Academemes: Leveraging Rust and WebAssembly to create an energy-efficient web service for serving Academic Memes

> *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*

Introduction

The rapid growth of cloud computing the past decade has led to the cloud consuming enormous amounts of energy. The entire ICT (Information- and Communication Technology) industry emits 2.1% to 3.9% of global green gas emission. Data centers, which are at the core of cloud computing, consume a significant amount of energy, estimated at around 200 TWh/yr or 1% of the worlds electricity (Freitag et al. 2021). This energy consumption could grow to between 15-30% of electricity consumption in some countries by 2030. Allthough data centers strive to reach a net zero sum carbon footprint, there are still a lot of electricity genereated by fossil fuels, a leading contributer to climate change (Mytton, D. 2021).

Important to note is that these measurements and estimates comes with some level of uncertainty, but give us a rough idea of the current and future situation. As demand for cloud services continues to rise, there is a pressing need to explore alternative methods that can help improve energy efficiency, while maintaining the performance, availability and scalability of these services.

This essay investigates the potential for utilizing technologies like Rust and WebAssembly to develop a prototype Function-as-a-Service (FaaS) platform. This prototype will aim to address the energy efficiency problem in cloud computing by serving academic memes through a web service.

Hypothesis

This master thesis explores the following hyphothesis:

> *It is possible to develop a Pure FaaS platform that scale to near-zero resource usage, using WebAssembly modules, without sacrificing availability.*

To test this, we will develop a prototype Pure Functions as a Service written in Rust, and attempt to scale to near-zero by implementing a simple service for serving memes on the internet that is able to scale down to near-zero and start up at fastly.

Furthermore, we hypothesise that the service will remain consistent, be virtually always available, and suffer no issues from partition tolerance (Consistency, Availability, Partition Tolerance - CAP) (Brewer, Fox. 1999) while also being more energy efficient than predominant alternatives.

Background

Cloud Computing: An Overview

Cloud computing, more commonly known as *the cloud*, refers to the delivery of computing services, such as storage, processing power, network, and software, served over the internet, instead of running on locally owned hardware (on-premise). For companies, this has proved to be a super power, where businesses can

focus on deploying their own applications and services to their users without worrying about the underlying infrastructure.

Some benefits include:

- Reduced total cost of ownership: Cloud computing has enabled companies to take their computing needs to the next level. Startups who can't afford neither the cost or time required to build their own infrastructure, and larger companies that want to iterate faster and decrease their lead time from idea to production (Thomas, Dave. 2009).
- Scalability is one of the most significant benefits of cloud computing. As organizations expand, so do their customer needs and the complexity of their application infrastructure. Each additional feature brings with it additional costs, highlighting the importance of efficient resource management to optimize hardware investments in a cloud computing environment (Thomas, Dave. 2008).

Some challenges include:

- Cost: Managing the cost of cloud computing is an ongoing challenge, with Gartner predicting that through 2024, 60% of infrastructure and operations (I&O) leaders will encounter cloud costs that are higher than budgeted for (Rimol, M. 2021).
- Energy usage: The challenge of energy usage in cloud computing is a significant concern, with data centers alone accounting for approximately 1% of the world's electricity consumption (Freitag et al. 2021). This staggering statistic highlights the need to explore strategies and measures to mitigate energy usage in data centers. By reducing energy consumption, not only can costs be reduced, but also the carbon footprint associated with running the cloud can be lessened. Decreasing energy usage in data centers is expected to yield cost savings and contribute to the overall sustainability goals by reducing the cloud's environmental impact.

Traditional Deployment Methods: Virtual machines and Containers

In the era preceding cloud computing, companies bought, set up and managed their own infrastructure. This necessitated having in-house infrastructure engineers to maintain on-premise data centers or servers, leading to significant cost. Sensing the potential in offering managed infrastructure, Amazon launched its subsidiary, Amazon Web Services, during the mid-2000s.

The launch of AWS's Amazon S3 cloud service in March 2006, followed by Elastic Compute Cloud (EC2) in August the same year (Barr, J. 2006), marked a major turning point in application development and deployment, and popularized cloud computing. EC2, as an Infrastructure-as-a-Service platform, empowered developers to run virtual machines remotely. While similar services existed before 2006, Amazon's large customer base helped them gain significant traction, effectively bringing cloud computing to the mainstream.

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 (Sharma, et al. 2016).

In contrast to VMs, which require installation of resource-intensive operating systems and minutes to start up, containers along with their required OS components, could start up in seconds. Typically managed by orchestration tools like Kubernetes, 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.

Serverless and Function-as-a-Service (FaaS)

Building your own developer platform on top of Kubernetes, much like building your own infrastructure, also entails a significant cost. Often, developers wish to launch specialized smaller services, without having to grapple with complicated orchestration. This led to the emergence of the Serverless model. Despite its somewhat misleading name, serverless doesn't imply the absence of a server. Instead, it means that the responsibility of server management has shifted from the developer to a third party provider.

From the advancements of serverless, we get its subset, Functions-as-a-Service, or FaaS. Companies already in the cloud game decided to develop their own FaaS platforms to attract developers interested in just writing their functions and running them, and not worry about anything underneath.

Some major vendors in Serverless

The concept of "the cloud" isn't owned by any single organization, but rather, through the collective effort of industry players including Amazon, Microsoft, Google, Alibaba and DigitalOcean, among others. This essay delves into some challenges faced by the biggest three vendors: Amazon, Google and Microsoft.

Amazon Web Services (AWS) provides AWS Lambdas, a technology that hinges on their proprietary Firecracker - a streamlined virtualization technology for executing functions. Interestingly, for this thesis, is that Amazon's Prime Video streaming service transitioned recently from a serverless architecture to a monolithic system to meet specific service demands. One might question whether this reflects the suitability of serverless systems for cloud computing, or for specific use cases like theirs (Kolny, M. 2023. Accessed 29.05.23). Some discussions suggest that their need to process videos frame by frame led to astronomical costs on their sibling company's FaaS, Amazon Lambda.

Google provides Google Cloud Functions, which allow developers to write and execute functions in languages such as Node.js, Python, Go and execute them in response to events. Google's approach to function execution centeres around container technology (Wayner, P. 2018. Accessed 29.05.23).

Microsoft's Azure Functions is a Faas platform that enables developers to create and execute functions written in languages like C#, JavaScript, Python. Similar to Google, they also harness the power of containers to execute these functions.

WebAssembly: A new paradigm?

WebAssembly (Wasm) is a binary instruction format designed as a stack-based virtual machine. It aims to be a portable target for the compilation of high-level languages like Rust, C++, Go and many others, enabling deployment on the web for client and server applications. Originally designed and developed to complement JavaScript in the browser, it now expands its scope to server-side applications, thanks to projects like WebAssembly System Interface (WASI), which provides a standarized interface for WebAssembly modules to interface with a system.

WebAssembly's design provides advantages over traditional deployments methods in the context of cloud native applications:

Efficiency and Speed: Wasm was designed to be fast, enabling near-native performance. Its binary format is compact and designed for quick decoding, contributing to quicker startup times, an important aspect for server-side applications. The performance gains could lead to less CPU usage, thereby improving energy efficiency.

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

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

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

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

Wasm+WASI: Towards Energy-efficient FaaS Platforms

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

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

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

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

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

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

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

The CAP Theorem and Its Relevance to This Thesis

As mentioned in the hypothesis, the CAP theorem was a theorem proposed by Eric Brewer in 1999 that states it is impossible for a distributed data store to simultaneously provide all three of the following guarantees: *Consistency*, *Availability*, and *Partition Tolerance* (CAP)(Brewer, Fox. 1999). Most distributed systems can only guarantee two out of these three properties at any given time.

In the context of this thesis, the hypothesis is that it is possible to develop a FaaS platform that scales to near-zero resource usage without suffering from the constraints of the CAP theorem. This implies that the proposed system would maintain high availability and consistency even while scaling down to minimal resource usage, without sacrificing partition tolerance.

Developing a system that achieves this balance would involve navigating the inherent trade-offs described by the CAP theorem. However, the unique characteristics of a FaaS platform and the benefits of using Rust and WebAssembly may provide opportunities for novel approaches to this challenge.

By focusing on executing pure functions without side effects (an approach often associated with functional programming), the proposed FaaS platform could minimize dependencies between operations, reducing the potential for conflicts and easing the enforcement of consistency. Meanwhile, the efficient startup times and lower runtime costs enabled by Rust and WebAssembly could allow the system to quickly scale up resources when needed, contributing to high availability even at times of peak demand.

Further exploration of these ideas, along with rigorous testing and validation, will be key components of this research project.

Methodology and preliminary ideas

To explore the central hypothesis of this thesis and gather pertinent data, we will follow an experimental research design that combines the development of a FaaS platform with comprehensive testing and data analysis.

Our focus will be on creating a prototype *PFaaS* named "Nebula" , developed using Rust and compiled to Wasm and Wasi that operates on the Wasmtime runtime. This platform will serve purely WebAssembly modules, narrowing down the execution to pure functions compiled to Wasm.

Spin, an open-source offering by Fermyon, will be a critical point of reference. It facilitates the development and deployment of applications, compiled to Wasm, on either Fermyon's own cloud, or any self-hosted variant. While Spin does manage side effects, our prototype will focus on pure functions.

Post-development, "Nebula" will be subject to benchmarking tests using a web application "Academeme.com", serving academic memes. These tests will simulate users accessing the website and measure the system's performance under varying loads, capturing data on startup time, execution time, and hopefully energy consumption. If "Purfity" garners interest and usage from other students, we may incorporate data from "real-life" use cases.

We will gather energy consumption measurements through a device connected to the motherboard of a testing device in UiO's "Energy Labs". This device will gather readings from specific parts of the CPU, transmitting this data through the Z-wave protocol to a USB device for integration into our benchmarking tools.

To contextualize our findings, we'll compare the power consumption to locally-installable FaaS platforms, such as serverless.com, or potentially replicated platforms built atop Docker and Kubernetes. This comparison could require an expansion of the research scope, including understanding and building a simple FaaS using these technologies.

Following this, we'll undertake data analysis, interpreting the information gathered during benchmarking and energy measurement to explore the relationship between load, performance, and energy consumption. The ultimate goal is to gauge the impact of utilizing WebAssembly on the power efficiency of FaaS Platforms.

A wealth of resources will be leveraged to attempt to measure accurate energy consumption. Luís Cruz's article provides a comprehensive guide on measuring energy on computers (Cruz, L. 2021), recommending tools such as Intel Power Gadget, built on top of Intel RAPL (Running Average Power Limit), a tool that was used in the study "RAPL in Action: Experiences in using RAPL for Power Measurements" (Khan, et al. 2018). This study provides foundational knowledge about the correlation between core frequency, load and power consumption for this thesis.

Powermetrics, a built-in tool on Macbooks, will be used during development to give a rough measure of power consumption during development on a Macbook running an M2 chip, which doesn't support the Intel tools mentioned above.

The formula for calculating energy consumption, as Cruz notes in his article, is $E = P \cdot \Delta t$, where E denotes energy consumption, P signifies average power in watts, and Δt represents the given sample in seconds. This calculation will be central to our power measurements.

By following this methodology and making use of these tools and technologies, the aim is to generate a practical analysis of the potential for WebAssembly to enhance the power efficiency of cloud computing platforms.

Design Specification

The design and development of the experiments for this thesis revolve around the creation and testing of "Nebula", a Pure Function as a Service platform. The experiments are primarily focused on testing the power efficiency of this platform, its responsiveness, and scalability under varying loads. Here, we detail the key aspects of these experiments:

Development Environment

The core development language for Nebula will be Rust, a language known for its memory safety and high performance. Its unique ownership semantics allow fine-grained control over memory management, which aligns with our pursuit of energy efficiency.

Compilation Target

The functions developed in Rust will be compiled to WebAssembly (Wasm) with the WASI target. WebAssembly is a binary instruction format designed as a portable target for the compilation of high-level languages. WASI, or WebAssembly System Interface, is a system interface for the WebAssembly platform. This combination allows us to run sandboxed service functions anywhere, making our platform both secure and widely compatible. The

Database Integration

In the Nebula system, the concept of pure functions is central but accommodating a database is critical for real-world application functionality, such as storing and fetching memes. To reconcile this, Nebula's design will incorporate a separate layer for interacting with the database. This layer retrieves necessary data, passes it to the pure functions for processing, and then handles subsequent operations like updating the database or passing results to other services.

Frontend Development

For the user interface, there are two potential directions. One approach is to use Rust for server-rendered pages, maintaining a uniformity in the tech stack. This approach could use a Rust framework like Yew.

The alternative approach is to develop a separate client-side application using Svelte or Elm, which will interface with our pure functions via HTTP. Svelte, being a JavaScript framework, offers the advantage of a more dynamic user interface. On the other hand, Elm, a purely functional language for frontend development, aligns more with our platform's philosophy of pure functions.

The above specifications will serve as a guide throughout the development of Nebula. In the next phase, we'll begin the implementation, continuously testing and adjusting these specifications as needed.

References