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
Web Engineering

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].

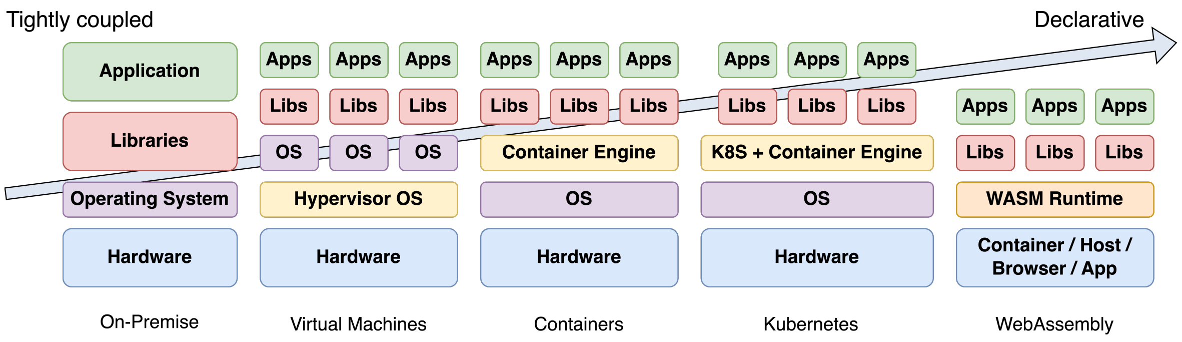

Figure 1.1: Evolution of computation from on-premise to Wasm based on [3]

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]."

2.1 WebAssembly Objectives

The WebAssembly standard is designed to meet the following objectives [7]:

- **Be fast and efficient**: WebAssembly modules can achieve near-native performance by having a compact binary format that is faster to decode compared to JavaScript parsing.
- **Ensure readability and debuggability**: The binary format is not intended to be read or written by humans, but it does have a text format that is easy to read and debug. The conversion from the text format to the binary format and vice versa is possible. More about this in section 2.2.1.
- **Maintain security**: The execution of WebAssembly modules are isolated from the host environment and other modules. Each module is executed in a sandboxed environment, meaning that the host environment has to explicitly grant access to resources outside the module, see section 2.5.3.
- **Preserve web compatibility**: The standard should not break the existing web APIs and it should maintain backward compatibility with older revisions of the standard.
- **Be hardware and platform independent**: The WebAssembly standard does not make any specific hardware or platform assumptions about the host environment, it is designed to take advantage of the common hardware capabilities. Platform specific features can be added through standard system interface extensions, see section 2.5.

2.2 WebAssembly Concepts

WebAssembly modules follow several major concepts that are necessary to mention before going through the details of the WebAssembly specifications. These concepts are [7]:

- **Module**: A WebAssembly module is a binary file that contains a sequence of sections. Each section has a unique identifier and a payload. The module can be loaded and executed by a WebAssembly runtime. Furthermore, a Wasm module does not have a state, thus stateless and can be shared between different threads and workers. Moreover, it consists of imports and exports similar to ES modules.
- **Memory**: In WebAssembly, memory is represented as a mutable, linear byte array that can dynamically grow in size. A Wasm memory can either be created within the module or imported from the host environment. A memory instance cannot be accessed by the host environment unless it is explicitly exported by the module or it was passed by the host initially. The size of a memory is measured in pages, a page has a size of 64KB.
- **Table**: A table is a data structure that holds a list of function references, which can be used to implement indirect function calls. Similar to a WebAssembly memory, tables can either be created within the module or imported from the host environment and it can grow in size. Important use cases for tables are indirect function calls and dynamic linking. Dynamic linking allows multiple modules to work together by sharing function references.
- **Instance**: An instance is a stateful, executable representation of a WebAssembly module. It consists of the module's imports, exports, memory, functions and table. An instance can be created by instantiating a module. The instance can be used to execute the module's functions and access its memory and table.

2.2.1 WebAssembly Text Format (WAT)

WebAssembly Text Format (WAT) [6] is a textual human-readable representation of a Wasm module. Unlike the binary representation of a Wasm module, which is designed to be efficient in size and fast to decode, the textual format is designed as an intermediate form for humans to read and understand or explain how the module works. The WAT format is not intended to be written by developers, but it is easier to understand and therefore, it is commonly used for specification descriptions and examples. Moreover, the WAT format has (`.wat`) file extension and there are tools like wabt[8] that can convert a WAT file into a Wasm binary (`.wasm`) file and vice versa.

The modules follow an S-expression format. S-expressions are simple textual formats representing a tree structure, where each parentheses `(...)` represents a node in the tree [9].

Listing 2.1 shows an example of an empty but valid Wasm module. The `module` keyword indicates the beginning of a module.

```
(module)
```

Listing 2.1: A WAT file containing an empty module

Moving on to a Wasm module that contains more functionality, listing 2.2 shows an internal function `$add_numbers` that takes two parameters of type `i32` and returns the sum of the two parameters. The function is then exported as `add_numbers` and can be called from the host runtime.

For this example, we use `i32` as the type of the parameters and the return value, however, WebAssembly supports `i64` and floating point numbers as well. WebAssembly does not support non primitive data types such as objects and strings, but there are proposals 2.3.2 to add ways to work with this types.

WebAssembly modules use stacks to pass parameters to functions and return values from functions. The stack is a fundamental data structure that follows the Last In First Out (LIFO) principle, meaning that the last item that was added to the stack is the first item to be removed. The stack is used to pass parameters to functions and return values from functions.

```
(module
  ;;internal function $add_numbers
  (func $add_numbers (param $num1 i32) (param $num2 i32) (result
   i32)
    ;; access the first parameter
    local.get $num1
    ;; access the second parameter
    local.get $num2
    ;; add the two numbers
    i32.add)

  ;;exported function add_numbers
  (export "add_numbers" (func $add_numbers))
)
```

Listing 2.2: A simple functions that adds two numbers and returns the value

In listing 2.2, the first `local.get` instruction is used to get the value of a local variable and push it to the stack. Then the second `local.get` pushes the second number to the stack. The `i32.add` instruction pops the two values from the stack, executes the operation, in this case it adds them together and pushes the result back to the stack. The `result` keyword is used to indicate the return value of the function. Figure 2.1 shows the explained process of adding the numbers 5 and 10.

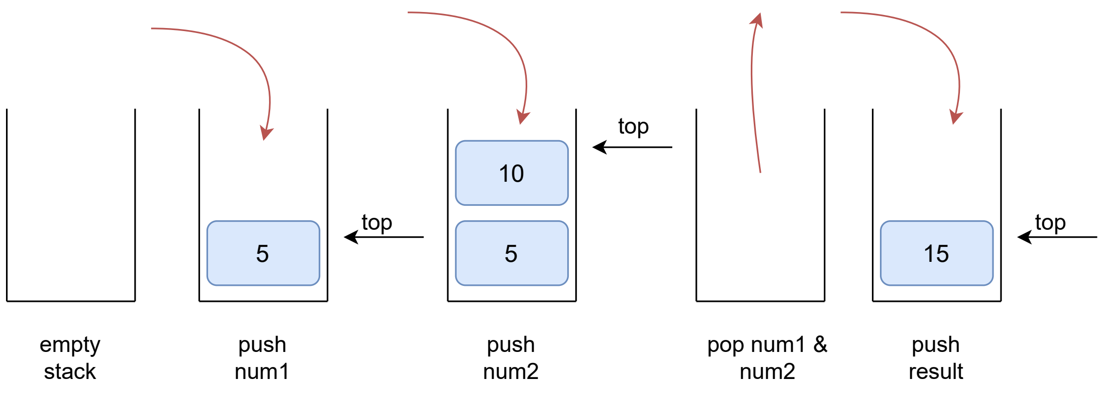

Figure 2.1: Visualization of how the stack is used to add two numbers in WebAssembly

2.3 WebAssembly Specification

The Wasm standard is developed by W3C Working Group (WG) and W3C Community Group (CG) [6]. The WebAssembly specification document [6] 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 [6].

From creating a proposal to fully integrating it into WebAssembly core specification, it must process through several phases. Each proposal follows this progression [10]:

- **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.
- **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.

Throughout these phases, proposals are refined, adjusted, and tested to ensure their seamless integration with the existing WebAssembly specification. The following table 2.1 lists the active proposals and their current phase. The proposal repository [11] contains more information about each proposal, including the proposal's current phase, champion, and links to the proposal's repository and design document, furthermore, the repository also contains a list of all the proposals that have been merged into the WebAssembly core specification.

2.3.1 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.

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 & 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.

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 and brings an extension type which is common in some programming languages like Rust or Python. Currently, the proposal has been already integrated into the WebAssembly core specification [11].

This proposal introduces a new type of arithmetic instruction "i32.divmod" which takes a numerator and divisor and returns the quotient and remainder. Moreover, it enables multiple values to stay on the stack without needing to be copied into linear memory.

The most effective way to demonstrate the proposal is by presenting a straightforward example. In example 2.3, we have a WAT file with an exported function called "reverseSub" that subtracts the second parameter from the first one and returns the result.

Firstly, we define a module that contains two functions. The first one is an internal function called "swap" that takes two i32 (32-bit integer) parameters and returns them in reverse order. The second function is the one that we want to export and is called "reverseSub". It takes two i32 parameters and returns an i32 value. An exported function, means it will be accessible outside the WebAssembly module. notice that "local.get 0" and "local.get 1" instructions are used to get the first and second parameters respectively. The "call $swap" instruction calls the "swap" function and passes the two parameters to it. Finally, the "i32.sub" instruction subtracts the second parameter from the first one and returns the result.

```
;; reverseSub.wat
(module
  (func $swap (param i32 i32) (result i32 i32)
    local.get 1
    local.get 0)

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

Listing 2.3: A reverse subtraction function that demonstrates the proposal for returning multiple values

2.3.2 Reference Types

Before the introduction of reference types, WebAssembly only supported four primitive value types: 32-bit integers, 64-bit integers, 32-bit floating-point numbers, and 64-bit floating-point numbers. With the introduction of reference types, WebAssembly capabilities are extended to include garbage-collected references. This will enable other proposals such as the garbage collection proposal, type import proposal 2.1 and more to utilize reference types without the need for glue code or dangerous workarounds. For instance the host stores objects in a side table and passes the indexes to the Wasm module. The Wasm module then uses these indexes to retrieve the objects from the side table. The usage of side tables requires glue code and is error-prone. Moreover, the glue code needs to be written in the host's language, making the Wasm module less portable.

The reference types proposal makes it possible for the host to specify and pass opaque handles to WebAssembly modules. These handles can be used to reference objects in the host environment, such as DOM nodes of a web page, a file handle or even open connection to a database.

The reference types proposal brings three new features:

- Makes it possible to have a `externref` type which is a opaque and unforgable reference to a object in the host environment.
- Makes it possible to store `externref` values in Wasm tables.
- Makes it possible to manipulate table entities with the help of new instructions.

As mentioned in the first bullet point, `externref` has two beneficial properties that fits the sandbox model of WebAssembly:

- **Opaque**: The `externref` type is opaque, meaning that the reference does not reveal any significant information about the object it is referencing or the memory layout of the host environment.
- **Unforgable**: The `externref` type is unforgable, meaning it can either return a null reference or the same reference that was passed to the Wasm module. This prevents the Wasm module from creating new references to objects in the host environment. It makes it impossible to forge the reference.

2.4 Garbage Collector Proposal

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.

The aim of WASI is to be a highly modular set of system interfaces, which includes both low-level and high-level interfaces like POSIX functions, neural networks, cryptography and so on. It is anticipated that more high-level APIs will be incorporated in the future depending on the priorities of the ecosystem. These interfaces must adhere to capability-based security principles to preserve the sandbox's integrity. Additionally, the interfaces should be portable across major operating systems, although system-specific interfaces may be acceptable for certain narrow use cases.

According to Dan Gohman's proposed roadmap, the development of WASI is set to progress through Preview1, Preview2, and Preview3. Currently, the community is actively working on Preview2, however, each stage will be backward compatible.

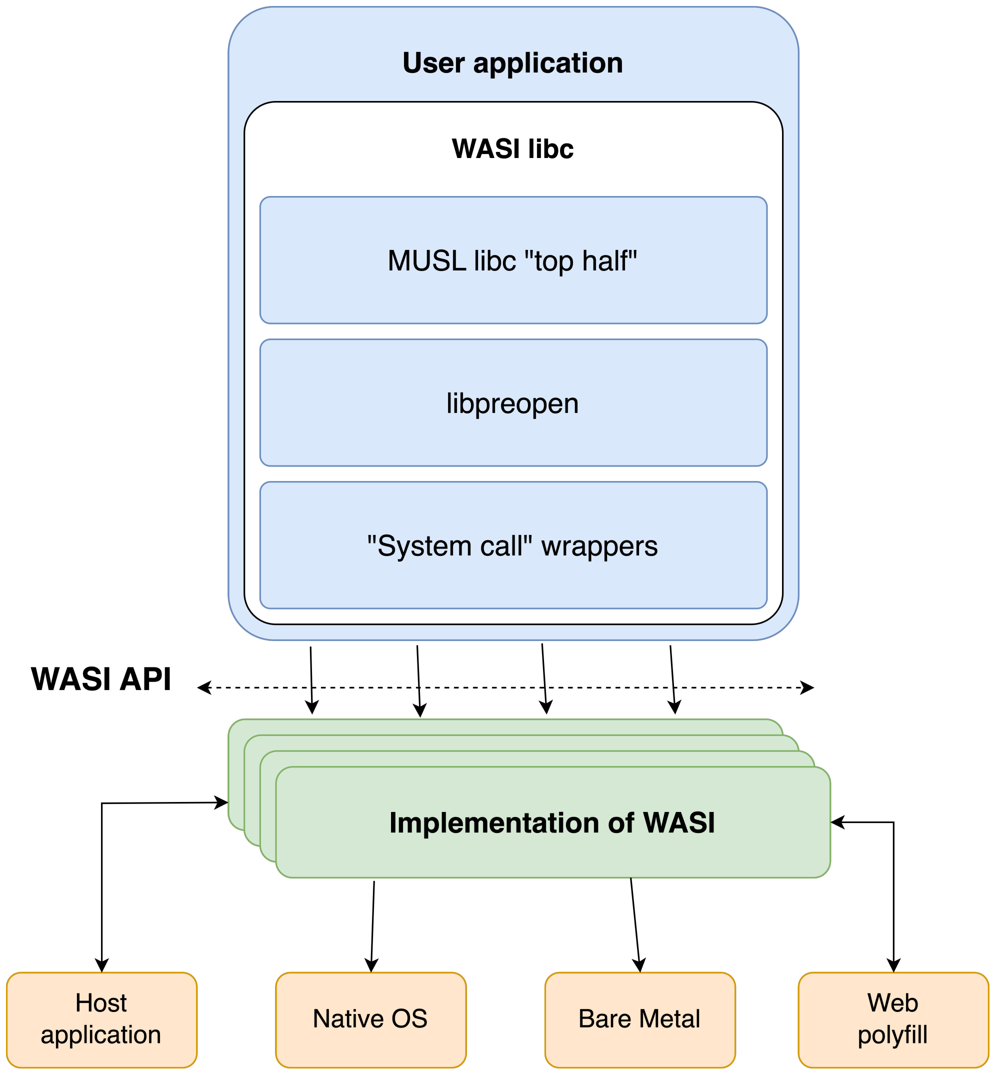

Figure 2.2: WASI Architecture

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 [12].

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.

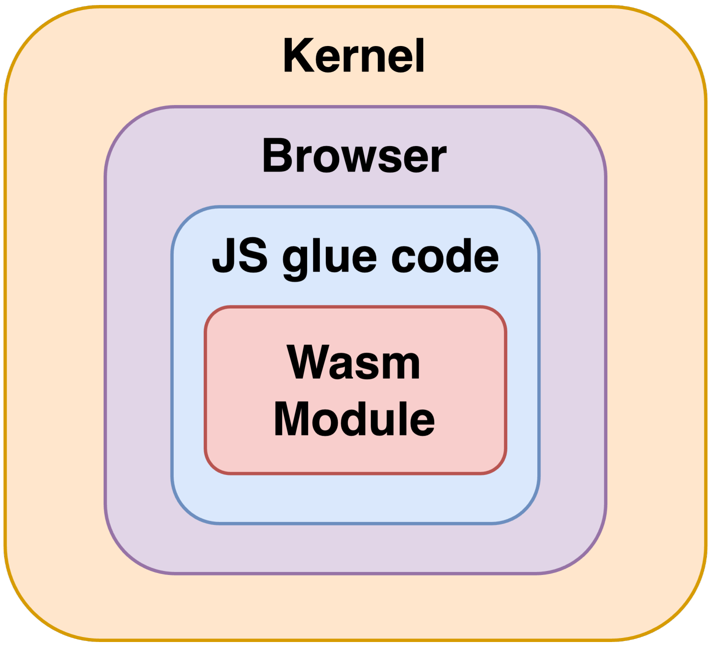

Figure 2.3: Structure of Emscripten compiled Wasm module

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 2.3 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

WebAssembly by design is a portable format, as mentioned before, the core specification does not make any platform specific assumptions. The first system API which the WASI community has been working on is the POSIX API. It involves essential functionalities like file operations, processes, threads, shared memory, networking, regular expressions and so on. WASI will consist of a collection of low level and high level APIs, which aim to be as portable as possible.

Figure 2.4 illustrates that a source code can be compiled with different versions of libc to target various platforms. However, in the case of WASI, the same source code can be compiled into a portable binary that targets a virtual host platform called "wasm32-unknown-wasi". This target comprises three parts: the underlying architecture "wasm32", the vendor (which usually specifies the platform or hardware vendor; in this case, it is "unknown"), and the operating system ("wasi").

Comparing the left and right sides of the figure, we can observe that the Wasm binary runs on a WASI runtime, also known as a WASI engine. The implementation details of the underlying OS are abstracted away from the Wasm binary targeting WASI. For instance, suppose a small C program that opens a file, reads its contents, and prints them to the console. In that case, on a Linux system, the program would call the "open" syscall. However, on WASI, the program would call the "wasi_fd_open" function, which is part of the WASI API. The WASI module expects the engine to create a file descriptor and return it to the module, which it can then use to read the file contents.

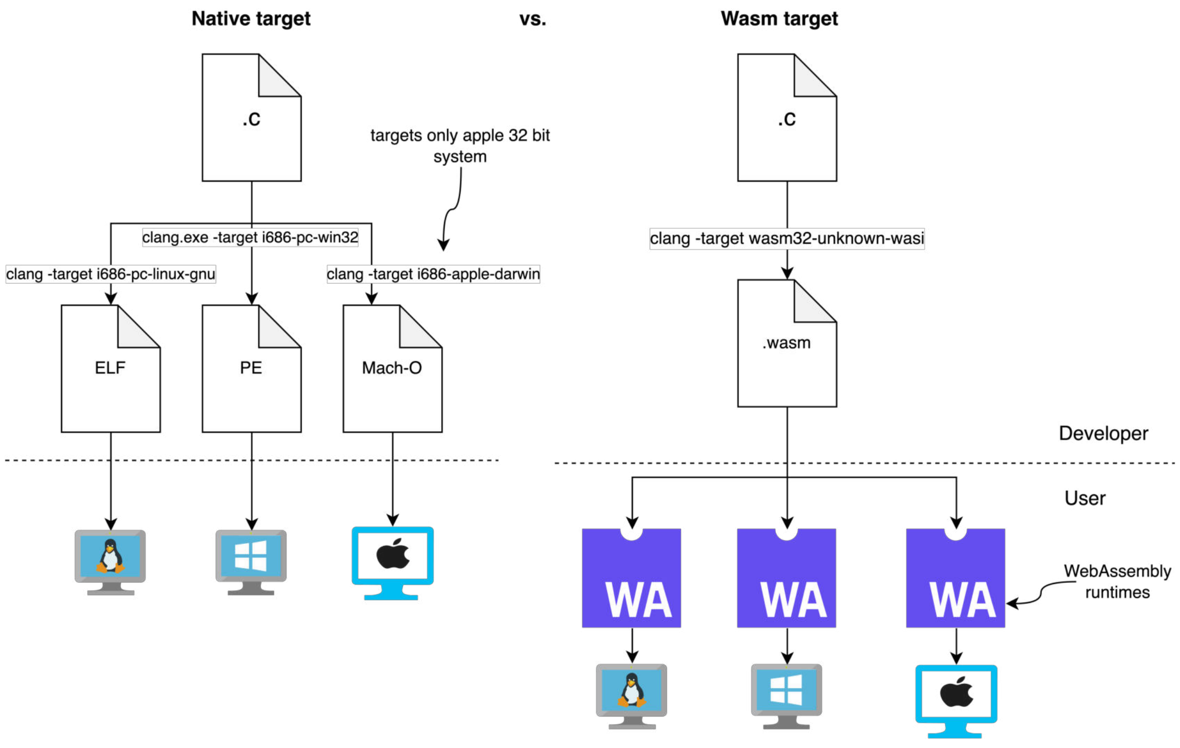
Native target
vs.
Wasm target
.C
targets only apple 32 bit system
.C
clang.exe -target i686-pc-win32
clang -target i686-pc-linux-gnu
clang -target i686-apple-darwin
clang -target wasm32-unknown-wasi
ELF
PE
Mach-O
.wasm
Developer
User
WA
WA
WA
WebAssembly runtimes

Figure 2.4: WASI portability model redrawn from [13]

Aside from the information provided in Figure 2.4, it should be noted that because of WASI's portability, the modules are not confined to a particular host platform. They have the flexibility to be integrated into an application, function as a plugin, or run as a serverless function in the cloud, as referenced in [14], as long as the host runtime supports the required APIs.

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 [13].

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.

In the context of serverless computing, for instance, a cloud provider may allow or deny access to specific functions, thus implementing security policies on a per-function basis. However, this level of granularity may not be sufficient. Since WebAssembly does not employ virtualization, all modules share common resources, such as the file system. As a result, the cloud provider may want to permit serverless functions to temporarily cache data in a specific "/tmp/<unique-id>/" directory while forbidding access to confidential files like "/etc/passwd". This is where WASI incorporates concepts from capability-based security.

3 Runtimes

Table 3.1

(Why different runtimes? Motivation?)

Runtime	**Embeddable langs.**	**Architecture**	**Compiler**	**Compilation**	**Platform**
Wasmtime	Rust, Python	x86, x86_64, ARM	Cranelift, LLVM	JIT, AoT	Windows, Linux, Mac OS
Wasmer	Rust, C++	x86, x86_64, ARM	Cranelift, LLVM, Singlepass	JIT, AoT	Windows, Linux, Mac OS, Free BSD, Android
Wasm3	Python3, Rust, GoLang, Zig, Perl, Swift, C# .NET	x86, x86_64, ARM, RISC-V, PowerPC, MIPS, Xtensa, ARC32	Custom	Interpreter	Windows, Linux, Mac OS, Free BSD, Android
WasmEdge	Solidity, Rust, C++, TinyGo, JavaScript, Python, Grain, Swift, Zig, Ruby	x86, x86_64, ARM	LLVM	Interpreter, AoT	Windows, Linux, Mac OS, Android

Table 3.1: WebAssembly Runtimes overview based on [15]

3.1 Serverless Requirements

The following summary of requirements should be met by an ideal serverless runtime, therefore, we will use them as a guideline to evaluate the different runtimes.

A serverless runtime should be able to:

1. Ensure security and isolation between functions.
2. Have a small footprint to reduce the startup time of functions, therefore, a small memory footprint and also fast startup time.
3. Integrate effortlessly with existing frameworks.
4. Support multiple programming languages but not less to the traditional serverless platforms.
5. Execute functions with a speed comparable to native speed.
6. Have the capability to run on various instruction set architectures, including but not limited to `x86_64` and `arm`.

7. Handle large amount of I/O operations concurrently.

3.2 Evaluation of features

3.3 Execution Models

3.3.1 JIT - Just In Time Compilation

3.3.2 Interpreter

3.3.3 WASM to native compilation

3.3.4 AOT - Ahead Of Time Compilation

3.3.5 Wasm Snapshots (Wizer)

As noted earlier, for serverless functions, having a fast startup time is crucial. Thus, techniques that can remove the repetitive initialization step from the critical path are highly beneficial. Wizer is a tool that can create a snapshot of an already initialized WebAssembly module. The snapshot is a pre-initialized Wasm module that should start faster without sacrificing any security or portability.

As shown in Figure 3.1, a program needs to go through four phases before it can be executed. The steps can be divided into two sections - the time of the developer spends to build and upload the program and the time of the user waiting for the program to become interactive on each request. The user experience can be enhanced by reducing the time spent in the second section.

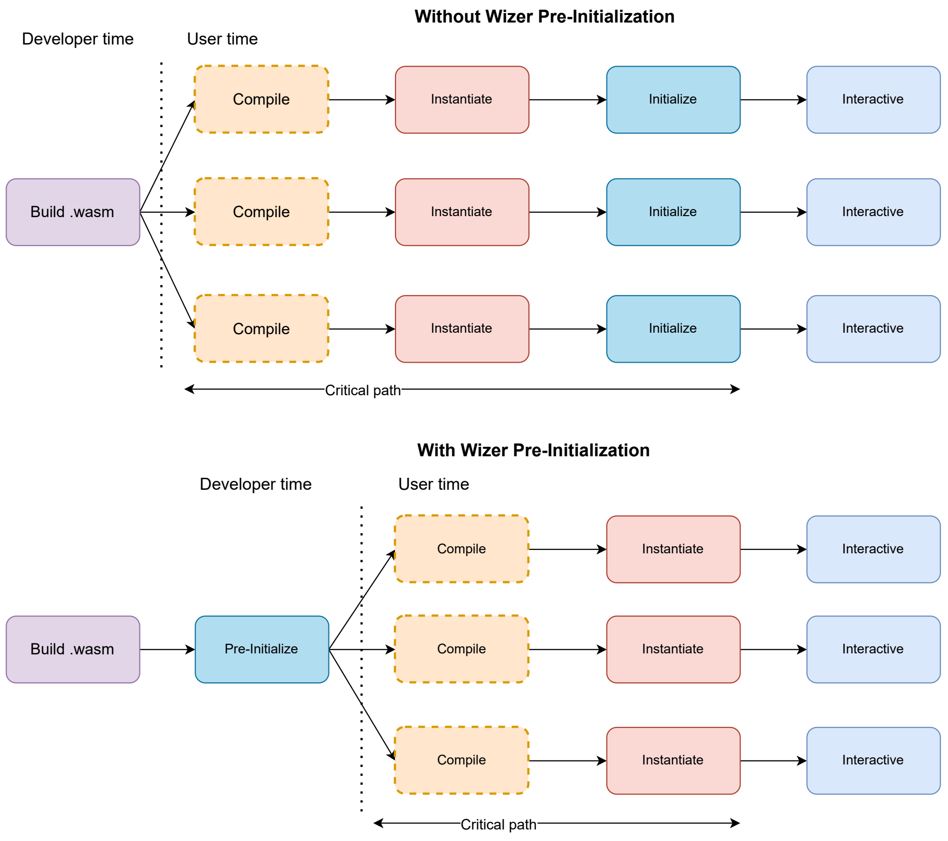

Figure 3.1: Overview of Pre-initialization vs. Non Pre-initialization, based on [16]

Wizer creates a pre-initialized snapshot through these four phases [16]:

1. Instrument: The first phase, known as the instrumentation phase, aims to export the internal state of the Wasm module so that Wizer can read it during the snapshot phase.
2. Initialize: In the second phase, Wizer uses the Wasmtime runtime to compile and initialize the Wasm module. It then calls the exported "wizer.initialize" function.
3. Snapshot: The third phase is the snapshot phase, in which Wizer reads the exports created in the first phase and generates the snapshot structure.
4. Rewrite: The final phase is the rewrite phase. Wizer takes the initial Wasm module and the snapshot to create a new Wasm module. It removes the start section, as the snapshot has already executed the initializations, and updates each memory's minimum size to the size of the snapshot memories, as they may have grown during the initialization phase.

Wizer benchmarks

Initial benchmarks indicate that Wizer can provide between 1.35 and 6.00 times faster instantiation and initialization, depending on the specific workload. Running the benchmarks included in the Wizer repository [17], we can confirm that Wizer can provide a significant speedup in the instantiation and initialization phases. For 3.2 we ran the User Agent Parsing benchmark from the Wizer repository. The benchmark creates a "RegexSet" using user agent parsing regexes from the Browserscope project. Then, it tests if the input string matches any known user agent patterns.

Our results on the User Agent Benchmark shows 14 times faster instantiation more than double the speedup of the original benchmark. We used the newest version of Wizer and Wasmtime ran on a M1 Pro MacBook with 32GB of RAM.

However, there is a caveat to this benchmark, it is important to note that not all programs will experience enhancements in instantiation and startup latency. For instance, Wizer can frequently enlarge the Data section of a Wasm module, potentially leading to adverse effects on network transfer times for web applications.

In the case of the User Agent Parsing benchmark, the orginal Wasm module is 3.1MB and the snapshot is 35.7MB. The size of the snapshot is 11.5 times larger than the original Wasm module. This is because the snapshot contains a large set of regexes in the data segment. In the context of edge computing, this could be a problem, as the larger the Wasm module, the longer it takes to transfer it to the edge hosts. Furthermore, both Cloudflare and Fastly cache the executable code across their global network of edge servers. Thus, making this solution on some instances less desirable.

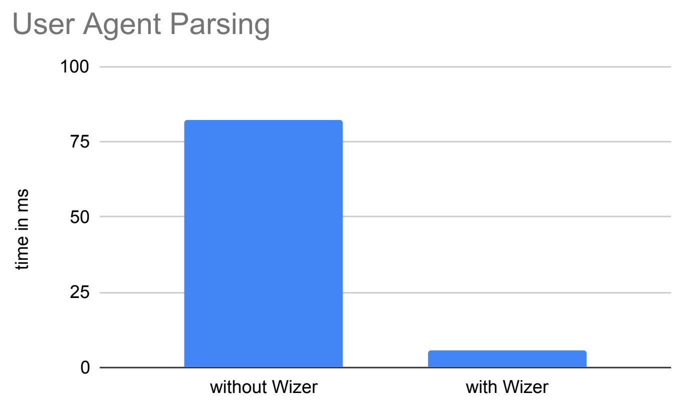

Figure 3.2: User Agent Parsing benchmark

3.4 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 [18]. An isolate can run scripts in a safe and isolated environment, without interfering with other isolates [19]. 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 [18].

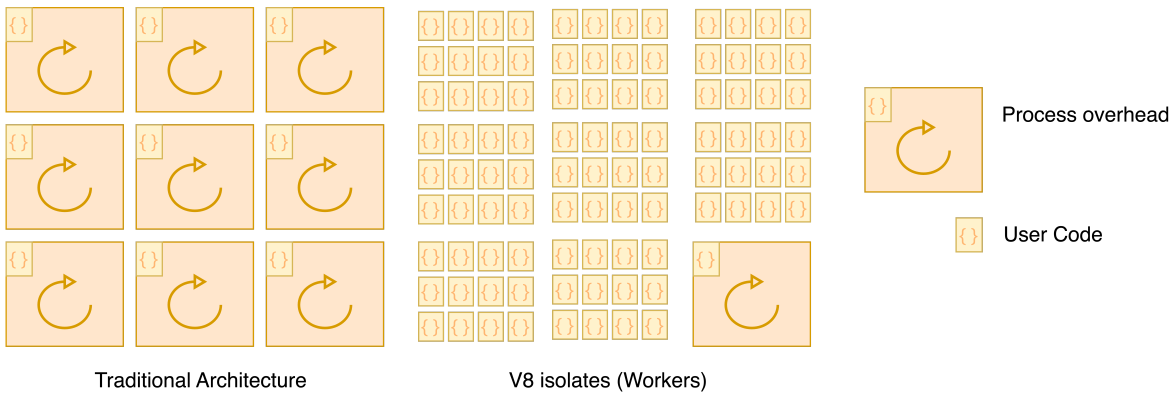

Figure 3.3: containerized vs. V8 isolates figure redrawn from [19]

3.4.1 Advantages and disadvantages

4 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.

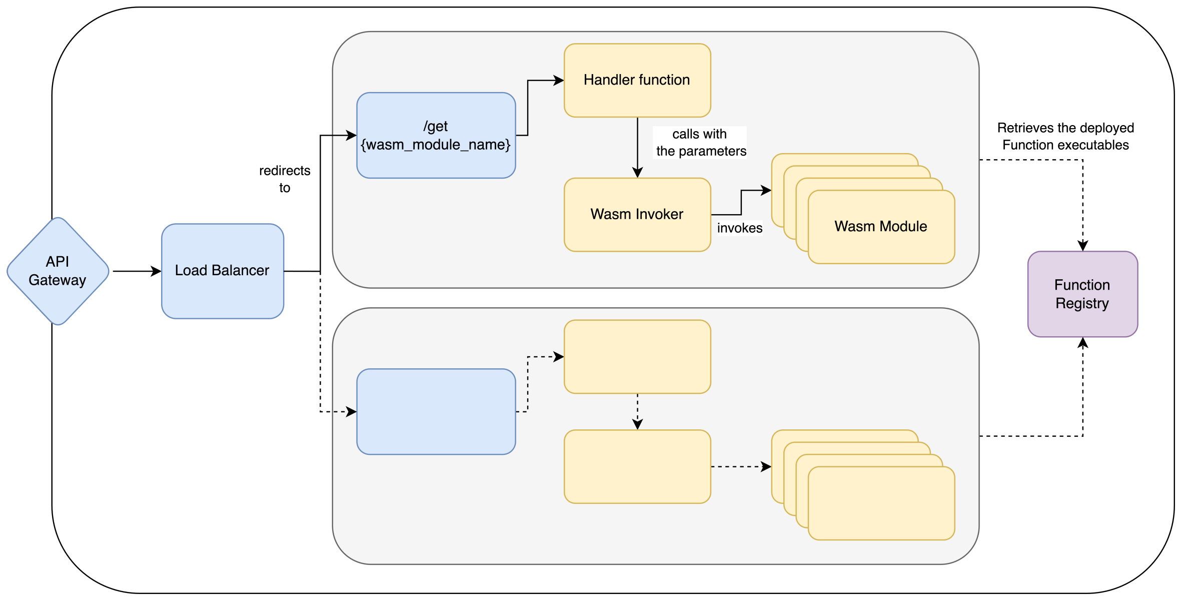

Figure 4.1: Execution flow of the simple POC serverless platform

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

```rust

#[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 Evaluation

5.1 Methodology

5.1.1 Goals

5.2 Setup

5.2.1 Programming Languages

5.3 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 [20].

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 [21], 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.

```
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.

```
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

```
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

```
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 [22]. 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, 34

[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, 4, 6

[7] M. Corporation, "WebAssembly Concepts - WebAssembly | MDN," MDN Web Docs, 03 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts 3, 4

[8] WebAssembly, "WebAssembly/wabt," Wabt project, 06 2020. [Online]. Available: https://github.com/WebAssembly/wabt 4

[9] M. Corporation, "Understanding WebAssembly text format," MDN Web Docs / WebAssembly, 02 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format 4

[10] W. . W3C, "WebAssembly W3C Proposal Process," WebAssembly / Meetings GitHub Repository, 05 2023. [Online]. Available: https://github.com/WebAssembly/meetings/blob/main/process/phases.md 6

[11] W. WG, "WebAssembly proposals," WebAssembly GitHub Page, 04 2023. [Online]. Available: https://github.com/WebAssembly/proposals 6, 8

[12] E. Community, "Emscripten Documentation," emscripten.org, 03 2023. [Online]. Available: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html 10

[13] 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/ 12, 13, 34

[14] ——, "Wasmtime reaches 1.0: Fast, safe and production ready!" Byte-

code Alliance, 09 2022. [Online]. Available: https://bytecodealliance.org/articles/wasmtime-1-0-fast-safe-and-production-ready 13

[15] S. Akinyemi, “Awesome WebAssembly Runtimes,” GitHub / Awesome WebAssembly Runtimes, 05 2023. [Online]. Available: https://github.com/appcypher/awesome-wasm-runtimes 14, 35

[16] N. Fitzgerald, “Hit the Ground Running: Wasm Snapshots for Fast Start Up,” fitzgeraldnick.com, 05 2021. [Online]. Available: https://fitzgeraldnick.com/2021/05/10/wasm-summit-2021.html 16, 34

[17] B. Alliance, “Wizer Github Repository,” GitHub, 04 2023. [Online]. Available: https://github.com/bytecodealliance/wizer 16

[18] “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 17

[19] C. Inc., “How Workers work · Cloudflare Workers Docs,” Cloudflare, 01 2023. [Online]. Available: https://developers.cloudflare.com/workers/learning/how-workers-works/ 17, 18, 34

[20] 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/ 23

[21] *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 23

[22] S. Community, “Adapter Documentation SvelteKit,” SvelteKit, 04 2023. [Online]. Available: https://kit.svelte.dev/docs/adapters 25

[23] P. Mell and T. Grance, “The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology,” 09 2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf 32

Glossary

cloud computing According to the National Institute of Standards and Technology (NIST) definition of cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models [23].. 1

edge computing Edge computing is the concept of bringing computation power closer to the end consumer. Due to the lower proximity between the server and the end device, this solution reduces network latency. Typical use cases are in the Internet of Things, mobile or gaming sectors where latency plays a huge role.. 1

FaaS FaaS stands for Function-as-a-Service, which is a cloud computing model where developers can create and deploy small, single-purpose functions that are executed on demand, in response to events or triggers.. 23

facade The Facade pattern is a software design pattern that provides a simplified interface to a larger body of code, making it easier to use and understand. It is a structural pattern that involves creating a single class, known as the facade, which acts as a front-facing interface for a complex system of classes and components.. 25

isolate The isolates runtime runs on the V8 engine, which is the same engine that powers Chromium and Node.js. The Workers runtime also supports many of the standard APIs that most modern browsers have.. 2, 18, 34

JS glue code It is the JavaScript code that bridges the gap between the compiled WebAssembly module and the browser. It is responsible for interfacing with the browser's JavaScript engine, allowing communication between the WebAssembly module and the web application.. 11

libc Short for C Standard Library, it is a library of standard functions that are a part of the C programming language, which define the functions used for file operations, input and output operations, string manipulation, and memory allocation etc.. 10

LLVM A toolkit used to build and optimize compilers. The LLVM Project consists of a set of modular and reusable compiler and toolchain technologies, which form a collection that can be utilized across various compilers and toolchains.. 10

LXC-Container LXC (Linux Containers) is a lightweight virtualization technology that allows multiple isolated Linux systems, known as containers, to run on a single Linux host. LXC-Containers provide a way to run applications in an isolated environment with their own file system, network interface, and resource allocation. Each LXC-Container shares the host operating system kernel but runs its own isolated user space.. 1

POSIX Portable Operating System Interface is a set of standards defined by the IEEE for maintaining compatibility between operating systems.. 9, 10, 12

serverless Serverless computing is a cloud computing model in which the cloud provider manages the infrastructure and automatically allocates computing resources as needed to execute and scale applications. In a serverless architecture, developers write and deploy code as small, independent functions that are triggered by specific events or requests, such as an HTTP request. The cloud provider then executes these functions on its own infrastructure, dynamically allocating computing resources and scaling automatically to meet demand. Because the cloud provider manages the infrastructure and abstracts away the underlying hardware and software, developers do not have to worry about managing servers, scaling infrastructure, or paying for idle resources, which can lead to reduced operational costs and increased agility.. 1, 24, 25

V8 V8 is a high-performance JavaScript engine developed by Google for use in their Chrome web browser and other applications. It is also used by Node.js to execute JavaScript code outside of a web browser. V8 is written in C++ and compiles JavaScript code to native machine code, providing significant performance benefits over interpreted JavaScript engines.. 18, 34

Wasm 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 employed in other environments as well.. 1, 3

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