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 why Fermyon decided to bet on WebAssembly for the next 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 WebAssembly (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 through this adventure by introducing the concept of the "Three waves of cloud computing", coined by the WebAssembly community (Butcher & Dodds, 2024). Where the two first waves of cloud compute represent the shit 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 essential to understand the landscape that preceded cloud computing. Prior to the cloud era, companies were required to building and mantaining 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)

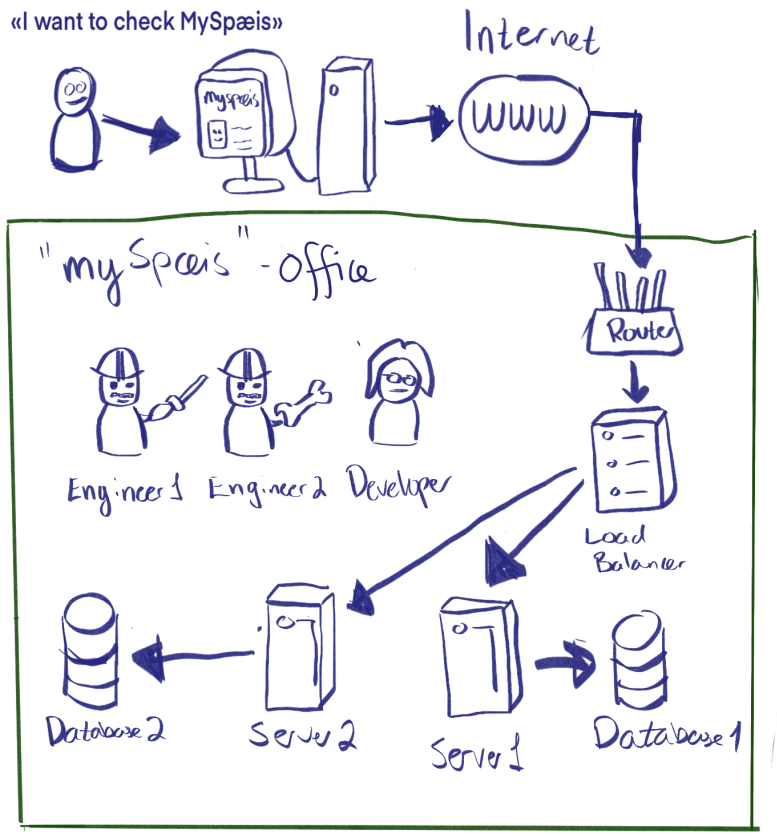

Figure 2.1: Example of a company that host their own infrastructure.

This setup mandates a significant upfront costs involved in setting up and maintaining such an infrastructure, which puts a considerable financial strain on organizations, and kept smaller companies that were unable to invest in this, at an disadvantage.

As a response to this, 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)

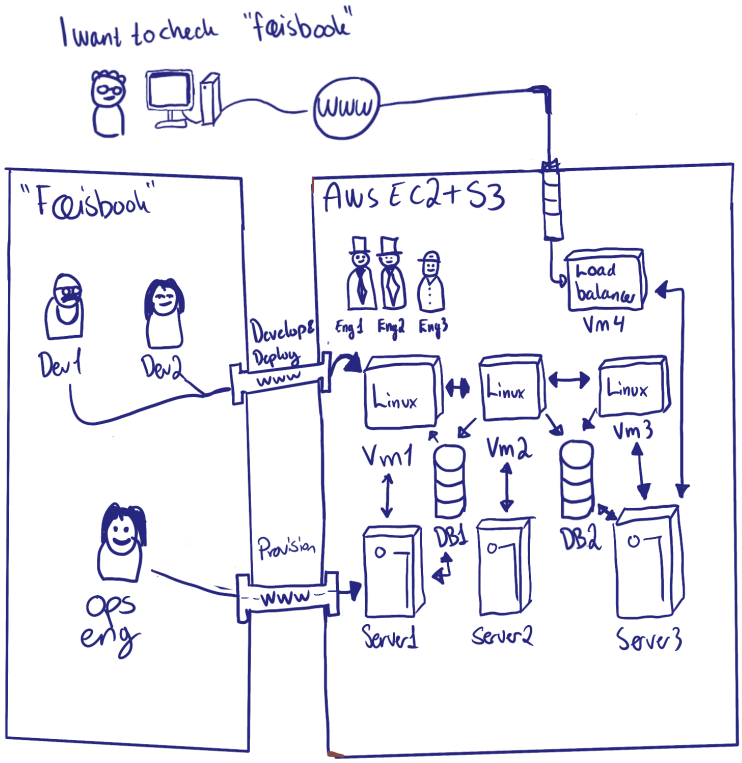

Figure 2.2: Example of "Fæisbook" building their services on EC2 and using S3 for storage.

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: Containerization

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)

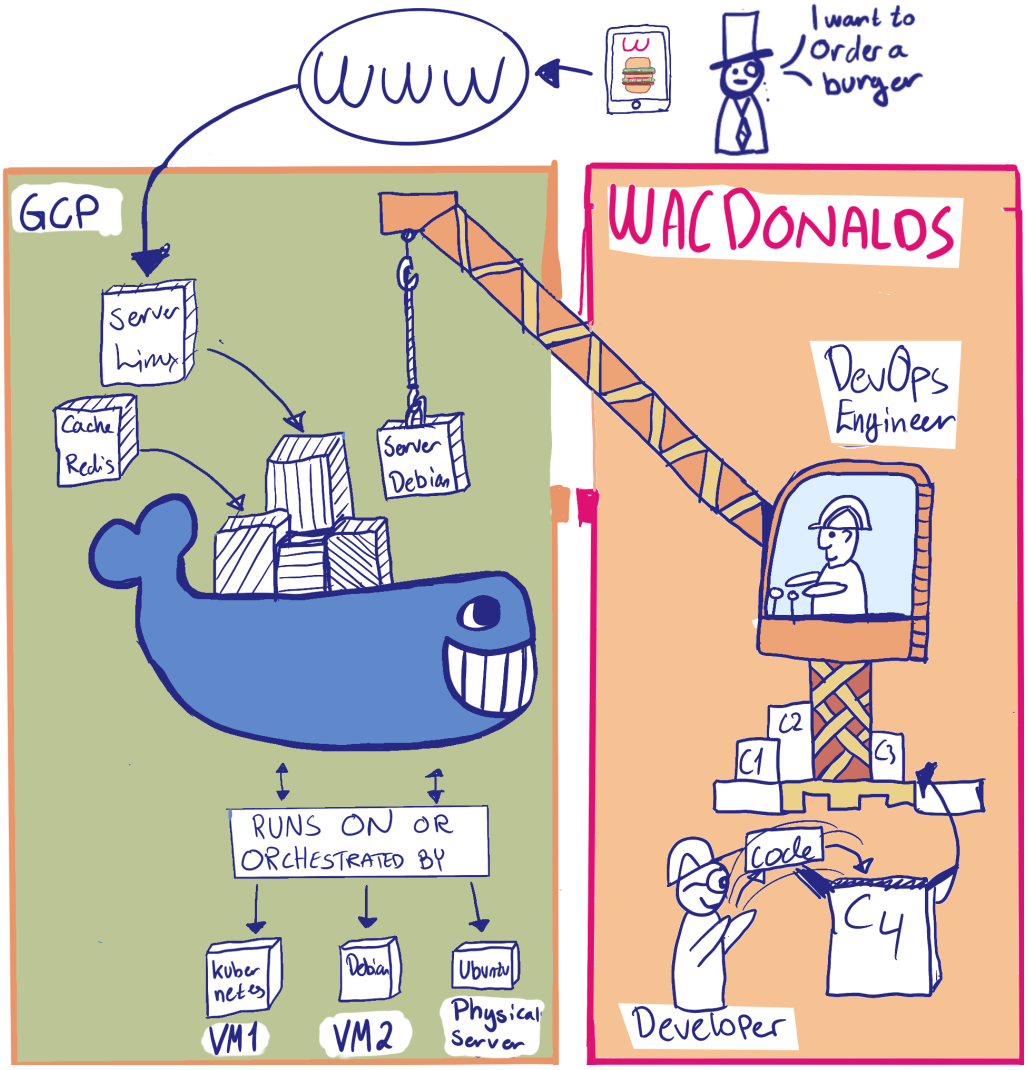

Figure 2.3: DevOps engineer deploying services as containers on GCP.

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.

[1]https://kubernetes.io

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

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, 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

One of the downsides to contrast the benefits of cloud computing, is the ever increasing demand for energy consumption required for running these servers in

giant 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.

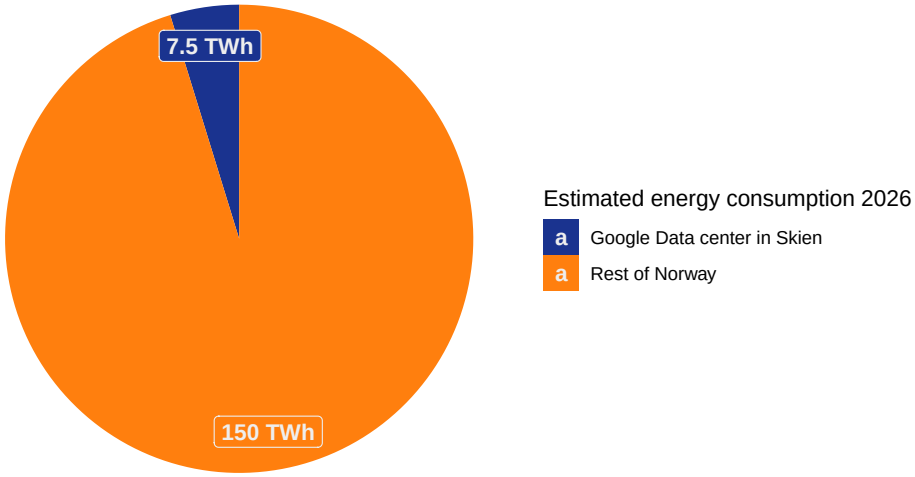

Figure 3.1: Projected energy consumption in 2026: Skien data center and Norway.

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)

[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 allows 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.

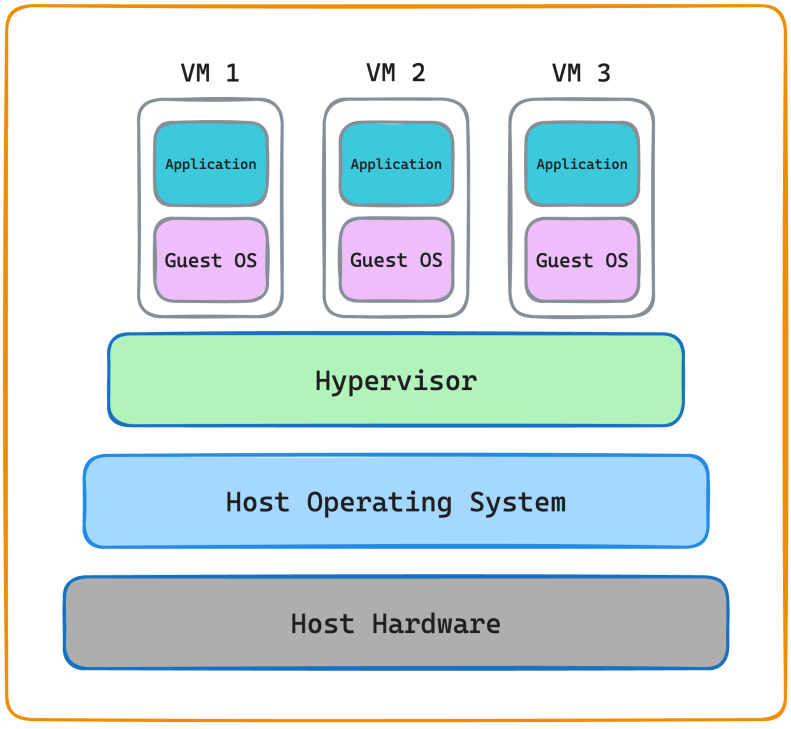

Figure 3.2: Virtual machines running on top of Hypervisor on a computer.

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, along with its dependencies, libraries, and configuration files, into a single, self-contained unit. This allows applications to be deployed consistently across different environments, ensuring predictable behavior and reducing compatibility issues (Sergeev et al., 2022)

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.

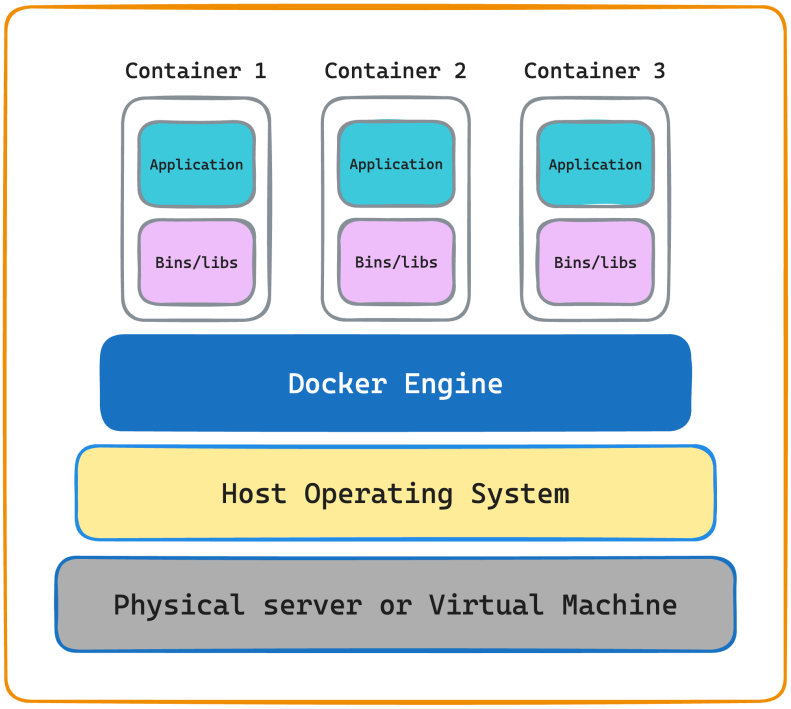

Figure 3.3: Containers running on the Docker Engine

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

[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

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, 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 designed it to be a subset of JavaScript, designed to allow web applications written in other languages than JavaScript, such as C or C++, to run in the browser. The intention of asm.js is to allow for web applications to run at performance closer to native code than applications written in standard JavaScript can achieve. 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.

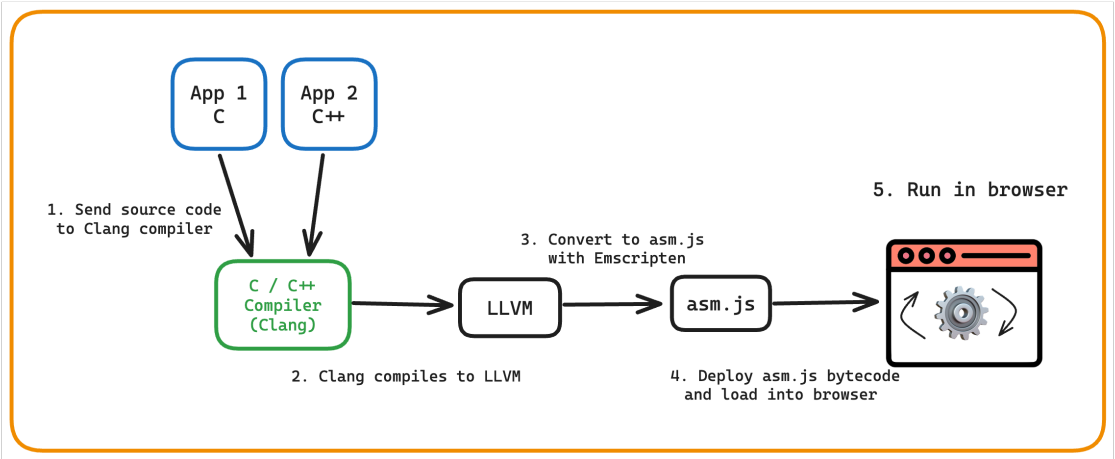

Figure 3.4: Source code in C/C++ compiled to asm.js and run in browser

While asm.js was a great leap forward, being a subset of JavaScript limited the scope of what it could become, 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 and launch the first public version in 2017. WebAssembly, which is a low-level code format designed to serve as a compilation target for high-level programming languages. 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).

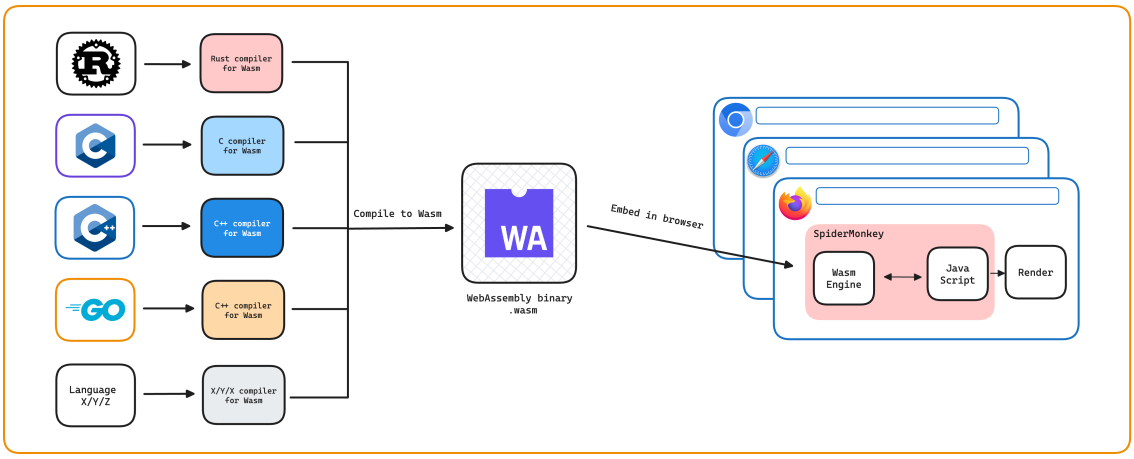

Figure 3.5: Source code compiled to WebAssembly and embedded in browser

Finish Wasm section

3.6.3 WebAssembly System Interface

Write WASI section

3.7 Energy monitoring

Write about MQTT and Zwave/Oh my Gude

WebAssembly, originally designed for running demanding computations in web browsers, present a promising technology that could help reduce the energy consumption of cloud services. It offers an interesting option for packaging functions with its compact binary format and fast execution time. This has the potiential to significantly reduce startup latency and resource overhead associated with traditional serverless platforms. This increased efficiency could lead to a direct decrease in energy consumption for cloud services, which in turn could motivate the industry to adopt alternative technology that enable a more sustainable cloud.

Rewrite the rest of this section, it's a bit of a mess right now

The WebAssembly team defines WebAssembly as such:

WebAssembly definition

> WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.
>
> webassembly.org

In other words, WebAssembly is a low-level code format designed to serve as a compilation target for high-level programming languages. It's a binary format that gets executed by a stack-based virtual machine, similar to how Java bytecode runs on the Java Virtual Machine (JVM). It was originally designed for running in a browser environment, and every major browser has implemented a way for running it.

WebAssembly have promising properties that makes it interesting to investigate if it can find a home outside the browser environment it was designed for:

Efficiency and Speed: WebAssembly was designed to be fast, enabling near-native performance. Its binary format is compact and designed for quick decoding, contributing to quicker startup times, important aspects of cloud native applications.

Safety and Security: WebAssembly is designed to run safely in a secure sandbox. Each WebAssembly module executes within a confined environment without direct access to the host system's resources. This isolation of processes is inherent in WebAssembly's design, promoting secure practices.

Portability: WebAssembly's platform-agnostic design makes it highly portable. It can run across a variety of different system architectures. For cloud native applications, this means WebAssembly modules, once compiled, can run anywhere - from the edge to the server - on any environment.

Language Support: A large amount of programming languages can already target WebAssembly. This means developers are not restricted to a particular language when developing applications intended to be deployed as WebAssembly modules. This provides greater flexibility to leverage the most suitable languages for particular tasks.

In contrast, traditional methods such as deployment with containers or VMs can be resource-intensive, slower to boot up, less secure due to a larger surface attack area, and less efficient. Given these, WebAssembly, with its efficiency, security, and portability, can potentially offer an attractive alternative deployment method for building and running cloud native applications, like the "Academemes" service we will explore in this essay.

WebAssembly (WASM) and WebAssembly System Interface (WASI) present promising choices to traditional ways of deploying and hosting Function as a Service (FaaS) platforms, offering several notable advantages, in terms of startup times and energy efficiency.

Reduced Startup Times: One of the greatest strengths of Wasm is its compact binary format designed for quick decoding and efficient execution. It offers near-native performance, which results in significantly reduced startup times compared to container-based or VM-based solutions. In a FaaS context, where functions need to spin up rapidly in response to events, this attribute is particularly advantageous. This not only contributes to the overall performance but also improves the user experience, as the latency associated with function initialization is minimized.

Improved Energy Efficiency: Wasm's efficiency extends to energy use as well. Thanks to its optimized execution, Wasm can accomplish the same tasks as traditional cloud applications but with less computational effort. The CPU doesn't need to work as hard, which results in less energy consumed. With data centers being responsible for a significant portion of global energy consumption and carbon emissions, adopting Wasm could lead to substantial energy savings and environmental benefits.

Scalability: Wasm's small footprint and fast startup times make it an excellent fit for highly scalable cloud applications. Its efficiency means it can handle many more requests within the same hardware resources, hence reducing the need for additional servers and thus reducing the energy footprint further.

Portability and Flexibility: WASI extends the portability of Wasm outside the browser environment, making it possible to run Wasm modules securely on any WASI-compatible runtime. This means that FaaS platforms can run these modules on any hardware, operating system, or cloud provider that supports WASI. This

portability ensures flexibility and mitigates the risk of vendor lock-in.

While runtime efficiency is an important aspect and typically a strength of Wasm, it might not be the primary focus of this thesis. That being said, it is worth mentioning that the efficient execution of Wasm modules does contribute to the overall operational efficiency and energy savings of Wasm-based FaaS platforms.

In summary, introducing WASM+WASI as a component for deploying and hosting FaaS platforms can offer significant benefits. Focusing on energy efficiency and reduced startup times, this approach could pave the way for more sustainable, efficient, and responsive cloud services. In the context of our “Academemes” service, this could lead to a scalable, performant, and environmentally friendly platform.

Part II

Project

Chapter 4

Methodology

This chapter outlines the research methodology employed to investigate the problem statements posed in Section 1.3, specifically whether Wasm is a more energy-efficient target for developing and deploying cloud native applications. A FaaS prototype, named 'Nebula', will be developed as part of an experimental reserach method. This prototype will be used to conduct experiments on startup latencies (cold starts), runtime efficiency, and energy consumption of function invocations in two different deployment environments: Wasm modules and Docker containers. The chapter details the research design, experimental setup, data collection methods, and analysis techniques, with a focus on ensuring the validity and reliability of the findings.

4.1 Research design

The objective of this thesis is to compare the performance and energy efficiency of functions written in Rust when compiled to and deployed as Wasm and WASI modules versus the same functions compiled to a Rust binary within slim Docker containers. The research design will facilitate an evaluation of these two environments for running functions in a serverless model.

4.2 Experimental framework

The main focus for this thesis is on building a prototype and perform experiments on it, to measure the capabilities of Wasm, augmented by WASI as a model for deploying and running cloud applications. An experimental approach was chosen to test the problem statements posed in Section 1.3, as it allows for controlled manipulation of variables, empirical measurements of outcomes, and direct comparison between two deployment environments (Wasm+WASI and Docker). This

section outlines the different methods employed as part of an experimental research method to investigate the research questions.

4.2.1 Prototyping

The prototype FaaS system, named 'Nebula', will be developed to serve as a controlled environment for invoking functions as either Wasm modules or Docker containers. It needs

To perform the desired experiments, a prototype will be developed and deployed to two types of hardware. It will need to support invoking functions compiled to Wasm and functions packaged as Docker images.

4.2.1.1 Hardware specifications

Experiments will be performed on two types of hardware:

1. Raspberry Pi: A small **SBCs!** (**SBCs!**) 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, to measure performance relative to what a standard cloud native application would perform.

4.2.1.2 Software specification

For developing Nebula prototype itself, the following software setup will be required:

- A programming language to write the prototype in.
- An interface to deploy and invoke functions deployed onto the prototype.
-

The software setup for developing functions will include:

- Rust programming language: Chosen for implementing functions due to its efficiency and compatibility when building with targeting both Wasm and traditional binaries.

- Wasm and WASI: The functions will be compiled as Wasm binaries with the `wasm32-wasi` target when building with Rust.
- Docker: Functions will also be packaged as Docker images for comparison purposes.

4.2.2 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.

4.2.3 Empirical measurements

Empirical measurement techniques will be utilized to quantify the energy consumption associated with function invocations on each deployment environment. The prototype and function binaries and images will be deployed to the Raspberry Pi, which will be connected to a power supply that can report energy readings.

4.2.4 Controlled experimentation

Controlled experimentation will be a crucial aspect of this research, as it will allow for isolating and studying 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 ensure the validity and reliability of the results.

4.2.5 Comparative analysis

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.

4.2.6 Data collection and analysis

Throughout the experiments, various data points, including startup and execution times, and energy consumption readings, will be collected and stored. Statistical

techniques, such as mean, median, and standard deviation calculations, will be employed to analyze the collected data.

Chapter 5

Designing Nebula

Chapter 6

Prototype implementation

This chapter provides the details of the implementation of Nebula.

6.1 Requirements

Chapter 7

Design

Chapter 8

Implementation

This is the chapter on implementing Nebula.

8.1 Tech stack

Rust/Docker/Etc.

Part III

Results

Chapter 9

Evaulation

Chapter 10

Discussion

Chapter 11

Conclusion

Chapter 12

Future work

References

Chapter 13

Appendices

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

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

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

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

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

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

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

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.

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/

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/

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

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

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

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.

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