FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Efficient Serverless Computing with WebAssembly

Master Thesis

Submitted in partial fulfillment of the requirements for the degree of

Master of Science in Engineering

to the University of Applied Sciences FH Campus Wien
Master Degree Program: Software Design and Engineering

Author:
Sasan Jaghori, B.Sc

Student identification number:
2110838018

Supervisor:
Dipl.-Ing. Georg Mansky-Kummert

Date:
31.05.2023

Declaration of authorship:

I declare that this Master Thesis has been written by myself. I have not used any other than the listed sources, nor have I received any unauthorized help.

I hereby certify that I have not submitted this Master Thesis in any form (to a reviewer for assessment) either in Austria or abroad.

Furthermore, I assure that the (printed and electronic) copies I have submitted are identical.

Date: Signature:

Abstract

(E.g. “This thesis investigates ...”)

Kurzfassung

(Z.B. “Diese Arbeit untersucht...”)

List of Abbreviations

AOT	Ahead-of-time compilation
DOM	Document Object Model
JIT	Just-in-time compilation
LLVM	Low Level Virtual Machine
POSIX	Portable Operating System Interface
SIMD	Single Instruction Multiple Data
VM	Virtual Machine
WASM	WebAssembly
WASI	WebAssembly System Interface
WAT	WebAssembly Text Format
WIT	WebAssembly Interface Type

Key Terms

WASM
WebAssembly
Serverless
Edge Computing
Cloud Computing
AWS Lambda
FaaS

Contents

1 Introduction

In recent years, serverless computing has emerged as a popular paradigm for building scalable and cost-efficient cloud applications. Serverless platforms allow developers to focus on writing code without worrying about server management or infrastructure scaling, while users only pay for the computed time.

While serverless computing is an effective model for managing unpredictable and bursty workloads, it has limitations in supporting latency-critical applications in the IoT, mobile or gaming segments due to cold-start latencies of several hundred milliseconds or more [1].

An edge computing paradigm has emerged to address this issue, which places cloud providers closer to customers to improve latency. However, the issue of cold starts remains a challenge in edge computing. Furthermore, limited CPU power and memory resources in the host environment can further increase latencies, making achieving the desired performance for latency-critical applications difficult.

The underlying issue can be referred to the current runtime environments for serverless functions, which primarily rely on LXC-Container technologies like Docker or microVMs like Firecracker. To address these challenges, a more innovative approach would be to execute users' code efficiently and minimize cold start impacts, for instance by replacing heavier containers with lighter alternatives that maintain the advantages of container isolation.

The same requirements for isolation, fast execution and security apply to Wasm as well. Wasm (Wasm) has gained traction as a portable binary format that is executed in an isolated sandbox environment. Wasm is a compilation target, originally designed for browser applications to run e.g. C++ compiled code at near-native speed. This makes WebAssembly the perfect candidate for server-side code execution especially for serverless computing where startup times play a huge role. However, while Wasm was developed with browser execution in mind, standardized system APIs are needed to enable its use on the server side in order to access basic system resources like standard output or file input. Bytecode Alliance is working on the WebAssembly standard and a system interface standard called WASI [2].

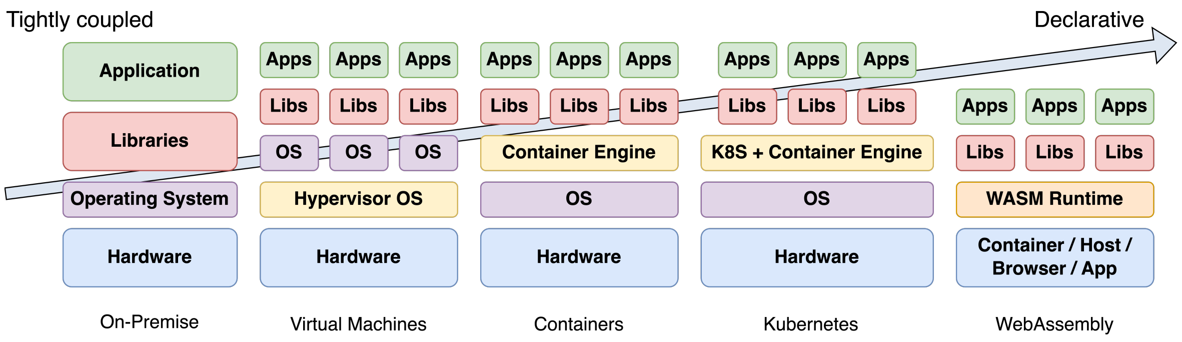

Figure 1.1: Evolution of computation from on-premise to Wasm based on [3]

The figure 1.1 below depicts the evolution of cloud computing from on premise model to the described WebAssembly serverless model.

Another promising browser technology is the usage of V8 isolates. According to Cloudflare, it achieves a startup time faster than a TLS handshake in under five seconds [4]. The downside of isolates is that they can only execute JavaScript code. This thesis will compare both runtime technologies, but will focus on WebAssembly based runtime technologies.

1.1 Research Objectives

The goal of this research project is to evaluate various factors that affect performance and usability, among other things. Factors such as containerization, selection of WebAssembly runtime, and programming language will be emphasized. Considerations will be made regarding vendor lock-in, and potential strategies for avoiding it will be evaluated. Additionally, the paper will address interesting aspects such as sustainability in terms of resource conservation in the serverless domain.

Based on these requirements, the thesis aims to answer the following research question:

"What are the effects of using WebAssembly technology in the serverless domain, particularly concerning the technologies employed (runtime environment, programming language), and how do they compare to existing solutions?"

1.2 Structure of the Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 describes the current state of WebAssembly, including the current specification and any implemented proposals, as well as future proposals. It also covers the WebAssembly System Interface (WASI).
- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7
- Chapter 8
- Chapter 9

2 WebAssembly (Wasm)

Since the dot-com era, JavaScript has remained the only client-side language for web browsers. As the web platform gained popularity and its standard APIs expanded, developers became increasingly interested in using faster programming languages on the web. To run these languages on the web, they had to be compiled into a common format, in this case, JavaScript. However, JavaScript, a high-level, dynamically typed, interpreted language, was not intended for this purpose, leading to performance issues.

In 2013, Mozilla engineers introduced a solution called asm.js, which focused on the parts of JavaScript that could be optimized ahead of time. This enabled C/C++ programs to be compiled into the asm.js target format and executed using a JavaScript runtime, achieving faster performance than equivalent JavaScript programs. However, benchmarks revealed that asm.js code ran about 1.5 times slower than the native code written in C++ [5].

As the need for improved web performance grew, asm.js was replaced by WebAssembly (wasm). Introduced in 2015, "WebAssembly (abbreviated Wasm), is a safe, portable, low-level code format designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be used in other environments as well [6, p. 1]." The Wasm standard is developed by W3C Working Group (WG) and W3C Community Group (CG).

2.1 Specification

This WebAssembly specification document [1] focuses on Wasm's core Instruction Set Architecture layer, defining the instruction set, binary encoding, validation, execution semantics, and textual representation. However, it does not specify how WebAssembly programs interact with the execution environment or how they are invoked within that environment.

From creating a proposal to fully integrating it into WebAssembly core specification, it must process through several phases. Each proposal follows this progression [2]:

- 0. Pre-Proposal [Individual]: An individual files an issue on the design repository to discuss the idea. The idea is then discussed and championed by one or more members. The proposal's general interest is voted on by the Community Group to ensure its scope and viability.
- 1. Feature Proposal [CG]: A repository is created, and the champion works on reaching broad consensus within the Community Group. The design is iterated upon, and prototype implementations may be created to demonstrate the feature's viability.
- 2. Feature Description Available [CG + WG]: A precise and complete overview document is produced with a high level of consensus. Prototype implementations create a comprehensive test suite, but updates to the reference interpreter and spec document aren't yet required.

[1]`https://webassembly.github.io/spec`
[2]`https://github.com/WebAssembly/meetings/blob/main/process/phases.md`

- 3. Implementation Phase [CG + WG]: WebAssembly runtimes implement the feature, the spec document is updated with full English prose and formalization, and the reference interpreter is updated with a complete implementation. Toolchains implement the feature, and any remaining open questions are resolved.
- 4. Standardize the Feature [WG]: The feature is handed over to the Working Group, which discusses the feature, considers edge cases, and confirms consensus on its completion. The Working Group periodically polls on the feature's "ship-worthiness." If only minor changes are needed, they are made; otherwise, the feature is sent back to the Community Group.
- 5. The Feature is Standardized [WG]: Once consensus is reached among Working Group members that the feature is complete, editors merge the feature into the main branch of the primary spec repository.

2.2 Proposals

In this section,

2.1

3

2.2.1 Multi-Value Return

In modern programming languages that support tuples, like Kotlin, Rust or Python, developers can effortlessly bundle several values into a single structure for returning from a function. Simple tasks, such as switching a pair of values or sorting an array, become challenging since they must be performed within the linear memory block. Some arithmetic functions, including modular operations and carry bits, can also yield multiple values.

Apart from function return values, another limitation in the WebAssembly MVP is that instruction sequences, such as conditional blocks and loops, cannot consume or return more than one result. It would be equally intriguing to exchange values, perform arithmetic with overflow, or receive a multi-value tuple response in these scenarios as well. Furthermore, compilers are no longer required to jump through hoops when generating multiple stack values for core WebAssembly. This results in smaller generated bytecode and consequently, faster loading times.

```
(module
  (func $swap (import "" "swap") (param i32 i32) (result i32 i32))

  (func $mySwapfunc (export "mySwapfunc") (param i32 i32) (result
     i32 i32)
    (call $swap (local.get 0) (local.get 1))
  )
)
```

Listing 2.1: Simple swap function that evaluates the multi return proposal

[3]https://github.com/WebAssembly/proposals

Phase 5 - The Feature is Standardized (WG)	
Proposal	**Champion**
Currently, there is no active Phase 5 proposal that has not been merged into the Wasm Core spec.	
Phase 4 - Standardize the Feature (WG)	
Tail call	Andreas Rossberg
Extended Constant Expressions	Sam Clegg
Phase 3 - Implementation Phase (CG + WG)	
Multiple memories	Andreas Rossberg
Custom Annotation Syntax in the Text Format	Andreas Rossberg
Memory64	Sam Clegg
Exception handling	Heejin Ahn
Web Content Security Policy	Francis McCabe
Branch Hinting	Yuri Iozzelli
Relaxed SIMD	Marat Dukhan & Zhi An Ng
Typed Function References	Andreas Rossberg
Garbage collection	Andreas Rossberg
Threads	Conrad Watt
JS Promise Integration	Ross Tate and Francis McCabe
Type Reflection for WebAssembly JavaScript API	Ilya Rezvov
Phase 2 - Proposed Spec Text Available (CG + WG)	
ECMAScript module integration	Asumu Takikawa & Ms2ger
Relaxed dead code validation	Conrad Watt and Ross Tate
Numeric Values in WAT Data Segments	Ezzat Chamudi
Instrument and Tracing Technology	Richard Winterton
Phase 1 - Feature Proposal (CG)	
Type Imports	Andreas Rossberg
Component Model	Luke Wagner
WebAssembly C and C++ API	Andreas Rossberg
Extended Name Section	Ashley Nelson
Flexible Vectors	Petr Penzin
Call Tags	Ross Tate
Stack Switching	Francis McCabe & Sam Lindley
Constant Time	Sunjay Cauligi, Garrett Gu, John Renner, Hovav Shacham, Deian Stefan, Conrad Watt
JS Customization for GC Objects	Asumu Takikawa
Memory control	Deepti Gandluri
Reference-Typed Strings	Andy Wingo
Profiles	Andreas Rossberg
Phase 0 - Pre-Proposal (CG)	
Currently, there is no active pre-proposal.	

Table 2.1: Active proposals in the WebAssembly CG and WG.

2.2.2 Reference Types

2.2.3 Bulk Memory Operations

2.3 Future proposals

2.4 Wasm bindgen

2.5 WebAssembly System Interface (WASI)

While WebAssembly is primarily designed for efficient operation on the web, as previously mentioned, it avoids making web-specific assumptions or integrating web-specific features. The core WebAssembly language remains independent of its surrounding environment and interacts with external elements exclusively through APIs. When operating on the web, it seamlessly utilizes existing web APIs provided by browsers. However, outside the browser environment, there is currently no standardized set of APIs for developing WebAssembly programs. This lack of standardization poses challenges in creating truly portable WebAssembly programs for non-web applications.

The WebAssembly System Interface (abbriveated as WASI) in short is a standard interface for WebAssembly modules to interact with their host environments, such as operating systems, without being tied to any specific host. This allows WebAssembly modules to be executed securely and portably across a wide range of environments.

According to Dan Gohman's proposed roadmap, the development of WASI is set to progress through Preview1, Preview2, and Preview3. Presently, the community is actively working on Preview2, ensuring that each milestone contributes to the ultimate goal of a stable and robust WASI 1.0 release.

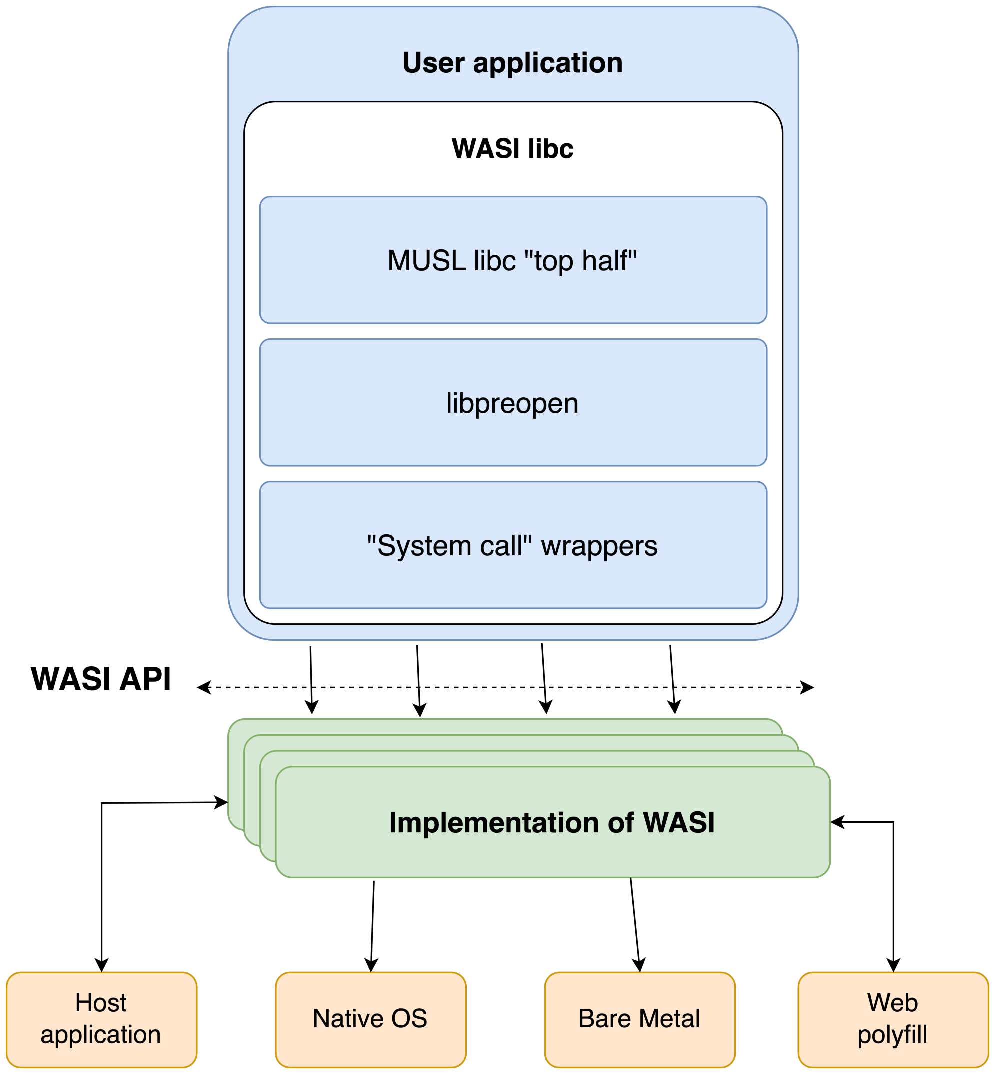

Figure 2.1: WASI Architecture

2.5.1 WASI as Emscripten replacement?

The first toolchain that enabled C, C++, or any other language with LLVM support to be compiled into WebAssembly was Emscripten. Essentially, Emscripten can compile almost any portable C or C++ codebase into WebAssembly, encompassing high-performance games requiring graphics rendering, sound playback, and file loading and processing, as well as application frameworks like Qt. Emscripten has been utilized to convert numerous applications like Unreal Engine 4 and the Unity engine, into WebAssembly [7].

To achieve this, Emscripten implemented the POSIX OS system interface on the web. As a result, developers are able to use the functions available in the C standard library (libc). Emscripten accomplished this by creating its own implementation of libc, which was divided into two parts. One part was compiled into the WebAssembly module, while the other part was implemented in JS glue code. The JS glue would subsequently communicate with the browser, which would in turn communicate with the Kernel and the OS.

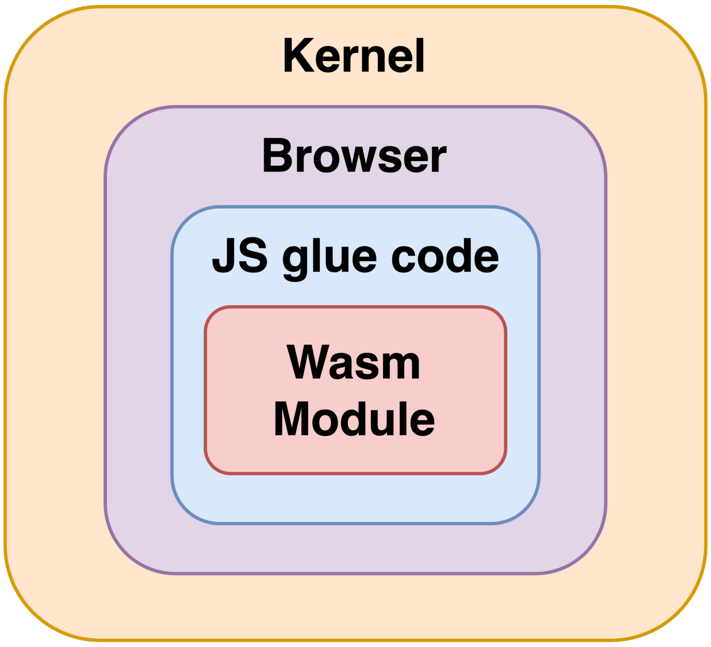

Figure 2.2: Structure of Emscripten compiled Wasm module

When users began to seek ways of running WebAssembly outside of a browser environment, the first approach was to enable the execution of Emscripten-compiled code on the corresponding environment. To achieve this, the runtimes had to develop their own implementations of the functions in the JS glue code. However, this presented a challenge as the interface provided by the JS glue code was not intended to be a standard or public-facing interface. It was designed to serve a specific purpose, and creating a portable interface was not part of its intended design. The above figure 2.2 shows the connection between the browser, the glue code and the Wasm module.

Emscripten aims to improve the standard by using WASI APIs as extensively as possible in order to minimize unnecessary API differences. As previously stated, Emscripten code accesses Web APIs indirectly via JavaScript on the web. By making the JavaScript API resemble WASI, an unnecessary API difference can be eliminated, allowing the same binary to run on a server as well. In simpler terms, if a Wasm application wants to log information, it must call into JavaScript, following a process similar to this:

```
wasm => function musl_writev(...) { ... console.log(...) ... }
```

"musl_writev" represents an implementation of the Linux syscall interface utilized by "musl libc" to write data to a file descriptor, ultimately leading to a call to console.log with the appropriate data. The Wasm module imports and invokes "musl_writev", thereby defining an ABI between the JavaScript and Wasm components. This ABI is arbitrary, by changing the existing ABI with one that aligns with WASI, the following can be achieved:

```
wasm => function __wasi_fd_write(...) { ... console.log(...) ... }
```

2.5.2 WASI Portability

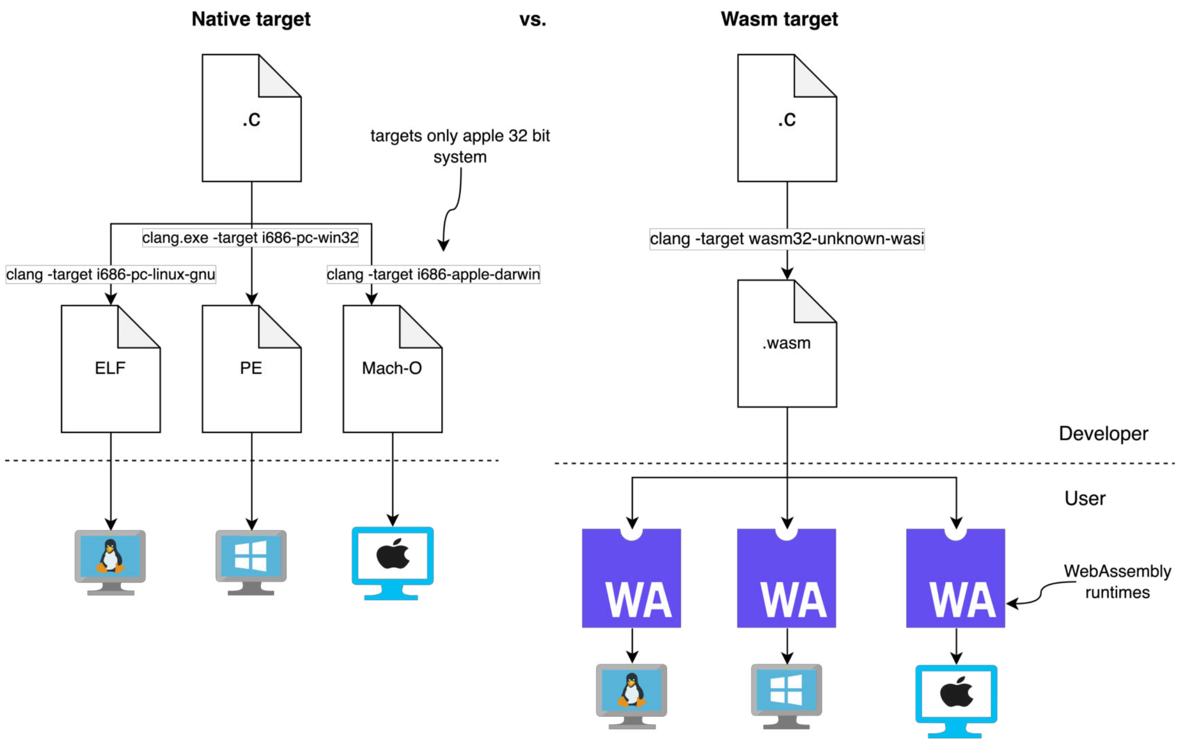
Native target
vs.
Wasm target
.C
targets only apple 32 bit system
clang.exe -target i686-pc-win32
clang -target i686-pc-linux-gnu
clang -target i686-apple-darwin
ELF
PE
Mach-O
.C
clang -target wasm32-unknown-wasi
.wasm
Developer
User
WA
WA
WA
WebAssembly runtimes

Figure 2.3: WASI portability model redrawn from [8]

2.5.3 WASI Security Model

An essential aspect of any system is the security model implemented to ensure the integrity and confidentiality of data and resources. When a code segment requests the operating system (OS) to perform input or output operations, the OS must ascertain whether the action is secure and permissible.

Operating systems generally employ an access control mechanism based on ownership and group associations to manage security. For instance, consider a scenario where a program seeks permission from the OS to access a specific file. Each user possesses a unique set of files they are authorized to access. In Unix systems, for example, the ownership of a file can be assigned to three classes of users: user, group, and others.

Upon initiating the program, it operates on behalf of the respective user. Consequently, if the user has access to the file—either as the owner or as a member of a group with access rights—the program inherits the same privileges. This approach to security helps maintain a robust and reliable system that adheres to the principles of access control and resource protection.

This approach was effective in multi-user systems where administrators controlled software installation, and the primary threat was unauthorized user access. However, in modern single-user systems, the main risk comes from the code that the user runs, which often includes third-party code of unknown trustworthiness. This raises the risk of a supply chain attack, particularly when new maintainers take over open-source libraries, as their unrestricted access could enable them to write code that compromises system security by accessing files or sending them over the network [8].

The security aspect of WASI is vital to the universal nature of WebAssembly. The WASI standard was built on a capabilities-based security model, which means the host has to explicitly permit capabilities such as file system access and establishing network sockets. As a result, a WASI module cannot run arbitrary code with direct access to memory.

3 Runtimes

3.1 Evaluation of features

3.1.1 JIT - Just In Time Compilation

3.1.2 Interpreter

3.1.3 WASM to native compilation

3.1.4 AOT - Ahead Of Time Compilation

3.1.5 Javascript snapshotting (Wizer)

3.2 Comparison to V8 Isolates

A V8 isolate is a separate instance of the V8 engine that has its own memory, garbage collector, and global object [9]. An isolate can run scripts in a safe and isolated environment, without interfering with other isolates [10]. An isolate also has its own state, which means that objects from one isolate cannot be used in another isolate. When V8 is initialized, a default isolate is created and entered, but you can also create your own isolates using the V8 API [9].

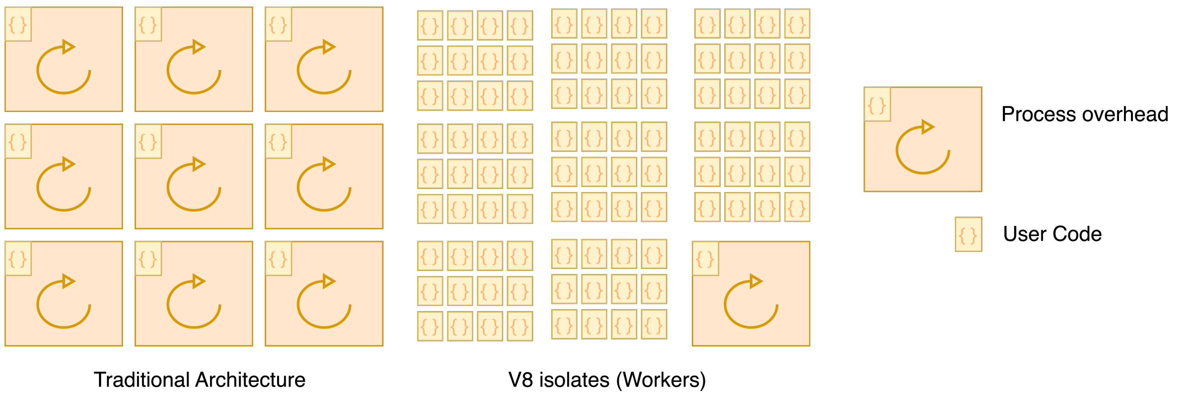

Figure 3.1: containerized vs. V8 isolates figure redrawn from [10]

3.2.1 Advantages and disadvantages

4 Simple POC WebAssembly Serverless Platform

In this chapter, the proof of concept for a simple serverless platform using WebAssembly technology will be explored. The platform utilizes an HTTP server to direct HTTP requests to specific WebAssembly modules. The popular Rust web framework, "Actix-web," has been chosen for the HTTP server implementation due to its lightweight and fast nature. Furthermore, the proof of concept employs Wasmtime as the Wasm-WASI runtime. As mentioned in the previous chapter, Wasmtime offers excellent support for the WASI standard and integrates various programming languages such as C++, Rust, Go, JavaScript, and more.

As Figure 4.1 illustrates, this proof of concept assumes the availability of a function registry service that stores the ahead-of-time compiled Wasm binaries. Additionally, a serverless platform needs to be distributed; therefore, API gateways and load balancers are necessary to serve requests across the network. However, this proof of concept primarily focuses on the WebAssembly aspect of the platform, leaving the API gateway and load balancers outside the scope of the discussion.

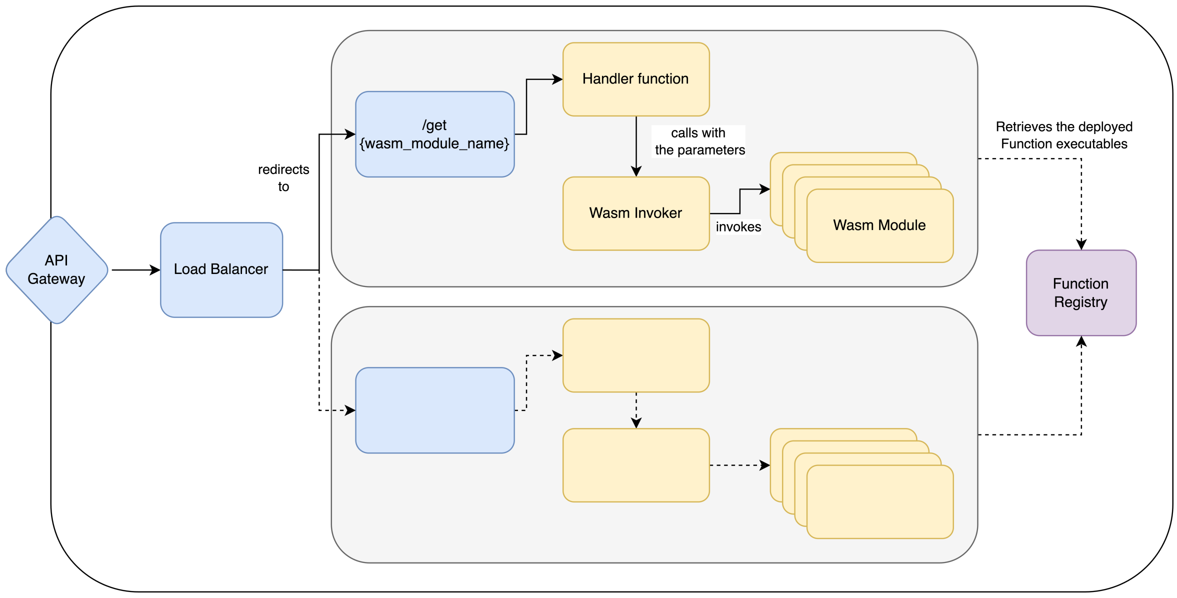

Figure 4.1: Execution flow of the simple POC serverless platform

```rust
#[actix_web::main]
async fn main() -> io::Result<()> {
    HttpServer::new(|| App::new().service(handler))
        .bind("127.0.0.1:8080")?
        .run()
        .await
}

#[get("/{wasm_module_name}")]
async fn handler(
    wasm_module_name: Path<String>,
    query: Query<HashMap<String, String>>,
) -> impl Responder {
    let wasm_module = format!("{}.wasm", wasm_module_name);
    match invoke_wasm_module(wasm_module, query.into_inner()) {
        Ok(val) => HttpResponse::Ok().body(val),
        Err(e) => HttpResponse::InternalServerError().body(format!(
            "Error: {}", e)),
    }
}

fn run_wasm_module(
    mut store: &mut Store<WasiCtx>,
    module: &Module,
    linker: &Linker<WasiCtx>,
) -> Result<()> {
    let instance = linker.instantiate(&mut store, module)?;
    let instance_main = instance.get_typed_func::<(), ()>(&mut
        store, "_start")?;
    Ok(instance_main.call(&mut store, ())?)
}
```

Listing 4.1: actix http server with a handler function to invoce the wasm modules

5 Benchmarks

6 Platform Limitations

The component model

7 Vendor lock-in

Cloud computing's vendor lock-in issue arises when customers become reliant (i.e., locked-in) on a specific cloud provider's technology implementation. This makes it challenging and costly to switch to another vendor due to legal constraints, technical incompatibilities, or significant expenses in the future. The issue of vendor lock-in in cloud computing has been identified as a major challenge because transferring applications and data to other providers is often an expensive and time-consuming process, making portability and interoperability essential considerations [11].

There are two primary factors contributing to the difficulty of achieving interoperability, portability, compliance, trust, and security in cloud computing. The first is the absence of universally adopted standards, APIs, or interfaces that can leverage the ever-changing range of cloud services. The second is the lack of standardized practices for deployment, maintenance, and configuration [12], which creates challenges for ensuring consistency and compatibility across cloud environments.

7.1 Key Motivations for shifting to a new Cloud Provider

There are various reasons why a customer may contemplate changing their service provider:

1. Inconsistent or unreliable service quality
2. Escalating costs associated with function execution, prompting a search for a more cost-effective alternative.
3. Runtime issues or limitations encountered with the current provider.
4. The need to integrate a function with a new back-end service that is only offered by a different provider.
5. Strategic or support considerations within the organization that may necessitate a switch to a different Cloud provider.

7.1.1 Service interoperability

7.1.2 Which parts need to be considered when migrating to a different provider?

To evaluate the feasibility of switching FaaS providers, it is necessary to examine the modifications needed for the function codebase as well as the adjustments required for the execution configuration, deployment, and triggers.

Differences in handler function

Each cloud computing service provider has its own unique function signature that must be adhered to for the code to be executed on their respective cloud platforms. These signatures can vary slightly between different providers. The subsequent Javascript code examples illustrate how input can be read from the event and responses can be sent back for each

cloud provider. These examples assume that the function is triggered through an HTTP POST request and with a JSON body conaining "myName" property.
The first example is an AWS Lambda Function, the body is within the event object, the function resolves the request by returning an object that contains a "statusCode" property along with a body.

```javascript
export const handler = async(event, context) => {
    const input = JSON.parse(event.body);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: `Hello ${input.myName}`,
        })
    };
};
```

Listing 7.1: Basic AWS Lambda Function

The next serverless function is running on Google Cloud, the difference is not only in the function signature, but also the fact that the function imports the framework.

```javascript
const functions = require("@google-cloud/functions-framework");

functions.http("/", (req, res) => {
  res.status(200).send({
        message: `Hello ${req.body.myName}`
    });
});
```

Listing 7.2: Basic Gen. 2 Google Cloud Functions

TODO: short description

```javascript
export default {
    async fetch(request, env, ctx) {
        const input = JSON.parse(await request.text());
        return new Response(JSON.stringify({
            message: `Hello, ${input.myName}`,
        }), {
            status: 200,
            headers: {
                "content-type": "application/json",
            },
        });
    },
};
```

Listing 7.3: Basic Cloudflare Workers

TODO: short description

```javascript
addEventListener("fetch", (event) => event.respondWith(
    handleRequest(event)));

```

```
async function handleRequest(event) {
  const request = event.request;
  const input = JSON.parse(await request.text());
  return new Response(JSON.stringify({
    message: `Hello, ${input.myName}`,
  }), {
    status: 200,
    headers: {
      "content-type": "application/json",
    },
  });
}
```

Listing 7.4: Basic Fastly Compute@Edge

7.2 Design principles

7.2.1 Facade pattern

The facade pattern can be used as a mitigation strategy to avoid or reduce serverless vendor lock-in. By creating a facade layer between the serverless function and the vendor-specific implementation, it is possible to decouple the function from the vendor's specific implementation details. The facade layer provides a simplified interface that abstracts the underlying vendor implementation, allowing developers to write code that is agnostic to the specific serverless vendor. If the vendor needs to be changed, the facade layer can be modified to adapt to the new vendor-specific implementation without affecting the business logic or the interface of the serverless function. This way, the codebase remains modular, and changing vendors becomes a relatively straightforward task.

7.2.2 Adapter pattern

SvelteKit is a modern web framework that effectively demonstrates the use of the Adapter pattern for deployment. Before deploying a SvelteKit application, it must be adapted to the specific deployment target by selecting an appropriate adapter in the configuration. This allows the application to be bundled with the platform-specific configuration [13]. The code snippet below illustrates the adapter configuration of a SvelteKit application, which enables the framework to be adapted to various cloud providers and allows the community to create new adapters. In this case, the deployment target is a Cloudflare Workers serverless function.

```
import adapter from '@sveltejs/adapter-cloudflare-workers';

/** @type {import('@sveltejs/kit').Config} */
const config = {
  kit: {
    adapter: adapter({
      // adapter options go here
    })
  }
};
```

```

export default config;
```

Listing 7.5: svelte.config.js SvelteKit adapter configuration

7.3 The role of Wasm Portability

8 Related Work

9 Conclusion

10 Future work

Future work ...

Bibliography

[1] S. Dustdar, Ed., *Pushing Serverless to the Edge with WebAssembly Runtimes*, IEEE. IEEE Computer Society, 05 2022. [Online]. Available: 10.1109/CCGrid54584.2022.00023 1

[2] WebAssembly, "WebAssembly/WASI: Webassembly system interface," GitHub, 03 2023. [Online]. Available: https://github.com/WebAssembly/WASI 1

[3] L. Randall, "Wasmcloud joins cloud native computing foundation as sandbox project | cosmonic," Cosmonic.com, 08 2021. [Online]. Available: https://cosmonic.com/blog/cosmonic-donates-wasmcloud-to-the-cloud-native-computing-foundation/ 1, 25

[4] A. Partovi, "Eliminating Cold Starts with Cloudflare Workers," The Cloudflare Blog, 07 2020. [Online]. Available: https://blog.cloudflare.com/eliminating-cold-starts-with-cloudflare-workers/ 2

[5] A. Zakai and R. Nyman, "Gap between asm.js and native performance gets even narrower with float32 optimizations – Mozilla Hacks - the Web developer blog," Mozilla Hacks – the Web Developer Blog, 12 2013. [Online]. Available: https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/ 3

[6] W. C. Group, "Webassembly specification (release 2.0 draft)," 03 2023. [Online]. Available: https://webassembly.github.io/spec 3

[7] E. Community, "Emscripten Documentation," emscripten.org, 03 2023. [Online]. Available: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html 7

[8] L. Clark, "Standardising WASI: a system interface to run webassembly outside the web," Mozilla Hacks – the Web Developer Blog, 03 2019. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ 9, 10, 25

[9] "Isolate V8 class reference," V8 Source Code Documentation, 10 2021. [Online]. Available: https://v8docs.nodesource.com/node-0.8/d5/dda/classv8_1_1_isolate.html 11

[10] C. Inc., "How Workers work · Cloudflare Workers Docs," Cloudflare, 01 2023. [Online]. Available: https://developers.cloudflare.com/workers/learning/how-workers-works/ 11, 25

[11] O. Guest, "Oracle brandvoice: Adopting cloud computing: Seeing the forest for the trees," Forbes, 09 2013. [Online]. Available: https://www.forbes.com/sites/oracle/2013/09/20/adopting-cloud-computing-seeing-the-forest-for-the-trees/ 16

[12] *Critical Review of Vendor lock-in and Its Impact on Adoption of Cloud Computing.* Research Gate, 11 2014. [Online]. Available: https://www.researchgate.net/publication/272015526_Critical_Review_of_Vendor_Lock-in_and_its_Impact_on_Adoption_of_Cloud_Computing 16

[13] S. Community, "Adapter Documentation SvelteKit," SvelteKit, 04 2023. [Online]. Available: https://kit.svelte.dev/docs/adapters 18

List of Figures

List of Tables

Appendix

```rust
use actix_web::{
    get,
    web::{Path, Query},
    App, HttpResponse, HttpServer, Responder,
};
use anyhow::Result;
use std::{
    collections::HashMap,
    io,
    sync::{Arc, RwLock},
};
use wasi_common::{pipe::WritePipe, WasiCtx};
use wasmtime::*;
use wasmtime_wasi::WasiCtxBuilder;

#[actix_web::main]
async fn main() -> io::Result<()> {
    HttpServer::new(|| App::new().service(handler))
        .bind("127.0.0.1:8080")?
        .run()
        .await
}

#[get("/{wasm_module_name}")]
async fn handler(
    wasm_module_name: Path<String>,
    query: Query<HashMap<String, String>>,
) -> impl Responder {
    let wasm_module = format!("{}.wasm", wasm_module_name);
    match invoke_wasm_module(wasm_module, query.into_inner()) {
        Ok(val) => HttpResponse::Ok().body(val),
        Err(e) => HttpResponse::InternalServerError().body(format!(
            "Error: {}", e)),
    }
}

fn run_wasm_module(
    mut store: &mut Store<WasiCtx>,
    module: &Module,
    linker: &Linker<WasiCtx>,
) -> Result<()> {
    let instance = linker.instantiate(&mut store, module)?;
    let instance_main = instance.get_typed_func::<(), ()>(&mut
       store, "_start")?;
    Ok(instance_main.call(&mut store, ())?)
}

fn invoke_wasm_module(wasm_module_name: String, params: HashMap<
    String, String>) -> Result<String> {
    // create a wasmtime engine
    let engine = Engine::default();

    let mut linker = Linker::new(&engine);
    wasmtime_wasi::add_to_linker(&mut linker, |s| s)?;

    // create a buffer to store the response
    let stdout_buf: Vec<u8> = vec![];
    let stdout_mutex = Arc::new(RwLock::new(stdout_buf));
    let stdout = WritePipe::from_shared(stdout_mutex.clone());

    // convert params hashmap to an array
    let envs: Vec<(String, String)> = params
        .iter()
        .map(|(key, value)| (key.clone(), value.clone()))
        .collect();

    let wasi = WasiCtxBuilder::new()
        .stdout(Box::new(stdout))
        .envs(&envs)?
        .build();
    let mut store = Store::new(&engine, wasi);

    let module = Module::from_file(&engine, &wasm_module_name)?;
    linker.module(&mut store, &wasm_module_name, &module)?;

    run_wasm_module(&mut store, &module, &linker).unwrap();

    // read the response into a string
    let mut buffer: Vec<u8> = Vec::new();
    stdout_mutex
        .read()
        .unwrap()
        .iter()
        .for_each(|i| buffer.push(*i));
    let s = String::from_utf8(buffer)?;
    Ok(s)
}
```

Listing 1: Simple Proof of Concept Wasm Serverless Platform using Actix and Wasmtime