Nebula

Comparing two waves of cloud compute

Marius Nilsen Kluften

Thesis submitted for the degree of

Master in Informatics: Programming and System Architecture

60 credits

Institute of Informatics

Faculty of mathematics and natural sciences

University of Oslo

2024

Nebula

Comparing two waves of cloud compute

Marius Nilsen Kluften

Todo list

Nebula

https://duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The ever increasing demand for cloud services has resulted in the expansion of energy-intensive data centers, the ICT industry accounts for about 1 % of global electricity use, highlighting a need for sustainable options in cloud computing architectures.

This thesis investigates WebAssembly, a technology originally intended for running in the browser, as a potential contender in the space of technologies to consider in cloud native applications. Leveraging the inherent efficiency, portability and lower startup times of WebAssembly modules, this thesis presents an approach that aligns with green energy principles, while maintaining performance and scalability, essential for cloud services.

Preliminary findings suggest that programs compiled to WebAssembly modules have reduced startup and runtimes, which hopefully leads to less energy consumption and offering a viable pathway towards a more sustainable cloud.

Acknowledgments

The idea for the topic for this thesis appeared in an episode of the podcast ”Rustacean station”. Matt Butcher, the CEO of Fermyon, told the story of his journey through the different waves of cloud computing, and how Fermyon was founded as a software company that aimed to build a cloud platform with WebAssembly and lead the charge into the ntext big wave of cloud compute.

The capabilities of WebAssembly running on the server, with the aid of the WebAssembly System Interface project, caught my interest and started the snowball that ended up as the avalanche that is this thesis.

I’d like to thank Matt Butcher and the people over at Fermyon for inadvertently inspiring my topic.

Furthermore I’d like to thank my two supervisors Joachim Tilsted Kristensen and Michael Kirkedal Thomsen, whom I somehow managed to convinced to help guide me through such a cutting edge topic. Their guidance and insight have been invaluable the past semesters.

Finally I would like to thank Syrus Akbary, founder of Wasmer, whom I met at WasmIO 2024 who showed me how to reduce my startup times by a further 100 times.

Contents

List of Figures

List of Tables

Part I

Overview

Chapter 1

Introduction

> *If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!*
>
> —Solomon Hykes, *Founder of Docker*

In the digital age, cloud computing has emerged as a foundational technology in the technological landscape, driving innovation and increased efficiency across various sectors. Its growth over the past decade has not only transformed how consumers store, process, and access data, but it has also raised environmental concerns as more and more data centers are built around the globe to accommodate the traffic, consuming vast amounts of power. The Information and Communication Technology (ICT) industry, with cloud computing at its core, accounts for an estimated 2.1% to 3.9% of global greenhouse gas emissions. Data centers, the backbone of cloud computing infrastructures, are responsible for about 200 TWh/yr, or about 1% of the global electricity consumption, a figure projected to escalate, potentially reaching 15% to 30% of electricity consumption in some countries by 2030 (Freitag et al., 2021).

The sustainability of cloud computing is thus under scrutiny, and while some vendors strive to achieve a net-zero carbon footprint for their cloud computing services, many data centers still rely on electricity generated by fossil fuels, a leading contributor to climate change (Mytton, 2020). This reality emphasizes an urgent need to explore alternative technologies that promise enhanced energy efficiency while meeting customers demands. In this vein, serverless computing has emerged as a compelling paradigm, offering scalability and flexibility by enabling functions

to execute in response to requests, rather than having a server running all the time. However, the inherent startup latency associated with containerized serverless functions pose a challenge, particularly for on-demand applications. To mitigate this, vendors often opt for keeping the underlying servers *warm* to keep the startup latency as low as possible for serving functions. Reducing the startup time for serving a function should reduce the need for keeping servers warm and therefore reduce the standby power consumption of serverless architectures.

1.1 Motivation

The environmental footprint of cloud computing, particularly the energy demands of data centers, is a pressing issue. As the digital landscape continues to evolve, the quest for sustainable solutions has never been more critical. This thesis is motivated by the need to reconcile the growing demand for cloud services with the pressing need for environmental sustainability. Through the lens of WebAssembly and WebAssembly System Interface (WASI), this thesis aims to investigate innovative deployment methods that promise to reduce energy consumption without sacrificing performance, thereby contributing to the development of a more sustainable cloud computing ecosystem.

1.2 The Project

This thesis explores Wasm with WASI as an innovative choice for deploying functions to the cloud, through developing a prototype Functions-as-a-Service (FaaS) platform named Nebula. This platform will run functions compiled to Wasm, originally designed for high-performance tasks in web browsers, which coupled with WASI, allows us to give WebAssembly programs access to the underlying system. This holds potential for a more efficient way to package and deploy functions, potentially reducing the startup latency and the overhead associated with traditional serverless platforms. Wasm and WASI offers a pathway, where the demands of today is met, while reducing the carbon footprint for cloud applications.

1.3 Problem Statement

The goal of the thesis is to:

1. Develop a prototype cloud computing platform for the FaaS paradigm.

2. Use this platform for conducting experiments that either prove or disprove the claim that WebAssembly is the more energy efficient choice.

By achieving these goals this thesis seeks to shed light on the feasibility and implications of adopting Wasm and WASI for a greener cloud.

1.4 Outline

The thesis has five chapters; this introduction, a chapter that goes through the background for how cloud computing got to this point, a chapter dedicated to the process of building Nebula, a chapter for discussing the results from the experiments, and ending with a chapter suggesting future works.

Feedback from Michael: When I write this, use it to point the reader to the most important parts of the thesis. Avoid listing the chapters; you already have a table-of-contents.

Chapter 2

Three waves of cloud compute

9 out of 10 cloud providers hate this one simple trick.

Joachim, my supervisor

The evolution of cloud computing represents a transformative adventure, driven by the pursuit for efficiency, scalability and reliability, yet it also poses challenges, notably it's environmental impact. This chapter steps introduces the concept of the "Three waves of cloud computing", coined by the WebAssembly community (Butcher & Dodds, 2024; Leonard, 2024). Where the two first waves of cloud compute represent the shift from virtual machines to containerization, the third wave encompasses utilizing Wasm and WASI to build the next era of cloud compute with the potential to significantly reduce the carbon footprint.

2.1 Ashore: Before the waves

Before delving into the waves themselves, it is useful to understand the landscape that preceded cloud computing. Prior to the cloud era, companies were required to building and mantaining the physical infrastructure for their digital services in-house. This required companies to invest heavily into both expensive hardware and expensive engineers to buy, upkeep and oversee their own physical servers and network hardware. (See Figure 2.1 for an example)

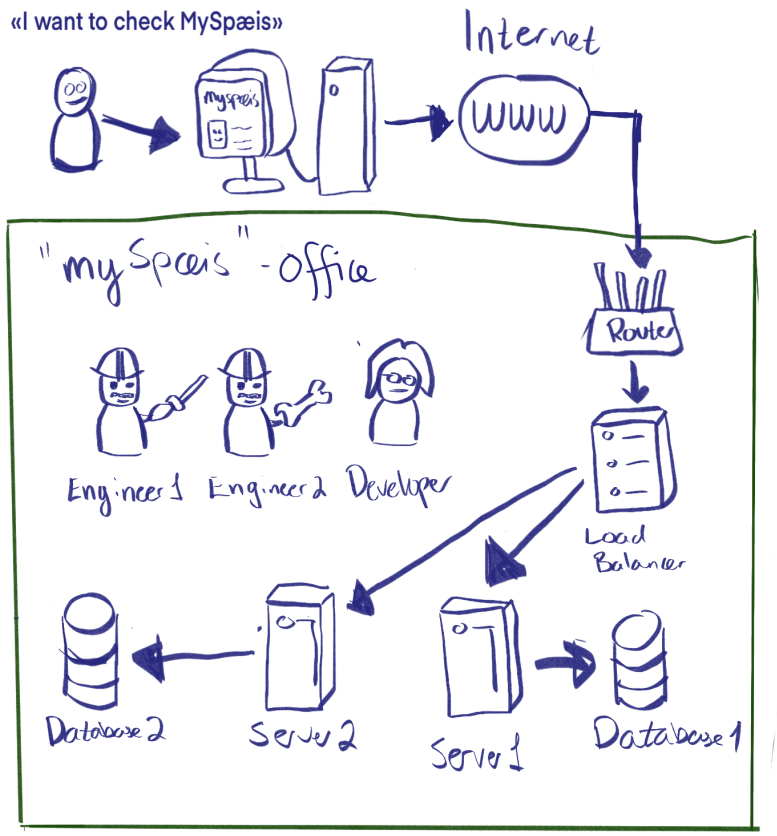

Figure 2.1: Example of a company that host their own infrastructure.

Setting up such an infrastructure comes with a significant cost, both upfront for purchasing such a setup and for employing engineers who can set up and maintain this. This puts a considerable financial strain on organizations, and kept smaller companies without the financial backing to invest in this at a disadvantage (Etro, 2009). Having to maintain infrastructure also poses a challenge when the services start to gain traction. To meet the increased traffic on the services, a company will have to scale up and extend the infrastructure, which also can be expensive and difficult (Armbrust et al., 2010).

As a response to these challenges, some companies found a market for taking on the responsibility of managing infrastructure, and offer Infrastructure-as-a-Service (IaaS) to an evolving market that relies more and more on digital solutions. IaaS provides consumers with the ability to provision computing resources where they can deploy and run software, including operating systems and applications (Mell & Grance, 2011). On these managed infrastructures companies could deploy their services on top of virtual machines that allowed more flexibility, and lowered the bar to new companies.

2.2 The First Wave: Virtual machines

The start of cloud computing can be traced back to the emergence of virtualization, more specifically virtual machines, a response to the costly and complex nature of managing traditional, on-premise data centers. During the mid-2000s, Amazon launched its subsidiary, Amazon Web Services (AWS), who in turn launched Amazon S3 in March 2006, followed by Elastic Compute Cloud (EC2) in August the same year (Barr, 2006). With these services, AWS positioned itself as a pioneer in this space, marking a major turning point in application development and deployment, and popularized cloud computing. EC2, as an IaaS platform, empowered developers to run virtual machines remotely. (See Figure 2.2 for an example of this kind of architecture)

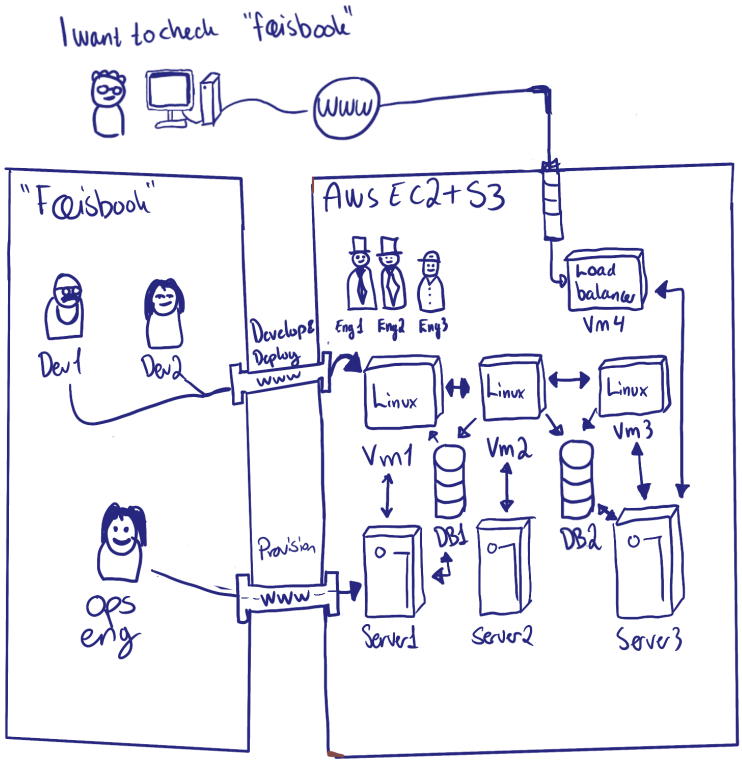

Figure 2.2: Example of "Fæisbook" building their services on EC2 and using S3 for storage.

While similar services existed before 2006, with Amazon's existing large customer base helped them gain significant traction, and ushered in a the first era, or wave, of *cloud computing*.

2.3 The Second Wave: Containers

As we entered the 2010s, the focus shifted from virtual machines to containers, largely due to the limitations of VMs in efficiency, resource utilization, and application deployment speed. Containers, being a lightweight alternative to VMs, designed to overcome these hurdles (Bao et al., 2016).

In contrast to VMs, which require installation of resource-intensive operating systems and minutes to start up, containers along with their required OS components, could start up in seconds. Typically managed by orchestration tools like Kubernetes[1], containers enabled applications to package alongside their required OS components, facilitating scalability in response to varying service loads. Consequently, an increasing number of companies have since established platform teams to build orchestrated developers platforms, thereby simplifying application development in Kubernetes clusters. (See Figure 2.3 for an example where a fictive company `WacDonalds` and their container workflow)

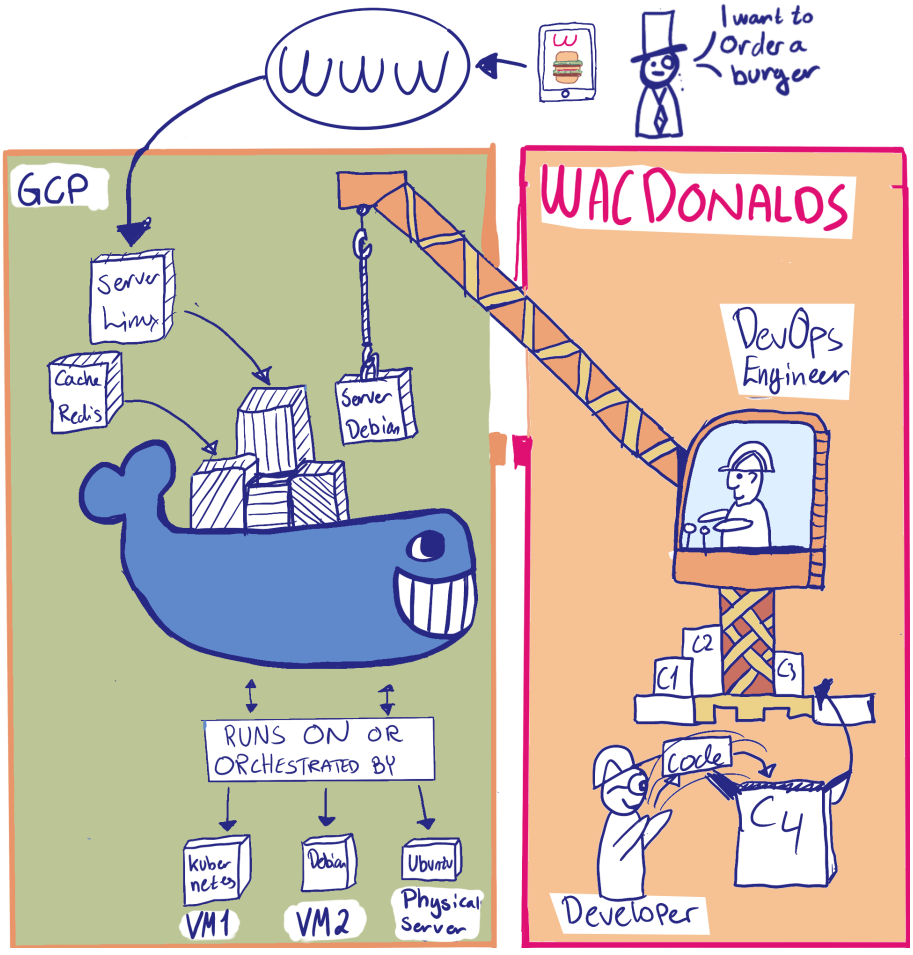

Figure 2.3: DevOps engineer deploying services as containers on GCP.

[1]https://kubernetes.io

Containers are not a perfect solution however, and while they simplify the means of developing and deploying applications, docker images can easily reach Gigabytes in image size (Durieux, 2024), can take a long time to start up, and building applications that target multiple platforms can be difficult.

These solutions are more efficient than manually installing an operating system on a machine, but they still have leave a large footprint. Is there a more efficient way to package and deploy our programs? Wasm and WASI, as mentioned in epigraph of chapter 1, has positioned itself as a potential contender for how applications are built, packaged and deployed to the cloud.

2.4 The Third Wave: WebAssembly modules

WebAssembly has had a surge of popularity the past three to four years when developers discovered that what it was designed for - to truly run safely inside the browser - translated well into a cloud native environment as well. Containers have, with the benefits mentioned in section 2.3, had a positive impact on the cloud native landscape. However, with the limitations - like large image sizes, slow startups and complexity of cross-platform - there is space for exploring alternative technologies for building our cloud-native applications.

WebAssembly is a compilation target with many languages adopting support, and by itself, it is sandboxed to run in a WebAssembly virtual machine (VM) without access to the outside world, meaning that it cannot access the underlying system. This means that a "vanilla" WebAssembly module cannot write to the file system, update a Redis cache or transmit a POST request to another service.

To make this possible, the WebAssembly System Interface project was created. This project allows developers to write code that compiles to WebAssembly that can access the underlying system. This is the key project that turned many developers onto the path of exploring WebAssembly as a potential contender for building cloud applications. With WebAssembly, developers can write programs in a programming language that supports it as a compilation target, such as Rust, C, C++, Go, and build tiny modules that can run on a WebAssembly runtime. These WebAssembly runtimes can run on pretty much any architecture with ease, the resulting binary size are quite small, and the performance is near-native. These perks combined with the potential for reduced overhead, smaller image sizes, and faster startup times make WebAssembly and WASI a promising candidate for the third wave of cloud compute with a lower impact on the environment.

In summary, the three waves of cloud computing - virtual machines, containers,

and now Wasm with WASI - represent the industry's pursuit of more efficient, scalable and reliable solutions for building cloud applications. While each wave has attempted to tackle pressing challenges of its time, it is exciting to see how Wasm and WASI can be leveraged in this third wave and see if it is promise of more efficient applications can lead to reducing the environmental impact of ICT.

Chapter 3

Background

Data has gravity, and that gravity pulls hard

David Flanagan

3.1 Cloud Computing Overview

Cloud computing, commonly referred to as "*the cloud*", refers to the delivery of computing resources served over the internet, as opposed to traditional on-premise hardware setups. The National Institute of Standards and Technology (NIST) defines cloud computing like so:

NIST definition of 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.

(Mell & Grance, 2011)

Cloud computing traces its root back to the 1960s, with the Compatible Time-Sharing System (CTSS) project at MIT, which demonstrated the potential for multiple users accessing and sharing computing resources simultaneously (Crisman, 1963). While CTSS was a localized system, it paved the way for the concept of

shared computing resources, a fundamental principle of cloud computing.

Over the following decades, advancements in networking, virtualization and the ubiquity of the internet led to the development of today's sophisticated cloud services. The term "cloud computing" was first coined in the year 1996 by Compaq, (Favaloro & O'Sullivan, 1996), but it was not until Amazon launched its subsidiary Amazon Web Services (AWS) in the 2006 that the adoption became wide spread.

The launch of AWS's Amazon S3 cloud service in March 2006, followed by Elastic Compute Cloud (EC2) in August the same year (Barr, 2006), marked a major turning point in application development and deployment, and popularized cloud computing. EC2, as an Infrastructure-as-a-Service platform, empowered developers to run virtual machines remotely. By providing these services over the internet on a pay-as-you-go basis, AWS drastically lowered the bar for accessing computing resources, making it easier and more cost-effective for businesses and developers to build and deploy applications without the need for considerable upfront investment in hardware and infrastructure.

With the success of AWS, other major technology companies saw their fit to enter the cloud computing market. In 2008, Google launched the Google App Engine (McDonald, 2008), a platform for building and hosting web applications in Google's data centers. Microsoft followed with the launch of Azure in 2010, its cloud computing platform that offers a range of services comparable to AWS.

The rapid growth of cloud computing also fueled the rise of DevOps practices and containerization technologies like Docker, which facilitate the development, deployment and management of applications on the cloud. Orchestration tools like Docker Swarm and Kubernetes further simplify the process of managing and scaling containerized applications across cloud enviroments (Bernstein, 2014).

Today, cloud computing has become an essential part of modern IT infrastructure, where major cloud provides, like AWS, Microsoft and Google, continue to innovate and expand their offerings. Cloud computing has also enable new paradigms like serverless computing and edge computing, allowing for even more efficient and distributed computing models. (Baldini et al., 2017)

3.2 Energy consumption and Sustainability in Cloud Computing

A major drawback of cloud computing is the increasing energy consumption required to power data centers. With the increased demand for energy consumption,

comes an increased impact on the environment. As mentioned in chapter 1, the ICT industry accounts for an estimated 2.1% to 3.9% of global greenhouse emissions (Freitag et al., 2021). According to the International Energy Agency (IEA), data centers across the globe consumed between from 240 to 340 TWh, accounting for 1 to 1.3% of the global electricity use.

In Norway, for example, Google is constructing a data center in Skien, expected to be fully operational by 2026. As of April 2024, they have been granted a capacity of 240 Megawatts, but they have applied for a total capacity of 860 Megawatts (Rivrud, 2024). At full capacity at 860 MW, Google's data center is aiming to consume 7.5 TWh each year, and according to Google's most recent sustainability report, they consumed a total of 22.29 TWh globally in 2022 ("Google 2023 Environmental Report", 2023). In other words, in 2026 the data center in Skien alone is projected to consume ~33% of the energy Google consumed globally in 2022. The energy consumption in Norway is projected to reach 150-158 TWh in 2026 (Gunnerød, 2022), meaning that the data center in Skien could account for 5% of the energy consumption in the country. Figure 3.1 below illustrates the amount of energy the new data center will consume compared to Norway.

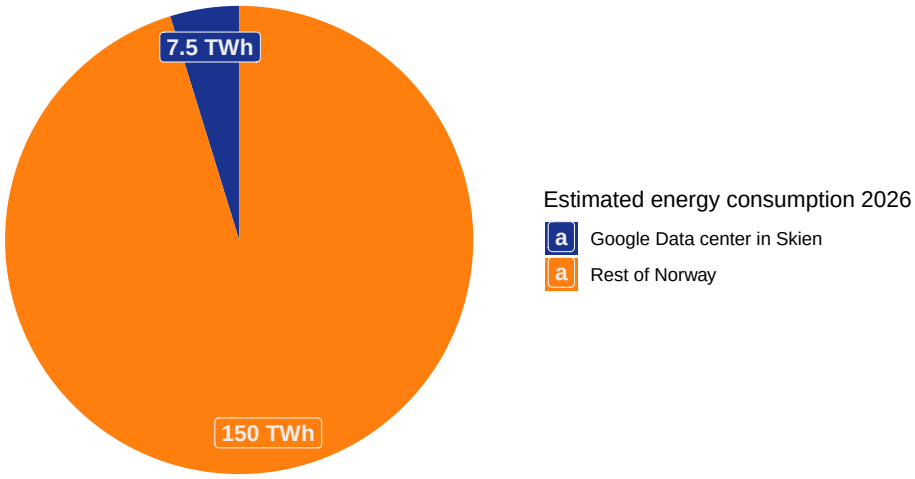

Figure 3.1: Projected energy consumption in 2026: Skien data center and Norway.

Consider this feedback from Michael: Google also does not make it rain more. If Google come and buys all the green energy in the area, it will just mean that others will use more black energy. They don't build new infrastructure. And this is only for Googles "usage". Not the users and network infrastructure.

The flipside of this, is that Google is commited to reach a net-zero carbon footprint by 2030, and the data center in Skien is built to reflect this. Google has been carbon neutral since 2007 and has matched 100% of its annual electricity consumption

with renewable energy since 2017 (Google, n.d.). Norway's abundant hydropower, rising wind power production, investment into solar energy and other renewable energy sources make it an ideal location for building data centers aiming to be powered by renewable energy (Norwegian-Energy, 2023).

Like Google, other major cloud providers have set ambitious targets for renewable energy adoption and have invested in large-scale renewable projects. Microsoft has committed to shifting to 100% renewable energy by 2025 for all its data centers, buildings and campuses, and to be carbon *negative* by 2030 (Smith, 2020), while Amazon has pledged to transition to 100% renewable energy by 2030 for its cloud subsidiary and to have a net-zero carbon footprint by 2040 (Amazon, 2019). These efforts have contributed to reducing greenhouse gas emissions in the cloud computing industry, but this commitment is not universally adopted, and many data centers still rely on electricity generated by fossil fuels, a leading contributor to climate change (Mytton, 2020).

Several factors make up the energy consumption required to service a data center. One of these factors is cooling down the servers while running, and a study from 2017 discovered that cooling accounted for about 38% of total energy consumption in data centers, ranging from 21% to 61% depending the effectiveness of the facility's heating, ventilation, and air conditioning (HVAC) system (Ni & Bai, 2017).

One innovative example of attempting to mitigate the environmental impact of cooling data centers can be found by data center providers like DeepGreen[1], who submerge their servers into dielectric fluid, which gets warmed up by the excess heat of the computers. This heat is then transferred to a host's hot water system via a heat exchange and used for heating up swimming pools in London. Another broad strategy cloud providers opt for is the implementation of power management techniques, such as dynamic voltage and frequency scaling (DVFS), which adjusts the power consumption of servers based on workload demands (Beloglazov et al., 2012).

Virtualization and resource pooling, two key components of cloud computing, also contribute to energy efficiency. By consolidating virtual machines onto shared physical servers, cloud providers are able to improve resource utilization and reduce the energy consumption of their data centers. (Beloglazov et al., 2012)

Feedback fra ingvild: include that data centers also increase the demand for power, so more power demand => more nature gets torn down to accomadate for the increase in demand.

[1]https://deepgreen.energy/faqs/

3.3 Virtualization and Virtual Machines

Virtualization is the process of creating a virtual version of a physical resource such as an operating system, a server, a storage device, or a network resource (Chiueh & Nanda, 2005). Virtualization allow multiple virtual instances to share the underlying physical hardware, enabling more efficient resource utilization and consolidation.

One of the most common forms of virtualization is the creation of virtual machines (VMs). Barham et al. (2003) described that a virtual machine is a software-based emulation of a physical computer system, including its processor, memory, storage, and network interfaces. Furthermore they describe that VMs run on top of a hypervisor, a software layer that manages and allocates the physical hardware resources to the virtual machines.

Figure 3.2 below illustrates virtual machines running in such an environment.

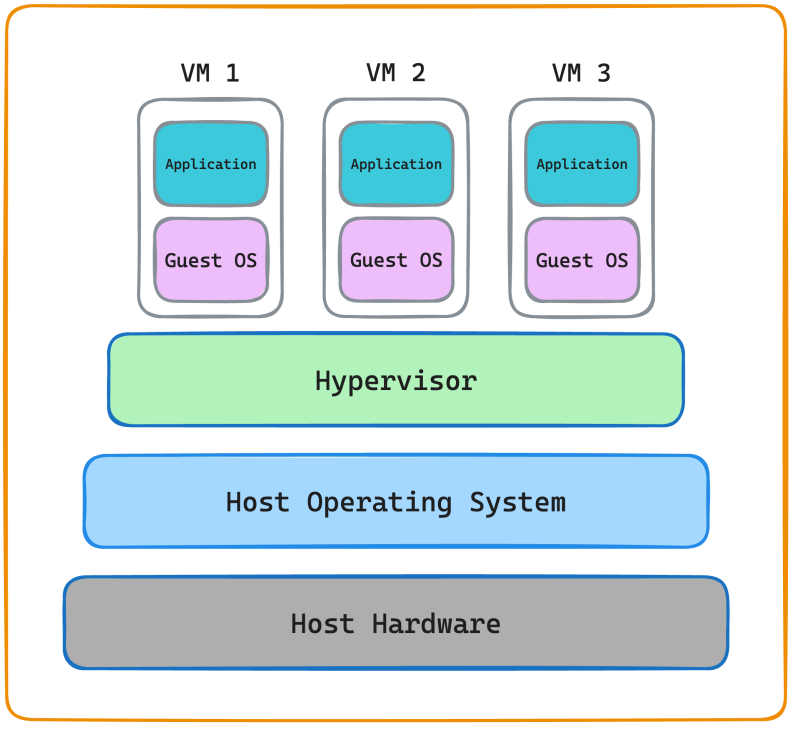

Figure 3.2: Virtual machines running on top of Hypervisor on a computer.

By running several virtual machines on the same physical server, data centers can drastically decrease the number of physical machines physical machines needed to operate, leading to a reduction in energy consumption (Kaplan et al., 2008).

3.4 Containers and Container orchestration

While virtual machines provide isolation at the hardware level, containers offer a more lightweight form of virtualization by isolating applications at the operating system level (Merkel, 2014). Containers share the host operating system's kernel, enabling them to be more lightweight and efficient compared to traditional virtual machines. They can run physical servers, as well as on VMs (Bernstein, 2014).

Merkel (2014) also explains that containers package an application and its dependencies into a single unit. This includes libraries, configuration files and other necessary files. This allows applications to be deployed consistently across different environments, ensuring predictable behavior and reducing compatibility issues (Sergeev et al., 2022)

By enabling more granular and efficient use of system resources, containers contribute to lower energy usage compared to virtual machines due to their lightweight nature and faster startup times (Shirinbab et al., 2020). The adoption of containerization technologies have been shown to optimize energy costs, as containers allow for a higher density of applications running on the same physical hardware without the overhead associated with full virtualization (Cuadrado-Cordero et al., 2018).

Docker is one of the most widely adopted container platforms, providing tools for building, shipping, and running applications in containers (Merkel, 2014). Docker containers are based on open standards and can run on various operating systems and cloud platforms (Sergeev et al., 2022). Figure 3.3 below illustrates how containers run ontop of a Docker Engine on a virtual or physical machine.

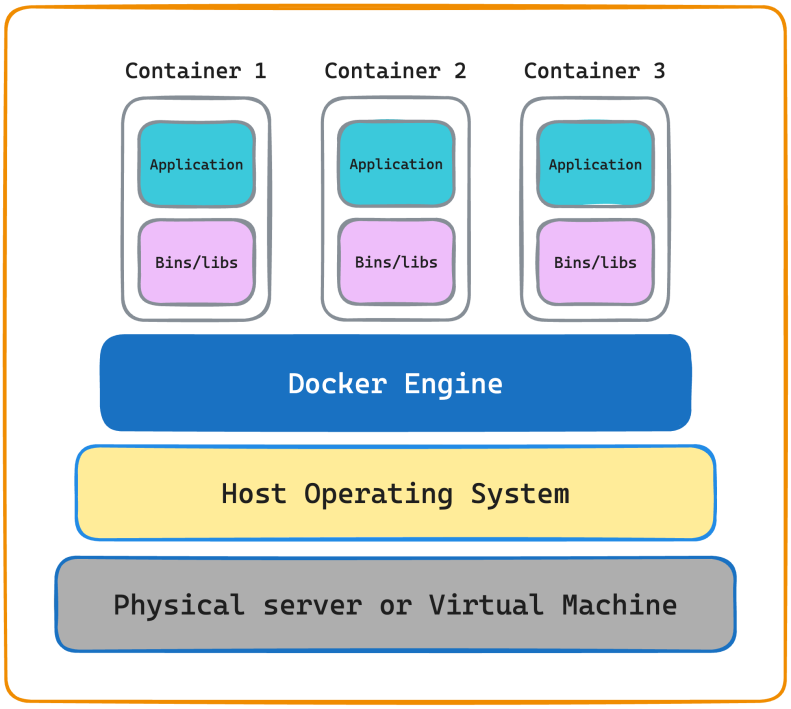

Figure 3.3: Containers running on the Docker Engine

As the number of containers in an environment grows, managing and orchestrating them becomes increasingly complex. Container orchestration tools, such as Kubernetes and Docker Swarm, help automate the deployment, scaling, and management of containerized applications across multiple hosts (Burns et al., 2016).

These tools provide features like:

1. Automated deployment and scaling: Containers can be automatically provisioned, scaled up or down based on demand, and load-balanced across multiple hosts (Burns et al., 2016).
2. Self-healing and monitoring: Orchestration tools can monitor the health of containers and automatically restart or reschedule them in case of failures (Kubernetes, n.d.).
3. Service discovery and load balancing: Applications running in containers can be easily discovered and accessed by other services, enabling microservice architectures (Kubernetes, n.d.).

Container orchestration has become an essential component of modern cloud-native architectures, enabling efficient management and scaling of containerized applications in dynamic environments.

3.5 Serverless Computing

Serverless computing has emerged as an alternative to traditional infrastructure management approahces, such as managing physical servers or building developer platforms as described in Section 3.4 (Baldini et al., 2017). In serverless computing, the underlying infrastrucutre is abstracted away, allowing developers to focus on writing code and deploying programs without the need to provision or manage servers (Baldini et al., 2017; Roberts, 2018). Castro et al. (2019) defines serverless as such:

Serverless definition

> Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scale and billed only for the code running.
>
> (Castro et al., 2019)

According to Baldini et al. (2017), in a serverless model, the cloud provider is responsible for managing the infrastructure, including resource allocation, scaling, and maintenance. This approach enables more efficient resource utilization, as resources are dynamically allocated based on demand. Furthermore, developers can deploy containers or functions independently, promoting modularity and scalability.

On top of this, severless computing offers several more benefits, including:

1. Reduced operational overhead: Developers do not need to manage servers or infrastructure, freeing up time and resources for application development (Baldini et al., 2017).
2. Faster time-to-market: With serverless, developers can quickly deploy and iterate on functions without the need for time consuming infrastructure setup (Adzic & Chatley, 2017).
3. Cost efficiency: Serverless platforms typically employ a pay-per-use pricing model, where users are charged based on the actual execution time and resources consumed by their applications (Eismann et al., 2021).

However, serverless architectures also introduce certain challenges, such as:

1. Potential vendor lock-in: Serverless platforms may have provider-specific

APIs and services, which can make it challenging to change providers or migrate applications (Gottlieb, 2018).

2. Cold start latencies: Containers or functions that have not beeen invoked recently may experience longer startup times, cold starts, which can impact application performance (Golec et al., 2023).
3. Resource overhead and efficiency: Serverless platforms typically rely on containerization technologies, such as Docker, to encapsulate and isolate function executions. The use of containers can introduce resource overhead and impact the efficiency of serverless applications (Akkus et al., 2018).

Examples of widely used platforms that build on the serverless model can be found at the major cloud providers, like Google Cloud Run[2], AWS Fargate[3] and Microsoft's Azure Container Instances (ACI)[4]. These three example platforms allow developers to deploy containers onto the cloud without worrying about orchestration behind the scenes.

3.5.1 Functions-as-a-Service

FaaS is a cloud computing model, derived from serverless, that allows developers to execute individual functions in response to events or triggers without needing to manage the underlying infrastructure (Sewak & Singh, 2018).

Examples of these FaaS platforms among the big three cloud providers are; AWS Lambda[5], Google Cloud Functions[6], and Microsoft's Azure Functions[7]. On FaaS platforms like these, developers write and deploy small, self-contained functions that perform specific tasks. These functions are typically stateless and can be written in various programming languages supported by the FaaS provider (Baldini et al., 2017).

When a function is triggered, the FaaS platform automatically allocates the necessary resources to execute the function, such as CPU, memory, and network bandwidth. The platform also handles the scaling of the function based on the incoming requests, ensuring that the function can handle varying workloads by itself (McGrath & Brenner, 2017).

FaaS platforms often utilize container technology, like Docker, to provide an isolated environment for each function execution. Containers offer several benefits,

[2]https://cloud.google.com/run
[3]https://aws.amazon.com/fargate/
[4]https://azure.microsoft.com/en-us/products/container-instances
[5]https://aws.amazon.com/lambda/
[6]https://cloud.google.com/functions
[7]https://azure.microsoft.com/en-us/products/functions

such as fast startup times, efficient resource utilization, and the ability to package functions with their dependencies (Van Eyk et al., 2018). However, the use of containers in FaaS also introduce some challenges including, cold start latency and the overhead associated with container initialization and management (Wang et al., 2018).

Cold start latency refers to the time it takes for a FaaS platform to provision a new container instance when a function is invoked after a period of inactivity. This latency can be significant, especially for applications with strict performance requirements (Wang et al., 2018). The limitations of containers in FaaS environments have led to the exploration of alternative approaches, such as using Wasm and {WASI}. As Solomon Hykes, the creator of Docker, stated:

> *"If WASM+WASI existed in 2008, we wouldn't have needed to create Docker. That's how important it is. WebAssembly on the server is the future of cloud computing. A standardized system interface was the missing link. Let's hope WASI is up to the task." (Solomon Hykes [@solomonstre], 2019)*

This statement highlights the potential of Wasm and WASI to face the challenges with containers in FaaS and to provide a more efficient and platform-agnostic approach to serverless computing, a potential explored and supported by Kjorveziroski et al. (2022).

3.6 WebAssembly and WASI

WebAssembly, commonly referred to as Wasm, is a modern binary instruction format that has risen to prominence as a versatile technology across a diverse amount of computing environments, originating in the web browser. This section introduces the project that WebAssembly evolved from - *asm.js* - and illustrate how WebAssembly lets developers write programs in a high-level language and run them across a multitude of platforms.

3.6.1 asm.js

Mozilla released the first version of asm.js in 2013, and it was designed be a subset of JavaScript, allowing web applications written in other languages than JavaScript, such as C or C++, to run in the browser. The intention of asm.js was to enable web applications to run at performance closer to native code than applications written in standard JavaScript. A simplified flow for how source code written in C/C++ is

compiled to bytecode that can be executed in the browser can be found in figure 3.4 below.

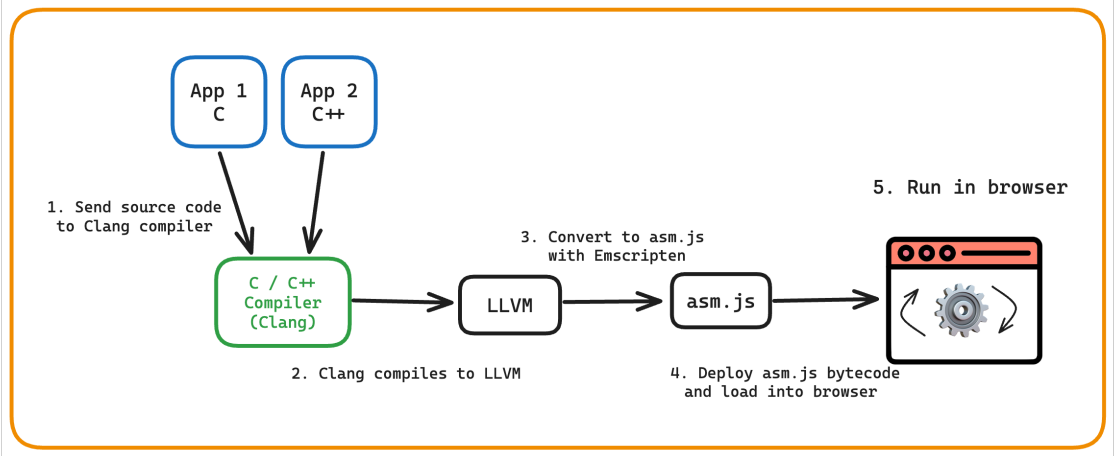

Figure 3.4: Source code in C/C++ compiled to asm.js and run in browser

While asm.js was a great leap forward, being a subset of JavaScript limited its scope, leading to its deprecation in 2017 and the development of a more efficient and portable format (WebAssembly.org, n.d.).

3.6.2 WebAssembly

The team at Mozilla built upon the lessons learned from asm.js and went on to develop WebAssembly, launching the first public version in 2017. WebAssembly is a low-level code format designed to serve as a compilation target for high-level programming languages (Haas et al., 2017). It is a binary format that gets executed by a stack-based virtual machine, comparable to how Java bytecode runs on the Java Virtual Machine (VM) (Haas et al., 2017). In Figure 3.5 below, a simplified flow for compiling web applications written in different languages can be compiled and run inside a web browser.

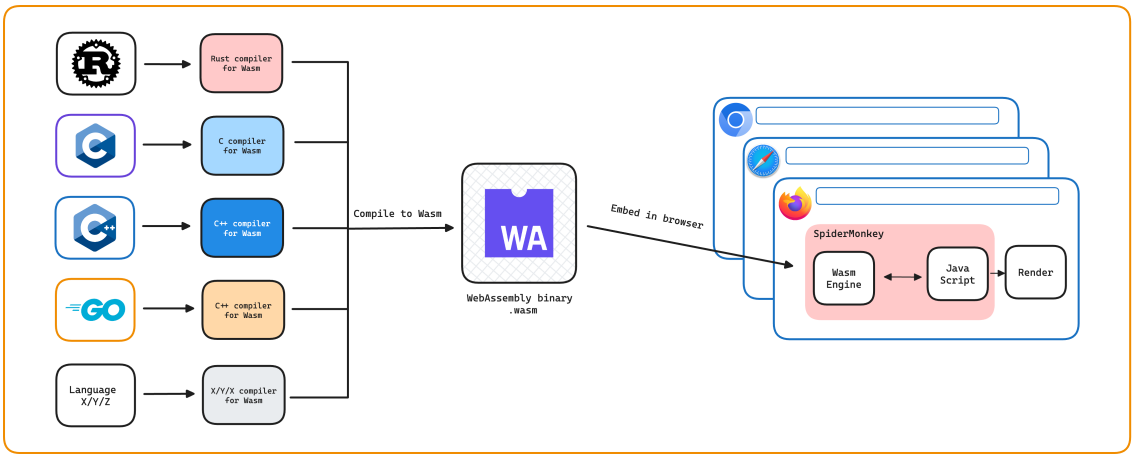

Figure 3.5: Source code compiled to WebAssembly and embedded in browser

According to Haas et al. (2017), some key features that the team behind Wasm sought out to implement were:

Table 3.1: WebAssembly features

Feature	**Description**
Portability	Wasm can run on different hardware and operating systems due to its hardware-independent design and sandboxed execution environment.
High Performance	Wasm aims to achieve near-native performance by leveraging hardware capabilities and ahead-of-time compilation.
Security	Wasm modules run in a sandboxed environment, isolated from the host system, with strict memory and control flow limits, mitigating security vulnerabilities.
Modular design	As an open standard with a modular design, Wasm can be extended and integrated with various programming languages and tools, enabling broad applications beyond web browsers.
Efficient compression	Wasm's binary format is designed for efficient compression, reducing code size and improving download times, especially on resource-constrained devices.

By addressing performance, security and portability concerns, Wasm offers an alternative to traditional approaches for running untrusted code on the web and in

other computing environments, such as cloud and edge computing (Haas et al., 2017).

3.6.3 WebAssembly System Interface

While Wasm was initially designed to run in web browsers, its potential for use in other environments, such as cloud and edge compuitng, led to the development of the WebAssembly System Interface. WASI is a modular system interface that provides a standardized set of functions for interacting with the host operating system, enabling Wasm modules to run outside of web browsers ("WASI.Dev", n.d.).

WASI defines a set of APIs for performing tasks like file system operations, networking, and other system-level operations, allowing Wasm modules to be portable across different platforms and environments ("WASI.Dev", n.d.). It's first iteration aimed to be implement as many POXIS-like features as possible, but has since extended beyond this and in their latest Preview 2, they have implemented support for websockets and HTTP interfaces ("WASI/Preview2/README.Md at Main · WebAssembly/WASI", n.d.).

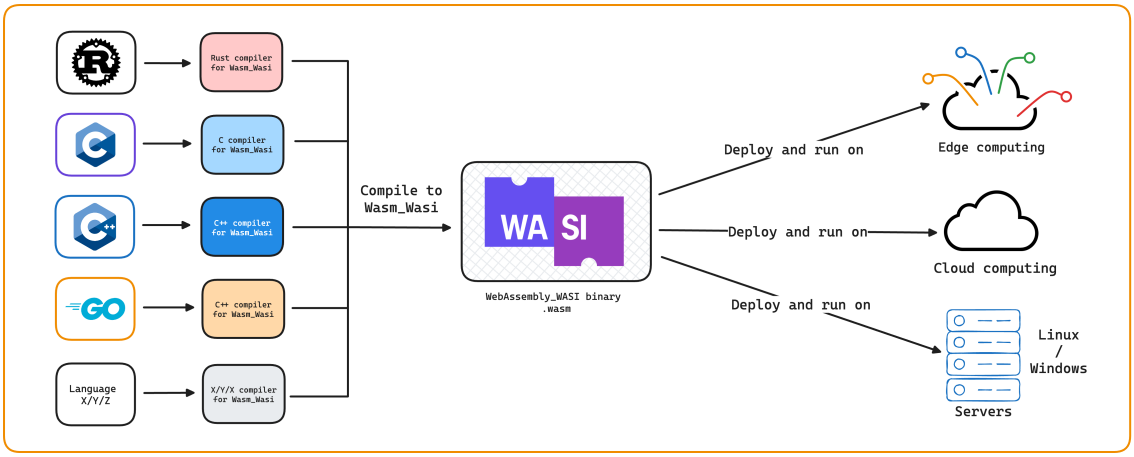

Figure 3.6: Source code compiled to wasm_wasi32 and deployed on platforms that support running the binaries.

3.6.4 WebAssembly runtimes

WebAssembly modules cannot run independently; they require a runtime environment to interpret and execute them. Several WebAssembly runtimes have been developed to support the execution of WebAssembly modules in different environments, such as Wasmtime, Wasmer, and WasmEdge (Zhang et al., 2024).

Zhang et al. (2024) explains that these runtimes provide a secure and efficient

execution environment for WebAssembly modules, enabling them to run on a wide range of platforms, including cloud servers, edge devices, and even embedded systems. They explain that some runtimes offer features like ahead-of-time (AOT) compilation, which can further improve the performance of WebAssembly modules. Figure 3.7 below illustrates how code written in a programming languages with support for compiling to Wasm can run cross-platform.

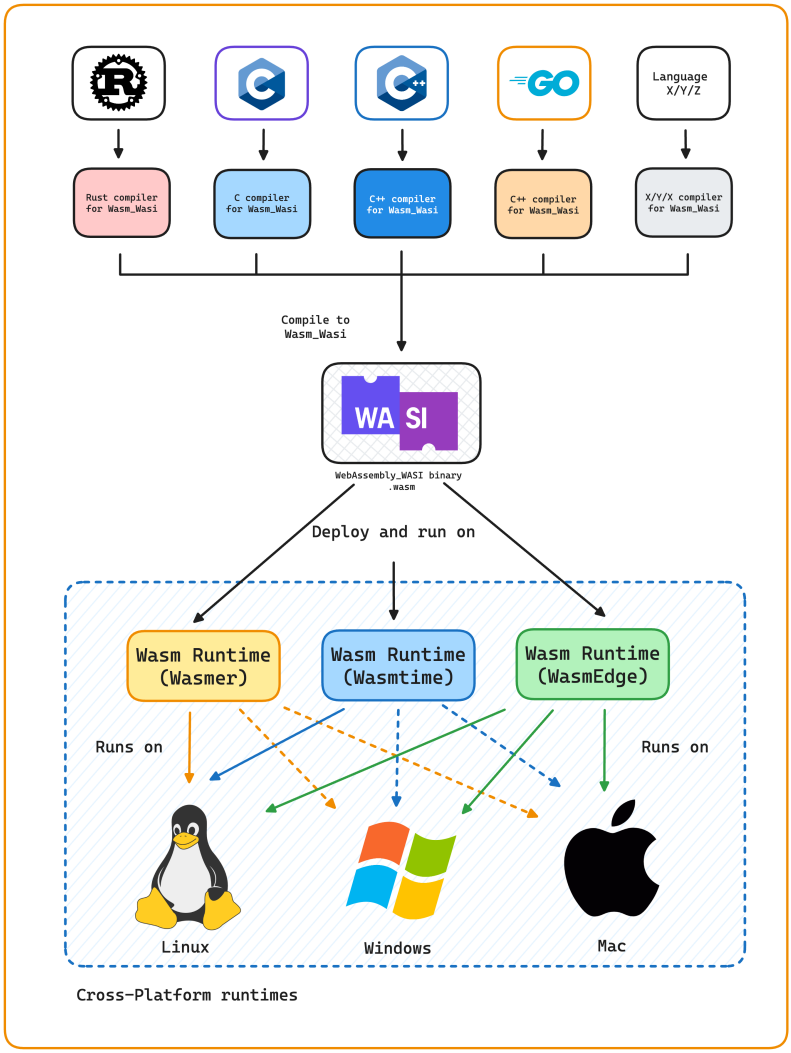

Figure 3.7: Source code compiled to wasm_wasi32 that can run anywhere a WebAssembly runtime can be installed.

The combination of WebAssembly, WASI, and efficient runtimes has sparked interest in using WebAssembly as an alternative to traditional containerization technologies, such as Docker, in cloud-native and serverless environments (Sebrechts et al., 2022; Shillaker & Pietzuch, 2020).

3.7 Energy monitoring

Monitoring and measuring energy consumption can be useful for understanding and optimizing the energy efficiency of cloud computing environments. Energy measurements provide insights into the impact of different technologies, architectures, and workloads on overall energy usage (Al-Fuqaha et al., 2015; Shehabi et al., 2016).

Several protocols and techniques have been explored to collect and analyze energy consumption data, and this thesis will explore some of these. These protocols enable the transmission of energy data, which can be used for optimizing energy efficiency (Al-Fuqaha et al., 2015).

3.7.1 MQTT

The Message Queuing Telemetry Transport (MQTT) protocol is a lightweight and efficient protocol that has been used for energy monitoring. In an MQTT setup, a broker server acts as an intermediary, facilitating communication between devices and servers. This enables efficient data exchange between publishers and subscribers (Al-Fuqaha et al., 2015). (See Figure 3.8).

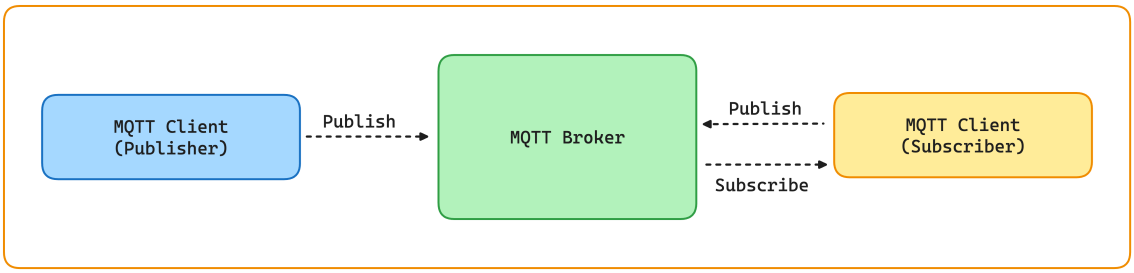

Figure 3.8: PubSub model of the MQTT protocol.

3.7.2 Z-Wave

Z-Wave is a wireless communication protocol designed for home automation and energy management. It has been used in research studies to develop energy monitoring systems for residential buildings, enabling monitoring and control of energy consumption (Al-Fuqaha et al., 2015).

3.7.3 Aeotec Smart Switch

The Aeotec Smart Switch 6 is a power switch that is primarily used for home automation. It allows its users to integrate with it through a Aeotec Z-Stick what

elaborate on why zwave is relevant. In the end, I ended up relying on modbus be-

communicates with the switch through Z-wave. A supporting library; zwave-js-ui[8], which can be setup as a Docker container on a computer, can read energy measurements from the Smart Switch at a given interval and publish the values on a MQTT-topic. This is in turn picked up by the MQTT broker and published to its subscribers. This switch supports reporting measurements at an interval of 1 second (Aeotec, n.d.).

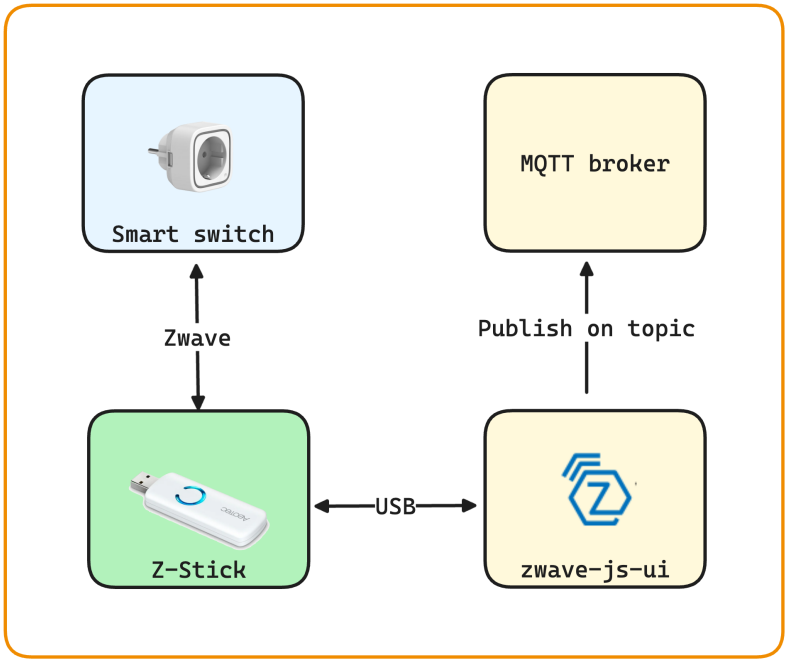

Figure 3.9: Smart Switch 6 communicating with zwave-js-ui through Aeotec Smart Stick

I ended up not using this in my final project, but I still spent considerable time getting it to work. Should I just cut it out and focus on modbus tcp, or keep it in the background?

3.7.4 Modbus TCP

An alternative to reading energy measurements through a pub-sub model with MQTT, is to utilize the Modbus TCP protocol. Modbus TCP is a widely adopted industrial communication protocol for exchanging data between devices and control systems. It has been used to develop energy monitoring and auditing systems for industrial applications, facilitating energy audits and energy-saving strategies (Tong et al., 2015). It communicates with TCP/IP packets, and an illustration on how a client-server request and response could look like can be found in Figure 3.10.

[8]https://github.com/zwave-js/zwave-js-ui

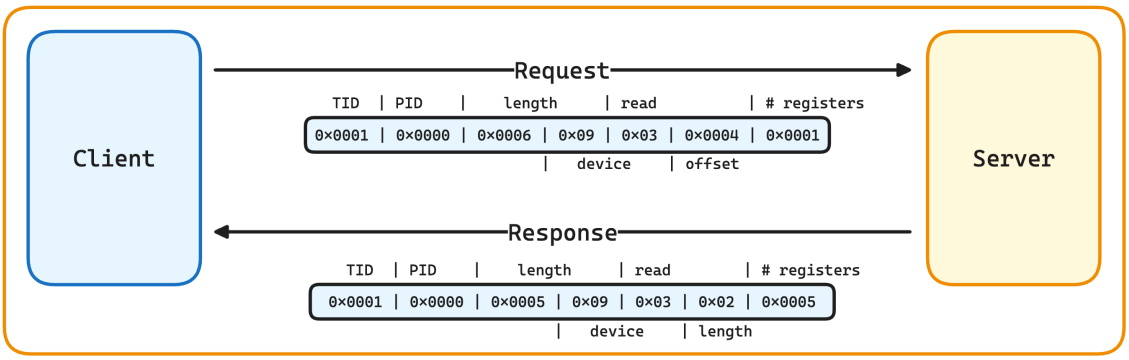

Figure 3.10: TCP/IP package exchange between a client and a server over Modbus TCP.

3.7.5 Gude Expert Power Control 1105

The Gude Expert Power Control 1105 is a switched PDU with an integrated current metering and monitoring that supports communicating over TCP/IP. It is effectively a light weight server that can measure energy, current, power factor, phase angle, voltage, and active / apparent / reactive power. To read these measurements, the PDU supports a handful of interfaces, such as REST API, HTTPS, SNMP, Telnet, MQTT and Modbus TCP (GmbH, 2023). In Figure 3.11 below, a simplified setup with the Gude acting as the server for communicating over Modbus TCP is shown.

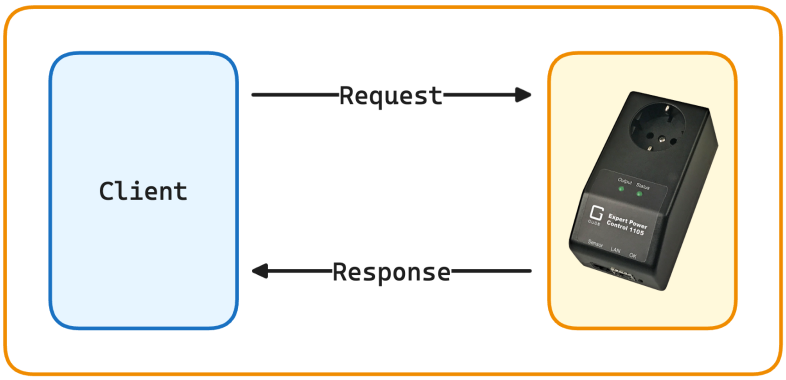

Figure 3.11: Client communicating with Gude over Modbus TCP.

Part II

Project

Chapter 4

Methodology

To reiterate, the goal of this thesis is to investigate the two problem statements presented in Section 1.3. We are going to develop a FaaS prototype, named Nebula, which we're going to use for conducting experiments to see if the claim that Wasm modules offer a more efficient way to develop and deploy our cloud native applications holds true. On this platform we are going to deploy functions written in Rust that are either compiled to Wasm modules or compiled to Rust binaries that get packaged into a Docker image.

This chapter will present the intended methodology on how we will conduct our experimental research method. We will employ an experimental research method, where we will build a prototype, perform experiments on it in a controlled environment, collect measurements and compare the results.

4.1 Experimental framework

The main focus of this thesis is to build a prototype and perform experiments on it, to measure the capabilities of Wasm modules as a model for deploying and running cloud applications. An experimental approach was chosen to test the problem statements, as we can have controlled manipulation of variables, measure the outcomes, and compare directly between two deployment environments; Wasm modules and Docker containers.

We will perform a series of controlled experiments designed to measure startup latencies (cold start), total runtime, and energy consumption for each invocation of a function. This allows for a precise evaluation of how Wasm modules compare against traditional container-based deployments in the context of cloud-native applications.

To capture energy measurements, we will need to use hardware locally that we

can control. For this we will look at the protocols mentioned in Section 3.7, and use a Raspberry Pi as our local deployment environment. We will also capture startup and runtime data on these invocations, but not many cloud applications today are deployed on Raspberry Pi's, so we will also need to deploy Nebula on a more typical setup.

4.1.1 Prototyping

The prototype we're going to build for this project will be named Nebula, named after the celestial phenomenon that can be observed in the form of a giant cloud out in space. On Nebula we will develop a way to deploy and invoke functions in the form of Wasm modules or as spun up Docker containers. To perform the desired experiments, the prototype will need to be able to deploy to both a Raspberry Pi 4b, and on a typical machine hosted on the cloud. For this, we will be using NREC , a IaaS platform available to students, where we can spin up a Debian virtual machine and set up a service on a public IP address.

add citation/link

4.1.1.1 Hardware specifications

Experiments will be performed on two types of hardware:

1. Raspberry Pi: A small Single board computer (SBC)s with an ARM64 architecture that can run flavors of Linux. This device will be used to measure energy consumption relative to function invocations, which it is well suited for, as it consumes small amounts of energy by itself, making it ideal for comparing power consumption under actual load (Bekaroo & Santokhee, 2016).
2. Virtual Machine: A Debian-based VM running on an IaaS platform (e.g., NREC), to measure performance relative to what a standard cloud native application would perform.

4.1.1.2 Nebula requirements

Based on this, we can summarize what we need Nebula to be able to meet these requirements.

4.1.2 Benchmarking

Benchmarking will be employed as a research method to evaluate the performance of different configurations. Benchmark functions representing various computational workloads will be implemented and executed on both environments.

Table 4.1: Nebula requirements

Requirement	Description
Function deployment	We should be able to deploy functions to Nebula.
Function execution	The prototype should execute functions in response to user invocations, either as Wasm modules or as Docker containers.
Measurement capabilities	The prototype should be able to collect data for startup latencies, total runtime, and energy consumption for each function invocation.
Portability	The prototype should be deployable on both a Raspberry Pi for local testing and measurement, and on a Debian virtual machine hosted on an IaaS platform (e.g., NREC) for a more typical cloud setup.
Performance	The prototype should be built in such a manner that it presents little overhead when running our functions, isolating the resulting efficiency and energy readings to each function invocation.

4.1.3 Controlled experimentation

Controlled experimentation will be an important aspect of this research, as it let us isolate and study the effects of the deployment environment on performance and energy consumption. Various factors, such as input parameters for the benchmark functions and hardware configurations, will be controlled or kept constant across experiments to minimize the influence of external factors and increase the validity and reliability of the results.

To experiment on Nebula, we will craft a set of benchmark functions. As the input value to these functions scale up, the computational load on the server will increase, letting us measure the execution time across different workloads.

4.1.4 Benchmark functions

A total of four benchmarking functions will be selected for testing, representing various computational workloads. These functions will be written in Rust and compiled to both Wasm modules and Rust binaries packaged in Docker images. The selection of these functions is based on their computational intensity and their ability to stress the CPU.

The functions we will implement are:

1. *Fibonacci*: This function calculates the *nth* Fibonacci number in the sequence, following the formula: $F_n = F_{n-1} + F_{n-2}$.
2. *Exponential*: This function calculates the value of Euler's number raised to the nth power, following the formula: $F_n = e^n$. This benchmark tests the performance of the prototype in handling intesive floating-point calculations.
3. *Factorial*: This function calculates the sum of the factorial of *n*, following the formula: $n! = \prod_{k=1}^{n} k$.
4. *Prime number*: This function finds the nth prime number. While it is harder to represent this in a mathematical formula, a brute-force approach will be used to stress the CPU and emulate a more typical scenario. The function will loop through a range of numbers (0 through $2^{64} - 1$), check if the number is a prime number, and append prime numbers to a list until the nth prime number is found.

By implementing, deploying and executing these functions as both Wasm modules and Docker containers, the research aims to provide a thorough evaluation of the performance and efficiency of each environment.

4.1.5 Measurement and Data Collection

To quantify the efficiency and energy consumption associated with function invocations on each environment, empirical measurement techniques will be employed. We will deploy Nebula, along with the Wasm modules and Docker images, to our target computers. For measuring startup and runtimes of each function akin to traditional deployments, we will deploy Nebula and the functions to a virtual machine running Debian on NREC. While a Raspberry Pi 4 will serve as our deployment target for measuring startup latencies and runtimes of a device more akin to embedded systems, but more importantly, allowing us to measure energy consumption on a controlled device.

The Raspberry Pi will be connected to a power supply that can report energy readings, enabling the collection of energy consumption data during function execution. See Figure 4.1 below for a rough sketch on how we will set up benchmarking and energy measurements for our experiments.

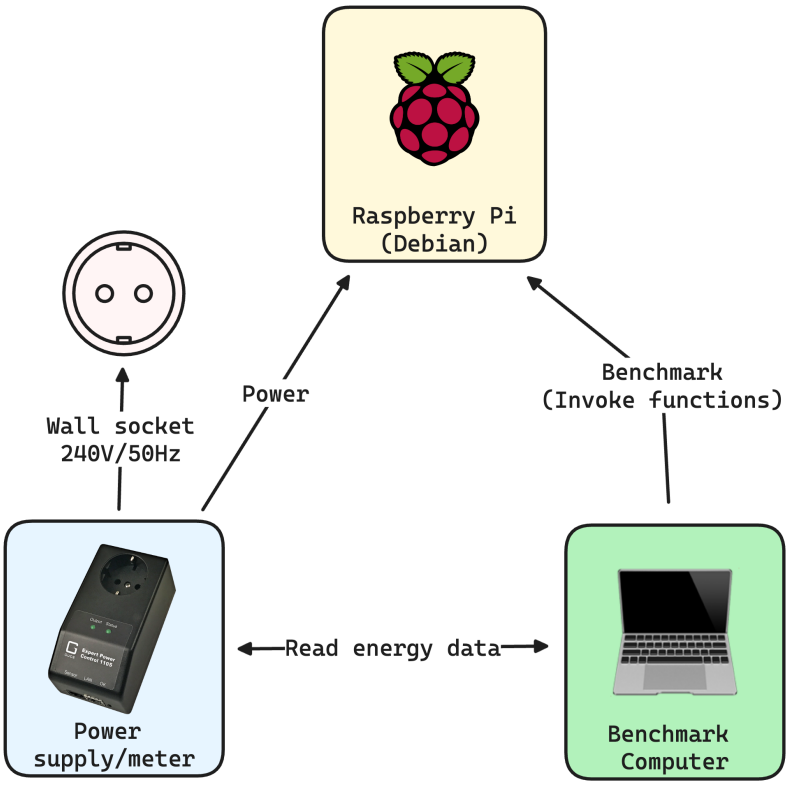

Figure 4.1: Reading energy consumption of our Raspberry Pi.

Throughout the experiments, we will collect various data points to evaluate the performance and energy efficiency of both Wasm modules and Docker containers.

To isolate how much power our functions invoke on top of the idle load of the underlying system that hosts Nebula, we will also benchmark a baseline power level, to measure how much power Nebula consumes while idle. Any power consumed above this should then reflect how much power each function invokes when called.

The following measurements will be recorded on both hardware configurations:

- Startup latency / Cold start: For each function invocation, the time taken from the intial request is received until the function start executing will be measured. This startup latency, known as cold start, is a critical metric for evaluating the responsiveness and scalability of serverless function deployments.
- Total runtime: The total execution time of each function will be measured, from the same time we start measuring cold start until the function returns the result will be measured. This metric will let us compare the computational efficiency of each environment under different workloads.

While these additional measurements will be gathered on our local system as seen in Figure 4.1:

- Power consumption: During the function execution on the Raspberry Pi, we will measure the power the server consumes at regular intervals. To get a more accurate reading, we will measure the current (I) in mA and the power factor (PF), while assuming the voltage (U) is a constant 240V. Using these numbers we will calculate the power with the following equation:

  Explain wtf PF is.

  $$P = U \times I \times PF$$

  Two sets of power consumption data will be collected.

  - Average Power: The average power consumption of the Raspberry Pi device, including any background processes, including Nebula, or idle draw, while invoking a function. This will be an estimation based on energy readings read during the lifetime of each function, and getting the average of these.
  - Isolated Average Power: An estimation on how much extra power consumption a function incurs, this will be a sum of Average power minus the Baseline power. $P_{\text{iso}} = P_{\text{total}} - P_{\text{baseline}}$

- Energy Consumption: Based on the power consumption measurements and the total runtime durations, the energy consumption for each function invocation will be calculated. We will calculate the energy consumption with the following formula:

  $$E_{\text{Wh}} = P_{\text{W}} \times t_{\text{h}}$$

  Two sets of energy consumption values will be derived:

  - Energy consumption: The total energy in Wh consumed of the entire Raspberry Pi device during function execution. The unit prefix will be adjusted according to the results.
  - Isolated Energy consumption: The energy in Wh consumed by the function, with the baseline power subtracted from the base W used to calculate this.

continue here

4.1.6 Comparative analysis

1. Deployment Environments: The choice of Wasm modules and Docker containers as the primary environments for deployment is aimed at testing the core capabilities of Wasm in a cloud-native setting, contrasting it against the well-established container technology. In a sense the experiments will compare two waves cloud compute.

2. Performance metrics: Metrics have been chosen to provide quantitative data on the efficiency and scalability of each technology. This includes measuring the time to initialize (cold start), the time to execute tasks (runtime), and the energy required for operations.

The following sections will dive deeper into the specific methodologies that will be employed for this thesis research.

A comparative anlysis will be performed to evaluate the cold starts, runtime, and the energy consumption of function invocations between the two deployment environments. Specific metrics, such as startup latency, execution time, and energy consumption per invocation, will be used for this comparison to assess the overall performance and energy efficiency of each deployment environment for serverless function execution.

4.1.7 Reliability and Validity

To ensure the reliability and validity of the results, the following measures will be taken: controlled experimentation will be used to minimize the influence of external factors; the experiments will be repeared multiple times to ensure consistency of the results; the data will be analyzed using statistical techniques to identify any patterns or trends; and the results will be validated by comparing them to existing research in the field, explored in Section 8.1.

```
Marius thoughts:

Some aspects of cloud-native applications, like "impure" functionality is
discounted for this research. The functions don't read to/fro the underlying
system or perform intensive I/O operations. This is to compare functions in more
of a controlled and isolated manner, where the time it takes to invoke a
function in wasm focuses on the calculation itself. This is also true for the
Docker images, so it's still a fair assessment.
```

Chapter 5

Designing Nebula

This chapter will present the design and overall architecture of Nebula, a FaaS prototype that can spin up functions both as Wasm modules, and as Docker images. The core components of the prototype will be presented, and by the end of this chapter, the reader should have a clear understanding on how Nebula was designed, how its core features were chosen for implementation and how one can interact with the prototype.

5.1 A word on this chapter

This is a prototype FaaS platform that is meant to illustrate the raw differences between two waves of cloud compute; Wasm modules and container. It's scope will be limited by the fact that it is a single students thesis project, and will naturlly lack many features that large-scale FaaS platforms typically have. None of the major cloud providers have open sourced their FaaS platform, meaning that exactly how they function behind the scenes is difficult to surmise. This prototype therefore aims to hone in on the aspect of deploying functions as containers and compare it with deploying functions as Wasm modules.

5.2 Nebula overview

This section will provide an overview of Nebula's architecture, including: - A high-level description of the system components and their relationship.

- An explanation of how users can interact with Nebula.
- a brief discussion of the design trade-offs made during development.

5.3 Core components

The following sections will delve into the individual core components of Nebula:

5.3.1 Web server

5.3.2 Function deployment

How functions are packaged, deployed and managed on the server.

5.3.3 Wasm Module execution

The process by which Wasm modules are executed on Nebula.

5.3.4 Docker Image execution

The process by which Docker images are executed on Nebula.

5.3.5 Orchestration and Management

How Nebula manages the execution of functions, including passing input/output and collecting metrics

5.4 Design rationale

Explain that this section provide an explanation of design decisions made during development of Nebula

5.5 Designing energy readings

Chapter 6

Implementing Nebula

This chapter provides the details of the implementation of Nebula.

6.1 Requirements

Part III

Results

Chapter 7

Evaulation

Chapter 8

Discussion

> *"So will wasm replace Docker?" No, but imagine a future where Docker runs linux containers, windows containers and wasm containers side by side. Over time wasm might become the most popular container type. Docker will love them all equally, and run it all :)*
>
> —Solomon Hykes, *Founder of Docker*

8.1 Related work

Flesh out about qian's study, which was an inspiration for my setup.

Qian (2022) wrote a dissertation at the University of Bristol where they

There are several cloud providers who have added support for running Wasm on their cloud offerings. Cloudflare Workers[1], and Fastly Compute@Edge[2] have added support, while Fermyon have built an entire cloud architecture around the binary format with their Fermyon Cloud[3]. Fermyon has also open sourced their developer tool Spin[4], a tool that lets developers create, build and test applications written in a supported language[5] and deploy it to a Fermyon Platform instance, either self-hosted or on their own service.

Sebrechts et al. (2022) wrote a paper where they described their research in developing a Wasm-based framework for running lightweight controllers on Kubernetes. In their research they found that with their novel Wasm-based operators were able to reduce memory footprint of 100 synthetic operators from 1405MiB to 227MiB

[1] https://developers.cloudflare.com/workers/runtime-apis/webassembly/
[2] https://www.fastly.com/products/edge-compute/serverless
[3] https://www.fermyon.com/cloud
[4] https://www.fermyon.com/spin
[5] https://developer.fermyon.com/spin/v2/language-support-overview

and from 1131MiB to 86MiB of idle operators, compared to traditional operators. One of their conclusions is that using Wasm proved to be more memory efficient than the container-based solution. See Figure 8.1 for their findings on memory usage with Wasm-based operators to Rust and Golang operators.

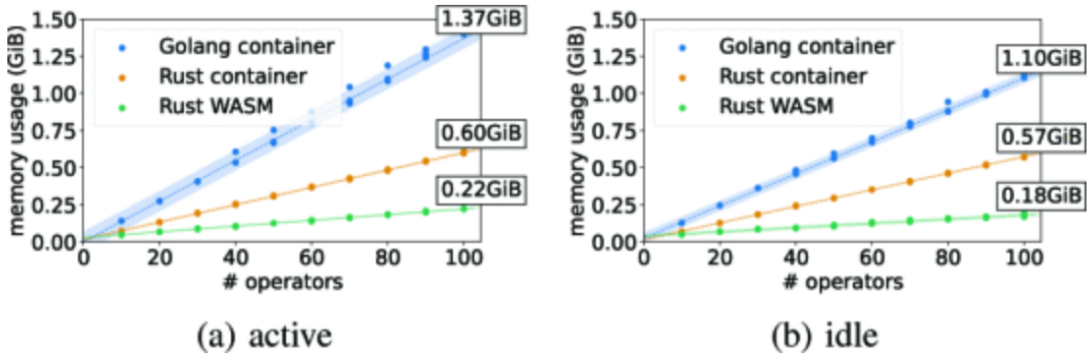

Figure 8.1: Different Kubernetes operators memory usage. Comparing Rust to WASM vs Rust container vs Golang container. © 2022 IEEE

Shillaker and Pietzuch (2020) introduced a novel serverless runtime framework using an abstraction called Faaslets, which employ Wasm for software-fault isolation (SFI). Their study showed that Faaslets in the Faasm runtime significantly enhance performance and efficiency over traditional container-based platforms. Notably, they achieved 2x speed-up in machine learning model training and doubled the throughput for interference tasks, while reducing memory usage by 90%, compared to traditional containers with Knative. See fig. 8.2 below for their findings while comparing their Faasm to Knative.

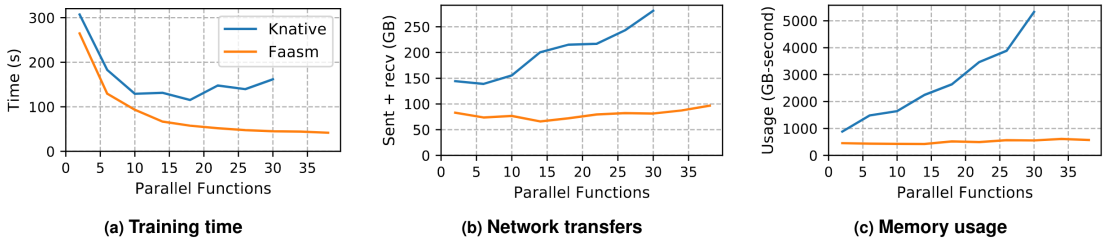

Figure 8.2: Different Kubernetes operators memory usage. Comparing Rust to WASM vs Rust container vs Golang container. © 2022 IEEE

Furthermore, Shillaker and Pietzuch (2020) documented the details of the cold starts they observed during their experiments. In their study they found that cold starts for operators based on Wasm were 538 times faster than the traditional container-based operators and occupied between 6.5 to 25 times less memory while in operation. See Table 8.1 below for their findings.

Table 8.1: The table is here

	Docker	Faaslets	vs.Docker
Initialization	2.8 s	5.2 ms	538×
CPU cycles	251M	1.4K	179K×
PSS memory	1.3 MB	200 KB	6.5×
RSS memory	5.0 MB	200 KB	25×
Capacity	~8 K	~70 K	8×

Chapter 9

Conclusion

Chapter 10

Future work

Acronyms

IaaS Infrastructure-as-a-Service

FaaS Functions-as-a-Service

Wasm WebAssembly

WASI WebAssembly System Interface

VMs virtual machines

VM virtual machine

VM Java Virtual Machine

NIST National Institute of Standards and Technology

AWS Amazon Web Services

GCP Google Cloud Platform

ACI Azure Container Instances

SBC Single board computer

AOT ahead-of-time

MQTT Message Queuing Telemetry Transport

Appendices

Bibliography

Adzic, G., & Chatley, R. (2017). Serverless computing: Economic and architectural impact. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 884–889. https://doi.org/10.1145/3106237.3117767

Aeotec. (n.d.). *Aeotec Smart Switch 6 user guide* [https://aeotec.freshdesk.com/helpdesk/attachments/6102433596]. Aeotec Help Desk. Retrieved April 29, 2024, from https://aeotec.freshdesk.com/support/solutions/articles/6000056905-smart-switch-6-user-guide-

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., & Hilt, V. (2018). SAND: Towards High-Performance Serverless Computing.

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015, Winter). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Commun. Surv. Tutorials*, *17*(4), 2347–2376. https://doi.org/10.1109/COMST.2015.2444095

Amazon. (2019, September 25). *The Climate Pledge*. EU About Amazon. Retrieved April 14, 2024, from https://www.aboutamazon.eu/news/sustainability/the-climate-pledge

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A View of Cloud Computing. *Commun. ACM*, *53*, 50–58. https://doi.org/10.1145/1721654.1721672

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., & Suter, P. (2017, June 10). *Serverless Computing: Current Trends and Open Problems* (1). arXiv: 1706.03178 [cs]. https://doi.org/10.48550/arXiv.1706.03178

Bao, W., Hong, C., Sudheer Chunduri, Chunduri, S., Krishnamoorthy, S., Sriram Krishnamoorthy, Sriram Krishnamoorthy, Pouchet, L.-N., Rastello, F., & Sadayappan, P. (2016). Static and Dynamic Frequency Scaling on Multicore CPUs. *ACM Transactions on Architecture and Code Optimization*, *13*(4),

51. https://doi.org/10.1145/3011017
MAG ID: 2567319156 S2ID: 7ee529c7a72f7f228ba1e60011d5e1d5078730d6.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. (2003). Xen and the Art of Virtualization.

Barr, J. (2006, August 25). *Amazon EC2 Beta | AWS News Blog*. Retrieved April 1, 2024, from https://aws.amazon.com/blogs/aws/amazon_ec2_beta/

Bekaroo, G., & Santokhee, A. (2016). Power consumption of the Raspberry Pi: A comparative analysis, 361–366. https://doi.org/10.1109/EmergiTech.2016.7737367

Beloglazov, A., Abawajy, J., & Buyya, R. (2012). Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems*, *28*(5), 755–768. https://doi.org/10.1016/j.future.2011.04.017

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, *1*(3), 81–84. https://doi.org/10.1109/MCC.2014.51

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, *14*(1), 70–93. https://doi.org/10.1145/2898442.2898444

Butcher, M., & Dodds, E. (2024, January 10). *How WebAssembly is Enabling the Third Wave of Cloud Compute with Matt Butcher of Fermyon Technologies* (No. 172). Retrieved April 1, 2024, from https://datastackshow.com/podcast/how-webassembly-is-enabling-the-third-wave-of-cloud-compute-with-matt-butcher-of-fermyon-technologies/

Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). The rise of serverless computing. *Commun. ACM*, *62*(12), 44–54. https://doi.org/10.1145/3368454

Chiueh, S., & Nanda, T.-c. (2005). A Survey on Virtualization Technologies.

Crisman, P. A. (1963). *Computer Time-Sharing System*. MIT. Retrieved March 29, 2024, from https://people.csail.mit.edu/saltzer/Multics/CTSS-Documents/CTSS_ProgrammersGuide_1963.pdf

Cuadrado-Cordero, I., Orgerie, A.-C., & Menaud, J.-M. (2018). Comparative Experimental Analysis of the Quality-of-Service and Energy-Efficiency of VMs and Containers' Consolidation for Cloud Applications. *International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2017)*, 1–6. https://doi.org/10.23919/SOFTCOM.2017.8115516

Durieux, T. (2024, March 12). *Empirical Study of the Docker Smells Impact on the Image Size* [Comment: Accepted at ICSE'24. arXiv admin note: text overlap

with arXiv:2302.01707]. arXiv: 2312.13888 [cs]. https://doi.org/10.1145/3597503.3639143

Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C. L., & Iosup, A. (2021, January 28). *A Review of Serverless Use Cases and their Characteristics* [Comment: 47 pages, 29 figures, SPEC RG technical report]. arXiv: 2008.11110 [cs]. Retrieved April 15, 2024, from http://arxiv.org/abs/2008.11110

Etro, F. (2009). The Economic Impact of Cloud Computing on Business Creation, Employment and Output in Europe. An application of the Endogenous Market Structures Approach to a GPT innovation. *Review of Business and Economics*, *54*, 179–208.

Favaloro, G., & O'Sullivan, S. (1996, November 14). *Internet Solutions Division Strategy for Cloud Computing*. Retrieved April 1, 2024, from https://s3.amazonaws.com/files.technologyreview.com/p/pub/legacy/compaq_cst_1996_0.pdf

Freitag, C., Berners-Lee, M., Widdicks, K., Knowles, B., Blair, G. S., & Friday, A. (2021). The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns*, *2*(9), 100340. https://doi.org/10.1016/j.patter.2021.100340

GmbH, G. S. (2023). Expert Power Control 1104 / 1105 Manual Guide. *Gude Systems*. https://files.gude-systems.com/pdf/gude/manual-epc1105-series_v1.5.1.pdf

Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., & Uhlig, S. (2023, October 12). *Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions* [Comment: 34 Pages, 16 Figures]. arXiv: 2310.08437 [cs]. Retrieved April 15, 2024, from http://arxiv.org/abs/2310.08437

Google. (n.d.). *Tracking Our Carbon-Free Energy Progress*. Google Sustainability. Retrieved April 14, 2024, from https://sustainability.google/progress/energy/

Google 2023 Environmental Report. (2023).

Gottlieb, N. (2018, June 20). *On serverless, data lock-in and vendor choice*. Retrieved April 15, 2024, from https://serverless.com/blog/data-lockin-vendor-choice-portability

Gunnerød, J. L. (2022). Statnett - Kortsiktig MarkedsanalyseNovember 2022.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). Bringing the web up to speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation, 185–200. https://doi.org/10.1145/3062341.3062363

Kaplan, J. M., Forrest, W., & Kindler, N. (2008). Revolutionizing Data Center Energy Efficiency. *McKinsey & Company*. Retrieved April 22, 2024, from https://docplayer.net/5045147-Revolutionizing-data-center-energy-efficiency.html

Kjorveziroski, V., Filiposka, S., & Mishev, A. (2022). Evaluating WebAssembly for Orchestrated Deployment of Serverless Functions. *2022 30th Telecommunications Forum (TELFOR)*, 1–4. https://doi.org/10.1109/TELFOR56187.2022.9983733

Kubernetes. (n.d.). *Kubernetes*. Retrieved April 16, 2024, from https://kubernetes.io/

Leonard, J. (2024, March 26). *WebAssembly heralds 'third wave of cloud computing'*. Retrieved April 19, 2024, from https://www.computing.co.uk/news/4189373/webassembly-heralds-wave-cloud-computing

McDonald, P. (2008, April 7). *Introducing Google App Engine + our new blog*. Google App Engine Blog. Retrieved April 8, 2024, from https://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html

McGrath, G., & Brenner, P. R. (2017). Serverless Computing: Design, Implementation, and Performance. *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 405–410. https://doi.org/10.1109/ICDCSW.2017.36

Mell, P., & Grance, T. (2011, September 1). The NIST Definition of Cloud Computing.

Merkel, D. (2014, May 9). *Docker: Lightweight Linux Containers for Consistent Development and Deployment | Linux Journal*. Retrieved April 16, 2024, from https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

Mytton, D. (2020). Hiding greenhouse gas emissions in the cloud. *Nat. Clim. Chang.*, *10*(8), 701–701. https://doi.org/10.1038/s41558-020-0837-6

Ni, J., & Bai, X. (2017). A review of air conditioning energy performance in data centers. *Renewable and Sustainable Energy Reviews*, *67*, 625–640. https://doi.org/10.1016/j.rser.2016.09.050

Norwegian-Energy. (2023, November 8). *Electricity production*. Norwegian Energy. Retrieved April 14, 2024, from https://energifaktanorge.no/en/norsk-energiforsyning/kraftproduksjon/

Qian, Z. (2022, January 7). *Energy footprinting of WordPress websites on the user side* (Master Thesis). University of Bristol.

Rivrud, K. (2024, February 7). *Investeringen av et Google-senter i Skien er gigantisk – det blir også strømforbruket*. NRK. Retrieved April 12, 2024, from https://www.nrk.no/vestfoldogtelemark/investeringen-av-et-google-senter-i-skien-er-gigantisk-_-det-blir-ogsa-stromforbruket-1.16753588

Roberts, M. (2018, May 22). *Serverless Architectures*. martinfowler.com. Retrieved April 15, 2024, from https://martinfowler.com/articles/serverless.html

Sebrechts, M., Ramlot, T., Borny, S., Goethals, T., Volckaert, B., & De Turck, F. (2022). Adapting Kubernetes controllers to the edge: On-demand control planes using Wasm and WASI. *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, 195–202. https://doi.org/10.1109/CloudNet55617.2022.9978884

Sergeev, A., Rezedinova, E., & Khakhina, A. (2022). Docker Container Performance Comparison on Windows and Linux Operating Systems, 1–4. https://doi.org/10.1109/CIEES55704.2022.9990683

Sewak, M., & Singh, S. (2018). Winning in the Era of Serverless Computing and Function as a Service. *2018 3rd International Conference for Convergence in Technology (I2CT)*, 1–5. https://doi.org/10.1109/I2CT.2018.8529465

Shehabi, A., Smith, S., Sartor, D., Brown, R., Herrlin, M., Koomey, J., Masanet, E., Horner, N., Azevedo, I., & Lintner, W. (2016, June 1). *United States Data Center Energy Usage Report* (LBNL–1005775, 1372902). https://doi.org/10.2172/1372902

Shillaker, S., & Pietzuch, P. (2020). Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing, 419–433. Retrieved April 22, 2024, from https://www.usenix.org/conference/atc20/presentation/shillaker

Shirinbab, S., Lundberg, L., & Casalicchio, E. (2020). Performance evaluation of containers and virtual machines when running Cassandra workload concurrently. *Concurrency and Computation: Practice and Experience*, *32*. https://doi.org/10.1002/cpe.5693

Smith, B. (2020, January 16). *Microsoft will be carbon negative by 2030*. The Official Microsoft Blog. Retrieved April 14, 2024, from https://blogs.microsoft.com/blog/2020/01/16/microsoft-will-be-carbon-negative-by-2030/

Solomon Hykes [@solomonstre]. (2019, March 27). *If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!* Twitter. Retrieved April 15, 2024, from https://twitter.com/solomonstre/status/1111004913222324225

Tong, W., Gao, J., Jin, X., & Li, Z. (2015). Study on Ethernet Communication Strategy for Electric Energy Data Acquisition System. *2015 Fifth International*

Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC), 1240–1244. https://doi.org/10.1109/IMCCC.2015.266

Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uta, A., & Iosup, A. (2018). Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Comput.*, *22*(5), 8–17. https://doi.org/10.1109/MIC.2018.053681358

Wang, L., Li, M., Zhang, Y., Ristenpart, T., & Swift, M. (2018). Peeking Behind the Curtains of Serverless Platforms.

WASI.dev. (n.d.). Retrieved April 22, 2024, from https://wasi.dev/

WASI/preview2/README.md at main · WebAssembly/WASI. (n.d.). GitHub. Retrieved April 24, 2024, from https://github.com/WebAssembly/WASI/blob/main/preview2/README.md

WebAssembly.org. (n.d.). *FAQ - WebAssembly*. Retrieved April 16, 2024, from https://webassembly.org/docs/faq/

Zhang, Y., Liu, M., Wang, H., Ma, Y., Huang, G., & Liu, X. (2024, April 19). *Research on WebAssembly Runtimes: A Survey*. arXiv: 2404.12621. Retrieved April 23, 2024, from http://arxiv.org/abs/2404.12621