FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Build pipeline for edge computing applications
Student:	Bc. Jan Chybík
Supervisor:	Ing. Daniel Sedlák
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

Edge computing has recently started gaining popularity among all significant CDN providers. It allows data to be processed closer to its clients rather than relying on a centralized data processing warehouse or distant cloud servers that can introduce higher latency.

This thesis aims to design and develop a build pipeline tailored for edge computing applications. The purpose of the pipeline is to accept untrusted source code and compile it to the WebAssembly executable, which can be later executed on edge servers. It's crucial to thoroughly consider potential security vulnerabilities that may arise during the compilation and execution of untrusted user input.

In the thesis consider the following requirements:
- Familiarize yourself with edge computing technology
- Learn about WebAssembly and its use cases for edge computing
- Analyze security vulnerabilities caused by untrusted code
- Design the build pipeline with security and scalability in mind
- Implement the pipeline based on the design

Optionally, try different build optimization techniques for WebAssembly code to optimize start-up and execution speed, thus limiting cold start problems for edge computing applications.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 8 January 2024 in Prague.

Master's thesis

BUILD PIPELINE FOR EDGE COMPUTING APPLICATIONS

Bc. Jan Chybík

Faculty of Information Technology
Department of theoretical computer science
Supervisor: Ing. Daniel Sedlák
May 8, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Chybík Jan. *Build pipeline for edge computing applications.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

I would like to thank my supervisor, Ing. Daniel Sedlák, for his guidance. His always-on-point advices made this thesis possible. I would also like to thank my family and my girlfriend, you are the reason I kept going, thank you.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on May 8, 2024

Abstract

Edge computing is currently gaining popularity among many CDN providers. It moves data processing closer to its clients, reducing latency compared to standard centralized data processing warehouses or cloud servers.

This thesis analyzes edge computing technology with WebAssembly and explores how it can be used together. Based on these findings, the thesis designs and implements a build pipeline that accepts untrusted source code and compiles it to the WebAssembly executable, which can be later executed on edge servers.

Keywords Edge computing, WebAssembly, Proxy Wasm, Untrusted code, Compilation

Abstrakt

Edge computing aktuálně nabírá na popularitě mezi mnoha poskytovateli CDN. Přesouvá zpracování dat blíže klientům, čímž snižuje odezvu oproti centralizovaným datovým skladům, nebo vzdáleným cloudovým serverům.

Tato práce analyzuje edge computing s technologii WebAssembly a zkoumá jak mohou být použity společně. Na základě těchto zjištění pak vytváří návrh a implementaci kompilační pipeline, která přijímá nedůvěryhodný zdrojový kód a zkompiluje ho do WebAssembly spustitelného souboru, který může být později spuštěn na edge serverech.

Klíčová slova Edge computing, WebAssembly, Proxy Wasm, Nedůvěryhodný kód, Kompilace

List of abbreviations

ABI	Application binary interface
AOT	Ahead of time
CDN	Content delivery network
CLI	Command line interface
CRUD	Create, Read, Update, Delete
DoS	Denail of service
EC	Edge computing
FFI	Foreign function interface
GDB	The GNU Project debugger
IDL	Interface definition language
IoT	Internet of things
JIT	Just in time
KVM	Kernel-base virtual machine
LLDB	Low-level debugger
SDK	Software development kit
VM	Virtual machine
VMM	Virtual machine manager
WASI	WebAssembly System Interface
Wasm	WebAssembly

Introduction

Edge computing is recently gaining traction among many CDN providers. It moves data processing closer to clients, reducing latency compared to centralized data warehouses or distant cloud servers.

By using edge computing, companies gain an advantage in the ever-changing landscape of modern technology, and thus, they must use it to its maximum potential.

A common problem for all edge computing platforms is the execution of untrusted code written in any language a client wants. This problem can be solved by using WebAssembly. A relatively new binary instruction format into which many languages can be compiled and safely executed.

This thesis looks into edge computing with WebAssembly and explores why and how they can be used together. Furthermore, it studies the dangers of untrusted code and how to mitigate them.

Based on these findings, the thesis proposes a design for a compilation pipeline that takes untrusted source code as its input and compiles it into the WebAssembly executable, which can be later executed on edge servers.

The design is then transformed into a production-ready implementation, following all software best practices with end-to-end tests, ensuring the pipeline works correctly. And lastly, the thesis tries to sketch out how it could optimize compilation to reduce cold starts of the WebAssemlby binaries.

Chapter 1

About this thesis

1.1 Thesis structure

This thesis is divided into two parts. The first part of the thesis, research (Part I), focuses on all necessary topics that are important to understand the subject of the thesis. This includes chapters about edge computing, WebAssembly, and security. The edge computing chapter goes into types of distributed systems and why they are beneficial. It explains in which ways edge computing is better than other distributed systems and establishes its use case.

The next chapter is about WebAssembly. It explains what exactly WebAssembly is, its core specification, and ways how to extend it. It also focuses on how WebAssembly can be used as a proxy, which operates on the network's edge. The last chapter of the research part is about security. It is crucial to consider security when handling any production application. This chapter focuses on ways in which an untrusted code can exploit an application and how to mitigate such attempts.

The second part of the thesis, the implementation (Part II), shows a way how to design and implement a solution that handles the building process of applications running on edge servers. It uses all concepts from the research part and creates a solution that is usable, safe, and easily extensible. After the implementation chapter, the thesis explores how WebAssembly startups can be optimized by going into each step of the process. The whole part ends with a chapter about future work, which states how the development will continue and in which ways the platform could be extended to provide the best outcome possible.

1.2 Related work

Edge computing is currently a hot topic because of its benefits against centralized cloud computing and the wide adoption of IoT devices. Several research papers were published analyzing edge computing as a whole or only its parts.

The first paper that this thesis would like to mention is *Edge Computing [Scanning the Issue]* [1]. It provides general information about edge computing, its history, present, and future. It gives an overview of why edge computing is required to develop IoT and other platforms further.

The second paper, *Resource Management in Fog/Edge Computing* [2], goes into many other works identifying and classifying different architectures, infrastructures, and algorithms for managing resources in edge computing.

And last, the *A New Multi-Target Compiler Architecture for Edge-Devices and Cloud Management* [3] analyzes how edge applications are used and proposes a new multi-target compiler architecture that can compile source code that can be distributed across different edge devices.

Part I

Research

Chapter 2

Computing systems

2.1 Distributed systems

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system. [4]

The computing element is usually referred to as a node. Node internals are generally hidden, it can be either a hardware device or a software process with high or low performance. *Nodes serve as hosts for computation and storage. They can execute a program, store data in volatile and stable memory, and send messages to other nodes.* [5] Nodes act independently from each other. However, even if they are independent, they are connected and communicate together to achieve a common goal.

This network of nodes, also known as a *distributed system*, appears as a single coherent system to the outside world. Ideally, users should not know they are dealing with a distributed system processing data all over the network. This creates a convenient abstraction for users because they can interact with the system like any other non-distributed system. Meanwhile, the distributed system can change and scale while keeping the same interface. However, achieving this goal poses a challenge.

A distributed system is usually more than just a single node. This creates unexpected behavior when the right conditions are met. A typical problem of distributed systems is a *partial failure* from which all complex systems suffer. At any given time, only part of the system fails. This leads to some applications working and some failing. Another problem that distributed systems have to face is a problem with latency. Because a system can be distributed worldwide, it may take a while for some nodes to communicate with each other or the user.

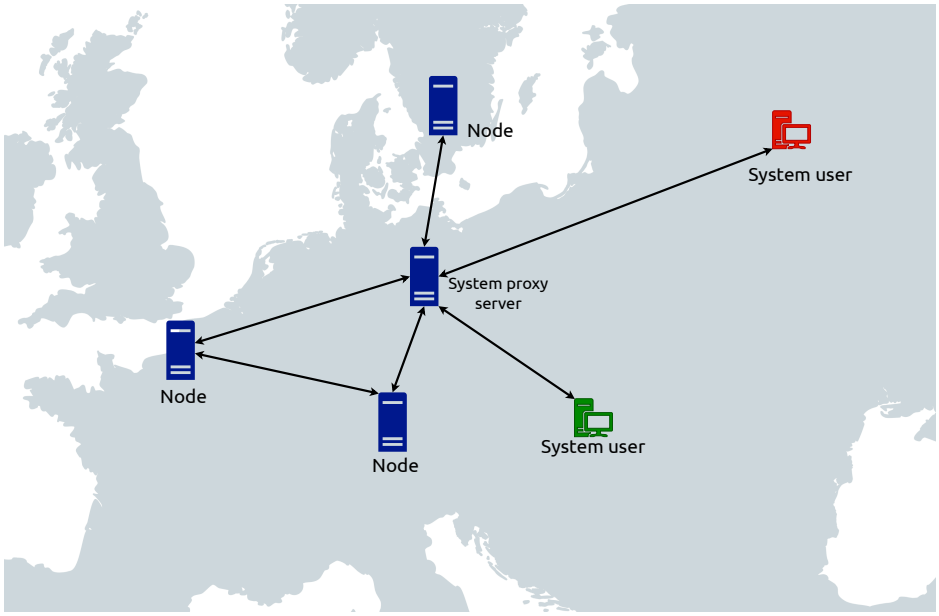


Figure 2.1 Example of distributed system

Nevertheless, a distributed system can sometimes reveal that it is distributed. It can leverage the fact that it is located globally and can offer location-specific services. Sometimes, it might make sense to be open about the system's distributed nature so users can understand the unexpected behavior. [4] Figure 2.1 shows what a distributed system could look like.

2.2 Scaling

An important aspect of distributed systems that was already briefly mentioned is scaling. One of the most significant benefits of a distributed system is that it can scale almost indefinitely by introducing new nodes, which are either wholly new or replicas of already existing ones.

2.2.1 Scalability dimensions

Scalability of a system can be measured along at least three different dimensions [4]:

1. *Size scalability*: Users do not notice performance loss while the system is stressed.
2. *Geographical scalability*: Users do not notice performance loss or delays while geographically far from the resources.

3. *Administrative scalability*: System can be easily managed while being administrated by many independent organizations

Each scalability dimension comes with its problems. Size scalability is tested with heavy loads or complicated operations. This is typically confronted when resources are located on single machines, which may create a bottleneck for the system's operations. Geographical scalability faces different problems. Communication over a long distance is inherently less reliable, thus creating more data loss. Another geographical scalability challenge arises with synchronous communication, where the client's request hinges on waiting for a response from the system. Being far from the system also means the messages take longer to travel, creating a higher latency. Administrative scalability has to deal with the problem of data privacy.

2.2.2 Scaling techniques

To solve the mentioned problems, one can apply *vertical* or *horizontal* scaling techniques. Vertical, also known as *scaling up*, contains methods such as increasing memory and upgrading CPU. In other words, scaling vertically improves already existing nodes. Scaling horizontally, or *scaling out*, is expanding by introducing new nodes to the system. [4, 6]

Another technique that does not quite fit into these two categories is using *asynchronous communication* instead of synchronous one. By communicating asynchronously, the client does not have to wait for a response from the server. This alleviates the problem of latency of geographical scaling to some extent.

Next in a list of possible techniques is *partitioning and distribution which involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system.* [4]

The last technique is *replication*. Replication can remedy the problem of data loss, node overload, and latency. [4] By replicating data or services on multiple nodes across the system, one can access any of them instead of overloading only one. Replicas can also be geographically dispersed to lower the latency of global systems. A special type of replication is *caching*, which creates copies based on demand and proximity. A particular example of caching being successfully used in distributed systems are *CDNs*.

2.3 Types of distributed systems

2.3.1 Clusters and grid computing

As time went on, multiple types of distributed systems emerged. The most simple one is *cluster computing*, which uses closely connected nodes that are similar to each other to gain performance and work as a high-performance unit of computation, so-called *cluster*. [4] Depicted in figure 2.2.

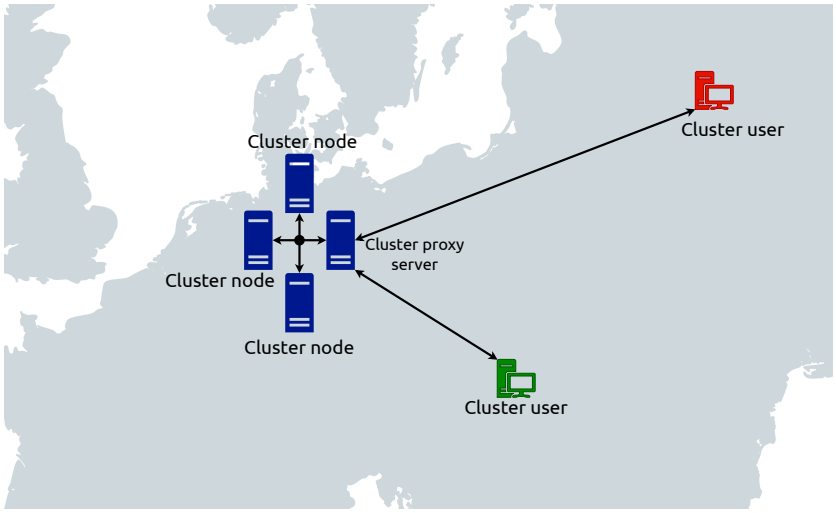


Figure 2.2 Example of a cluster

The second one is *grid computing*. Grid computing builds upon cluster computing and takes it further by creating a world-spanning interconnected network called *grid*. The grid comprises various processing units, such as personal computers, clusters, or mobile phones, illustrated in figure 2.3. Grids are used for high-performance computing, mainly in the research area. Grids can be connected to create an even bigger grid. An example of a grid is *Worldwide LHC computing grid* used to compute data captured at the Large Hadron Collider at Cern.

Multiple organizations must work simultaneously on a grid while keeping their data private. This is achieved by having various virtual organizations which cannot access each other's data. [7] Security is a big concern in grid computing because it is used heavily for research. Somebody could delete months of work or steal critical data if it was not handled.

2.3.2 Cloud computing

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 comprises five essential characteristics, three service models, and four deployment models. [8]

Virtualized machines are started on provisioned physical resources to simulate operating systems and offer better isolation from other cloud users. One concrete isolation technique called *sandboxing* will be discussed in section 4.3. Computing resources are then carefully monitored to charge users appropriately. [7]

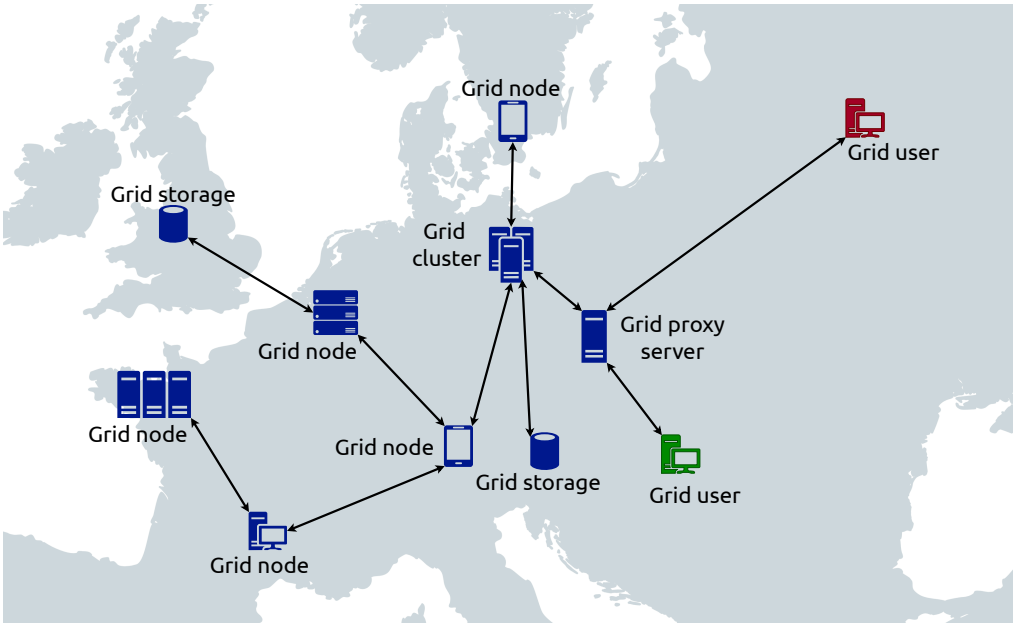


Figure 2.3 Example of a grid

The three layers of cloud computing mentioned in the definition are according to the abstraction level following:

1. Software as a Service (SaaS)
2. Platform as a Serice (PaaS)
3. Infrastructure as a Service (IaaS)

Software as a service offers a single software or system, such as video processing or an office suite. Platform as a service contains a development environment that allows the development of applications without dealing with low-level details. Examples of such platforms are *Google AppEngine*, or *Microsoft Azure*. The last type, infrastructure as a service, offers a wide variety of infrastructure elements such as virtual servers, physical servers, or storage. [9, 4]

In order for a service to represent a cloud computing model successfully, it has to have five essential characteristics [8, 10]:

1. *On-demand self-service* – Service has to be self-serviced. Cloud users expect simple on-demand computing without needing to deal with low-level elements constantly.
2. *Broad network access – Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin* [8]

3. *Resource pooling* – The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, according to consumer demand. The customer generally has no knowledge or control over the exact location of the provided resources.

4. *Rapid elasticity* – Capabilities can be elastically provisioned to scale rapidly, offering the illusion of infinite computing resources available on demand.

5. *Measured service* – Cloud systems automatically control and optimize resource use. Resource usage can be monitored, controlled, and reported.

According to cloud definition, there are four deployment models that reflect who the cloud is purposed for [8]:

1. *Private cloud* – Exclusive use for a single organization.

2. *Community cloud* – Exclusive use for a specific community.

3. *Public cloud* – The infrastructure is provisioned for use by the general public. It exists on the premises of the cloud provider.

4. *Hybrid cloud – Infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities.*

2.3.3 Edge computing

The network edge is the connection or interface between a device or network and the outside world. The edge is close to the devices it is communicating with and is the entry point to the network. [11]

It is a complementary extension of cloud computing, which considers geographical proximity. If the computation allows, it can be done directly on the edge nodes instead of processing data in large data centers. Nevertheless, the edge might still need large storage or computing power of cloud computing centers. [1] The difference between cloud and edge computing can be seen in figures 2.4 and 2.5, where in cloud computing, the user request always reaches the cloud datacenter, but when using the edge the requests to centralized datacenter are reduced as they are processed directly on the edge.

An important distinction to make is that the cloud is not a place. It is a logical entity distributed on multiple nodes over various locations. [12] The edge, on the other hand, is the place closest to the outside world as possible.

Because of its nature, edge computing solves many scaling problems described in section 2.2.1. It is a combination of *partitioning and distribution* and *replication* techniques from section 2.2.2. Having edge nodes take care of computation reduces pressure on the rest of the network and reduces the risk of network leakage by not sending sensitive data further. Moreover, being at the system's entry point is the ideal place for low-latency applications where data must be processed and sent back as fast as possible. [1]

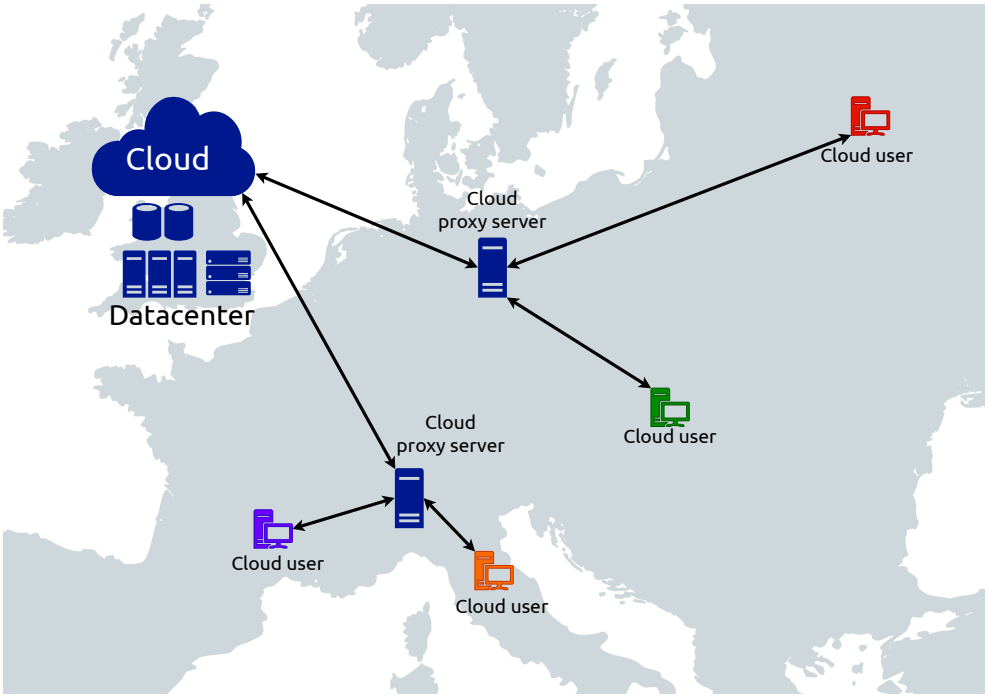


Figure 2.4 A simple example of a cloud infrastructure

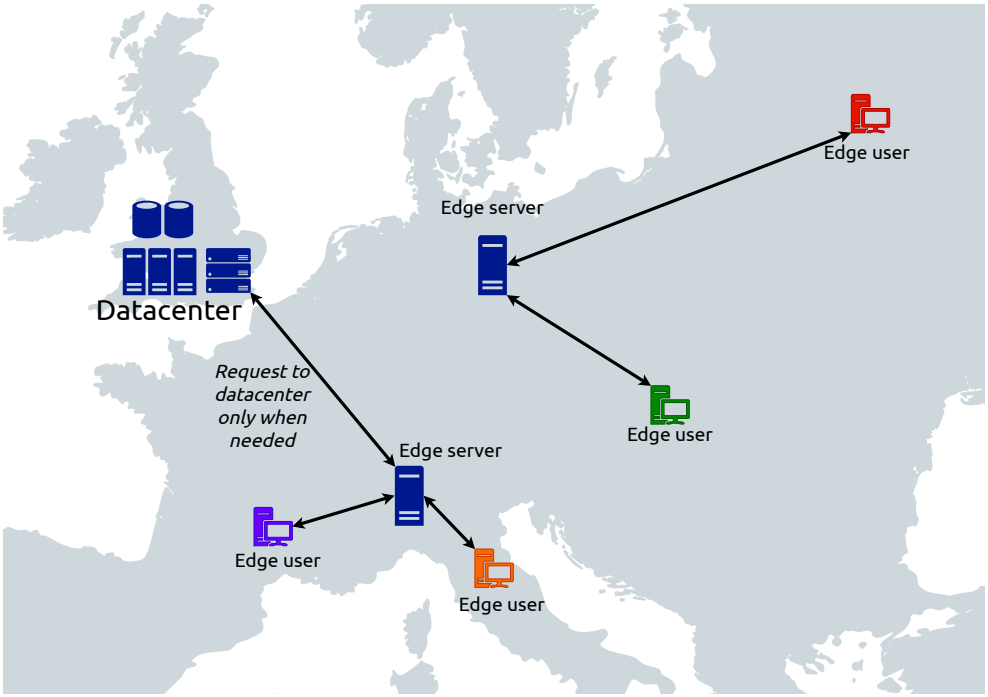


Figure 2.5 A simple example of an edge computing infrastructure

2.3.4 Reverse proxy

At a basic level, a proxy server is an intermediate piece of hardware/software that receives requests from clients and relays them to the backend origin servers. Typically, proxies are used to filter requests, log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compression). [6] The proxy acts as a ”router” routing requests into multiple other nodes chosen based on some arbitrary metric. A typical use case of a reverse proxy is *load balancing*, where the rerouting metric is the load on the nodes. [6]

2.3.5 Nginx

NGINX is open-source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. It started as a web server designed for maximum performance and stability. In addition to its HTTP server capabilities, NGINX can function as a proxy server for email (IMAP, POP3, and SMTP) and a reverse proxy and load balancer for HTTP, TCP, and UDP servers. [13]

Nginx has a modular architecture. It can be configured by writing a configuration file containing directives for the Nginx server or by writing a custom module that does practically any custom logic the author wants it to. Using these two approaches, Nginx server administrators can manipulate HTTP requests and responses as they see fit.

Chapter 3

WebAssembly (Wasm)

3.1 Core specification

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. [14]

It is an open standard byte code format developed by W3C Community Group. Designed to be fast, safe, hardware, language, and platform independent, and open so programs can simply interoperate with their host environment. It is designed to be easily processed by just-in-time or ahead-of-time compilation in a single pass. [14]

Wasm provides no access to its host environment. Every operation, such as I/O operations, has to be provided by the host, also known as *embedder*, and imported into the Wasm module. The embedder is responsible for controlling all functionalities provided through imports. [14] However, even if core Wasm does not specify these operations by itself, there are extensions of core Wasm format that do. An example of an extension is *Wasi* (WebAssembly System Interface), which provides a group of standard API specifications. Wasi is described in more detail in section 3.3.

The structure of Wasm is described in the form of *abstract syntax* and encoded in one of two formats:

- Binary format
- Text format

The binary format is a dense linear encoding of the abstract tree. Files containing binary format usually have the *.wasm* extension. This format is very close to machine code, making it less understandable for humans. Conversely, the text format is a human-readable representation of the abstract syntax. It

has a form of *S-expressions* and files containing it usually have *.wat* extension. An example of the text format is in listing 3.1.

```
(module
  (func (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add))
```

Code listing 3.1 A function taking two parameters and returning their sum in Wasm text format. [15]

3.1.1 Concepts

The computational model of WebAssembly is based on a stack machine. Code consists of sequences of instructions that are executed in order. Instructions manipulate values on an implicit operand stack. [14] Some instructions can generate *traps*, which abort execution and are reported to the *embedder* which may catch them. Another responsibility of the embedder is defining how the code is initiated. [14]

Code is organized into separate functions. Each function takes a sequence of values as parameters and returns a sequence of values as results. Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly. Functions may also declare mutable local variables that are usable as virtual registers. [14] A program can call a function either directly by function call instruction or indirectly via the index to a *table*. The table is *array of opaque values of a particular element type.* [14], which can be, during the writing of this thesis, only untyped function reference. [14]

A similar to the table is *linear memory. The linear memory is a contiguous, mutable array of raw bytes.* [14] It is initialized with a specified size and dynamically grows as the program stores data in it. The linear memory can load and store data at any address. When an instruction tries to access an address outside the memory bounds, it generates a trap.

Definitions for tables, linear memory, and functions are contained with initial values inside a Wasm binary, which takes the form of a *module*. These definitions can be exported, and definitions from other modules can be imported. *Module can also define a start function that is automatically executed.* [14] The start function initializes the module.

3.1.2 Semantic phases

The Wasm semantics is divided into three phases:

Decoding – The Wasm module is decoded into an internal representation.

Validation – The decoded module is checked for validity. This includes type checking of functions and instructions and validating operand stack use.

Execution – The module is executed. First, it is *instantiated* by initializing globals, tables, and memory and calling the start function if defined. The results of instantiation are *instances* of the module's exports. *A module instance is the dynamic representation of a module, complete with its own state and execution stack.* [14] After instantiation, the exported functions can be invoked on a module instance, returning its result.

3.2 Application binary interface (ABI)

Application binary interface (ABI) is a set of conventions for application access to the operating system functionality and other low-level services designed for portability of executable code between machines. [16] In other words, it is a convention dictating how binaries can work with other binaries to work correctly. It defines, for example, calling conventions that specify how arguments are passed and returned to/from a function.

3.3 Wasi

The WebAssembly System Interface (WASI) is a group of standard API specifications for software compiled to the W3C WebAssembly (Wasm) standard. [17] It takes inspiration from POSIX and CloudABI and provides Wasm code with a unified way of communicating with the outside world. [18] Wasi currently has two versions: *Preview 1*(0.1) and *Preview 2*(0.2). Preview 2 was released in January 2024, and Preview 3 is planned. [18]

Preview 1 and 2 differ in *interface definition language (IDL)*. Preview 1 uses *WITX IDL* composed into Wasm modules binaries. [19] This corresponds with the Wasm core specification, which states that Wasm binaries take the form of a module. [14] Meanwhile, preview 2 uses *WIT IDL* that is composed into *Wasm component binaries*. [19] *Wasm component is a wrapper around the core Wasm module, that specifies its imports and exports using WIT defined interfaces.* [20] It is used to enrich the core module's exposed functionality by adding more complex types, such as strings, lists, and records. By default, core modules do not have those and have to operate essentially only on integers and floating-point numbers. [19]

Wasi Preview 2 contains the following APIs:

- I/O – Providing I/O stream abstractions. [21]
- Clocks – Reading time and measuring elapsed time. [22]
- Random – Obtaining pseudo-random data suitable for cryptography. [23]
- Filesystem – Accessing host filesystems. This includes operations such as opening, reading, and writing. [24]
- Sockets – Adds TCP and UDP sockets and domain name lookup. [25]
- CLI – Provides command-line facilities such as command-line arguments, environment variables, and studio. It also contains APIs commonly available in CLI environments, such as filesystems and sockets. [26]
- HTTP – Sending and receiving HTTP requests and responses. [27]

3.4 Wasmtime

Wasmtime is a standalone runtime for WebAssembly, WASI, and the Component Model by the Bytecode Alliance. [28] It is a runtime intended for use outside the web, striving to be fast, secure, configurable, and compliant with the Wasm standard. [28]

It is written in *Rust* programming language on top of the *Cranelift* code generator used for JIT and AOT compilation. Its inherently sandboxed design (must import all functionality, more on this in section 4.3.4) is excellent for running untrusted code. As an additional precaution, once new features are implemented, the implementation undergoes 24/7 fuzzing. [28]

Wasmtime also offers additional capabilities like debugging. One can use *gdb* or *lldb* to live debug and step through the guest Wasm and the host simultaneously, or it can generate Wasm core dumps for crash analysis. [28]

Another capability is profiling. Wasm strives to be performant, so it is essential that Wasmtime offers code profiling. For the best results, one would ideally use hardware counters. However, because Wasmtime uses JIT, it requires providing hooks to other native profiling tools because the JIT-generated code can be located at arbitrary locations. Because of this, wasmtime currently offers support for only three tools: *perf*, *VTune*, and *samply*. Nevertheless, Wasmtime also provides a cross-platform profiler that works on every platform at the cost of preciseness. The native profilers can show almost everything from time spent in guest code to time spent in the kernel. Meanwhile, the cross-platform option can show only time spent in the guest code. [28]

3.5 Proxy-Wasm

Proxy-Wasm or *WebAssembly for proxies* is an emerging specification of ABI and conventions to use between L4/L7 proxies (and/or other host environments) and their extensions delivered as WebAssembly modules. [29] It was initially developed for the *Envoy*, but by being proxy agnostic, it can be used by different proxies. The ABI contains definitions for *callbacks*, *functions*, types, and serialization.

3.5.1 Callbacks and functions

A *callback* is a function exposed by the Wasm module. They are identified by entry points named as `proxy_on_<event>`. All are optional and will only be called if the Wasm module exposes them. On the other hand, the *function* is a function exposed by the host. All functions are required and return type `proxy_status_t` (details in section 3.5.2), which indicates the status of the call. Return values are written into the memory pointed by `return_<value>` parameters.[29]. Definitions are split into multiple categories, which can be found in appendix A.1.

3.5.2 Types

The proxy-wasm specifies the types used for callbacks and functions. One such type is already mentioned `proxy_status_t`, the return type of all functions. All types the ABI specifies are in appendix A.2:

3.5.3 Proxy-wasm implementation

For the purposes of this thesis, nginx terminology is used to describe the implementation, but principles should also apply to other proxies. The proxy-wasm specifies two components:

- Proxy (host, exposing functions)
- Wasm module (exposing callbacks)

To implement proxy-wasm, a new nginx module can be created containing all the necessary logic. The module has to contain Wasm runtime, such as Wasmtime, exposed function definitions, and general orchestration. Considering some wasm modules should be part of the proxy lifetime, the orchestration is responsible for instantiating them and calling correct callbacks. The runtime is responsible for module execution which can call the exposed functions.

This is enough for the proxy-wasm to work correctly. However, the ABI is too low-level to develop custom proxy logic comfortably. The best user experience would allow the logic to be written in any programming language.

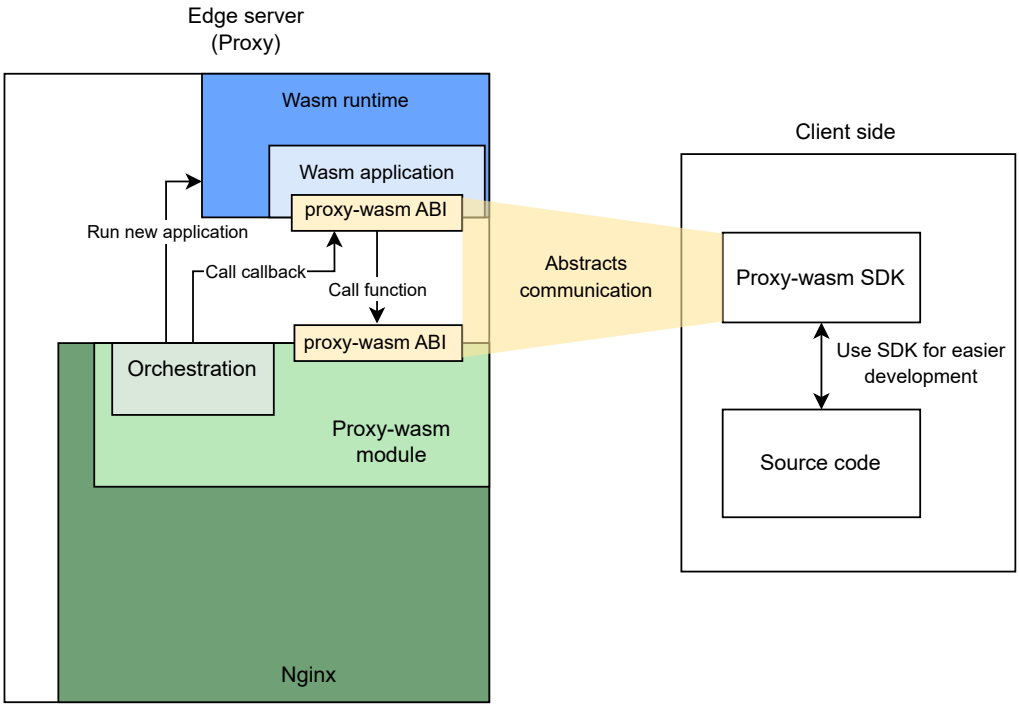


Figure 3.1 Implemented proxy-wasm with SDK

This can be achieved by providing an SDK for every supported language that would handle conforming to the ABI specification. Clients writing the logic can then use the SDK to develop modules in a comfortable, idiomatic way and do not have to worry about details. An overview of this approach can be seen in the figure 3.1.

Chapter 4
Security

4.1 Untrusted code

An untrusted code is a code one decided not to trust. The reasons for not trusting the code can be many, ranging from "The code is old, and nobody knows what it is doing." to the "Code was written by the company's competitor." However, generally speaking, an *untrusted code is a code that one is not certain what it will do.* In extreme cases, one could consider all code untrusted because there can always be a bug, which is not expected. Some amount of paranoia is healthy, but too much is harmful. In practice, what is considered untrusted code depends on the author's identity. Usually, people are divided into two sections: *trusted* and *untrusted*. The trusted group consists of all the people working directly on a product, and untrusted people are outsiders or users who can provide their code via some input to the product (figure 4.1). This thesis follows the same premise and considers the authors of the build pipeline as trusted and all pipeline users as untrusted.

As already mentioned above, with untrusted code, one cannot be sure what it will do. This is not always bad. Sometimes, it is intended because the author of the code decides what they want to do. Nevertheless, it becomes a huge problem when the code does something malicious. This includes things like *Denial of service (DoS)*, data breach, or complete destruction. Thus, untrusted code must be dealt with accordingly because it poses a significant risk for the system handling it. The following sections will describe a few attacks that potentially could affect the results of this thesis.

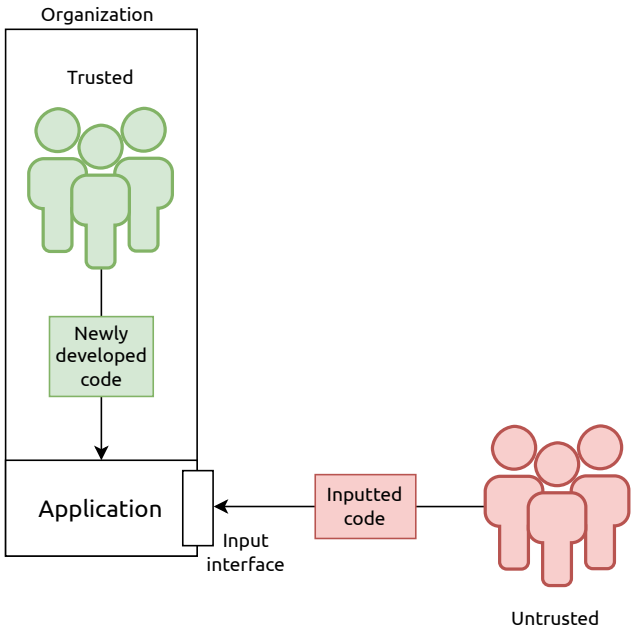


Figure 4.1 Trusted/untrusted groups

4.2 Potential risks

4.2.1 Compression bomb

Data compression is a coding technique that aims at reducing the number of bits required to represent a string by removing redundant data. When the compressed data can be restored to the exact original state, the compression is called *lossless*, otherwise *lossy*. [30] This is useful for storing and sending data, it saves space and makes data transportation faster. However, there are some caveats when decompressing data, which stem from three aspects:

- Decompression is CPU, memory, and disk-intensive task
- Decompression amplifies the amount of data
- Compressed data can often be pre-computed, so input can be sent really fast, while the decompressed has to spend a lot of resources to process it

The second point is the main reason for an attack called *compression bomb*. The concept is pretty simple. An attacker creates an archive containing a lot of redundant data, which is compressed to a relatively small archive compared to the amount of data it stores. This is possible because of the extensive redundancy. When the archive is decompressed, it overwhelms the target by consuming a lot of memory or CPU time. As an example, the classic archive bomb was a 42-kilobyte zip file archive that contained five nested layers of

compressed files whose total size amounted to 4.5 petabytes. [30] The compression bomb is an example of what can happen when untrusted code is poorly handled. One does not even need to execute the code in order to be dangerous.

4.2.2 Code execution

The code execution is not necessarily an attack but a prerequisite for many of them. When an untrusted code is allowed to execute, it can do almost anything without the proper restrictions. For this reason, the untrusted code has to be tightly controlled to deny anything malicious it could do. Examples of malicious behaviors are: Trying to allocate a lot of memory until the machine fails, creating infinite loops so the machine cannot process anything else, stealing sensitive data, and more. One could allow only certain APIs to be used by the running program so it cannot allocate memory and provide sufficient access rights so it cannot steal sensitive data, but that would still leave infinite loops and other exploits unsolved.

Another problem would arise when multiple instances of different untrusted code were executed at the same time. Ensuring they are sufficiently isolated and cannot access each other's data is essential. Moreover, if one instance of malicious code is successful and gains access to the machine, there has to be a way to mitigate the damage and not endanger other non-malicious instances running and potentially the whole infrastructure.

Because of all of this, there is a need for a more complex solution. The solution is a technique called *sandboxing*. It isolates the code in its own environment called a *sandbox* in which it can do anything without affecting the host system. However, naturally, the sandbox must somehow utilize its host resources to work. How to utilize the resources is implementation-specific for each sandboxing technique.

4.3 Sandboxing

Sandboxing, as briefly mentioned above, is a security technique responsible for providing a running code with its own environment in which it cannot harm the host. It is an encapsulation mechanism that imposes a security policy on software components. [31] It limits the access to system resources and allows the sandboxed program to be monitored. Sandboxes can be divided into multiple categories [32]:

- Jail – The operating system limits the resources a program running inside a jail has.
- Virtual machine – A program is running inside a virtual machine that has dedicated resources and limited access to the host.

- Rule-based execution – Rule-based execution like *SELinux* or *Apparmor* controls the behavior of a program by explicitly allowing which programs and files it can interact with and control their registry access.
- Os build-in – Build-in techniques like seccomp allows to restrict what system calls a program can call.
- Application sandboxes – An application offers sandboxing via a combination of mentioned and new techniques

Choosing a proper sandboxing technique is essential. Although sandboxes should be ideally perfect, they never are, and different implementations offer different benefits and drawbacks. Based on the selected sandbox, an attacker might try different techniques to exploit or escape it.

4.3.1 Dedicated hardware

Probably the most simple form of a sandbox is a separate dedicated hardware. It physically separates the dangerous operations from the rest of the trusted infrastructure and provides perfect isolation. Nevertheless, when attackers get hold of the hardware, they have access to everything inside. Thus, the hardware must be carefully monitored, and no suspicious communication outside should be allowed.

Although this solution works as a perfect sandbox because it cannot affect any system outside, it still has its problems. Considering that some use cases require thousands of sandboxes to be run simultaneously, this solution could be costly. To mitigate this, machines are often virtualized so that multiple virtual machines can operate on the same hardware.

4.3.2 Virtual machines (VM)

Virtualization provides a way of relaxing the foregoing constraints and increasing flexibility. When a system (or subsystem), e.g., a processor, memory, or I/O device, is virtualized, its interface and all resources visible through the interface are mapped onto the interface and resources of a real system implementing it. [33] *Virtual machines* are virtualized operating systems with virtualized hardware.

Virtual machines are beneficial in many ways. They offer partitioning of large software systems and provide isolation between guest VMs running on the same host hardware. Virtual machines can be provided with a limited amount of system resources, and they can emulate any architecture necessary for the running software. VMs are managed and controlled by a *virtual machine manager (VMM)*, also known as *hypervisor*. *The hypervisor presents to the guest operating systems a virtual operating platform and manages the execution of the guest operating systems. Multiple instances of a variety of operating*

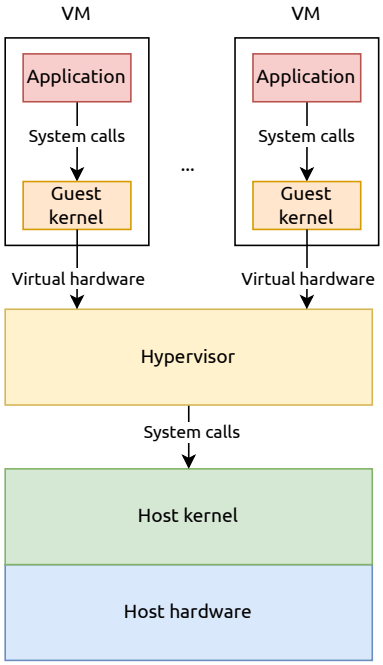


Figure 4.2 VM structure

systems may share the virtualized hardware resources. [34] Figure 4.2 shows structure of VMs.

Virtual machines are a great choice for sandboxing because they isolate the running application from the host kernel by every VM having its own kernel. Hypervisor is then the only way how to interact with the host system, which creates a performance bottleneck but also a great control. Hypervisors are also intensely scrutinized and tested by time so it is unlikely a malicious program will escape the VM confinments. A drawback of VMs is that they are very heavy weight. They need a whole kernel, and it takes a lot of time and space to spin up a new VM. This problem is solved in a different sandboxing method, a *containerization*, which will be described in section 4.3.3.

4.3.2.1 Kernel-based Virtual Machine (KVM)

KVM (Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor-specific module, kvm-intel.ko or kvm-amd.ko. Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc. [35]

KVM effectively transforms a Linux running the KVM into a hypervisor.

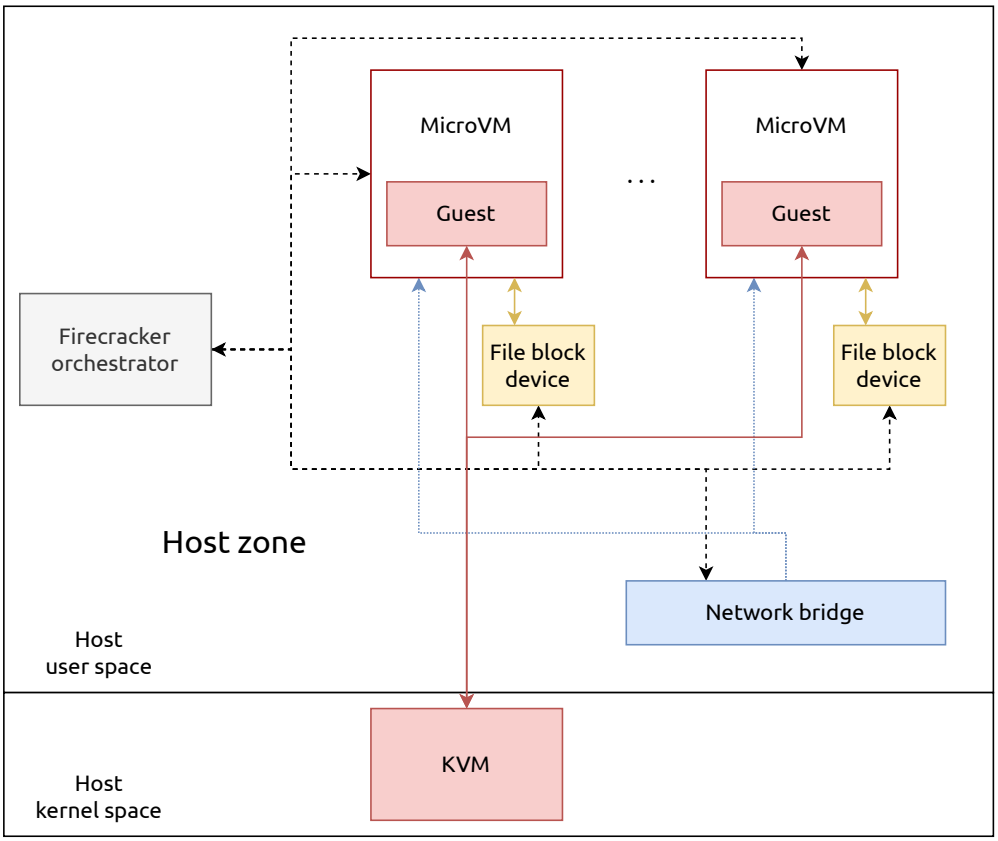


Figure 4.3 An example host running Firecracker microVMs (adapted from Firecracker documentation [38])

VMs are implemented as regular Linux processes with dedicated virtual hardware. By VMs being regular Linux processes, they are scheduled by a regular scheduler, which can be controlled to prioritize or deprioritize VMs. KVM also utilizes known security techniques like *SELinux* which defines security policies for files and processes. [36]

4.3.2.2 Firecracker

Firecracker is an open-source virtualization technology that is purpose-built for creating and managing secure, multi-tenant container and function-based services. It enables the creation of lightweight virtual machines, called microVMs, which solve the problem that VMs are normally too slow. It tries to match containerization efficiency without compromising the standard hypervisor's safety. By limiting the number of resources and kernel configuration, the microVM can be faster than other traditional VMs. [37]

Firecracker is a hypervisor developed by Amazon web services (AWS) for *AWS lambda* that builds on top of KVM. Figure 4.3 shows an example host

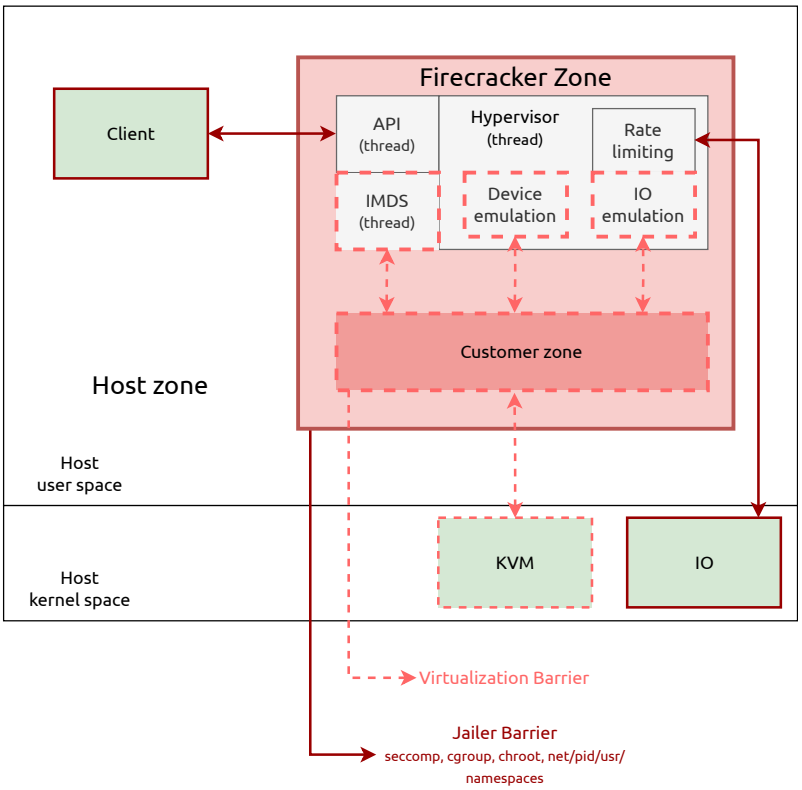


Figure 4.4 Firecracker threat containment (adapted from Firecracker documentation [38])

running Firecracker microVMs. Its main priorities are to be secure and performant. In order to be secure, it limits what devices guest VM might use and offers only the bare minimum. It can be executed inside of a companion program called Jailer, which provides an additional layer of security in case the virtualization is compromised. [37] *From a security perspective, all vCPU threads are considered to be running malicious code as soon as they have been started; these malicious threads need to be contained. Containment is achieved by nesting several trust zones, incrementing from least trusted or least safe (guest vCPU threads) to most trusted or safest (host). These trusted zones are separated by barriers that enforce aspects of Firecracker security. For example, all outbound network traffic data is copied by the Firecracker I/O thread from the emulated network interface to the backing host TAP device, and I/O rate limiting is applied at this point. These barriers are marked in the figure 4.4.* [38]

Firecracker can be configured via a RESTful API to limit the number of resources and start new VMs. It provides a metadata service for communication between the host and the VM, which can be configured with the API. [37] It is generally considered to be safe thanks to its relatively small code base

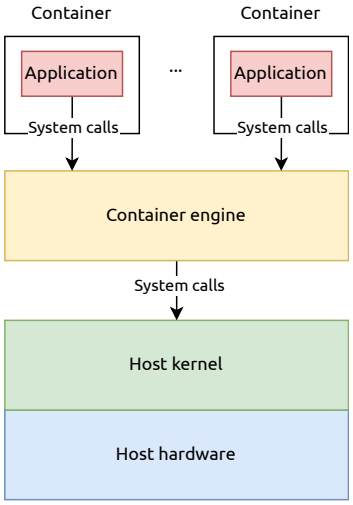


Figure 4.5 Containerization structure

written in rust. Nevertheless, while being considered safe, there are still some vulnerabilities exploited by microarchitectural attacks, like Medusa variants or Spectre, which the firecracker cannot mitigate. [39]

4.3.3 Containerization

Containerization is a type of virtualization that is more lightweight than creating full-blown virtual machines. A unit of virtualization is a *Container*, which is an alternative to the virtual machine. Containers are standard Linux processes managed by a *container runtime*, which is an alternative to the VM's hypervisor. Containers are defined by a *container image* or a *container repository*. There are several standards for container images, such as Docker, Appc, LXD, and Open Container Initiative (OCI).

The most significant difference from VMs is that containers share the host kernel and thus can be more efficient in utilizing and sharing resources. Containers are isolated by the use of *cgroups*, *SELinux*, or *AppArmor*. [40] A containerization structure can be seen in figure 4.5 and can be compared to the figure 4.2 to spot the missing layer in containerization.

Cgroups or, in their whole name, control groups, allow the allocation of resources such as CPU time, system memory, network bandwidth, or combinations of all among user-defined groups of tasks (processes) running on a system. Cgroups can be monitored, configured, and deny or allow access to certain resources. [41]

Nevertheless, cgroups are not enough to isolate everything. Containers must also utilize *kernel namespaces* to enable each container to have its own mount points, network interfaces, user identifiers, process identifiers, etc. [40]

All this functionality is provided by a *container engine*. A container engine

is responsible for downloading container images, running containers based on said images, preparing container mount points, managing metadata, and calling container runtime. There are many container engines like docker, podman, or LXD, with other engines being part of cloud platforms provided through PaaS. [40]

Containers are used for many use cases. They can be run in privileged and non-privileged modes. Privileged mode makes users in the container have the same privileges as on the host machine. This is a problem when the user is root and could potentially harm the host. Thus, the privileged mode should be avoided and only used in special cases. [42] On the other hand, there is a non-privileged mode, which gives the user inside the container no privileges on the host other than specifically defined ones.

Based on use cases, containers can be divided into multiple categories [40]:

- Application containers – Probably the most popular form of containers. Containers contain code that developers develop. Containers also can contain databases and other necessary applications.
- Operating system containers – They operate similarly to the full virtual machine. Nevertheless, the kernel is still shared with the host.
- Super privileged containers – Specialized containers used for administrating container infrastructure.
- Sandbox containers – Container used as a sandbox to run potentially malicious software isolated from the host.

This thesis is interested in the last category, sandbox containers. Containers certainly have capabilities for their use as sandboxes. Cgroups, namespaces, and SELinux are potent tools for isolating malicious intentions. They can be leveraged to offer a safe location to execute untrusted code. But only with additional effort.

While being its most significant asset in terms of performance, the shared kernel of containers is a big security problem. Containers are more closely connected and can access data more efficiently. Container sandboxing is also more recent than the traditional VM approach and is generally discouraged. Because of this, it is not as battle-tested as a VM hypervisor. However, some technologies try to mitigate the shared kernel vulnerabilities by implementing an alternative to the missing hypervisor layer to a container runtime. One such technology is called *gVisor*, which will be discussed in future section 4.3.3.3.

4.3.3.1 Docker

Docker is the most well-known container engine out there. Docker has a two-part client-server architecture. First, the docker daemon (`dockerd`). The daemon acts like a server and listens for API requests managing docker objects like images and containers. The second part is the docker client, which interacts with the user and sends commands to `dockerd`.

Docker is written in the Go programming languages and uses namespaces and groups, as described in the previous section. [43] A big concern for docker was that it used to run `dockerd` only as root, which allowed somebody with access to creating containers to do almost anything. *Specifically, Docker allows to share a directory between the Docker host and a guest container, and it allows to do so without limiting the access rights of the container. This means somebody can start a container where the `/host` directory is the `/` directory on the host.* [44] Nowadays, docker offers a *rootles mode*. However, the rootles mode has limitations on what storage drivers it can use; Cgroup is supported only when running with a group v2 and systemd, and AppArmor and other features are not supported. [45]

4.3.3.2 Podman

Podman is a daemonless, open source, Linux native tool designed to make it easy to find, run, build, share, and deploy applications using Open Containers Initiative (OCI) Containers and Container Images. It uses OCI-compatible container runtime to create running containers. It also tries to keep its command line interface (CLI) similar to the docker interface so people who come from docker do not have to learn new commands. [46] By being daemonless, it allows running containers without root access, making it safer to use.

4.3.3.3 gVisor

gVisor is an open-source Linux-compatible sandbox that runs anywhere existing container tooling does. It enables cloud-native container security and portability. [47] In more depth, it is an application kernel, written in Go programming language, that implements a substantial portion of the Linux system call interface. It includes OCI-compliant container runtime, which can be used with existing container engines like docker or podman. [48]

It is an alternative to a VM hypervisor and rule-based executions such as SELinux. The difference between their architecture can be seen in figure 4.6. The gVisor acts as a guest kernel that intercepts containerized application system calls without translation through virtualized hardware. [48]

The gVisor has two distinct parts: *Sentry* and *Gofer*. The Sentry is a kernel that runs containers and responds to system calls made by the application. It is started in a seccomp container without access to file system resources which have to be mediated through Gofer. The Gofer is used to provide file

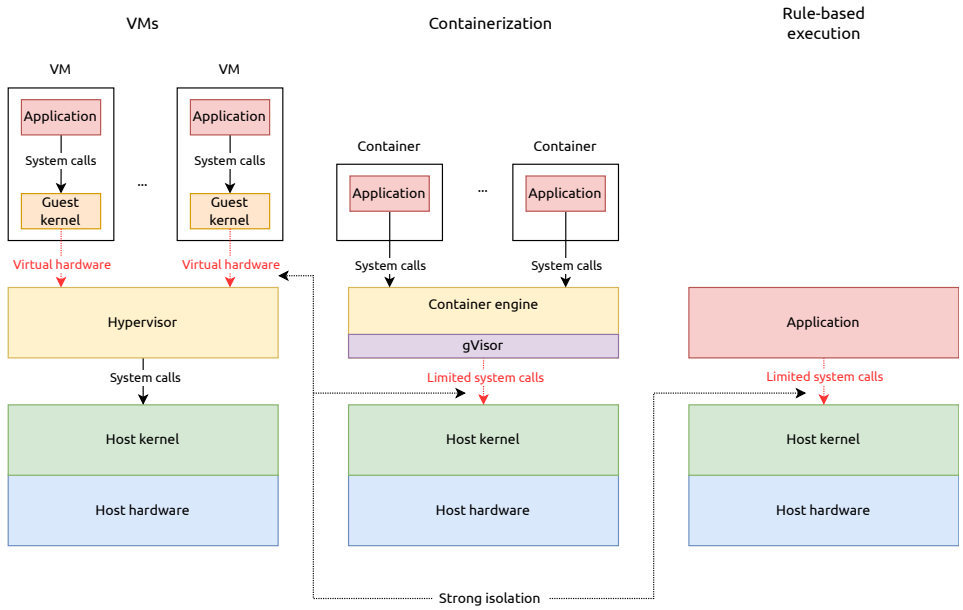


Figure 4.6 Differences between VMs, containerization and ruled-based execution

system access to the containers. Both are instantiated for each container and communicate together via the 9P protocol. [48] The two parts are pictured on figure 4.7

4.3.4 Wasm Runtime

Another interesting sandboxing method is to develop a runtime where all the computation runs. This is an efficient method when one has access to a source code that must be run or when an application does not have the necessary permissions to start a VM or a container. A use case from the real world can be an internet browser that has to handle the execution of untrusted JavaScript code. The execution of the javascript code can be made safer when running it in a runtime, which acts like a sandbox. An example is a *V8*, javascript runtime used in Google Chrome. [49]

A Wasm has a similar story. It was designed to be run in a browser with near native speed. As a result, its design is inherently sandboxed. The core Wasm specification has multiple features that create this sandbox environment [28]:

1. The call stack is inaccessible – Return addresses from calls and spilled registers are stored in memory only the implementation can access. This mitigates stack-smashing attacks that target return addresses.

2. Pointers are offset to linear memory – Pointers in source languages are compiled to offsets into linear memory. This hides implementation details

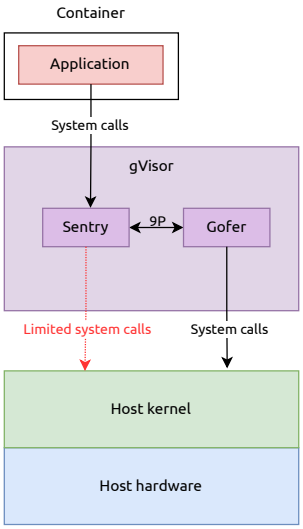


Figure 4.7 gVisor components (adapted from gVisor documentation [48])

like virtual addresses from applications. As a bonus, all accesses within linear memory are checked to stay in bounds.

3. Checked jump destinations – All destinations of direct and indirect branches and calls are checked, so it is not possible to jump into the middle or outside of a function.

4. Interactions with the host are through imports and exports – All interactions with the host have to be made explicit by either importing a function from the host or exporting from the Wasm module.

5. No undefined behavior – Wasm spec does not permit undefined behavior. Although there sometimes might be multiple possible behaviors, they are always defined.

Together with these features, different Wasm runtimes can offer even more protection. For instance, the Wasmtime runtime adds a 2GB guard region to a linear memory, and where it can, it zeroes out the memory after a Wasm instance is finished. Moreover, it adds explicit checks to determine whether a Wasm function should be considered to stack overflow. [28]

Part II

Implementation

Chapter 5
Build pipeline design

5.1 Requirements

The result of the design and implementation has to be a complete pipeline, which takes source code as its input and returns Wasm binary on its output. The design has to consider the security and scalability of the solution. The pipeline results must be observable and communicated to the outside. The following sections will describe the functional and non-functional requirements of the work in greater detail.

5.1.1 Functional

FR 1 – The user can send application source code written in Rust with its metadata to a REST API. The source code has to be compiled and stored accordingly.

FR 2 – The source code can be versioned. Every successfully built application has to have a version. A user has to be able to create a new version of the application via REST API.

FR 3 – User can delete an application version or edit its metadata via REST API.

FR 4 – User can observe the success or failure of the running build via REST API.

FR 5 – User can observe information about running builds and completed builds via REST API.

FR 6 – User can observe information about uploaded application versions via REST API.

5.1.2 Non-functional

NFR 1 – Support only applications written in Rust, but design the solution in such a way that it will be possible to add other languages with a small effort.

NFR 2 –Design solution in such a way that all storage technology can be easily switched to another technology.

NFR 3 – Compilation has to be secure. Design the solution in such a way that compilation will not endanger anything running on the infrastructure.

NFR 4 – Use sandboxing techniques to secure compilation.

NFR 5 – The sandboxing has to be designed in such a way that in case it is deemed not secure enough, it will be easily replaceable for another solution.

NFR 6 – The solution has to store source code, compiled binary, and logs on the object storage platform.

NFR 7 – Solution should work asynchronously to not block user requests.

5.2 Architecture

Considering functional and non-functional requirements, a few design choices were considered. The architecture will be composed of three components based on the operations needed to be done:

Client facing API – According to functional requirements, the client has to be able to work with the pipeline via REST API. This component is responsible for defining the API and, based on user requests, making sure the source code is compiled and stored and that the build results are observable.

Compilation – The compilation component has to do one job, compile source code safely, and return Wasm binary together with build logs in case something goes wrong.

Persistent storage – The persistent storage component must ensure all data is stored correctly. This includes storage of source code, logs, and Wasm binary in the object storage platform and storage of metadata about the build.

These three parts roughly correspond with the three-layer architecture, which has a presentation layer (Client-facing API), business logic (Compilation), and persistent storage. However, future sections will study why such architecture is not ideal in more detail. Nevertheless, the pipeline will need to represent the three elements somehow.

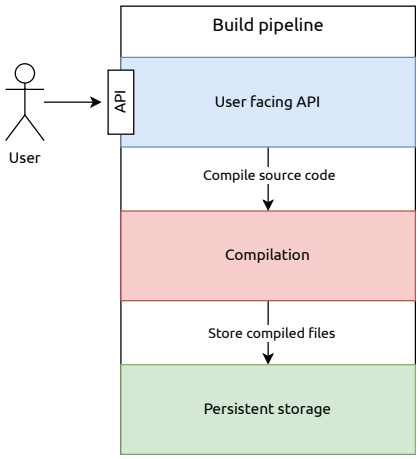


Figure 5.1 Single component architecture

5.2.1 Single component

The most simple architecture is putting everything into a single component. The component is then supposed to handle all logic, no matter its purpose. It provides REST API to its users, compiles applications, and, in the end, saves data to persistent storage. This directly corresponds to the three-layered architecture. Its diagram can be seen in figure 5.1. Although easy to implement, this component does not provide sufficient isolation of the compilation process, which could endanger other processes running on the same machine and potentially get access to a database. That is a big problem in contradiction with NFR 3. It also suffers from another issue: scaling.

The compilation is inherently a more complex operation than processing API requests or saving data. This means that when faced with a large number of requests, the compilation will become a bottleneck, the API will not be able to process more requests, and data will not be saved at the pace it potentially could be. This could be dealt with by spawning more compilation machines. However, having everything in a single component introduces unnecessary overhead because API and saving to persistent storage would also be duplicated, which it does not have to.

5.2.2 Multiple components

Based on the previous reasoning about the single-component solution, it is clear that the compilation component has to be isolated on its own to achieve better security and scaling potential. A similar reasoning could also be applied to the persistent storage component. This creates a design that is close to microservice architecture, where every functionality is a service. A diagram of this approach can be seen in figure 5.2.

Having multiple components is good, it enables scaling by spawning more

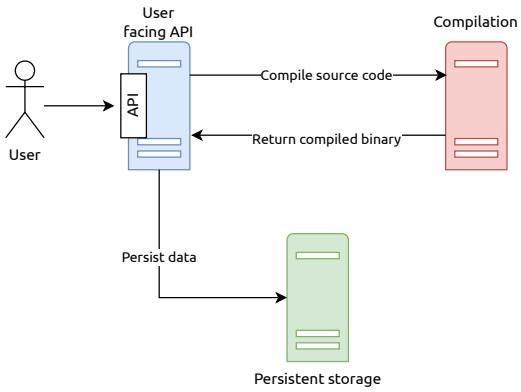


Figure 5.2 Every component separated into its own service

of them, and with the use of load balancing it enables utilizing as much as possible. However, there is a slight oversight in the use case of the build pipeline. Storage needs to save source code, binary, and potentially logs, all these files combined could have based on testing on real user data around 5MB, which is a considerable amount of data that would be sent between multiple components.

This issue could be mitigated by using a slightly different service topology. The initial design suggested sending all files that came from the compilation back to the caller which would then manage saving of the data. However, the compilation could send data directly to storage. This would help with the problem. Moreover, considering the compilation has to be asynchronous, the compilation component cannot return its results to the caller immediately anyway.

This last note signals that compilation has to send data to some other endpoint other than the caller, which could be the storage. Nevertheless, the results of the compilation have to be preprocessed before they are saved to the storage because the compilation component should have as minimal amount of responsibility as possible to limit potentially dangerous operations. That creates space for another component that would be responsible for evaluating compilation results. The component will be called *evaluator*. The new architecture with evaluator would look like this 5.3.

5.2.3 Somewhere in between

Although creating a custom microservice for every functionality might be beneficial, it introduces additional complexity to the system. This complexity can be justified by scaling potential and resilience, but sometimes, when these two values are not the system's primary concern, it can be better to merge some services together and instead build a singular system with clearly separated boundaries.

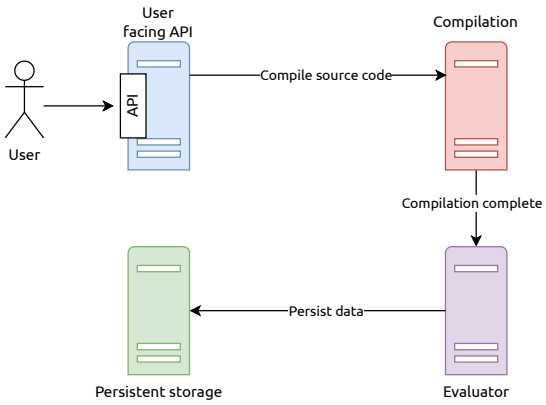


Figure 5.3 Multiple components with evaluator

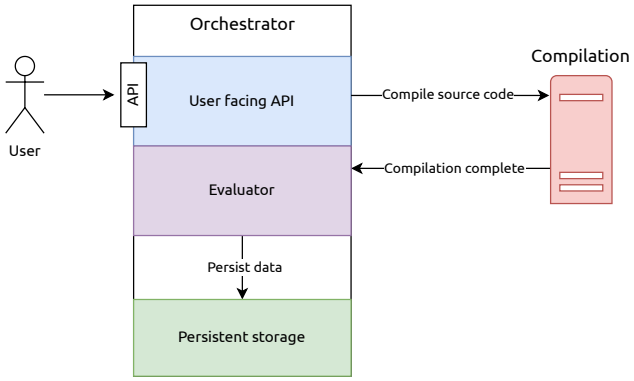


Figure 5.4 Final design with two separate services

This thesis utilizes the latter approach, while it would not be wrong to create a service for every functionality some of them are so small that they could be integrated into another service. For these reasons, the final chosen design has only two components: The isolated compilation service and customer-facing API with storage and evaluator. The final design is in figure 5.4.

Each component will be described in the following sections. The *Orchestration (5.3)* will design customer-facing API with the evaluator. The *Compilation (5.4)* shows how a compilation component with a compilation sandbox could work. Furthermore, at last, the *Storage (5.5)* describes what storage elements are needed and how other components should work with it.

5.2.4 Comunication between components

When multiple components are involved, they have to communicate in some way. In a world of microservices, there are two prevalent methods of communication between services: REST API and using messages with a message

queue. [50] Both offer their pros and cons.

The REST API is a simple interface used primarily for lightweight, maintainable, and scalable services. They are not dependent on any particular protocol, but the most used is HTTP. It is designed around stateless communication, so all information has to be included in the request. Additionally, the request also expects a corresponding response. [50] In the case of the build pipeline this would mean the orchestration component would send a request to the compiler and the compiler would have to respond immediately, the response would however be only information that the source code was accepted and not result of compilation, because compilation takes long time. Although nothing is wrong with this approach, it is unidiomatic, and data could be lost when the compilation component is overwhelmed with too many requests. Similar would be the communication on the other end when the compiler completes the build and wants to send built data to the evaluator. However, this second case is more in terms with the request/response way of REST APIs.

The second approach using messages and message queues utilizes *publish/subscribe* pattern where one service can publish a message to a message queue, and other services can subscribe to the message queue. This eliminates the problem when the subscriber (compilation component) is unavailable because the queue would store the compilation request until some subscriber is again available. It would also offer a way how to load balance between multiple compilation workers. The downside of this approach is the introduction of a new technology that has to be maintained and additional complexity.

Even though it seems at first like only one of these approaches can be used. There is nothing stopping someone who would like to use a combination of both. For example, a message queue can be used to send compilation requests to the compilator, and REST API can be used to send built data to the evaluator. This approach would make perfect sense and would be in terms with both styles. Ultimately, the thesis chose only the REST API approach because of the added complexity and technology that would have to be maintained.

5.3 Orchestration

To describe and design the workflow of the pipeline as best as possible, the thesis first looks into the orchestration component, which is the entry point of the pipeline. For now, it assumes the orchestration can persist with any necessary data. How the orchestration saves data will be explained in the storage section.

The orchestration component is responsible for orchestrating the build pipeline. Based on input via API, it manages what is saved to the database and what is sent for compilation. It is divided into two distinct parts:

Customer facing API – The public entry point to the pipeline that users of the pipeline can use. It contains all operations necessary to manage applications and submit source code for build.

Evaluator – The evaluator is a private endpoint that can be utilized by the compilation component when it completes its work.

5.3.1 Customer facing API

The API has to contain all functionality needed by functional requirements. This means there has to be an endpoint for the following operations:

- Build a new application (FR 1)
- Create a new version of an application (FR 2)
- Edit information about an application (FR 3)
- Delete version of an application (FR 3)
- Show status of version build (FR 4)
- Show build information (FR 5)
- Show application version information (FR 6)

After looking into the required functions, a few observations can be made. Building a new application could be considered as building version 0 or 1 of a new application. Another one is that the status of a version build could be contained in the build information. This squashes FR 1 with FR 2 into one function and FR 4 with FR 5 into another one. Some functions will also be added to have some way of grouping and retrieving application versions. These functions will create, update, and delete empty applications without any versions. All operations will be described in the following sections with a detailed definition of the API in appendix B.

5.3.1.1 Application operations

The application is a way how to group versions. An application has to contain three essential fields: application owner, application name, and description. Users of the pipeline should be able to create, update, and delete applications. The update operation should be possible only on name and description, the owner should not be able to change.

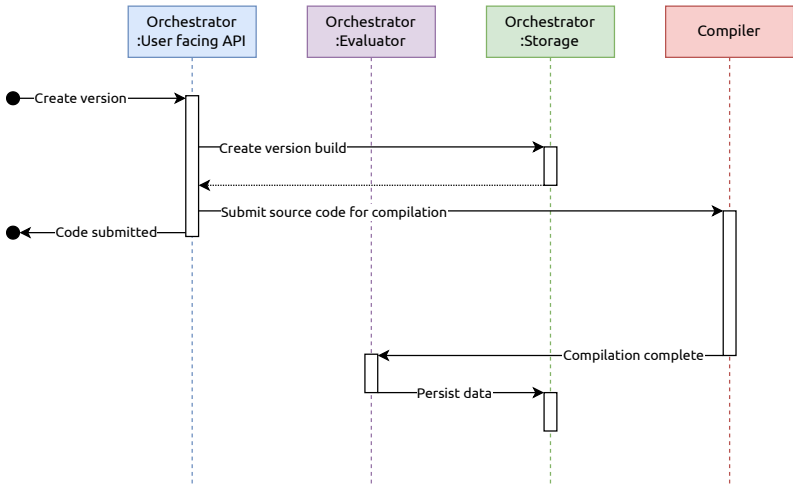


Figure 5.5 Sequence diagram of version creation

5.3.1.2 Version operations

Version operations are a bit different from regular CRUD. However, all CRUD operations should be possible to execute only on a specific subset of version fields. The Version must contain the following fields: application it belongs to, semantic version it represents, description, application configuration, source code, binary, and logs.

Create – Creation has to create a new version build and then submit source code for compilation to the compilation service. It has to be asynchronous so the user does not wait until the compilation is complete. A sequence diagram documenting how the creation should behave is in figure 5.5.

Read – Read should return all information about the version except source code, binary, and logs. The user does not need to see those, so they can be omitted. An exception is when the version build fails. Then, the logs should be visible. However, this functionality does not belong to the version but to the version build.

Update – The user should be able to update all version fields mentioned except the application it belongs to, source code, binary, and logs.

Delete – Deletion will remove the application version with all data. This includes all information about the application build and files stored in object storage.

5.3.1.3 Version build operations

Version build is an internally managed entity, so it must only be observable via read operation. Users should be able to retrieve information about builds

of specific applications and look at their logs. A new version build should be created when the source code is submitted to the compiler component so it can be tracked. The result should be updated by the evaluator component described below based on the build result.

5.3.2 Evaluator

The evaluator is responsible for accepting successful and failed builds from the compilation component. Its job is to store data based on the build status. When the build finishes successfully, it should be marked as successful in persistent storage, and binary and logs should be saved with it. A similar case is when the build fails, the build has to be marked as failed and only logs should be persisted so a user can see what happened.

This can be achieved by having two private endpoints which are available only to the compiler component. They cannot be public because someone could easily post the wrong data to them. The first endpoint is for a successful build, and the second is for a failed one. Definitions for both are in appendix C.

5.4 Compilation

The compilation is the main element of this thesis. It is supposed to be safe and extensible so future changes can be implemented easily. As already stated, the compilation is separated into its own isolated component, so if anything fails, it does not affect other processes and databases. The compiler component communicates with the outside world via REST APIs. It exposes one endpoint to the orchestration component, which is used to submit source code for build. At the end of the build, it calls the evaluator API to signal that the build finished either successfully or with a failure. The compilation can be altogether separated into six stages:

1. Accept build request.
2. Prepare sandbox.
3. Supply data to the sandbox.
4. Build the code in the sandbox.
5. Retrieve data from the sandbox.
6. Send build result to the evaluator.

Based on this workflow, a compilation could be further separated into three components: *Request handler*(1,6), *Build service* (2), *Build pipeline*(3-5). The request handler is responsible for the input and output of the compilation. Its job is to accept requests, parse data from them, call the build service, which

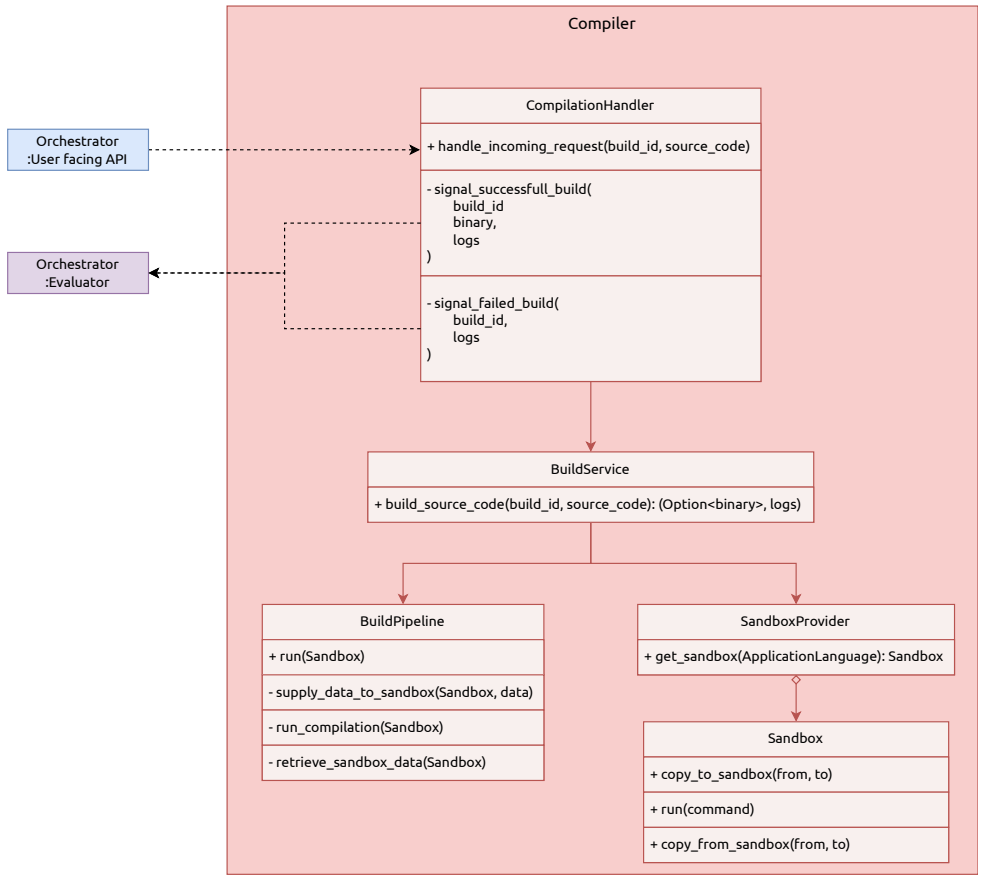


Figure 5.6 Architecture of compiler component

does the compilations, and send the results of the build when the compilation ends. Build service is a reusable service used to build source code. If, for some reason, the input and output communication method of the whole compilation component would change, the build service should remain the same. It prepares a sandbox in which everything potentially dangerous runs and starts a build pipeline with that sandbox. A build pipeline is the definition of all the steps that have to be taken to build a Wasm binary. A diagram of the structure is in figure 5.6.

5.4.1 Compilation sandbox

The sandbox is the core of the whole compilation process. It has to make sure nothing malicious happens during the build process, and it has to be replaceable if a chosen technology is shown to be insufficient. Because of the latter part, the solution is divided into a *Sandbox* and *Sandbox provider*.

5.4.1.1 Sandbox

Sandbox is the unit of isolation. It must have a clearly defined interface for working with it.

Sandbox's primary function is to execute a command safely. This leads to the first method of the sandbox `run(command: String)`. The run command is a method that accepts a string, which defines the command that is supposed to be executed inside the sandbox.

The command is a regular string because the sandbox's concrete implementation must be replaceable and every sandbox could support different functionality. The string is so generic that it does not matter what sandbox is used, and concrete implementation can handle it. Another solution would be to create a common command abstraction that concrete sandboxes could implement. However, that would be too much work for a relatively small benefit.

Sandbox also has to be able to receive data. This can be achieved by having a second method `copy_to_sandbox(from: String, to: String)`. This method serves as an input to the sandbox. Parameter `from` is a string defining input file from the host, and `to` is the location inside the sandbox under which the file should be located. Concrete sandboxes have to ensure that this method's implementation is correct and does not provide access to the host via this file.

The third and final operation that the sandbox should be able to do is to copy files from inside the sandbox to the host. This is because after the source code is compiled inside the sandbox, the binary has to be brought out to the host. This method is `copy_from_sandbox(from: String, to: String)`. The `from` parameter is a string defining location inside the sandbox, which should be copied into the host. The second parameter, `to`, is a string defining the location in the host where the file will be copied.

5.4.1.2 Sandbox provider

The sandbox provider is the component responsible for creating, maintaining, and providing sandboxes to the build service. It was designed as a way for the build service to retrieve a sandbox without knowing concrete sandbox implementation beforehand.

The second responsibility is ensuring the sandboxes are prepared when a build request is issued. This can make the build process faster by eliminating the startup time of the sandbox. This could be particularly useful when using VM sandboxes because of their well-known slow boot times.

Sandboxes can be requested by two criteria: Sandbox technology and sandbox purpose. It could be possible to have two sandboxing technologies used simultaneously. Nevertheless, for simplicity, the thesis chose to select only one. The second criterion, the sandbox purpose, corresponds to a compiled language. To comply with NFR 1, the thesis has to assume that it works with

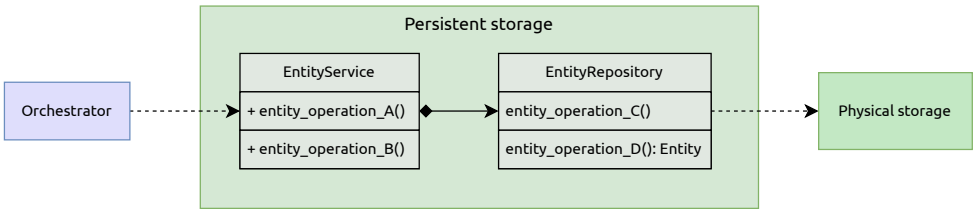


Figure 5.7 Architecture of storage

multiple languages and that every language uses a different toolchain for compilation. The solution could put all possible technologies inside one sandbox, but it would be too cumbersome. The sandbox definition would take more space and instantiation more time. As a result, the thesis chose to separate the sandboxes for each language.

The sandbox provider should expose one method responsible for retrieving sandboxes: `provide_sandbox(language: ApplicationLanguage)`. This method takes one argument that specifies which language the sandbox targets. Based on the language, the provider selects a prepared sandbox and returns it to the caller. As a side-effect, it issues the creation of a new sandbox to keep a buffer of prepared sandboxes at hand.

5.5 Storage

Permanent storage is required so applications, their versions, and builds can be tracked across requests and retrieved later, in compliance with FR 4, FR 5, and FR 6. To also comply with NFR 2, the chosen storage must be easily replaceable by another technology. This requires a level of abstraction over physical storage. This can be achieved by utilizing the *repository pattern*, designed for precisely this reason.

A repository contains all the logic one can execute over an entity. This includes CRUD and other operations. Creating an abstract contract (the repository interface) that concrete implementations must uphold will allow replacing one implementation with another. Every entity will have its repository, which will define its operations.

Repositories will then be used inside *Storage services*. These services will act as an entry point into the storage without creating a separate microservice. Services will offer a public interface that clearly defines what operations can be done over the storage. They will be used by the *orchestrator component* to persist build information and save data after compilation is done. Figure 5.7 shows a diagram of this structure.

While repositories and services create a *horizontal layering* of the storage architecture, a *vertical layering* can also be described that will outline storage features. The thesis will define two vertical slices: Metadata and Object storage. Figure 5.8 outlines the layering. The following sections will closely

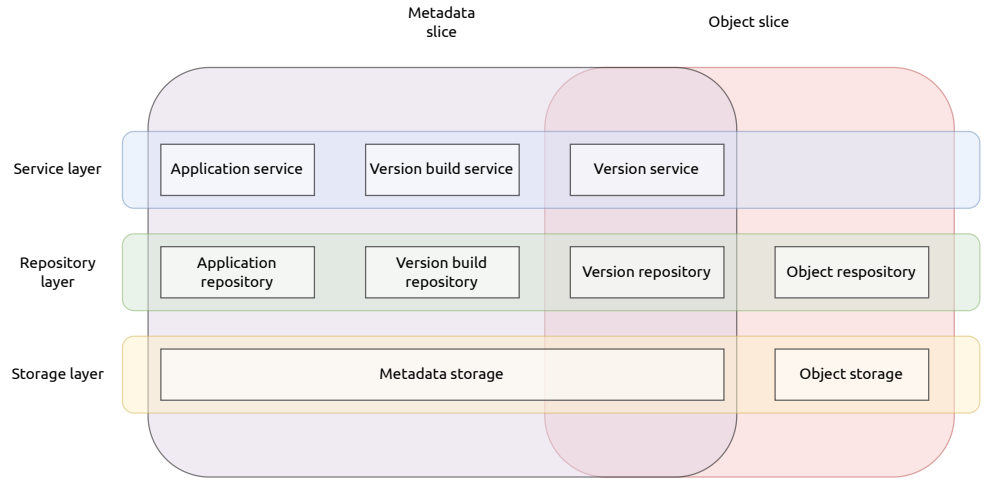


Figure 5.8 Layering of storage architecture

examine them and describe what entities, repositories, and services they work with.

5.5.1 Metadata

The metadata part of the storage stores information about applications, versions, and version builds, for instance, names, dates of creation, and keys to object storage.

Altogether, the metadata segment will contain three entities, one for each mentioned part. Entities are shown in figure 5.9. Services for those repositories/entities will have the following functionality:

Application : Create, Read, Update, Delete, ReadAll

Application version : Create, Read, Update, Delete, ReadAll

Version build : ReadAll,

5.5.2 Object storage

The object storage is, according to NFR 6, used to store all files generated by compilation. Keys to the files will be stored inside the version and version build entity in the metadata database. An alternative to storing keys in the metadata database would be storing all information in some special file in object storage, but this solution has bad ergonomics and would be difficult to work with.

The object storage cannot be accessible by itself. This means there will not be a public service that someone could use to retrieve these files. Object

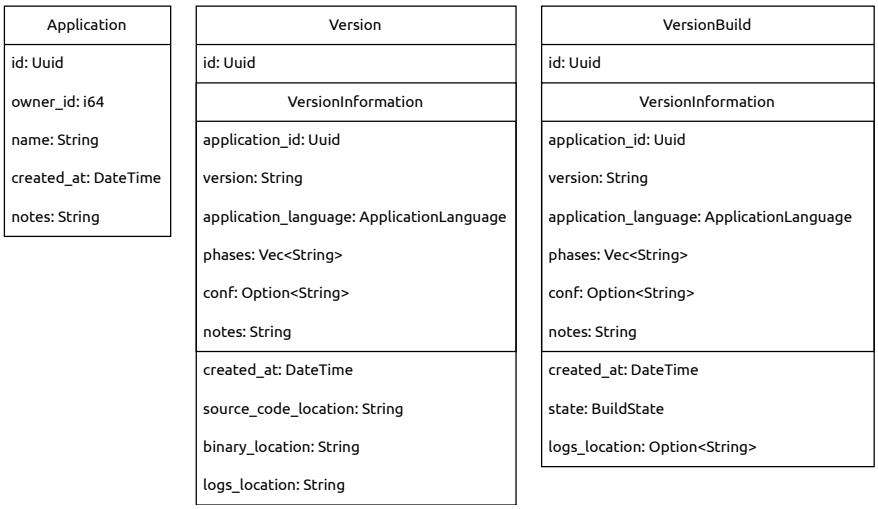


Figure 5.9 Metadata entities

storage will be only accessible via a repository, which will be used by some other service maintaining versions and version builds. This arrangement is shown in figure 5.8.

Object storage will have only one entity, the `ApplicationData` entity, which will have only two fields: `id` and `data`.

Chapter 6

Build pipeline implementation

6.1 Related work

This thesis builds upon existing CDN edge computing infrastructure and adds an important part so applications can be compiled and executed on it. The infrastructure can be divided into four parts:

1. Nginx
2. SDK
3. Build pipeline
4. Application distribution

Nginx is the technology used to power edge servers. The nginx contains a proxy-wasm application that implements the necessary pieces described in section 3.5.3. It uses Wasmtime as its Wasm runtime of choice, controlled by bindings written in C language. Wasm modules are instantiated and called from a Lua enabled by OpenResty [51]. Applications can be written in Rust language with the use of SDK, which hides the proxy-wasm implementation. Once applications written in Rust are developed, they have to be compiled by hand and put into a special folder from which automatic file syncer will distribute them across edge servers.

This thesis aims to automatize the manual compilation part, and thus, it does not focus on any other parts already implemented.

6.2 Used technologies

The implementation part of this thesis uses many different technologies to enable building the best solution possible. This section will describe what are the selected technologies and the thinking process that led to their selection.

6.2.1 Rust

Rust was selected as the main programming language in which all code will be written. It was partly due to the already existing infrastructure, which was written in rust, and also due to its key values and ecosystem. The Rust's key values are: *Performance*, *Reliability* and *Productivity.* [52]

Performance is achieved by having no runtime or garbage collector. Nevertheless, having no garbage collector doesn't mean the language is not memory-safe. The second key value of Rust is reliability. It has a rich type system and ownership model that guarantees the aforementioned memory safety together with thread safety, enabling the elimination of many classes of bugs at compile-time. [52]

Lastly, the best value for active development is probably productivity. Rust has vast documentation and a friendly compiler with useful error messages. It offers tooling with an integrated package manager and build tool, smart multi-editor support with type auto-completion and type inspections, automatic formatter, and more. [52]

Rust is also very close to the WebAssembly. It offers the tolling necessary to build Wasm apps, and it is also used by notable Wasm runtimes like *Wasmtime* (3.4) and *Wasmer*[1].

6.2.2 Tokio async runtime

Tokio is an asynchronous runtime for the Rust programming language. It provides the building blocks needed for writing network applications. It gives the flexibility to target a wide range of systems, from large servers with dozens of cores to small embedded devices.[53] *Async runtimes are libraries used for executing async applications.* [54]

Tokio was chosen because it is a popular asynchronous runtime with HTTP, gRPC, and tracing frameworks. It is also used in other CDN projects, so there is no need to learn other technologies.

6.2.3 S3 SDK

The thesis chose an S3 SDK for working with object storage due to S3 being prominent on the market and because many object storage solutions have a compatible API. The S3 SDK is provided as a part of the *AWS SDK for Rust. The AWS SDK for Rust (the SDK) provides Rust APIs to interact with Amazon Web Services infrastructure services. Using the SDK, one can build applications on top of Amazon S3, Amazon EC2, DynamoDB, and more.* [55]

[1] https://github.com/wasmerio/wasmer

6.2.4 MinIO

MinIO is a high-performance, S3 compatible object store. It is built for large-scale AI/ML, data lake, and database workloads. It is software-defined and runs on any cloud or on-premises infrastructure. [56] The thesis uses it as an object storage for testing. This is possible because of the S3-compatible store.

MinIO comes with a prepared Docker image, which is designed to be run in Podman or Docker. Only editing `MINIO_ROOT_USER` and `MINIO_ROOT_PASSWORD` environment variables that set user credentials and access keys to the storage is enough to be able to use MinIO comfortably.

6.2.5 Sqlx

SQLx is an async, pure Rust SQL crate featuring compile-time checked queries without a DSL. [57] It is an easy way how to implement a persistent layer that utilizes relational databases and SQL. By being truly asynchronous, it can be leveraged by other async Rust libraries like *Tokio* to its maximal potential.

It is database agnostic supporting *PostgreSQL*(6.2.6), *MySQL*[2], *MariaDB*[3] and *SQLite*[4] and async runtime agnostic supporting *async-std*[5], *tokio*(6.2.2) and *actix*[6] runtimes.

6.2.6 PostgreSQL

For metadata storage, the thesis chose the *PostgreSQL* database. *It is a powerful, open-source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads.* [58]

It is a popular database that has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open-source community behind the software to consistently deliver performant and innovative solutions. [58]

It runs on all major operating systems, has a long history, and has a container Image prepared for an easy start without needing any bigger preparation. Only providing a few environment variables is enough to start a configured PostgreSQL database. These variables are:

- `POSTGRES_USER` – Creates database user.
- `POSTGRES_PASSWORD` – Sets password for the used.
- `POSTGRES_HOST` – Hostname of the postgres database.

[2] https://www.mysql.com/
[3] https://mariadb.org/
[4] https://www.sqlite.org/
[5] https://async.rs/
[6] https://actix.rs/

- `POSTGRES_DB` – A database name created in the PostgreSQL.

6.2.7 Firecracker vs gvisor

When selecting which technology should be used for sandboxing, two of them stood out from the research as a safe and efficient choice: firecracker (4.3.2.2) and gVisor (4.3.3.3).

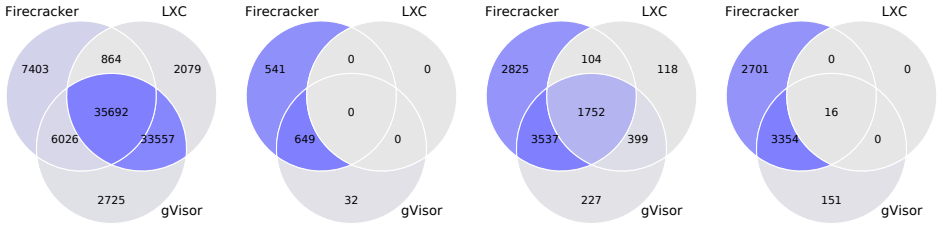
They are used to achieve practically the same goal: providing a fast and secure sandbox. However, both technologies had to implement a different part of the puzzle. Firecracker had to make safe, isolated VMs faster, while gVisor had to make fast containers more secure. In the end, both are successful products that are used in production by giant cloud computing companies, firecracker by *AWS Labmda and AWS Fargate* [37] and gVisor by *Google* [47]. This comparison naturally starts a debate about which one to use, which was explored in a study *Blending Containers and Virtual Machines: A Study of Firecracker and gVisor* [59], comparing both firecracker and gVisor with additional Linux containers (LXC).

This study focuses on two criteria of comparison: Isolation and performance. The isolation is measured by comparing how they utilize the Linux kernel by measuring code coverage of the Linux kernel in response to different workloads on each platform. And the performance by tracing memory utilization, execution times, and network workload.

6.2.7.1 Code coverage

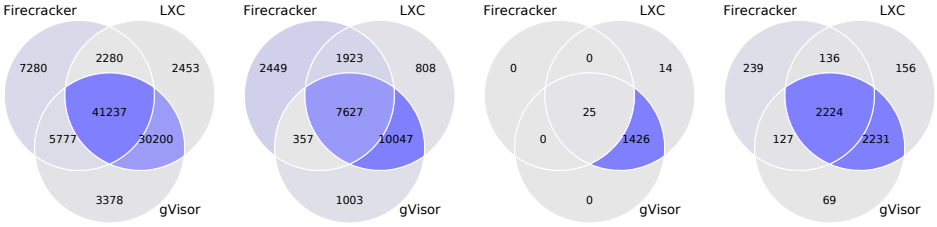
When measuring system call coverage, it is important to state that both gVisor and firecracker operate in *secure computing mode – seccomp*. Which restricts system calls a running process may make. The firecracker uses a seccomp filter, which allows only 36 system calls to the host kernel, and meanwhile, gVisor allows 53 without host networking and up to 68 with host networking. Firecracker also supports *advanced* filtering mode, which restricts values allowed as arguments to the system calls. [59]

The total code footprint was measured on four microbenchmarks testing CPU, network, memory, and file write on the tested systems. Table 6.1 shows the resulting union of line coverage across all workloads out of 806,318 total lines in the Linux kernel. However, this table doesn't show anything about whether the platforms are executing the same code or if there are large non-overlapping bodies of code. To get this information the study takes a detailed look at the code executed for different kernel subsystems when running microbenchmarks exercising that subsystem. It filters out invocations not related to the application running in the isolation platform and creates a set of Venn diagrams representing the intersections of code executed by each system. These diagrams are in figures 6.1, 6.2, 6.3, and 6.4. [59]



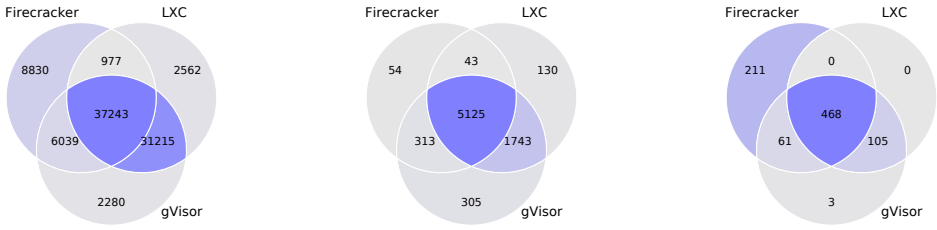
(a) CPU workload overall **(b)** /virt **(c)** /arch **(d)** /arch/x86/kvm

Figure 6.1 CPU workload coverage [59]



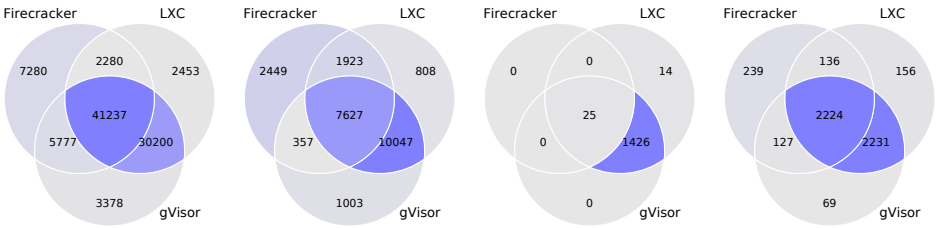
(a) Network workload overall **(b)** /net **(c)** /net/bridge **(d)** /net/core

Figure 6.2 Network Workload Coverage [59]



(a) Memory workload overall **(b)** /mm **(c)** /arch/x86/mm

Figure 6.3 Memory workload coverage [59]

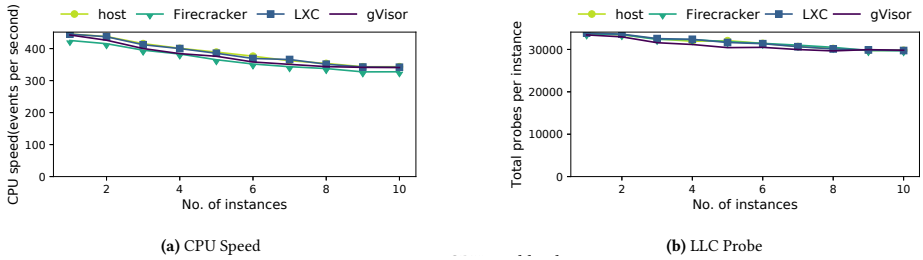


(a) Network workload overall **(b)** /net **(c)** /net/bridge **(d)** /net/core

Figure 6.4 File write workload coverage [59]

	Host	Firecracker	gVisor	LXC
Lines	63.163	77.392	91.161	90.595
Coverage	7.83%	9.59%	11.31%	11.23%

Table 6.1 Union of line coverage across all workloads out of 806,318 total lines in the Linux kernel. [59]



(a) CPU Speed **(b)** LLC Probe

Figure 6.5 CPU workload [59]

6.2.7.2 Performance

Performance was observed in four categories:

1. CPU – Measured by running a CPU benchmark for ten seconds and observing the number of events it executes.
2. Network – Measured by bandwidth and latency.
3. Memory allocation cost – Measured by allocating and freeing memory with mmap() and munmap() system calls.
4. I/O throughput – Measured with a microbenchmark that performs reads and writes of various sizes on a 1 GB file.

Results for each respective category are in figures 6.5, 6.10, 6.6, 6.7, 6.8, 6.9, 6.11 6.12. [59]

6.2.7.3 Outcomes

Firecracker's microVMs are effective at reducing the frequency of kernel code invocations but had a much smaller impact on reducing the footprint of kernel code. Running workloads under Firecracker often expanded the amount of kernel code executed with support for virtualization.

The gVisor design leads to much-duplicated functionality: while gVisor handles the majority of system calls in its user-space Sentry, it still depends on the same kernel functionality as Linux containers. Thus, its design is inherently more complicated, as it depends on multiple independent implementations of similar functionality. In contrast, LXC relies on a single implementation

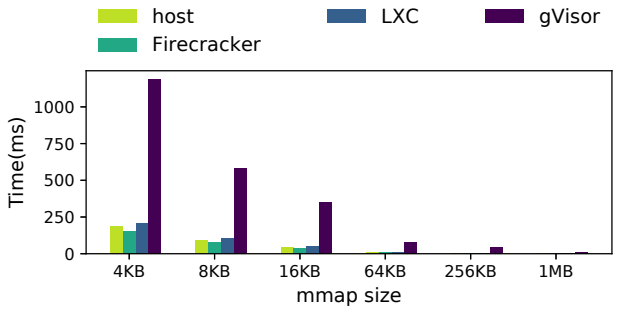


Figure 6.6 Total allocation time (without munmap) for 1GB [59]

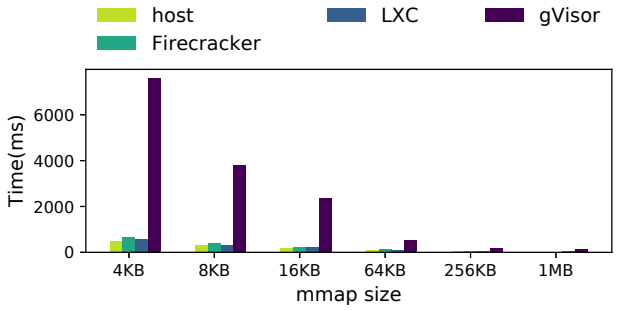


Figure 6.7 Total allocation+unmap time for 1GB [59]

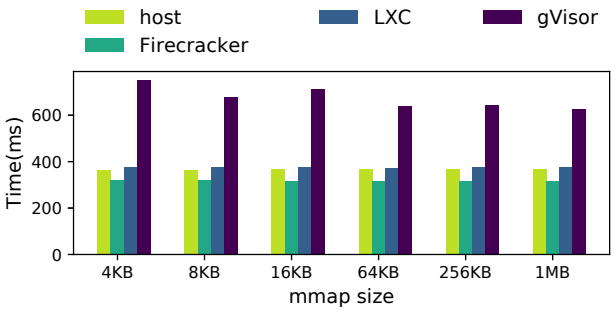


Figure 6.8 Total touch time (without munmap) for 1GB [59]

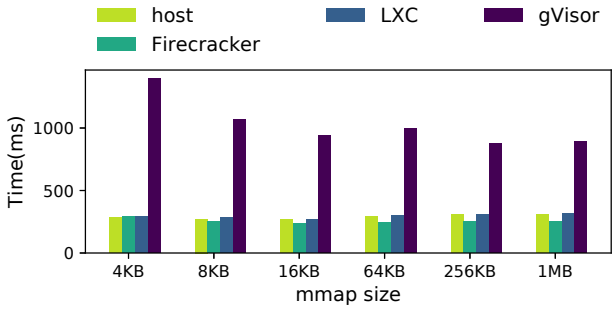


Figure 6.9 Total touch time (with munmap) for 1GB [59]

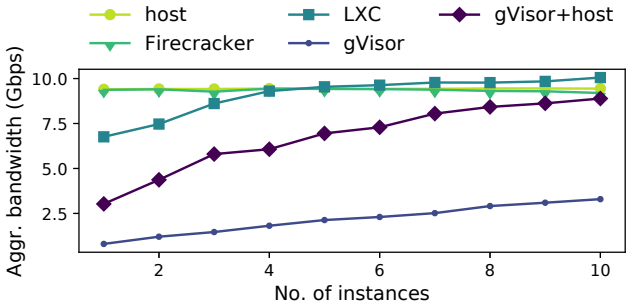


Figure 6.10 Aggregate Network Bandwidth [59]

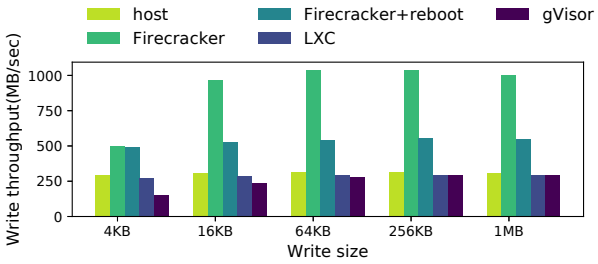


Figure 6.11 Write Throughput [59]

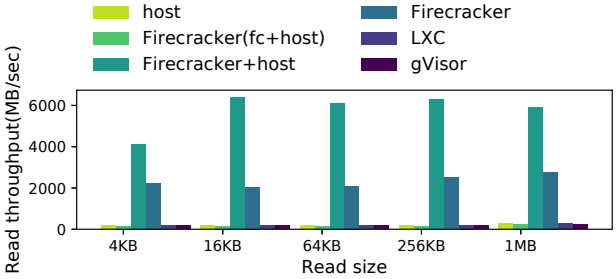


Figure 6.12 Read Throughput [59]

of OS functions in the Linux kernel, and Firecracker relies on a much-reduced set of kernel functionality for performance.

All things considered, the firecracker did better in terms of kernel code utilization by using the smallest amount of host kernel in all categories except the virtualization and surprisingly enough the firecracker did better even in terms of performance. It had lower allocation time and higher I/O throughput. On the other hand, gVisor won by a small margin in the CPU workload category. Nevertheless, the differences were so small that they can be disregarded.

The decision of whether to use Firecracker or gVisor was a tough one. Based on the outcomes of this study together with the facts, that the Firecracker uses a traditional hypervisor that was tested by time and the Firecracker has a much smaller and simpler design. It was chosen as the sandboxing technology that will be used in the implementation. [59]

6.2.7.4 Development

To make development as smooth and fast as possible for the first iteration of the pipeline the firecracker is not used. As containerization is used in other projects the infrastructure and technologies like Gitlab CI were already established. For this reason, the first iteration uses Podman containers as its sandbox of choice because of its capability to run as non-root. When paired with gVisor it also offers a strong isolation which could be used on its own.

6.2.8 Docker

Even though gVisor and containerization were decided not to be used. The implementation still requires orchestrating its services, like PostgreSQL and MinIO. For this, the containerization is the perfect tool. Docker will be used in the project as it is the most prevalent container engine today.

6.3 Orchestration

Orchestration is implemented as a binary rust crate. It is the main entry point to the system and as a result, also contains end-to-end tests. As proposed in the design it can be split into two smaller parts: Customer facing API and Evaluator.

6.3.1 Customer facing API

This part is split into three modules: `application`, `version`, `version_build`. Each provides API for their respective use case. From these three, the `version` module is probably the most interesting as it handles the creation of version build pictured in the sequence diagram in figure 5.5.

All handler functions use a shared state `APIState`, which contains services of the persistent storage and the `BuildClient` that sends requests to the compilation component. Implementation wise the `create_version` function roughly corresponds to the listing 6.1.

The data is received in a multipart format, which has to be parsed and converted to a more practical structure for implementation. Then a version build is created in permanent storage, with information supplied in the request. After the build is created source code is sent to the compilation service over HTTP and the version build entity is returned to the `create_version` caller.

Services used in the code examples are implementing the `tower::Service` trait, which is a simplified interface that makes it easy to write network applications in a modular and reusable way, decoupled from the underlying protocol. [60] It notably specifies two methods: `poll_ready` and `call`. The `poll_ready` must be used to determine whether the service can process a request. It is used to handle backpressure when a service is not ready for the next request. The `call` is used to call the service and use its functionality. The `ready()` method in the example is a simplified way how to call the `poll_ready` method.

6.3.2 Evaluator

The evaluator has one module `build_operations`, which contains three handlers: `build_success`, `build_fail` and `build_crash`. The first two were discussed in the design chapter of this thesis however the `build_crash` wasn't.

It is used as a way how to note that the build failed due to internal error. All information in this type of build can be wrong and should be treated as such. The difference between a `failed` build and a `crashed` build is that a `failed` build failed because of the compilation and always has output logs from the said compilation. The `crashed` build doesn't have to have any information. Listing 6.2 shows a rough example of the `build_fail` handler.

First, the data has to be parsed from the multipart format and then the `VersionBuildService` is used to update version status to failed. Again, the `tower::Service` trait is used to make the code composable and reusable.

```rust
pub async fn create_version(
    State(state): State<Arc<APIState>>,
    multipart: Multipart,
) -> Result<Json<VersionBuild>, StatusCode> {
    let mut version_build_service = state.version_build_service
        .clone();

    let new_version = multipart_to_new_version(multipart)
        .await?;

    let version_build = NewVersionBuild {
        version_information: new_version.version_information,
    };

    let version_build_created = version_build_service
        .ready()
        .await
        .call(VersionBuildRequest::Create(version_build))
        .await;

    let source_code = new_version.source_code;

    // Start of the build pipeline
    let build_request = BuildRequest {
        id: version_build.id,
        language: version_build.application_language.clone(),
        source_code,
    };

    let mut build_service = state.build_service.clone();
    let build_task = build_service
        .ready()
        .await
        .call(build_request);

    let build_task_submit_result = build_task.await;

    Ok(Json(version_build))
}
```

Code listing 6.1 Create version handler

```
async fn build_fail_exec(
    state: Arc<WasmithState>,
    version_id: Uuid,
    multipart_data: Multipart,
) -> Result<StatusCode, StatusCode> {
    let logs = multipart_to_logs(multipart_data).await?;

    let mut version_build_service = state.version_build_service
        .clone();

    let version_build_request = VersionBuildRequest::BuildFail(
        version_id, logs);

    let version_build_set_as_failed = version_build_service
        .ready().await.call(version_build_request).await;

    Ok(StatusCode::OK)}
```

Code listing 6.2 Function handling build fail operation

6.3.3 End-to-end tests

The end-to-end tests are for determining whether all components work together and provide a build pipeline that was designed. The tests are composed of the following steps:

1. Docker containers with PostgreSQL, MinIO, and Nginx are started, using docker compose.
2. Prepared Rust source code is archived.
3. The two servers, the user-facing orchestrator and private compilator, are started.
4. The archived source code is sent to the orchestrator for processing.
5. Every 10 seconds the orchestrator is polled to check whether a new version is available. If the version is not available in 30 seconds the test fails.
6. Once a new version is available, the Wasm binary is uploaded to nginx and activated
7. A request is sent to the resource where the new application should be active.
8. The response is checked to determine whether the application is working correctly. The application can, for example, set a new header, which is not present when the application is not running.

The tests are also part of the GitLab CI/CD solution which is used for the versioning of the build pipeline.

6.4 Compilation

The compilation component is separated into its own binary crate. It contains one handler the `builder` which controls the build requests. It, similar to the orchestrator, uses a shared state to contain configuration and all necessary information.

As proposed in the design, the handler uses a `BuildService` which works regardless of the communication model and does everything necessary to compile the code. Build service is also a `tower::Service` which accepts a request on its input and as its output returns compiled code with logs. The actual `BuildRequest` and response can be seen in the listing 6.3:

```
pub struct BuildRequest {
    pub id: Uuid,
    pub language: ApplicationLanguage,
    pub source_code: Vec<u8>,
}

pub struct SourceCode(pub Vec<u8>);
pub struct WasmBinary(pub Vec<u8>);
pub struct Logs(pub Vec<u8>);

pub enum BuildServiceResult {
    Success(SourceCode, WasmBinary, Logs),
    Fail(Logs),
    Crash,
}
```

Code listing 6.3 `BuildService` request and result

Another interesting part is the implementation of the sandbox. Because the sandbox has to be replaceable, a trait defining all its operations was created. The trait is in the next code listing 6.4:

The trait is split into three: `UnitializedSandbox` and `PreparedSandbox` and `Sandbox`. This is because the implementation wanted to separate a sandbox that is ready to execute a command and a sandbox that doesn't have everything prepared.

These traits are implemented by the `PodmanContainer` which uses a type state pattern with two states: `Inactive` and `Running`. In the case of podman, the `Inactive` state corresponds to an inactive container. The inactive container can be used to copy data to and from but cannot execute any com-

```rust
#[trait_variant::make(Sandbox: Send)]
pub trait LocalSandbox {
    type Id: Debug + Display + Eq + Clone + Send + Sync;
    type RunError: Error + Send + Sync;

    fn get_id(&self) -> &Self::Id;

    async fn copy_to(
        &mut self,
        host_dir: &Path,
        sandbox_dir: &Path
    ) -> Result<(), Self::RunError>;

    async fn copy_from(
        &mut self,
        sandbox_file: &Path,
        host_target: &Path,
    ) -> Result<(), Self::RunError>;
}

#[trait_variant::make(UnitializedSandbox: Send)]
pub trait LocalUnitializedSandbox {
    type Id: Debug + Display + Eq + Clone + Send + Sync;
    type RunError: Error + Send + Sync;

    type AfterInit: PreparedSandbox<RunError = Self::RunError>
        + Sandbox<Id = Self::Id, RunError = Self::RunError>;

    async fn init(self) -> Result<Self::AfterInit, Self::RunError>;
}

#[trait_variant::make(PreparedSandbox: Send)]
pub trait LocalPreparedSandbox {
    type RunError: Error + Send + Sync;

    async fn run_command(
        &mut self,
        command: &str
    ) -> Result<Output, Self::RunError>;
}
```

Code listing 6.4 Sandbox traits

mand. The `init` method is implemented by starting the container and keeping it running. This transforms the container into `Running` state and can execute commands.

All operations and sandbox preparations are done in the `build_pipeline` module. The `build_pipeline` contains all parts necessary to compile an application in a sandbox not depending on the sandbox's concrete implementation. The `run` method of the pipeline is roughly pictured in the code listing 6.5:

The pipeline is divided into stages based on the proposed design, which outlines the steps necessary to compile source code in the sandbox. There are three stages: Copy to the sandbox, run compilation, and copy from the sandbox.

6.4.1 Sandbox provider

Before anything is done in a sandbox, the sandbox has to be created first. That is done by the `podman_provider` module, which maintains and provides the podman container sandboxes. The `podman_provider` has method `get_sandbox` which finds an available sandbox, returns it, and issues the creation of a new sandbox.

The `podman_provider` contains a `PodmanImage` for every possible source code language (currently only Rust) which it uses to instantiate containers for the language. The `get_sandbox` method is roughly outlined in the code listing 6.6:

The creation of the new container is done with every `get_sandbox` call which is not ideal. Because it effectively destroys the advantage of having a buffer of prepared containers and every request creates a new container regardless. This will be later replaced by an async call which will create a new container asynchronously.

6.5 Storage

The storage is a library crate, providing services for working with databases. The crate is divided into four modules: `entity`, `repository`, `service`, and `testing`. The odd one out of these four is the testing module. It contains mock implementations of repositories, so they can be used inside unit tests.

The crate also contains a *migration* folder, that contains SQL scripts that when used will create tables and enums necessary to use the database. The *migration* folder is used by the *sqlx* database library during build. The concrete use of the migration functionality is in the rust build script in listing 6.7.

It by default locates all scripts inside the *migration* folder of the project. The files can also be located in another folder, but it would have to be specified explicitly in the `sqlx::migrate!(migration_folder: String)` call.

6.5.1 Shared definitions

The storage can be logically split into two sections *metadata* and *objects* storage. Each has its own entities and repositories, however some definitions are common to both. This section will look into them and provide details about the implementation.

The first element that comes into mind is the `Entity` trait. The entity trait is independent on the used storage technology and defines a unit of storage for the application. The trait is very simple. it has only one method `get_id(&self) -> &Self::Id` which returns id of the entity. The full definition is in listing 6.8.

Another common element is the `Repository` trait. The trait defines operations `create`, `read`, `delete`, the `update` operation is defined in separate trait the `UpdateRepository`. In hindsight, this design choice is not ideal, repositories should not share almost any functionality, except some very general ones, like `read_all`. Concrete functions should be defined ideally in entity-specific repositories like `ApplicationRepository` or `VersionRepository`. The actual definition of `Repository` is in listing 6.9.

6.5.2 Metadata

The metadata section of the storage contains the most storage functionality. It contains entities: `Application`, `Version`, `VersionBuild` with two enums: `ApplicationLanguage` and `ApplicationPhase`. These entities are independent on the used database technology and should remain mostly the same if different database technology is used. However, some things would need to change.

Sqlx offers two possible ways how to do a SQL query: `query` function and `query!` macro. These functions also have versions `query_as` and `query_as!` that automatically convert database results into the Rust struct. In order for the `query_as` function to work entities have to derive or implement `FromRow` trait, which defines how the struct is converted to the database entity. However, the `query_as!` macro does not and converts entities automatically. Because implementation chose to use the macro version and some databases don't always support the same data types. When a database or database library would change, there would be a need to also change data types that are not supported in the used technology. Nevertheless, that is not a big problem and it could be solved by having explicit conversion functions. An example of an entity is in listing 6.10.

Besides entities, the metadata section also contains repositories. All repositories are implemented with the use of sqlx library. The listing 6.11 shows the implementation of the version build read operation. It uses `query_as!` macro to automatically convert to `VersionBuild`.

6.5.3 Object storage

The object storage contains only one entity and one repository. The entity is in listing 6.12. The entity contains only data and ID. The repository is implemented by using *AWS S3 SDK for Rust*. The create operation of the repository is in listing 6.13 and shows how the S3 client is used.

The configuration of the object storage is provided at the instantiation of repositories, this includes access key, access secret, region, and bucket. All data is stored inside this one bucket.

```
    pub async fn run<S: Sandbox + UnitializedSandbox>(&mut self,
        source_code_location: &str,
        sandbox_sdk_location: &str,
        sandbox_source_code_location: &str,
        mut sandbox: S,
    ) -> PipelineResult {
        // Copy base dir into the sandbox
        let copy_result = sandbox.copy_to(
                &self.base_dir,
                Path::new(sandbox_source_code_location)).await
            .map_err(|_| Crash)?;
        let output_file_name = self.get_wasm_binary_name();
        let sandbox_output_location = format!(
            "{sandbox_source_code_location}/{output_file_name}");
        // Init sandbox
        let mut sandbox = match sandbox.init().await
            .map_err(|_| Crash)?;

        // Run the build in sandbox
        let Ok((pipeline_continue, logs)) = run_build_stage(
            &mut sandbox, &self.source_language,
            sandbox_sdk_location, sandbox_source_code_location,
            &archive_name, &sandbox_output_location,
        ).await else { return Crash; };

        if !pipeline_continue {
            return Fail(logs);
        }
        // Copy sandbox output to the output folder
        let pipeline_continue = copy_to_output_stage(
            &self.id, &mut sandbox,
            &self.output_dir, &output_file_name,
            &sandbox_output_location,
        )
        .await;
        if !pipeline_continue {
            return Fail(logs);
        }
        finish(&self.output_dir, &output_file_name, logs).await
    }
}
```

Code listing 6.5 Build pipeline

```
 pub async fn get_sandbox(
        &self,
        source_language: &ApplicationLanguage,
    ) -> Result<PodmanContainer, PodmanProviderError> {
        let available_containers = self.available_containers
            .get(source_language);

        // Pop a container
        let container = available_containers.pop();

        let image = self.build_images.get(source_language);

        // Create a new container
        let new_container = image.create_container().await;
        available_containers.push(new_container);

        // Return the container
        Ok(container),
}
```

Code listing 6.6 Get sandbox method of sandbox provider

```
 #[tokio::main]
async fn main() {
    let db_url = env!("DATABASE_URL");
    let db_pool = PgPool::connect(db_url)
        .await
        .expect("Could not connect to db.");

    sqlx::migrate!()
        .run(&db_pool)
        .await
        .expect("Could not run migrations.");
}
```

Code listing 6.7 Storage library build script

```
pub trait Entity: Send + Sync {
    type Id: Display + Send + Sync;

    fn get_id(&self) -> &Self::Id;
}
```

Code listing 6.8 Entity trait

```rust
pub trait Repository<T: Entity>: Send + Sync {
    type CreateError: Error + Send + Sync;
    type ReadError: Error + Send + Sync;
    type DeleteError: Error + Send + Sync;

    fn create(
        &self,
        entity: T,
    ) -> impl std::future::Future<
            Output = Result<T::Id, Self::CreateError>>
            + Send;

    fn read(
        &self,
        id: &T::Id,
    ) -> impl std::future::Future<
            Output = Result<Option<T>, Self::ReadError>>
            + Send;

    fn delete(
        &self,
        id: &T::Id,
    ) -> impl std::future::Future<
            Output = Result<(), Self::DeleteError>>
            + Send;
}
```

Code listing 6.9 The definition of the `Repository` trait

```rust
#[derive(Eq, PartialEq, Debug, Clone, Serialize, Deserialize)]
pub struct Application {
    pub id: Uuid,
    pub owner_id: i64,
    pub name: String,
    pub created: DateTime<Utc>,
    pub notes: String,
}

impl Entity for Application {
    type Id = Uuid;

    fn get_id(&self) -> &Self::Id {
        &self.id
    }
}
```

Code listing 6.10 The entity for application

```
async fn read(
        &self,
        id: &<VersionBuild as Entity>::Id,
    ) -> Result<Option<VersionBuild>, Self::ReadError> {
        let found = sqlx::query_as!(
            VersionBuild,
            r#"
            SELECT
                id,
                application_id,
                version,
                application_language as "application_language: _",
                conf,
                created,
                notes,
                phases,
                state as "state: _",
                path_to_logs
            FROM version_build
            WHERE id = $1
            "#,
            id
        )
        .fetch_one(&self.db_pool)
        .await;

        match found {
            Ok(value) => Ok(Some(value)),
            Err(Error::RowNotFound) => Ok(None),
            Err(err) => Err(err),
        }
    }
```

Code listing 6.11 The version build read operation using sqlx

```rust
#[derive(Eq, PartialEq, Debug, Clone)]
pub struct ApplicationData {
    pub id: ApplicationDataId,
    pub data: Vec<u8>,
}

pub type ApplicationDataId = String;

impl Entity for ApplicationData {
    type Id = ApplicationDataId;

    fn get_id(&self) -> &Self::Id {
        &self.id
    }
}
```

Code listing 6.12 The entity representing data

```rust
async fn create(
        &self,
        entity: ApplicationData,
    ) -> Result<<ApplicationData as Entity>::Id, Self::CreateError>
    {
        let key = entity.id;
        let body = ByteStream::from(entity.data);

        self.s3_client
            .put_object()
            .bucket(&self.bucket)
            .key(&key)
            .body(body)
            .send()
            .await?;
        Ok(key)
    }
```

Code listing 6.13 The create operation using S3 SDK

Chapter 7

Build optimizations

This brief chapter will go into detail on how the compilation can be made better to make resulting Wasm binaries startup and execute faster, limiting cold start problems for edge computing applications.

7.1 Problem definition

Before the Wasm binary executes, three steps must be taken: Compilation, instantiation, and initialization. Each step takes some time. Figure 7.1 shows how these steps are distributed through a developer, build pipeline, and startup time.

Compilation – Most of this time is taken by compilation. Compilation here does not mean compilation from source code to the Wasm binary but from the Wasm binary to the concrete executable. Because of this, runtimes try to optimize this step the most. [61]

Instantiation – The instantiation is the process that takes a static Wasm module and makes it a dynamic instance with its state. The instantiation notably allocates and initializes linear memory and tables. The memory can be allocated either *on-demand* or via *pooling* with the use

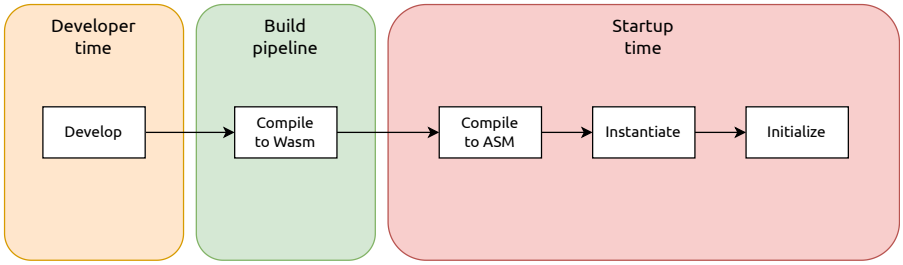


Figure 7.1 Distribution of steps required to run Wasm application

of `userfaultfd`. Allocating memory on demand with `mmap` is slow but simple. The pooling method, on the other hand, is faster. It preallocates memory into a pool and allows a program to initialize memory pages lazily when *page fault* occurs. [61, 62]

Initialization – The last step, the initialization, is a one-time code setup, which has to be done before the code is executed. It is the responsibility of the developer that the initialization runs. The initialization can be done in three ways: Always at the start of the program, Lazily when initialized data is needed, or ahead-of-time (A). The ahead-of-time is the best option to reduce time spent on execution. [61]

7.2 Solutions

All mentioned steps introduce an overhead that results in users experiencing higher latency when initializing Wasm applications. One by one, the thesis will go into each step and discuss what can be done to reduce the latency.

7.2.1 Compilation

As already mentioned, compilation takes up the most time. Although Wasm could be theoretically interpreted, wasmtime, for example, does not support the interpretation of wasm code. Other than that, interpreted code would be much slower than code compiled into native binary. [62]

To achieve the best compilation time possible, it could be done ahead of time (AOT). This moves the compilation from the startup time into the build pipeline time. This could be done quite easily because, for example, wasmtime offers AOT compilation. However, having a separate build pipeline with a different CPU architecture from the real machine executing the code creates a problem. This, however, could be solved by simply compiling the code for the target architecture or running inside a VM. [61]

7.2.2 Instantiation

Instantiation offers the least amount of pre-computation that can be moved into the build pipeline time. But one potential for speedup is the pool allocation mentioned before. Initializing memory lazily increases cold start times by moving the initialization into the run time, which could be better depending on the application. However, some applications might require running fast, and for them, the lazy initialization would introduce too much overhead. [62]

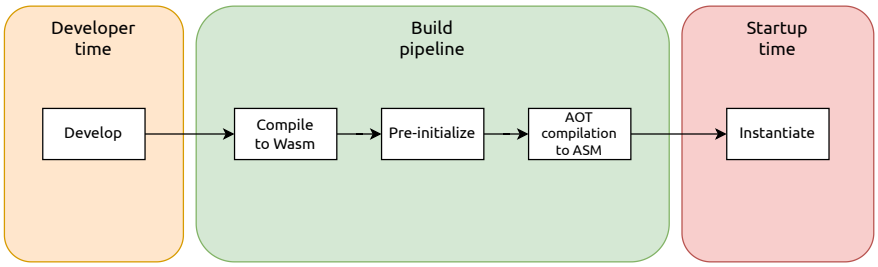


Figure 7.2 Distribution of steps required to run Wasm application after optimization

7.2.3 Initialization

Initialization is another step, although not always possible, which can be done ahead of time. For this purpose, a tool called *Wizer* has been created. The Wizer initializes the code ahead of time and saves a snapshot of the initialized state to a new Wasm binary. [61]

The pre-initialization with Wizer cannot use host-imported functions with one exception. The Wasi is optionally allowed. However, the developer must note that the pre-initialization can be done on completely different machines, so for example, files do not have to be present when created during initialization, similar to the AOT compilation. [61]

By applying the Wizer together with other optimization techniques, the time distribution of the application's startup would look like in the figure 7.2. Currently, the build pipeline does not implement any build optimizations but will in the future.

Chapter 8

Future work

In this chapter, the thesis will go through some aspects of the work that will be worked on in the future.

8.1 Build optimization implementation

The most notable work that will be done in the future is the implementation of build optimization techniques. All of the mentioned techniques can be used to make cold starts as fast as possible. The optimization techniques should also be configurable, and the user should be able to select whether they want to enable or disable the optimization.

8.2 Test bed

With the implementation of the build pipeline, users can submit source code for compilation, which can later be uploaded to Nginx servers. However, users cannot currently efficiently test their implementation on live servers. Future pipeline versions should ensure the users can test their application on some test server so they can debug the code before it goes into production.

8.3 Build client

The build pipeline offers REST API, which the user can interact with. However, this is not always the best possible way of interacting and some users might want to use a prepared client through which they can work with the pipeline more easily. The client could be a CLI application offering commands that would translate into the calls.

Another solution could be to create a web UI that offers the same functionality as the CLI client but through a web application. This would provide a nice user interface where users could see all applications created with all their versions and active builds. They could also deploy applications to selected servers with one click.

Summary

This thesis had multiple goals. The first was to analyze edge computing with WebAssembly and study how they can be used together. Second, to design and implement a compilation pipeline, which compiles source code into Wasm binary. There was also an optional goal to try multiple build optimization techniques to reduce cold start times of Wasm applications running on the edge.

All required goals were completed successfully. While writing the thesis, much understanding was gained about edge computing and WebAssembly, together with information about executing untrusted code with sandboxing.

The thesis analyzed how WebAssembly can safely power edge computing with its inherently safe runtimes. Furthermore, found a new emerging standard for using Wasm as a proxy. Based on this analysis, the thesis designed a solution for how a source code can be compiled to Wasm to be later executed on edge servers.

The design was then implemented in a Rust programming language, using other technologies that provided the compilation with a sandbox and other necessary infrastructure. The implementation was tested to ensure it worked as expected and the compiled Wasm binary could be deployed to edge servers.

The final optional goal to try different techniques to limit cold start times of the Wasm binary was completed only partially. The thesis explored ways to lower the time necessary to load Wasm plugins. However, they were not added to the final implementation.

Appendix A
Proxy-wasm

A.1 Function and callback categories

- Integration – Definitions for initialization.
- Memory management – Definitions for memory allocation.
- Context lifecycle – Definitions for operations on context lifecycle events.
- Configuration – Definitions for operations on Wasm VM configuration event.
- Logging – Definitions for logging.
- Clocks – Definitions for retrieving time.
- Timers – Definitions for setting a timer that executes a function every timer tick.
- Randomness – Definitions for generating random data.
- Environment variables – Definitions for retrieving environment variables from the host.
- Buffers – Definitions for reading and writing to special buffers. Access to these buffers using functions is restricted. Only specific callbacks can access specific buffers. There are nine buffers:
  - `HTTP_REQUEST_BODY`
  - `HTTP_RESPONSE_BODY`
  - `DOWNSTREAM_DATA`
  - `UPSTREAM_DATA`
  - `HTTP_CALL_RESPONSE_BODY`

  - `GRPC_CALL_MESSAGE`
  - `VM_CONFIGURATION`
  - `PLUGIN_CONFIGURATION`
  - `FOREIGN_FUNCTION_ARGUMENTS`
- HTTP Fields – Definitions for reading and writing to HTTP fields such as request/response headers, request/response trailers, and gRPC initial/trailing metadata. Again, as seen in the Buffers section, only selected fields can be accessed from selected callbacks.
- Common stream operations – Definitions for working with streams. Stream types the definitions can work with are `HTTP_REQUEST`, `HTTP_RESPONSE`, `DOWNSTREAM`, `UPSTREAM`. Downstream is the connection between the client and proxy, and upstream is the connection between the proxy and the backend.
- TCP/UDP/QUIC streams – Definitions for operations on connection events
- HTTP – Callbacks that are executed on data receive events and functions that send HTTP responses.
- HTTP calls – Callbacks that are executed on data receive events and functions that send HTTP requests.
- gRPC calls – Definitions for working with gRPC.
- Shared Key-Value Store – Definitions for working with a shared key-value store.
- Shared queues – Definitions for working with shared queues.
- Metrics – Definitions for working with metrics.
- Properties – Definitions for working with properties. Properties are currently implementation-dependent and not stable.
- Foreign function interface (FFI) – Callbacks that execute when a foreign function is called and functions that call a foreign function.
- Unimplemented WASI functions – Definitions for unimplemented WASI functions that are expected to be present when the Wasm module is compiled for the wasm32-wasi target.

All function and callback categories are documented on proxy-wasm github [29].

A.2 Types

- `proxy_log_level_t` – The proxy's log level.
- `proxy_status_t` – The return type of all functions.
- `proxy_action_t` – The type returned in callbacks for proxy events. It can be either `CONTINUE` or `PAUSE`, and it signals whether the caller should stop processing whatever it does during the callback.
- `proxy_buffer_type_t` – All possible buffer types. All possible values in this type are described in the previous section 3.5.1 subsection *Buffers*.
- `proxy_header_map_type_t` – All possible types usable in functions that are in section *HTTP field*.
- `proxy_peer_type_t` – Type of peer. It can be either `LOCAL`, `REMOTE` or `UNKNOWN`. It is used in the callbacks executed on closing downstream/upstream connection events.
- `proxy_stream_type_t` – The stream type that must be specified in functions for closing or continuing stream.
- `proxy_metric_type_t` – Type of metric used in functions defining metrics.
- `wasi_errno_t` – The return type of functions defined by Wasi.
- `wasi_fd_id_t` – Type of wasi file descriptor. It can be either `STDOUT` or `STDERR`.
- `wasi_clock_id_t` – Type of wasi clock. It can be either `REALTIME` or `MONOTONIC`.

All types are documented on proxy-wasm github [29].

Appendix B

User facing API

B.1 Application

B.1.1 Create

This endpoint creates a new application without any version. Application is used for grouping versions.

Method : POST

Endpoint : `/application`

Request body : JSON

```
{
    owner_id: i64,
    name: String,
    notes: String
}
```

Response :

- Success: Status code `201` with JSON body of created entity.
- Fail: Status code according to the failure type.

B.1.2 Read

This endpoint returns an application based on id.

Method : GET

Endpoint : `/application/<application_id>`

Response :

- Success: JSON of application entity.
- Fail: Status code according to the failure type.

B.1.3 Update

Update application information.

Method : PATCH

Endpoint : `/application/<application_id>`

Request body : JSON

```
{
    name: String,
    notes: String
}
```

Response :

- Success: JSON of the updated entity.
- Fail: Status code according to the failure type.

B.1.4 Delete

Delete application based on id with all its versions and version builds

Method : DELETE

Endpoint : `/application/<application_id>`

Response :

- Success: Status code `200`.
- Fail: Status code according to the failure type.

B.2 Version

B.2.1 Create

Create new version by submitting source code for build.

Method : POST

Endpoint : `/version`

Request body : JSON

```
{
    plugin_id: Uuid,
    version: Semantic version,
    plugin_language: ApplicationLanguage,
    phases: Vec<ApplicationPhase>,
    conf: Option<String>,
    notes: Option<String>,
    source_code: Bytes
}
```

Response :

- Success: Status code `201` with JSON of created version build.
- Fail: Status code according to the failure type.

B.2.2 Read

Read version based on id.

Method : GET

Endpoint : `/version/<version_id>`

Response :

- Success: JSON of version entity.
- Fail: Status code according to the failure type.

B.2.3 Update

Update version information.

Method : PATCH

Endpoint : `/version/<version_id>`

Request body : JSON

```
{
    version: Semantic version,
    phases: Vec<ApplicationPhase>,
    conf: Option<String>,
    notes: Option<String>,
}
```

Response :

- Success: JSON of the updated entity.
- Fail: Status code according to the failure type.

B.2.4 Delete

Delete version based on id.

Method : DELETE

Endpoint : `/version/<version_id>`

Response :

- Success: Status code `200`.
- Fail: Status code according to the failure type.

B.3 Version build

B.3.1 Read all

Read all version builds.

Method : GET

Endpoint : `/version_build`

Response :

- Success: List of all version build entities.
- Fail: Status code according to the failure type.

Appendix C

Evaluator API

C.1 Build success

This endpoint is called when compilation is successfull.

Method : POST

Endpoint : `/build_success/<version_id>`

Request body : JSON

```
{
    source_code: Bytes,
    binary: Bytes,
    logs: Bytes
}
```

Response :

- Success: Status code `200`.
- Fail: Status code according to the failure type.

C.2 Build fail

This endpoint is called when compilation has failed.

Method : POST

Endpoint : `/build_fail/<version_id>`

Request body : JSON

```
{
    source_code: Bytes,
    logs: Bytes
}
```

Response :

- Success: Status code `200`.
- Fail: Status code according to the failure type.

C.3 Build crash

This endpoint is called when compilation has crashed for any reason

Method : POST

Endpoint : `/build_crash/<version_id>`

Request body : JSON

```
{
    source_code: Bytes,
    logs: Option<Bytes>
}
```

Response :

- Success: Status code `200`.
- Fail: Status code according to the failure type.

Bibliography

1. SHI, Weisong; PALLIS, George; XU, Zhiwei. Edge Computing [Scanning the Issue]. *Proceedings of the IEEE*. 2019, vol. 107, no. 8, pp. 1474–1481. ISBN 0018-9219.
2. HONG, Cheol-Ho; VARGHESE, Blesson. Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms. *ACM Comput. Surv.* 2019, vol. 52, no. 5. ISSN 0360-0300. Available from DOI: 10.1145/3326066.
3. GÖKÇAY, Erhan. A new multi-target compiler architecture for edge-devices and cloud management. *Gazi University Journal of Science* [online]. 2022, vol. 35, no. 2, pp. 464–483 [visited on 2024-02-16].
4. MAARTEN VAN STEEN, Andrew S. Tanenbaum. *Distributed systems*. Published By Maarten Van Steen, 2020. ISBN 9781543057386.
5. TAKADA, Mikito. *Distributed systems for fun and profit* [online]. 2024. [visited on 2023-11-19]. Available from: https://book.mixu.net/distsys/.
6. MATSUDAIRA, Kate. *The Architecture of Open Source Applications (Volume 2) Scalable Web Architecture and Distributed Systems* [online]. 2024. [visited on 2024-02-22]. Available from: https://aosabook.org/en/v2/distsys.html.
7. MUSTAFEE, Navonil. Exploiting grid computing, desktop grids and cloud computing for e-science. *Transforming Government: People, Process and Policy*. 2010, vol. 4, no. 4, pp. 288–298. Available from DOI: https://doi.org/10.1108/17506161011081291.
8. MELL, Peter M; GRANCE, Timothy. The NIST definition of cloud computing. 2011. Available from DOI: https://doi.org/10.6028/nist.sp.800-145.

9. BUYYA, Rajkumar; BROBERG, James; GOSCINSKI, Andrzej. *Cloud computing: principles and paradigms*. Vol. 81. 1st ed. Newark: WILEY, 2010;2011; ISBN 9780470887998.

10. ARMBRUST, Michael; FOX, Armando; GRIFFITH, Rean; JOSEPH, Anthony D.; KATZ, Randy H.; KONWINSKI, Andrew; LEE, Gunho; PATTERSON, David A.; RABKIN, Ariel; STOICA, Ion; ZAHARIA, Matei. *Above the Clouds: A Berkeley View of Cloud Computing*. 2009-02. Tech. rep., UCB/EECS-2009-28. EECS Department, University of California, Berkeley. Available also from: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

11. *What is the network edge?* [online]. 2022. [visited on 2023-12-22]. Available from: `https://www.fortinet.com/resources/cyberglossary/network-edge#:~:text=The%20network%20edge%20the%20connection,administrators%20must%20provide%20solutions%20for.`.

12. MARCHAM, Alex. *Understanding Infrastructure Edge Computing: Concepts, Technologies, and Considerations*. 1st ed. Newark: John Wiley & Sons, Incorporated, 2021. ISBN 1119763231;9781119763239;

13. *What is NGINX* [online]. 2023-06. [visited on 2023-03-10]. Available from: `https://www.nginx.com/resources/glossary/nginx/`.

14. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly Specification Release 2.0 (Draft 2024-03-05* [online]. Ed. by ROSSBERG, Andreas. 2024. [visited on 2024-03-05]. Available from: `https://webassembly.github.io/spec/core/index.html`.

15. *Understanding WebAssembly text format* [online]. 2024-04. [visited on 2024-04-27]. Available from: `https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format`.

16. DORONIN, Oleg; DERGUN, Karina; LOGINOV, Ivan; KORENKOV, Iurii; DERGACHEV, Andrey. PROBLEM RESEARCH AND DEVELOPMENT OF A TOOL FOR CHECKING APPLICATION BINARY INTERFACE COMPATIBILITY OF VIRTUAL METHOD TABLES. In: Sofia: Surveying Geology & Mining Ecology Management (SGEM), 2019, vol. 19, chap. 2.1, pp. 531–537. ISBN 1314-2704.

17. WASI SUBGROUP. *WASI introduction* [online]. 2024. [visited on 2024-03-05]. Available from: `https://wasi.dev/`.

18. WASI SUBGROUP. *WASI gihub* [online]. 2024. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/WASI`.

19. WASI SUBGROUP. *WASI interfaces* [online]. 2024. [visited on 2024-03-05]. Available from: `https://wasi.dev/interfaces`.

20. BYTECODE ALLIANCE. *The WebAssembly Component Model* [online]. 2024. [visited on 2024-03-05]. Available from: `https://component-model.bytecodealliance.org`.

21. WEBASSEMBLY COMMUNITY GROUP. *WASI I/O* [online]. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/wasi-io`.

22. WEBASSEMBLY COMMUNITY GROUP. *WASI Clocks* [online]. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/wasi-clocks`.

23. WEBASSEMBLY COMMUNITY GROUP. *WASI Random* [online]. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/wasi-random`.

24. WEBASSEMBLY COMMUNITY GROUP. *WASI Filesystem* [online]. 2024. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/wasi-filesystem`.

25. WEBASSEMBLY COMMUNITY GROUP. *WASI Sockets* [online]. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/wasi-sockets`.

26. WEBASSEMBLY COMMUNITY GROUP. *WASI CLI* [online]. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/wasi-cli`.

27. WEBASSEMBLY COMMUNITY GROUP. *WASI HTTP* [online]. [visited on 2024-03-05]. Available from: `https://github.com/WebAssembly/wasi-http`.

28. BYTECODE ALLIANCE. *Wasmtime documentation* [online]. 2023. [visited on 2024-03-05]. Available from: `https://docs.wasmtime.dev`.

29. PROXY-WASM CONTRIBUTORS. *WebAssembly for Proxies (ABI specification)* [online]. 2024. [visited on 2024-03-05]. Available from: `https://github.com/proxy-wasm/spec?tab=readme-ov-file`.

30. PELLEGRINO, Giancarlo; BALZAROTTI, Davide; WINTER, Stefan; SURI, Neeraj. In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 801–816. ISBN 978-1-939133-11-3. Available also from: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pellegrino`.

31. MAASS, Michael; SALES, Adam; CHUNG, Benjamin; SUNSHINE, Joshua. A systematic analysis of the science of sandboxing. *PeerJ Computer Science*. 2016, vol. 2, e43. Available from DOI: `10.7717/peerj-cs.43`.

32. AL AMEIRI, Faisal; SALAH, Khaled. Evaluation of popular application sandboxing. In: *2011 International Conference for Internet Technology and Secured Transactions*. 2011, pp. 358–362.

33. NAIR, Ravi; SMITH, Jim. *Virtual Machines*. Morgan Kaufmann, 2005. ISBN 1558609105;9781558609105;

34. *What is virtual machine manager (VMM)?* [online]. 2024-01. [visited on 2024-03-17]. Available from: `https://www.infoblox.com/glossary/virtual-machine-manager-vmm/`.

35. *Kernel Virtual Machine* [online]. 2023. [visited on 2024-03-17]. Available from: `https://linux-kvm.org/page/Main_Page`.

36. RED HAT, INC. *What is KVM?* [online]. 2024. [visited on 2024-03-17]. Available from: `https://www.redhat.com/en/topics/virtualization/what-is-KVM`.

37. AMAZON WEB SERVICES. *Firecracker official website* [online]. 2014. [visited on 2024-03-17]. Available from: `https://firecracker-microvm.github.io/`.

38. FIRECRACKER-MICROVM. *Firecracker Design* [online]. 2017. [visited on 2024-04-27]. Available from: `https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md`.

39. WEISSMAN, Zane; TIEMANN, Thore; EISENBARTH, Thomas; SUNAR, Berk. *Microarchitectural Security of AWS Firecracker VMM for Serverless Cloud Platforms* [online]. 2023. [visited on 2024-03-17]. Available from arXiv: `2311.15999 [cs.CR]`.

40. MCCARTY, Scott. A Practical Introduction to Container Terminology. *Red Hat Developer*. 2018. Available also from: `https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#basic_vocabulary`.

41. *Introduction to Control Groups (Cgroups)* [online]. 2024. [visited on 2024-03-17]. Available from: `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01`.

42. *Container: privileged mode* [online]. 2021. [visited on 2024-03-17]. Available from: `https://learn.snyk.io/lesson/container-runs-in-privileged-mode/`.

43. *Docker overview* [online]. 2024. [visited on 2024-03-25]. Available from: `https://docs.docker.com/get-started/overview/`.

44. *Security* [online]. 2024. [visited on 2024-03-25]. Available from: `https://docs.docker.com/security/`.

45. *Run the Docker daemon as a non-root user (Rootless mode)* [online]. 2024. [visited on 2024-03-25]. Available from: `https://docs.docker.com/engine/security/rootless/`.

46. *Podman official documentation* [online]. 2019. [visited on 2024-03-25]. Available from: `https://docs.podman.io/en/latest/`.

47. *gVisor official website* [online]. 2024. [visited on 2024-03-25]. Available from: `https://gvisor.dev/`.

48. *gVisor official documentation* [online]. 2024. [visited on 2024-03-25]. Available from: `https://gvisor.dev/docs/`.

49. *V8 official website* [online]. 2024-04. [visited on 2024-03-25]. Available from: `https://v8.dev/`.

50. DAYA, Shahir; ORGANIZATION, International Business Machines Corporation International Technical Support. *Microservices from theory to practice : creating applications in IBM Bluemix using the microservices approach*. Poughkeepsie, Ny Ibm Corporation, International Technical Support Organization, 2015. ISBN 9780738440811.

51. *OpenResty offical web* [online]. 2024. [visited on 2024-02-10]. Available from: `https://openresty.org/en/`.

52. *Rust official website* [online]. 2018. [visited on 2024-02-10]. Available from: `https://www.rust-lang.org/`.

53. *Tokio tutorial* [online]. 2023. [visited on 2024-02-10]. Available from: `https://tokio.rs/tokio/tutorial`.

54. *Asynchronous Programming in Rust* [online]. 2024. [visited on 2024-02-10]. Available from: `https://rust-lang.github.io/async-book/08_ecosystem/00_chapter.html`.

55. *AWS SDK for Rust* [online]. 2024. [visited on 2024-02-10]. Available from: `https://docs.aws.amazon.com/sdk-for-rust/latest/dg/welcome.html`.

56. MINIO, Inc. *MinIO — S3 & Kubernetes Native Object Storage for AI* [online]. 2022. [visited on 2024-02-10]. Available from: `https://min.io/`.

57. LAUNCHBADGE. *sqlx github* [online]. 2019. [visited on 2024-02-10]. Available from: `https://github.com/launchbadge/sqlx`.

58. *PostgreSQL official documentation* [online]. 2024. [visited on 2024-02-10]. Available from: `https://www.postgresql.org/about/`.

59. ANJALI; CARAZA-HARTER, Tyler; SWIFT, Michael M. Blending containers and virtual machines: a study of firecracker and gVisor. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Lausanne, Switzerland: Association for Computing Machinery, [n.d.], pp. 101–113. ISBN 9781450375542. Available from DOI: `10.1145/3381052.3381315`.

60. *Trait tower::Service documentation* [online]. 2024. [visited on 2024-02-10]. Available from: `https://docs.rs/tower/latest/tower/trait.Service.html`.

61. *Hit the Ground Running: Wasm Snapshots for Fast Start Up* [online]. [visited on 2024-02-16]. Available from: https://fitzgeraldnick.com/2021/05/10/wasm-summit-2021.html.

62. STACKENÄS, William. *An Evaluation of WebAssembly Pre-Initialization for Faster Startup Times* [online]. 2023. [visited on 2024-02-16]. Available from: https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1763085&dswid=-4303.

Contents of the attachment

`thesis` ..... Thesis source code in LaTeX format
`thesis.pdf` ..... Thesis in PDF format