance gap. One direction focuses on runtime-level offloading, where the WebAssembly runtime itself detects or annotates suitable workloads for parallel execution and delegates them to accelerator backends. Another approach uses ahead-of-time (AOT) compilation to convert Wasm bytecode into optimized native code, leveraging SIMD instructions or specialized hardware features.

A complementary method involves the use of host functions, where Wasm runtimes such as Wasmtime or Wasmer expose native operations to Wasm code through controlled interfaces. This mechanism enables

Wasm modules to perform complex native computations, including GPU kernel execution, while preserving the sandbox guarantees. Collectively, these strategies aim to bring Wasm closer to native-level compute efficiency without sacrificing portability or security.

In the realm of high-performance computing (HPC) and cloud-edge continuum, Chadha et al. [7] evaluated the feasibility of running HPC workloads using WebAssembly, outlining its potential and limitations in resource-intensive scenarios. Expanding this perspective, Kakati et al. [32] conducted a cross-architecture evaluation, emphasizing the importance of optimizing WebAssembly for diverse hardware platforms in edge-cloud environments.

GPU acceleration has emerged as a key enabler for performance-intensive tasks. Sisyukov et al. [83] demonstrated the utility of GPU acceleration for embodied agent training in web-based workflows, highlighting its effectiveness in handling complex simulations. Oishi et al. [63] focused on image convolution, showing that WebAssembly can achieve competitive performance in real-time image processing tasks with GPU support. Additionally, Besard et al. [2] explored how WebAssembly can integrate with GPU programming in Julia, achieving high-performance computations.

The potential of WebAssembly for IoT and edge computing has also been extensively studied. Li et al. [46] demonstrated how WebAssembly could enhance device-cloud integration for resource-constrained IoT devices. In the context of AIoT, Mika et al. [58] and Kaiser et al. [31] showcased VEDLIoT, a system integrating WebAssembly with AI technologies to deliver scalable solutions. Similarly, Hampau et al. [23] provided an empirical analysis of WebAssembly's performance and energy efficiency for AI containerization in edge scenarios.

Several works address the security and trustworthiness of WebAssembly in constrained environments. Ménétrey et al. [56] introduced WaTZ, a trusted runtime for WebAssembly with remote attestation, enhancing security in sensitive applications. Pop et al. [69] tackled the challenges of secure migration of WebAssembly enclaves, offering robust solutions for maintaining integrity during migration. Additionally, Liu et al. [51] proposed SecDeep, a secure inference framework for IoT devices, leveraging WebAssembly for efficient and secure deep learning.

Emerging frameworks and tools further demonstrate the versatility

of WebAssembly. Gao et al. [20] introduced ZKWASM, combining zero-knowledge proofs with WebAssembly for enhanced privacy and security. In edge computing, Gackstatter et al. [16] highlighted the lightweight and portable nature of WebAssembly for serverless architectures. Wen et al. [95] proposed Wasmachine, a WebAssembly operating system designed to optimize edge computing devices.

These studies collectively illustrate the broad applicability of WebAssembly, spanning embedded systems, GPU-accelerated workflows, IoT, and secure computing environments. They underscore the growing significance of WebAssembly as a versatile technology that bridges performance, flexibility, and security across a wide range of domains.

The WebAssembly System Interface serves as the foundation for enabling system-level capabilities in Wasm applications. By abstracting OS-level functionalities through stable, capability-based APIs, WASI allows Wasm modules to behave like native binaries while maintaining sandbox isolation. The modular nature of WASI encourages the development of domain-specific extensions, which can expose hardware acceleration or specialized computation features.

Recent proposals have focused on extending WASI for AI, graphics, and compute workloads. These extensions, such as *WASI-NN* and *WASI-GFX*, demonstrate how standardized APIs can enable Wasm applications to interact with accelerators across diverse hardware backends.

7.2.1 WASI-NN: Neural Network Acceleration

The WASI-NN proposal defines a portable API for performing neural network inference within Wasm environments, as shown in Figure 7.2 and Figure 7.1. It provides a standard interface for loading machine learning models, initializing execution contexts, and performing inference across multiple hardware targets, including CPUs, GPUs, and neural accelerators. Implementations of WASI-NN typically rely on existing inference engines such as OpenVINO, TensorFlow Lite, or TensorRT to execute model graphs efficiently.

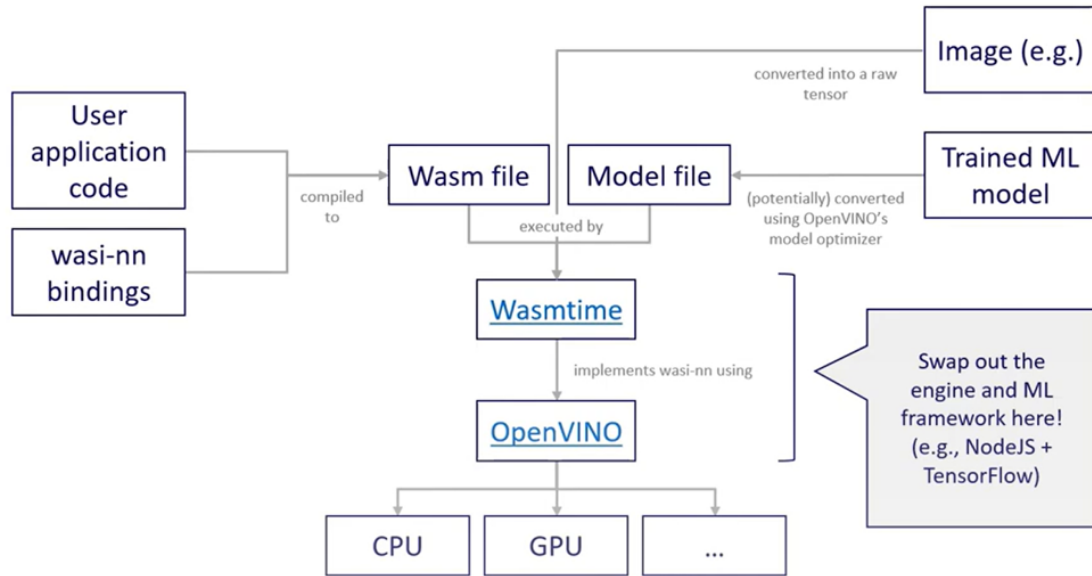


Figure 7.1: Wasi-nn Backend and Bindings

The key contribution of WASI-NN is the abstraction of heterogeneous hardware through a unified API surface. This allows developers to deploy pre-trained models in Wasm environments without needing to manage backend-specific details or memory synchronization. It also demonstrates that the WASI model can be extended to expose compute-intensive operations while preserving WebAssembly’s sandboxed execution semantics.

- **Neural Network Model Execution:** WebAssembly modules can offload tensor computations, typically involving neural network operations, to GPUs.
- **Standardized API for ML Tasks:** WASI-NN defines a set of APIs for machine learning tasks that WebAssembly modules can call, which can then route the computation to GPU-accelerated backends like TensorFlow or ONNX.

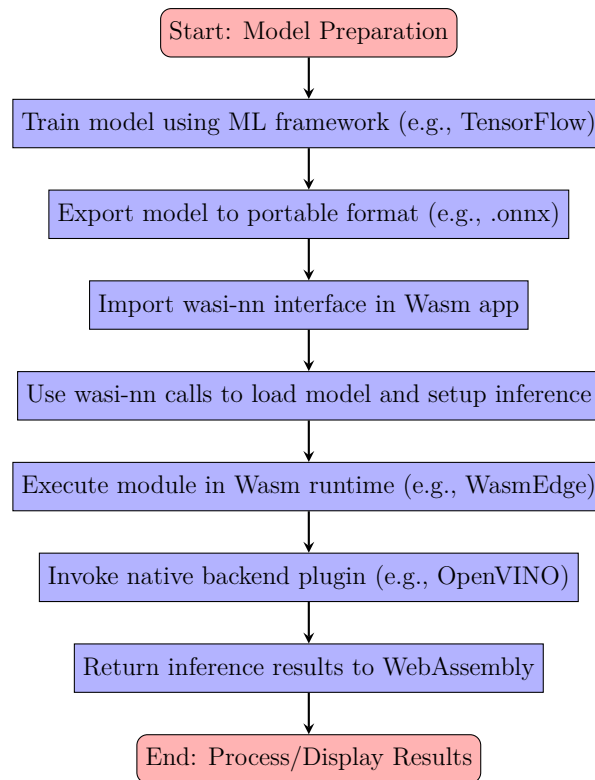


Figure 7.2: Workflow of the WASI-NN Interface

- **Accelerated Inference:** Using specialized hardware like GPUs to run inference tasks on neural networks, offloading heavy computations from the CPU to the GPU for faster results.

7.2.2 WASI-GFX: Graphics and Compute Acceleration

While `wasi-nn` targets machine learning inference workloads, `wasi-gfx` extends the WASI ecosystem toward graphics rendering and GPU-based computation. The proposal is informed by modern graphics and compute APIs, including WebGPU as well as lower-level systems such as Vulkan and Metal, and defines interfaces for managing GPU devices, memory buffers, pipelines, and shader execution.

Early discussions around GPU support in WASI considered a more general-purpose accelerator interface. However, this direction was ultimately refined to focus explicitly on graphics and graphics-adjacent compute use cases, leading to the consolidation and renaming of the effort as `wasi-gfx`. As a result, `wasi-gfx` represents the current standardization path for GPU access within the WASI ecosystem, rather than a separate or parallel “WASI-GPU” interface.

Although its primary motivation is graphics, **wasi-gfx** also exposes compute capabilities through GPU compute shaders. This allows WebAssembly applications to offload certain classes of data-parallel workloads to GPUs, making the interface relevant beyond purely graphical applications. In this sense, **wasi-gfx** provides an important foundation for accelerator access in WASI, even if it does not aim to fully generalize GPU programming in the manner of traditional compute APIs.

An overview of the **wasi-gfx** architecture is shown in Figure 7.3.

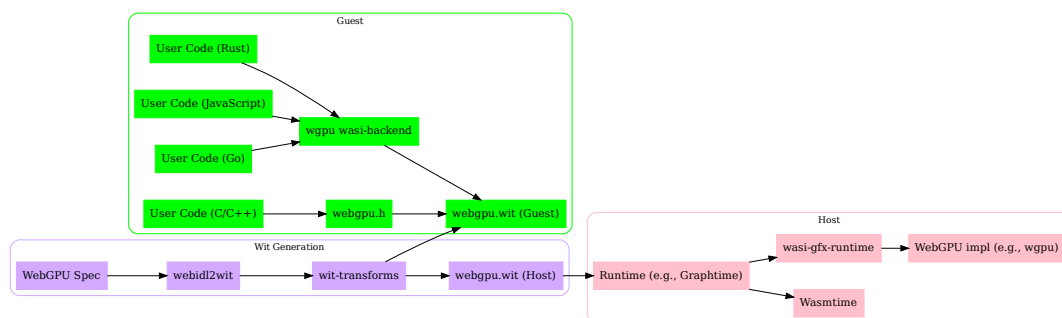


Figure 7.3: Illustration of the WASI-GFX architecture and its relationship to Web and native layers.

7.2.3 Comparison of GPU Acceleration Approaches

Table 7.1 summarizes different approaches for GPU acceleration in WebAssembly environments. Each framework or proposal offers a distinct level of abstraction, backend integration, and maturity. While WebGPU provides a cross-platform abstraction layer primarily for browser and native use, WASI-NN and WASI-GFX extend this functionality into standardized WASI modules. Research efforts, such as the proposed WASI-GPU interface, aim to generalize GPU compute support beyond specialized domains like AI inference or graphics rendering.

7.2.4 WebAssembly Host Functions and WIT Integration

When a WebAssembly module invokes a host function, the runtime mediates between the module and the underlying system by translating data structures through the WebAssembly Interface Types (WIT) format. This enables high-level API definitions in WIT to be automatically

Table 7.1: Comparison of GPU acceleration approaches in WebAssembly runtimes.

Approach	Description	GPU Access	Status
WebGPU (via wgpu)	Cross-platform GPU abstraction for browsers and native runtimes	Indirect (via Vulkan, Metal, DX12)	Stable for graphics; limited compute
WASI-NN	Standardized interface for neural network inference	Indirect (via ML back-ends)	Experimental
WASI-GFX	API for GPU graphics and compute workloads	Direct (through GPU abstraction)	Prototype
WASI-GPU (proposed)	General-purpose GPU interface with memory handle model	Direct (via host bindings)	Research stage

mapped into native host bindings. For example, WASI-NN defines interfaces for loading a model, initializing execution contexts, setting inputs, and computing outputs. A simplified example can be expressed as follows:

```
interface wasi-nn {
    load(model: array<u8>) -> Graph;
    init_execution_context(graph: Graph) -> Context;
    set_input(ctx: Context, data: array<f32>);
    compute(ctx: Context);
    get_output(ctx: Context) -> array<f32>;
}
```

In a similar fashion, a general-purpose WASI-GPU interface could define buffer management, data transfer, and kernel execution functions. These would allow WebAssembly programs to allocate host memory, copy data, and invoke GPU kernels through handle-based operations, abstracting away platform-specific APIs.

The evolution of WebAssembly toward accelerated computing is transforming it from a lightweight web runtime into a versatile execution layer for cloud and edge environments. Proposals like WASI-NN and WASI-GFX illustrate how WASI can expose specialized hardware capabilities through standardized interfaces. Building on these foundations,

a general-purpose WASI-GPU interface can unify GPU compute access across architectures, enabling Wasm modules to offload diverse workloads efficiently and securely across the edge-cloud continuum.

The WebAssembly Component Model and WIT specification enable strongly typed interfaces between guest modules and host implementations. Host-side acceleration allows high-performance compute offloading while maintaining portability and safety. Our work builds upon Wasmtime’s component system and the Polars library, leveraging handle-based patterns such as the `ResourceTable` for safe host-managed resources.

As WebAssembly grows beyond the browser, supporting offloading and GPU acceleration within WASI runtimes has become increasingly important. This evolution parallels the development of `wasi-nn` and experimental interfaces such as `wasi-gfx`, which aim to unify GPU access and general-purpose compute acceleration.

The host-side acceleration described above aligns with broader strategies for improving WebAssembly performance through GPU offloading and specialized runtimes. Approaches such as `wasi-nn`, `wasi-gfx`, and AOT compilation extend WebAssembly’s reach beyond CPUs into heterogeneous hardware. Our work with `wasi:accelerator` parallels these proposals by demonstrating an extensible host abstraction layer for numerical compute.

7.3 Design

7.3.1 APIs

The WIT interfaces define the following host functions:

- **`wasi:dataframe/imports`**, functions: `load_csv`, `from_rows`, `filter`, `group_by`, `aggregate`, `to_json`.
- **`wasi:accelerator/imports`**, functions:
`allocate_buffer`, `free_buffer`, `write_to_host`, `read_from_host`,
`register_matrix_dimensions`,
`get_matrix_dimensions`, `matrix_multiply_f32`.

7.3.2 Resource Model

Both crates expose opaque handles (`u32`) referencing host-owned buffers or lazy frames within context structs (`WasiAcceleratorCtx`, `WasiDataframeCtx`). The host stores data in `HashMap`s keyed by handles. Each new resource is allocated via a `new_handle()` generator, ensuring uniqueness and explicit lifetime control via `free_buffer` or replacement semantics.

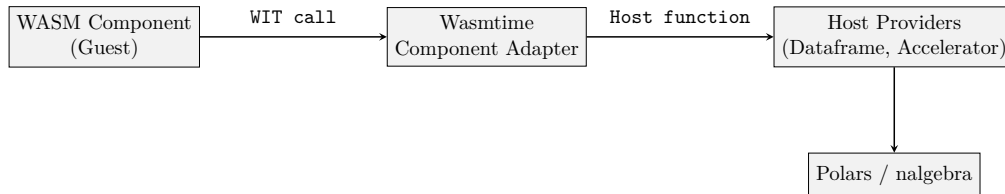


Figure 7.4: High-level dataflow between guest, component adapter, and host providers.

7.4 Implementation Details

7.4.1 wasi-dataframe

- Uses `polars::lazy` APIs (`LazyFrame`) to represent dataframe handles for efficient cloning and deferred computation.
- `from_rows` constructs UTF-8 columns and casts numeric values as needed during filtering.
- `to_json` materializes a small sample (`limit(100)`) and serializes rows to JSON.
- Aggregations currently support `count`; `mean` requires WIT updates for column specification.

7.4.2 wasi-accelerator

- Host buffers stored as `Vec<u8>` keyed by handles.
- Conversion functions handle f32-byte mapping safely (`bytes_to_f32_slice`, `f32_slice_to_bytes`).

- `matrix_multiply_f32` reads buffers, constructs `nalgebra::DMatrix<f32>`, performs multiplication, and stores the result in a new handle.

We implemented our DataFrame acceleration interface as a new WASI proposal within a modified version of Wasmtime [93]. The implementation is available at our public repository `wasmtime`¹, where the full set of crates and build steps can be reproduced.

To demonstrate the standalone use of our interface, we developed a simple WebAssembly component that imports the `wasi:dataframe/dataframe-analysis` interface and executes a Polars-based example. This client, located at `wit-interface`², demonstrates the ability to create and manipulate DataFrames within Wasm using the WIT-defined API.

7.5 Experimental Methodology

The primary goal of this work was to see the feasibility of offloading general-purpose computation from WebAssembly components to host accelerators using a unified abstraction. Instead of focusing on quantitative benchmarking such as throughput or latency, the evaluation centered on validating the correctness, interoperability, and execution flow of the proposed interface.

Specifically, we aimed to:

1. Implement a minimal proof-of-concept interface (*Polars*) that allows WebAssembly modules to allocate host memory, copy data between guest and host, and perform compute operations via integer memory handles.
2. Demonstrate the successful execution of representative workloads including matrix multiplication and basic dataframe operations (`filter`, `group_by+aggregate(count)`) using the proposed WASI-based host interface in Wasmtime.
3. Validate that host-side GPU acceleration can be accessed transparently through a WebAssembly component without breaking sandboxing or portability guarantees.

¹<https://github.com/matsbror/wasmtime>

²<https://github.com/sangeeta1998/wit-interface>

The focus was therefore on confirming the end-to-end call chain from a WebAssembly component to host-executed compute operations and ensuring that the proposed design could serve as a foundation for more comprehensive acceleration frameworks in future work.

7.5.1 Benchmarks

1. **Matrix Multiply Benchmark:** Allocate three buffers (A,B,C) for matrices of varying sizes (64, 256, 1024). Measure host call latency.
2. **Dataframe Pipeline:** Build frames via `from_rows` or `load_csv`, apply filters and aggregations, and measure total runtime.

7.5.2 Reproducible Commands

```
# Run the wasi-accelerator integration test
cargo test -p wasmtime-wasi-accelerator --test main

# Run the wasi-dataframe integration test
cargo test -p wasmtime-wasi-dataframe --test main

# Run example component
cargo run --package wasmtime -- ../path/to/component.wasm
```

7.5.3 Example Client Implementation

To validate our interface design and demonstrate real-world usage, we implemented a standalone WebAssembly component in our [polars-client](#) repository. This client showcases how Wasm components can utilize the dataframe interface:

```
// Import the dataframe interface
use dataframe::*;

#[component]
impl Example {
    // Example showing dataframe manipulation
    pub fn run_dataframe_example() -> Result<(), Error> {
        // Create sample data
        let columns = vec!["city", "group", "val"];
        let rows = vec![
            vec!["A", "x", "10"],
            vec!["A", "y", "5"],
            // ... more rows
        ];
    }
}
```

```

    // Create dataframe from rows
    let df = from_rows(columns, rows)?;

    // Apply filter: val > 5
    let filter = ColumnFilter {
        column: "val".to_string(),
        op: Comparator::Gt,
        value: Scalar::Value(5.0),
    };
    let filtered = filter(df, vec![filter])?;

    // Get results as JSON
    let json = to_json(filtered)?;
    println!("Filtered results: {}", json);
    Ok(())
}
}

```

The component can be built and run using:

```

cargo component build --target=wasm32-wasip2
wasmtime-dev run -S dataframe=y --invoke
'run-dataframe-example()' target/wasm

```

This example demonstrates the complete workflow from component creation through execution in Wasmtime with our host implementation.

7.6 Conclusion

This chapter presented a novel approach to host-side acceleration for WebAssembly components through two complementary interfaces: wasi-dataframe and wasi-accelerator. Our implementation demonstrates how WebAssembly components can efficiently offload data processing and numerical computations to host-provided libraries while maintaining the security and isolation guarantees that make WebAssembly attractive for safe code execution.

The wasi-dataframe interface, backed by Polars, shows how high-level data processing operations can be exposed to WebAssembly components through a clean, handle-based API. By representing dataframes as opaque handles managed by the host, we achieve both efficiency avoiding unnecessary data copying between guest and host, and safety through controlled resource access. The LazyFrame implementation enables deferred computation and potential optimization of operation sequences, similar to how database engines optimize query plans.

The wasi-accelerator interface provides a foundation for general-purpose acceleration patterns, managing host-side buffers and matrix operations through a similar handle-based approach. This design anticipates future extensions to GPU and other specialized hardware backends, while maintaining a consistent programming model for component authors. The separation between abstract handles and concrete implementations allows for different acceleration strategies without changing the component-facing API.

Feedback from the WebAssembly research community highlighted several promising directions. The suggestion to adopt command-buffer patterns from graphics APIs could improve operation batching and optimization. The potential integration with Google’s Dawn and alignment with wasi-gfx points toward a more unified approach to acceleration interfaces. The discussion of performance contracts and metadata, inspired by the EU CloudWave project, suggests ways to make acceleration capabilities more predictable and manageable for developers.

Our proof-of-concept implementation, demonstrated through a standalone WebAssembly component in the polars-client repository, shows the practical viability of this approach. The component successfully uses the dataframe interface to perform common data operations, validating both the API design and the implementation strategy. While current functionality is intentionally limited to core operations, the architecture supports progressive enhancement with more sophisticated features and acceleration techniques.

Critical areas for future work include migrating to component-model resource types for better lifetime management, developing performance profiling and metadata systems, and exploring integration with existing acceleration frameworks. The research questions raised about automatic identification of acceleratable patterns and the balance between generality and performance will guide the evolution of this work.

This work contributes to the broader conversation about WebAssembly’s role in high-performance computing and data processing. By demonstrating how host capabilities can be safely exposed to WebAssembly components, we provide a pattern for building acceleration interfaces that maintain WebAssembly’s security properties while enabling efficient computation. The handle-based abstraction, combined with lazy evaluation and potential for backend optimization, offers a promising model for future acceleration interfaces in the WebAssembly ecosystem.

Looking ahead, the integration of performance metadata, formal safety guarantees, and expanded acceleration backends will make this approach increasingly valuable for real-world applications. As WebAssembly continues to evolve beyond the browser, frameworks like this will be essential for bridging the gap between portable, secure code execution and high-performance computing resources.

We demonstrated a minimal host-accelerated API for numeric workloads integrated with a dataframe interface. This enables WebAssembly components to offload heavy computation to the host while maintaining sandboxing and portability. These abstractions can serve as a foundation for integrating GPU compute backends into WASI runtimes, bridging the gap between Wasm portability and accelerator performance.

Chapter 8

Discussion

The rapid transformation of computing paradigms from centralized cloud architectures toward distributed and heterogeneous infrastructures has fundamentally reshaped the way modern software systems are developed, deployed, and executed. This thesis explored how WebAssembly can act as a technological and conceptual bridge across this evolving continuum. The individual studies in this work collectively examined Wasm from several complementary perspectives, ranging from its runtime performance across architectures to its role in portable containerization and its potential for standardized host-accelerated computation through the WebAssembly System Interface.

The first overarching insight drawn from this work is that WebAssembly provides a genuine opportunity for cross-architecture unification. Unlike traditional binary formats that require architecture-specific compilation, Wasm offers a standardized intermediate representation capable of being executed on diverse hardware platforms. The empirical evaluations across x86_64, ARM64, and RISC-V architectures demonstrated that Wasm runtimes can provide consistent performance behavior despite architectural differences, and that runtime-level optimizations can mitigate many of the overheads traditionally associated with portable bytecode systems.

These findings are significant because they suggest that performance portability which is a long-standing challenge in heterogeneous computing and can be approached not through hardware abstraction layers alone, but through a language-independent, sandboxed execution layer. The experiments revealed that while there is still a measurable performance gap between Wasm and native binaries, this gap is narrowing as runtimes evolve, particularly when leveraging ahead-of-time compilation

and efficient memory management.

A major contribution of this research lies in the comparative evaluation of Wasmtime and WAMR. The analysis revealed that each runtime exhibits distinct optimization priorities and trade-offs, reflecting its design philosophies. Wasmtime’s emphasis on Just-In-Time compilation and modularity allows for dynamic optimization and ease of embedding, while WAMR’s focus on static footprint reduction makes it well-suited for constrained environments. Together, they represent the dual directions in which the Wasm ecosystem is expanding toward powerful edge servers on one end and lightweight IoT and embedded systems on the other.

This dual capability reinforces the notion that interoperability and portability need not come at the cost of efficiency. The findings suggest that by selecting appropriate runtimes and compilation strategies, developers can achieve near-native performance even on resource-constrained platforms. Moreover, the study contributes to the broader discourse on the role of Wasm as an execution substrate capable of abstracting away the complexity of heterogeneous architectures while still preserving predictable performance semantics.

The second major dimension of this work concerned the integration of Wasm within containerized infrastructures. By comparing WebAssembly-based containers with traditional Docker containers under both Docker and containerd environments, the research demonstrated that Wasm containers exhibit substantial benefits in image size and cold-start behavior. These results have practical implications for the deployment of microservices and serverless functions, especially in bandwidth-constrained or dynamically scaling edge environments.

The ability of Wasm containers to reduce image sizes by up to seventy percent and improve cold pull times by twenty-five percent suggests that Wasm can play a crucial role in energy-efficient and sustainable computing. Smaller and faster-deploying artifacts translate directly into reduced network traffic, lower storage requirements, and decreased energy consumption. This aligns closely with the sustainability objectives of next-generation distributed infrastructures, where minimizing data movement and resource waste are becoming as critical as raw computational power.

At the same time, the experiments also revealed the immaturity of

the tooling ecosystem surrounding Wasm containerization. Although support through runtimes such as `containerd-shim-wasmtime` is growing, integration with existing orchestration frameworks remains limited. Addressing these challenges will require continued collaboration between standardization bodies, cloud providers, and open-source maintainers to fully integrate Wasm into the mainstream container landscape.

The final research contribution explored the extension of the WASI ecosystem toward host-accelerated and data-centric execution models. By defining and implementing experimental WIT-based interfaces for host memory management, matrix multiplication, and dataframe analytics, this work demonstrated that WASI can be expanded beyond traditional system calls into a general-purpose compute abstraction layer. The introduction of host-managed buffers, capability-based access, and opaque resource handles established a foundation for offloading computation between Wasm and host accelerators.

This direction has substantial implications for heterogeneous computing at the edge and in the cloud. Future workloads will increasingly depend on specialized accelerators such as GPUs, TPUs, and NPUs. The ability to interact with such hardware in a portable, language-neutral way through WASI extensions could enable a new class of distributed applications capable of dynamically adapting to the hardware capabilities available at runtime. The experimental results show that such integration is technically feasible and efficient, though further work is needed to standardize the interface and guarantee security isolation.

While the outcomes of this research demonstrate the growing maturity of the Wasm ecosystem, several limitations remain. The current performance advantage of Wasm containers is primarily realized in startup latency and image size rather than sustained computational throughput. Furthermore, while WASI provides a promising abstraction for system and hardware interactions, its support across runtimes is still uneven, and many proposals remain experimental. Another important consideration concerns the observability and debugging of Wasm components. As they become increasingly deployed in distributed environments, tools for tracing, performance profiling, and security auditing will be necessary to ensure reliability and trust.

From a methodological perspective, this work also underscores the challenges of evaluating performance across multiple architectures and runtimes in a consistent way. Benchmarking Wasm performance re-

quires careful control over compilation modes, runtime configurations, and host system variations. The experience gained through this research highlights the need for community-driven benchmarking standards to ensure comparability and reproducibility across studies.

The collective findings of this thesis point toward a clear trajectory: WebAssembly is evolving from a browser technology into a universal runtime for distributed, portable, and secure computing. The transition from the web to the cloud and edge is not merely a change in deployment context but represents a broader philosophical shift toward modularity, interoperability, and sustainability in computing.

Looking forward, the integration of WebAssembly into autonomous and adaptive deployment systems presents one of the most exciting opportunities. With continued standardization of WASI and maturing of the component model, it may become possible for distributed applications to automatically reconfigure their execution strategy based on available hardware and environmental conditions. This could finally realize the long-standing vision of write once, run anywhere efficiently not as an abstract ideal, but as a practical engineering reality.

8.1 Exploratory Analysis

This section consolidates the foundational research and exploratory work conducted at the inception of this project. It outlines the core concepts of WebAssembly, its evolution beyond the browser, the performance landscape of its runtimes, and the rationale for its application in edge and cloud environments, which collectively form the basis for the research questions and experimental work presented in this thesis.

8.1.1 WebAssembly Fundamentals

Wasm is a low-level, portable binary instruction format originally designed as a compilation target for high-performance web applications. Unlike traditional assembly languages tied to specific machine architectures, Wasm is designed for a conceptual machine, enabling true “compile once, run anywhere” portability. A key differentiator from JavaScript is its delivery as a compact binary format, which bypasses the heavy parsing and abstract syntax tree construction required by

JavaScript, leading to faster decoding and startup times.

The execution of Wasm code is managed by a *runtime*, which provides a secure, sandboxed environment. Runtimes operate primarily in three execution modes:

- **Interpretation:** The Wasm bytecode is executed instruction-by-instruction. This mode offers the fastest startup time and is highly portable but generally has the highest execution overhead.
- **Just-in-Time (JIT) Compilation:** The bytecode is compiled to native machine code at runtime. This approach introduces a slight initial compilation overhead but results in significantly faster execution speeds for medium to long-running tasks. Prominent runtimes like Wasmtime and Wasmer utilize this method.
- **Ahead-of-Time (AOT) Compilation:** The bytecode is pre-compiled to native code before execution. This mode eliminates JIT compilation overhead, yielding fast startup and consistent performance, which is particularly valuable in resource-constrained and embedded systems. WAMR provides a strong AOT compilation capability.

The performance characteristics of these modes are distinct. JIT-based runtimes typically outperform interpreters, and AOT compilation can effectively mitigate JIT overhead, especially for workloads with large binary code sizes.

8.1.2 The Expansion Beyond the Browser: WASI and WasmCloud

Recognizing its potential for performance, security, and portability, the WebAssembly ecosystem rapidly expanded beyond the browser. This was made possible by the development of the WebAssembly System Interface. This modular system interface standardizes how Wasm modules interact with host system resources like files, networks, and clocks. WASI provides the foundational API that allows Wasm programs to run consistently on any platform, from servers to edge devices, without modification.

This server-side evolution has been further accelerated by frameworks like **WasmCloud**, which provides an application runtime built on an

actor model. WasmCloud leverages Wasm’s capability-based security model to create a secure, portable, and manageable environment for distributed application components, streamlining the software supply chain by virtualizing non-functional requirements.

8.1.3 WebAssembly in Edge and Cloud Computing

The intrinsic properties of WebAssembly make it an exceptionally compelling technology for edge and cloud computing environments, particularly when compared to traditional container technologies like Docker.

- **Lightweight Isolation:** Wasm provides sandboxed isolation at the application level, which is significantly more lightweight than OS-level containerization. This results in a drastically reduced memory footprint and faster startup times. Community benchmarks have demonstrated Wasm container images as small as 2.8 KiB, compared to 9.9 MiB for optimized distroless Docker images. Cold-start latency can be reduced by up to 99.5%, which is critical for responsive edge applications and serverless functions.
- **Cross-Platform Portability:** A single Wasm binary can run unchanged on diverse CPU architectures (x86-64, ARM64, RISC-V) and operating systems. This “compile once, run anywhere” capability eliminates the need to build and maintain separate container images for different edge hardware, simplifying deployment across a heterogeneous fleet.
- **Security:** The Wasm sandbox is designed to be secure by default. Modules have no default access to system resources and can only interact with the outside world through the well-defined WASI APIs, presenting a reduced attack surface compared to traditional containers.
- **Orchestration Integration:** The technology is not positioned to entirely replace Docker but to integrate with and complement the existing cloud-native ecosystem. Projects like `runwasi` allow containerd and Kubernetes to natively orchestrate Wasm workloads alongside Linux containers, enabling a hybrid deployment model where the optimal runtime (Wasm or container) can be selected based on the specific application requirements.

8.1.4 Performance Landscape and Research Challenges

Initial benchmarking and literature review revealed a complex performance landscape. While Wasm runtimes can achieve near-native execution speed for computational kernels, performance is highly dependent on the runtime engine (e.g., Wasmtime’s Cranelift vs. Wasmer’s LLVM backend), the execution mode (JIT vs. AOT), and the underlying host architecture.

Furthermore, performance discrepancies were observed across different CPU architectures. For instance, runtimes like Wasmtime, heavily optimized for widespread X86 and ARM platforms, may not perform as favorably on emerging RISC-V architectures due to less mature compiler backends and toolchains. Differences in microarchitecture—such as deeply out-of-order, superscalar pipelines in ARM Cortex-A57 versus simpler, in-order dual-issue pipelines in RISC-V cores can also lead to varying performance benefits from Wasm’s execution model.

These observations, combined with the growing interest in deploying Wasm across the edge-cloud continuum, crystallized the core research questions for this thesis: How does Wasm performance compare to native execution and containers across heterogeneous architectures? What are the overheads and optimization opportunities when integrating Wasm with container orchestration? And finally, under what conditions does WebAssembly provide a tangible advantage for building efficient, scalable, and portable distributed systems?

Chapter 9

Limitations and Future Work

Contents

9.1 Current Limitations	152
9.2 Future Work	153
9.3 Further Research Questions	155

9.1 Current Limitations

The work presented in the final contributing chapter of this thesis (Chapter 7) constitutes an early exploration toward a general-purpose acceleration interface for WebAssembly, designed to bridge the gap between domain-specific WASI proposals such as `wasi-nn` and `wasi-gfx`. Our approach, embodied in the prototype system *Polars*, demonstrates how host-allocated memory and handle-based abstractions can enable WebAssembly components to interact with accelerated backends such as GPUs or other co-processors. While this foundation establishes a promising direction, several technical and conceptual limitations currently constrain the scope of the implementation. These limitations, along with the major avenues for future research, are discussed below.

At this stage, the prototype is deliberately minimal. It defines a `wasi:dataframe` interface supporting a small number of core operations, such as `load-csv`, `filter`, and `aggregate`, and uses integer handles to represent host-managed memory regions. These design choices allowed for an early proof of concept within Wasmtime, but they also reveal certain constraints that must be addressed before the system can evolve into a general solution.

First, the current implementation operates purely on the CPU and does not yet interface with any actual GPU or hardware acceleration API. Although the proposed design anticipates such integration, the absence of a backend limits evaluation to host-side performance and correctness validation rather than measuring true acceleration.

Second, the manual management of memory handles introduces fragility. Because the current prototype does not rely on the WASI component model’s formal **resource** abstraction, lifetime management, ownership tracking, and safety guarantees must be handled manually. This increases complexity and diverges from the idiomatic design principles of the emerging component model.

Third, the range of supported data operations remains narrow. The current abstraction treats data as a single homogenous **DataFrame** type, limiting its suitability for workloads involving tensors, unstructured memory buffers, or streaming data. Extending the model to accommodate richer data representations and composable operators would be necessary for wider applicability.

Finally, the development process itself exposed practical challenges. Implementing the WASI interface in Wasmtime required navigating limited documentation and reverse-engineering structure from examples such as `wasi-keyvalue`. Tasks like generating WIT bindings, integrating with CI pipelines, and writing unit tests proved non-intuitive, reflecting the broader need for better tooling and developer guidance within the WASI ecosystem.

9.2 Future Work

Building on the current foundation, several directions for future research emerge. The most immediate is to establish a true accelerated backend. Integrating the proposed interface with an existing general-purpose compute API such as **Dawn**, the C/C++ implementation of **WebGPU** would provide a concrete execution target for offloaded operations. This would transform *Polars* from a conceptual model into a practical, hardware-backed acceleration layer capable of exploiting GPU parallelism for data processing.

Another key step is aligning the handle-based design with the WASI component model. The resource abstraction introduced in the compo-

ment model provides a structured mechanism for managing ownership, lifetimes, and safety of host-managed entities. Adopting these semantics would not only simplify implementation but also enhance interoperability and composability between different WASI components.

A complementary direction involves embedding performance awareness into WASI interfaces. Currently, a module using a WASI API cannot query the expected performance profile or determine the capabilities of the underlying platform. Introducing performance metadata such as estimated throughput, latency, or supported instruction sets would allow both developers and runtimes to make informed decisions about when and where to offload computation. This could also enable deterministic compute guarantees for edge deployments and real-time workloads.

From a broader perspective, it may also be worthwhile to explore acceleration at the bytecode or intermediate representation (IR) level. Although this is challenging due to WebAssembly’s isolation semantics and memory model, selective acceleration of self-contained code regions, guided by compiler annotations or static analysis, remains a promising long-term goal. This could complement the API-driven approach by providing a second path to performance improvement without requiring explicit use of specialized WASI interfaces.

Lastly, improving the developer experience is an essential meta objective. Creating new WASI proposals remains a complex and poorly documented process. Future work should therefore aim to produce clearer templates, automated testing scaffolds, and comprehensive developer guides to lower the barrier for experimentation. Making this process more approachable would not only accelerate this particular line of research but also encourage community-driven innovation in the WASI ecosystem.

Our current handle-based implementation would benefit from migrating to the component model’s resource types for safer lifetime management. This transition requires implementing formal verification of memory access patterns and buffer lifecycle management. To support broader adoption, we need to define serialization protocols for safely sharing accelerator resources across different Wasm runtimes.

A systematic survey of existing accelerator interfaces, including `wasi-nn`, `wasi-gfx`, and `WebGPU`, will help identify shared patterns and ab-

stractions. Based on these findings, we can design a common interface capable of targeting multiple acceleration backends such as GPUs, FPGAs, and specialized hardware. Creating bidirectional bridges to Python data science tools would enable seamless integration with existing ML and data workflows, making our interface more practical for real-world applications. The implementation experience we had is that there is a need for better documentation for implementing WASI interfaces in Wasmtime, particularly regarding component model integration. We plan to develop comprehensive testing tools for WASI interface implementations and create templates and tools for continuous integration. The first step is to create a minimal, reproducible benchmark that compares native Polars, Polars compiled to Wasm, and Polars-on-Wasm via our host-accelerated WASI interface, all using the same dataset and a small set of dataframe operations. This will allow us to measure both the overhead and the performance gains of the handle-based acceleration strategy before we expand the feature set any further. These improvements will make it easier for other researchers and developers to build upon this work and contribute to the ecosystem.

9.3 Further Research Questions

Several fundamental research questions emerged during this phase that warrant further investigation. First, how can we automatically identify accelerator-suitable code patterns in WebAssembly modules? This connects to broader questions about compiler optimization and static analysis.

Second, what abstraction level provides the optimal balance between generality and performance? The tension between generic interfaces and specialized optimizations requires careful consideration. Third, how can we maintain WebAssembly's security properties while enabling efficient accelerator access?

Finally, what role should standardization play in defining accelerator interfaces? The answers to these questions will shape the evolution of WebAssembly acceleration patterns.

There exists already a strong interest in general-purpose acceleration patterns for WebAssembly, suggesting our handle-based approach could evolve into a broader framework for safe and efficient host acceleration.

Integration with existing efforts like Dawn and wasi-gfx, while maintaining focus on data processing workloads, presents a promising path forward. The combination of performance metadata, robust resource management, and careful integration with existing frameworks will be crucial for realizing the full potential of this approach.

Chapter 10

Conclusion

This thesis set out to explore how distributed applications can be deployed so that software can automatically and seamlessly adapt to the multitude of hardware platforms that will form future cloud infrastructures. The motivation for this work emerged from the increasing fragmentation and heterogeneity of modern computing systems, where the cloud, the edge, and end devices together constitute a complex continuum of architectures and execution environments. Within this setting, it has become increasingly difficult to ensure both portability and performance without imposing excessive demands on developers or infrastructure operators.

The research presented in this work has examined how WebAssembly, together with the WebAssembly System Interface, can offer a unifying abstraction for heterogeneous computing. By investigating the potential of WebAssembly as a common execution format beyond the browser, this thesis has demonstrated its relevance as a foundation for deploying distributed applications that can execute across diverse hardware while maintaining efficiency, security, and interoperability. The work has shown that WebAssembly can bridge the gap between high-level software portability and low-level hardware diversity, providing a practical path toward performance portability in real-world distributed systems.

The implementation and experimentation conducted throughout this research have provided insights into how WebAssembly runtimes can be extended to support general-purpose computing across different platforms. By introducing a general GPU interface within the WASI framework, this work has taken a significant step toward enabling WebAssembly modules to access and utilize hardware accelerators in a standardized

manner. The design and evaluation of this interface have revealed both the strengths and the current limitations of WebAssembly as a platform for high-performance and heterogeneous workloads. In particular, it has demonstrated that with appropriate host integration, Wasm modules can interact effectively with devices such as GPUs, enabling meaningful acceleration while preserving the security and modularity that make WebAssembly an attractive option.

Beyond the technical contributions, this thesis also reflects on the conceptual shift that WebAssembly represents. Originally envisioned as a mechanism for delivering fast and safe code within the browser, WebAssembly has evolved into a technology capable of serving as a universal execution layer across the computing spectrum. Its minimal and well-defined semantics make it suitable for environments ranging from small embedded systems to large-scale distributed infrastructures. This work reinforces the notion that WebAssembly can become an essential element of the emerging cloud-edge ecosystem, enabling applications to be deployed with far greater flexibility and efficiency.

The exploration of the ACE5G project context has also highlighted the practical relevance of this research. As cloud infrastructures evolve, the need for adaptable deployment models that can utilize diverse hardware resources will continue to grow. The execution model proposed and developed here contributes to that vision by showing how hardware capabilities can be made available through a common abstraction layer. This approach allows applications to dynamically adapt to changing environments, achieving optimal use of resources without manual intervention. It also opens the door to more sustainable computing practices, where workloads can be rescheduled or migrated based on energy efficiency or resource availability, aligning with broader goals of environmental responsibility in computing.

The findings of this thesis carry implications for both research and practice. For researchers, they suggest a fertile area of further study in extending WebAssembly's capabilities, including support for additional hardware types, dynamic scheduling, and distributed resource management. For practitioners, they demonstrate that WebAssembly-based systems can already provide tangible benefits in terms of portability, deployment simplicity, and security. As the technology and surrounding ecosystem mature, the gap between performance-critical native execution and portable modular computation will continue to narrow.

In reflecting upon the research journey, it becomes clear that the true significance of this work lies not only in technical implementation but also in the broader perspective it provides. It challenges traditional assumptions about how software must be written, compiled, and executed, proposing instead a model where portability and adaptability are inherent qualities of the system rather than afterthoughts. In this sense, the work serves as a step toward a more unified computing paradigm, one that embraces heterogeneity rather than treating it as a burden.

In conclusion, this thesis contributes to the growing understanding of how WebAssembly can act as an enabling layer for distributed, adaptive, and sustainable computing. The methods, designs, and evaluations presented here demonstrate that it is possible to combine the efficiency of native code with the portability of a standardized virtual architecture. The path forward involves refining these mechanisms, expanding interoperability with diverse hardware, and developing richer abstractions for distributed resource management.

Ultimately, this research envisions a computing landscape where applications are no longer constrained by the underlying platform, where software can move freely and intelligently across the continuum from cloud to edge to device. WebAssembly, as explored in this thesis, provides one of the most promising foundations for realizing that vision, bringing us closer to a future of truly adaptive and resilient distributed computing.

Chapter 11

List of Publications

The publications listed below are the author’s work. The material has been significantly revised, adapted, and integrated to form the unified and original contribution presented in this doctoral thesis.

Journal Articles

1. S. Kakati and M. Brorsson, “The Small Container: Lightweight Compute for Multi-Architecture Cloud-Edge Systems,” *Springer Nature Computer Science Journal*.

Conference Proceedings

1. S. Kakati and M. Brorsson, “Performance and Usability Implications of Multiplatform and WebAssembly Containers,” in *Proceedings of the 15th International Conference on Cloud Computing and Services Science*, pp. 15–25, 2025. **(Best Paper Award)**.
2. S. Kakati and M. Brorsson, “A Cross-Architecture Evaluation of WebAssembly in the Cloud-Edge Continuum,” in *CCGrid 2024*, pp. 337–346, IEEE, 2024.
3. S. Kakati and M. Brorsson, “An Investigative Study of WebAssembly Performance in Cloud-to-Edge,” in *PCDS 2024*, pp. 1–5, IEEE, 2024.
4. S. Kakati and M. Brorsson, “WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum,” in *CONIT 2023*, pp. 1–8, IEEE, 2023.

Bibliography

- [1] Joshua Bachmeier, Vladimir Yussupov, Jörg Henß, and Heiko Koziolk. WebAssembly with wasi-nn for Edge Machine Learning Inference: Experiences and Lessons Learned. *European Conference on Software Architecture*, 2025.
- [2] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: unleashing julia on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, 2018.
- [3] Davide Borsatti, Walter Cerroni, Luca Foschini, Genady Ya. Grabarnik, Lorenzo Manca, Filippo Poltronieri, Domenico Scotece, Larisa Shwartz, Cesare Stefanelli, Mauro Tortonesi, and Mattia Zaccarini. KubeTwin: A Digital Twin Framework for Kubernetes Deployments at Scale. *IEEE Transactions on Network and Service Management*, 2024.
- [4] Bill Bosshard. On the use of web assembly in a serverless context. In *Agile Processes in Software Engineering and Extreme Programming–Workshops*, page 141, 2020.
- [5] Florian Breitfelder, Tobias Roth, Lars Baumgärtner, and Mira Mezini. Wasma: A static webassembly analysis framework for everyone. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 753–757. IEEE, 2023.
- [6] Mohak Chadha, Nils Krueger, Jophin John, Anshul Jindal, Michael Gerndt, and Shajulin Benedict. Exploring the Use of WebAssembly in HPC. *null*, 2023.
- [7] Mohak Chadha, Nils Krueger, Jophin John, Anshul Jindal, Michael Gerndt, and Shajulin Benedict. Exploring the use of webassembly in hpc. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 92–106, 2023.
- [8] Abhinav Chennubhotla, Siddharth Tewari, Harsh Somvanshi, Chinthana M.J, and D. C. Moving Past Docker for Serverless: WebAssembly for Kubernetes Clusters at the Edge. *2025 10th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, 2025.
- [9] Suchismita Das. Container Orchestration and Kubernetes Enhancements. *International Journal for Sciences and Technology*, 2025.
- [10] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In

- Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2019.
- [11] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE, 2015.
 - [12] Diego Ferrari, Eugenio Sciascio, Roberto Gioiosa, Osvaldo Krieger, Konstantinos Krikellas, and Giovanni Giuliani. Webassembly for cloud-native applications: A performance evaluation. *Future Generation Computer Systems*, 115:335–348, 2021.
 - [13] Markus Fiedler, Hossain Alrowaily, Karoline Menzel, and Burkhard Stiller. Performance evaluation of webassembly as a cloud-edge continuum technology. In *2020 IEEE 6th International Conference on Network Softwarization (NetSoft)*, pages 334–340. IEEE, 2020.
 - [14] Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, and Pramod Bhatotia. Cage: Hardware-accelerated safe webassembly. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 538–552, 2025.
 - [15] Daigo Fujii, Katsuya Matsubara, and Yuki Nakata. Stateful vm migration among heterogeneous webassembly runtimes for efficient edge-cloud collaborations. In *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking*, pages 19–24, 2024.
 - [16] Philipp Gackstatter, Pantelis A Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149. IEEE, 2022.
 - [17] Philipp Gackstatter, Philipp Gackstatter, Pantelis A. Frangoudis, Pantelis A. Frangoudis, Schahram Dustdar, and Schahram Dustdar. Pushing Serverless to the Edge with WebAssembly Runtimes. *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2022.
 - [18] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, pages 265–279, 2020.
 - [19] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615. IEEE, 2019.
 - [20] Sinka Gao, Guoqiang Li, and Hongfei Fu. Zkwasm: A zksnark wasm emulator. *IEEE Transactions on Services Computing*, 2024.

- [21] Panagiotis Gkonis, Anastasios Giannopoulos, Panagiotis Trakadas, Xavi Masip-Bruin, and Francesco D’Andria. A survey on iot-edge-cloud continuum systems: Status, challenges, use cases, and open issues. *Future Internet*, 15(12):383, 2023.
- [22] Tom Goethals, Maxim De Clercq, Merlijn Sebrechts, Filip De Turck, and Bruno Volckaert. Feather: Lightweight container alternatives for deploying workloads in the edge. In *CLOSER*, pages 27–37, 2024.
- [23] Raluca Maria Hampau, Maurits Kaptein, Robin Van Emden, Thomas Rost, and Ivano Malavolta. An empirical study on the performance and energy consumption of ai containerization strategies for computer-vision tasks on the edge. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pages 50–59, 2022.
- [24] Masatoshi Hidaka and Tatsuya Harada. WgPy: GPU-accelerated NumPy-like array library for web browsers. *null*, 2025.
- [25] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*, pages 2696–2708, 2021.
- [26] Devon Hockley and Carey Williamson. Benchmarking runtime scripting performance in webassembly. 2022.
- [27] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. Computing Without Borders: The Way Towards Liquid Computing. *IEEE Transactions on Cloud Computing*, 2022.
- [28] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of webassembly vs. native code. In *USENIX Annual Technical Conference*, pages 107–120, 2019.
- [29] Abhinav Jangda, Bobby Powers, Arjun Guha, and Emery Berger. Mind the gap: Analyzing the performance of webassembly vs. native code. *arXiv preprint arXiv:1901.09056*, 2019.
- [30] Matthijs Jansen, Maciej Kozub, Alexandru Iosup, and Daniele Bonetta. Memory Efficient WebAssembly Containers. *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2025.
- [31] Martin Kaiser, Rene Griessl, Nils Kucza, Carola Haumann, Lennart Tigges, Kevin Mika, Jens Hagemeyer, Florian Porrmann, Ulrich Rückert, Michav vor dem Berge, et al. Vedliot: Very efficient deep learning in iot. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 963–968. IEEE, 2022.
- [32] Sangeeta Kakati and M. Brorsson. A Cross-Architecture Evaluation of WebAssembly in the Cloud-Edge Continuum. *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2024.

- [33] Sangeeta Kakati and M. Brorsson. Performance and Usability Implications of Multiplatform and WebAssembly Containers. *International Conference on Cloud Computing and Services Science*, 2025.
- [34] Sangeeta Kakati and Mats Brorsson. Webassembly beyond the web: A review for the edge-cloud continuum. In *2023 3rd International Conference on Intelligent Technologies (CONIT)*, pages 1–8. IEEE, 2023.
- [35] Sangeeta Kakati and Mats Brorsson. An investigative study of webassembly performance in cloud-to-edge. In *2024 International Symposium on Parallel Computing and Distributed Systems (PCDS)*, pages 1–5, 2024.
- [36] Sangeeta Kakati and Mats Brorsson. The small container: Lightweight compute for multi-architecture cloud-edge systems. *SN Computer Science*, 6(8):1023, 2025.
- [37] Saif Eddine Khelifa, Miloud Bagaa, Oussama Bekkouche, Messaoud Ahmed Ouameur, and Adlen Ksentini. Extending WebAssembly for Deep-Learning Inference Across the Cloud Continuum. *IEEE Transactions on Network and Service Management*, 2025.
- [38] Saif Eddine Khelifa, Miloud Bagaa, Ahmed Ouameur Messaoud, and Adlen Ksentini. Case study of WebAssembly Runtimes for AI Applications on the Edge. *IEEE*, 2024.
- [39] Jinyeol Kim, Raehyeong Kim, Jongwon Oh, and Seung Eun Lee. Hardware-based webassembly accelerator for embedded system. *Electronics*, 13(20):3979, 2024.
- [40] Vojdan Kjorveziroski and Sonja Filiposka. WebAssembly as an Enabler for Next Generation Serverless Computing. *Journal of Grid Computing*, 2023.
- [41] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly as an enabler for next generation serverless computing. *Journal of Grid Computing*, 21(3):34, 2023.
- [42] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly orchestration in the context of serverless computing. *Journal of Network and Systems Management*, 31(3):62, 2023.
- [43] Vojdan Kjorveziroski, Sonja Filiposka, and Anastas Mishev. Evaluating webassembly for orchestrated deployment of serverless functions. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4. IEEE, 2022.
- [44] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 217–234, 2020.
- [45] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1045–1058, 2019.

- [46] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 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*, pages 261–272, 2022.
- [47] Zhexiong Li, Deze Zeng, and Ranzhao Chen. WebAssembly or Container? Joint Optimization of Microservice Consolidation and Deployment towards Cost Efficient Edge-End Consortium. *International Workshop on Quality of Service*, 2024.
- [48] Zhexiong Li, Deze Zeng, and Ranzhao Chen. Webassembly or container? joint optimization of microservice consolidation and deployment towards cost efficient edge-end consortium. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2024.
- [49] Mugeng Liu, Haiyang Shen, Yixuan Zhang, Hong Mei, and Yun Ma. WebAssembly for Container Runtime: Are We There Yet? *ACM Transactions on Software Engineering and Methodology*, 2025.
- [50] Mugeng Liu, Haiyang Shen, Yixuan Zhang, Hong Mei, and Yun Ma. Webassembly for container runtime: Are we there yet? *ACM Transactions on Software Engineering and Methodology*, 34(6):1–22, 2025.
- [51] Renju Liu, Luis Garcia, Zaoxing Liu, Botong Ou, and Mani Srivastava. Secdeep: Secure and performant on-device deep learning inference framework for mobile and iot devices. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pages 67–79, 2021.
- [52] Niko Mäkitalo, Victor Bankowski, Paulius Daubaris, Risto Mikkola, Oleg Beletski, and Tommi Mikkonen. Bringing webassembly up to speed with dynamic linking. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1727–1735, 2021.
- [53] Gabriele Proietti Mattia and Roberto Beraldi. Leveraging Reinforcement Learning for online scheduling of real-time tasks in the Edge/Fog-to-Cloud computing continuum. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, pages 1–9, November 2021. ISSN: 2643-7929.
- [54] Pankaj Mendki. Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*, pages 161–166, 2020.
- [55] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 205–216. IEEE, 2021.
- [56] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Watz: A trusted webassembly runtime environment with remote attestation for trust-zone. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 1177–1189. IEEE, 2022.

- [57] Jàmes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Webassembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, pages 3–8, 2022.
- [58] Kevin Mika, René Griessl, Nils Kucza, Florian Porrmann, Martin Kaiser, Lennart Tigges, Jens Hagemeyer, Pedro Trancoso, Muhammad Waqar Azhar, Fareed Qararyah, et al. Vedliot: Next generation accelerated aiot systems and applications. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 291–296, 2023.
- [59] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. Cloud Continuum: The Definition. *IEEE Access*, 2022.
- [60] Konrad Moron and Stefan Wallentowitz. Benchmarking WebAssembly for Embedded Systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2025.
- [61] Otoya Nakakaze, I. Koren, Florian Brilowski, and R. Klamma. Adaptive retrofitting for industrial machines: utilizing webassembly and peer-to-peer connectivity on the edge. *World wide web (Bussum)*, 2024.
- [62] Stefan Nastic, Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Deepak Vii, and Ying Xiong. Polaris Scheduler: Edge Sensitive and SLO Aware Workload Scheduling in Cloud-Edge-IoT Clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 206–216, September 2021. ISSN: 2159-6190.
- [63] Sou Oishi, Kazuya Ishikawa, Haruki Nogami, and Norishige Fukushima. Performance evaluation of image convolution with webassembly. In *International Workshop on Advanced Imaging Technology (IWAIT) 2023*, volume 12592, pages 492–497. SPIE, 2023.
- [64] Adrián Orive, Aitor Agirre, Hong-Linh Truong, Isabel Sarachaga, and Marga Marcos. Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications. *Sensors*, 22(5):1755, January 2022. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.
- [65] G. Palma, Saverio Giallorenzo, J. Mauro, Matteo Trentin, and G. Zavattaro. FunLess: Functions-as-a-Service for Private Edge Cloud Systems. *2024 IEEE International Conference on Web Services (ICWS)*, 2024.
- [66] Giuseppe De Palma, Saverio Giallorenzo, J. Mauro, Matteo Trentin, and G. Zavattaro. WebAssembly at the Edge: Benchmarking a Serverless Platform for Private Edge Cloud Systems. *IEEE Internet Computing*, 2024.
- [67] Xingguo Pang, Liu Liu, Yanze Zhang, Zhuofu Chen, Zhijun Ding, Dazhao Cheng, and Xiaobo Zhou. Featherlight Stateful WebAssembly for Serverless Inference Workflows. *IEEE Transactions on Parallel and Distributed Systems*, 2025.

- [68] Steven Pham, Kaue Oliveira, and Chung-Horng Lung. Webassembly modules as alternative to docker containers in iot application development. In *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, pages 519–524. IEEE, 2023.
- [69] Vasile Adrian Bogdan Pop, Arto Niemi, Valentin Manea, Antti Rusanen, and Jan-Erik Ekberg. Towards securely migrating webassembly enclaves. In *Proceedings of the 15th European Workshop on Systems Security*, pages 43–49, 2022.
- [70] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1274. IEEE, 2019.
- [71] Xiuquan Qiao, Pei Ren, Schahram Dustdar, Ling Liu, Huadong Ma, and Junliang Chen. Web ar: A promising future for mobile augmented reality—state of the art, challenges, and insights. *Proceedings of the IEEE*, 107(4):651–666, 2019.
- [72] Xiaojia Rao, Aina Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.*, 2023.
- [73] Partha Pratim Ray. An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, 15(8):275, 2023.
- [74] Matthias Rupp, Jonathan Schröter, and Stefan Wallentowitz. Accelerating webassembly interpreters in embedded systems through hardware-assisted dispatching. In *International Conference on Architecture of Computing Systems*, pages 207–220. Springer, 2024.
- [75] José Santos, Efstratios Reppas, Tim Wauters, B Volckaert, and Filip De Turck. Gwydion: Efficient auto-scaling for complex containerized applications in Kubernetes through Reinforcement Learning. *Journal of Network and Computer Applications*, 2024.
- [76] Merlijn Sebrechts, Tim Ramlot, Sander Borny, Tom Goethals, Bruno Volckaert, and Filip De Turck. Adapting kubernetes controllers to the edge: on-demand control planes using wasm and wasi. In *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, pages 195–202. IEEE, 2022.
- [77] Merlijn Sebrechts, Tim Ramlot, Sander Borny, Tom Goethals, Bruno Volckaert, and Filip De Turck. Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI. *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, 2022.
- [78] Boris Sedlak, Víctor Casamayor Pujol, Praveen Kumar Donta, and Schahram Dustdar. Equilibrium in the Computing Continuum through Active Inference. *Future generations computer systems*, 2023.

- [79] Shu Sekigawa, C. Sasaki, and Atsushi Tagami. Web Application-Based WebAssembly Container Platform for Extreme Edge Computing. *Global Communications Conference*, 2023.
- [80] Dimitrios Serpanos and Dimitrios Serpanos. Industrial Internet of Things: Trends and Challenges. *Computer*, 2024.
- [81] Xinyu She, Yanjie Zhao, and Haoyu Wang. Wadec: Decompiling webassembly using large language model. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 481–492, 2024.
- [82] Simon Shillaker, Simon Shillaker, Peter Pietzuch, and Peter Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. *arXiv: Distributed, Parallel, and Cluster Computing*, 2020.
- [83] Artem N Sisyukov, Vyacheslav Dobrynin, and Olga S Yulmetova. Web based gpu acceleration in embodied agent training workflow. In *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, pages 686–689. IEEE, 2021.
- [84] Benedikt Spies and Markus Mock. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10. IEEE, 2021.
- [85] Benedikt Spies, Christian Obermaier, and Jorg Ott. Exploring Composable Network Stacks from Isolated Components with WebAssembly and QUIC. *IEEE/IFIP Network Operations and Management Symposium*, 2025.
- [86] Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. The security risk of lacking compiler protection in webassembly. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 132–139. IEEE, 2021.
- [87] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. Leaps and bounds: Analyzing webassembly’s performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 256–268. IEEE, 2022.
- [88] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. Potential of webassembly for embedded systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4. IEEE, 2022.
- [89] Weihang Wang. Empowering web applications with webassembly: are we there yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1301–1305. IEEE, 2021.
- [90] Wenwen Wang. How far we’ve come—a characterization study of standalone webassembly runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 228–241. IEEE, 2022.
- [91] Zhen Wang, Jianda Wang, Zhendong Wang, and Yang Hu. Characterization and implication of edge webassembly runtimes. In *2021 IEEE 23rd Int Conf on*

- High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 71–80. IEEE, 2021.
- [92] Muhammad Waseem, Teerath Das, Aakash Ahmad, Peng Liang, and Tommi Mikkonen. Issues and their causes in webassembly applications: an empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 170–180, 2024.
- [93] Wasmtime Runtime. Wasmtime Runtime. <https://wasmtime.dev>, 2025. Accessed 2025.
- [94] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, pages 53–65, 2018.
- [95] Elliott Wen and Gerald Weber. Wasmachine: Bring the edge up to speed with a webassembly os. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 353–360. IEEE, 2020.
- [96] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 533–549, 2021.
- [97] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, et al. Bringing the webassembly standard up to speed with spectec. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1559–1584, 2024.
- [98] Zhao Yu, D. Zhan, Lin Ye, Haining Yu, Hongli Zhang, Zhihong Tian Harbin Institute Of Technology, and Guangzhou University. Exploring and Exploiting the Resource Isolation Attack Surface of WebAssembly Containers. *null*, 2025.
- [99] Wenxuan Zhao, Ruiying Zeng, and Yangfan Zhou. Wapplique: Testing webassembly runtime via execution context-aware bytecode mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1035–1047, 2024.
- [100] Yusheng Zheng, Tong Yu, Yiwei Yang, and Andrew Quinn. Wasm-bpf: Streamlining eBPF Deployment in Cloud Environments with WebAssembly. *arXiv.org*, 2024.