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 lower energy consumption, offering a viable pathway towards more sustainable cloud computing.

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

Furthermore I would 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. If I hadn’t met him at the WasmIO 2024 conference in Barcelona, my startup times would be 100 times slower.

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 (**freitag2021**).

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 (**mytton2020**). 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 **WASI!** (**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!** (**Wasm!**) with **WASI!** as an innovative choice for deploying functions to the cloud, through developing a prototype **FaaS!** (**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 its environmental impact. This chapter steps introduces the concept of the "Three waves of cloud computing", coined by the WebAssembly community (**butcherDodds2024**; **leonardWebAssemblyHeraldsThird2024**). 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 **??** for an example)

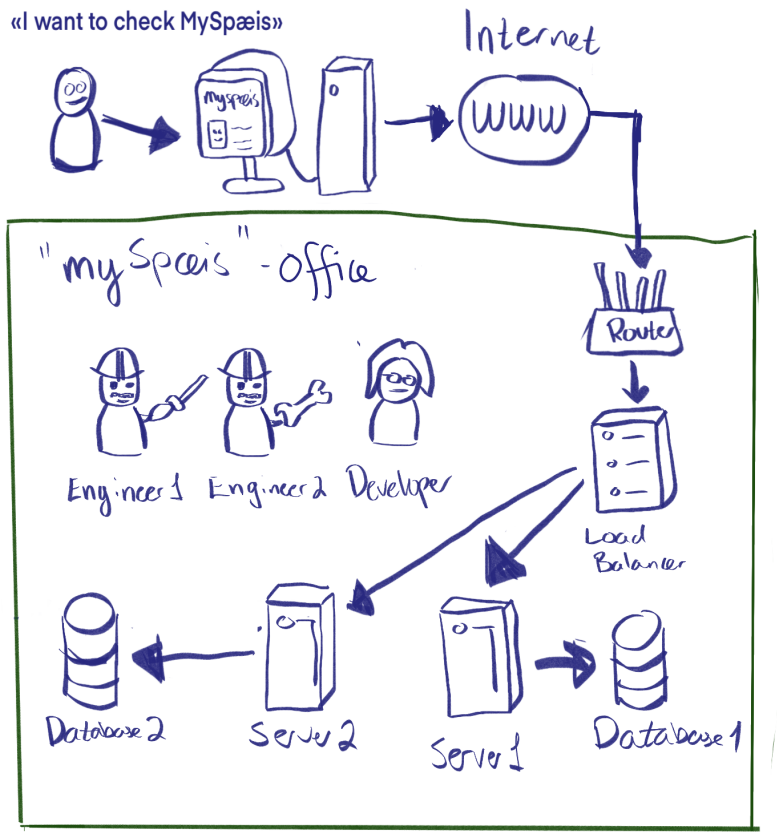

Figure 2.1: Example of a company that host their own infrastructure.

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 (**etroEconomicImpactCloud2009**). 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 (**armbrustViewCloudComputing2010**).

As a response to these challenges, some companies found a market for taking on the responsibility of managing infrastructure, and offer **IaaS!** (**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 (**nist-def**). 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 (**barrAmazonEC2Beta2006**). 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 **??** for an example of this kind of architecture)

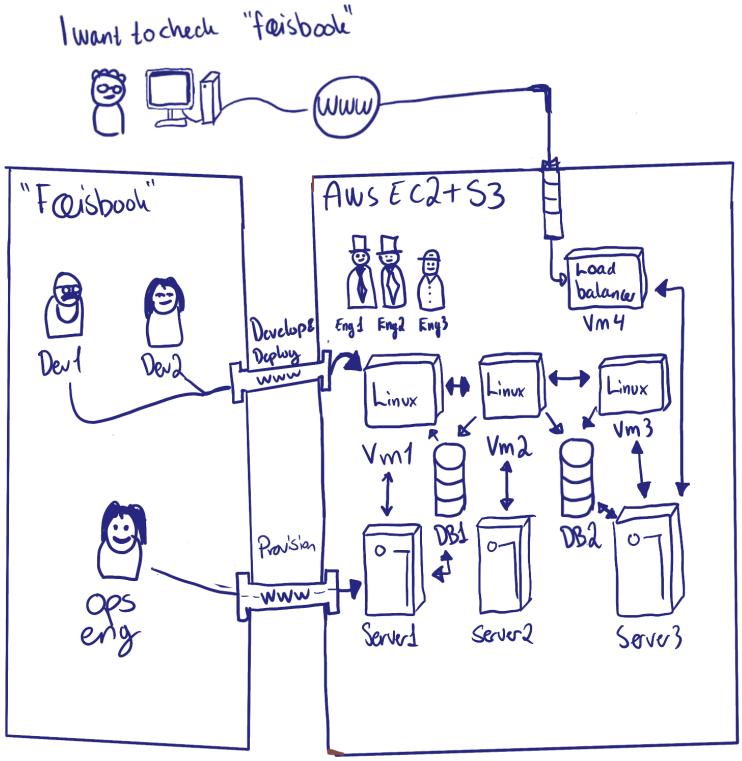

Figure 2.2: Example of "Fæisbook" building their services on EC2 and using S3 for storage.

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 (**bao2016**).

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 **??** for an example where a fictive company `WacDonalds` and their container workflow)

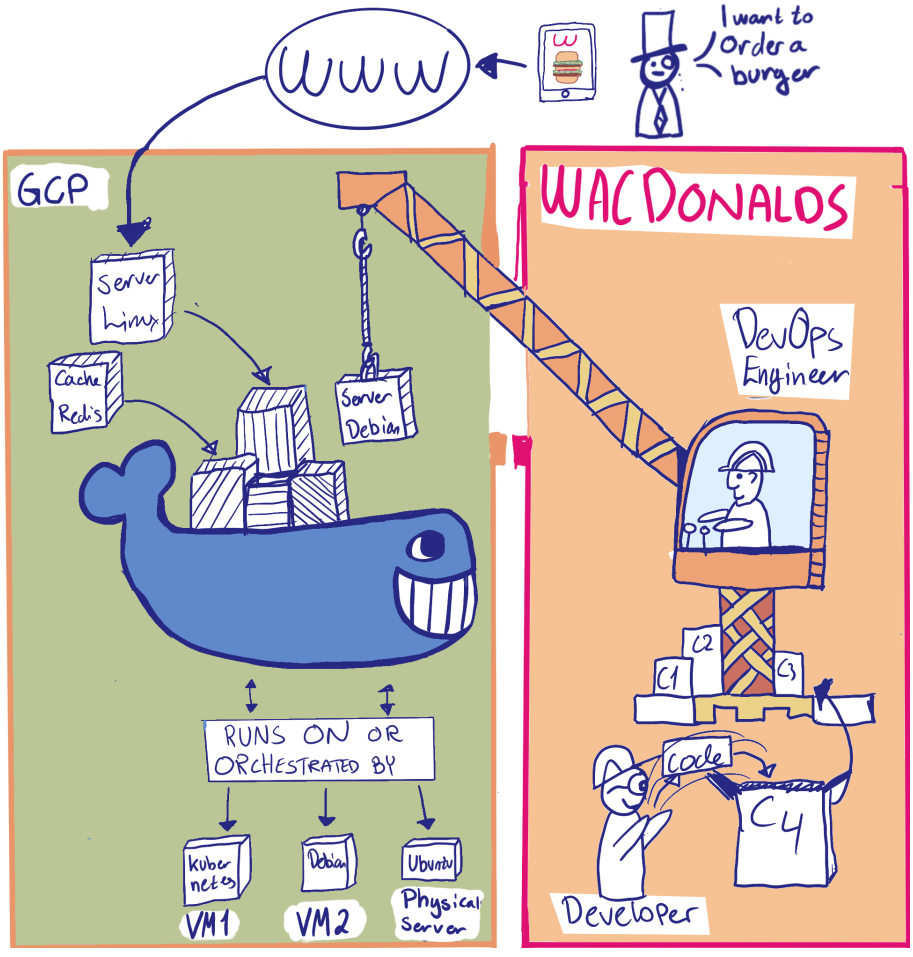

Figure 2.3: DevOps engineer deploying services as containers on **GCP!**.

[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 (**durieux2023**), 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 **??**, 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 **??**, 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 **VM!** (**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 **NIST!** (**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.

(**nist-def**)

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 (**crisman1963**). 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 todays sophisticated cloud services. The term “cloud computing” was first coined in the year 1996 by Compaq, (**favaloroInternetSolutionsDivision1996**), but it was not until Amazon launched its subsidiary **AWS!** (**AWS!**) in the 2006 that the adoption became wide spread.

The launch of AWS Amazon S3 cloud service in March 2006, followed by Elastic Compute Cloud (EC2) in August the same year (**barrAmazonEC2Beta2006**), 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 (**mcdonaldIntroducingGoogleApp2008**), 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 (**bernsteinContainersCloudLXC2014**).

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. (**baldiniServerlessComputingCurrent2017**)

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 **??**, the ICT industry accounts for an estimated 2.1% to 3.9% of global greenhouse emissions (**freitag2021**). 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 (**rivrudInvesteringenAvGooglesenter2024**). 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 (**Google2023Environmental2023**). 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 (**gunnerodStatnettAnalyse2022**), meaning that the data center in Skien could account for 5% of the energy consumption in the country. **??** below illustrates the amount of energy the new data center will consume compared to Norway.

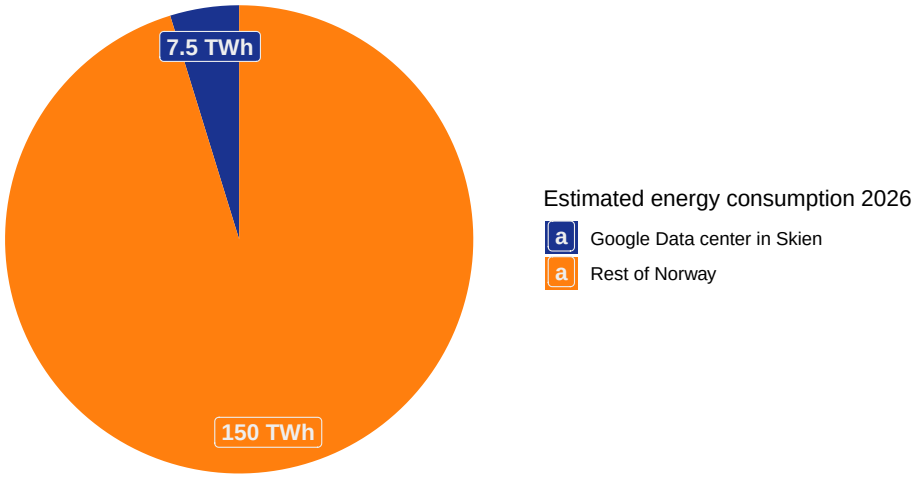

Figure 3.1: Projected energy consumption in 2026: Skien data center and Norway.

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 not 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 (**googleTrackingOurCarbonFree**). 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-energyElectricityProduction2023**).

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 (**smithMicrosoftWillBe2020**), 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 (**amazonClimatePledge2019**). 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 (**mytton2020**).

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 (**niReviewAirConditioning2017**).

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

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

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

management techniques, such as dynamic voltage and frequency scaling (DVFS), which adjusts the power consumption of servers based on workload demands (**beloglazovEnergyawareResourceAllocation2012**).

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. (**beloglazovEnergyawareResourceAllocation2012**)

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 (**chiuehSurveyVirtualizationTechnologies2005**). 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 **VMs!** (**VMs!**). **barhamXenArtVirtualization2003<empty citation>** 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.

?? below illustrates virtual machines running in such an environment.

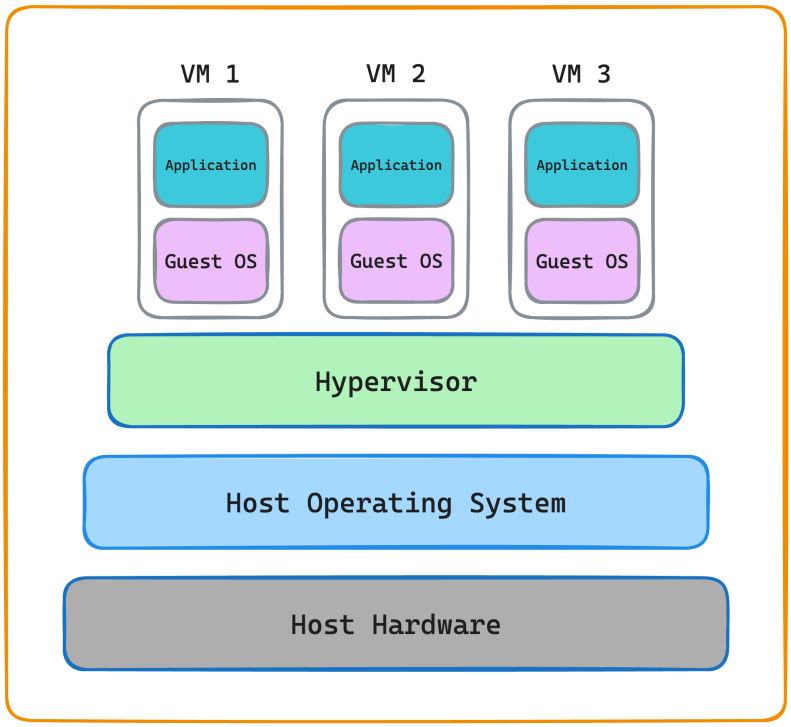

Figure 3.2: Virtual machines running on top of Hypervisor on a computer.

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 (**kaplanRevolutionizingDataCenter2008**).

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 (**merkelDockerLightweightLinux2014**). 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!** (**bernsteinContainersCloudLXC2014**).

merkelDockerLightweightLinux2014<empty citation> 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 (**sergeevDockerContainerPerformance2022**)

By enabling more granular and efficient use of system resources, containers con-

tribute to lower energy usage compared to virtual machines due to their lightweight nature and faster startup times (**shirinbabPerformanceEvaluationContainers2020**). 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-corderoComparativeExperimentalAnalysis2018**).

Docker is one of the most widely adopted container platforms, providing tools for building, shipping, and running applications in containers (**merkelDockerLightweightLinux2014**). Docker containers are based on open standards and can run on various operating systems and cloud platforms (**sergeevDockerContainerPerformance2022**). **??** below illustrates how containers run ontop of a Docker Engine on a virtual or physical machine.

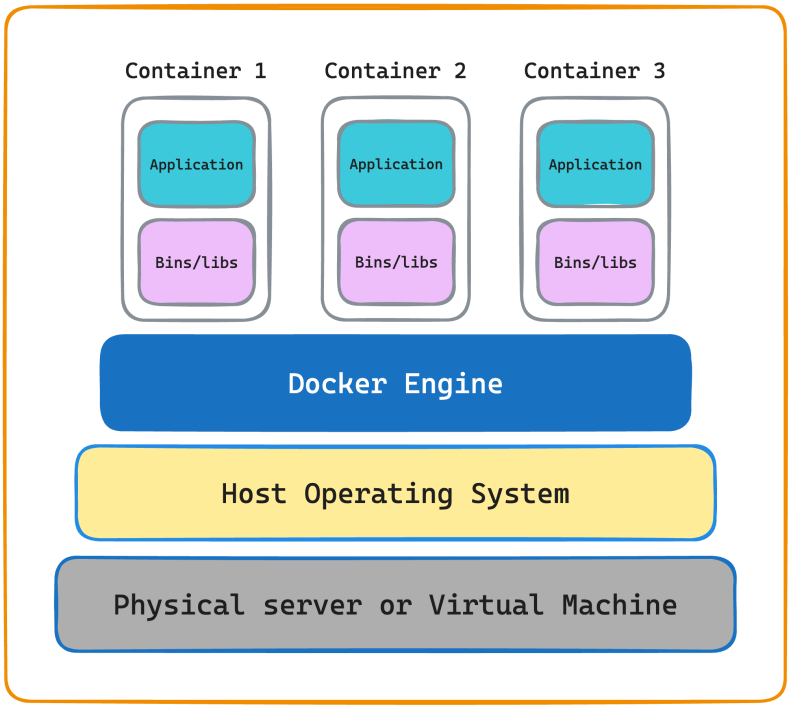

Figure 3.3: Containers running on the Docker Engine

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 (**burnsBorgOmegaKubernetes2016**).

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 (**burnsBorgOmegaKubernetes2016**).
2. Self-healing and monitoring: Orchestration tools can monitor the health of containers and automatically restart or reschedule them in case of failures (**kubernetesKubernetes**).
3. Service discovery and load balancing: Applications running in containers can be easily discovered and accessed by other services, enabling microservice architectures (**kubernetesKubernetes**).

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 **??** (**baldiniServerlessComputingCurrent2017**). 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 (**baldiniServerlessComputingCurrent2017**; **robertsServerlessArchitectures2018**). **castroRiseServerlessComputing2019<empty** 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.
>
> (**castroRiseServerlessComputing2019**)

According to **baldiniServerlessComputingCurrent2017<empty citation>**, 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 (**baldiniServerlessComputingCurrent2017**).
2. Faster time-to-market: With serverless, developers can quickly deploy and iterate on functions without the need for time consuming infrastructure setup (**adzicServerlessComputingEconomic2017**).
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 (**eismannReviewServerlessUse2021**).

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 (**gottliebServerlessDataLockin2018**).
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 (**golecColdStartLatency2023**).
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 (**akkusSANDHighPerformanceServerless2018**).

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 **ACI!** (**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 (**sewakWinningEraServerless2018**).

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

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 (**baldiniServerlessComputingCurrent2017**).

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 (**mcgrathServerlessComputingDesign2017**).

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 (**vaneykServerlessMorePaaS2018**). However, the use of containers in FaaS also introduce some challenges including, cold start latency and the overhead associated with container initialization and management (**wangPeekingCurtainsServerless2018**).

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 (**wangPeekingCurtainsServerless2018**). 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." (***hykesOne2019***)*

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 **kjorveziroskiEvaluatingWebAssemblyOrchestrated2022<empty citation>**.

[5]https://aws.amazon.com/lambda/
[6]https://cloud.google.com/functions
[7]https://azure.microsoft.com/en-us/products/functions

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 **??** below.

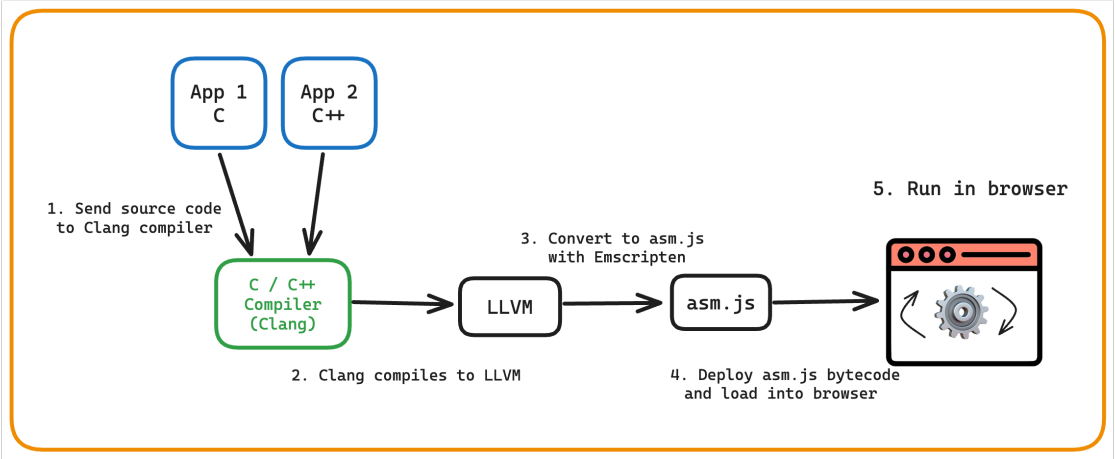

Figure 3.4: Source code in C/C++ compiled to asm.js and run in browser

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.orgFAQWebAssembly**).

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 (**haasBringingWebSpeed2017**). It is a binary format that gets executed by a stack-based virtual machine, comparable to how Java bytecode runs on the **JVM!** (**JVM!**) (**haasBringingWebSpeed2017**). In **??** below, a simplified flow for compiling web applications written in different languages can be compiled and run inside a web browser.

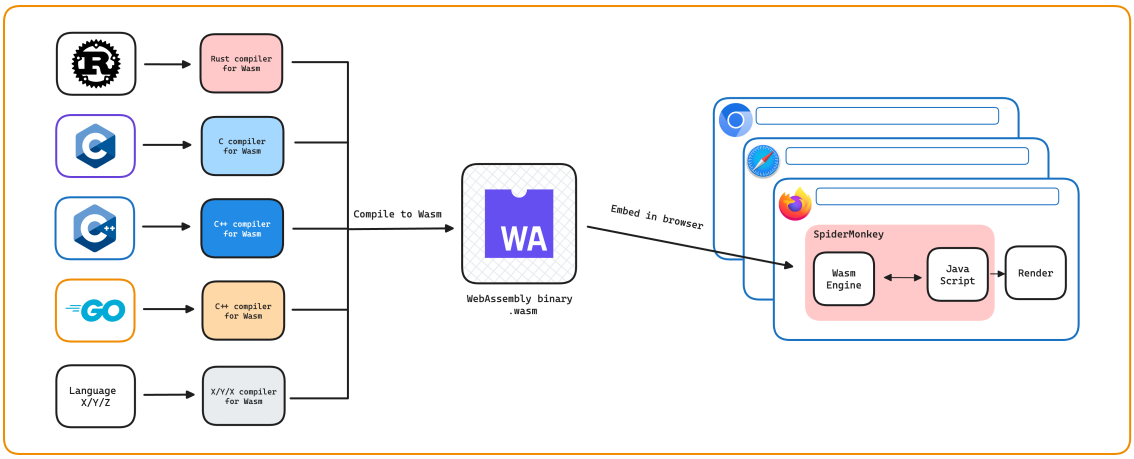

Figure 3.5: Source code compiled to WebAssembly and embedded in browser

According to **haasBringingWebSpeed2017<empty citation>**, some key features that the team behind **Wasm!** sought out to implement were:

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 (**haasBringingWebSpeed2017**).

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 (**WASIDev**).

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 (**WASIDev**). Its 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 (**WASIPreview2README**).

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.

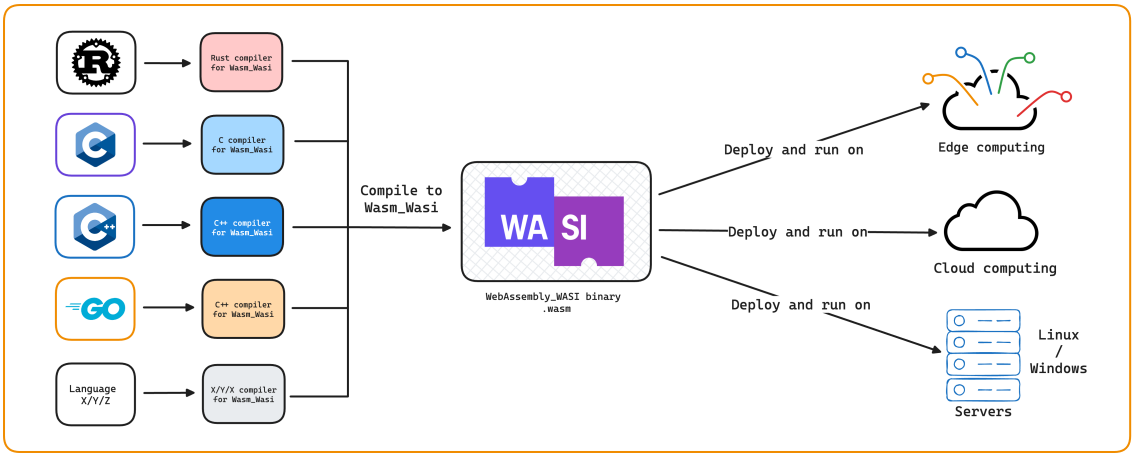

Figure 3.6: Source code compiled to wasm_wasi32 and deployed on platforms that support running the binaries.

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 (**zhang2024**).

zhang2024<empty citation> 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 **AOT!** (**AOT!**) compilation, which can further improve the performance of WebAssembly modules. **??** below illustrates how code written in a programming languages with support for compiling to **Wasm!** can run cross-platform.

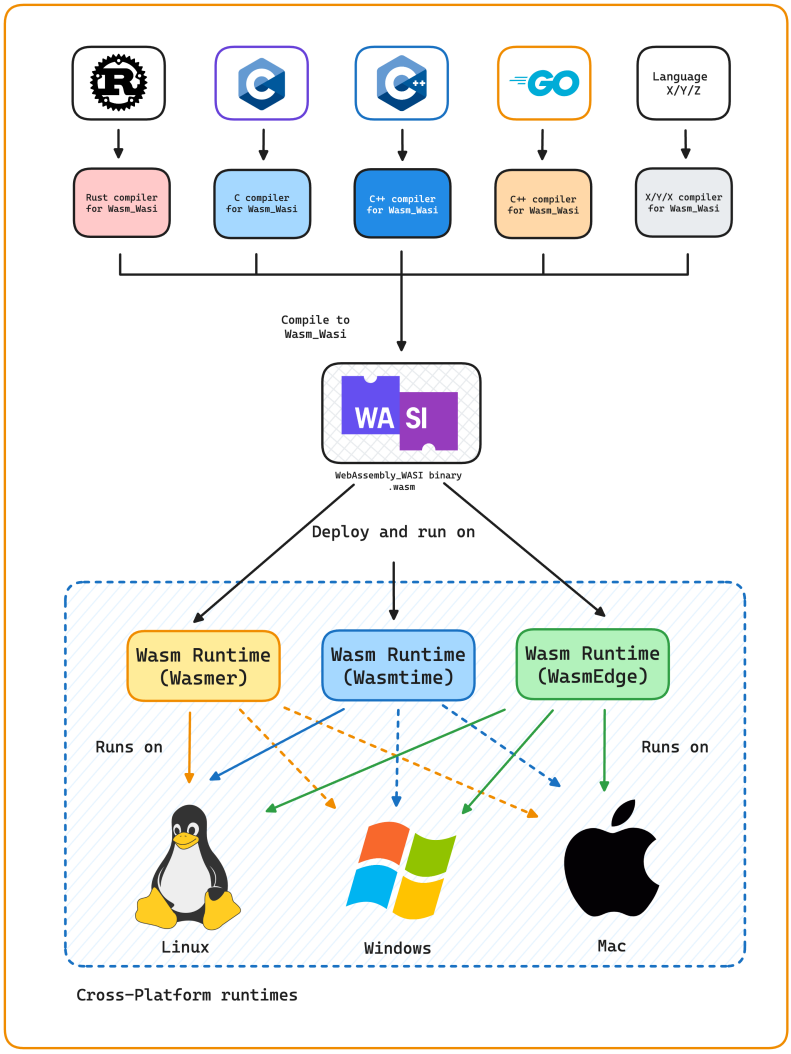

Figure 3.7: Source code compiled to wasm_wasi32 that can run anywhere a WebAssembly runtime can be installed.

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 (**shillakerFaasmLightweightIsolation2020a**; **sebrechtsAdaptingKubernetesControll**

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 (**shehabiUnitedStatesData2016**; **al-fuqahaInternetThingsSurvey2015**).

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-fuqahaInternetThingsSurvey2015**).

3.7.1 MQTT

The **MQTT!** (**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-fuqahaInternetThingsSurvey2015**). (See **??**).

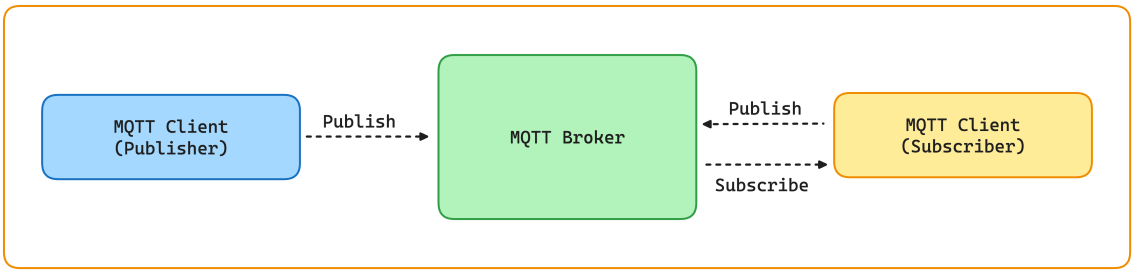

Figure 3.8: PubSub model of the MQTT protocol.

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-fuqahaInternetThingsSurvey2015**).

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

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

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 (**aeotecAeotecSmartSwitch**).

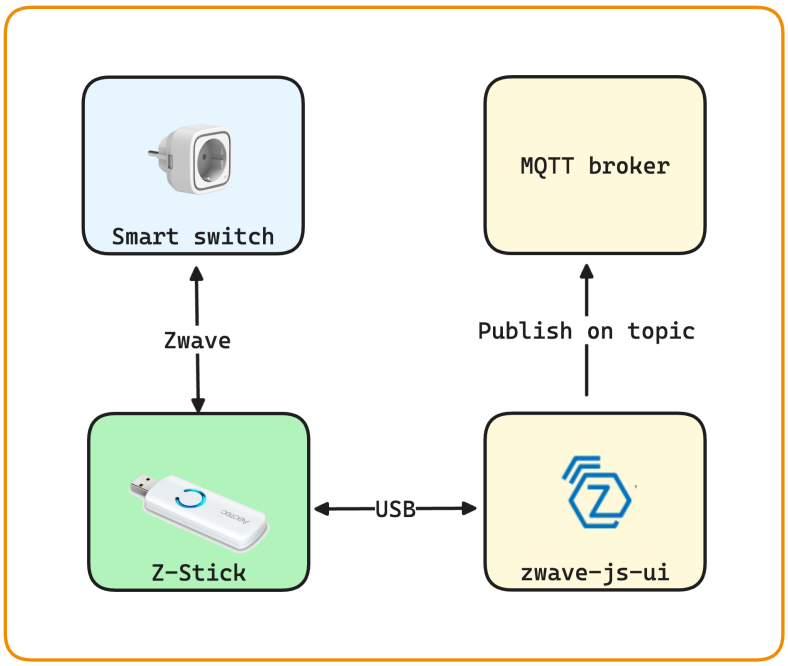

Figure 3.9: Smart Switch 6 communicating with zwave-js-ui through Aeotec Smart Stick

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 (**tongStudyEthernetCommunication2015**). It communicates with TCP/IP packets, and an illustration on how a client-server request and response could look like can be found in **??**.

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

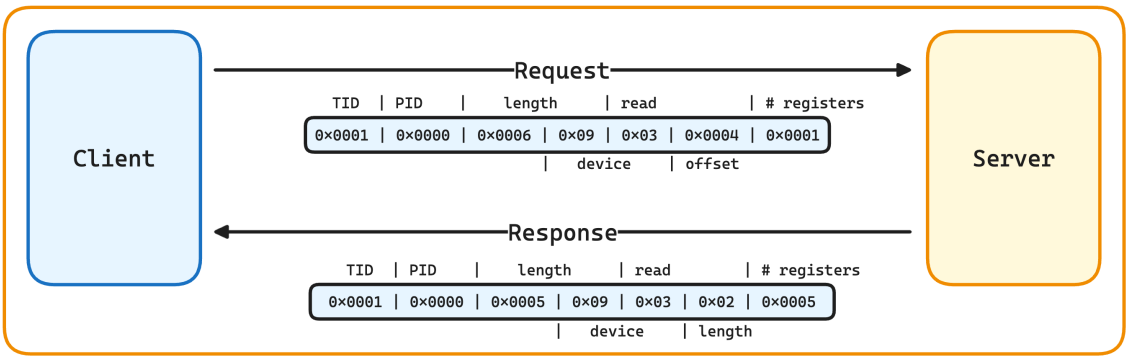

Figure 3.10: TCP/IP package exchange between a client and a server over Modbus TCP.

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 (**gmbhExpertPowerControl2023**). In **??** below, a simplified setup with the Gude acting as the server for communicating over Modbus TCP is shown.

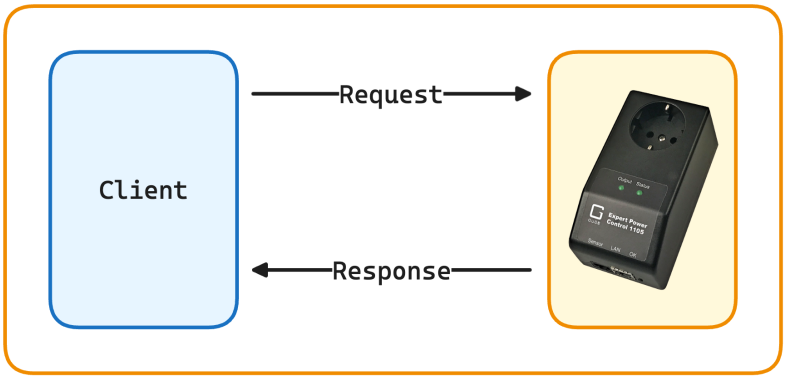

Figure 3.11: Client communicating with Gude over Modbus TCP.

Part II

Project

Chapter 4

Methodology

To reiterate, the goal of this thesis is to investigate the two problem statements presented in **??**. We are going to develop a FaaS prototype, named Nebula, which we are 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 **??**, 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 are 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 **SBC!** (**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 load (**bekarooPowerConsumptionRaspberry2016**).
2. Virtual Machine: A Debian-based VM running on an **IaaS!** platform (e.g., NREC), to measure performance relative to how a typical 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.

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.

4.1.2 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.

4.1.3 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.

4.1.4 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, 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 like a typical setup, 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 **??** below for a rough sketch on how we will set up benchmarking and energy measurements for our experiments.

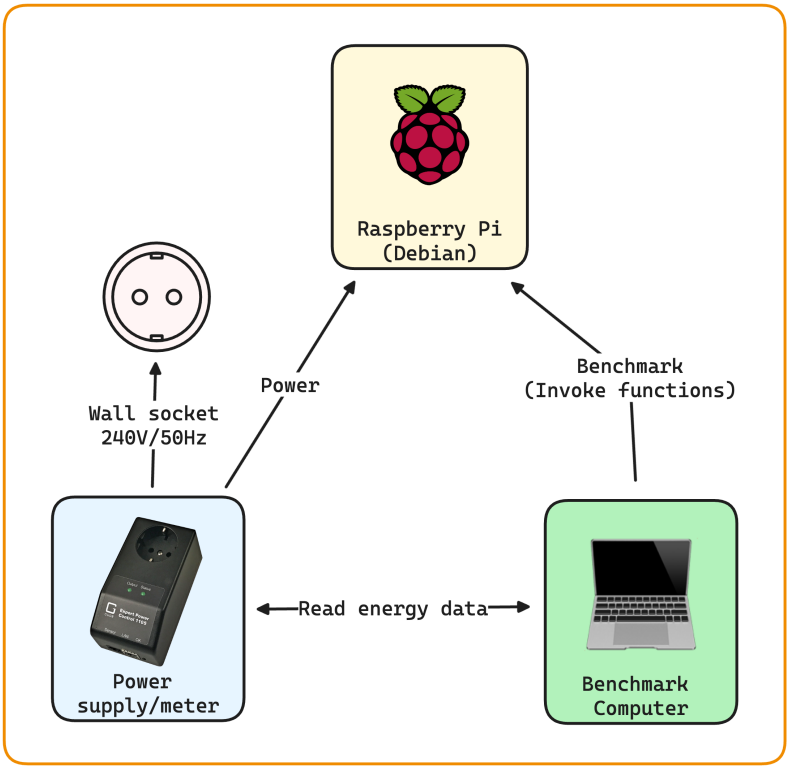

Figure 4.1: Reading energy consumption of our Raspberry Pi.

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.

Additionally, the following measurements will be gathered on the Raspberry Pi system system, as shown in **??**:

- 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: The average power consumption attributed specificall to function execution, calculated by subtracting the baseline power from the average 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:

$$E_{\text{Wh}} = P_W \times t_h$$

  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.

- Isolated Energy consumption: The energy in *Wh* consumed by the function, with the baseline power subtracted from the base *W* used to calculate this.

This data will then be saved in a JSON format and used for analysis.

4.1.6 Data analysis

4.1.7 Comparative analysis

Land on analysis method for my findings.

Finish methodology

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.

4.1.8 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 compared to existing research in the field, explored in **??**.

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

Based on the requirements outlined in **??**, we will design a prototype named Nebula, which is a serverless **FaaS!** platform capable of executing deployed functions as both **Wasm!** modules and Docker containers. The primary goal of Nebula is to serve as an experimental testbed for evaluating the performance and energy efficiency of **Wasm!** modules to traditional container-based deployments in the context of cloud-native applications. On top of this, we will need supporting infrastructure and utilities for benchmarking our prototype.

5.1 Prototype scope and Objectives

While popular serverless platforms offered by major cloud providers are typically closed-source, meaning how the work behind the scenes is hard to evaluate from the outside, this prototype focuses on the aspect of function deployment and execution of **FaaS!**. This controlled approach lets us concentrate on an evaluation of deploying and runnings functions as **Wasm!** modules versus Docker containers.

As a student thesis project, the prototype's scope is inherently limited. We are not going to implement features such as automatic scaling, distributed function deployments, authentication and authorization, integrations with file systems and databases, and more. However, this focused design is intentional, as we want to minimize other factors, and rather hone in on a controlled comparison between the two deployment methods.

5.2 System architecture

We will design Nebula to be a **FaaS!** platform that lets us execute functions deployed as both **Wasm!** modules and Docker containers. The system architecture will consist of several key components that work together to facilitate the development, deploymend, and execution of functions, as well as the benchmarking and evaluation of their performance and energy efficiency.

1. *Nebula core*: The central component of the prototype, responsible for handling incoming requests from users, spinning up deployed functions, and executing them. Nebula will support both **Wasm!** modules and Docker containers, providing an interface for interacting with the deployed functions.

2. *Function development and deployment*: A separate environment where example functions can be developed, either compiled to **Wasm!** modules or packaged into Docker images, and then uploaded to Nebula.

3. *Infrastructure*: Two environments will be used for performing the experiments: a Raspberry Pi, a low-power ARM-based **SBC!** serving as the deployment target for measuring energy, and a Debian-based virtual machine hosted on an **IaaS!** platform to simulate a typical cloud-native application deployment environment.

4. *Energy measurement setup*: A power supply capale of reporting energy readings will be used to supply the Raspberry Pi with power, allowing the collection of energy consumption data during function execution.

5. *Benchmarking utility*: A separate benchmarking application for performing our experiments. We will develop this to let us create a benchmark suite that can be reused and test a variety of variables. This utility will be responsible for reading energy measurements from our power supply, and consolidate energy readings with function executions.

?? illustrates the overall architecture of our Nebula prototype system, including benchmarking and energy measurements for our experiments.

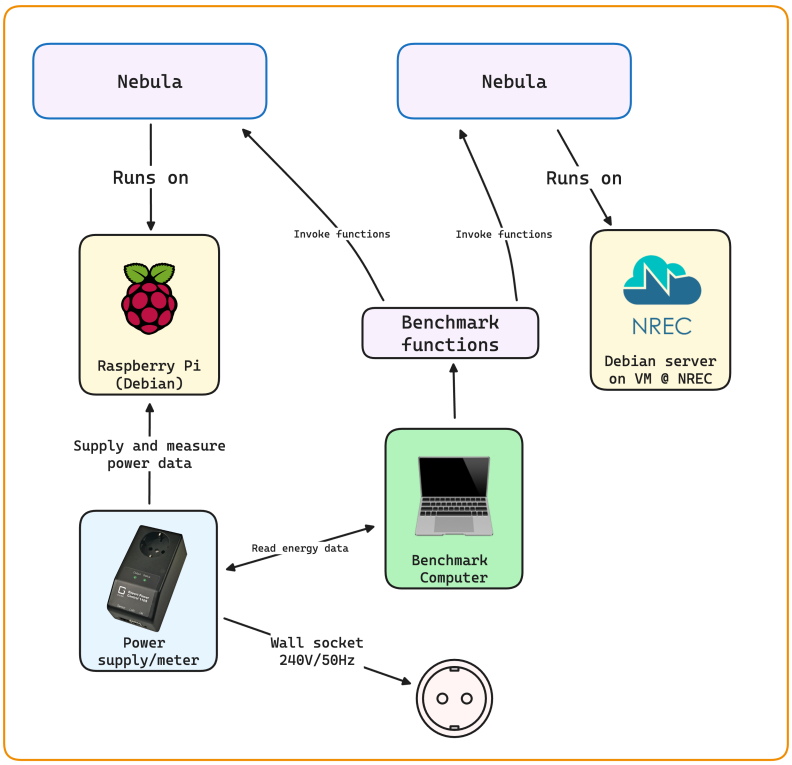

Figure 5.1: Overall architecture of our prototype system.

5.3 Nebula

Nebula will serve as the core component of the prototype. It will handle incoming requests, manage the lifecycle of deployed functions, execute them, and measure startup and runtimes of each execution. We will develop an interface for users to interact with the deployed functions, either as **Wasm!** modules or Docker containers.

Nebula's function execution process will involve several steps, as illustrated in **??**. When a user invokes a function through the interface, Nebula receives the request and routes it to the corresponding function based on the specified deployment method. Nebula will then start the function, execute it with the provided input data, and return the result back to the user.

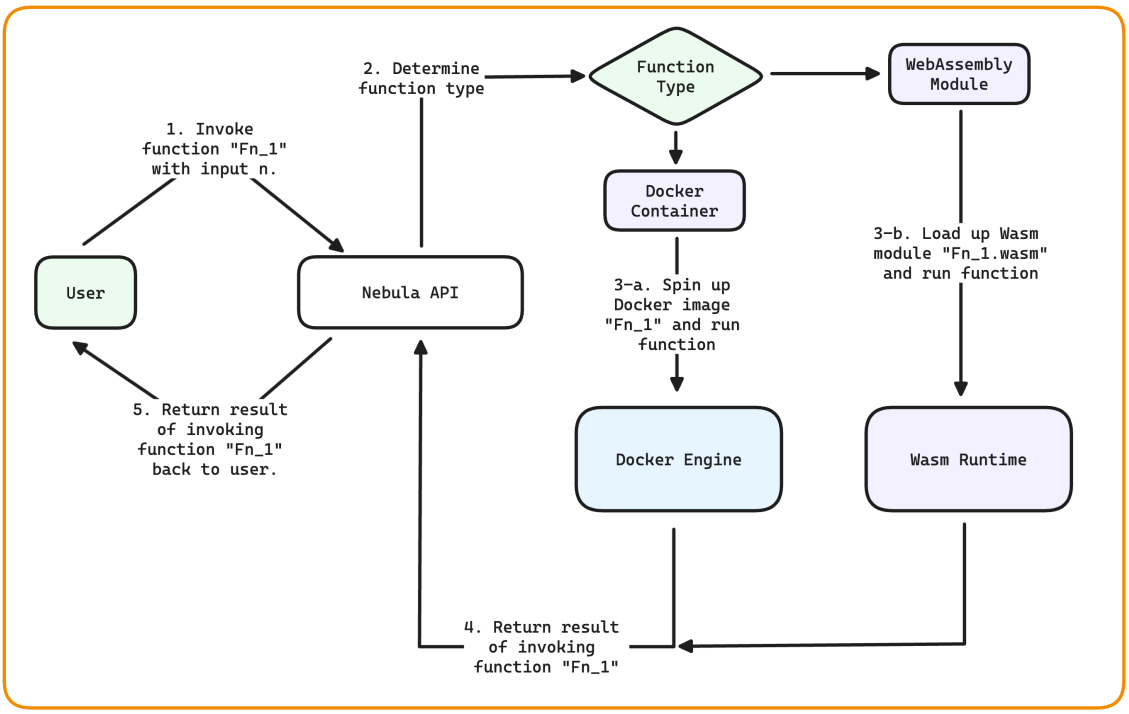

Figure 5.2: Flow of a user invoking a function on Nebula.

This function execution process will also include collecting metrics for each function invocation. We will start a timer at the start of our function execution that first times when our function is loaded into memory and starts executing, and then takes the time again at the end of the function. These metrics will be added to our function response and provide accurate measurements on how fast our functions start up and how much time is attributed to the function running itself.

5.3.1 Function packaging and deployment

Functions deployed to Nebula will either be in the form of a **Wasm!** module binary file, or as a Docker image that is uploaded to the server and loaded into the servers image repository. The packaging process will involve compiling the source code to **Wasm!** module binaries, or creating a Docker image with the use of a Docker file that describes its layers.

Our server will provide a way to let users upload their functions to the system. The deployment process will handle the necessary steps to make the functions available for invocation, such as storing the **Wasm!** modules or Docker image, and preparing them for invocation.

rephrase this mess

5.3.2 Function execution

To run functions we will need to provide the proper environments they can run inside of. For **Wasm!** modules, we will need a way to include a **Wasm!** runtime, as mentioned in **??**, which will let us run our functions in a sandboxed environment. The runtime will handle the loading and instantiation of the modules.

For Docker containers, we will use the containerization capabilities of Docker. Each function will run in its own isolated container, with the Docker Engine runtime managing its startup and shutdown.

For both methods, we will need a way to pass the users input to the function as a parameter and receive the result of each function invocation. The chosen way to pass input and receive output should be the same for each environment, to ensure controlled experimentation.

5.3.3 Interface

To serve as an interface for the user to invoke our functions, we will also develop a web server that will expose end points that our users can communicate with over HTTP in a REST-ful manner. In other word, we are going to develop a web server that exposes a REST API, where users can send requests that result in function invocation based on the request body.

5.4 Infrastructure

The Nebula prototype will be deployed on two different environments to evaluate its performance and energy efficiency:

- Raspberry Pi: A low-power, ARM-based **SBC!** will serve as the deployment target for measuring energy consumption during function execution. The Raspberry Pi will run Nebula and the deployed functions, and it will be connected to the energy measurement setup to collect power consumption data.
- Virtual Machine: A Debian-based virtual machine hosted on an **IaaS!** platform will be used to simulate a typical cloud-native application deployment environment. Nebula will be deployed on this virtual machine, and the functions will be executed in this environment to assess their performance in a cloud-like setting.

5.5 Energy measurements

To measure the energy consumption of our functions executed on the Raspberry Pi, a power supply capable of reporting energy readings will be used. The Raspberry Pi will be connected to the power supply and provide energy readings continually over the lifetime of our benchmarks. In **??** a couple of devices and protocols were presented that will be relevant for our system. We will attempt to set up energy measurements with each configuration, and see which best matches our desired outcome.

I ended up with Gude + Modbus tcp, should I just cut out Smart switch and MQTT/Zwave from the thesis, as they turned out to be too slow on reporting? Or could they still be relevant, considering they are also valid ways to measure energy readings, at a more affordable level?

The collected energy measurements data will be stored in a fitting data format and parsed for analysis and comparison between **Wasm!** modules and Docker containers.

5.6 Benchmarking utility

We want to minimize the amount of computing power attributed to our prototype as much as possible. With the exception of invoking the functions from a client, we want to isolate the computing power used to run our functions as much as possible. Because of this, we will use a separate computer for performing benchmarking. While there exists tools out there that could help us achieve parts of this, a custom application will be built that can both interface with the server through the REST API, and interface with the power supply over a chosen protocol.

The utility will be responsible for invoking the functions deployed on Nebula, both as **Wasm!** modules and Docker containers, and collect our metrics. The collected data will be stored and organized in a JSON format, and then analysed using the statistical techniques mentioned in **??**.

5.7 Experiment design

To evaluate the performance and energy efficiency of **Wasm!** modules compared to Docker containers, we will perform a series of experiments on our prototype. The experiments, as described in {sect:experiments}, will focus on measuring startup times, runtimes, power measurements and calculate the resulting energy

consumption. Each experiment will involve deploying the same function as both a **Wasm!** module and as a Docker container on Nebula. The functions will be invoked using the benchmarking utility, which will collect the relevant metrics for analysis.

The experiments will be performed on both the Raspberry Pi and the virtual machine environments to assess the performance in different deployment scenarios. The Raspberry Pi setup will additionally give us insights into the energy consumption of our functions, while the virtual machine will represent a cloud-native deployment environment, with a more powerful CPU and more memory, like we would expect of our cloud servers.

The data we end up with from the experiments will be analyzed and compared to evaluate the performance and energy efficiency of **Wasm!** modules versus Docker containers. We will perform multiple invocations of the same set of independent variables; function, module type and input, and derive median, mean and average of each invocation to strengthen our findings.

Finally, the results of our experiments will be visualized and discussed, relating the findings to the problem statements posed for this thesis. Here we will highlight the potential benefits and trade-offs of using **Wasm!** modules for serverless function deployment compared to traditional Docker containers, if the hypothesis we set out to explore ends up holding true.

Chapter 6

Implementing Nebula

As mentioned in the acknowledgments at the start of this thesis, the idea for this project came from a podcast episode where the CEO of Fermyon, Matt Butcher, told the journey of how they ended up developing their cloud platform as a way to offer a **FaaS!** powered by **WebAssembly!** (**WebAssembly!**) modules. They have built a cloud application platform for **Wasm!**-based serverless functions named Fermyon Cloud, which is supported by their developer framework, Spin. Spin is a framework for building **Wasm!**-based functions that can be deployed on their cloud, and this setup acts as an inspiration for the prototype we are going to build. Both of these projects are open-sourced and available on Github, so we can take a look at how they've implemented their services to run **Wasm!** modules in a serverless environment.

In this chapter, the code snipets are presented in a condensed form, focusing on the parts relevant to our experiments. There is also an interactive GUI built with HTMX and Askama HTML templating which the user can use to manually interact with each deployed function, but this aspect is more a novelty for manual testing, rather than a feature that supports this thesis. A lot of the source code will not be shown, we are going to focus on the code that lets us run functions as either **Wasm!** modules or Docker containers.

6.1 Choosing a Tech Stack

The programming language Rust was chosen to build our prototype. This choice was driven by Rust's strong relationship with **Wasm!** and its suitability for building efficient and scalable applications. Developed by Mozilla Research, the same company that developed **Wasm!**, Rust emphasizes memory safety, concurrency, and parallelism while combining high performance with strong safety guarantees. Its syntax is somewhat similar to TypeScript, allowing developers write low level

programs in a high-level manner

6.1.1 Rust and WebAssembly

Rust's close relationship with **Wasm!** stems from their shared origin at Mozilla Research, where both technologies stems from. Rust's ecosystem offers a wide range of crates (Rust's naming convention for libraries) for various **Wasm!** runtimes, enabling us to ship the entire Wasm runtime required for running our Wasm modules as part of the web server itself. This approach eliminates the need to install a separate runtime on our server, simplifying the prototype development.

6.1.2 Wasmtime

For our **Wasm!** runtime, we opted for Wasmtime[1]. Wasmtime offers seamless integration with Rust through the official wasmtime crate and provides portability across platforms, including ARM for our Raspberry Pi deployment. The choice for Wasmtime was inspired by Fermyon's choice to build their platform on it. A viable alternative is Wasmer[2], which can also be installed as a crate and shipped with a Rust application.

6.1.3 Docker Environment

For our Docker environment, we installed Docker Desktop on both our development computer and each deployment target. Docker Desktop supports Debian on ARM64 for our Raspberry Pi deployment. It also supports Debian on amd for our virtual machine. We implemented a Rust function to spin up a command for running our Docker images during execution.

6.2 Nebula Core

In this section we will describe how we built the core of Nebula, with the following components:

1. **Web Server**: A RESTful API server written in Rust andand utilities used by both the web server and function runners. built using the Actix Web framework, responsible for receiving function invocation requests and managing the execution of functions, serving as the interface for our users.

[1]https://wasmtime.dev/
[2]https://wasmer.io/

2. **Function Runners**: Separate modules for executing functions deployed as Wasm modules or Docker containers, encapsulating the logic for invoking and managing the respective execution environments.

3. **Shared Library**: A common library containing shared types the function runners.

6.2.1 Building the Web Server

The web server serves as the entry point for Nebula, exposing a RESTful API for function invocation requests. We built it using the Actix Web framework, which leverages the asynchronous Rust runtime Tokio for handling concurrent requests.

```
// Include crates
use axum::{routing::post, Router};
use std::net::{IpAddr, Ipv4Addr, SocketAddr};
use tokio::net::TcpListener;
// Supporting library for our function execution
use nebula_server::api::call_function::call_function;
// Entry point, Tokio main function
#[tokio::main]
async fn main() {
  let router =
    Router::new().route("/invoke_function", post(call_function));
  // Precompile and serialize our Wasm modules, more on that later.
  serialize_modules();
  let localhost = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));
  let listener = TcpListener::bind(&SocketAddr::new(localhost, 3090))
    .await
    .unwrap();

  axum::serve(listener, router.into_make_service()).await?;
}
```

The main entry point defines a router with a single POST endpoint `/invoke_function`, which is bound to the `call_function` handler. This handler function determines the requested function type (Wasm or Docker) and routes the request to the corresponding function runner module.

6.3 Calling functions

To handle function invocation requests, Nebula defines a set of types to represent the request parameters, function results, and associated metrics. This is stored in the shared library and lets us reuse the same types for our runner functions and the web server implentation.

```
mod serialize_wasm;

enum FunctionType {
  Docker,
  Wasm,
}
struct FunctionRequest {
  function_name: String,
  input: String,
  module_type: FunctionType,
}
struct Metrics {
  startup_time: u64,
  total_runtime: u64,
  start_since_epoch: u64,
  end_since_epoch: u64,
}
struct FunctionResult {
  func_type: FunctionType,
  func_name: String,
  input: String,
  result: String,
  metrics: Metrics,
}
```

The `call_function` handler function is responsible for dispatching the request to the appropriate function runner based on the requested function type.

```
use axum::{http::StatusCode, response::IntoResponse, Form};
use nebula_lib::{docker_runner::run_docker, wasm_runner::run_wasm};
use serde::Serialize;
// Define response body, that can be serialized to JSON
```

```rust
#[derive(Serialize)]
struct Response {
  result: FunctionResult,
}
// Call function that takes the request as a form and invoke the correct type
async fn call_function(
  Form(request): Form<FunctionRequest>,
) -> impl IntoResponse {
  let result: FunctionResult = match request.function_type {
    FunctionType::Docker => {
      let docker_function = format!(
        "nebula-function-{}-{}",
        request.function_name, request.base_image
      );
      run_docker(
        &docker_function,
        &request.input,
        request.function_name,
        request.base_image,
      )
      .unwrap()
    }
    FunctionType::Wasm => {
      let function_path = get_file_path(&request.function_name);
      run_wasm(&request.input, function_path, &request.function_name)
        .unwrap()
    }
  };

  let body = serde_json::to_string(&Response { result }).unwrap();

  return (StatusCode::OK, body).into_response();
}
```

The function runners for Wasm modules and Docker containers are implemented in separate modules within the shared library. This separation of concerns makes it easier to maintain and possibly extend the codebase later, in case anyone wants to build upon this project in the future.

6.3.1 Executing Wasm modules

The `run_wasm` function is responsible for executing a Wasm module based on the provided input and module path. We are using the Wasmtime crate, the official Rust library for executing Wasm modules, and the wasmtime-wasi crate for providing a WebAssembly System Interface (WASI) implementation, which lets us pass input as stdin to our functions.

```
use std::{path::PathBuf, time::Instant};
use wasi_common::pipe::{ReadPipe, WritePipe};

use wasmtime::*;
use wasmtime_wasi::sync::WasiCtxBuilder;

fn run_wasm(
  input: &str,
  wasm_module_path: PathBuf,
  func_name: &str,
) -> Result<FunctionResult, anyhow::Error> {
  let start_since_epoch = current_micros()?;
  let start = Instant::now();
  // 1.
  let engine = Engine::default();
  let mut linker = Linker::new(&engine);
  // 2.
  wasmtime_wasi::add_to_linker(&mut linker, |s| s)?;
  let stdin = ReadPipe::from(input);
  let stdout = WritePipe::new_in_memory();
  let wasi = WasiCtxBuilder::new();
  let mut store = Store::new(&engine, wasi)
    .stdin(Box::new(stdin.clone()))
    .stdout(Box::new(stdout.clone()))
    .build();
  // 3.
  let module = load_module(&engine, wasi_module_path)?;
  let startup_time = start.clone().elapsed().as_micros();
  linker.module(&mut store, "", &module)?;
  // 4.
  linker
    .get_default(&mut store, "")?
```

```
        .typed::<(), ()>(&store)?
        .call(&mut store, ())?;
    drop(store);
    // 5.
    let contents: Vec<u8> = stdout
        .try_into_inner()
        .map_err(|_err| anyhow::Error::msg("sole remaining reference"))?
        .into_inner();
    let result = String::from_utf8(contents)?.trim().to_string();
    Ok(FunctionResult {
        result,
        metrics: Metrics {
            startup_time,
            start_since_epoch,
            total_runtime: start.elapsed().as_micros(),
            end_since_epoch: start_since_epoch + total_runtime,
            startup_percentage: ((startup_time as f64
                / total_runtime as f64)
                * 100.0)
                .round(),
        },
        func_type: ModuleType::Wasm,
        func_name: func_name.to_string(),
        input: input.to_string(),
    })
}
```

The key steps involved in executing a Wasm module are:

1. Initialize the Wasmtime engine and linker.
2. Set up the WASI context, including stdin and stdout pipes for passing input and receiving output.
3. Load the correct Wasm module from the provided path. This path is determined in the `call_function` and lets our runner know where our .wasm binary files are stored on our server. In our case it's in `/root/modules/wasm/`.
4. Instantiate the module and execute the modules default function, which corresponds to each functions main function, which we will show later.
5. Capture the output from stdout and return it as the function result.

Throughout the execution process, we collect startup and runtime metrics which

are included in the returned `FunctionResult` struct. As we will see in **??**, our **Wasm!** modules startup

6.3.2 Executing Docker Containers

The `run_docker` function is responsible for executing a function deployed as a Docker container. It spawns a new Docker container based on the provided image and function name, passing the input data as stdin.

```
use std::{
  io::{Error, Result, Write},
  process::{Command, Stdio},
  time::Instant,
};

pub fn run_docker_image(
  image_name: &str,
  input: &str,
  func_name: String,
) -> Result<FunctionResult> {
  let start_since_epoch = current_micros()?;
  let start = Instant::now();
  // 1.
  let mut child = Command::new("docker")
    .args(["run", "--rm", "-i", image_name])
    .stdin(Stdio::piped())
    .stdout(Stdio::piped())
    .spawn()?;
  let cmd_start = current_micros()?;
  // 2.
  child.stdin.as_mut().unwrap().write_all(input.as_bytes())?;
  // 3.
  let output = child.wait_with_output()?;
  let stdout = String::from_utf8_lossy(&output.stdout);
  // 4.
  let (result, actual_startup) = parse_output(&stdout, cmd_start)?;
  Ok(FunctionResult {
    result,
    metrics: Metrics {
      startup_time: actual_startup,
```

```
      total_runtime: start.elapsed().as_micros(),
      start_since_epoch,
      end_since_epoch: start_since_epoch + total_runtime,
    },
    func_type: ModuleType::Docker,
    func_name,
    input: input.to_string(),
  })
}
```

The key steps involved in executing a Docker container are:

1. Start up a Command process, from Rust's standard library, which spawns a `docker run` command that we provide with image name and append stdin and stdout processes.
2. Write input to the stdin, which the function that is packaged inside our container is waiting on.
3. Wait for function inside the Docker container to execute and write out the result as stdout.
4. Parse output from the Docker container with a helper function, which comes in the form of a tuple of (result, micro_seconds_since_epoch). We will use the second item in this tuple pair for determining the actual startup time of our function, which when the Docker container starts executing.

Similar to the **Wasm!** module execution, we collect relevant metrics for startup and runtime of our functions invoked as Docker containers.

6.3.3 Challenges and insights

During the implementation of Nebula's core functionality, several challenges were encountered that required creative problem-solving and valuable insights from experts from the **Wasm!** community.

Initially, passing input data to Wasm modules proved challenging due to the need for shared memory and exported functions. This issue was resolved by leaning into the the POSIX-like behavior of WASI and provide input via stdin and output to be captured from stdout, similar to traditional command-line programs. As **malmgrenGettingDataOut2022<empty citation>** suggested in their helpful article, using this feature of WASI provided a simpler approach for handling input and output in our Wasm module execution model. This approach was also

implemented for passing input and output to and from our Docker containers, and while using stdin and stdout should incur some cost in terms of our metrics, this cost applies to both deployment methods, ensuring a fair assessment.

Another challenge arose when timing the startup of Docker containers. The first iteration involved defining a startup_time variable immediately after spawning the Command process. However, this naive implementation resulted in misleading "cold start" times of practically zero seconds, which would have invalidated our hypothesis. Fortunately, a former colleague, Roger Slaaen, suggested timing the startup at the moment the Docker images start running. This proved to be the key for solving this challenge as this accurately captures the cold start of our Docker-based functions when the container are loaded and start running.

Finally, the first implementation of the **Wasm!** module execution had longer cold start times than expected, despite maintaining a fast runtime. While attending the WasmIO conference in 2024, I met Syrus Akbary, the founder of Wasmer, whom I showed my prototype to and mentioned my cold start issue.

Akbary, who works on the core of Wasmer runtime, shared key insights into how a runtime prepares a **Wasm!** module for execution. The long startup latency turned out to be caused by a naive understanding of how the modules should be loaded, and with the first iteration, Wasmtime had to load "raw" Wasm binary files. To execute our modules on the runtime, we have to compile raw binary files into a binary file that Wasmtime can understand. This step was crucial in unlocking the perfomance we will explore in **??**. This method was backed by the employees at Fermyon, who were also attending the same conference, and shared that their framework Spin does the same thing while building **Wasm!** modules for their platform. The code that implements this feature is as follows:

```
use std::{fs::File, io::Read, io::Write, path::PathBuf};
use wasmtime::{Engine, Module};
// Serialize modules
fn serialize_wasm_modules(
  module_dir: PathBuf,
  serialized_dir: PathBuf,
) {
  // 1.
  let modules = list_files(module_dir.to_str().unwrap())?;
  // 2.
  let engine = Engine::default();
  // 3.
```

```
  for module in modules {
    let module_name = module.file_name().unwrap();
    // 4.
    let module = Module::from_file(&engine, &module)?;
    let module_bytes = module.serialize()?;

    let target_module = serialized_dir.join(module_name);

    // 5.
    let mut file = std::fs::File::create(target_module)?;

    file.write_all(&module_bytes)?;
  }
}
```

The steps involved in serializing our modules are:

1. Get a list of all the **Wasm!** modules present in our deployment folder.
2. Instantiate the same kind of Wasmtime engine that we use in our function execution function.
3. Loop through all the deployed modules in our folder.
4. Load the raw **Wasm!** binary and compile into a `Module` which Wasmtime can run on its engine. Then serialize the resulting compiled binary represented by bytecode.
5. Write the resulting bytecode into a file that we store in a separate folder from our deployment storage.

This sequence precompiles our raw **Wasm!** binaries into bytecode we can load and execute upon request. The code for loading the binaries can be seen here:

```
use std::{fs::File, io::Read, path::PathBuf};
use wasmtime::{Engine, Module};
// Read modules from file and deserialize as bytes
fn load_module(
  engine: &Engine,
  wasi_module_path: PathBuf,
) -> anyhow::Result<Module, anyhow::Error> {
  // 1.
  let mut module_file = File::open(wasi_module_path)?;
```

```
    let mut module_bytes = Vec::<u8>::new();
    // 2.
    module_file.read_to_end(&mut module_bytes)?;
    // 3.
    let module = unsafe { Module::deserialize(engine, &module_bytes)? };
    // 4.
    Ok(module)
}
```

Here we load the requested precompiled and serialized **Wasm!** binary from our file system and deserialize it into a **Wasm!** module we can run on Wasmtime. This function is called on each invocation of our **Wasm!** modules. It is important to note the presence of an unsafe code block surrounding the `Module::deserialize` function. As Rust is a statically typed program, we have to provide the types for our code ahead of time, before compilation, and even if we are "pretty sure" that we know what the contents of our module_bytes are, we can not be entirely sure, so we have to opt out of Rust's safety in this case. This is not a crucial downside for our experiments, but hopefully something that can be changed in a later version of Wasmtime, in order to minimize the risk of this potential insecurity.

The resulting application built from this source is now ready to execute functions as either **Wasm!** modules or Docker containers. However, to achieve this, we need to implement our environment for developing and deploying these functions.

6.4 Function development and deployment

Now that we have a web server that we can start on our deployment targets, we are going to need some functions that we can deploy and run on Nebula. For thatj we are going to develop the functions described in **??** in Rust and set up a way to package our functions in a way that we can spin them up and conduct the experiments we have planned.

6.4.1 Development Environment

For our development environment, we are going to set up some parts:

- A shared library: We are relying on stdin and stdout to pass input and output from our functions, and ideally we won't have to implement that individually for each of our functions. Therefore we are going to build a shared library that exposes functions for reading from stdin and printing to stdout.

- Timing docker cold start: In this library we will also implement a helper function for getting the actual time of cold start for our Docker containers.
- A Make file and Dockerfile: for streamlining the build and deployment of our functions to our target.

6.4.2 Shared library

We have three goals with our shared library code. First we need a single way to run our functions that wraps our example functions with reading from stdin and writing to stdout. Secondly, we want a way to time the actual startup times of our docker containers. Finally we will write a Dockerfile that we will use to build your Rust functions into Docker images and a Makefile we can use to run to streamline the build and deploy of each function with ease.

The pipeline we want to support is:

```
cargo new example_function
# Implement example function in ./example_function/src/main.rs
make build_wasm build_docker
make deploy_wasm deploy_docker
```

To achieve this pipeline we will implement the following:

```
fn get_stdin<T: FromStr>() -> Result<T> {
  let mut input = String::new();
  // Lock process and wait for stdin to be passed in
  stdin()
    .lock()
    .read_line(&mut input)
    .context("Failed to read line")?;
  // Parse input to a String and return the value
  input.trim().parse::<T>().map_err(|_| {
    Error::msg(format!(
      "Failed to parse input as {}",
      type_name::<T>()
    ))
  })
}
// If timestamp was recorded for function, print it ot stdout
fn print_stdout<T: Display>(output: T, timestamp: Option<String>) {
```

```
  match timestamp {
    Some(timestamp) => println!("{}|{}", output, timestamp),
    None => println!("{}", output),
  }
}
pub fn run_function<T, F, R>(func: F, func_type: FunctionType)
where
  T: FromStr,
  F: Fn(T) -> R,
  R: std::fmt::Display,
{
  // If function is Docker, get timestap, else if Wasm it is of type "None"
  let timestamp = match func_type {
    FunctionType::Docker => docker::get_epoch_timestamp(),
    FunctionType::Wasm => wasm::get_epoch_timestamp(),
  };
  // Read from input
  let input: T = get_stdin().expect("To parse correctly");
  // Call function with input and receive result
  let result = func(input);
  // Print input to stdout, with timestamp if func_type was Docker
  print_stdout(result, timestamp)
}
```

In this library we have exposed a `run_function` wrapper that we will import into the code for each of our benchmark functions. For each of our functions we want to start with timing the start of our function, if it is Docker. If it is not, the timestamp will be empty and not returned to Nebula. In hindsight, we could have measured the start time in the same way for our **Wasm!** modules, but some manual testing revelead a negligble difference, which shouldn't have any major impact on our results.

After timing the start of the function, we read from the stdin, run the function with the input value of type T (generic type, which can change depending on our functions). Then we print the result to stdout, either as a pair of `result | timestamp`, or just the result.

Then we had to specify a Dockerfile that our Docker build process will read from that describes how our functions are to be packaged into a Docker image. When we build with this Dockerfile, we also provide the target architecture we want our

Docker container to run on. For our NREC deployment, we will target linux/amd64 and for the Raspberry Pi, we will target linux/arm64.

```
FROM rust:1.70 as build
WORKDIR /usr/src/nebula
# Copy shared library into layer
COPY shared /usr/src/shared
# Expose function name (folder) as an argument for Docker
ARG PROGRAM_NAME
COPY $PROGRAM_NAME .
# Build release binary with timestamp
RUN cargo build --release --features docker
# Target smaller release docker image
FROM debian:bullseye-slim
# Install required libraries/dependencies and update
RUN apt-get update && \
    apt-get install -y libc6 && \
    rm -rf /var/lib/apt/lists/* ;\
# Copy over rust binary, and a custom run.sh script
COPY --from=build /usr/src/nebula/target/release/$PROGRAM_NAME /usr/local/bin/
COPY --from=build /usr/src/nebula/run.sh /usr/local/bin/
# When container is invoked, run this shell command
CMD ["run.sh"]
```

In this Dockerfile, we use the official Docker image for Rust as the build layer. Then we copy over the required images into this layer, build our functions with the release flag (ensuring best performance), and flag that we want to print the timestamp for these Docker functions through a feature flag.

Then we use a smaller debian:bullseye-slim image as a target for our final Docker image as we do not need the entire Rust toolchain installed in our final image to run our functions as Rust binaries.

Finally, we define a simple run.sh script, which is identical for every function, as shown here:

```
#!/bin/sh
/usr/local/bin/<function_name>
```

This serves as the script that we run when we invoke our Docker functions as described in **??**, invoking the main function of our benchmark functions.

Finally, we created a `Makefile` to streamline the commands required for building and deploying our functions, following the process described earlier.

While explaining the code for this file is not crucial for our research, **??** illustrates the flow it follows.

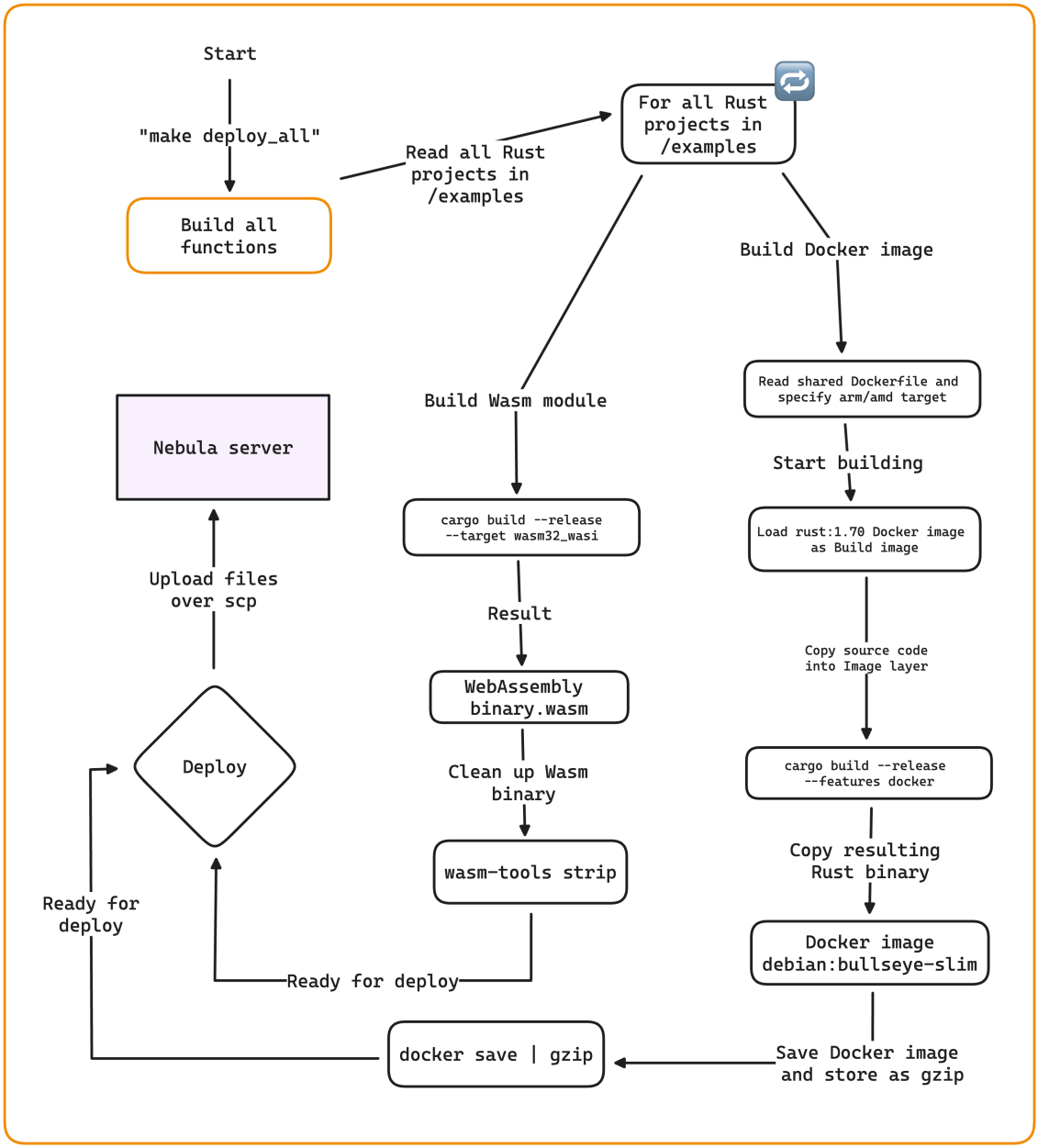

Figure 6.1: Streamlined flow of building and deploying functions.

One thing to note in this flow, is the step *wasm-tools strip*. *wasm-tools* is a CLI tool maintained by the Bytecode Alliance that includes a series of tools we can use to manipulate our **Wasm!** modules. Compiling a Rust program to **Wasm!** includes some unnecessary sections that we can strip away. The **Wasm!** binaries we built for our experiments resulted in ~2Mb in size. Using this tool lets us cut

this down to ~200Kb instead, while still maintaining its function. More on that in our **??**.

6.4.3 Implementing a benchmark function

Now that we have set up an environment for building and deploying functions to Nebula, we can look at how we developed the benchmark functions themselves. We will implement the recursive Fibonacci function in this section.

```
use shared::{run_function, FunctionType};
// Reads std in as input, retrieves the fibonacci sequence and returns the last num
// fibonacci sequence of the provided size
fn main() {
  // Determine if function type is docker or wasm
  let func_type = if cfg!(feature = "docker") {
    FunctionType::Docker
  } else {
    FunctionType::Wasm
  };
  run_function(fib, func_type)
}
// Recursive function that calculates the nth fibonacci number
fn fib(size: i64) -> i64 {
  match size {
    0 => 0,
    1 => 1,
    n => fib(n - 1) + fib(n - 2),
  }
}
```

Our shared library handles the core functionality. As described in **??**, we treat our functions as normal programs, which we do by running them by invoking their main function. In this main function we determine if the function will be compiled to a **Wasm!** module or a Docker container based on the `--features docker` flag we pass in our build command. (See **??**)

Then we implement the `fib` function which takes in what number in the Fibonacci sequence we're interested in, which we have implemented recursively, and pass this into our `run_function` wrapper from our shared library.

With that in place, we just need to set up our deployment targets - the Raspberry Pi and the Debian-based virtual machine on NREC.

6.5 Deployment Setup

For our research we have decided to set up Nebula both on a Raspberry Pi running Debian on an ARM-based CPU, and on a virtual machine running on **NREC!** (**NREC!**), with gives us an AMD-based virtual CPU.

6.5.1 Raspberry Pi Setup

The Raspberry Pi, a low-cost and energy-efficient **SBC!**, was chosen as one of our deployment targets to measure power consumption of our function executions. The model we chose was the Raspberry Pi 4 Model B, with 4GB of RAM, running the lite version of the Raspberry Pi OS, which is built on top of Debian.

6.5.1.1 Hardware setup

1. Use the Raspberry Pi image tool for installing Raspberry Pi OS (lite, without desktop). Configure the OS with:
2. Wifi credentials, if not using ethernet.
3. SSH key, created with ssh-keygen, used to authenticate when connecting to device through SSH. This is also useful for uploading our files through scp.
4. Insert the microSD into the device and connect it to power, which boots it up.
5. If device connects to internet, it will be discoverable as `raspberrypi.local`, which can be used instead of its IP-address.

6.5.2 Virtual Machine setup (NREC)

For our experiments, we chose **NREC!** as our **IaaS!** platform for hosting our virtual machine and benchmark a more typical deployment environment. To use this service, we had to contact the administration and verify that we were a student at the University of Oslo.

Once inside the NREC platform, we can launch an instance of our virtual machine, following their guide for doing so. For our experiments we chose the pre-configured flavor of `m1.medium`, which gives us 1 virtual CPU, 4GB of RAM and 20GB storage space. The virtual machine was built with the base image of Debian 12. After this is set up, we can ssh into the server with a ssh-key we provided during setup.

add guide?

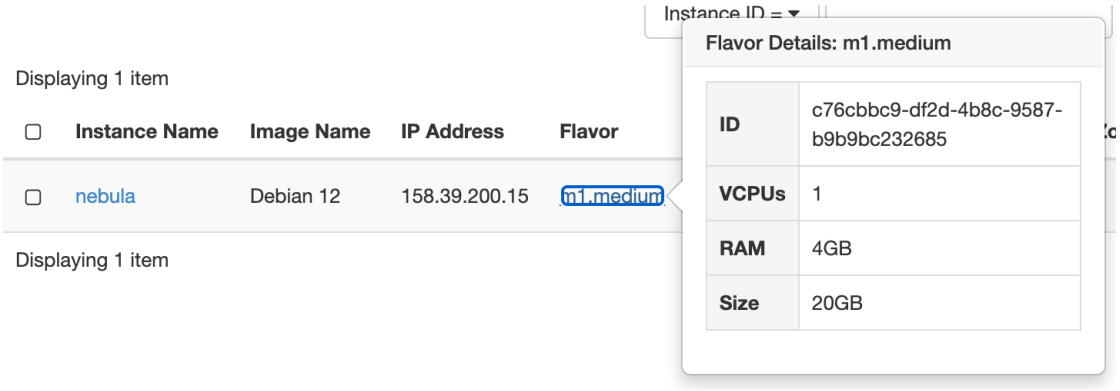

Figure 6.2: Our virtual machine configured on NREC.

6.5.3 Software setup

As both our setups are running a flavor of Debian, setting up the software required to run our experiments are identical.

- First we update the package lists and upgrade pre-installed packages on our server:

```
sudo apt update
sudo apt full-upgrade
```

- Then we install Docker Engine following their official guide from their manual[3].

At this point, we are ready to deploy Nebula to each server and deploy our functions.

6.5.4 Tying it all together

Once we have install the required software on our servers, we can start to deploy our Nebula web server and upload our **Wasm!** modules and Docker images.

In our main project folder, we build Nebula using the cargo build command. We add the flag `--release`, which optimizes the binary for production, and the flag `--target <target-arch>`, which cross-compiles Nebula to the target architecture. Our prototype was developed on a M2-based Macbook pro, so the default build target is not supported on our target platforms. Cross-compiling works fine, but the easiest and fastest way to build Nebula is to do it on the target architecture itself.

[3]https://docs.docker.com/engine/install/debian/

For the Raspberry Pi, this was done by cloning the Github repository with the source code onto the device, and building it from there. For the NREC implementation, a Github action was implemented for building the application on a Linux VM as part of their CI/CD service. This is not a crucial detail for our research, but valuable for anyone who would want to recreate the experiment and expand upon it.

Finally, we build and deploy our functions as described in **??**, and upload our **Wasm!** binaries and our gzipped Docker images to each server. Part of deploying the Docker images is loading them into the local Docker image list, done with the command `docker load -i /path/to/image.tar.gz`. When we start our server we call the `serialize_wasm()` function and serialize our **Wasm!** modules into byte_code that we can load and run on Wasmtime. With all this in place, we can now perform our experiments.

6.6 Benchmarking

Now that we have our Nebula **FaaS!** prototype up and running on both our Raspberry Pi and virtual machine on NREC, we can start implementing the benchmarking utility that will be used to conduct the experiments and collect our data. While there exist tools out there for performing experiments like this, the setup we have implemented would be difficult to cover with an already existing tool. Because of this, we will develop a custom benchmark utility in Rust, which will let us collect the data we're interested in.

For our experiments we will need to implement two core functionalities. First, we need to implement a function that loops through a pre-defined set of input values for each function and perform POST requests to Nebula, storing the function results on our computer. We also need a function for measuring power measurements while performing our POST requests to Nebula on our Raspberry Pi.

6.6.1 Stress-testing Nebula

To benchmark Nebula's cold starts and runtime efficiencies, we will need to perform a range of POST requests with the intent of gathering a wide range of data that we can compare later. For this we rely on the widely used Rust crate, `reqwest`, a library for performing http requests in Rust. The code for performing the request in itself is omitted from this example, but the suite of functions and their corresponding inputs we're testing with can be seen here:

```rust
async fn stress_nebula(
  client: Client,
  url: &str,
) -> anyhow::Result<Vec<FunctionResult>> {
  // 1. function name, how many requests, steps
  let functions: Vec<(&str, u32, u32)> = vec![
    ("exponential", 353, 2),
    ("factorial", 130, 1),
    ("fibonacci-recursive", 40, 1),
    ("prime-number", 600, 10),
  ];
  let mut function_results: Vec<FunctionResult> = Vec::new();
  // 2.
  for function in functions {
    // 3.
    for input_value in 0..=function.1 {
      // 4.
      let input_value = input_value * function.2;
      // 5.
      for _ in 0..6 {
        // 6.
        let wasm_results = make_request(
          &client, url, function.0,
          "Wasm", &input_value.to_string()
        ).await?;
        function_results.extend(wasm_results);
        let docker_results = make_request(
          &client, url, function.0,
          "Docker", &input_value.to_string(),
        ).await?;
        function_results.extend(docker_results);
      }
    }
  }
  // 7.
  Ok(function_results)
}
```

For our benchmark functions, we have found a range of values that through manual

testing, that will give us a clear picture on how our prototype performs when running functions as **Wasm!** modules or Docker containers.

The steps we take in our stress-test function are:

1. Define a list of modules that we are going to benchmark. For each function we have a tuple that consist of three values.
    - The name of the function.
    - The amount of input values we want to test for each function.
    - The factor we want to multiply our input values with, to cover a wider range of input values for our experiments.
2. Loop through our list of functions.
3. For each function, loop through a range of <0, steps>.
4. Determine input value by multiplying current step with factor.
5. Perform each set of function and input 6 times. This is in order to gather a mean of each metric for our results later, to account for variance in our results.
6. Perform a function invocation for each function as both Wasm and Docker sequentially, then append result to the final result.
7. Return the list of results.

The final result of this, is a long list of values that we store as JSON, for further analysis.

An example of a function result from benchmarking on our Raspberry Pi looks like this:


```
{
  "func_name": "fibonacci-recursive",
  "func_type": "Wasm",
  "input": "2",
  "result": "1",
  "metrics": {
    "startup_time": 3778,
    "total_runtime": 8197,
    "start_since_epoch": 1714755235731991,
    "end_since_epoch": 1714755235740188
  }
}
```

Part of our metrics is the timings for microseconds since epoch for the start and end of each function invocation, this is important for our power measurements.

6.6.2 Measuring power

For communicating with our Gude expert control over Modbus TCP, we are going to use the `tokio-modbus` crate. This library will let us write an interface against the device over our network.

As was presented in **??**, we saw how a client and server communicates through Modbus TCP packets. In figure **??** below, we zoom into this packet, as they are important for our implementation for communicating with our Gude power supply.

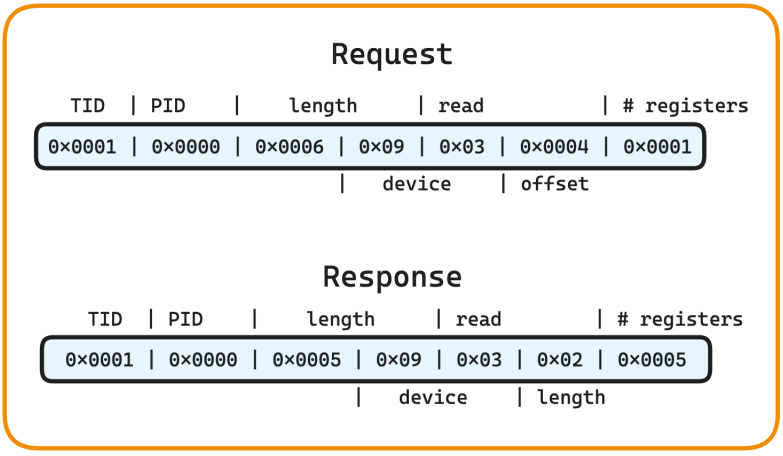

Figure 6.3: A Modbus TCP packet.

To read from the packets we receive from our Gude power supply, we need to know which register each measurement it can report to are registered. From the manual of our Gude Expert Power Control 1105, we find this table of sensor that it can report on. In **??** below, we have extracted the most relevant sensors for our measurements, it's register address, and how accurate it's reported units are.

Table 6.1: Gude Expert Power Control 1105 sensors.

Offset	Sensor.Field	Address	Unit
0	Absolute Active Energy	0x400	Wh
1	Power Active	0x402	W
2	Voltage	0x404	V
3	Current	0x406	mA
4	Frequency	0x408	0.01hz
5	Power Factor	0x40a	0.001

As we can see from **??**, we can read Absolute Active Energy, a value that is the cumulative sum of all energy consumed on our device, measured in Wh. Power active reports the current W being drawn at the moment of reading the sensor, the product of Voltage and Current. At first glance, these two sensors seem ideal for our benchmarkings, but as we will reveal in **??**, the energy it takes to invoke a function as either a **Wasm!** module or Docker container, will be measured in µWh. Measurements in whole Watts will not provide accurate enough data for our experiments. The three most important sensors for our experiments will be Voltage, Current and Power Factor.

To re-iterate, the formula for calculating power is $P = U \times I$, where U is voltage and I is current. For example, with a constant voltage of 240V and a current of 24mA, we will get $240V \times 24mA = 5,76mW$. Another important sensor to read from is the Power factor, which is a factor that indicates how much power the connected device is actually consuming, compared to the total draw of current through the power cable. For our system it varied between 0.50 and 0.55, so with an example of $PF = 0.55$ for our example above, we get $240V \times 24mA \times 0.55 = 3,168mW$. This product is what we're going to collect data on.

In **??** we will discuss some challenges regarding this setup, and most importantly that we are going to assume a constant voltage of 240V for calculating our power measurements. This is because for every sensor we want to read from our Gude device at a given measurement adds up to the total amount of time it takes to read this measure. When we limit our readings to focus on current and power factor, our energy measurements come in fast enough to accurately time them against our function executions.

6.6.3 Reading from Modbus TCP

For our sensor data, we are going to define a set of types and configure a mapping of our sensor to register address data. For our measurements, we will measure the current, power factor and time the start and end of our readings. These timings will help us map power readings to the function execution.

```
#[derive(Serialize, Deserialize)]
pub struct SensorData {
  pub power: f32,
  pub start_read: u128,
  pub end_read: u128,
}

pub enum LineInEnergySensor {
  Current = 0x406,
  PowerFactor = 0x40a,
}
```

With this in place, and we can define our function for reading from modbus. The goal of this function is to run in a parallell thread while we conduct our stress-testing of Nebula on our Raspberry Pi. This function will be run continually until the parent thread breaks, so we will read measurements from Gude, described in code:

```
use serde_derive::{Deserialize, Serialize};
use tokio_modbus::prelude::*;

pub async fn read_modbus_data(
  addr: &str,
) -> Result<SensorData, anyhow::Error> {
  // 1.
  let mut client = tcp::connect("192.168.68.70:502".parse()?).await?;
  let start_read = current_micros()?;

  // 2.
  let current_register = client
      .read_input_registers(LineInEnergySensor::Current as u16, 2)
      .await?;
```

```
  let raw_current =
    (i32::from(current_register[0]) << 16) | i32::from(current_register[1]);
  let current = (raw_current as f32) / 1000.0;
  // 3.
  let power_factor_register = client
      .read_input_registers(LineInEnergySensor::PowerFactor as u16, 2)
      .await?;
  let raw_power_factor =
      (i32::from(power_factor_register[0]) << 16) |
      i32::from(power_factor_register[1]);
  let power_factor = (raw_power_factor as f32) / 1000.0;

  // 4.
  let power = 240.0 * current * power_factor;
  let end_read = current_micros()?;
  Ok(SensorData { power, start_read, end_rad })
}
```

The steps we go through here are:

1. Define our tcp client and connect it to our Gude device on the modbus tcp port (e.g., 192.168.68.71:502), and create an instance of our SensorData struct, where we set values to 0, and time the start of our measurement in microseconds since epoch.
2. Read current from its register, cast it as a floating point number and divide by 1000, so we get the value in A, instead of mA.
3. Read power factor from the register, which is multiplied by 1000 in the measurements, so we cast it to f32 and divide it by 1000 to get the power factor.
4. Calculate power at the time of measurement to $240V \times I \times PF$ and return the power product and start and end times for the measurement.

6.6.4 Attaching power measurement to function invocations

When we benchmark Nebula on our Raspberry Pi, we spawn two threads that run in paralell, using the Tokio async runtime. One of these threads performs out stress-test function from **??**, while the other reads from our read_modbus_data function from **??**. We continue to read sensor data until the stress-test function is completed, and save both lists as a variable. Then we pass both the resulting function invocation results and the power readings to a function that finds power measurements that coincide with a function invocation.

?? below illustrates how power measurements are paired with coinciding function invocations to estimate the power required to run our functions.

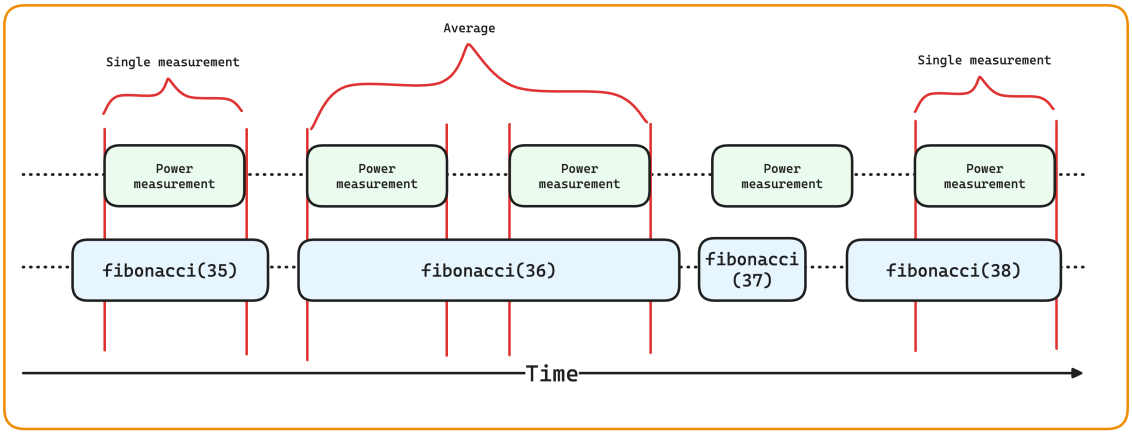

Figure 6.4: Power measurements that overlap with function invocations

In some of our benchmarks, there were instances of function invocations being too fast to be able to pair with an power measurement. In these cases, we ran the functions with its input multiple times, to ensure that every input value for each function has the same amount of invocations to improve the validity of our data. In other cases, we saw functions that were so slow that multiple power measurements were performed over the functions lifetime, and in this case we sum all the corresponding measurements and get the average power consumption. For example, a slow recursive fibonacci function might have 30 power readings associated with it.

The code below shows how this is done. It loops through each function result, finds the energy measurements that begin and end within the start and end of each function invocation. Then it updates each function result with the average power that was consumed by the system during the functions execution. We also calculate the estimated *Wh* in energy consumption with the formula:

$$P \times \frac{t}{36000000000}$$

```
pub fn associate_power_measurements(
  mut function_results: Vec<FunctionResult>,
  sensor_data: &Vec<SensorData>,
) -> Vec<FunctionResult> {
  // 1.
  let mut processed_results = Vec::new();
  let microseconds_to_hour = 1000.0 * 1000.0 * 60.0 * 60.0;
```

```
  // 2.
  for mut function_result in function_results {
    // 3.
    let function_start_time =
      function_result.metrics.start_since_epoch;
    let function_end_time = function_result.metrics.end_since_epoch;
    let mut total_power = 0.0;
    let mut num_readings = 0;
    // 4.
    for data in sensor_data {
      if data.start_read >= function_start_time
        && data.end_read < function_end_time
      {
        total_power += data.power;
        num_readings += 1;
      }
    }
    // 5.
    if num_readings > 0 {
      let avg_power = total_power / num_readings as f32;
      let duration = function_end_time - function_start_time;
      let energy_consumption_wh =
        (avg_power * duration as f32) / microseconds_to_hour;

      function_result.metrics.average_power = avg_power;
      function_result.metrics.energy_consumption_wh =
        energy_consumption_wh;
      processed_results.push(function_result);
    }
  }

  processed_results
}
```

With this in place, we can now perform our experiments on both our virtual machine and on the Raspberry Pi. **??** below shows the physical setup for the experiments on Nebula on our Raspberry Pi.

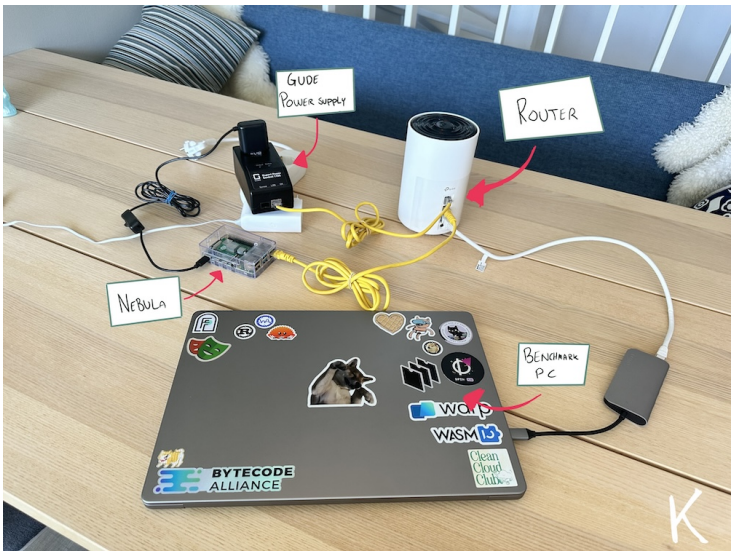

Figure 6.5: The physical setup, including the Raspberry Pi connected to power through our Gude power supply. The Raspberry Pi and Gude controller is connected to the benchmark computer on a local network through ethernet.

6.6.5 Energy measurement challenges

In **??**, we saw several protocols and measurement devices, and how they work, described. The first iteration of measuring power was inspired by **qianEnergyFootprintingWordPress2022<empty citation>**, and consisted of an Aotec Smart Switch 7, Aotec Z-stick 7 and relied on the zwave-js-ui application to act as a publisher for reporting energy measurements over MQTT. This setup had one major flaw; energy reporting through MQTT could only be reported once every 30 seconds at a minimum. This turned out to be too slow for attempting to measure accurate readings for our experiments.

This limitation was unexpected, but Qian had used an Aotec Smart Switch 6, the previous generation, which turned out to support reporting once every second. This device was luckily found on Finn.no, and allowed us to upgrade the accuracy own to measuring every second instead.

However, as we will present in **??**, our function executions on our Raspberry Pi can come down to single-digit milliseconds, meaning that once every second is still not accurate enough to pinpoint the current draw on our system for each function invocation.

The Gude Expert Power Control 1105 turned out to be the most suited device for our experiments. This device supports many protocols, one of which is the Modbus TCP protocol. By switching to this power supply, connecting our Raspberry Pi and benchmarking computer to the same network over ethernet, and changing

the communcation method from MQTT to Modbus TCP, we were able to perform energy readings in microseconds.

6.7 Testing - Validity and Rigidity

While formal unit testing was not exercised during the development of Nebula, the choice of the Rust programming language provided inherent benefits in terms of code reliability and correctness, ensuring the validity and rigidity of the prototype.

6.7.1 Rust's Compiler Guarantees

As outlined in **??**, chosing Rust as the programming language was driven by its strong relationship with **Wasm!** and its suitability for building efficient and scalable applications. Significant factors to this are Rust's strict compiler checks and emphasis on memory safety, concurrency, and parallelism. These factors help

The Rust compiler enforces a set of rules that prevent common programming errors, such as null pointer dereferences, data races, and memory leaks. By catching these issues at compile-time, Rust eliminates entire classes of runtime errors, increasing the overall stability and predictability of the codebase.

Furthermore, Rust's ownership model and borrowing rules ensure safe and efficient memory management, without the need for manual intervention or a garbage collector. This approach reduces the risk of memory-related bugs, which are very difficult to detect and resolve. We can't guarantee that there are not any present in Nebula, as testing will never be able to guarantee the absence of bugs. However, we saw that during extensive stress-testing, our prototype consistently for several hours straight during our benchmarks.

One exception to these guarantees is the deserialization of our **Wasm!** modules, as discussed in **??**. In this case, we opt out of Rust's compiler guarantees by telling the compiler that we have a better understanding of the situation and can safely read our **Wasm!** binary files as bytecode. While this represents a potential risk if malicious binaries were to be introduced to the web server, it did not pose an issue for our research.

6.7.2 Rigorous Testing and Benchmarking

Although formal unit testing was not a primary focus, the code underwent rigorous manual testing and code review processes. As detailed in **??**, the components of

Nebula were thoroughly tested with various input scenarios, edge cases, and stress tests to ensure correctness and robustness.

Moreover, the benchmarking suite itself, described in **??**, served as a form of integration testing, exercising the entire system end-to-end and validating the correctness of the results.

The combination of Rust's language features, idiomatic coding practices, and extensive manual testing through the benchmarking suite provided a strong foundation for the validity and rigidity of the Nebula prototype, ensuring reliable and robust performance during the experiments and evaluation.

Part III

Discussion

Chapter 7

Results

Chapter 8

Analysis

8.1 Related work

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

qianEnergyFootprintingWordPress2022<empty citation> 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.

sebrechtsAdaptingKubernetesControllers2022<empty citation> 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 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 **??** for their findings on memory usage with **Wasm!**-based operators to Rust and Golang operators.

[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

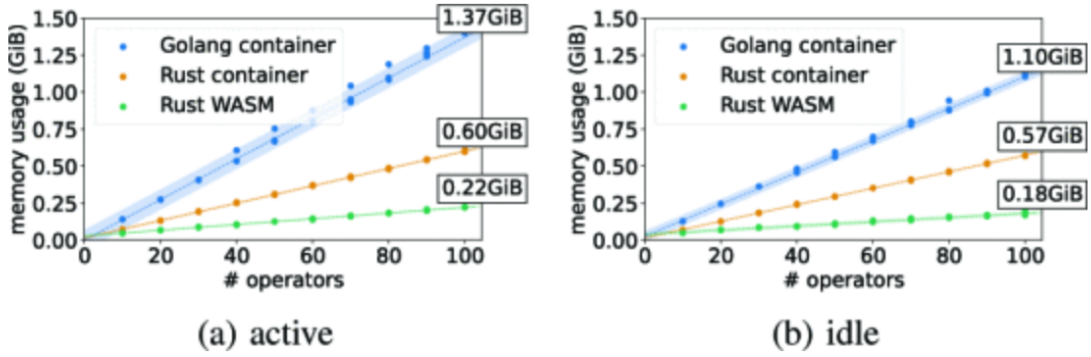

Figure 8.1: Different Kubernetes operators memory usage. Comparing Rust to WASM vs Rust container vs Golang container. © 2022 IEEE

shillakerFaasmLightweightIsolation2020a<empty citation> 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 **??** below for their findings while comparing their Faasm to Knative.

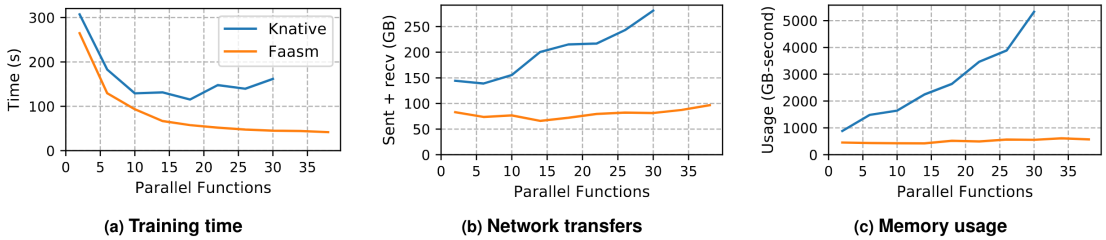

Figure 8.2: Different Kubernetes operators memory usage. Comparing Rust to WASM vs Rust container vs Golang container. © 2022 IEEE

Furthermore, **shillakerFaasmLightweightIsolation2020a<empty citation>** 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 **??** 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

Future work

> *"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*

Chapter 10

Conclusion

Chapter 11

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