Daniel Dagomer Franco López

WEBASSEMBLY FOR EDGE-CLOUD COMPUTING

FINAL MASTER'S PROJECT

Directed by Dr. Marc Sánchez Artigas

Master's Degree in Computer Security Engineering and Artificial Intelligence

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2023

Contents

List of Tables

List of Figures

Abstract

WebAssembly (also known as Wasm) is a powerful technology by which application code can be executed at the machine level in web browsers. The new development of the WebAssembly System Interface (WASI) introduced by Mozilla has extended its capabilities beyond the browser, making it an attractive solution for cloud and edge computing scenarios.

WebAssembly is a low-level binary instruction format (Machine Level), which is used as a portable destination for high-level languages, for example: C, C++, Rust, among others. It was originally created to mitigate the performance limitations of traditional web browsers by providing an efficient and compact binary format that can run at near native speed. Such efficiency, combined with its platform and architecture independence, empowers Wasm to run applications that can reach high efficacy, reliability, and adaptability across diverse devices and contexts.

In addition, the flexible and extensible design of WebAssembly has motivated the development of several Wasm runtimes, for example, WasmEdge, Wasmtime or Wasmer, each offering unique features and optimizations. These runtimes play a key role in the performance and execution of Wasm applications, making them an essential research focus for this master's thesis.

One of WebAssembly's main applications is native cloud computing, where it offers a lightweight virtualization solution. By encapsulating applications in Wasm modules, programmers can achieve efficient and isolated execution, not requiring direct access to the underlying infrastructure. This feature is primarily advantageous for edge computing deployments, where resources are constrained, and applications need to be closer to end users to reduce latency and improve responsiveness.

Finally, Wasm offers a new possibility to the world of containers, making them lighter and faster to implement. Docker is a software platform that enables the creation and deployment of lightweight containers that isolate applications from the underlying system. Docker offers a new feature that allows run WebAssembly modules alongside Linux containers in Docker Desktop. This feature is currently in beta and requires enabling the "*containerd image store*" feature in Docker Desktop.

Resumen

WebAssembly (también conocido como Wasm) es una poderosa tecnología mediante la cual el código de la aplicación se puede ejecutar a nivel de máquina en navegadores web. El nuevo desarrollo de WebAssembly System Interface (WASI) introducido por Mozilla ha extendido sus capacidades más allá del navegador, convirtiéndolo en una solución atractiva para escenarios de computación en la nube y en el borde.

WebAssembly es un formato de instrucción binaria de bajo nivel (Nivel máquina), que se utiliza como destino portátil para lenguajes de alto nivel, por ejemplo: C, C++, Rust, entre otros. Fue creado originalmente para mitigar las limitaciones de rendimiento de los navegadores web tradicionales al proporcionar un formato binario eficiente y compacto que puede ejecutarse a una velocidad casi nativa. Dicha eficacia, combinada con su independencia de plataforma y arquitectura, permite a Wasm ejecutar aplicaciones que pueden alcanzar alta eficiencia, confiabilidad y adaptabilidad en diversos dispositivos y contextos.

Además, el diseño flexible y extensible de WebAssembly ha motivado el desarrollo de varios tiempos de ejecución de Wasm, por ejemplo, WasmEdge, Wasmtime o Wasmer, cada uno de los cuales ofrece características y optimizaciones únicas. Estos tiempos de ejecución desempeñan un papel clave en el rendimiento y la ejecución de aplicaciones Wasm, lo que los convierte en un foco de investigación esencial para esta tesis de maestría.

Una de las principales aplicaciones de WebAssembly es la computación en la nube nativa, donde ofrece una solución de virtualización liviana. Al encapsular aplicaciones en módulos Wasm, los programadores pueden lograr una ejecución eficiente y aislada, sin requerir acceso directo a la infraestructura subyacente. Esta característica es principalmente ventajosa para implementaciones de computación perimetral, donde los recursos son limitados y las aplicaciones deben estar más cerca de los usuarios finales para reducir la latencia y mejorar la capacidad de respuesta.

Finalmente, Wasm ofrece una nueva posibilidad al mundo de los contenedores, haciéndolos más ligeros y rápidos de implementar. Docker es una plataforma de software que permite la creación e implementación de contenedores livianos que aíslan las aplicaciones del sistema subyacente. Docker ofrece una nueva característica que permite ejecutar módulos WebAssembly junto con contenedores de Linux en Docker Desktop. Esta función se encuentra actualmente en versión beta y requiere habilitar la función "almacenamiento de imágenes en contenedores" en Docker Desktop.

Resum

WebAssembly (també conegut com Wasm) és una tecnologia potent mitjançant la qual el codi d'aplicació es pot executar a nivell de màquina en navegadors web. El nou desenvolupament de la WebAssembly System Interface (WASI) introduït per Mozilla ha ampliat les seves capacitats més enllà del navegador, convertint-lo en una solució atractiva per a escenaris de computació en núvol i de vora.

WebAssembly és un format d'instrucció binària de baix nivell (Machine Level), que s'utilitza com a destinació portàtil per a llenguatges d'alt nivell, per exemple: C, C++, Rust, entre d'altres. Originalment es va crear per mitigar les limitacions de rendiment dels navegadors web tradicionals proporcionant un format binari eficient i compacte que es pot executar a una velocitat gairebé nativa. Aquesta eficàcia, combinada amb la seva independència de plataforma i arquitectura, permet a Wasm executar aplicacions que poden assolir una alta eficiència, fiabilitat i adaptabilitat en diversos dispositius i contextos.

A més, el disseny flexible i extensible de WebAssembly ha motivat el desenvolupament de diversos temps d'execució de Wasm, per exemple, WasmEdge, Wasmtime o Wasmer, cadascun oferint característiques i optimitzacions úniques. Aquests temps d'execució tenen un paper clau en el rendiment i l'execució de les aplicacions Wasm, la qual cosa els converteix en un focus de recerca essencial per a aquest treball de fi de màster.

Una de les principals aplicacions de WebAssembly és la computació en núvol nativa, on ofereix una solució de virtualització lleugera. En encapsular aplicacions en mòduls Wasm, els programadors poden aconseguir una execució eficient i aïllada, sense necessitat d'accés directe a la infraestructura subjacent. Aquesta característica és principalment avantatjosa per als desplegaments informàtics de vora, on els recursos estan restringits i les aplicacions han d'estar més a prop dels usuaris finals per reduir la latència i millorar la capacitat de resposta.

Finalment, Wasm ofereix una nova possibilitat al món dels contenidors, fent-los més lleugers i ràpids d'implementar. Docker és una plataforma de programari que permet la creació i el desplegament de contenidors lleugers que aïllen les aplicacions del sistema subjacent. Docker ofereix una nova funció que permet executar mòduls WebAssembly juntament amb contenidors Linux a Docker Desktop. Aquesta funció es troba actualment en versió beta i requereix activar la funció "containerd image store" a Docker Desktop.

1. Introduction

Edge computing is a paradigm that allows the processing and analysis of information closer to its origins and consumers, reducing latency, bandwidth consumption and privacy risks. However, this concept presents various challenges, such as heterogeneity, resource limitations, security, and scalability [1]. To address these challenges, there is a need for a common platform that can run applications through different new generation devices and cloud environments in a fast, secure, and portable way.

WebAssembly (Wasm) is a binary instruction format that can be executed by various runtimes on different platforms [2]. Wasm offers various advantages for edge computing, such as high performance, low footprint, sandboxing, and interoperability. Wasm can also support multiple programming languages and frameworks, enabling developers to write applications in their preferred tools and programming language.

Despite this, Wasm is not yet a popularly adopted option in edge computing scenarios, presenting certain limitations and open issues that need to be addressed. For example, Wasm does not have native support for network communications, asynchronous operations, or hardware access [2]. Also, Wasm runtimes vary in their features and compatibility with different platforms and standards. There is a lack of benchmarks and evaluations that compare the performance and efficiency of Wasm applications based on their runtimes, those that are currently being maintained.

The main objective of this master's final project (MFP) is to investigate the potential and challenges of using Wasm for edge and cloud computing. This MFP tries to answer the following questions:

- What is Wasm?
- What is the current state of technology?
- What is a Wasm runtime?
- Which is the best option for future developments?

To answer these questions, this MFP will conduct a literature review on Wasm and its applications in edge-cloud computing. Several use cases involving diverse types of data and applications will then be designed and implemented using Wasm. Finally, the results will be evaluated and compared using various metrics and criteria.

This project is conducted in the context of Horizon Europe's CLOUDSKIN research project, which aims to develop an AI-enabled cloud edge framework that dynamically adapts to changes in application behavior and variability of data. CLOUDSKIN's platform seamlessly and securely integrates various computing and data environments, from the cloud core to the edge. This MFP is intended to be a starting point in creating a common execution platform for cloud applications based on or supported by Wasm.

1.1. Context

Nowadays, the number of edge devices and the information they generate has grown exponentially, due to the rise of the Internet of things (IoT) and the growth of the area affected by wireless networks.

"According to International Data Corporation (IDC) prediction [20], global data will reach 180 zettabytes (ZB), and 70% of the data generated by IoT will be processed on the edge of the network by 2025." [3]

The current centralized, cloud-based model, in which all information is sent to supercomputers for processing, is inefficient. Here are the main disadvantages of this:

Disadvantage	Description
Latency	Today's most modern edge applications require immediate responses, sending information to cloud servers increases latency.
Bandwidth	Current infrastructures do not support sending the substantial amounts of information generated by modern edge devices.
Availability	Distinct reasons may affect availability of cloud services, for example: lack of internet connection, server downtime, among others.
Energy	Data centers consume a lot of energy.
Security and Privacy	Information sent from edge devices over the internet is always at risk of being intercepted by malicious users.

Table 1: Disadvantages of cloud computing [3].

"Edge computing is a new paradigm in which the resources of an edge server are placed at the edge of the Internet, in close proximity to mobile devices, sensors, end users, and the emerging IoT." [3]

Edge and cloud computing are complementary models. The advantage of edge computing is that to be independent, it processes temporary information without uploading it to the cloud completely, reducing the use of network bandwidth. In addition, its location helps reduce system latency and improves service availability, not requiring a response from a cloud server. Finally, it improves the security of the system by avoiding the sending of unnecessary information.

Edge computing has been developing rapidly since 2014. Currently, it is in a stable development period. There are various challenges arising related to various modern application scenarios, including public safety, autonomous driving, deep learning, wireless communication, and edge intelligence. One of the main challenges is related to the use of security measures due to the resource limitations of edge devices, leaving them vulnerable to cyber and physical attacks.

Edge computing aims to unify IoT, big data and mobile computing, in an integrated and ubiquitous platform. The ability to deliver computing power on demand and the ability to process copious amounts of information is highly valued within the field of artificial intelligence (AI) [3].

WebAssembly emerges as a terrific opportunity to address the benefits that edge computing offers, allowing applications to run the code efficiently, quickly, and securely at near native speed, across different platforms and environments, abstracting from the underlying architecture.

1.2. Motivation

WebAssembly presents a lightweight and flexible way to pass processing from edge applications, taking advantage of serverless computing with limited memory, CPU, and storage resources, added to its portability and low power consumption. However, there are few works that analyze and compare the different WebAssembly runtimes, as well as the frameworks that can manage or integrate them.

In addition, there is a lack of studies that evaluate the performance, scalability and cost efficiency of modern applications oriented to the implementation of algorithms related to artificial intelligence, using WebAssembly.

Therefore, this MFP aims to analyze the bases for the implementation of a new execution abstraction based on WebAssembly for cloud and edge computing, using tools such as Docker Desktop as well as to analyze the different existing Wasm runtimes and that currently present maintenance by comparing them following different metrics.

Many modern technology experts agree that WebAssembly is crucial and will play a huge role in the next generation of cloud computing. In 2019 Solomon Hykes, creator of Docker, commented on a well-known social network that if Wasm and WASI had existed in 2008, it would not have been necessary to create Docker [4].

Figure 1: Solomon Hykes tweet about Wasm Docker.

Due to the widespread use of Docker today, it is ridiculously hard to imagine a world without it. However, Wasm is considered a perfect complement to work with, in fact, there is currently a feature under development whereby Wasm workloads can be run using Docker Desktop, replacing traditional Linux-based images [5]. (View appendix A)

This work aims to contribute to the advancement of scientific knowledge about WebAssembly and its application to edge computing, as well as to provide a practical guide for the development and implementation of applications based on this technology.

1.3. Objectives

The main objective of this project is to develop and evaluate different algorithms related to machine learning that take advantage of WebAssembly to execute functions in different runtimes.

The specific objectives are:

- Acquire fundamental knowledge of WebAssembly and its build toolchain by reviewing existing literature and documentation.
- Learn how to use various runtimes and extensions for WebAssembly by implementing and testing distinct functions in each of them.
- Design and implement different algorithms that integrate WebAssembly with Docker Desktop as a management system and orchestrator.
- Evaluate the performance, scalability, and reliability of the proposed platform by conducting experimental tests using different workloads and scenarios.
- Compare the results obtained for each of the scenarios proposed by applying appropriate metrics and criteria.

2. State of the art

In this section, Wasm and WASI will be reviewed in depth. It will also study the main Wasm runtimes, reviewing their characteristics, advantages, and disadvantages. There will also be an introduction to the world of containers and Docker and its integration with Wasm.

2.1. WebAssembly

WebAssembly emerges in December 2019 as a browser-based technology introduced by Mozilla. WebAssembly is a universal bytecode for interoperable compute units. Interoperable means that the compute unit should work in any Wasm runtime. A compute unit is a Wasm module [6]. Being a universal bytecode allows Wasm abstract it from the target architecture. The latest version of the standard Wasm core is 2.0 (Draft 2023-07-24). The latest version of WebAssembly is 1.1, which was published as W3C Recommendation on December 22, 2020.

There are various high-level programming languages that can be compiled in a Wasm-based bytecode. When it obtains the bytecode, Wasm runtime can execute it. Below is a table with the details of the main programming languages and their support in the integration with Wasm.

Language	Core	Browser	WASI
JavaScript	Usable	Usable	Usable
Python	Usable	In progress	Usable
Java	Usable	Usable	Usable
Ruby	Usable	Usable	Usable
C# and .NET	Usable	Usable	Usable
C++	Usable	Usable	Usable
Swift	Usable	Usable	Usable
Go	Usable	Usable	Usable
Rust	Usable	Usable	Usable

Table 2: Programming languages and their Wasm integration [7].

- Core: Implementation of WebAssembly 1.0.
- Browser: Browser implementation.
- WASI: Supports WASI proposal.

Wasm is based on a broad industry collaborative effort. The Bytecode Alliance is a non-profit organization dedicated to creating new secure software foundations, based on standards such as WebAssembly and WebAssembly System Interface (WASI) [8].

With its low resource consumption and fast startup time, Wasm can be provisioned on devices with limited memory, CPU, and storage, making it ideal for cloud and edge serverless deployments. Due to its sandboxing capability and low overhead, it is ideal for preventing security vulnerabilities such as buffer overflows and control flow integrity issues. Wasm has a static type system, where it separates code and data, as well as a well-structured control flow [6].

Wasm modules do not have access to the API's or system calls in the OS. The WebAssembly System Interface (WASI) allows Wasm to interact with the operating system, this specification makes possible interoperable Wasm code that can be ported to any Wasm runtime once the Wasm compiler generates the bytecode [6].

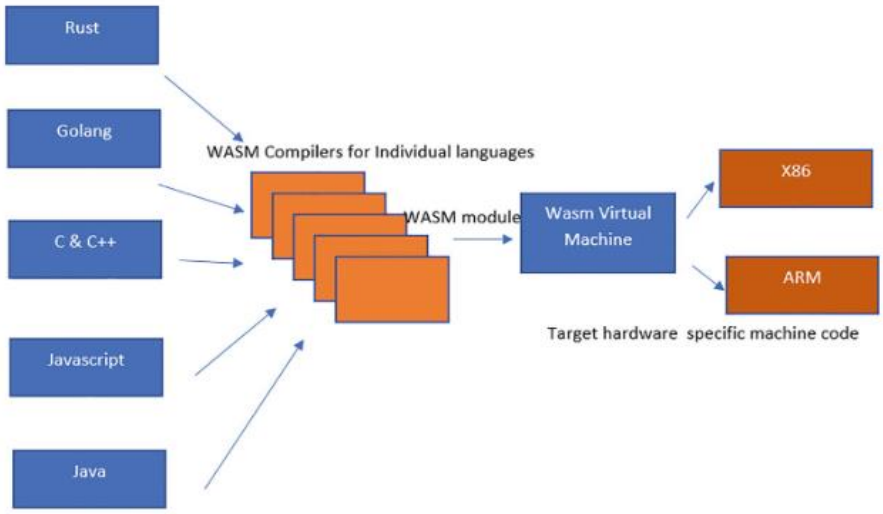

Figure 2: Elevated level Wasm architecture. [6]

2.1.1. Design goals

Wasm pursues the following goals with its design:

- Fast: Runs at near native performance.
- Secure: its secure memory and sandboxed environment prevents data corruption or security breaches.
- Well-defined: defines in a complete and precise way, valid programs, and their behavior.
- Hardware-independent: it is suitable to compile on all modern architectures.
- Language-independent: Major programming languages can compile to Wasm.

- Platform-independent: it can work inside browsers, as its own virtual machine, or with other environments.
- Open: programs can interact with their environment in a simple and universal way.
- Compact: The binary format is smaller than traditional text or native code formats.
- Modular: Programs can be streamed, cached, and consumed separately.
- Efficient: Programs can be decoded, validated, and compiled in one quick step.
- Streamable: allows priority decoding, validation, and compilation.
- Parallelizable: allows decoding, validation, and compilation breaking down into several independent parallel tasks.
- Portable: It is abstracted from the underlying architecture through modern hardware [9].

2.1.2. WebAssembly key objects

The following are the eight objects that are key to WebAssembly [6]:

WebAssembly.Module, it contains the stateless WASM code. The code is precompiled.

WebAssemly.Global, it is accessible by both WebAssembly.Module and JavaScript.

WebAssembly.Instance, it is a stateful executable instance of WebAssembly.Module.

WebAssembly.Memory, it is a dynamic buffer of either ArrayBuffer or SharedArrayBuffer type that stores unstructured binary data, which can be manipulated by a WebAssembly.Instance object.

WebAssembly.Table, it contains references to functions and acts as a JavaScript wrapper.

WebAssembly.CompileError, it contains all errors during validation and decoding.

WebAssembly.LinkError, it indicates an error during a module instantiation.

WebAssembly.RuntimeError, it lists all runtime errors.

2.1.3. WebAssembly specification

The core specification defines the syntax and semantics of Wasm modules. Wasm is a low-level language, structured around the following concepts:

- Wasm provides four types of values: i32, i64, f32 and f64, which correspond to 32-bit and 64-bit integers and floating-point numbers, respectively. It also defines two basic reference types: funcref and externref, corresponding to references to functions and external values, respectively.
- Instructions that manipulate values fall into two main categories, Simple and Control instructions. Simple ones perform basic operations on data. The control ones alter the flow of control.

- Wasm has two kinds of structures: linear memory and table. Linear memory is a contiguous array of bytes accessible by memory operators. A table is a reference array accessible by table operators. Structures have dynamic size and can grow at runtime.
- Wasm has four kinds of components: functions, globals, memories, and tables. Functions are code units that can be invoked by call instructions. Global operators can access to globals variables. Memories and tables are structures that can be accessed by memory and table operators. Each component has a type that describes its signature or properties.
- There are two kinds of sections used by Wasm: custom sections and module sections. Custom sections are optional sections that can contain random data for extensions or annotations. Module sections are mandatory sections that define the components, entities, and code of a Wasm module.
- Under certain conditions, some instructions can cause a trap, which immediately aborts execution.

Wasm semantics is divided into three phases. Namely:

- Decoding. Process that converts the format into an internal representation of a module, through an abstract syntax or directly to machine code.
- Validation. Process that checks the conditions to ensure that the module is safe.
- Execution. The last step is divided into two stages:
    - Instantiation. This stage executes the body of the module, initializing globals, memories and table and calls the start function of the module.
    - Invocation. It starts the Wasm computations, executing the respective functions and returning the results.

  The execution occurs in the embedded environment [45].

2.1.4. Use-cases

Among the main use cases of Wasm both inside and outside the browser are, video and audio editing, gaming in the web browser, visualization and scientific simulation, encryption, among others. There are several success stories using WebAssembly to improve the performance, functionality, and user experience of your web applications. Among others, we have:

- Google earth, a web application from Google that allows you to explore the world in 3D using WebAssembly to render the graphics and manage the geospatial data.
- AutoCAD, an Autodesk web application that allows you to create and edit 2D and 3D designs using WebAssembly to port native AutoCAD code to the web.
- Blazor Microsoft, technology that makes it possible to create web applications using C#. This uses Wasm to improve performance directly in the browser.
- Docker, an open-source project that automates the deployment of applications within software containers, providing an additional layer of abstraction and automation of application

virtualization across multiple operating systems. It uses Wasm for presenting a fast, light alternative to the traditional Linux and Windows containers.

2.2. Wasm runtimes

The Wasm runtime is a virtual machine that uses a stack data structure. It executes the Wasm bytecode by pushing data onto the stack and popping data off the stack. Different runtimes can implement the specification WASI.

When Wasm runs in the browser, the interface with the operating system is overseen by the browser. For servers or edge applications, this is facilitated by the runtime hosting the Wasm module. The types of calls to the system would be like a system or network I/O [6].

There are terms that are related to Wasm runtimes, among the main ones the following are defined:

- Single instruction multiple data (SIMD). It refers to a technique that allows a processor to perform the same operation on multiple data elements in parallel.
- Parallelization and vectorization, which are not related to the runtime but to the compiler, are techniques used to improve the performance of a program by exploiting the level of parallelism.
- Backend, it is a term that refers to the part of a Wasm engine that is responsible for compiling the modules into native code that can be executed by the hardware. The backend can use different techniques to generate the code, such as:
    - Just-in-time (JIT) compilation: Compile the Wasm module on the fly, while they are loaded or executed. This technique can achieve fast startup and adaptive optimization but may incur runtime overhead and memory consumption.
    - Ahead-of-time (AOT) compilation: Compiles the Wasm module before it is loaded or executed. It achieves high performance and low memory consumption; on the other hand, it can generate longer compilation time.
    - Interpretation: This technique executes the Wasm module without transforming it into native code. It works by processing the binary format of a Wasm module and converting each instruction into a corresponding operation in the host environment, which enables portability and simplicity, but also results in reduced performance and increased memory usage.
    - Hybrid: It is the combination of one or more of the techniques mentioned above to balance the trade-offs between them.

There are currently several Wasm runtimes available, each with distinct characteristics and use cases, the main ones are detailed below.

2.2.1. WasmEdge

One of the most popular and mature runtimes today, focused on edge and cloud computing and decentralized applications. WasmEdge is an official sandbox project hosted by CNCF (Cloud Native Computing Foundation). The key features of WasmEdge are:

- High performance. Taking advantage of its LLVM-based compiler. WasmEdge can switch between AOT or JIT compilations depending on the configuration or the environment.
- Native cloud extensions. For example: non-blocking network sockets, web services with Rust, C and JavaScript SDK, MySQL-based database driver, AI inference with TensorFlow, Pytorch and OpenVINO.
- JavaScript support, through the WasmEdge-Quickjs project.
- Cloud native management and orchestration, using an OCI runtime crun (The C version of runc – mostly used by Red Hat) or using a containerd-shim to start Wasm containers (Docker desktop) via runwasi.
- Cross platform. It supports a wide range of operating systems and hardware platforms.
- Supports exception handling, multithreading, and SIMD as experimental features.
- WasmEdge also supports two levels of optimization: O0 and O3. For our example we have used the third level O3 that is the highest level, an aggressive optimization.
- Easily extensible and easy to embed in a host application, thanks to its integration with the main programming languages [10].

Its main applications are oriented to web with architectures based on the cloud and the edge, together with React or Vue, it supports rendering functions on edge servers. It is compatible with orchestrator services such as Kubernetes, supporting stateless and stateful business logic.

2.2.2. Wasmtime

Wasmtime is a Bytecode Alliance project. It is a fast and secure runtime that aims to create a versatile and powerful platform for running Wasm. Its primary features are:

- Fast. It is built on Cranelift, a JIT redirectable low-level code generator, to quickly generate high-quality machine code at runtime.
- Secure. Its focus is on correctness and security. Built on Rust's runtime security guarantees, each feature is reviewed considering the Request for Comments (RFC) process, a tool for obtaining feedback on design and implementation. There is a thorough process to validate the versions that are released.
- Configurable. Resources such as CPU and memory consumption can be configured, depending on the characteristics of the host environment.
- WASI. The interface allows integration with the host environment.
- Supports exception handling, Multithreading and SIMD as experimental features.

- Standards compliance. Wasmtime not only passes the official WebAssembly test suite, but it also implements the official Wasm C API, implements future proposals for Wasm, and commits to its standard process [12].

Some practical cases that can be mentioned are Kubernetes, Suborbital, a platform to build web services with wasm, or Compute@edge, a platform that compiles custom code to Wasm and executes it using WASI for each compute request. Wasmtime can support smart contracts for blockchain platforms, OpenEthereum is an example, client running Wasm contract.

2.2.3. Wasmer

Wasmer is a current and well-maintained runtime Wasm. It is secure, fast, and pluggable, running on any platform and almost any chipset. The major features are detailed below:

- Secure. No access to files, network, or environment, by default.
- Fast. Runs Wasm at near native speed.
- Pluggable. Embedded in multiple programming languages. You can use various backends depending on your needs, for example: Singlepass, Cranelift or LLVM (Native). It uses a JIT compiler.
- Caching. Compiled Wasm modules can be reused, speeding up the start of subsequent executions.
- Supports exception handling, Multithreading and SIMD as experimental features.
- Metering. Resource consumption can be monitored and limited (Gas metering).

Wasmer has significant use cases today. For example, Meta (Facebook), runs untrusted code from other developers in a safe and isolated environment. Another use case is Vercel, a complete cloud frontend for a more personalized web, Wasmer provides it with an interface for fast and secure middleware and edge functions in a multitude of languages [14].

2.2.4. GraalWasm

GraalWasm is a Wasm engine implemented in GraalVM, a program that was originally created to compile ahead of time (AOT) java applications, improving the overall performance of applications running on the Java Virtual machine (JVM).

GraalWasm implements the WebAssembly MVP (Minimum Viable Product) specification, the initial release of the Wasm API that was agreed upon by the WebAssembly Community Group (CG), which aims to provide the basic functionality and performance of Wasm that can be implemented by environments execution. It uses GraalVM as its backend with JIT compilation capabilities.

GraalWasm is in the initial stages of its development, but the idea is to increase the set of languages GraalVM can run and support Wasm extensions in the future.

Wasm support is not available by default, but it can be added to GraalVM using the GraalVM Updater tool. This allows programs written in the language that was compiled to Wasm to be executed in GraalVM and then compile using the latest version of the Emscripten compiler frontend (used to call the Emscripten compiler to transform high level programming language into Wasm), which produces a standalone .wasm file. The Wasm binary, once compiled, can be manipulated programmatically using the GraalVM Polyglot API, which enables the integration of GraalVM WebAssembly with user applications [18].

One of the main advantages of Wasm is that it can benefit from the interoperability of the GraalVM ecosystem, which can integrate with different languages and platforms. At this moment, GraalWasm does not support WASI.

GraalVM has several use cases today, as a platform, on Facebook it is used as a Java runtime to accelerate Spark workloads by reducing memory and CPU. It is also used by Twitter as a JIT compiler for the JVM to run its Tweet service, among other uses. However, the GraalWasm component, being a test version, does not have the same recognition as its parent framework.

2.2.5. Wasm3

This project was released under The MIT License (MIT). It is a fast and universal Wasm player. Its key features are:

- Import/export of global mutable with structured execution tracking.
- Support to multi value and gas metering.
- Bulk memory operations: mem.copy and mem.fill. Linear memory limit (<64KiB).
- It does not support WASI.

It uses an interpreter instead of a compiler, it is one of the fastest interpreters for Wasm code, but still slower than compiled ones.

Wasm3 has not received an update for a long time and no use cases have been found that implement Wasm3 in any way at present [19].

2.2.6. WAVM

It is a WebAssembly virtual machine that uses LLVM to achieve high performance, security, and portability. It can compile WebAssembly code to machine code that is optimized for the specific CPU and execute it with minimal overhead. It also isolates WebAssembly code from accessing or calling unauthorized native code but warns of potential side-channel attacks. WAVM is written in C/C++ and supports various platforms, especially X86-64 systems. It has some limitations on AArch64 and 32-bit systems [20].

WAVM fully supports WebAssembly 1.0, in addition many proposed extensions to it:

- WASI.
- 128-bit SIMD.

- Threads.
- Reference types.
- Multiple results and block parameters.
- Bulk memory operations.
- Non-trapping float-to-int conversions.
- Sign-extension instructions.
- Exception handling.
- Extended name section.
- Multiple memories.

WAVM has not received an update for some time and no use cases have been found that implement WAVM in any way at present.

2.2.7. Conclusion

The latest version of each of the Wasm runtimes that were mentioned above is detailed below in a table, in this way it can be determined which are currently receiving the most maintenance:

Runtime	Version	Date
WasmEdge	0.13.3	2023-07-25
Wasmtime	11.0.1	2022-09-22
Wasmer	4.1.1	2023-08-03
GraalWasm	22.3.3	2023-07-25
Wasm3	0.5.0	2021-06-02
WAVM	0.0.7	2021-03-29

Table 3: Wasm runtimes version.

In general, thanks to the general bases that the generated compiled bytecode has, most high-level programming languages have an integration covered for Wasm, which is detailed below in a table, in which almost no difference can be seen:

Runtime	Rust	C/C++	Python	.NET	GO	SWIFT	Java
WasmEdge	✓	✓	✓	✓	✓	✓	✓
Wasmtime	✓	✓	✓	✓	✓	×	✓
Wasmer	✓	✓	✓	✓	✓	×	✓
GraalWasm	✓	✓	×	×	✓	×	×
Wasm3	✓	✓	✓	✓	✓	✓	✓
WAVM	✓	✓	✓	✓	✓	✓	✓

Table 4: Programming languages Wasm integration.

Something similar happens with the target platforms. The following table details the integration of the Wasm runtimes with the main existing platforms:

Runtime	Windows	Linux	MacOS	Android	IOS
WasmEdge	✓	✓	✓	✓	✓
Wasmtime	✓	✓	✓	✓	✓
Wasmer	✓	✓	✓	✓	✓
GraalWasm	✓	✓	✓	×	×
Wasm3	✓	✓	✓	✓	✓
WAVM	✓	✓	✓	×	×

Table 5: Platforms Wasm integration.

Based on the literature investigated for this work, it is determined that the most important current Wasm runtimes are WasmEdge, Wasmtime and Wasmer. While the other three have been included, GraalWasm is in an early stage of development and not extremely popular compared to its competitors, wasm3 and WAVM have not been maintained for a long time, so it is understood that they have been left behind in reference to the others mentioned. It is worth mentioning that there are several Wasm runtimes that have been omitted in this work, but it was concluded that those that have been included cover all the characteristics that are intended to be managed by each of them.

The main disadvantage for Wasm runtimes in general is that the technology is new and in its prime, so they are still under development, and some might not support all WebAssembly features and extensions, such as threads, SIMD (Single Instruction Multiple Data), reference types, among others.

The main Wasm runtimes particularly offers a JIT compilation strategy, which optimizes execution speed by compiling Wasm at runtime. This can be a disadvantage in incurring some overhead at startup time compared to AOT compilation. Regarding the security scope, this could limit some capabilities or access rights that some Wasm modules might need from the host system.

2.2.8. Spin

It is a framework for building and running event-driven microservice applications with WebAssembly components. Spin is open source and built on standards. There are implementations for local development, self-hosted servers, Docker, and cloud-hosted services [21]. It has been decided to include this framework, regardless of not being a runtime, this simulates its functionality, it is currently being widely recognized and its maintenance has not stopped since its creation.

Spin has been developed under the Apache-2.0 license; the latest version released was v1.4.1 dated July 12, 2023. Spin allows us to use a wide variety of main programming languages, such as: Python, Rust, Go

or JavaScript. It allows applications to be deployed in its private cloud, Fermyon Cloud, a platform that natively supports Wasm and offers high performance and scalability.

Spin provides a template library that contains various starter templates for different use cases and languages. Spin being an open-source project, users can create their own templates and share them with others. It also offers data services that can be used to store and return information from various sources, such as PostgreSQL, Redis, or file storage. It is extremely fast and lightweight because of its LLVM JIT compiler and custom memory allocator.

It does not support WASI so it can only perform pure computational tasks that do not require interaction with the host system.

2.3. Related work

Our work is related to WebAssembly performance measurement and studies [42, 46, 47, 48, 49, 50, 51].

Wasm runtimes benchmarks. [42] is one of the most popular web benchmarks that evaluated the different runtimes that existed up to that time. [48] is a benchmark for evaluating the performance of Wasm runtimes on server-side applications. Compared to them, our study evaluates new runtimes such as: WasmEdge (Second State in [42]), GraalWasm and Spin.

Wasm and native code comparison. [46] is a document focused on performance comparison between WebAssembly and native code in the browser. It must be considered that this study is one of the oldest of those mentioned and is oriented at a lower level, in contrast to our work, which is more oriented towards final applications.

Wasm and JavaScript comparison. [49] measured the Wasm performance and JavaScript in the browser. [50] evaluated performance and portability between Wasm and JavaScript. Unlike them, our work is not aimed at comparing Wasm with other programming languages.

Wasm and containers studies. [47] measured the performance of Wasm with native code and Docker. [51] studied a way for extending Kubernetes, allowing it to orchestrate natively executed WebAssembly modules, in addition it evaluates the performance of the proposed solution. By contrast, our work uses an official container solution extension to evaluate the execution performance of the proposed examples. However, the results obtained are similar in both cases.

Our work attempts to update the related concepts in this field, due to its high dynamism. To the best of our knowledge, our work is the first to evaluate Spin's features and performance in the context of main Wasm runtimes, as well as Wasm's functionality integrated with an industrial product like Docker.

3. Analysis and design of the implementation

This section will discuss the design choices, implementation technologies, and metrics that we will use to evaluate the performance of each of the proposed examples.

3.1. Algorithms

Next, we will proceed to detail the algorithms that will be implemented, as well as the code written in Rust for its execution.

3.1.1. K-means

K-means is a popular unsupervised machine learning algorithm that partitions a set of data points into k groups, where k is a predefined number. The algorithm aims to minimize the within-cluster sum of the squared distances. It randomly selects k centroids, which are the representative points of each cluster, and then assigns each data point to the nearest centroid. The centroids are updated by taking the average of the data points assigned to them, the process is repeated until reaching convergence or a maximum number of iterations [26].

For this case we will use a Rust crate called *linfa_clustering::KMeans* through which the most complex operations of the algorithm are performed, such as modeling and prediction. Other standard libraries will also be used to help perform data manipulation and operations [27].

For the generation of load within the information analysis, we will provide the algorithm with several versions through the manipulation of the variable that controls the number of random points generated. This data is initially conceived to be one thousand, to become ten thousand, hundred thousand, and finally one million. This will generate different performance information when running different versions of the same algorithm with more workload.

3.1.2. Linear regression

Linear regression is a statistical approach that estimates the relationship between a scalar dependent variable and one or more independent variables. This helps to analyze how changes in the independent variable affect the dependent variable and thus make predictions based on the information.

Linear regression has several practical uses, oriented to two categories:

- Error reduction in prediction or forecasting can be used to fit predictive models to observe a data set.
- Explain variation in the dependent variable that can be attributed to variation in the independent variables. It quantifies the strength of relationships between variables [28].

For this example, we will use the crate linfa_linear::LinearRegression which will facilitate modeling tasks, such as training and prediction as well as the calculation of the squared error [29]. The linfa-datasets crate will also be used, which provides a collection of datasets ready to be used in tests and examples.

Currently the following dataset is provided:

Name	Description	#Samples #Features #Targets	Targets
diabetes	The diabetes dataset gives samples of human biological measures, such as BMI, age, blood measures, and tries to predict the progression of diabetes.	441, 10, 1	Regression

Table 6: Linfa datasets Rust [30].

3.1.3. Decision Trees

A decision tree is a supervised learning technique that performs classification and regression by constructing a hierarchical structure of decisions and outcomes. It has a hierarchical tree structure consisting of a root node (no incoming branches), branches, internal or decision nodes (with children), leaf or terminal nodes (no children). The leaf nodes represent the possible outputs in the dataset [31].

This is the most complex of the examples, as a non-integrated data source will be used. The main crate that we will use is *linfa_trees::DecisionTree*, which allows us to perform the training and prediction task on the model associated with the information. Then we also have other libraries that allow the manipulation of the CSV file, such as*: csv::ReaderBuilder*, or *csv::Reader*.

The data source is a CSV containing information about The Behavioral Risk Factor Surveillance System (BRFSS) is the nation's leading health-related telephone survey system that collects statewide data on US residents. Concerning their health-related risk behaviors, chronic health conditions, and utilization of preventive services. It is decided to set the characteristic "*Heart_Disease*" as the target variable. In addition, there are categorical variables that are associated with numerical values for a better manipulation of the information. The values that have been changed and their corresponding mapping are detailed in the appendix (see Appendix B). The file contains 308,858 records.

3.1.4. Metrics and criteria

For standalone implementations, the following metrics and criteria will be considered:

- Execution time, valued at seconds.
- CPU usage during execution, in % usage, in the case of which it applies.
- Memory usage during execution, virtual memory snapshots, in the cases to which it applies.

For server implementations, in this case on Docker Desktop, the following will be evaluated:

- Image creation time, in seconds.
- Image size, in MB.
- Container execution time, in seconds.

4. Implementation

This section will indicate the tools that have been used both for the implementation of the algorithm code, as well as for the capture of the information required for the performance analysis of the executions of each one of them.

For these tests, a computer with OS Microsoft Windows 10 Enterprise 22H2 v. 64 bits, with 16 GB of RAM, AMD Ryzen 7 3750H 2.3 GHz x64 based processor.

The code of the algorithms has been added to the appendix (See Appendix C - D- E).

4.1. Programming language.

The most appropriate programming language considered for this work has been Rust v. 1.71.1 (eb26296b5 2023-08-03). A multi-paradigm programming language allows you to choose the best approach for your problem domain. Its primary features are focused on performance, reliability, and productivity, but it really is not something that falls within the scope of this work. The main reason it was chosen for this work is its toolchain and integration with Wasm, through which the ".rs" code can be directly compiled into ".wasm".

Wasm-pack is the one-stop shop for building, testing, and publishing Rust-generated WebAssembly [23].

Rust has integrated libraries that will facilitate the implementation of the algorithms that are to be evaluated in this work.

4.2. IDE and Code Editor

Visual Studio Code is the IDE chosen for this work, due to its open-source condition and the availability of the necessary extensions that integrate it with Rust (rust-analyzer), and that allow debugging tasks for its validation and correction in a way faster and more efficient [25].

4.3. Other tools

For server implementations, Docker Desktop v.4.18.0 has been used, which is a tool that allows you to build and share containerized applications. It has a graphical user interface that makes it easy to manage containers and images [34].

Recently, Docker added integration with Wasm workloads, a feature that at the time of writing this work is still in Beta starting from version 4.15.0 [35]. The runtimes that you can use in this environment are the following:

- io.containerd.slight.v1
- io.containerd.spin.v1
- io.containerd.wasmedge.v1
- io.containerd.wasmtime.v1

All these runtimes, including WasmEdge, use the runwasi library. Runwasi is a project whose objective is to make it easier to run Wasm modules as containers with containerd, a platform that manages containers. This one is written in Rust and was donated to the CNCF. Runwasi is still under development, but it has enough features for testing.

Containerd provides the ability to use OCI-compatible artifacts and containerd shims. This shim retrieves the Wasm module from the OCI artifact and executes it using the Wasm runtime.

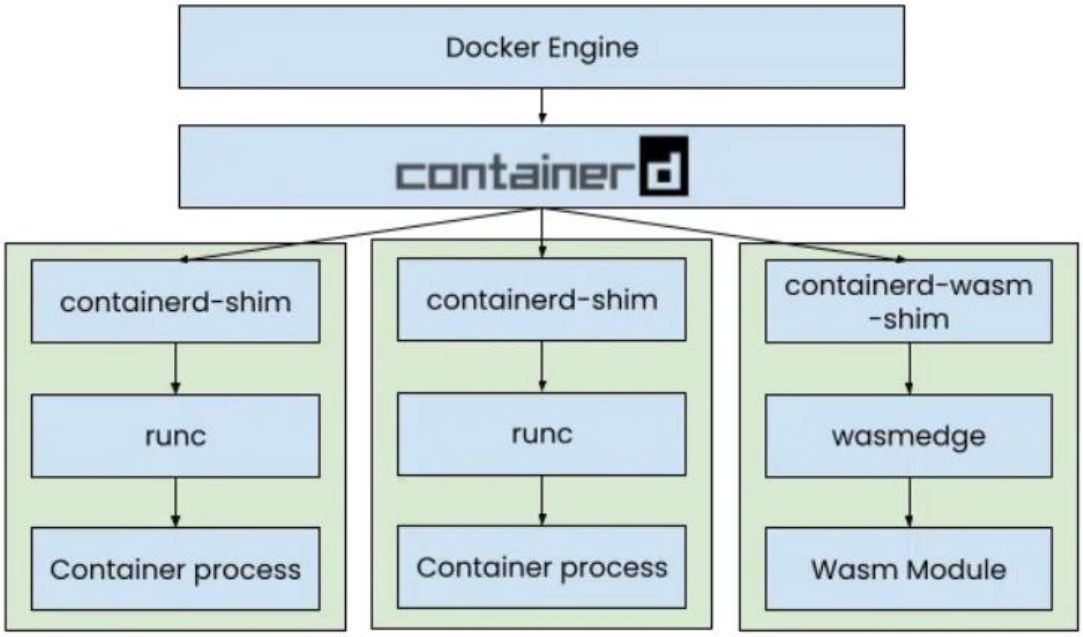

Figure 3: Docker Wasm architecture [44].

For the capture and analysis of performance information, we have used Windows Performance Recorder (WPR v.10.0.2262.1) and Windows Performance Analyzer (WPA v.11.4.53.1473). Which are performance monitoring tools, which are included in the Windows Assessment and Deployment Kit (Windows ADK). WPR allows recording of system events using Event Tracing for Windows (ETW), while WPA allows analysis of recorded events using graphs, tables, and other features [36] [37].

For this work, we have used three features, which group the metrics to be analyzed: time, CPU usage and virtual memory usage. Below is a more detailed explanation of each of them:

- Processes (Lifetime by process). This feature allows us to observe the duration of the processes captured by the WPR as well as their start and end times. It also allows you to see the command line arguments.

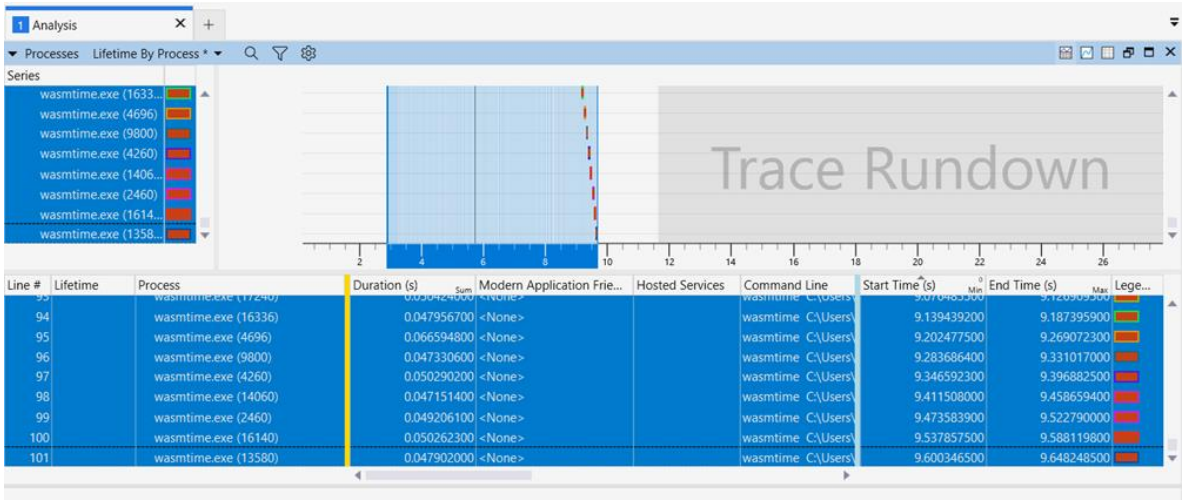

Figure 4: WPA Processes sample view.

- CPU usage (Attributed – Utilization by process). This feature allows you to analyze the CPU usage of each process, thread and stack recorded by the WPR. The "% CPU Usage" column shows the percentage of CPU time that was consumed by the stack relative to the total CPU time of all processors.

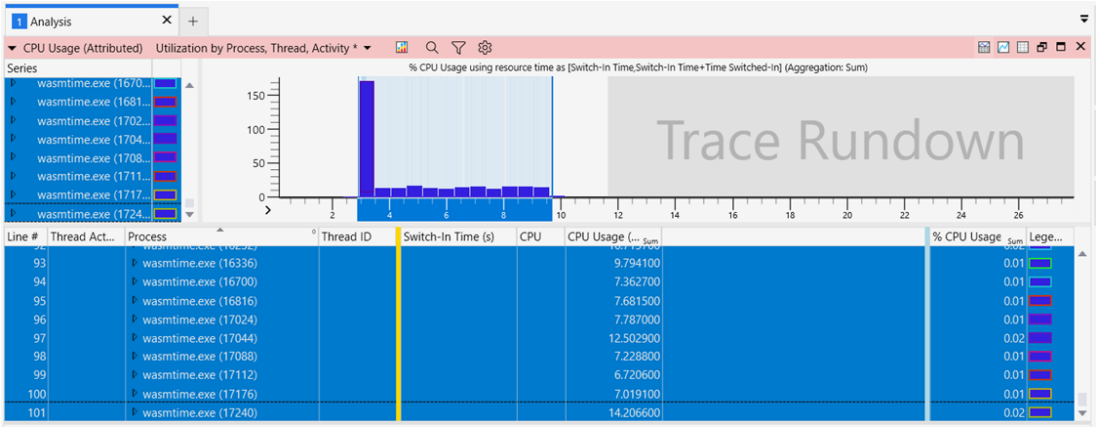

Figure 5: WPA CPU Usage sample view.

- VirtualAlloc Commit LifeTimes (Outstanding commit by process). This feature shows the lifetime and size of the memory allocations that were made by the VirtualAlloc function in the application. The column "Size (MB)" displays the size of the memory region in MB. A positive value indicates an allocation, and a negative one a deallocation.

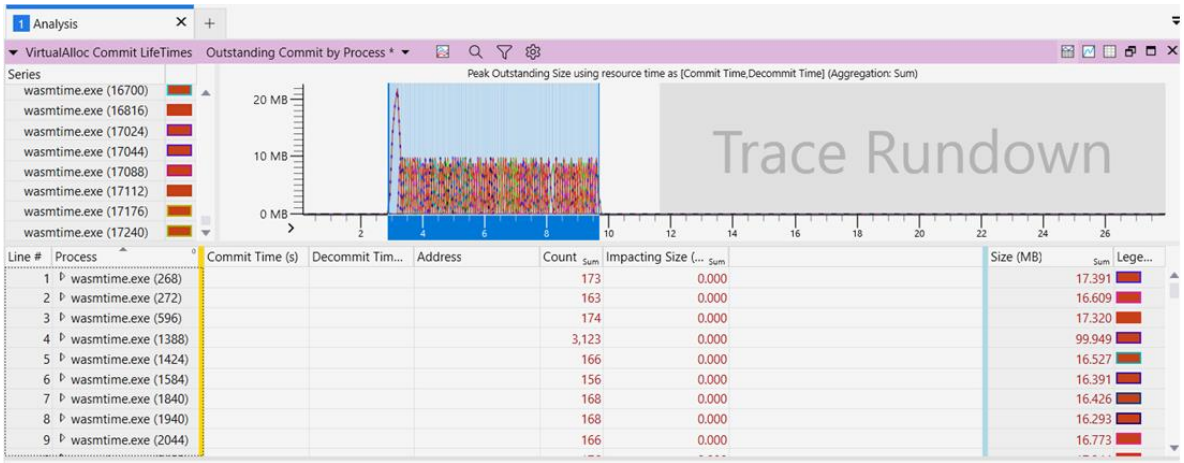

Figure 6: WPA VirtualAlloc Commit LifeTimes sample view.

5. Analysis

In this section the results obtained during the execution of the different proposed examples will be analyzed.

- The algorithms are computational tasks that do not require any system interaction or host interface. Therefore, it can run on any Wasm runtime that supports the basic Wasm specification, regardless of whether it supports WASI or not.
- They are likely to benefit from compilation rather than interpretation, as it involves arithmetic operations and memory access that can be optimized by compilers. Therefore, runtimes that use AOT or JIT compilers are expected to perform better than runtimes that use interpreters.
- The algorithms may also benefit from vectorization or parallelization, as it involves matrix operations and data processing that can be accelerated by SIMD instructions or multithreading. Therefore, runtimes that support these features are expected to perform better than runtimes that do not support them.

5.1. K-Means

To begin with, the analysis of the execution of the k-means algorithm is carried out, which, as indicated above, is conducted in 4 versions, “1,000”, “10,000”, “100,000” and “1,000,000” observations to have a better perspective of the performance of the different runtimes for each case. In short, the goal of the code is to generate and cluster synthetic data using the k-means clustering algorithm.

Runtime	1,000	10,000	100,000	1,000,000
WasmEdge	0.0587	0.0933	1.4181	9.4543
Wasmtime	0.0543	0.1567	1.2799	10.2881
Wasmer	0.2240	0.3532	1.6326	12.6438
Wasm3	0.2239	1.7493	18.4450	147.7515
WAVM	5.2734	5.4081	6.1254	13.5126
GraalWasm	0.3900	1.1995	6.5069	16.8267
Spin	0.0359	0.1791	1.3795	11.3631

Table 7: K-mean average processing time (s).

Runtime	1,000	10,000	100,000	1,000,000
WasmEdge	0.0518	0.0391	0.0518	0.1207
Wasmtime	0.0247	0.0046	0.0008	0.0000
Wasmer	0.5697	0.2406	0.2107	0.1396
Wasm3	0.0852	0.1173	0.1217	1.1200

WAVM	0.1210	0.1217	0.1221	0.1236
GraalWasm	0.3304	0.3506	0.3102	0.2097
Spin	0.0304	0.0102	0.0000	0.0000

Table 8: K-mean CPU usage percentage (%).

Runtime	1,000	10,000	100,000	1,000,000
WasmEdge	14.0289	14.0986	35.5341	231.3116
Wasmtime	17.1734	18.2982	38.7121	233.4973
Wasmer	59.4422	60.3085	80.4097	222.0128
Wasm3	10.0717	26.6434	78.7303	1,057.3722
WAVM	232.4163	232.9880	254.5793	450.3450
GraalWasm	135.9072	335.8457	461.3891	1,540.0616
Spin	0.6218	0.6195	0.6228	0.6295

Table 9: K-mean virtual memory size (MB).

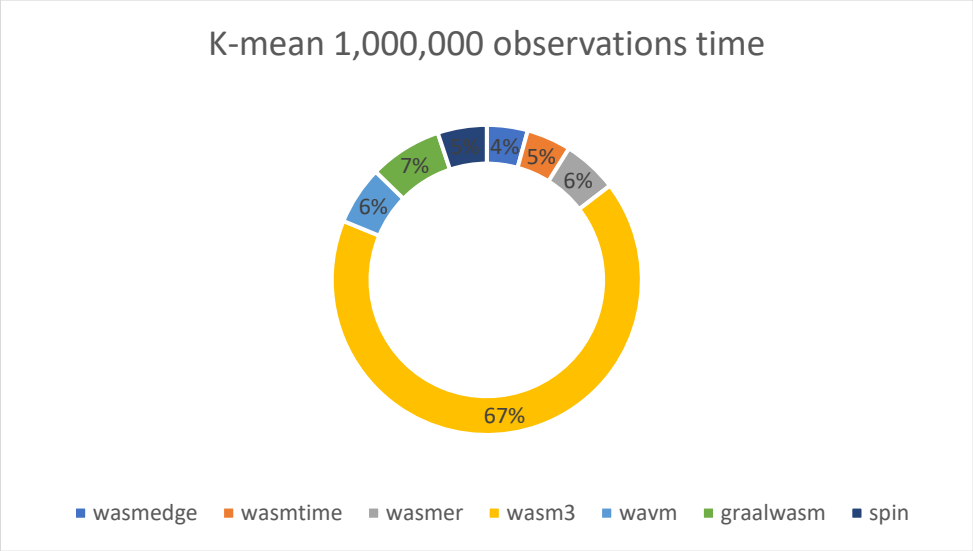

Figure 7: K-mean one million observations time.

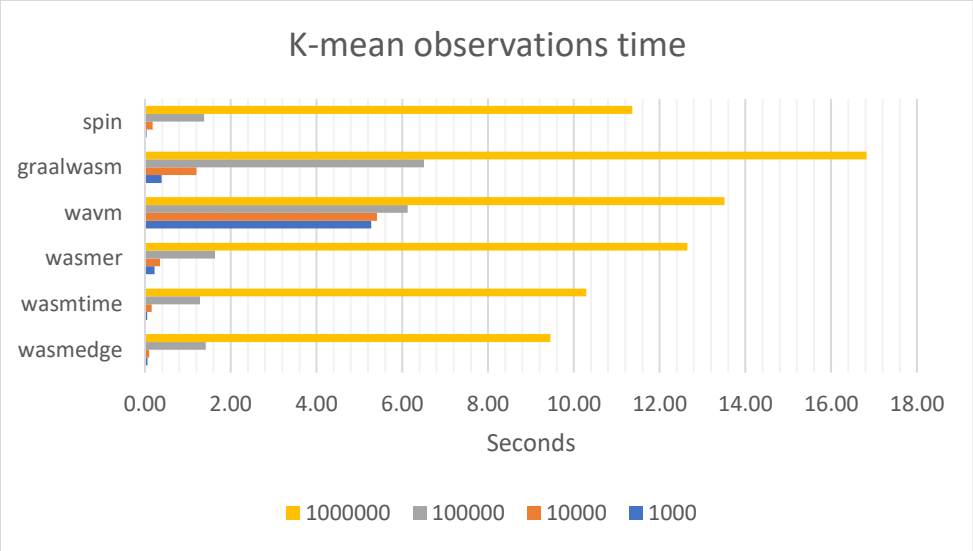

Figure 8: K-mean observations time.

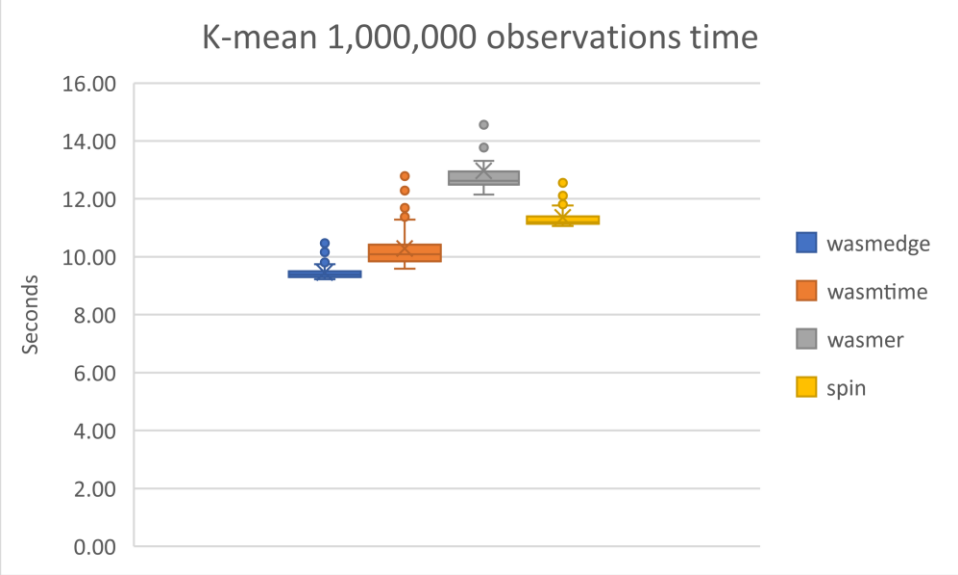

Figure 9: K-mean one million observations time.

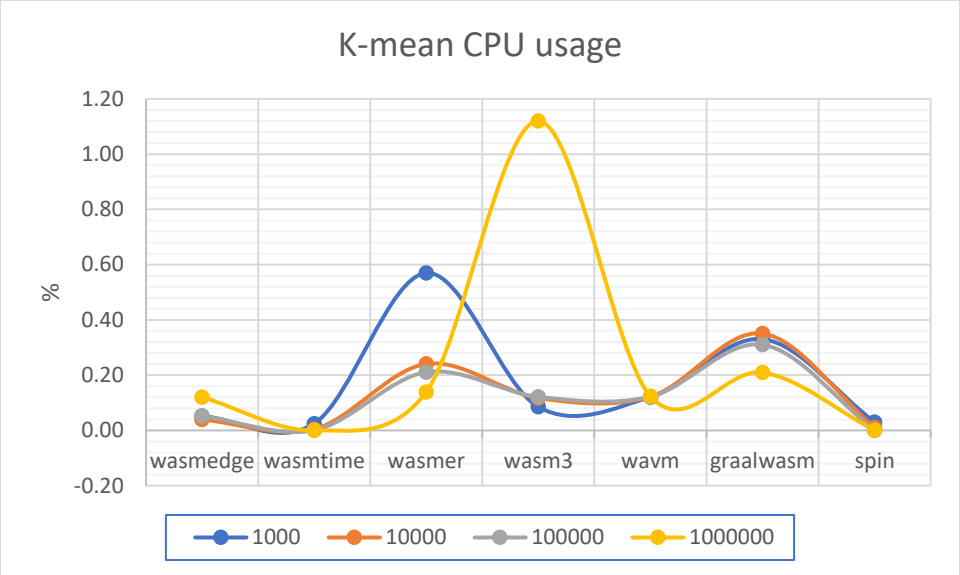

Figure 10: K-mean CPU usage.

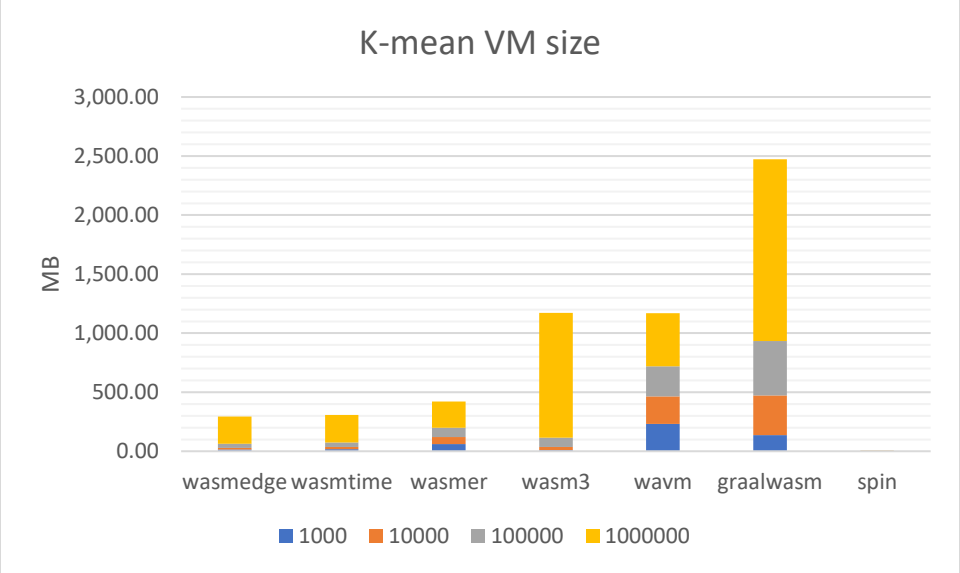

Figure 11: K-mean VM size.

For each version of the algorithm, one hundred samples have been carried out, except for the version of one million observations, for which, as it was too extensive, it was decided to only conduct ten samples, which seemed sufficient for our purposes.

From the data obtained, we can deduce the following points:

- The processing times increase as the number of observations increases, which is expected since more data means more computation.

- The Wasm runtimes have different processing times for the same number of observations, which indicates that they have different efficiency and scalability.
- The fastest one for all numbers of observations is spin, which has the lowest processing times across the board. Even though Spin presents the best results, its limitations and trade-offs should be considered, such as compatibility, functionality, security, and portability. For example, Spin may not run the algorithms correctly if it uses exceptions, and it may not be compatible with other platforms or applications that require WASI or other host interfaces. WasmEdge and Wasmtime are more general-purpose and compliant runtimes, but they may have higher memory consumption or startup time than Spin.
- The slowest Wasm runtime for all numbers of observations is wasm3, which has the highest processing times by far. It is also the most variable in terms of performance since it has a significant difference between its processing times for different numbers of observations.
- The most consistent Wasm runtime in terms of performance is WAVM, which has similar processing times for different numbers of observations.
- For the scope of processor use, in general the times are ridiculously small, as well as their variations, to be considered relevant, except for some cases such as wasm3, which already presented large processing times in the section former.
- As far as memory is concerned, the trend continues, with WasmEdge and Wasmtime presenting the best performance, closely followed by Wasmer, with slight difference. In this case, the last position has changed owners, being GraalWasm the one that consumes the most resources.
- We must consider that Spin, as it is a web framework, there is an additional service that is constantly running to receive requests. The data taken in the examples belongs to the requests made by the browser.

5.2. Linear regression

The second example is the linear regression algorithm. Its objective is to train and evaluate a linear regression model on a rust-integrated data source, known as diabetes. As happened with the previous example, one hundred samples have been made for each Wasm runtime, thus obtaining more relevant values.

Runtime	S
WasmEdge	0.0282
Wasmtime	0.0349
Wasmer	0.2193
Wasm3	0.0579
WAVM	4.6458
GraalWasm	0.3192
Spin	0.0202

Table 10: Linear regression average processing time (s).

Runtime	%
WasmEdge	0.0456
Wasmtime	0.0108
Wasmer	0.4917
Wasm3	0.0615
WAVM	0.1125
GraalWasm	0.2384
Spin	0.0226

Table 11: Linear regression CPU usage percentage (%).

Runtime	MB
WasmEdge	14.7817
Wasmtime	16.5325
Wasmer	61.6638
Wasm3	7.9272
WAVM	76.1280
GraalWasm	191.3354
Spin	0.6491

Table 12: Linear regression virtual memory size (MB).

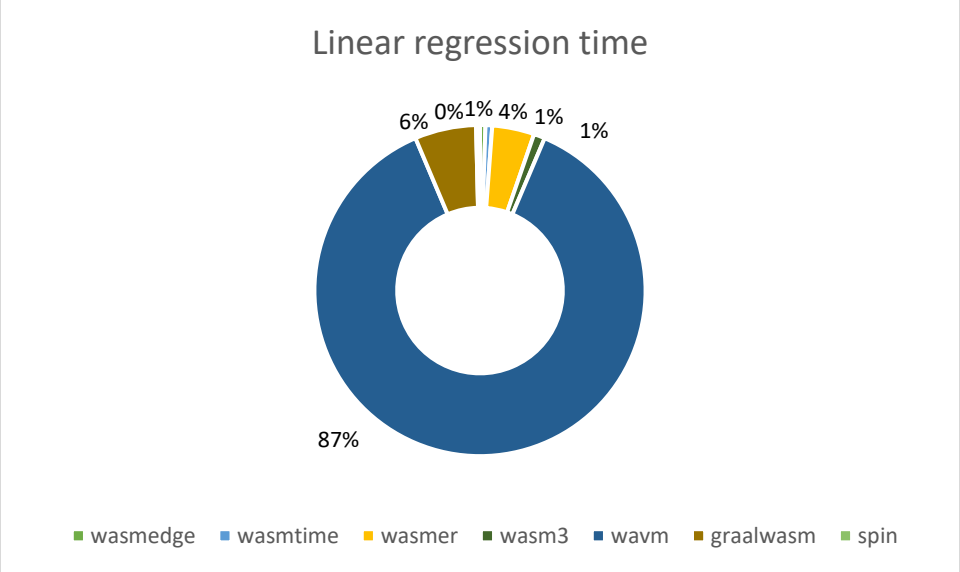

Figure 12: Linear regression time.

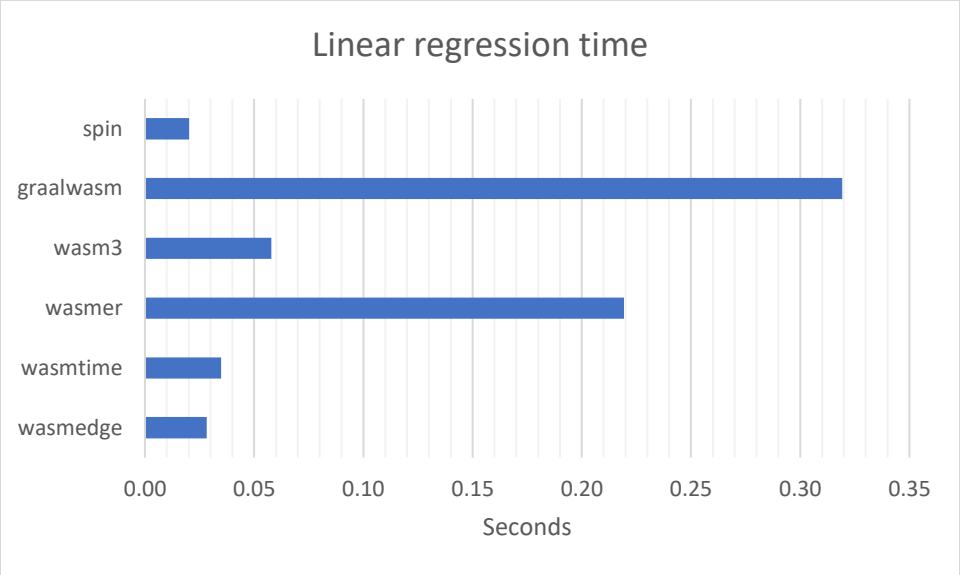

Figure 13: Linear regression time.

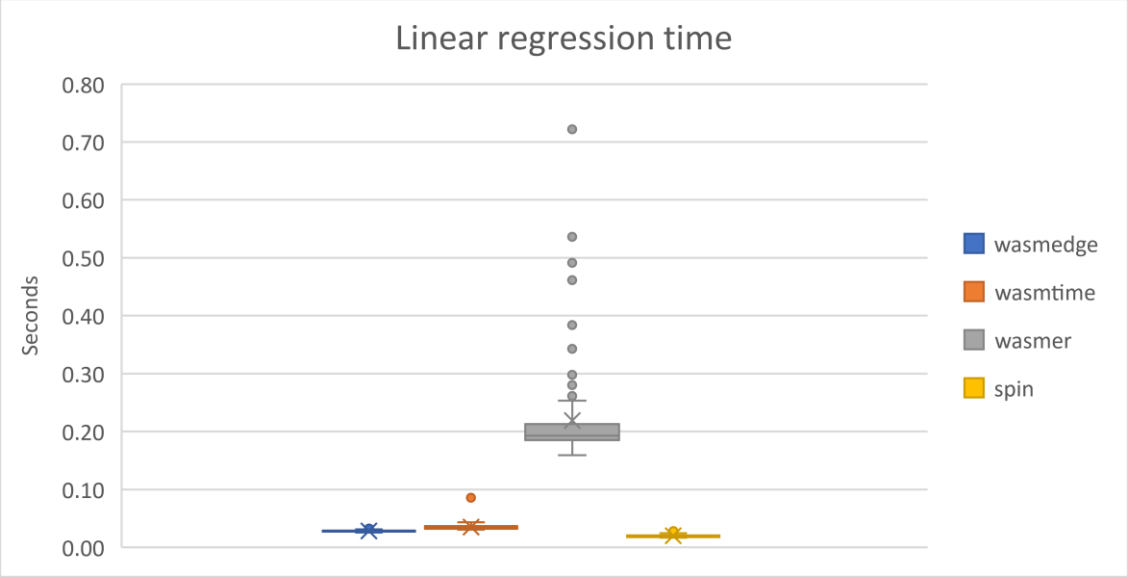

Figure 14: Linear regression time.

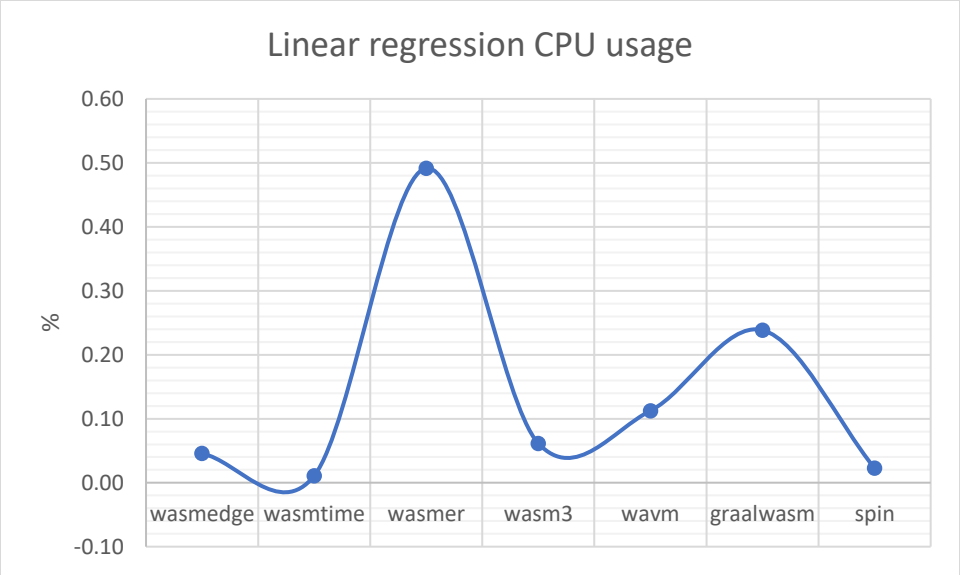

Figure 15: Linear regression CPU usage.

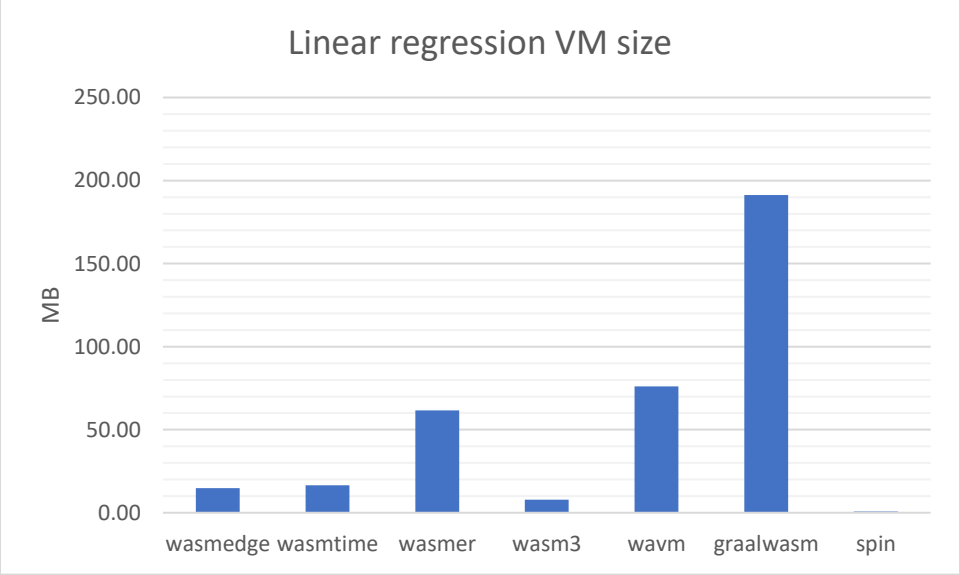

Figure 16: Linear regression VM size.

The results show that Spin is the fastest framework in terms of time, followed by the WasmEdge and Wasmtime runtimes. It is consistent with the results obtained in the previous k-mean example, as well as its goals and features since they are optimized for speed and efficiency.

The results also show that Wasmer, GraalWasm and WAVM are the slowest runtimes for this algorithm. This may be due to the complex mathematical operations that are required during the process. WAVM

might perform better with a larger workload that can take advantage of its multithreading and SIMD capabilities.

In terms of percentage of processor use, the one that gives the worst results is Wasmer, however, the tendency to be values and differences that are too low, which are quite insignificant.

Regarding memory, the results are consistent with the time and according to the characteristics of each runtime mentioned above.

5.3. Decision tree

Decision trees is the last algorithm on which performance tests will be performed for the different Wasm runtimes, for this example a csv file is included as a large data source. It was decided to include it through a macro instead of using WASI to access the file system, because it will be the same example that will be used for tests with Docker. The purpose of this algorithm is to train and evaluate a decision tree classifier on a cardiovascular disease risk data source.

Due to the complexity of the algorithm and the consumption of resources necessary for its execution, in this case one hundred samples have not been repeated for each Wasm runtime, but only ten.

Runtime	S
WasmEdge	33.3281
Wasmtime	73.8908
Wasmer	110.2868
Wasm3	609.4514
WAVM	38.5740
GraalWasm	243.7430
Spin	141.6622

Table 13: Decision tree average processing time (s).

Runtime	%
WasmEdge	1.2200
Wasmtime	0.0010
Wasmer	1.2620
Wasm3	1.2130
WAVM	1.2360
GraalWasm	2.4720
Spin	0.0000

Table 14: Decision tree CPU usage percentage (%).

Runtime	MB
WasmEdge	160.0209
Wasmtime	204.6116
Wasmer	241.6327
wasm3	906.4774
WAVM	355.1692
GraalWasm	8,939.3180
Spin	0.6285

Table 15: Decision tree virtual memory size (MB).

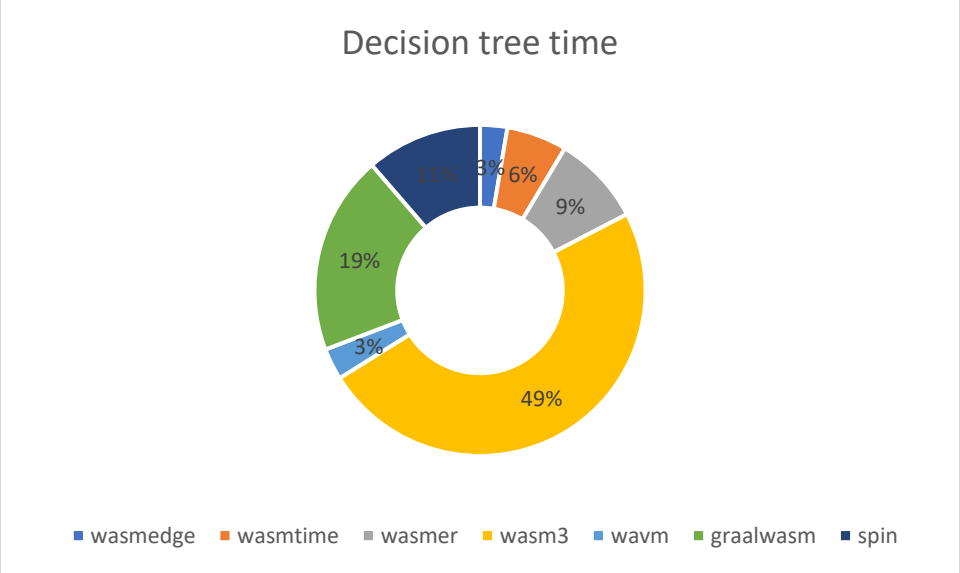

Figure 17: Decision tree time.

Figure 18: Decision tree time.

Figure 19: Decision tree time.

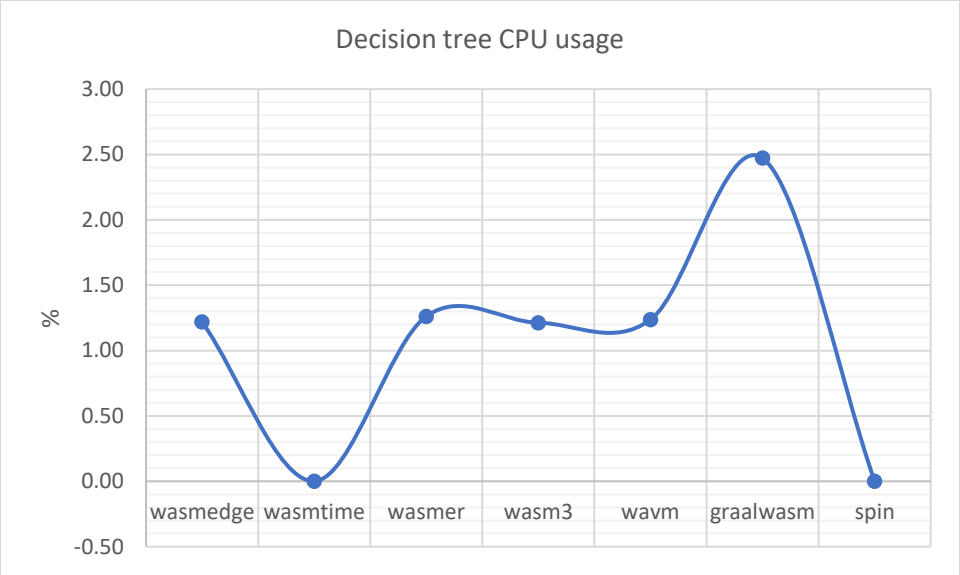

Figure 20: Decision tree CPU usage.

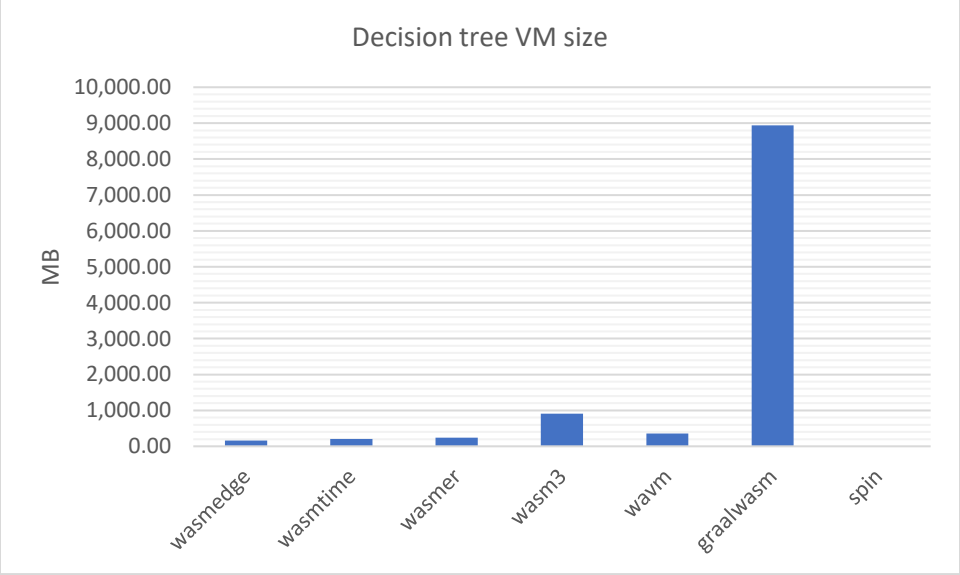

Figure 21: Decision tree VM size.

Based on the results, WasmEdge has the shortest execution time, followed by WAVM, highlighting the consistency of Wasmtime, taking account of the results of the previous examples. On the other hand, wasm3 is the one that has taken the longest with a substantial difference compared to the others.

WasmEdge is a compiler-based runtime that supports various extensions and host APIs, such as TensorFlow, ONNX, or MQTT. This can also run Wasm modules on edge and cloud devices. These features can give you an advantage over other runtimes in terms of speed and efficiency.

In the case of WAVM, this is a compiler-based runtime that also uses LLVM to optimize and execute the Wasm modules. It supports SIMD instructions, which could explain why it has a low execution time compared to the others.

Wasm3 is an interpreter-based runtime and does not support WASI or SIMD instructions which limits its functionality and performance and explains the reason for its high execution time.

In the processor usage percentage area, most have low CPU utilization, ranging from 0% to 2%. This means that they are efficient and lightweight in terms of resource consumption (processor). This is one of the main benefits of Wasm.

In terms of memory, Spin is seen to have the lowest consumption, followed by WasmEdge. These two runtimes are the most memory efficient among those that have been assessed. At the other extreme, GraalWasm has the highest memory consumption, by far.

The other runtimes without exceeding 1000 MB, are of moderate consumption compared to Spin and GraalWasm.

GraalWasm is a virtual machine-based runtime that uses GraalVM to run Wasm modules. It supports multiple languages and various platforms. However, the Wasm add-in is in a state of development, so its results may not be definitive.

5.4. Docker

For this exercise, the three previous algorithms have been executed, that is, k-mean (one million observations), linear regression and decision tree, in two ways, traditional docker and Wasm docker. For said tests, twenty-five samples have been made for algorithm. For the decision tree algorithm, two types of tests have been carried out, one with the original dataset and another with the dataset reduced to 100,000 records, to verify the scalability of the technologies. The results obtained are shown:

Runtime	K-mean	Regression	Tree1	Tree2
Docker	3.7903	5.7757	3.0049	4.9623
Wasm	0.9997	1.0478	1.0120	1.0573

Table 16: Docker build time (s).

Runtime	K-mean	Regression	Tree1	Tree2
Docker	4.0600	4.0700	20.8200	33.7700
Wasm	2.9700	3.0400	9.0500	22.0000

Table 17: Docker image size (MB).

Runtime	K-mean	Regression	Tree1	Tree2
Docker	5.4586	1.2203	16.3730	67.0527
Wasm	11.7617	0.8447	21.9301	87.2101

Table 18: Docker container execution time (s).

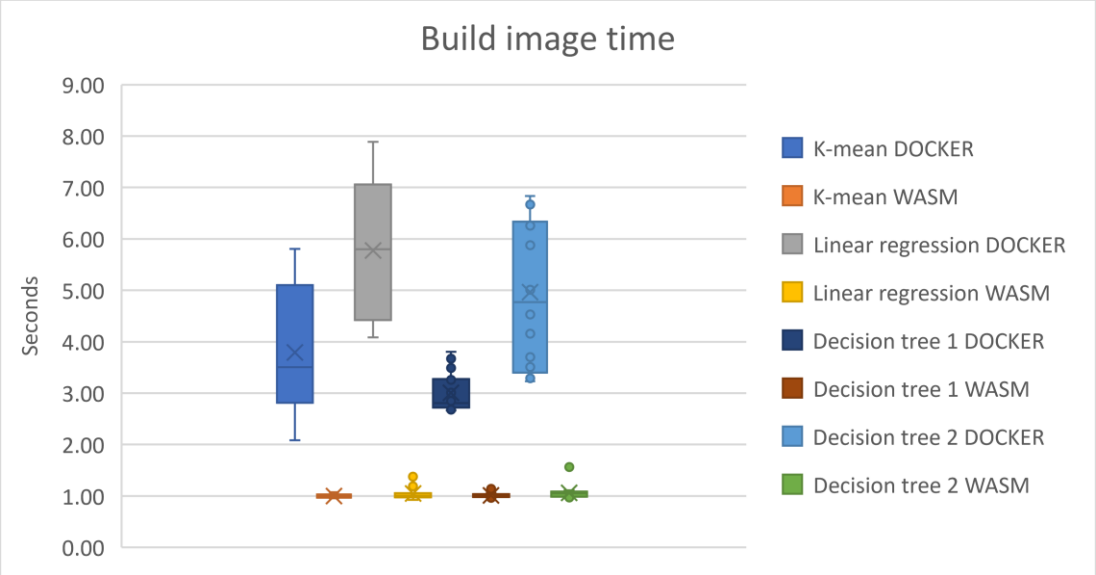

Figure 22: Docker build image time.

Figure 23: Docker K-mean execution time.

Figure 24: Docker linear Regression execution time.

Figure 25: Docker decision tree one execution time.

Figure 26: Docker decision tree two execution time.

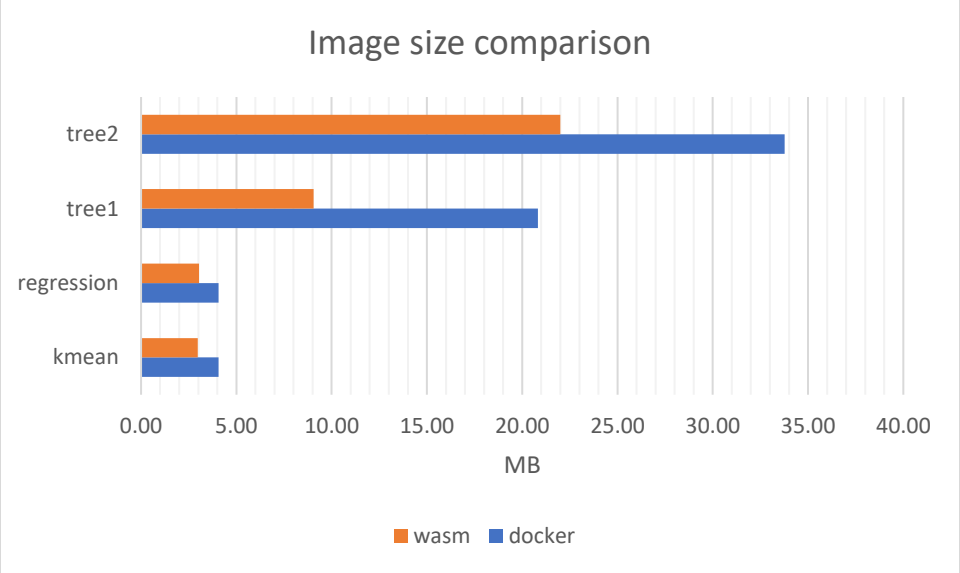

Figure 27: Docker image size comparison.

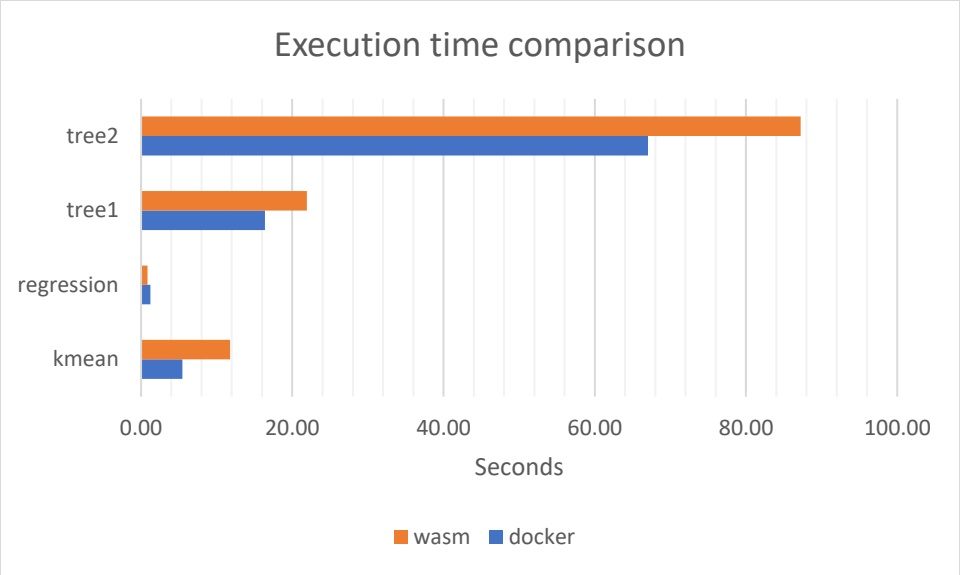

Figure 28: Docker execution time comparison.

Next, the results obtained are analyzed in more detail, according to the three metrics mentioned above, which will be: compilation time, image size and execution time.

Compilation time. The results for this area can be summarized in that the time to build the image for Docker Wasm is approximately one second, regardless of the algorithm, while for traditional Docker the time varies between 3 and 5 seconds depending on the algorithm, with the results being favorable to Wasm. It must be considered that the cold times have been ignored, which is the first compilation, for Docker, so as not to deviate the values of the samples, since they are higher. However, Wasm still takes first place. This is because Docker Wasm uses a lightweight and portable format, while traditional docker uses Linux-specific features that require a virtual machine or compatible layer to run on top of other operating systems. Therefore, building a Docker Wasm image is simpler and faster than building a Linux container image.

Image size. It is evidenced, according to all the observations, that the images produced by Docker Wasm are smaller. The images produced by Docker have varied sizes determined by the base image that is used to build it, for example, the image that has been used to do these tests, alpine, has a size of approximately 3MB. There are different base images with unusual sizes, for example: Debian has 125 MB, and Ubuntu has 188 MB. If a smaller base image is used, the size of the final image will be smaller as well.

Another aspect to consider is the number of layers, which causes some overhead to the size of the image. The more layers you have, the larger the final image will be. Each layer contains the files and directories that are added or modified by the instructions specified in the Dockerfile (View Appendix F).

Taking these factors into account, the results make sense, since Docker Wasm uses a smaller base image (.wasm file), has fewer layers (one), and less content in each layer (the essential files to run Wasm). While

Docker uses a larger base image, it has more layers, and its content is larger, taking up an entire operating system and its libraries.

Run time. This could be the most disparate and controversial section that has been found within the tests that have been carried out. Since according to the theory, Docker Wasm should be more agile in execution than traditional Docker, however it should be noted that Docker Wasm is a recent technology that is still in the development phase, and this could vary in subsequent versions.

K-mean is a clustering algorithm that requires a lot of iterations and computations to find the optimal centroids for the data points. Running it on Docker Wasm could be slower because being a modern technology it is not fully optimized for complex numerical operations.

Linear regression is a simple algorithm that finds the best-fit line for a set of points. Docker Wasm could be faster because Wasm has a smaller memory footprint and faster startup time than Linux containers. This means that Wasm can load and run code more efficiently, as opposed to Linux containers, which must boot an entire operating system and load additional binary libraries.

For decision tree algorithms, they are computationally intensive and may require more memory and CPU resources than other algorithms. Docker Wasm may have some limitations or overheads in accessing these resources compared to traditional Docker. The data size, quality, and complexity may also affect the execution time of decision tree algorithms. Docker Wasm may have some challenges or inefficiencies in handling large or complex data sets compared to traditional. Due to the increase calculated for the two examples conducted (tree1 = 100,000 records / tree2 = 300,000 records), which is approximately 30% for both cases, workload can be omitted as one of the causes of the low efficiency in runtime for Wasm.

5.4.1. Docker/Wasm runtimes

Finally, it was decided to make a final comparison, crossing the execution times of the Wasm runtimes and Docker, in this way to be able to visualize the performance of both technologies.

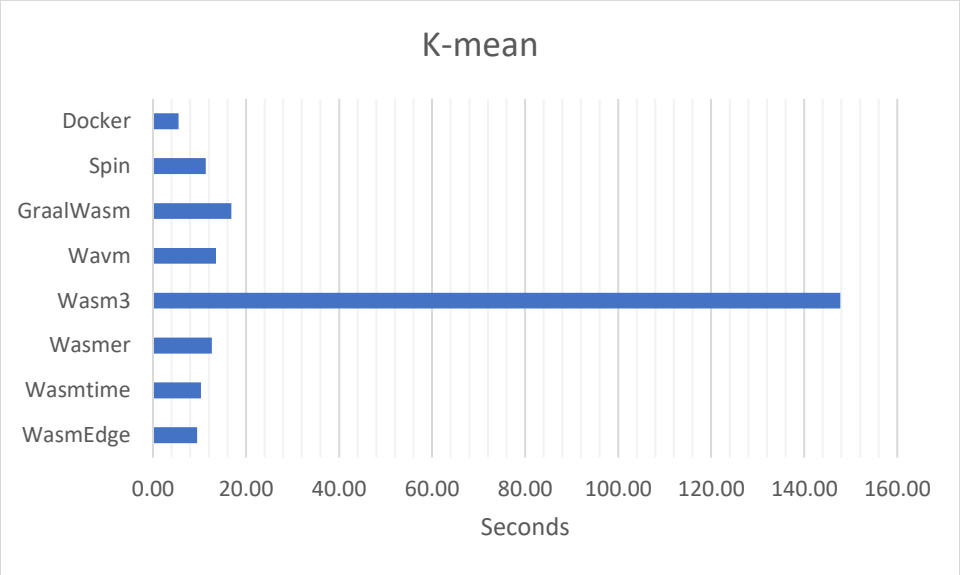

Figure 29: K-mean Docker/Wasm runtimes execution time comparison.

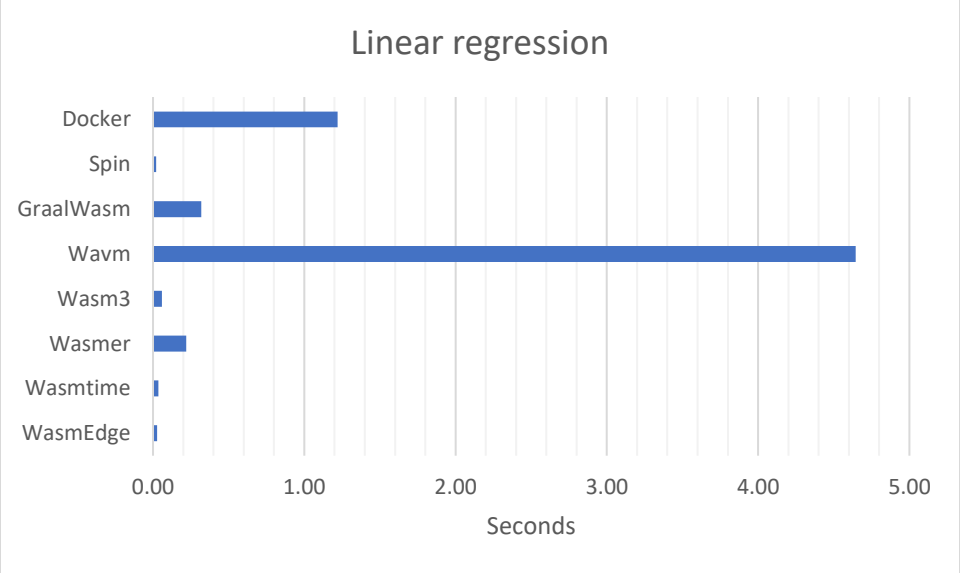

Figure 30: Linear regression Docker/Wasm runtimes execution time comparison.

Figure 31: Decision tree Docker/Wasm runtimes execution time comparison.

In general, the Docker times, with respect to the Wasm runtimes, are quite acceptable, never exceeding the worst time of a Wasm runtime, even obtaining the best time in the case of the K-mean algorithm.

For the particular case of the linear regression algorithm, it can be seen how it presents a greater difference with the best Wasm runtime times, this can be explained because Docker images are still operating systems and being the example with the least cost in execution times, the overload generated by the fact of having to launch the OS as well as the dependent libraries can be evidenced at that time.

In the case of K-mean algorithm, docker has the better time, no with more difference with the Wasm runtimes, but it may be since the algorithm is more computationally intensive and requires more memory and CPU resources than the other algorithms, and Docker may have better optimization and allocation of these resources.

6. Summary of main insights

The main insights ensuing from this work are the following ones:

- This study has been conducted as a basis for the analysis of an AI-enabled cloud edge framework. If we start from this premise, the runtime that best meets the required characteristics would be WasmEdge, but considering that the integration should allow various possibilities, these should include Wasmtime and Spin, since it is considered that their results in the tests have been quite acceptable. All of them are important projects and they are receiving considerable maintenance and support from the community.
- WasmEdge is a Wasm runtime that is designed for edge, cloud, and decentralized applications. It is the fastest and most consistent in resource consumption, according to the tests conducted within our research work. It supports several features and extensions, such as: WASI, Ewasm, TensorFlow, database connectors, and JavaScript APIs. This is an official sandbox project hosted by the CNCF. It also has support for integration with Docker.
- For Docker Wasm tests, it is possible to demonstrate the improvement of image construction times, as well as the size of the image at the end of its creation, however, for the proposed algorithms, there is no evidence of improvement of times during the container execution. For which, several aspects must be considered, among which we have that to be a technology in full development, it could have limitations when accessing system resources.

7. Conclusions.

Most browsers support Wasm as well as other platforms and runtimes. It currently has a growing ecosystem of tools, frameworks, and libraries that allow developers to create applications with high performance, security, and interoperability.

In this work, experimental tests have been conducted to demonstrate the performance of certain algorithms related to machine learning. In summary, the results of our experiments indicate that the different Wasm runtimes have different strengths and weaknesses to execute the different algorithms presented in this work. The choice should be based on needs and preferences, considering not only performance, but also runtime functionality, compatibility, security, and portability.

Another of the tools analyzed in this work is Docker. Docker Desktop currently has a feature under development through which Wasm workloads can be deployed as images to be executed as containers within the platform, with the aim of speeding up their implementation, as well as saving resource consumption.

From a personal perspective, this work has been a demanding task for me, as it reflects my academic growth and progress. It has also enabled me to enhance my technical abilities such as coding, and to apply the knowledge I gained from this master's degree, such as machine learning techniques, which are at the forefront of technology and attract a lot of research funding from major market players.

8. Future work

Once the necessary research has been conducted to find out the current situation of Wasm technology, it is expected that this study will serve as a basis for the analysis and/or development of a container-like execution platform based on Wasm technology in the context of the Horizon Europe CLOUDSKIN research project.

The data collected in this work will serve as a guide for similar exercises carried out in the new framework born because of this research project.

This field is dynamic and ever-changing, so it is advisable to stay updated on the latest advancements of the technologies discussed in this document, as well as the novel innovations that emerge from the research conducted by external parties, such as Docker Wasm or spin.

From this position, we also urge that new experiments be tried focused on specific characteristics of the most important runtimes mentioned in this document, such as access to databases or integrations with innovative technologies, such as TensorFlow within the field of AI applications.

Finally, I hope that this work not only serves as a basis for the context of the research project for which it was created, but also serves as a starting point for any research work that is related to Wasm, the data collected in this document can serve as a guide for a better understanding of the technology and the environment that surrounds it.

References

[1] Liam Randall, “How WebAssembly will transform edge computing”, https://www.infoworld.com/article/3703052/how-webassembly-will-transform-edge-computing.html, July 2023.

[2] WEBASSEMBLY, “WebAssembly”, https://webassembly.org/, August 2023.

[3] Shi Weisong, Pallis George, Xu Zhiwei, “Edge computing [Scanning the Issue]”, IEEE, 2019, Vol.107 (8), p.1474-1481.

[4] Matt Butcher, “WebAssembly and Containers”, “https://www.youtube.com/watch?v=OGcm3rHg630”, June 2022.

[5] Docker, “Wasm workloads (Beta)”, https://docs.docker.com/desktop/wasm/, March 2023.

[6] Jain Shashank, “WebAssembly for cloud”, Standard Apress, 2022.

[7] Fermyon, Wasm language support matrix, https://www.fermyon.com/wasm-languages/webassembly-language-support, August 2023.

[8] Bytecode Alliance, “Bytecode Alliance”, https://bytecodealliance.org/, August 2023.

[9] WebAssembly, “WebAssembly core specification”, https://webassembly.github.io/spec/core/_download/WebAssembly.pdf, August 2023.

[10] WasmEdge, “WasmEdge”, https://wasmedge.org/, August 2023.

[11] WasmEdge github, “WasmEdge”, https://github.com/WasmEdge/WasmEdge, August 2023.

[12] Wasmtime, “Wasmtime”, https://wasmtime.dev/, August 2023.

[13] Wasmtime github, “Wasmtime”, https://github.com/bytecodealliance/wasmtime, August 2023.

[14] Wasmer, “Wasmer”, https://wasmer.io/, August 2023.

[15] Wasmer github, “Wasmer”, https://github.com/wasmerio/wasmer, August 2023.

[16] GraalWasm, “GraalWasm”, https://github.com/oracle/graal/blob/master/wasm/README.md/, August 2023.

[17] GraalVM, “GraalVM for JDK 20 Community 20.0.2”, https://github.com/graalvm/graalvm-ce-builds/releases, August 2023.

[18] GraalVM, “GraalVM for JDK 20”, https://www.graalvm.org/release-notes/JDK_20/, August 2023.

[19] Wasm3, “Wasm3”, https://github.com/wasm3/wasm3, August 2023.

[20] Wavm, “Wavm”, https://github.com/WAVM/WAVM, August 2023.

[21] Fermyon, “Fermyon”, https://www.fermyon.com/spin, August 2023.

[22] Fermyon, “Fermyon github”, https://github.com/fermyon/spin, August 2023.

[23] Rust, “Rust Web-Assembly”, https://www.rust-lang.org/es/what/wasm, August 2023.

[24] Visual Studio Code, “Visual Studio Code”, https://code.visualstudio.com/, August 2023.

[25] Visual Studio Code, “Visual Studio Code - Rust”, https://code.visualstudio.com/docs/languages/rust, August 2023.

[26] Wikipedia, “k-means clustering”, https://en.wikipedia.org/wiki/K-means_clustering, August 2023.

[27] Rust Docs, “K-means”, https://docs.rs/linfa-clustering/latest/linfa_clustering/struct.KMeans.html, August 2023.

[28] Wikipedia, “Linear regression”, https://en.wikipedia.org/wiki/Linear_regression, August 2023.

[29] Rust Docs, “Linear regression”, https://docs.rs/linfa-linear/latest/linfa_linear/struct.LinearRegression.html, August 2023.
[30] Rust Docs, “Linfa datasets”, https://docs.rs/linfa-datasets/latest/linfa_datasets/, August 2023.
[31] IBM, “What is a Decision Tress”, https://www.ibm.com/topics/decision-trees, August 2023.
[32] Rust Docs, “Linfa decision tree”, https://docs.rs/linfa-trees/latest/linfa_trees/struct.DecisionTree.html, August 2023.
[33] Kaggle, “Cardiovascular Diseases Risk Prediction Dataset”, https://www.kaggle.com/datasets/alphiree/cardiovascular-diseases-risk-prediction-dataset, July 2023.
[34] Docker, “Docker Desktop”, https://www.docker.com/products/docker-desktop/, August 2023.
[35] Docker, “Wasm workloads”, https://docs.docker.com/desktop/wasm/, August 2023.
[36] Learn Microsoft, “Windows Performance Recorder”, https://learn.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-recorder, August 2023.
[37] Learn Microsoft, “Windows Performance Analyzer”, https://learn.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-analyzer, August 2023.
[38] Djordje Lukic, “Announcing Docker + Wasm Technical Preview 2”, https://www.docker.com/blog/announcing-dockerwasm-technical-preview-2/, August 2023.
[39] Nigel Poulton, “Getting started with Docker + Wasm”, https://nigelpoulton.com/getting-started-with-docker-and-wasm/, November 2022.
[40] developerro, “Dockerfile for Rust”, https://www.developerro.com/2022/05/11/dockerfile-rust/, May 2022.
[41] Nimesha Jinarajadasa, “WebAssembly vs Docker: Exploring their Connection and Potential”, https://kodekloud.com/blog/webassembly-vs-docker/, March 2023.
[42] Frank Denis, “Benchmark of WebAssembly”, https://00f.net/2021/02/22/webassembly-runtimes-benchmarks/, August 2023.
[43] Ashish Singh, “Comparing WebAssembly Runtimes”, https://medium.com/@siashish/comparing-webassembly-runtimes-wasmer-vs-wasmtime-vs-wasmedge-unveiling-the-power-of-wasm-ff1ecf2e64cd, May 23.
[44] Michael Irwin, “Introducing the Docker + Wasm Technical Preview”, https://www.docker.com/blog/docker-wasm-technical-preview/, November 2022.
[45] Andreas Rossberg, “WebAssembly Specification”, https://webassembly.github.io/spec/core/index.html, July 2023.
[46] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha, “Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code”, https://www.usenix.org/conference/atc19/presentation/jangda, July 2019.
[47] Jonah Napieralla, “Considering WebAssembly Containers for Edge Computing on Hardware-Constrained IoT Devices”, https://www.diva-portal.org/smash/get/diva2:1451494/FULLTEXT02, June 2020.
[48] Wenwen Wang, “How Far We’ve Come – A characterization Study of Standalone WebAssembly Runtimes”, http://cobweb.cs.uga.edu/~wenwen/papers/iiswc2022.pdf, 2022.
[49] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, Weihang Wang, “Understanding the performance of webassembly applications”, https://dl.acm.org/doi/abs/10.1145/3487552.3487827, November 2021.

[50] Benedikt Spies, Markus Mock, “An Evaluation of Webassembly in Non-Web Environments”, https://ieeexplore.ieee.org/abstract/document/9640153, December 2021.
[51] Vojdan Kjorveziroski, Sonja Filiposka, “WebAssembly Orchestration in the Context of Serverless Computing”, https://link.springer.com/article/10.1007/s10922-023-09753-0, July 2023.

Appendix

Appendix A. Enable Wasm workloads.

Wasm workloads require the containerd image store feature to be enabled. Unless you are already using the containerd image store, you will not be able to access the images and containers that existed before.

1. Open the Docker Desktop Settings.
2. Go to the Features in development tab.
3. Check the following checkboxes:
    - Use containerd for storing and pulling images.
    - To enable Wasm.
4. Select Apply & restart to save the settings.
5. In the confirmation dialog, select Install to install the Wasm runtimes [5].

Appendix B. Mapping categorical values “cardiovascular diseases”

Categorical value	Numeric value
General Health	
Poor	0
Fair	1
Good	2
Very Good	3
Excellent	4
Checkup	
Never	0
Within the past year	1
Within the past 2 years	2
Within the past 5 years	3
5 or more years ago	4
No-Yes	
No	0
Yes	1
Diabetes	
No	0
No, pre-diabetes or borderline diabetes	1
Yes	2
Yes, but female told only during pregnancy	3
Sex	
Male	0
Female	1
Age	
18-24	0
25-29	1
30-34	2
35-39	3
40-44	4
45-49	5
50-54	6
55-59	7
60-64	8
65-69	9
70-74	10
75-79	11
80+	12

Table 19: Mapping categorial values “cardiovascular diseases”.

Appendix C. K-Means rust code.

```rust
use linfa::DatasetBase;
use linfa::traits::{Fit,  Predict};
use linfa_clustering::{ KMeans, generate_blobs};
use ndarray::{Axis, array};
use ndarray_rand::rand::SeedableRng;
use rand_isaac::Isaac64Rng;

fn main() {
    // Random number generator, seeded for reproducibility
    let seed = 42;
    let mut rng = Isaac64Rng::seed_from_u64(seed);

    // Expected_centroids has shape (n_centroids, n_features)
    // Three points in the 2-dimensional plane
    let expected_centroids = array![[0., 1.], [-10., 20.], [-1., 10.]];
    println!("Initialized Centroid: {:?}\n",expected_centroids);

    // Generates 100 data points for each expected centroid
    // using the random number generator.
    // This creates a synthetic dataset of three clusters of observations.
    let data = generate_blobs(100, &expected_centroids, &mut rng);
    println!("Randomly Generated Data: {:?}\n",data);

    // For this case 3 clusters
    let n_clusters = expected_centroids.len_of(Axis(0));
    println!("Expected Number of Cluster: {:?}\n",n_clusters);

    let observations = DatasetBase::from(data.clone());

    // Configure and run the K-means algorithm
    // The builder pattern to specify the hyperparameters
    // n_clusters is the only mandatory parameter.
    let model = KMeans::params_with_rng(n_clusters, rng.clone())
        .tolerance(1e-2)
        // Fitting the model means finding a set of centroids that
        // minimize the within-cluster variance of the data points
        .fit(&observations)
        .expect("KMeans fitted");
    println!("Resultant Centroid: {:?}",model.centroids());

    // Once we found our set of centroids,
    // we can also assign new points to the nearest cluster
```

```
    let new_observation = DatasetBase::from(array![[20., 20.5]]);

    // Predict returns the **index** of the nearest cluster
    let dataset = model.predict(new_observation);

    // We can retrieve the actual centroid of the closest cluster using
`.centroids()`
    let closest_centroid = &model.centroids().index_axis(Axis(0),
dataset.targets()[0]);
    println!("Closest Centroid of new input data {} :  {}\n",array![20., 20.5],
closest_centroid);
}
```

Appendix D. Linear regression rust code.

```rust
use linfa::prelude::*;
use linfa::traits::{Fit, Predict};
use linfa_linear::LinearRegression;

fn main() {
    // Load  dataset
    let dataset = linfa_datasets::iris();

    // Convert targets from usize to f64 except diabetes
    let dataset = dataset.map_targets(|x| *x as f64);

    // Split the dataset
    let (train, valid) = dataset
    .split_with_ratio(0.8);

    // Train the model
    let model = LinearRegression::default().fit(&train).unwrap();

    // Test the model
    let pred = model.predict(&valid);

    // Calculate the R-squared value
    // R-squared is a measure of how well the model
    // explains the variation in the target variable
    let r2 = pred.r2(&valid).unwrap();
    println!("r2 from prediction: {}", r2);
}
```

Appendix E. Decision tree rust code.

```
// First we have to import the necessary packages
use csv::Reader;
use ndarray::{Array, Array1, Array2};
use linfa::Dataset;
use linfa_trees::DecisionTree;
use linfa::prelude::*;
use csv::ReaderBuilder;

// Main function to read the CSV
fn get_dataset(mut reader: Reader<&[u8]>) -> Dataset<f32, usize, ndarray::Dim<[usize;
1]>> {
    // Extract headers and data
    let headers = get_headers(&mut reader);
    let data = get_data(&mut reader);

    // Calculate the index of the target in the header
    let target_index: usize = headers.len() - 1;

    // Get the features from the header
    let features = headers[0..target_index].to_vec();

    // Get the records and targets from data
    let records = get_records(&data, target_index);
    let targets = get_targets(&data, target_index);

    //Build the dataset with records, targets and features and return it
    return Dataset::new(records, targets)
      .with_feature_names(features);
}

fn get_headers(reader:&mut Reader<&[u8]>) -> Vec<String> {
    return reader
      .headers().unwrap().iter()
      .map(|r| r.to_owned())
      .collect();
}

fn get_records(data: &Vec<Vec<f32>>, target_index: usize) -> Array2<f32> {
    let mut records: Vec<f32> = vec![];
    for record in data.iter() {
      records.extend_from_slice( &record[0..target_index] );
    }
```

```
    // Change if rows file change
  return Array::from( records ).into_shape((308854, 18)).unwrap();
}

fn get_targets(data: &Vec<Vec<f32>>, target_index: usize) -> Array1<usize> {
    let targets: Vec<usize> = data
      .iter()
      .map(|record| record[target_index] as usize)
      .collect::<Vec<usize>>();
     return Array::from( targets );
}

fn get_data(reader:&mut Reader<&[u8]>) -> Vec<Vec<f32>> {
    return reader
      .records()
      .map(|r|
        r
          .unwrap().iter()
          .map(|field| field.parse::<f32>().unwrap())
          .collect::<Vec<f32>>()
      )
      .collect::<Vec<Vec<f32>>>();
}

fn main() {
    // Load CSV content from the include_bytes! macro
    let csv_content = include_str!("CVD.csv");

    // Convert csv_content to a byte slice
    let csv_content_bytes = csv_content.as_bytes();

    // Create a CSV reader from the CSV content byte slice
    let reader = ReaderBuilder::new().from_reader(csv_content_bytes);

    // Begin to process file
    let datan = get_dataset(reader);

  // Split file
    let (train, test) = datan
        .split_with_ratio(0.9);

  // Train the model
    let model = DecisionTree::params().fit(&train).unwrap();

  // Test the model
```

```
    let predictions = model.predict(&test);

    println!("{:?}", predictions);
    println!("{:?}", test.targets);
}
```

Appendix F. Dockerfile.

Docker

```
FROM rust:alpine3.15 AS builder
RUN apk add musl-dev --no-cache
WORKDIR /src
COPY . .
RUN cargo build --release

FROM alpine:3.15
WORKDIR /app
COPY --from=builder /src/target/release/decision_tree .
ENTRYPOINT [ "./decision_tree" ]
```

Docker Wasm

```
FROM scratch
COPY ./target/wasm32-wasi/release/decision_tree.wasm /decision_tree.wasm
ENTRYPOINT [ "decision_tree.wasm" ]
```