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:
01.08.2023

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.

I have used the ChatGPT AI model for proofreading purposes, and every result was carefully evaluated by me before adaption; I affirm that no content was copied directly.

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

Date: Signature:

Acknowledgement

First and foremost, I would like to express my deepest gratitude to my supervisor Mr. Georg Mansky-Kummert for his support and guidance throughout the whole process of writing this thesis. His feedback and suggestions helped me greatly improve this thesis's quality.

Many thanks to my family for motivating me and supporting me the whole time.

Additionally, I would also like to thank my study colleagues for their support and teamwork during the last two years, who made this journey look easy.

Abstract

In recent years, serverless computing has emerged as a popular paradigm for building scalable and cost-efficient cloud applications. However, this paradigm was not sufficient for latency-critical applications in the IoT, mobile, or gaming segments due to cold-start latencies. An edge computing paradigm has emerged to address this issue, which places cloud provider servers closer to customers to improve latency. However, using existing virtualization techniques, the problem 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 it difficult to achieve the desired performance for latency-critical applications.

Because of these challenges, there is a need for an execution environment that doesn't have cold start latencies and is as secure as traditional containerization technologies and micro virtual machines such as Firecracker used by serverless cloud providers. One promising technology is WebAssembly. Initially designed as a common compilation target in the browser, it has shown that the same technology can be used in the server environment as well. It is lightweight, language-agnostic, cross-platform, secure, and fast.

The aim of this thesis is to assess whether the usage of WebAssembly technology will satisfy the challenging requirements of the edge computing paradigm, allowing us to build applications that depend on fast response times. For this, we evaluate the performance of WebAssembly engines and consider various factors that affect performance and usability, among other things.

We use both microbenchmarks and macrobenchmarks to evaluate the performance of WebAssembly runtimes and existing serverless offerings. Through microbenchmarks, we find that the instantiation time is about 40 microseconds compared to alternatives such as Firecracker's microVM, which takes about 125 milliseconds. The results demonstrate that using WebAssembly in the serverless environment represents significant progress towards enabling a wide range of latency-critical applications.

Kurzfassung

In den letzten Jahren hat sich das serverlose Computing zu einem beliebten Paradigma für die Erstellung skalierbarer und kosteneffizienter Cloud-Anwendungen entwickelt. Dieses Paradigma war jedoch für latenzkritische Anwendungen im IoT-, Mobil- oder Spielesegment aufgrund von Kaltstartlatenzzeiten nicht ausreichend. Zur Lösung dieses Problems hat sich ein Edge-Computing-Paradigma herausgebildet, bei dem die Server der Cloud-Anbieter näher am Kunden platziert werden, um die Latenzzeit zu verbessern. Bei der Verwendung bestehender Virtualisierungstechniken bleibt das Problem der Kaltstarts jedoch eine Herausforderung beim Edge Computing. Darüber hinaus können begrenzte CPU-Leistung und Speicherressourcen in der Host-Umgebung die Latenzzeiten weiter erhöhen, was es schwierig macht, die gewünschte Leistung für latenzkritische Anwendungen zu erreichen.

Aufgrund dieser Herausforderungen besteht ein Bedarf an einer Ausführungsumgebung, die keine Kaltstartlatenzen aufweist und ebenso sicher ist wie herkömmliche Containerisierungstechnologien und "microVMs" wie Firecracker, die von serverlosen Cloud-Anbietern verwendet werden. Eine vielversprechende Technologie ist WebAssembly. Ursprünglich als gemeinsames Kompilierungsziel im Browser entwickelt, hat sie gezeigt, dass dieselbe Technologie auch in der Serverumgebung eingesetzt werden kann. Sie ist leichtgewichtig, sprachunabhängig, plattformübergreifend, sicher und schnell.

In dieser Arbeit soll untersucht werden, ob die Verwendung der WebAssembly-Technologie den anspruchsvollen Anforderungen des Edge-Computing-Paradigmas gerecht wird und es uns ermöglicht, Anwendungen zu entwickeln, die auf schnelle Antwortzeiten angewiesen sind. Zu diesem Zweck bewerten wir die Leistung von WebAssembly-Engines und berücksichtigen verschiedene Faktoren, die unter anderem die Leistung und die Benutzerfreundlichkeit beeinflussen.

Wir verwenden sowohl Mikrobenchmarks als auch Makrobenchmarks, um die Leistung von WebAssembly-Laufzeiten und bestehenden serverlosen Angeboten zu bewerten. Durch Mikrobenchmarks finden wir heraus, dass die Instanziierungszeit etwa 40 Mikrosekunden beträgt, verglichen mit Alternativen wie Firecrackers microVM, die etwa 125 Millisekunden benötigt. Die Ergebnisse zeigen, dass die Verwendung von WebAssembly in der Serverless-Umgebung einen bedeutenden Fortschritt bei der Ermöglichung eines breiten Spektrums an latenzkritischen Anwendungen darstellt.

List of Abbreviations

API	Application Programming Interface
AS	AssemblyScript
AOT	Ahead-of-time compilation
DOM	Document Object Model
ICMP	Internet Control Message Protocol
JIT	Just-in-time compilation
JS	JavaScript
LLVM	Low Level Virtual Machine
POSIX	Portable Operating System Interface
SIMD	Single Instruction Multiple Data
TLS	Transport Layer Security
VM	Virtual Machine
WASI	WebAssembly System Interface
WASM	WebAssembly
WAT	WebAssembly Text Format

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 computing 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 WebAssembly as well. WebAssembly (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].

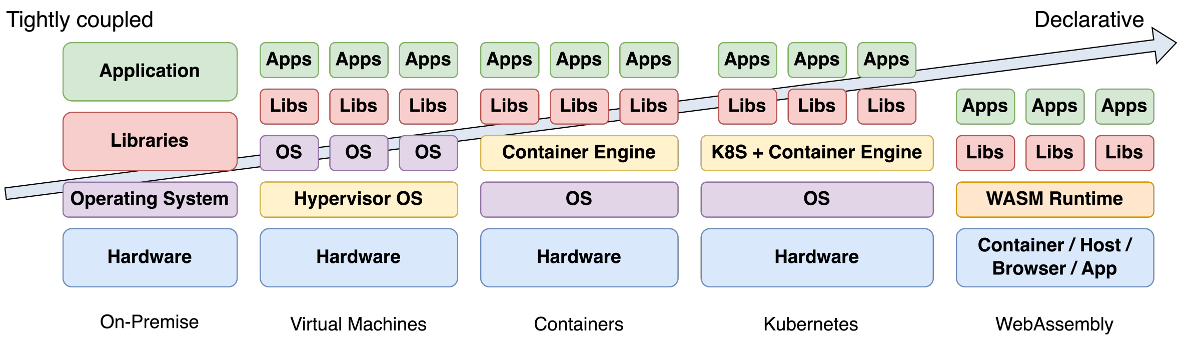

Figure 1.1: Evolution of computation from on-premise to Wasm based on [3]

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 an overview of the current active WebAssembly proposals and the explanation of three important proposals. It also covers the WebAssembly System Interface (WASI).
- **Chapter 3** provides an overview of the existing WebAssembly runtimes. It also compares V8 isolates to WebAssembly runtimes.
- **Chapter 4** explores the design of a simple WebAssembly compute platform. It also describes the implementation of the platform and evaluates the initial performance results.
- **Chapter 5** includes the evaluation, starting with the methodology and setup, followed by the benchmarks and the results.
- **Chapter 6** discusses the vendor lock-in issues in a serverless environment. The chapter discusses design principles that can be used to mitigate some of the vendor lock-in issues.
- **Chapter 7** presents an overview of the existing research and developments in the field of WebAssembly-based serverless platforms.
- **Chapter 8** concludes the thesis. It also discusses the aspects of platform limitations and sustainability in regards to the environment.
- **Chapter 9** outlines the future work that can be carried out based on the content of this thesis.

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. 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 subsection 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 subsection 2.7.4.
- **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 2.7.

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 [6, sec. 1.2.1]. Moreover, it contains imports and exports similar to ES modules [8].
- **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 [6, sec. 4.2.8].
- **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 [6, 9].
- **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 [6, sec. 1.2.2].

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[10] 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 [11].

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` (32 bit integers) 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 32 bit integer (`i32`) as the type of the parameters and the return value, however, WebAssembly supports 64 bit integers (`i64`) and floating point numbers as well. WebAssembly does not support non primitive data types such as objects and strings, but there are proposals 2.5 to add ways to work with this types.

WebAssembly modules use stacks [6, sec. 4.2.14] 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 resulting in 15.

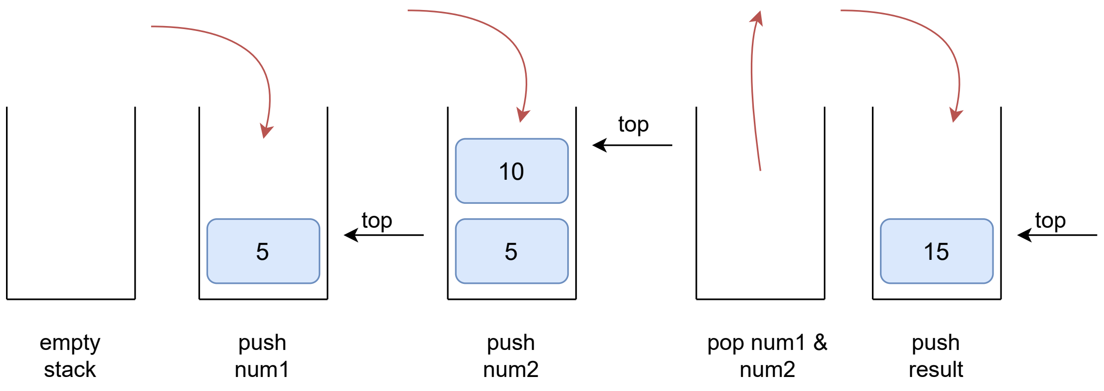

Figure 2.1: Visualization of how the stack is used to add two numbers in WebAssembly

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].

The WebAssembly specification gets extended through proposals. From creating a proposal to fully integrating it into WebAssembly core specification, it must process through several phases. Each proposal follows this progression [12]:

- **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 of WebAssembly.

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 [13] 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.

In the next three sections, we will elaborate two proposals that have been already integrated into the WebAssembly core specification and one that is still an active proposal. The first proposal is the multi-value return, which allows functions to return multiple values a common feature in modern programming languages. The second proposal is the reference types, which allows WebAssembly to handle complex data types [14]. The third proposal is the Wasm GC proposal, which depends on the reference types proposal and adds garbage collection to WebAssembly and thereby expanding the range of languages that can be compiled to

WebAssembly. These proposals were especially chosen because they make the WebAssembly target more accessible to a broader range of languages and use cases.

2.4 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.

Aside from functions that return tuples or multiple values, another limitation in the WebAssembly MVP was that loops and conditional blocks, in the code base cannot process or return more than one result [9]. 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 [13].

This proposal introduces a new type of arithmetic instruction "i32.divmod" [15, 6] 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 32 bit integer (i32) parameters and returns an integer (i32) result 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 [6] subtracts the second parameter from the first one and returns the result.

```
(module ;; reverseSub.wat
  (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 Wasm module with a reverse subtraction function returning 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 obtained from [13]

2.5 Reference Types

Before the introduction of reference types, WebAssembly only supported four primitive value types [6]: 32-bit integers, 64-bit integers, 32-bit floating-points, 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 [14] such as the garbage collection proposal, type import proposal (see Table 2.1) and more to utilize reference types without the need for glue code or dangerous workarounds.

For instance, as Nick Fitzgerald noted in his publication [16], 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.

2.6 Garbage Collection Proposal (Wasm GC)

The garbage collection proposal [17] is currently under active development. This is one of the most anticipated proposals since it will enable developers to use managed-memory languages such as Kotlin, Java, Dart and many more to the WebAssembly ecosystem.

Prior to this proposal, managed-memory languages have to ship and instantiate their own garbage collector every time the app loads, thus, increasing the Wasm module size and increasing the startup time, even when every standard browser already has a garbage collector. Figure 2.2 illustrates this problem, marked as "#1 bloat problem", very well. Another issue is the fact that developers need to know how large the heap is going to be, to avoid running out of memory. A typical thing todo is to allocate a large amount of memory and hope that it is enough. This architecture treats the Wasm module as a separate entity from the host environment, even though it might reference objects in the host environment. This brings us to the next issue, marked as "#2 split brain problem", there is no guarantee that the references in the JavaScript Heap are still valid, because there is no way to inform the garbage collector that the Wasm module is no longer using the reference or vice versa. This is a problem because the garbage collector might free the memory that the reference is pointing to, which will result in a dangling pointer.

Even supposing that developers keep the references on the Wasm module side, there is still the possibility that we might need the JavaScript heap, due to the fact that the Wasm module might need to call Web APIs. Web APIs accept only JavaScript objects that live on the JavaScript heap. The Wasm GC proposal makes it possible to have a joint heap

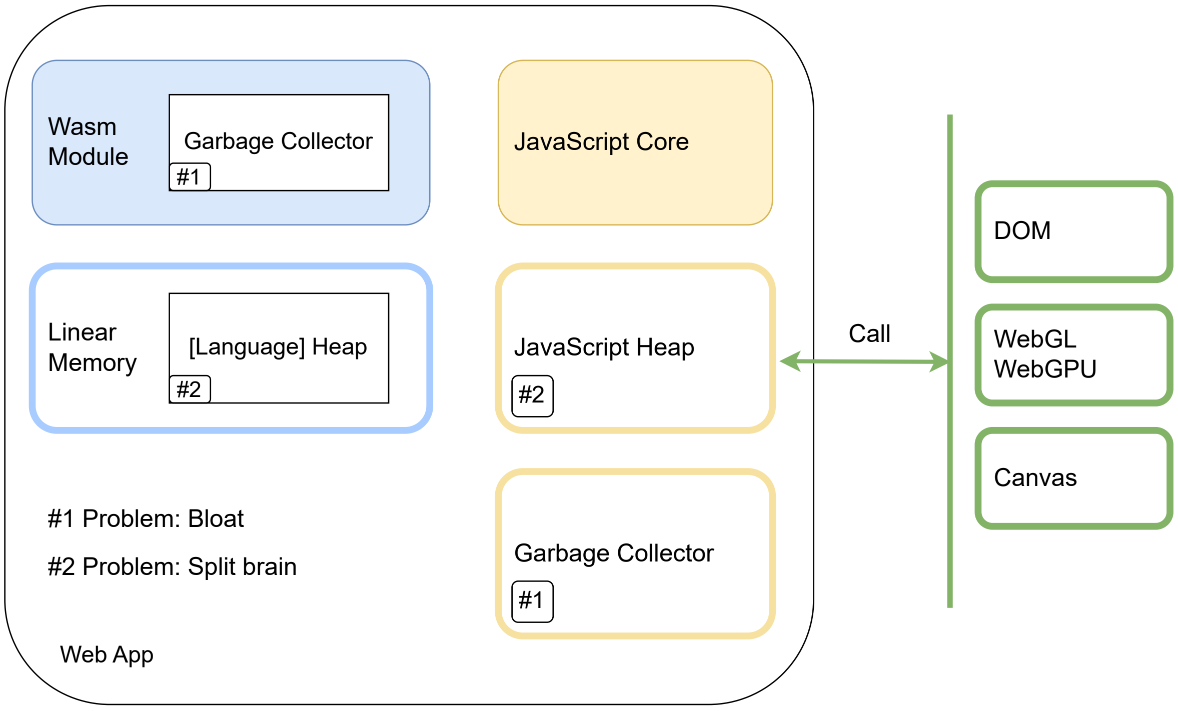

Figure 2.2: Wasm module without garbage collection proposal based on [18]

between JavaScript and WebAssembly GC code. This way the managed-memory code can allocate objects on the joint heap and when the JavaScript garbage collector runs, it will also collect the WebAssembly GC memory, because they are using the same heap memory (see Figure 2.3). Furthermore, depending on the runtime implementation, the runtime can return the unused memory back to the operating system to ensure the efficiency and responsiveness of the application [18].

This means that the Wasm module doesn't have to include it's own garbage collector, which will reduce the Wasm module size and startup time. Moreover, the Wasm module can easily grow or shrink the memory based on consumption.

To summarize, according to Vivek Sekhar in Wasm I/O 2023 conference, the garbage collection proposal brings the following three benefits [18]:

- **Smaller binaries for managed-memory languages**: Languages with dedicated garbage collectors no longer need to ship the garbage collector with the runtime. The hosts garbage collector can be used to garbage collect JavaScript and WebAssembly memory.
- **Enhanced interoperability with JavaScript code and Web APIs**: Wasm mod-

ules can directly access Web APIs, such as DOM, WebGL, Canvas, and more. Furthermore, objects can be passed to the JavaScript code, eliminating the need to copy objects between JavaScript and WebAssembly memories.

- **Resizable memory footprint**: The memory footprint can adjust to the usage and therefore additional memory can be allocated to WasmGC module. Unused memory can be returned to the host environment.

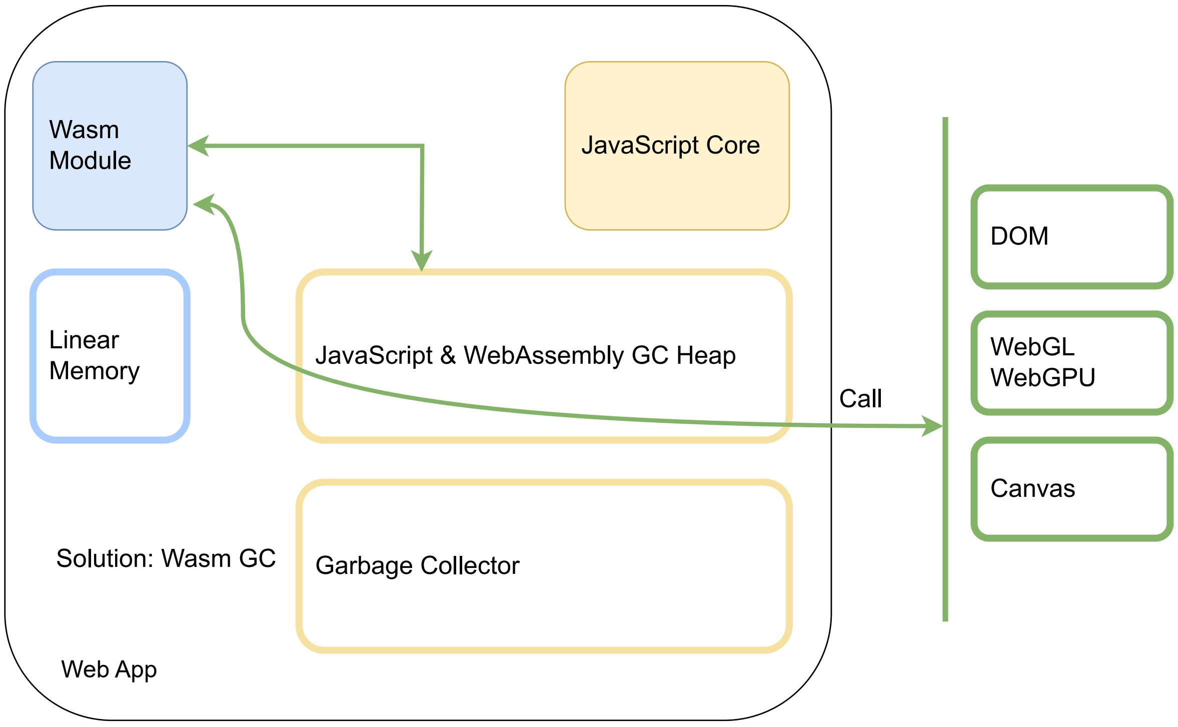

Figure 2.3: Wasm module with garbage collection proposal based on [18]

Even though the garbage collection proposal is still under active development, the Kotlin team has already rolled out an experimental compiler based on the proposal [19].

2.7 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 (abbreviated 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 [20], which includes low-level interfaces like POSIX functions and high-level interfaces like 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 [21], a Mozilla developer working on WASI, Wasmtime and WebAssembly projects, 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. Preview2 will incorporate the lessons learned from Preview1 and add new features such as "sockets", "timezones" and "file locking". After Preview2, the community will begin working on Preview3, which will be the final preview before the official release of WASI 1.0.

2.7.1 WASI Software Architecture

The WASI software architecture is similar to the native counterpart, as shown in Figure 2.4, at the top of the stack, we have the User application, this can be serverless function, a microservice, a web application, or any other application.

Then we have the libc implementation, which is responsible for providing the standard C library functions to the user application.

"The musl-based libc interface, implemented on top of a libpreopen-like layer and a system call wrapper layer as noted in [22, para. 1]."

The system call wrappers are responsible for calling the actual WASI implementation (see Figure 2.4, highlighted in green). Each WASI implementation is specific to the target environment, for example, the WASI implementation for Linux is different from the WASI implementation for Windows.

2.7.2 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, including high-performance games requiring graphics rendering, sound playbacks, 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 [23].

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 JavaScript glue code would subsequently communicate with the browser, which would in turn communicate with the Kernel and the OS.

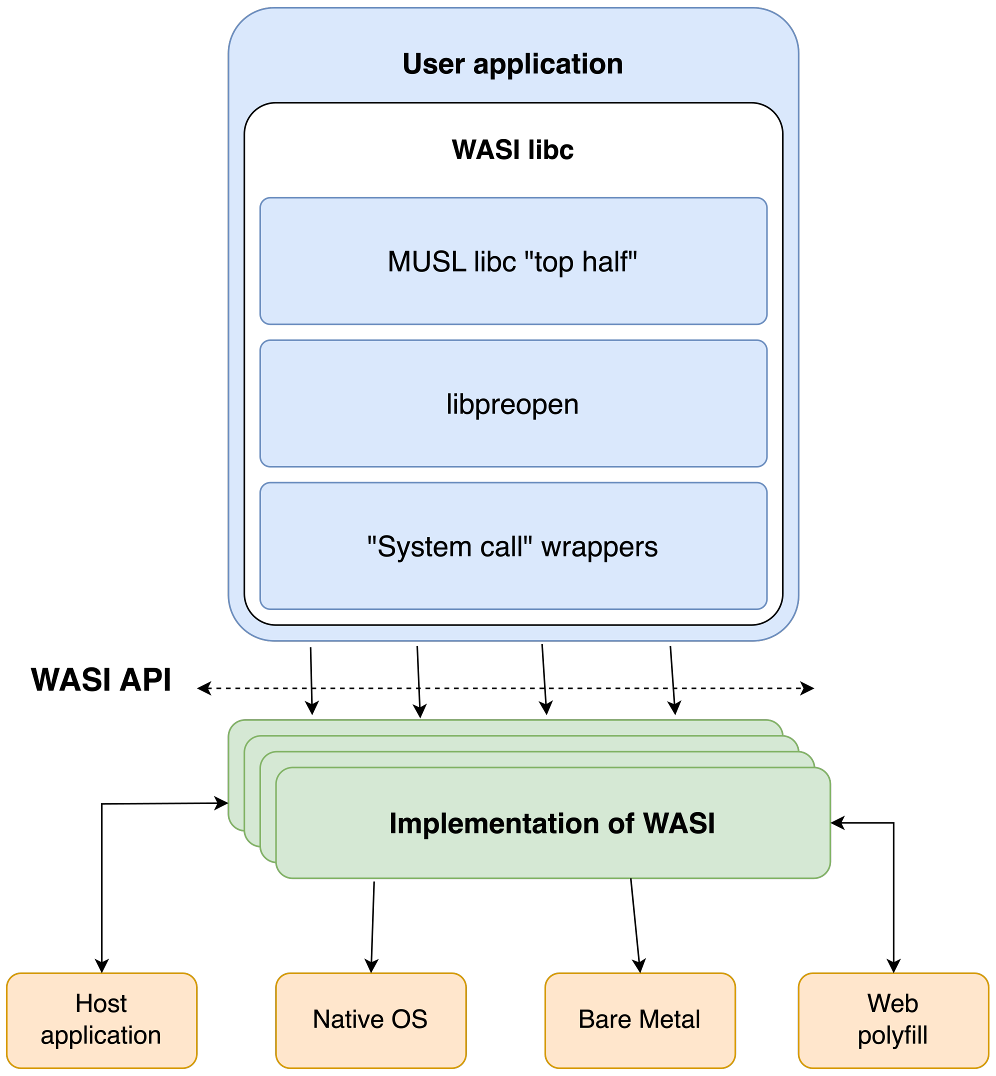

Figure 2.4: WASI software architecture redrawn from [22]

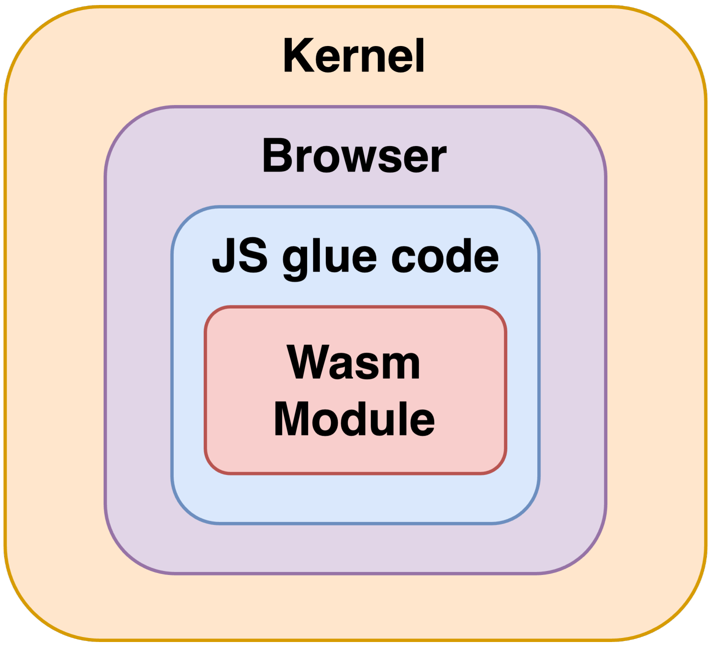

Figure 2.5: Structure of Emscripten compiled Wasm module, inspired by [24]

Emscripten did a great job for compiling C/C++ code into browser compatible Wasm modules and JS glue code. However, 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 would have 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 2.5 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 [25].

2.7.3 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 2.6 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.

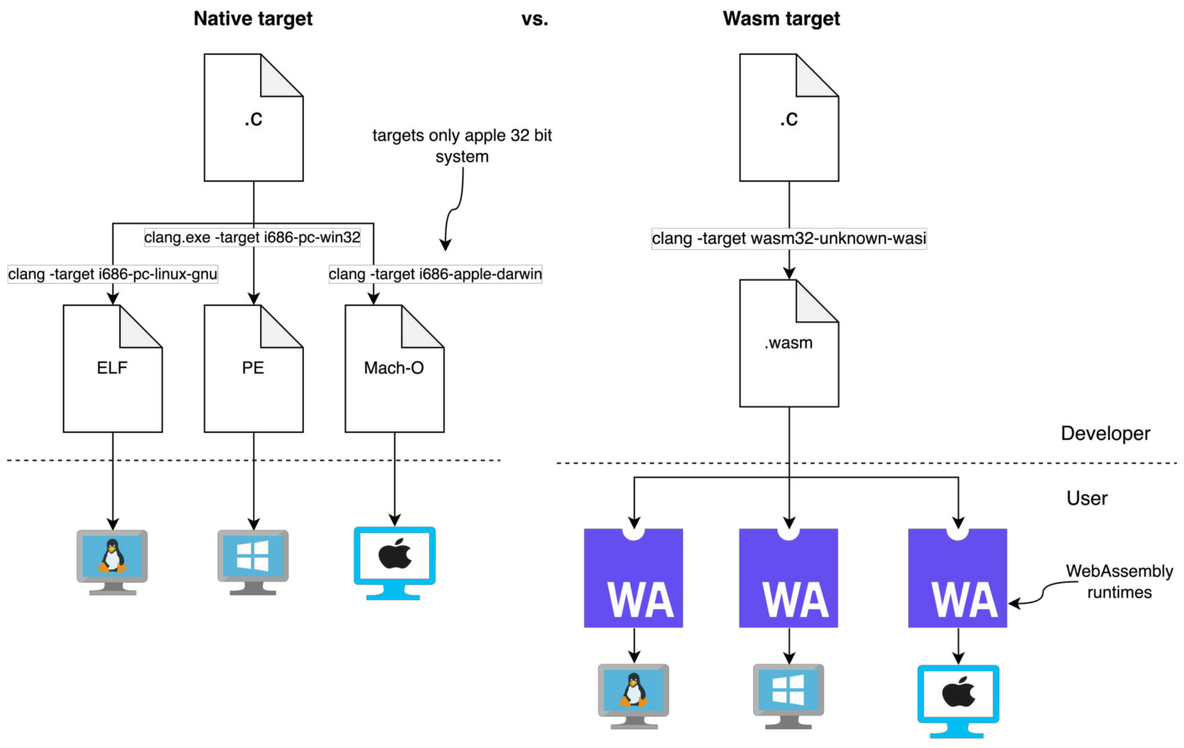
Native target
vs.
Wasm target
.C
targets only apple 32 bit system
clang.exe -target i686-pc-win32
clang -target i686-pc-linux-gnu
clang -target i686-apple-darwin
ELF
PE
Mach-O
.C
clang -target wasm32-unknown-wasi
.wasm
Developer
User
WA
WA
WA
WebAssembly runtimes

Figure 2.6: WASI portability model redrawn from [24]

Aside from the information provided in Figure 2.6, 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 [26], as long as the host runtime supports the required APIs.

2.7.4 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 [9], particularly when new maintainer of open-source libraries take over, as their unrestricted access could enable them to write code that compromises system security by accessing files or sending them over the network [24].

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 use virtualization, all modules share common resources, such as the file system. As a result, the cloud provider may want to setup strict permission rules and only 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

An implementation of the WebAssembly specification [6] is referred to as a WebAssembly runtime or a WebAssembly engine. The initial WebAssembly runtimes were implemented into major browser virtual machines, such as Mozilla's SpiderMonkey and Google's V8 engine. However, as WebAssembly became more mature, it was clear that the characteristics of WebAssembly are also beneficial for other use cases outside of the browser. Therefore, the need for standalone WebAssembly runtimes each with their own characteristics and features emerged.

This chapter provides an overview of the most popular WebAssembly runtimes, focusing on their distinct characteristics. Additionally, we will delve into the different execution models and explore how they influence the performance of these runtimes. Moreover, we will try to identify the requirements for a serverless runtime and compare this approach to something similar like V8 isolates.

3.1 Runtimes Overview

Table 3.1 presents an overview of the most standalone popular WebAssembly runtimes in each category. The selected runtimes are based on their popularity, support for the WASI standard, and their respective compilation models (JIT, AOT, Interpreter). This selection aims to provide a good representation of the current state of WebAssembly runtimes.

Runtime	Embeddable langs.	Architecture	Compiler	Compilation	Platform
Wasmtime	Rust, Python	x86_64, ARM, System/390	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 [27]

3.2 Wasmtime

Wasmtime [28] is the official runtime of the Bytecode Alliance. This runtime, written in the Rust programming language, provides a Rust-based API for embedding within a Rust application. Additionally, it supports WASI out of the box. Wasmtime's default compilation model is JIT, but it also supports AOT compilation. The runtime supports both cranelift and LLVM backends for JIT and AOT compilation (see Table 3.1).

As mentioned in Chapter 1, the Bytecode Alliance is a consortium of individuals from Mozilla, Fastly, Red Hat and Intel, who are working on the WebAssembly standard and the WebAssembly System Interface (WASI).

3.3 Wasmer

Wasmer [29] is an independent WebAssembly runtime developed in Rust. It offers support for both WASI and Emscripten, adhering to the most recent WebAssembly proposals. Unlike Wasmtime, Wasmer isn't solely limited to Rust, but can be embedded into a wide variety of programming languages, including C/C++, Go, Java, Ruby, Python, among others. Furthermore, Wasmer incorporates three different compiler backends: "Singlepass", "Cranelift", and "LLVM". "Singlepass" delivers quicker compilation time, but at the expense of slower runtime performance due to non-optimized code. Conversely, "LLVM" may take longer to compile, but its runtime performance is faster. "Cranelift" presents a balance between the two extremes.

3.4 Wasm3

Wasm3 [30] is a fast and universal WebAssembly interpreter written in C language. Wasm3 can be used as a library in many programming languages. This runtime is designed to be embedded into microcontrollers, IoT devices, and edge devices, thus it has a small footprint and low memory usage. The supported platforms include x86, x86_64, ARM, RISC, Xtensa and MIPS.

The benchmark included in the repository [30] show that Wasm3 is the fastest Wasm runtime with a interpreter execution model. However, the benchmark also shows that Wasm3 is about four times slower than JIT-based runtimes like Wasmtime and Wasmer.

However, the advertised speed of Wasm3 relative to the memory usage and startup time makes it a very interesting runtime for further evaluations.

3.5 WasmEdge

WasmEdge [31] is a WebAssembly runtime specifically designed for cloud-native, decentralized, and edge computing applications. WasmEdge is one of the Cloud Native Computing Foundation (CNCF) [32] projects, the same entity responsible for the Kubernetes project. Recently, WasmEdge has been introduced as a faster and more lightweight alternative to Linux and Windows Containers in the beta version of Docker. The runtime is written mainly in C++ and Rust, it also supports both WASI and Emscripten.

3.6 V8 Engine and V8 Isolates

V8 engine is Google's open-source JavaScript and WebAssembly runtime (engine), that is used in Google Chrome browser and Node.js.

On the other hand, a V8 isolate is a separate instance of the V8 engine that has its own memory, garbage collector, and global object [33]. An isolate can run scripts in a safe and isolated environment, without interfering with other isolates [34]. An isolate also has its own state and context, which means that objects from one isolate cannot be used in another isolate [33].

The following Figure 3.1 shows how a single engine can have many isolates. Contrary to traditional approaches where a process is started along with its own container or microVMs, a V8 engine is started on top of a container and then isolates can be created on demand. This approach is much faster than starting a new container or microVM for each request. The overhead happens only once when the load is high enough to require a new container with a new V8 engine process [34].

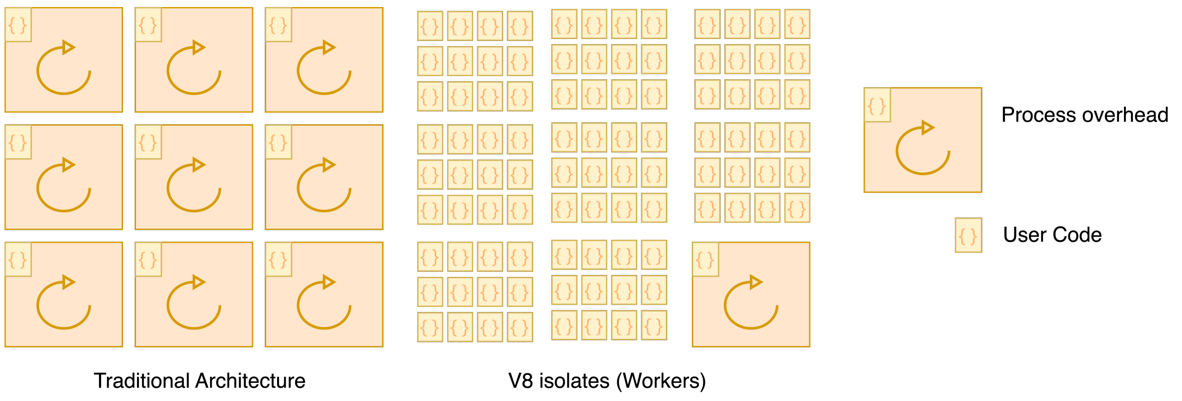

Figure 3.1: containerized vs. V8 isolates figure redrawn from [34]

3.6.1 Comparing V8 Isolates to WebAssembly Runtimes

Both V8 isolates and WebAssembly are similar in the way that they both execute code in a sandboxed environment. However, a V8 isolate is a JavaScript powered worker, while WebAssembly on the other hand is a binary instruction format. On the bright side, a V8 engine can include a WebAssembly runtime, which means that a V8 based cloud providers will be able to support multiple languages through WebAssembly.

When it comes to performance, according to Lin Clark's article on the release of Wasmtime [26], "a JS isolate startup time is about 5 milliseconds while a WebAssembly runtime startup time is about 50 microseconds." She then writes that WebAssembly is more lightweight and therefore, it can have many more instances running at the same time compared to other coarse-grained isolates.

We will compare the performance of V8 isolates to the performance of WebAssembly runtimes in chapter 5 and measure the startup time of both approaches.

3.7 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. There are many requirements for a serverless platform; however, we believe that the following requirements are important for a serverless runtime. 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.

3.8 Execution Models and Performance Implications

One of the most important aspects of WebAssembly is its versatility. WebAssembly can be used in a wide variety of use cases, from running in the browser to running on microcontrollers. However, the different use cases require different execution models. For example, in a serverless function, the WebAssembly module could be compiled and executed ahead of time. On the other hand, in a microcontroller, the WebAssembly module is preferably executed by a WebAssembly interpreter.

Moreover, Wasm supports many programming languages and runs on various instruction set architectures, because the specification is designed to be fast, secure and portable [35].

This section is dedicated to exploring the different execution models and how they influence the performance of the runtimes, so that we can evaluate and discuss the benchmark results in chapter 5.

3.8.1 Choice of Programming Languages

The choice of a programming language does affect the performance and the size of the WebAssembly module. System programming languages like C/C++ and Rust are lightweight and require little runtime overhead. On the other hand, languages like Java and .NET bring a lot of runtime overhead, which increases the size of the WebAssembly module [35].

Compliler Code Optimization

Some compilers, like the Rust compiler, can utilize compiler flags to prioritize the quality of the output code over the compilation time. In case of Rust, the `opt-level` flag can be used to set the optimization level. The default value is `opt-level=0`, which means that the compiler will prioritize the compilation time over the quality of the output code. The highest value is `opt-level=3`, which means that the compiler will prioritize the quality of the output code over the compilation time. As a result, the output code will have a better execution performance and a smaller size [36].

3.8.2 JIT - Just In Time Compilation

A Just In Time Compilation (JIT) is a technique in which WebAssembly module is compiled into native machine code when it's about to be executed. The code is optimized for the running hardware. The JIT compilation comes at the cost of a longer startup time compared to an Interpreter or Ahead of Time (AOT) compilation. This penalty is specially noticeable in smaller functions, where the execution time is short [35].

3.8.3 Interpreter

An interpreter runtime, such as Wasm3, processes and executes small chunks of code in sequence [30]. This execution model incurs minimal, if any, startup time penalty as it only uses resources when required. Furthermore, because it minimizes the workload, it has a small memory footprint, making it ideally suited for resource-constrained devices like microcontrollers. Notwithstanding, for long-running functions, its execution speed is generally slower compared to JIT-based runtimes [35].

3.8.4 AOT - Ahead Of Time Compilation

An Ahead of Time (AOT) compilation is a technique in which the WebAssembly module is compiled into native machine code before it's executed. The code is optimized for the running hardware, thus, it can only run on the specific machine they were compiled for. The AOT compilation comes at the cost of a longer build time compared to an Interpreter or JIT compilation. However, the AOT compilation has the advantage of a fast startup time and a fast execution time. Moreover, each runtime possesses its unique format for these optimizations, meaning for example a Wasmtime AOT-compiled program will not be compatible with other Wasm runtimes like Wasmer [35].

The Wasmtime runtime has the capability to compile a Wasm module into an AOT-compiled binary. For example, given we have a Wasm module called `myfunction.wasm`, we can compile it into an AOT-compiled binary `myfunction.cwasm` by running the following command [28]: `wasmtime compile myfunction.wasm`

The performance benefits of AOT compilation are likely noticeable on long-running complex functions outperforming JIT compilation.

Nonetheless, the compiled binaries are larger than the original Wasm modules, due to the fact that many of the optimizations are incorporated into the binary. This is a trade-off between the size of the binary and the performance of the function. In regards to serverless functions, the AOT compilation model is the most suitable, because it has a fast startup time and a fast execution time. Moreover, the drawbacks namely the larger binary size can be neglected, because the serverless functions are usually small in size and the portability of the Wasm modules is not a concern, because the functions can be compiled for the specific machine on deployment phase.

3.8.5 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 3.2, 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.

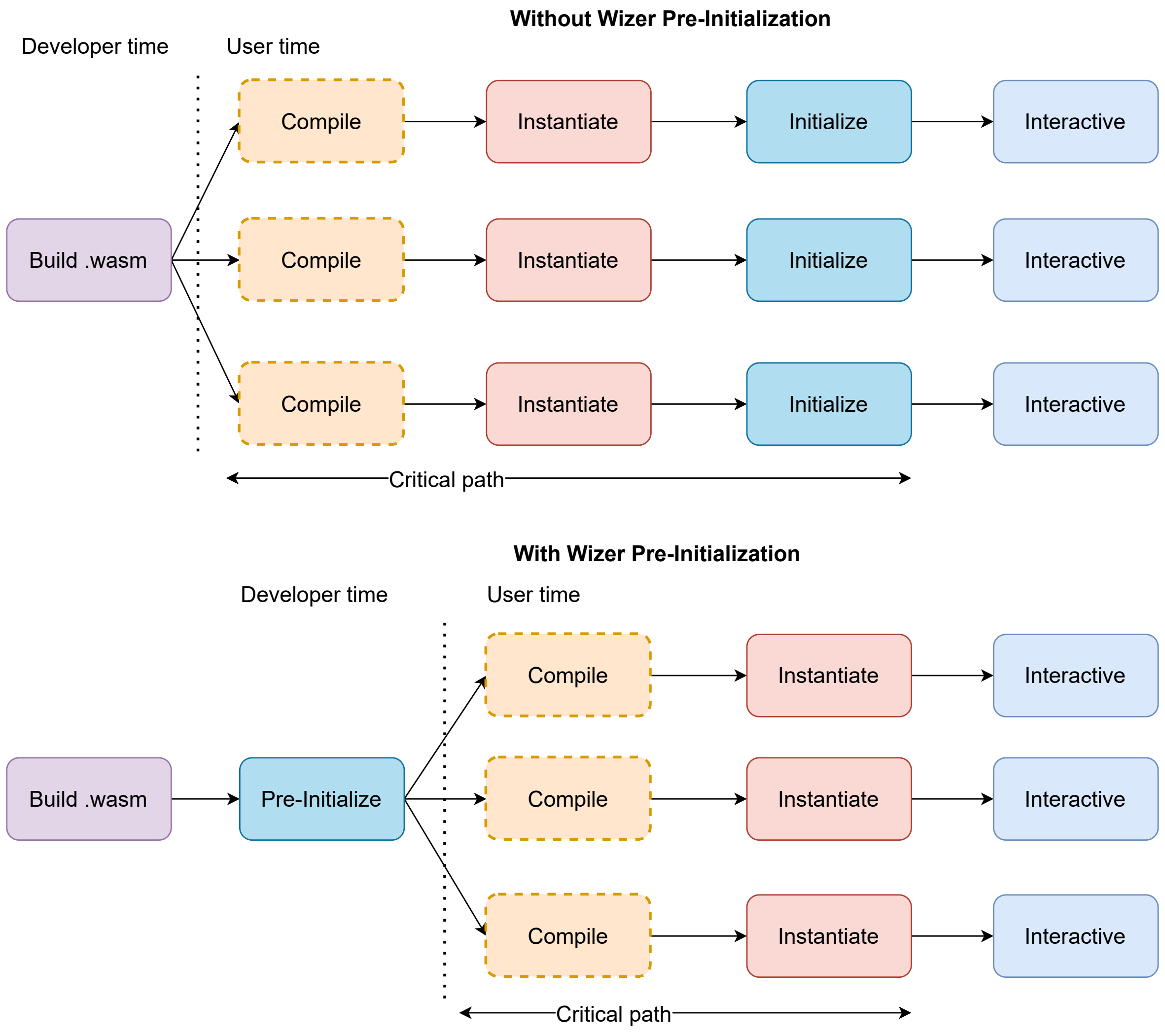

Figure 3.2: Overview of Pre-initialization vs. Non Pre-initialization, based on [37]

Wizer creates a pre-initialized snapshot through these four phases [37]:

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 [38], we can confirm that Wizer can provide a significant speedup in the instantiation and initialization phases. For Figure 3.3, 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.

The 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.

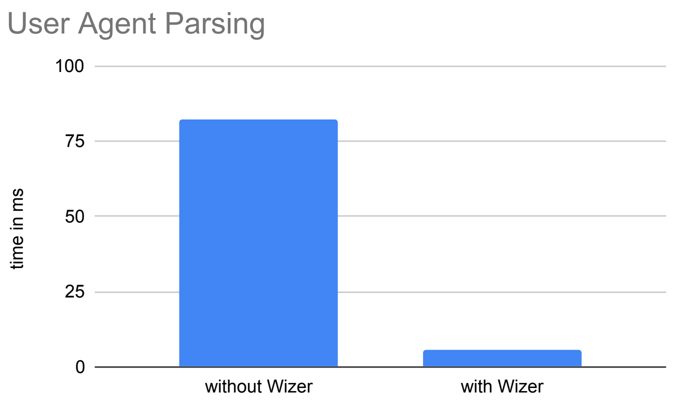

Figure 3.3: User Agent Parsing benchmark

4 WebAssembly Serverless Platform Design

In this chapter, the proof of concept for a simple serverless compute platform using Wasm technology will be explored. For simplicity, the platform runs on an HTTP server to redirect HTTP requests to specific WebAssembly modules. For this, we will use Wasmtime [28] as the WebAssembly runtime, but we could also use Wasmer [29]. Since Wasmtime does only have Rust integration, we will use the Rust language along with the popular Rust web framework, "Actix-web [39]", due to its lightweight and fast nature. As mentioned in the previous chapter, Wasmtime offers excellent support for the latest WASI standards and features.

As Figure 4.1 illustrates, this proof of concept assumes that there will be 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. Nonetheless, this proof of concept primarily focuses on the WebAssembly aspect of the platform, leaving the deployment platform, API gateway and load balancers outside the scope of the discussion. The platform has a single endpoint

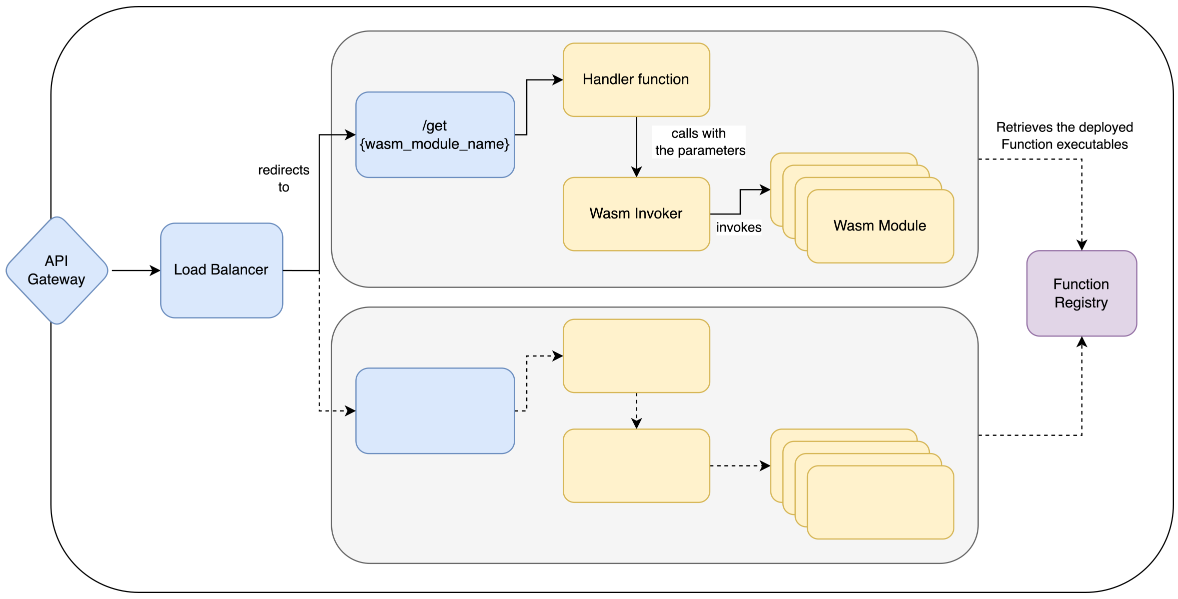

Figure 4.1: Execution flow of the simple POC serverless platform, inspired by [1, p. 142]

that accepts HTTP requests with a Wasm module name in the path and query parameters. The endpoint will then load the Wasm module from the same directory and call the defined `invoke_wasm_module` in Listing 4.1. This function is the main part of our platform, it is responsible for creating a Wasmtime runtime engine and linker, adding the WASI APIs to the linker. The function then creates a WASI context with the query parameters and loads the module with the file name. The linker is used to link the Wasm module with the later created instance together. There is also a buffer that will be used to store the response from the Wasm module.

```rust
fn invoke_wasm_module(wasm_module_name: String, params: HashMap<
    String, String>) -> Result<String> {
    let engine = Engine::default();
    let mut linker = Linker::new(&engine);
    wasmtime_wasi::add_to_linker(&mut linker, |s| s)?;

    let stdout_vec_buf: Vec<u8> = vec![]; // a buffer that will
    contain the response
    let stdout_vec_mutex = Arc::new(RwLock::new(stdout_vec_buf));
    let stdout = WritePipe::from_shared(stdout_vec_mutex.clone());

    // params to 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();

    // get the response
    let mut buffer: Vec<u8> = Vec::new();
    stdout_vec_mutex
        .read()
        .unwrap()
        .iter()
        .for_each(|i| buffer.push(*i));

    let s = String::from_utf8(buffer)?;
    Ok(s)
}
```

Listing 4.1: invocation function

Directly after, we then call the run_wasm_module function shown in Listing 4.2. This is where the linker links the module with the newly created instance. The instance is used to search for the start function, the "_start" function is the starting point of the Wasm modules. It will call the function and return the result. The result is then stored in the buffer and returned to the caller.

```rust
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 4.2: calling the wasm start function

To test the platform, we can use any WASI complaint Wasm binary. The following AssemblyScript file contains a simple statement that outputs "Hello World!" on the terminal. The AssemblyScript compiler will compile the file to a WASI complaint Wasm binary. For this we will also need the "wasi-shim" which is a standalone dependency.

```
// calling AssemblyScript's console.log,
// which calls the corresponding WASI interface
console.log('Hello World!');
```

Listing 4.3: AssemblyScript file with a simple console output

4.1 Wind-up

This example shows how easy it is to create a simple serverless platform that can invoke any WASI complaint Wasm module. The above Wasm module example was written in AssemblyScript, but in fact, we can use any language that compiles to Wasm and its WASI complaint. If we build this project with the highest compiler optimization, then the resulting binary is only 15 MB in size. This is a modest size for a serverless platform, with less than 80 lines of code. If we run the server and call the mentioned "hello-world" example, we get the following result:

```
curl -w %{time_total}\\n  http://localhost:8000/hello-world
Hello World!
0.009886
```

The result contains the response which, in this case is "Hello World!" and the measured time until the response was received. The measured time is around 10 milliseconds and a deviation of ±15%. The response time is quite fast, but we should consider the following points:

- The server is running on a local machine, therefore it does not take dns lookup, tcp handshake, load balancing overhead and other network related overheads into account
- The system has more resources (M1 Pro CPU and 32 GB RAM) than a typical edge computing device
- The server lacks various features typically associated with a serverless platform, therefore it has less overhead than a typical serverless platform
- The Wasm module is very simple and does not contain any heavy computation
- The large portion of the response time is caused by the Actix Web framework

To concrete the last point, we can run the same test without the Wasmtime runtime and the Wasm module. This time we will use the Actix Web framework to return the "Hello World!" string directly and we use the same optimization level as before. The result is shown in the following listing:

```
curl -w %{time_total}\\n  http://localhost:8000/hello-world
Hello world!
0.005646
```

This time the response time is around 6 milliseconds with a deviation of ±10%. This shows that the Actix Web framework is responsible for 60% of the response time compared to the previous test setup. This is not surprising, as the Actix Web framework is a full-fledged web framework with many features.

This is as an excellent starting point for running more precise measurements on the actual runtime in the next chapter, with the aim of gathering a more better picture of the performance of various factors.

5 Evaluation

In this chapter, we will conduct our evaluation in two parts. In the first part, we will evaluate the performance of different WebAssembly runtimes. In the second part, we will evaluate the performance of the WebAssembly-deployed functions compared to other non-WebAssembly functions.

5.1 Goals

Referring back to our research objectives from section 1.1, the goals of this evaluation are to answer the following questions derived from the research question:

1. What is the startup time of a simple Wasm module?
2. How do different programming languages compare?
3. How do various runtimes compare to each other?
4. How do the WebAssembly runtimes compare to native code?
5. How does a WebAssembly-powered cloud platform measure up against a V8-powered cloud platform and a cloud platform with microVMs?

5.2 Methodology

To answer the above questions, we need to perform a series of different benchmarks. Each benchmark is targeted at answering a specific question. In chapter 3, we described the different runtimes with the corresponding compilation models, that we will be using in this evaluation. The next subsections will elaborate on the distinction between microbenchmarks and macrobenchmarks, as well as their application in answering the questions in section 5.1.

- **Microbenchmarks**: Microbenchmarking is a technique used to measure the performance of a small function or a unit of code. It has the advantage of precisely measuring the performance of a specific function or code snippet. However, it can be difficult to extrapolate the results to a real-world scenario. Results from a specific scenario can look very different from the overall performance of a system. The evaluation does contain some microbenchmarks, but the main focus is on macrobenchmarks.
- **Macrobenchmarks**: Macrobenchmarking is not a well-defined term. It can be defined as a type of benchmark that measures the performance of a whole critical path. It is a more realistic approach to compare different systems with similar setups. The drawback of macrobenchmarks is that the results include the performance of the entire system, which is influenced by a lot of variables. To interpret the results, we will first try to subtract the noise and the system's overhead, and then we will compare the results with those of the microbenchmarks.

5.3 Setup

The evaluation setup consists of an AWS EC2 t2.medium instance, deployed on eu-central1 which is located in Frankfurt Germany. The instance is running Ubuntu 22.04 LTS with 2 vCPU and 4 GB of RAM. This instance is used to send requests and conduct benchmarks on different cloud platforms. The advantage of using a ec2 instance is the location. The latency between the ec2 instance and the cloud platforms is very low as shown in Figure 5.1. The ICMP packets have an average round-trip time of 1ms for Fastly, 1.35ms for Cloudflare, and 0.3ms for AWS, as the functions are served by a node in the same availability zone.

Additionally, we ran the benchmarks on this t2.medium instance to measure the runtime performance because it provides a comparable environment with the minimal required resources, to perform the runtime benchmarks. Figure 5.1 shows the deployment setup:

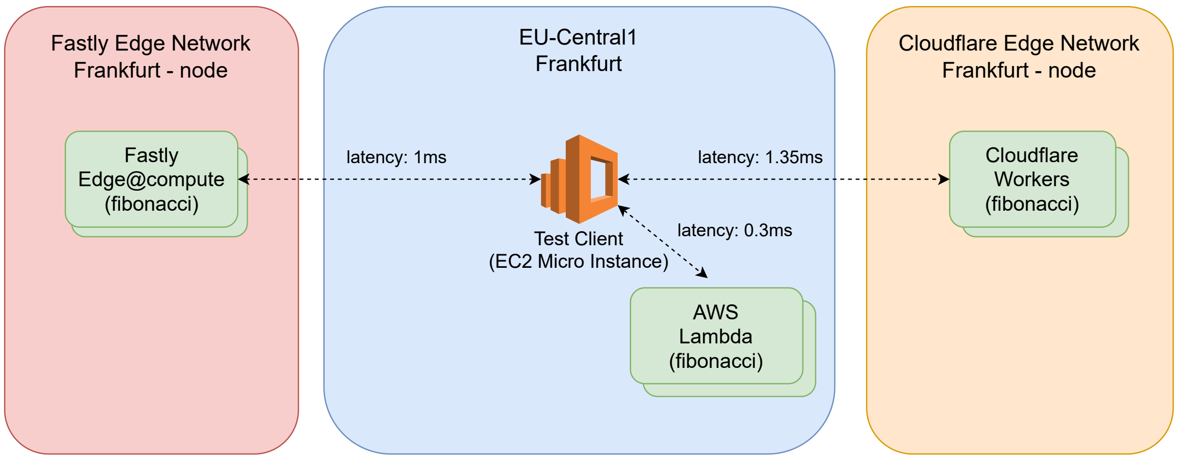

Figure 5.1: Deployment setup of the serverless functions

Moreover, we used a MacBook Pro with M1 Pro CPU and 32GB of RAM to run the runtime benchmarks locally. The benchmarks were executed while the MacBook was plugged into the power supply, and no other major processes were running during the benchmarking process. The M1 Pro CPU is an ARM-based CPU, which is different from the t2.medium's Intel x86 CPU. The results of the two setups cannot be compared directly, but we can use them as a high-level comparison to inspect factors like CPU architecture compatibility and identify any significant outliers. We have tried to perform the benchmarks on a Raspberry PI 2, but the ARM 32-bit architecture is not supported by runtimes like Wasmtime and Wasmer. Aside from that, we also tried a more limited setup with t2.micro instance, but the benchmark code did not compile in a reasonable time. The t2.medium instance is the smallest instance that can run the benchmarks in a reasonable time.

5.4 Criterion

Criterion.rs [40] is a statistics-driven benchmarking library for Rust. In our evaluation, Criterion is used to measure the performance of different runtimes. Criterion is a powerful tool that can be employed to measure the performance of a function or a code snippet. Criterion.rs is inspired by Haskell's criterion library. The library provides HTML reports with statistical analysis of the benchmark results, including the mean, median, standard deviation, as well as the regression with previous runs. Moreover, we utilize the library's utility functions such as `black_box` to prevent the compiler from optimizing the code away.

Criterion process consists of four phases:

- **Warmup**: In the warmup time the routine is repeatably executed for a given time. This gives the CPU, OS caches and also the JIT compiler if present, time to adapt and optimize the code.
- **Measurement**: Similar to the warmup time the code is repeatably executed in this phase, but unlike the warmup time, the measurement results will be used for analysis. The measurements consists of many samples. Each sample has one or multiple iteration of routines.
- **Analysis**: This is the phase where the statistical analysis is performed. Values outside of the 25th and 75th percentile are classified as outliers. Values outside of the 5th and 95th percentile are classified as severe outliers. The mean, median, standard deviation, and the regression metrics with previous runs are calculated in this phase.
- **Comparison**: In this phase the statistics of the current run are compared to the previous runs. The comparison is performed using a regression analysis. The results of the comparison are shown in the HTML report and in console output.

5.5 Benchmarks

In this section, we will perform the benchmarks that address the questions outlined in section 5.1. The benchmarks are split into two categories: *runtime* and *serverless platform*. Under the *runtime* category, we will measure the "cold start" time and conduct performance tests with various workloads. In the *serverless platform* category, we will benchmark the response time and compare the performance of different platforms.

5.5.1 Cold Start - Instantiation

One of the challenges of this thesis has been to precisely define what "cold start" means and how to measure it. In the context of WebAssembly, we define "cold start" as the time it takes from creating a runtime context to loading the module and executing the module's "start" function. This definition is specifically applicable to WebAssembly and V8 setups. The reason for this specificity is the difference between the WebAssembly runtime and a virtualized solution like Firecracker's microVM, which powers the Lambda and AWS Fargate instances. In a virtualized solution, the "cold start" time is the time it takes to boot the virtual machine and load the application. In the case of WebAssembly, the runtime is already running, and the module is loaded into the runtime.

As mentioned before, we utilized the criterion library to measure the instantiation process. Initially, we created a *WAT* file (as shown in Listing 5.1) that included a small memory and a "start" function. The benchmark was executed both locally and on a t2.medium instance. The results are illustrated in Figure 5.2.

```
(module
    (memory 1)
    (func (export "_start"))
)
```

Listing 5.1: WASI module with small memory

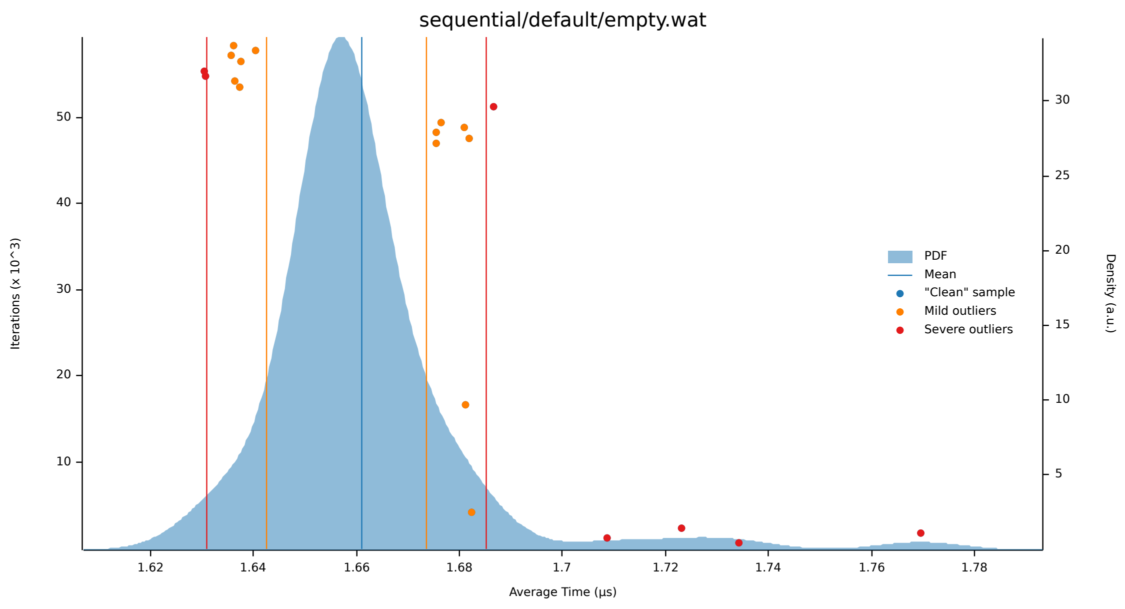

Figure 5.2: Instantiation distribution of an empty Wasm module with Wasmtime and M1 Pro CPU

Furthermore, we also conducted an instantiation benchmark, where we first compiled a simple Rust "Hello World" program to a WASI module and then instantiated it using Wasmtime. The results of this benchmark are shown in Figure 5.3. Please note that we did not measure the compilation time.

Additionally, we performed another benchmark where we defined a *WAT* file with a large data segment and memory. The results for this benchmark are included in the appendix section.

5.5.2 Execution Performance

In this section, we will assess the execution performance of different runtimes. Once again, we will use the criterion library for analysis. For this evaluation, we have created both an executable binary and an executable Wasm module. Both the native binary and the Wasm module contain the benchmark function. To ensure realistic production performance, we compiled with the `-release` flag. Moreover, the Wasm module was automatically optimized using `wasm-opt`. During benchmarking, the criterion's `black_box` utility is used to prevent the compiler from optimizing the code away.

For this evaluation, we have chosen the following runtimes: Wasmtime and Wasmer as the JIT runtimes, Wasm3 as the interpreter runtime, and WasmEdge as the AOT runtime. The benchmarks were conducted using the following functions:

Fibonacci

The Fibonacci function is a well-known benchmark used to assess the performance of a runtime. It is chosen for two main reasons: Firstly, it involves a recursive computation,

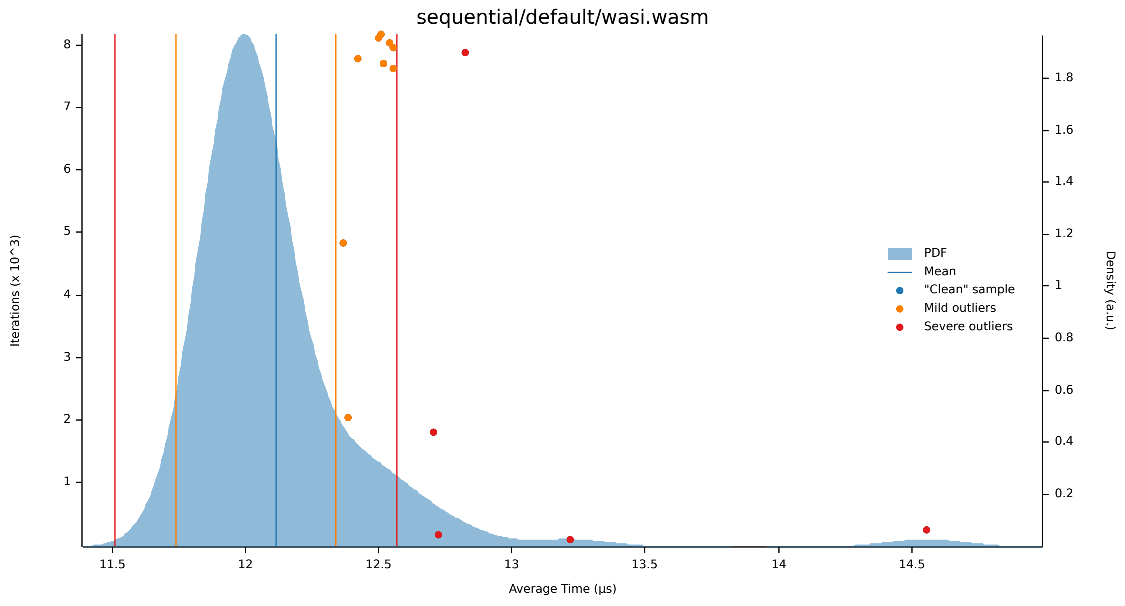

Figure 5.3: Instantiation distribution of a WASI module with Wasmtime and M1 Pro CPU

Fibonnaci 40, t2.medium		
Runtime	**Baseline**	**Time**
Native	1.00	396.1±0.42ms
Wasmtime	1.35	534.74±2.24ms
Wasmer	1.38	546.61±2.61ms
NodeJS	1.42	562.46±2.41ms
WasmEdge	2.13	843.69±4.21ms
Wasm3	4.82	1909.2±8.52ms

Fibonnaci 32, t2.medium		
Runtime	**Baseline**	**Time**
Native	1.00	9.11±0.02ms
Wasmtime	1.31	11.92±1.56ms
Wasmer	1.33	12.1±1.74ms
NodeJS	1.45	13.2±2.56ms
WasmEdge	2.21	20.11±3.23ms
Wasm3	4.51	41.04±2.51ms

Table 5.1: Runtime Benchmark: Fibonacci 40 and 32 on t2.medium; Wasmtime, Wasmer, NodeJS (JIT), WasmEdge (AOT), Wasm3 (Interpreter)

making it a CPU-intensive task. Secondly, it is a simple function that can be easily understood and implemented in any environment.

Blake3

Blake3 is an efficient and fast cryptographic hash function. With the increasing use of cryptographic functions in modern applications, it becomes important to measure the performance of such functions. As a deterministic one-way function, Blake3 hashes a "Hello World" string in the benchmark. Unlike the Fibonacci function, Blake3 is expected to execute much faster, leading us to assume that an interpreter runtime or an AOT runtime may have an advantage over a JIT runtime.

The results of this evaluations are shown in Figure 5.4, Table 5.1 and Table 5.2. We will discuss the results in section 5.6.

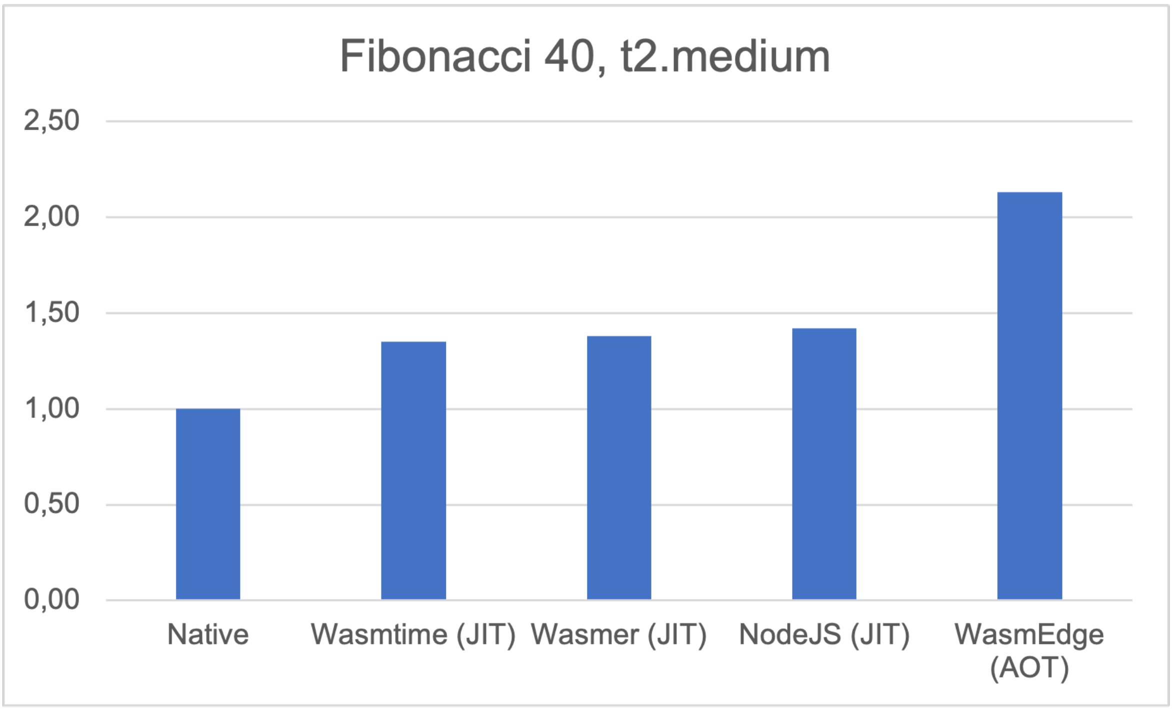

Figure 5.4: Runtime comparison on t2.medium: Fibonacci 40 results, excluding Wasm3

Blake3, M1 PRO CPU		
Runtime	**Baseline**	**Time**
Native	1.00	83.8±1.86ns
NodeJS	1.10	92.3±2.24ns
Wasmtime	1.13	94.3±2.61ns
Wasmer	1.35	113.13±2.41ns
WasmEdge	2.05	171.5±4.21ns
Wasm3	4.82	403.91±8.52ns

Blake3, t2.medium		
Runtime	**Baseline**	**Time**
Native	1.00	114.7±0.19ns
NodeJS	1.32	150.084±1.56ns
Wasmtime	1.41	161.727±1.74ns
Wasmer	1.46	167.462±2.56ns
WasmEdge	2.15	246.605±3.23ns
Wasm3	4.97	570.059±2.51ns

Table 5.2: Runtime Benchmark: Blake3 on M1 PRO and t2.medium; Wasmtime, Wasmer, NodeJS (JIT), WasmEdge (AOT), Wasm3 (Interpreter)

5.5.3 Evaluating Serverless Platforms

Evaluating various serverless platforms is a challenging task, because there are a handful of variable factors that can influence the measurements.

We deployed the same functions on different platforms and with different programming languages. The functions were invoked by a timestamp in order to prevent caching of the response by the CDN. The functions were invoked sequentially with a delay of 15 minutes between each invocation. The delay was chosen to prevent warm starts, we conducted a few tests to find the idle time before the functions are terminated. A bash script was used to invoke the functions with curl and write the response metrics into a file. As mentioned in the section 5.3 the test instance has a latency of 0.3 to 1.3 milliseconds to the functions. Here is a list of the deployed functions:

- **Fibonacci**: The Fibonacci function calculates the Fibonacci for a given number n. The function is implemented in Rust, JavaScript and Go. We deployed the function on Fastly Compute@Edge, Cloudflare Workers (only JavaScript) and AWS Lambda (only JavaScript).
- **Blake3**: The Blake3 function calculates the hash of a given string. We used the string "Hello World". The function is implemented in Rust, JavaScript and Go. We deployed the function on Fastly Compute@Edge, Cloudflare Workers (only JavaScript) and AWS Lambda (only JavaScript).

Curl Performance Metrics

The curl command is a powerful built-in tool in the bash shell. We use it to send the requests from the test client to each serverless function. An overview of how a request is sent and what processes are involved is shown in Figure 5.5. The curl command is executed with the following parameters: `curl -w "@curl-format.txt" -o /dev/null -s "https://serverless-function-address.app/"`

the `curl-format.txt` formats the response. The following metrics are recorded (more details on [41]):

- **time_namelookup**: The time in seconds it took from the start until domain resolving was completed.
- **time_connect**: The time in seconds it took from the start until the three way handshake was completed from the clients perspective.
- **time_appconnect**: Is the time where the TLS setup is done. The client is able to send the HTTP GET request.
- **time_starttransfer**: The time it took until the first byte of the response was by curl. This is also known as the TTFB (Time to first byte). If we calculate `TTFB - (time_connect - time_namelookup)` then we get the time it took for the server to process the request.
- **time_total**: The total time in seconds is the duration between the time the request was sent and the time the last byte of the response was received.

These metrics provide valuable insights into the actual performance of the serverless functions and analyze the outliers caused by the network.

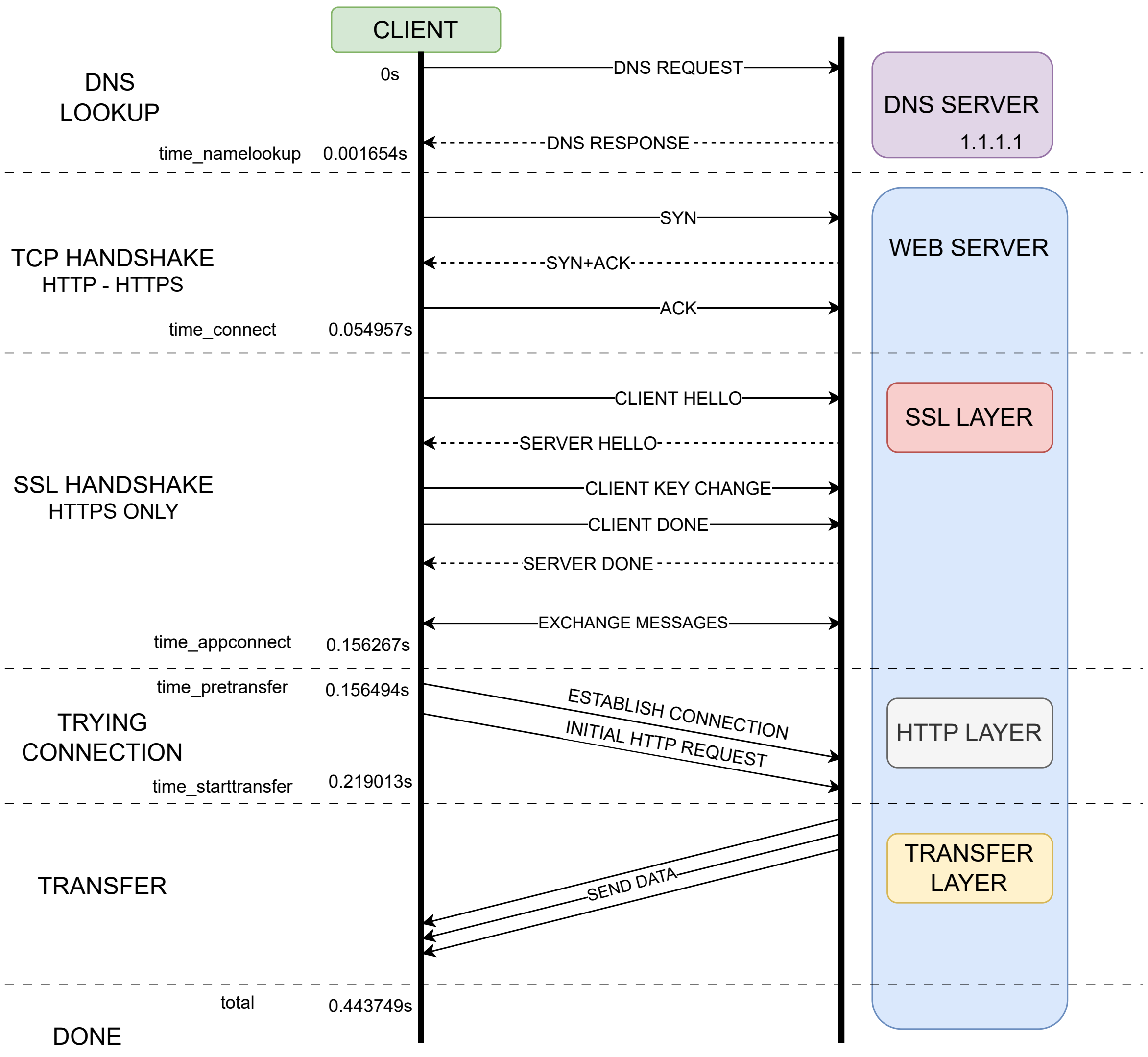

Figure 5.5: Curl performance metrics visualization, redrawn from [41]

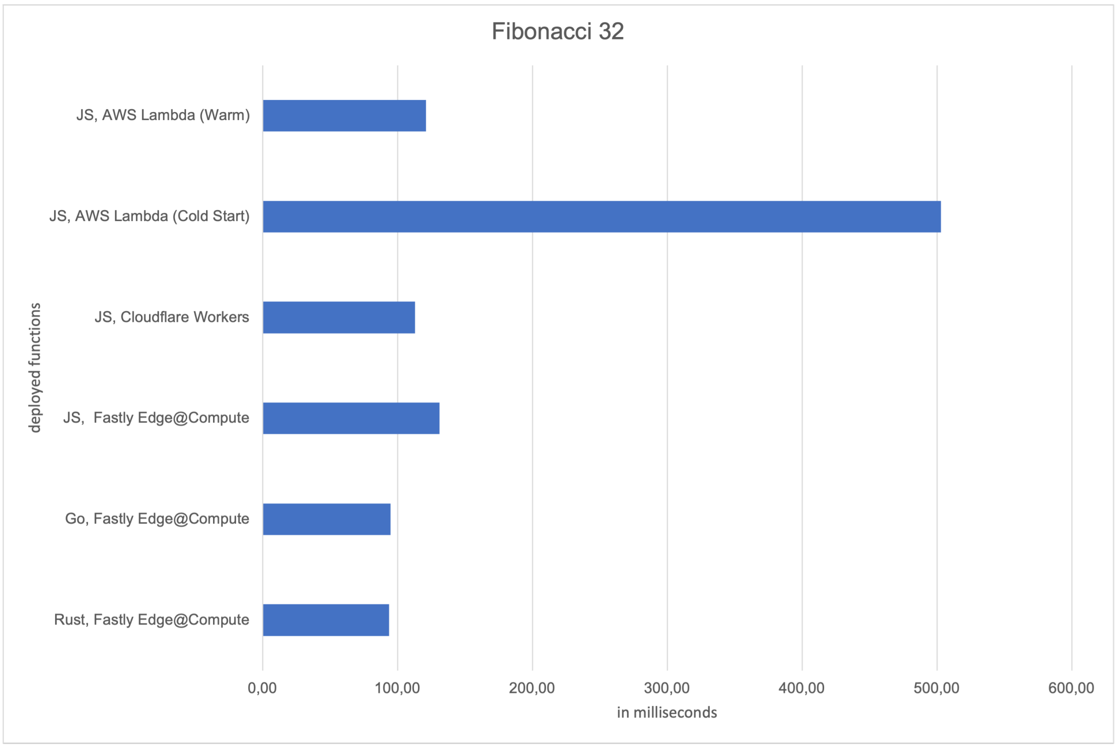

Figure 5.6: Derived results of deployed functions: Fibonacci 32, the result is calculated with TTFB - (time connect - time namelookup)

5.6 Discussion

In this section we will answer the questions from section 5.1 and discuss the findings of the evaluation. We will also address the limitations encountered during the evaluation.

5.6.1 Evaluation Questions

1. What is the startup time of a simple Wasm module?

We evaluated the startup time of a simple *WAT* file, loading a Rust to WASI-compiled file, and a Wasm module with a large data segment. These benchmarks, as shown in Figure 5.3, indicate a startup time ranging from 5 to 40 microseconds overall. Additionally, we evaluated the startup time of deserializing and loading a deserialized Wasm module. The startup time of a deserialized Wasm module is about 98 microseconds for a t2.medium instance (shown in Figure 0.5 in the appendix). Under adverse conditions, we could see a startup time of 1 to 5 milliseconds depending on the size of the Wasm module. The size of a bundled serverless function is typically small, and some cloud providers impose size limits.

2. How do different programming languages compare?

We evaluated the performance of the programming languages Rust, JavaScript, and Go. The performance of these programming languages is similar, with Rust being slightly faster than Go and JavaScript, as shown in Figure 5.6.

3. How do various runtimes compare to each other?

The results show that Wasmtime and Wasmer are similar in performance, with Wasmtime being slightly faster than Wasmer. In contrast, Wasm3, as shown in Table 5.2, is the slowest runtime in our evaluation. It is not surprising that Wasm3 is slower than Wasmtime and Wasmer, as Wasm3 is an interpreter runtime with a main focus on small footprint and portability. Wasm3 can find its use case in embedded systems, where the host devices may not be powerful enough to run a JIT compiler. It should be noted that our results show a worse performance for Wasm3 than the results presented in the official GitHub repository [30].

4. How do the WebAssembly runtimes compare to native code?

Initially, we assumed that WebAssembly runtimes are about 30% slower than native code. However, the results show that WebAssembly runtimes are approximately 40% slower on long-running tasks and about 35% slower on short-running tasks, as shown in Table 5.1. It is essential to note that WebAssembly runtimes outside of the browser are still in an early stage, and we can expect performance improvements in the future.

Secondly, we did not compile our Wasm module to a native binary, and it is reasonable to anticipate a performance improvement when doing so. This technique is used by the Fastly Compute@Edge platform once the optimized binary is deployed.

5. How does a WebAssembly-powered cloud platform measure up against a V8-powered cloud platform and a cloud platform with microVMs?

Cloudflare's Workers run on V8 isolates, and the only supported language for an isolate is JavaScript. The results show that Cloudflare Workers are slightly faster than the JavaScript version of Fastly Compute@Edge. We believe that the JavaScript runtime of Fastly Compute@Edge is not as mature as the V8 isolates employed by Cloudflare. However, the Rust version of Fastly outperforms the Cloudflare version.

Finally, we compared the performance of Fastly Compute@Edge with AWS Lambda. The results show that the cold start delay of AWS is about 400 milliseconds, as the first request took about 500 milliseconds, and the subsequent request took about 114 milliseconds. Even in a warm state, AWS Lambda is approximately as fast as the Fastly Compute@Edge JavaScript version. The results are shown in Figure 5.6.

5.6.2 Technical Limitations

The evaluation was limited to only three languages. We could not use Java for the evaluation because the WasmGC proposal is not merged in the specification.

Furthermore, a criterion dependency did not support the WASI target; therefore, we could not use it for plotting the runtime performance evaluation. Instead, we used the stored data to plot the results.

6 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 [42].

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 [43], which creates challenges for ensuring consistency and compatibility across cloud environments.

6.1 Key Motivations for shifting to a new Cloud Provider

There are various reasons why a customer may contemplate changing their service provider, the following are inspired by [43]:

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.

6.1.1 Service interoperability

The interoperability between different vendors is a notable factor in the decision to switch providers. It is an active decision of a cloud provider to make their services interoperable with other providers. Due to lack of standardization, we can already see smaller cloud providers offering services that are compatible with the larger cloud providers. For example, Cloudflare's R2 service is compatible with AWS S3 [44]. This allows customers to use Cloudflare's R2 service as a drop-in replacement for AWS S3, which is a significant advantage for customers who want to avoid vendor lock-in.

Another alternative is the collaboration between cloud providers. An excellent example of such a partnership is WinterCG, which is described as "a community group to provide a space for JavaScript runtimes to collaborate on API interoperability [45]". With this approach, cloud providers don't have to reinvent existing APIs and can focus on developing new features. On the other hand, this approach will make it easier for the developers to use the same codebase for different cloud providers.

6.1.2 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 containing "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 6.1: Basic standard AWS Lambda handler 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 6.2: Basic Gen. 2 Google Cloud handler function

Cloudflare Workers utilize V8 engine and thus they offer only JavaScript and TypeScript at the moment. The following example shows the handler function for a Cloudflare Worker. The fetch function comes with three parameters: "request", "env" and "ctx". The "request" parameter contains the HTTP request object, the "env" parameter contains the bindings assigned to the Worker and the "ctx" parameter contains commands like `ctx.waitUntil` or `ctx.passThroughOnException` to control the Worker's behavior.

```
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 6.3: Basic Cloudflare Workers hanlder function

Lastly, the following example shows a basic Fastly Compute@Edge [46] function. Fastly functions are working with WebAssembly and Wasmtime runtime. The function explicitly adds an event listener to the "fetch" event. The "fetch" event is triggered when a request is made to the function. The function then parses the request body and returns a response with a status code of 200 and a JSON body containing a "message" property. As shown in the example below, the fetch handler does only have one parameter, the "event" parameter. The "event" parameter contains the request object.

```
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 6.4: Basic Fastly Compute@Edge handler function

6.2 Design principles

6.2.1 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.

6.2.2 Adapter pattern

SvelteKit

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 [47]. 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 6.5: svelte.config.js SvelteKit adapter configuration

Hono Web Framework

Hono [48] is a fast web framework which is also following the adapter pattern to allow developers to deploy their applications to various cloud providers without the need to change the business logic. The listing below shows a simple Hono endpoint which would look and work the same for Cloudflare Workers, Fastly Compute@Edge, Deno, Bun, Vercel, Lagon, AWS Lambda, and Node.js according to Hono js documentation [48]. The only difference is in the adapter configuration and depending on the platform also the export line.

```
import { Hono } from 'hono'
const app = new Hono()

app.get('/', (c) => c.text('Hono!'))

export default app
```

Listing 6.6: Standard Hono Handler function

7 Related Work

This chapter presents related approaches for handling cold start problem in serverless computing such as pre-warming, caching, and container reuse. It also presents related work on WebAssembly and V8 isolates.

7.1 Workaround Approaches

There is a lot of research in the area of mitigating cold start problem in serverless computing. This section presents the incremental approaches, techniques to work around the cold start problem.

7.1.1 Pre-warming

In [49] Glikson and Lin propose to maintain a pool of pre-provisioned "warm" container, ready for invocation with zero cold start time. They implemented the pool for Knative platform. The authors show that the first request is 2.3 times faster, however this approach still suffers from the cold start problem, because a burst of requests can still exhaust the pool of warm containers. Moreover, the pre-warming approach is not cost effective, because the user is billed for the idle containers.

7.1.2 Prediction

With the current state of serverless platforms, the vendors have a fixed keep-alive policy, for example AWS Lambda functions shuts down after 7 minutes of inactivity. Shahrad et al. describes an improved alternative to the fixed keep-alive policy [50] by observing the entire functions workload. The results of the observation show that functions are invoked very infrequently, in fact by an 8-order-of-magnitude in terms of range of invocation frequencies. Based on this results the authors propose policies by set of rules that significantly reduces the number of function cold starts. The first part is the histogram policy, the algorithm observes and captures the times and frequency of function invocation. On periodic invocations the algorithm updates the histogram and predicts the next invocation time. With this data the algorithm can predict the next invocation time and keep the function alive or pre-warm the function until the next invocation. On the other hand, if there is too many out of bound invocations, the prediction model with histogram won't work, thus the algorithm will switch to time series forecast. The authors show that only 18% of the functions are invoked infrequently and the rest 82% of the functions are invoked within 1 minute average.

This approach is very promising, especially for the existing serverless platforms that have a slow cold start time with fixed keep-alive policy. Moreover, it is also framework independent, thus it can also be applied additionally to our WebAssembly approach.

7.1.3 Container Reuse

In [51] Oliver Stenbom explores the idea of checkpointing and restoring a container while it runs. The thesis also considers the added complexity to the setup. The authors results show that restoring NodeJS, Python and Java functions take 5-100ms to restore, 20 times faster than OpenWhisk's 2 second cold start. The author concludes that the approach is feasible and beneficial but requires more work to be production ready.

While this approach demonstrates improved results, it comes with increased overhead and complexity. Moreover, the cold start time (5-100ms) with this approach is inferior to our approach.

7.2 Containerless Compute - WebAssembly

Now in this section we will go through the WebAssembly and V8 isolates related work.

Philipp Gackstatter et al. designed and implemented a prototype for WebAssembly execution in Apache OpenWhisk [1, 52]. The authors show that the cold start latency can be reduced by up to 99.5% compared to the current OpenWhisk implementation. The authors also show that the WebAssembly approach is more cost effective than the current OpenWhisk implementation. The authors conclude that the WebAssembly approach is a promising alternative to the current OpenWhisk implementation.

7.2.1 Fastly Compute@Edge

Fastly Compute@Edge [46] is a leading WebAssembly-based serverless platform. The platform uses the Wasmtime runtime, the successor to its Lucet runtime, and is actively collaborating with Bytecode Alliance to further develop it. Fastly Compute@Edge is a key focus of evaluation in this thesis. The results of the evaluation show a consistently fast cold start of 50-350 microseconds, which represents a significant improvement compared to the abovementioned approaches. Currently, the platform only officially supports Rust, AssemblyScript, and JavaScript.

7.2.2 Cloudflare Workers

Cloudflare Workers is a serverless platform offering from Cloudflare that uses Google's V8 Engine to run JavaScript code with V8 isolates [4]. The platform's performance is also evaluated in this thesis. With V8 isolates, the platform achieves a cold start time of 5-10 milliseconds, which is much faster than traditional serverless offerings but slower than Fastly Compute@Edge. Additionally, the platform supports WebAssembly, profiting from the V8 Engine's built-in WebAssembly runtime. Cloudflare is actively working on projects like workers-rs to facilitate easier WebAssembly binding and the support of languages such as Rust and Go with the help of WebAssembly.

8 Conclusion

In this thesis, we have evaluated the WebAssembly technology for serverless computing. The results of this work show a significant improvement in the cold start time of serverless functions by utilizing WebAssembly as the runtime environment. The main challenge for the adaptation of WebAssembly in mainstream cloud computing is the current support of the technology. While the support for WebAssembly is growing, it still lacks crucial features such as direct access to the system clock or the ability to spawn new threads. Another limitation is the lack of support for the Java language, which is widely used in the industry.

Aside from the current limitations, which are expected to be resolved in the near future through proposals, WebAssembly is a promising technology for serverless computing.

We have demonstrated how easy it is to integrate a WebAssembly runtime environment into an existing serverless platform. With the help of the runtime API, we can configure the runtime environment to our needs.

Another important aspect of this work is Vendor Lock-in. In summary, we can split vendor lock-in into two categories. The first one is the lock-in to a specific cloud provider service offering, which is not influenced by the employed technology. The second category is the lock-in to a specific technology. In the case of WebAssembly, we see no lock-in to a specific technology, as WebAssembly acts as the underlying runtime. We have evaluated the adapter pattern, which can be used to abstract the underlying runtime and allow the developer to switch between different cloud providers.

8.1 A Note on Sustainability

While sustainability is a broad topic and not the focus of this thesis, it is a factor to consider when it comes to choosing a technology, especially considering the impact of data centers power consumption which accounts for approximately 1% to 1.25% of the world's total consumption [53]. Serverless functions are typically short-lived, which means that the cold start time plays a major role in the total execution time. As shown by the results in the evaluation, by utilizing WebAssembly, the cold start time can be almost eliminated, making serverless functions more sustainable in that area.

9 Future work

This work provides a solid foundation for future work on the capabilities and aspects of WebAssembly in serverless environment. This section will discuss some of the important areas that can be explored in the future works:

- **Wasm GC**: The WebAssembly garbage collection proposal, though not yet finalized, it is in a state that allows initial evaluations. Kotlin has already published an experimental compiler that targets the Wasm GC proposal. Once the proposal is merged into the primary spec repository and memory-managed languages like Kotlin and C# and Java adapt the Wasm garbage collection proposal, it would be interesting to evaluate the performance of these languages in serverless environment. The future work can also answer the question of whether memory-managed languages can be used in edge computing environment and how they perform in comparison to their non memory-managed counterparts like Rust and C++.
- **Web Frameworks**: The serverless area of WebAssembly can gain traction if popular web frameworks such as SvelteKit, Next.js, Nuxt.js, Solid, and many others could be hosted on WebAssembly-powered serverless platforms. We've observed that SvelteKit, along with possibly other web frameworks, is adopting the adapter pattern [47] to support multiple serverless platforms. Therefore, a future work can be to create an adapter for an existing WebAssembly platform, then assessing the feasibility and challenges associated with hosting web frameworks on these WebAssembly-powered serverless platforms.
- **Asynchronous with WASI**: Soon the WASI community will start with preview3. The preview3 [21] will include the support for asynchronous operations, which will also introduce new types such as `future<T>` and `stream<T, E>`. Our approach of benchmarking the serverless platforms can be extended to benchmark the asynchronous operations in WebAssembly. The future work can extensively evaluate I/O intensive workloads with WASI.
- **Security**: The security aspects of WebAssembly in serverless environment is a very important area of research. We have seen that AWS Lambda is using Firecracker microVM to run the serverless functions. The microVMs are chosen in favor of containers because of the security benefits they provide. Therefore, future research is essential to thoroughly evaluate the security facets of WebAssembly within a serverless environment.
- **Benchmarking I/O bound workloads**: The current work focuses on CPU intensive workloads. However, the serverless functions are sometimes I/O bound. Therefore, a future work can be to benchmark the I/O bound workloads in WebAssembly.

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, 24, 43, 52

[2] WebAssembly, "WebAssembly/WASI: Webassembly system interface," GitHub, 03 2023. [Online]. Available: https://github.com/WebAssembly/WASI 1, 51

[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, 52

[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, 43

[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, 5, 6, 7, 9, 17

[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] M. Foundation, "JavaScript modules," MDN Web Docs / ES-Modules. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules 4

[9] B. Sletten, *WebAssembly: the Definitive Guide.* O'Reilly Media, Inc., 12 2021. 4, 7, 16

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

[11] 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

[12] 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

[13] W. WG, "WebAssembly proposals," WebAssembly GitHub Page, 04 2023. [Online]. Available: https://github.com/WebAssembly/proposals 6, 7, 8, 53

[14] B. Couriol, "WebAssembly Reference Types Implemented in wasmtime, Lets Wasm Modules Handle Complex Types," InfoQ, 09 2020. [Online]. Available: https://www.infoq.com/news/2020/09/wasm-reference-types-wasmtime 6, 9

[15] N. Fitzgerald, “Multi-Value All the Wasm!” Bytecode Alliance, 11 2019. [Online]. Available: https://bytecodealliance.org/articles/multi-value-all-the-wasm 7

[16] ——, “WebAssembly Reference Types in Wasmtime,” fitzgeraldnick.com, 08 2020. [Online]. Available: https://fitzgeraldnick.com/2020/08/27/reference-types-in-wasmtime.html 9

[17] W. Community, “GC Proposal for WebAssembly,” GitHub / Wasm GC, 05 2023. [Online]. Available: https://github.com/WebAssembly/gc 9

[18] V. Sekhar, “Mobile app development with WebAssembly GC,” Wasm I/O 2023, 03 2023. [Online]. Available: https://www.youtube.com/watch?v=fDNXomIn53Y 10, 11, 52

[19] Kotlin, “Kotlin Wasm | Kotlin,” Kotlin, 05 2023. [Online]. Available: https://kotlinlang.org/docs/wasm-overview.html 11

[20] L. Clark, “WASI: a New Kind of System Interface,” InfoQ / QCon plus, 02 2022. [Online]. Available: https://www.infoq.com/presentations/wasi-system-interface 12

[21] D. Gohman, “WASI roadmap,” GitHub / WebAssembly Meetings Repository, 05 2023. [Online]. Available: https://github.com/WebAssembly/meetings/blob/main/wasi/2023/presentations/2023-02-09-gohman-wasi-roadmap.pdf 12, 45

[22] ——, “WASI overview,” GitHub / Wasmtime Respository, 11 2019. [Online]. Available: https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md 12, 13, 52

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

[24] 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/ 14, 15, 16, 52

[25] A. Zakai, “Outside the web: standalone WebAssembly binaries using Emscripten · V8,” v8.dev, 11 2019. [Online]. Available: https://v8.dev/blog/emscripten-standalone-wasm 14

[26] L. Clark, “Wasmtime reaches 1.0: Fast, safe and production ready!” Bytecode Alliance, 09 2022. [Online]. Available: https://bytecodealliance.org/articles/wasmtime-1-0-fast-safe-and-production-ready 15, 19

[27] S. Akinyemi, “Awesome WebAssembly Runtimes,” GitHub / Awesome WebAssembly Runtimes, 05 2023. [Online]. Available: https://github.com/appcypher/awesome-wasm-runtimes 17, 53

[28] B. Alliance, “Wasmtime,” GitHub / Wasmtime Repository, 11 2022. [Online]. Available: https://github.com/bytecodealliance/wasmtime 18, 21, 24

[29] W. Inc., “Wasmer,” GitHub / Wasmer Repository, 05 2023. [Online]. Available: https://github.com/wasmerio/wasmer 18, 24

[30] V. Shymanskyy, S. Massey, and M. Graey, “Wasm3,” GitHub / Wasm3, 05 2023. [Online]. Available: https://github.com/wasm3/wasm3 18, 21, 37

[31] Y.-Y. He and WasmEdge, “WasmEdge,” GitHub / WasmEdge, 05 2023. [Online]. Available: https://github.com/WasmEdge/WasmEdge 18

[32] C. N. C. Foundation, “Cloud Native Computing Foundation,” Cloud Native Computing Foundation (CNCF), 05 2023. [Online]. Available: https://www.cncf.io 18

[33] "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 19

[34] C. Inc., "How Workers work · Cloudflare Workers Docs," Cloudflare, 01 2023. [Online]. Available: https://developers.cloudflare.com/workers/learning/how-workers-works/ 19, 52

[35] M. Butcher and R. Matei, "The Six Ways of Optimizing WebAssembly," InfoQ, 01 2023. [Online]. Available: https://www.infoq.com/articles/six-ways-optimize-webassembly 20, 21

[36] R. L. Community, "Customizing Builds with Release Profiles / The Rust Programming Language," Rust Lang Documentation, 02 2021. [Online]. Available: https://doc.rust-lang.org/book/ch14-01-release-profiles.html 21

[37] 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 22, 52

[38] B. Alliance, "Wizer Github Repository," GitHub, 04 2023. [Online]. Available: https://github.com/bytecodealliance/wizer 23

[39] A. Team, "Actix," actix.rs. [Online]. Available: https://actix.rs 24

[40] B. Heisler, "Criterion.rs: Statistics-driven benchmarking library for Rust," GitHub / Criterion.rs, 05 2023. [Online]. Available: https://github.com/bheisler/criterion.rs 29

[41] S. T. Demon, "Cheat Sheet on Curl Performance Metrics: how to benchmark server latency with curl," Speed Test Demon, 07 2021. [Online]. Available: https://speedtestdemon.com/a-guide-to-curls-performance-metrics-how-to-analyze-a-speed-test-result 34, 35, 52

[42] 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/ 38

[43] *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 38

[44] I. Cloudflare, "Cloudflare R2 | Zero Egress Distributed Object Storage," Cloudflare. [Online]. Available: https://www.cloudflare.com/en-gb/products/r2 38

[45] W. interoperable Runtimes Community Group, "WinterCG," Web-interoperable Runtimes Community Group. [Online]. Available: https://wintercg.org 38

[46] I. Fastly, "Serverless Edge Compute Solutions | Fastly," www.fastly.com. [Online]. Available: https://www.fastly.com/products/edge-compute 40, 43

[47] S. Community, "Adapter Documentation SvelteKit," SvelteKit, 04 2023. [Online]. Available: https://kit.svelte.dev/docs/adapters 41, 45

[48] H. Community, "Hono - Ultrafast web framework for the Edges," hono.dev, 05 2023. [Online]. Available: https://hono.dev 41

[49] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," 2019. 42

[50] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a

large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad 42

[51] O. Stenbom, "MEng Individual Project Refunction: Eliminating Serverless Cold Starts Through Container Reuse," Ph.D. dissertation, 06 2019. [Online]. Available: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1819-ug-projects/StenbomO-Refunction-Eliminating-Serverless-Cold-Starts-Through-Container-Reuse.pdf 43

[52] T. A. S. Foundation, "Apache OpenWhisk is a serverless, open source cloud platform," Apache.org, 2016. [Online]. Available: https://openwhisk.apache.org/ 43

[53] P. Patros, J. Spillner, A. V. Papadopoulos, B. Varghese, O. Rana, and S. Dustdar, "Toward sustainable serverless computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021. 44

[54] 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 50

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 [54].. 2, 38, 39

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, 18, 23, 26

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.. 39

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. This pattern provides an abstraction layer to decouple the more complex business logic from the client code.. 40

ICMP ICMP, short for Internet Control Message Protocol, is a network layer protocol that operates at the network layer (Layer 3) of the OSI model. It is used for error reporting, network diagnostics, and network congestion control.. 29

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, 19, 52

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.. 12, 14

Knative Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. Serverless Containers in Kubernetes environments.. 42

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.. 12, 14

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.. 12, 18

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.. 12, 14

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. The customer only pays for the execution time of the functions.. 1, 2, 16, 17, 22, 24, 26, 39, 40, 41, 45

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.. 2, 19, 39, 52

WASI The WebAssembly System Interface is not a monolithic standard system interface, but is instead a modular collection of standardized APIs. None of the APIs are required to be implemented to have a compliant runtime. Instead, host environments can choose which APIs make sense for their use cases [2].. 36, 37

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, 4, 6, 9, 14, 18, 20, 24, 28, 36, 45

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("127.0.0.1:8000")?
        .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> {
    let engine = Engine::default();
    let mut linker = Linker::new(&engine);
    wasmtime_wasi::add_to_linker(&mut linker, |s| s)?;

    let stdout_vec_buf: Vec<u8> = vec![]; // a buffer that will contain the response
    let stdout_vec_mutex = Arc::new(RwLock::new(stdout_vec_buf));
    let stdout = WritePipe::from_shared(stdout_vec_mutex.clone());

    // params to 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();

    // get the response
    let mut buffer: Vec<u8> = Vec::new();
    stdout_vec_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

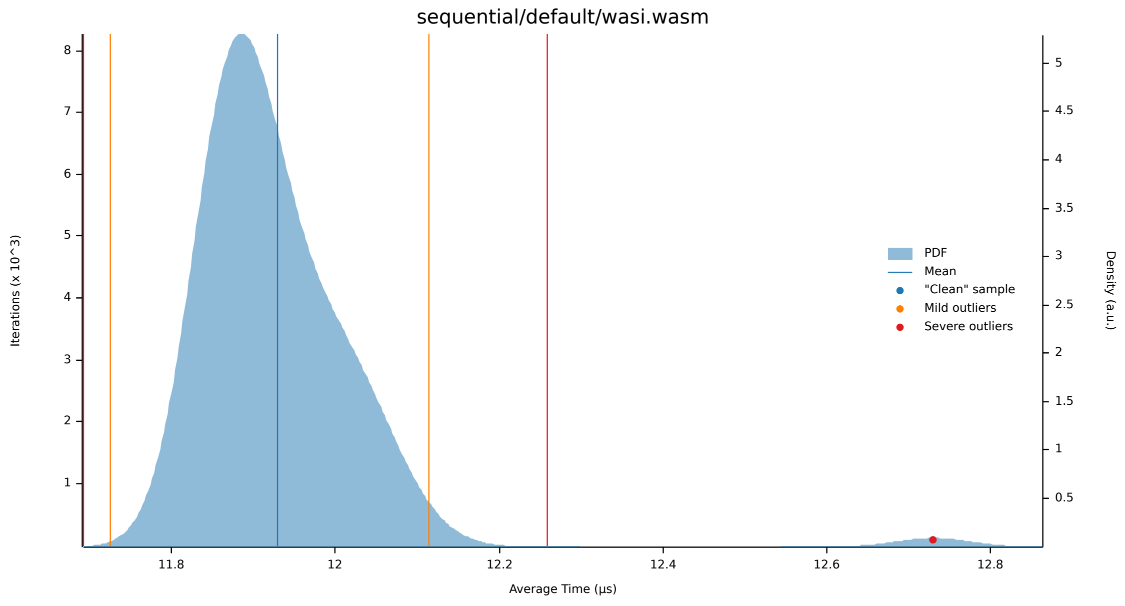

Figure 0.1: t2.medium, distribution: average instantiation time of a WASI module with Wasmtime

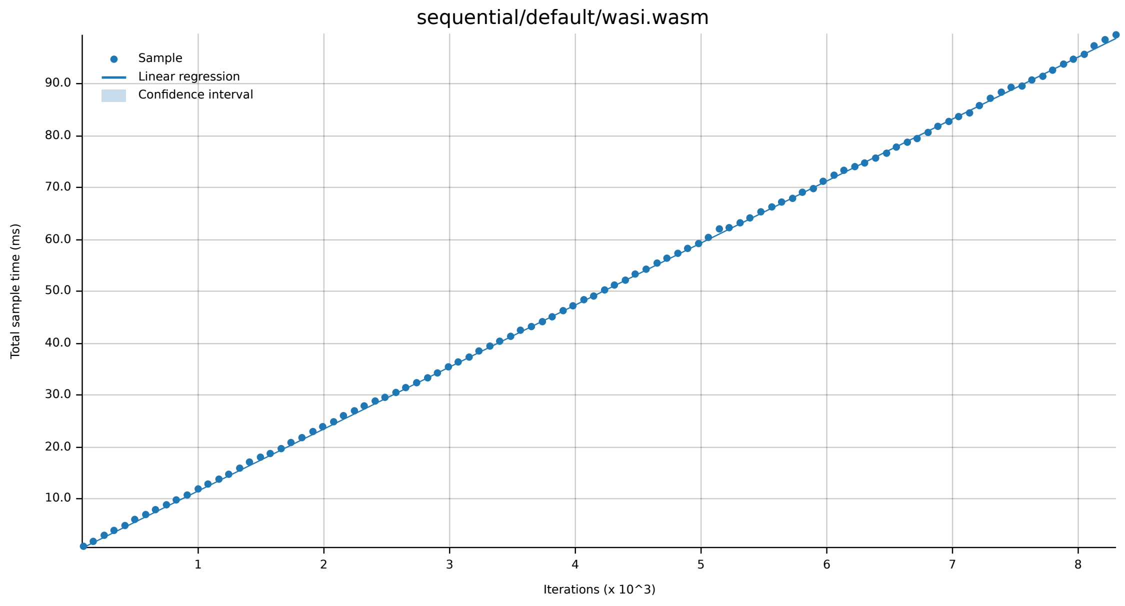

Figure 0.2: t2.medium, linear regression: instantiation of a WASI module with Wasmtime (a smaller distance to the regression line indicates better accuracy)

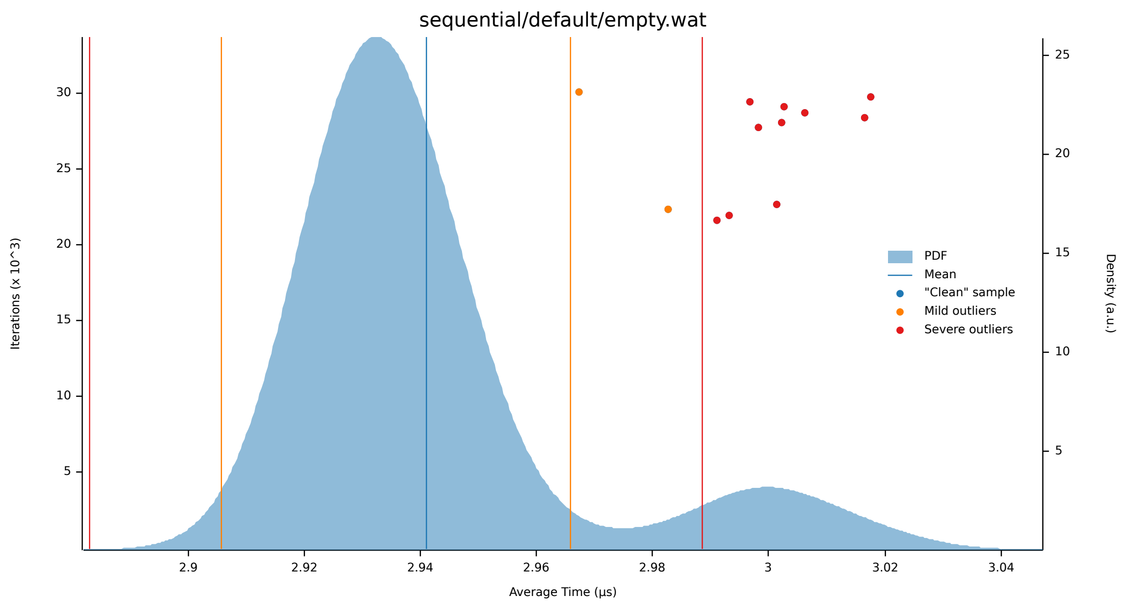

Figure 0.3: t2.medium, distribution: average instantiation time of an empty Wasm module with Wasmtime

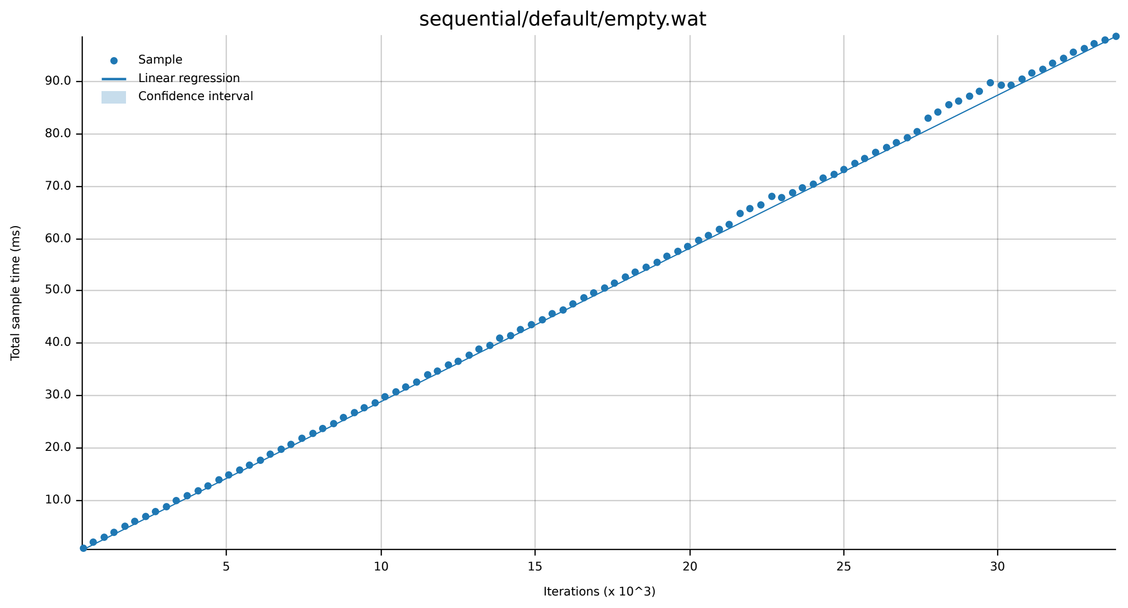

Figure 0.4: t2.medium, linear regression: instantiation of an empty Wasm module with Wasmtime (a smaller distance to the regression line indicates better accuracy)

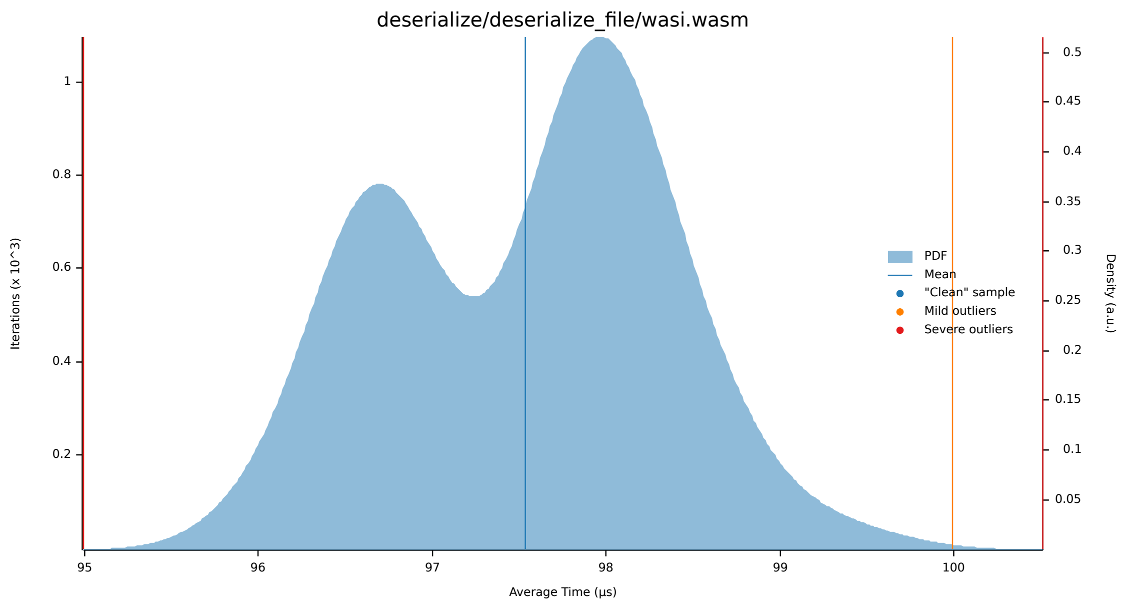

Figure 0.5: t2.medium, distribution: average deserialization and instantiation time of a WASI module with Wasmtime

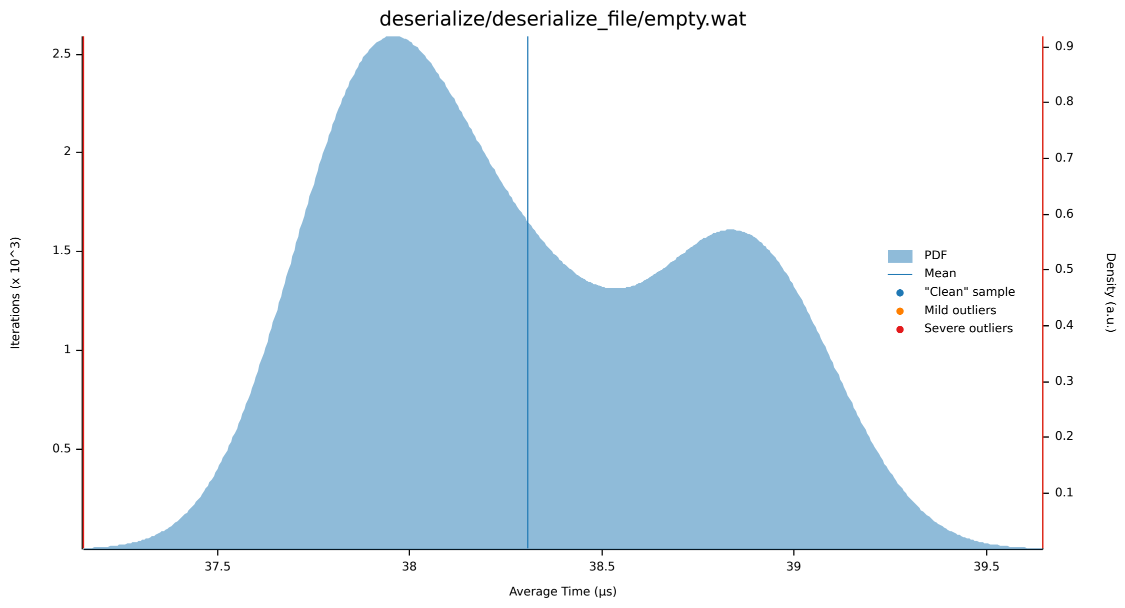

Figure 0.6: t2.medium, distribution: average deserialization and instantiation time of an empty Wasm module with Wasmtime

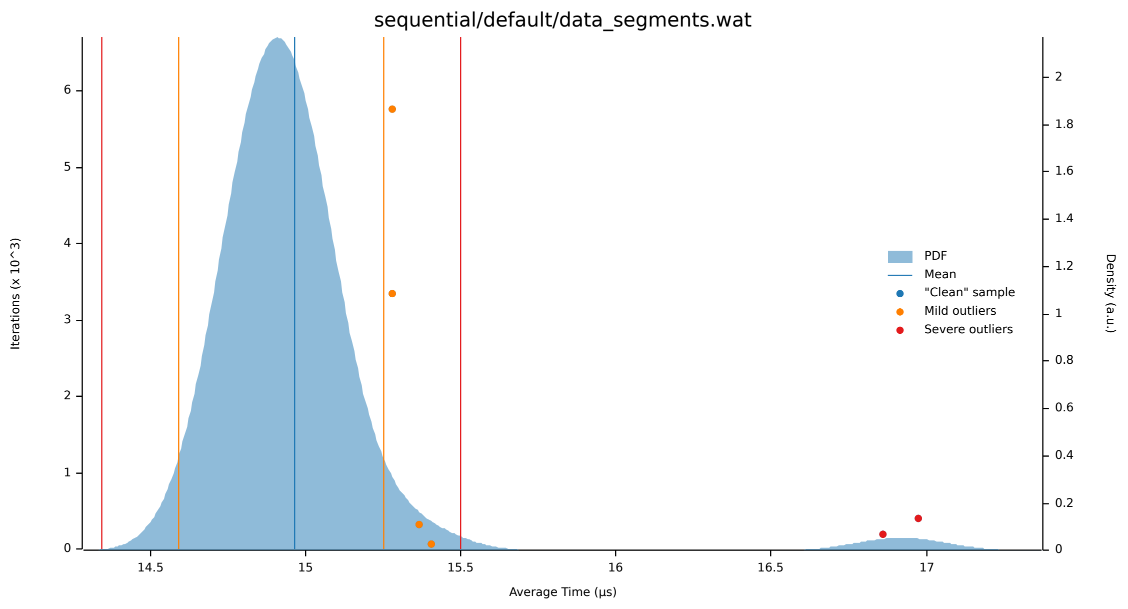

Figure 0.7: M1 Pro CPU, distribution: average instantiation time of a WASI module containing a large data segment with Wasmtime

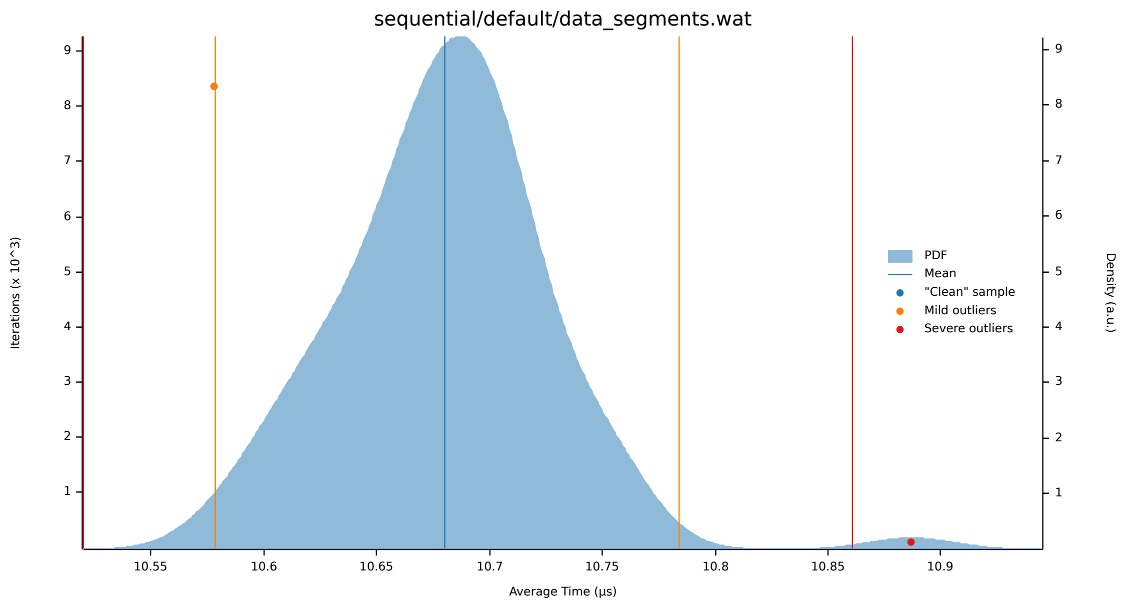

Figure 0.8: t2.medium, distribution: average instantiation time of a WASI module containing a large data segment with Wasmtime

Serverless Platform Evaluation Curl Results

```
# process_time is TTFB − (time_connect − time_namelookup)
# Time To First Byte (TTFB) alias time_starttransfer

# Rust, Fibonacci 32, Fastly Compute@Edge
time_namelookup: 0.000820s
time_connect: 0.001207s
time_appconnect: 0.086769s
time_pretransfer: 0.086885s
time_redirect: 0.000000s
time_starttransfer: 0.094141s
−−−−−−−−−−
time_total: 0.094196s
process_time: 0.094141 − (0.001207 − 0.000820) = 0.093754s = 93,75ms

# Go, Fibonacci 32, Fastly Compute@Edge
time_namelookup: 0.000813s
time_connect: 0.001200s
time_appconnect: 0.086132s
time_pretransfer: 0.086228s
time_redirect: 0.000000s
time_starttransfer: 0.095094s
−−−−−−−−−−
time_total: 0.095142s
process_time: 0.095094 − (0.001200 − 0.000813) = 0.094707s = 94,71ms

# JavaScript, Fibonacci 32, Fastly Compute@Edge
time_namelookup: 0.000821s
time_connect: 0.001200s
time_appconnect: 0.085285s
time_pretransfer: 0.085414s
time_redirect: 0.000000s
time_starttransfer: 0.131382s
−−−−−−−−−−
time_total: 0.131449s
process_time: 0.131382 − (0.001200 − 0.000821) = 0.131003s = 131ms

# JavaScript, Fibonacci 32, Cloudflare Workers
time_namelookup: 0.000758s
time_connect: 0.001870s
time_appconnect: 0.090422s
time_pretransfer: 0.090531s
time_redirect: 0.000000s
time_starttransfer: 0.114824s
−−−−−−−−−−
time_total: 0.114920s
process_time: 0.114824 − (0.001870 − 0.000758) = 0.113712s = 113ms
```

```
# JavaScript, Fibonacci 32, AWS Lambda, COLD START
time_namelookup:      0.004100s
time_connect:         0.005111s
time_appconnect:      0.085122s
time_pretransfer:     0.085169s
time_redirect:        0.000000s
time_starttransfer:   0.504372s
------------------------------------------
time_total:           0.504477s
process_time: 0.504372 - (0.005111 - 0.004100) = 0.503361s = 503ms

# JavaScript, Fibonacci 32, AWS Lambda, WARM START
time_namelookup: 0.000788s
time_connect: 0.001157s
time_appconnect: 0.085358s
time_pretransfer: 0.085399s
time_redirect: 0.000000s
time_starttransfer: 0.121778s
----------
time_total: 0.121837s
process_time: 0.121778 - (0.001157 - 0.000788) = 0.121409s = 121ms

# Rust, Blake3 100.000 iterations, Fastly Compute@Edge
time_namelookup:      0.024949s
time_connect:         0.026306s
time_appconnect:      0.106740s
time_pretransfer:     0.106850s
time_redirect:        0.000000s
time_starttransfer:   0.135364s
------------------------------------------
time_total:           0.135422s
process_time: 0.135364 - (0.026306 - 0.024949) = 0.133007s = 133ms
```