Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Flask-Faasm: Deploying Serverless Web APIs on Faasm

Author:
Hyunhoi Koo

Supervisor:
Prof. Peter Pietzuch
Mr. Carlos Segarra

Second Marker:
Dr. Marios Kogias

June 19, 2023

Abstract

The rise of the serverless paradigm and its Function-as-a-Service (FaaS) realisation has proven to revolutionise the deployment of cloud applications by providing the promise of infinite scalability without the operation overhead. The rise in popularity of FaaS platforms have naturally raised the question on whether it is viable to deploy a web API onto such platforms, and while it does have benefits with a reduced development and operational cost, it unfortunately shows limitations due to the prohibitively expensive latency overhead associated with the use of hardware-based isolation mechanisms like containers. To solve this limitation, alternative isolation mechanisms are proposed, such as *Faaslets* that use WebAssembly runtime and other components to provide a more lightweight isolation mechanism, along with `Faasm`, a serverless runtime that utilises *Faaslets* instead of container for isolating the function invocation instances.

This research presents *Flask-Faasm*, a system that provides a seamless zero-cost interoperability solution to deploy Flask web applications on `Faasm`. This research also demonstrates that the additional overhead associated with *Flask-Faasm* is about 1.5 times higher than a 'native' deployment of the application, which is much lower than existing serverless solutions with orders of magnitude increase in latency. This research also shows that the majority of the latency overhead of *Flask-Faasm* involves loading the Python function script, and offer potential improvements to further reduce the additional overhead, thus demonstrating that deploying web APIs on serverless solutions is in fact viable.

Acknowledgements

I would like to first express my warmest thanks to my supervisors, Prof. Peter Pietzuch and Mr. Carlos Segarra, for providing me with their support and guidance throughout the project, whether it is when suggesting me with potential initial steps to take the project in, providing me with guidance when there are difficult choices to be made, or even technical issues and difficulties with the system and `Faasm`.

I would also like to thank my second marker, Dr. Marios Kogias, for providing me with advice and guidance during the Project Review. A discussion with him about the implementation details of *Flask-Faasm* and potential evaluation strategies have provided me with valuable insight to further advance my research.

I would finally like to thank my friends and family for providing me with the emotional support and much-needed fun and laughter throughout my journey completing this research and my degree.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

1.1 Motivation

The evolution of applications on the cloud can be characterised by a transition over time from monolithic to microservice architectures, and subsequently to the serverless paradigm. Microservice architecture emerged as a solution to the problems with traditional monolithic architecture–known to show difficulties in cost-effective scaling with non-uniform request load [1]–by promoting applications as a collection of independent, individually scalable services [7]. However, it also incurs a significant increase in development and operational overhead due the necessity for a manual management of complex cloud infrastructures with the additional components for deploying each individual microservice. Serverless architecture offers a solution by enabling developers to upload services as individual functions, offloading the responsibility of server maintenance, scaling, and load balancing to the Cloud Service Providers and enabling developers to focus on the functionality of applications [2]. The most common realisation of serverless architecture is the Function-as-a-Service (FaaS) model, which enables developers to upload functions with automatic resource scaling that matches the incoming function invocation rates [8], and it has already been adopted by several leading cloud platforms like AWS Lambda [9] and Azure Functions [10].

Although deploying a web-based API (Application Programming Interface) on FaaS platforms do have some promising benefits, such as the promise of 'infinite' scalability without the development and maintenance overhead [3] and the pay-per-use model offering potential cost savings [1], FaaS has several limitations that significantly impact the practicality of deploying web APIs. As these systems use container images for deploying functions, it exhibits a significant latency overhead, including 'cold-start' latency, a delay caused by the first initialisation of function containers [3]. This latency is especially problematic for web APIs that are ephemeral in nature, where the latency might be orders of magnitude higher than the actual runtime of the function [3].

To make FaaS systems more viable for deploying web APIs, alternative isolation mechanisms have been proposed that use software-based isolation mechanisms to reduce the overhead associated with traditional hardware-based isolation mechanisms like containers and Virtual Machines (VMs). One promising isolation mechanism is *Faaslets*, a novel isolation mechanism that provides resource isolation through the use of WebAssembly runtime, Software-Fault Isolation, and standard Linux `cgroups`, along with `Faasm`, a serverless runtime that uses *Faaslets* to execute functions in an isolated environment [5].

This research explores whether deploying a web API developed using standard web frameworks on alternative serverless FaaS platforms like `Faasm` is viable, both as an alternative solution to the problems with existing FaaS platforms, and in providing an easy-to-use experience for developers by providing a zero-cost transition with web applications developed using standard frameworks like Flask [11].

1.2 Objectives

The objective of this research is to demonstrate that deploying a web API on serverless systems is in fact viable, if the FaaS platform uses alternative resource isolation mechanisms that avoid the use of hardware-based isolation mechanisms like containers or VMs.

To demonstrate this objective, a novel product needs to be developed, which requires an alternative FaaS platform, along with an adapter layer that provides a transparent zero-cost support for deploying the web applications on the platform. `Faasm` is chosen as the FaaS platform to implement this product, and *Flask-Faasm* is introduced that provides the features of the aforementioned adapter layer.

The product also needs to be evaluated to verify whether *Flask-Faasm* does provide a significantly reduced latency overhead compared to a 'native' deployment of the web API.

1.3 Contributions

The contributions of this research are summarised as follows:

1. We introduce *Flask-Faasm*, which augments the execution of Flask web applications by providing an adapter layer to `Faasm`, where the incoming requests are forwarded to be executed on `Faasm`, providing a transparent zero-cost experience both to the developers by not requiring any modifications to their web application to deploy their app to `Faasm`, and to the end-user by maintaining the existing Flask application to serve as a gateway server for forwarding the requests to `Faasm`.

   The design requirements and limitations of *Flask-Faasm* are defined in Chapter 4, and the implementation details of *Flask-Faasm* are explained in Chapter 5.

2. We evaluate the performance characteristics of *Flask-Faasm* in Chapter 6 by performing experiments on a benchmark web API adapted from a standard benchmark suite: Python Performance Benchmark Suite [12]. The benchmark web API is deployed both on *Flask-Faasm* and a 'native' deployment to discuss the latency overhead introduced by *Flask-Faasm* and its magnitude (§6.2), and a detailed breakdown of the latency for the different benchmark functions to analyse the most significant source of overhead introduced by *Flask-Faasm* (§6.3).

3. *Flask-Faasm* is released as a fully open-source product. The source code for *Flask-Faasm* is available at https://github.com/hhoikoo00/flask-faasm.

Chapter 2

Background

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

This section explains how serverless architecture emerged from the issues present in a monolithic and user-managed microservice architecture, along with a realisation of the serverless architecture: Function-as-a-Service (FaaS) model.

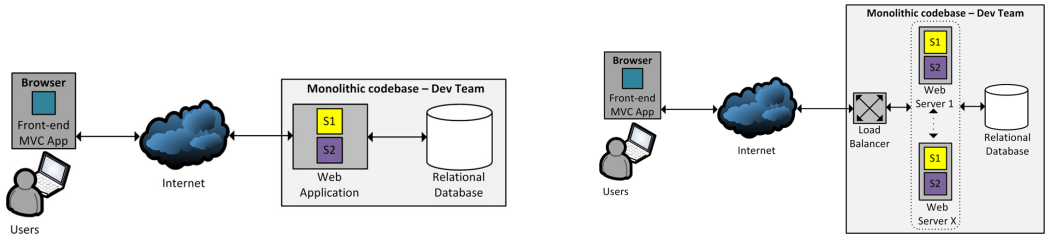
2.1.1 From Monolithic to Serverless

Traditionally, applications–especially enterprise applications–on the cloud were deployed with an architectural style known as Monolithic Architecture. Fowler and Lewis [7] defines Monolithic Architecture as a structure of applications where a single codebase exposes tens to hundreds of various services to its clients and external systems, and multiple developers or teams work on the same codebase, making changes and upgrades that can potentially propagate through the whole set of services. Monolithic applications are usually deployed on the cloud using Infrastructure-as-a-Service (IaaS) [13] solutions such as AWS EC2 [14], and are deployed on a single server, or potentially on multiple servers with a load balancer, where each server hosts a copy of the entire codebase [1]. Figure 2.1 shows an example application structured using a monolithic architecture, with a browser frontend, a backend web application that exposes two RESTful services (§2.2) S1 and S2 to its clients, and a relational database for permanent storage.

Monolithic applications are known to struggle when it is supplied with a non-uniform load across different services deployed on the application [1]. This is because the only way to scale out the application on a multi-server environment is to deploy more duplicates of the whole codebase, which drastically reduces the cost-effectiveness of the application, as additional infrastructure provisioned for the new copies of the codebase is mostly wasted by the execution of unused services. Figure 2.1b shows that, for the backend application to scale out, new web servers must be spun up that host the entire copy of the application for both S1 and S2, despite often only requiring infrastructure for one of the services. Monolithic applications are also notoriously complex to develop and deploy, as changes made for one service can have a ripple effect across the whole codebase that may unintentionally change the behaviour of other services, or break them entirely [7].

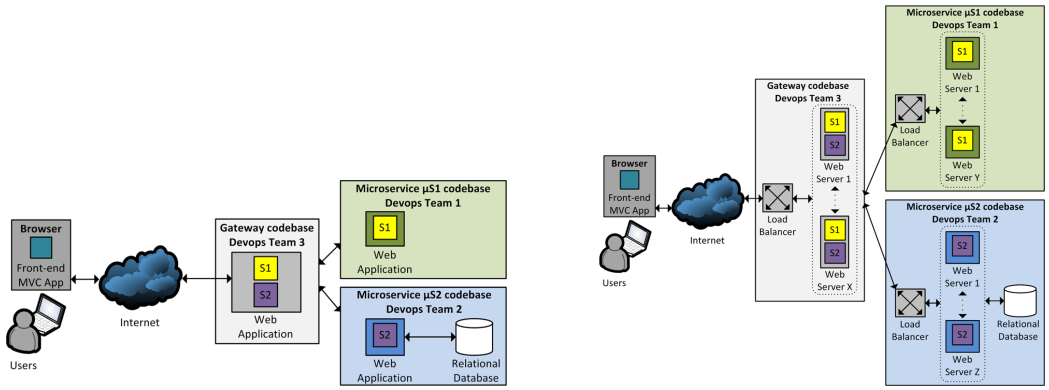
To address the issues with monolithic applications, Microservice Architectural style was invented. Microservice architecture aims to solve these issues by deploying applications as a collection of independent services that are individually deployable and scalable. It also imposes a strict boundary between the different services, allowing them to be developed and managed by independent teams, even using different tech stacks appropriate to the specific service being developed [7].

Microservice applications allow each service to scale independently, which allows the underlying infrastructure to be better matched specifically to each service being deployed.



(a) Architecture (b) Deployment

Figure 2.1: A Web Application Using A Monolithic Architecture [1]



(a) Architecture (b) Deployment

Figure 2.2: A Web Application Using A Microservice Architecture [1]

In an ideal scenario, this flexibility causes the infrastructure cost to diminish eventually [1]. Figure 2.2 shows the same application in Figure 2.1 but created using a microservice architecture. Unlike a monolithic application, each REST web service is deployed on separate microservice codebases, with its own relational database (as necessary) and a separate gateway codebase that exposes the API endpoints to the frontend client. Each microservice–including the gateway–can be scaled independently with its own dedicated load balancer, which can handle the load to the specific services without affecting how the other services are scaled. Moreover, as each team manages its own independent microservice codebase, developing each service is much easier as individual services are simpler in complexity and the deployment of a microservice does not affect the status of other microservices, meaning that routine maintenance and upgrade is easier than in a monolithic application, where the entire application must be taken down for maintenance and upgrade to be performed on a subset of services [1].

However, managing such architecture manually on user-managed cloud servers on an IaaS platform requires additional effort from the developers to deploy each microservice to the Cloud Service Providers (CSP), along with managing cloud infrastructures for scaling and operating [1]. Comparing Figure 2.1b with Figure 2.2b shows that microservice applications have additional components like the gateway codebase and dedicated load balancers for the individual microservice codebases, which require additional setup and maintenance to ensure that it works properly. This additional effort requires additional development time from the developers and operations engineers, which ultimately translates to additional maintenance costs.

Due to these concerns, Serverless architecture has emerged that enables developers to deploy a microservice application without needing to manage the servers themselves [1]. The architecture aims to offload the responsibility of server operations, scaling, and load balancing to the Cloud Service Providers by pooling all resources–including hardware, OS,

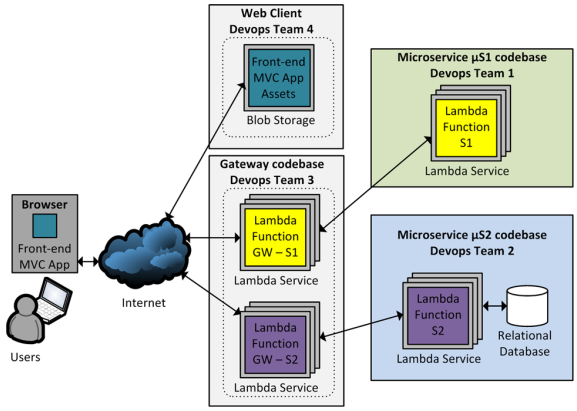


Figure 2.3: A Web Application Deployed Using a Serverless Architecture [1]

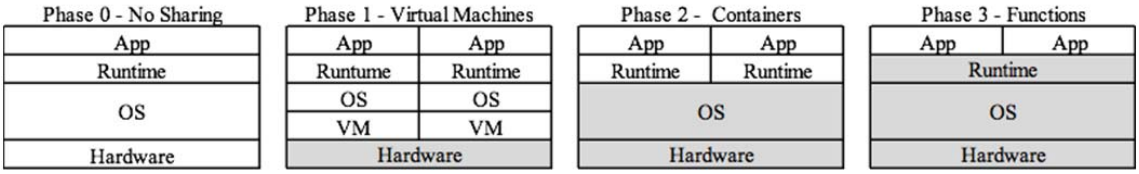


Figure 2.4: Phases of Virtualization (Grey layers are shared) [2]

and runtime environments–that are automatically provisioned at a platform level to user-defined functions that can infinitely scale as the demand increases. Lynn [2] describes serverless architecture as "a software architecture where an application is decomposed into 'triggers' (events) and 'actions' (functions), and [...] provides a seamless hosting and execution environment." Users provide the CSPs (relatively) lightweight, single-purpose functions, which are automatically executed and scaled on-demand. By allowing the CSPs to operate the cloud infrastructure and scaling mechanisms, the maintenance costs are lifted from the developers who can now focus on the application's functionality which can now be ensured to operate and scale seamlessly without manual oversight [2]. It also enables new pricing mechanisms, where the developers are charged only for the functions' runtime, meaning that the developer does not have to pay for unused resources, unlike in other architectures where the user pays for the underlying servers' uptime [3].

Figure 2.3 shows how the same app can be deployed in a serverless platform (AWS Lambda [9] chosen as an example platform). The main difference compared to the (user-managed) microservice architecture is that the REST web service and gateway codebases are developed and deployed using functions, and automatic scaling and load balancing are handled by the Cloud Service Providers and do not have to be managed by the developers [1]. (Note that the static assets for the web frontend are now deployed on a dedicated permanent cloud storage solution.)

2.1.2 Serverless and Phases of Virtualization

The move to microservices and serverless in the cloud has an interesting parallel with increasing levels of virtualization and resource sharing. Lynn et al. [2] distinguish the phases of virtualization as follows: Virtual Machines, Containerization, and Serverless. Figure 2.4 illustrates the differences between the different virtualization phases.

A Virtual Machine (VM) is a software-based emulation of a physical computer and runs a full copy of an operating system [2]. Virtualization using VMs enables sharing of common hardware (as seen in Figure 2.4), which is beneficial for applications on the cloud because deploying each application on its own physical hardware gets very costly

due to the high costs of buying and maintaining a large number of machines and the fact that the hardware is often underutilized [15]. With virtualization, servers can simultaneously host multiple services and applications on the same physical hardware, significantly reducing maintenance costs and operations overhead, thus providing Infrastructure-as-a-Service [15]. Unfortunately, whether hosting a monolithic or microservice application, managing resources at the VM level is still too heavyweight for fast provisioning and scaling of resources. This results in either over-provisioning of resources, which leads to an increase in hosting costs, or under-provisioning, which leads to poor performance [16].

Containers emerged as a potential solution to a lighter-weight virtualization mechanism. Containers are essentially a "server-oriented repackaging of Unix-style processes with additional namespace virtualization." [15] It virtualizes at an OS level, with limited access to physical hardware. It does not need to emulate an entire OS and instead relies on a minimal subset interface to the host machine's OS to perform system calls [2]. Therefore, it is much less resource (CPU, memory, etc.) intensive than VMs, and new services can be provisioned at a more granular level, enabling more rapid and elastic scaling of server infrastructure. Cloud service providers can load-balance many more containers across servers to maximize the utilization of idle servers, as the memory footprint of containers is much less than that of Virtual Machines [16]. Linux containers especially are useful for implementing lightweight containers, as Linux kernel supports *cgroups* (or Control Groups). *cgroups* allow system resources, such as CPU and system memory, to be partitioned and isolated among multiple groups of processes, enabling the kernel to limit the amount of resources a particular process group can consume [17]. Each container runs on its own *cgroup*, which allows multiple containers to run on the same host machine without interfering with each other's resources, and also supports features like container pausing [18].

Serverless architecture also emerges here as a natural extension to the level of virtualization. Serverless systems can be seen as an additional layer of virtualization, where the language runtime is also abstracted away from the developer, and developers supply individual functions to the serverless platform. The Cloud Service Provider takes full responsibility for the provisioning of the services, scaling, and load balancing of the individual functions. Serverless internally uses OS containers like Docker [19] to deploy and scale the functions [20], and Virtual Machines to further isolate the containers to enforce isolation guarantees necessary for a multi-tenanted cloud environment [3].

2.1.3 The Rise of the Function-as-a-Service (FaaS) Model

There are various possible realizations of the serverless architecture. Cloud Native Computing Foundation (CNCF) [21] outlines two: Function-as-a-Service (FaaS) and Backend-as-a-Service. CNCF defines BaaS as "third-party API-based services that replace core subsets of functionality in an application [...] those APIs are provided as a service that auto-scales and operates transparently." Examples of BaaS include storage platforms and vendor-provided multi-tenanted, auto-scaling services, such as S3 (large object storage), DynamoDB (key-value storage), SQS (queueing service), and SNS (notification service) for AWS [8]. These are considered serverless because the operations, provisioning, and scaling are transparent to the end user [3]. However, BaaS solutions are incompatible with deploying application code with user-defined functionality. For this, a Function-as-a-Service solution should be used instead.

Function-as-a-Service enables developers to upload functions and declare events that trigger the functions [8], either from the end user's HTTP request or platform-created events from its BaaS services. Resources are scaled automatically to match the incoming function invocation rate and execute more instances of the function [3]. Several enterprise FaaS cloud computing platforms exist, such as AWS Lambda [9], Microsoft Azure Functions [10], Google Cloud Functions [22], and IBM Cloud [23], which is an enterprise fork of

Apache OpenWhisk [24]. AWS Lambda is the dominating FaaS platform at 80% [25], which is likely due to how AWS itself also has the largest market share at 47.8% and AWS Lambda was released two years before other Cloud Service Providers released their FaaS solutions [25].

FaaS solutions are often priced based solely on the individual function invocations' execution time, ignoring platform (e.g., network, cluster-level scheduling, queueing) and system-level details (e.g., container management and OS scheduling) [3]. Developers also pre-set a maximum memory limit on the functions, which is usually scaled with the performance of the virtual CPU associated with the function invocation. Some platforms also allow users to set a custom limit on CPU usage. The hosting cost is often measured in the units of Gigabyte-seconds (GB-s) for Gigahertz-seconds (GHz-s), depending on whether the memory limit or CPU power is used as the unit [3].

2.2 Web API

To evaluate whether deploying web API to a serverless platform has any merit, it is important to precisely define what web API is and what its characteristics are. An API, or Application Programming Interface, is a set of protocols and definitions for components of an application to interoperate with each other or with other applications [26]. Having a well-defined API means that the functionalities of the application are well-documented (or self-documenting) and are easy to use, whilst abstracting away the implementation details that are irrelevant to the end-user. APIs can improve the development and operations/-maintenance process by providing a well-defined contract or promise on the functionalities between parties, and they can rely on them to focus on the functionalities of their product without worrying about how it interacts with other components unexpectedly [26].

A web API is a type of API that focuses on the contracts allowing communication between a client and a server over the internet. Communications between clients and servers typically are in the form of a Request from the client and a Response from the server, which are coordinated through the use of standard HTTP protocols. These APIs enable the creation of web-based systems and applications that provides users with remotely-accessible functionalities, such as social media platforms, weather services, and e-commerce websites [26].

REST, or *REpresentational State Transfer*, is an architecture with a set of constraints that can be applied to design web APIs that are simple, scalable, and easy to use [27]. For an API to be considered RESTful, the following criteria must be satisfied [28]:

- *Uniform interface*: Clients and servers should communicate in a single standard consistent format, such as JSON (JavaScript Object Notation), across the entire API provided by the server. Communications are standardised through the use of protocols like HTTP, and resources on the server should be identifiable with a standardised URI (Uniform Resource Identifier).
- *Client-server*: Clients and servers must be able to evolve independently without any dependencies on each other.
- *Stateless*: No context or intermediate state should be stored on the server between requests. If the client application must implement functionalities of a stateful application, each request should contain all necessary information–including authentication details–to service the request without context.
- *Cacheable*: If a resource can be cached, it must be declared as so, to improve performance on the client-side, and enhance the scope for scalability by reducing the load on the server side.

- *Layered system*: Servers may consist of multiple layers that serve independent purposes (e.g. microservices), and the internal architecture of servers is invisible to the client.
- *Code on demand* (optional): Servers may return executable code to implement some functionality.

In the context of serverless deployment, one of the most important constraints of RESTful architecture is the idea of statelessness. As the server does not have to manage a session state between each request, the server can gain more control over the scalability of functions by deploying each request on different hardware without the concern of managing a session through physical connection or virtual abstractions over it [27].

2.3 Deploying Web API on Serverless Platforms

This section discusses the potential benefits and current limitations regarding deploying web API onto serverless platforms.

2.3.1 Potential Benefits

There are several advantages to FaaS systems that could benefit the developers of web API. As discussed in Section §2.1.2, FaaS developers do not have to perform any server provisioning or scaling, eliminating the need for manual management of compute bottlenecks. As the FaaS functions are abstracted away from physical hardware, the FaaS providers can retain full control over which functions are assigned to which servers, meaning that FaaS functions increase server utilization [3]. Ivan et al. [29] conducted an experiment that compares the performance of applications deployed on Monolithic VM, Microservice VM, and FaaS platforms. It shows that on small, predictable loads the monolithic app performs better than other platforms, but the performance degrades much more quickly as the load increases, and applications deployed on a FaaS platform have a near-constant response time due to how the service provider manages the servers, enabling a near-infinite scaling. This is beneficial for web APIs, as web applications can potentially have thousands to millions of concurrent users at any given time.

FaaS is priced based on the function invocations' actual execution time, rather than per API call or VM uptime. This 'pay-per-use' model means that the developers are only charged for the exact amount of resources and time their functions use. This–with how FaaS applications can easily scale to zero–makes it especially beneficial for deploying web applications, which are known to have potentially bursty workloads [25]. When there is a sudden surge in the load on a service, the FaaS system can easily scale out to accommodate the increase in the number of users, and once the load fades away the system can then decommission the additional infrastructure that is no longer necessary. This could potentially be saving on the deployment costs, as the developer only pays for actual use, rather than paying for the uptime of VMs regardless of the volume. Critics do exist that claim the cost savings highly depend on the execution behaviour and volume of execution [30], but others have observed that for a bursty workload using a serverless platform compared to monolithic architecture saves up to 77% in infrastructure costs due to its pay-per-use model [1].

In fact, studies have shown that serverless applications are already used to implement web APIs. Eismann et al. [25] conducted a comprehensive survey on various enterprise and open-source serverless applications, and about 29% are implementing APIs. This is a surprising result as the consensus is that serverless applications are suitable for operations tasks and batch jobs, but is unsuitable for implementing core functionality, such as a

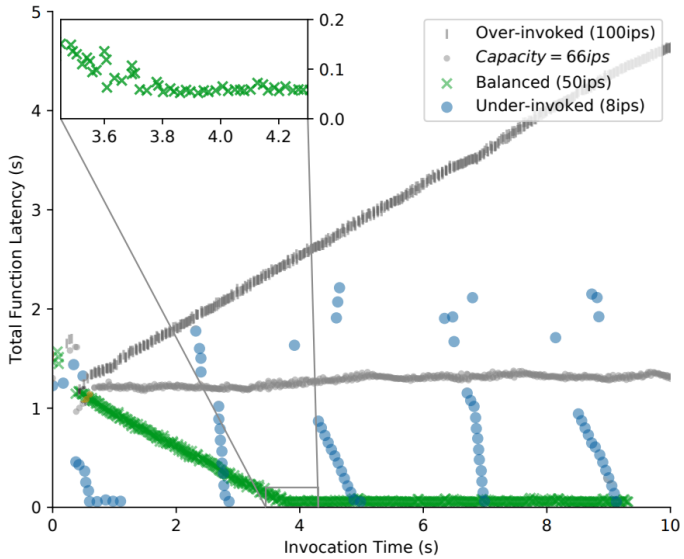


Figure 2.5: Function Latency over time for Various Invocation Rates [Adapted from 3]

web backend and video delivery, due to its cold-start latency (discussed in more detail on §2.3.2) [31]. However, this is potentially due to the advantages of FaaS despite the problems outlined in the consensus, as the survey outlines that the main drivers of serverless adoption are reduced hosting costs (47%), reduced operations effort (34%), and high scalability (34%) [25], which indicates that the developers implemented the core functionalities of the web APIs were developed on serverless platforms regardless of its latency characteristics. In fact, they could have used serverless platforms despite recognising its latency issues, or even was not aware of its severity due to the marketing materials of commercial serverless platforms de-emphasising the latency of function invocations [29].

2.3.2 Problems with Existing FaaS Systems

As discussed in Section §2.1.2, FaaS functions are often deployed using container images (often Docker) provided by the developer. This is mainly to make setting up the dependencies and runtime of the function instance easier, as container images provide detailed instructions on how to set up the functions based on an isolated and consistent environment for the function to run in, regardless of the underlying cloud infrastructure [3]. As the functions need to run in a multi-tenanted cloud environment, containers provide insufficient isolation guarantees [3]. Therefore, containers themselves are internally run on top of Virtual Machines.

The latency from the function execution can be divided into three components: cold-start time, wait time, and execution time. Cold-start time or cold-start latency indicates the time spent on initializing the function containers, wait time indicates the time spent waiting inside the platform's internal queue before it starts executing, and execution time indicates the actual time spent to run the function [3]. The cold-start latency can vary significantly depending on how 'warm' the containers are, which is correlated with the internal virtualization layers of FaaS platforms. Lloyd et al. [16] define four types of function invocations regarding the warm-up of infrastructures. The 'provider cold' state indicates the very service invocation made to the cloud service provider, where the container image is initially built and compiled. The 'VM cold' state indicates when a new VM host needs to be spun up and the container image to be transferred to the new hosts and initialized. The 'container cold' state indicates that the container image is in an existing VM host, but has not been initialized yet. The 'warm' state is achieved when a container

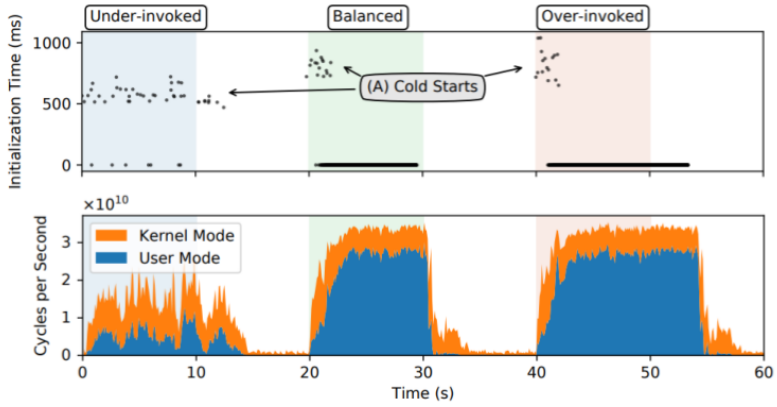


Figure 2.6: Function Start Time for Various Invocation Rates (Top) and CPU Cycle Usage for Userspace Code and Kernel Code (Bottom) [Adapted from 3]

image is already initialized for the repeated invocation and can be reused.

Cloud providers often conserve server resources and energy by de-provisioning function containers when service demand is low [16]. This practice ensures that the infrastructure is not over-utilized from unused services and reduces costs, which is especially beneficial for FaaS providers to remain profitable by maintaining a high throughput of function executions [3]. However, this also means that containers must be paused to release memory for other functions to run, effectively resetting the functions into the 'container cold' state. This can lead to an increased latency from restarting the containers as pausing and unpausing containers make lots of complex system calls to the kernel [18].

This can have a significant impact on performance. Although using containers in itself has an impact on the execution time due to the overhead of virtualization, paused containers have a significant impact on the overall function runtime. Studies show that paused containers have a 1-2 magnitude slowdown compared to native execution, whereas live containers only had a proportional slowdown [3]. In addition, a sudden surge in the number of requests can lead to more containers being provisioned, which can also result in additional cold starts, exacerbating the effect of cold-start latency for the additional requests [3].

Figure 2.5 shows the overall function runtime/latency for various invocation modes as defined in [3]. Focusing on the under-invoked mode (blue circles), it can be seen that periodically the overall latency increases in multiple orders of magnitudes from the baseline overall latency (green cross marks). This is because, in an under-invoked mode, the containers are kept idle for long enough for the FaaS platform to de-provision the containers, leading to a latency ripple effect from the cold-start latency.

Figure 2.6 (Top) shows that the function cold-start latency of 500-1000ms for a function with an execution runtime of only 50-200ms [3]. This shows that the cold-start latency has an outsized impact on the overall runtime of the functions, which is especially problematic for hosting web APIs on FaaS platforms, as web API functions are in general ephemeral with an execution time in time quantum of 1, 10, and 100ms [3]. In fact, the slowdown caused by paused or uninitialized containers have a power-law relation with function execution time, where shorter functions experience a longer cold-start latency relative to their runtime since the cold-start overhead is usually fixed/constant regardless of the function invocations' runtime [3].

Furthermore, FaaS providers also schedule the functions in a fine-grain interleaved manner, which compromises the temporal locality-exploiting hardware structures like branch predictors to significantly underperform. An experiment from Shahrad et al. [3] shows that running the same function on a serverless environment compared to native execution

causes 20 times more branch mispredictions per kilo-instructions (MPKI). This effect is exacerbated when the functions are short in length, where functions with shorter execution times have been shown to exhibit 18.8 times more MPKI than functions with the longest execution time on a FaaS platform [3]. This is due to the fixed overhead of the language runtime initialization, where for the shorter function 60.9% of executed instructions are taken up by the initialization of the language runtime. This can cause a significant impact on the branch predictor performance and overall runtime of the functions, as language runtimes often make lots of system calls, which confuses the branch predictor and ultimately slows down the functions significantly [3]. This is even worse for FaaS systems as multiple interleaved function invocations do not necessarily run on the same language runtime, further corrupting the branch predictor between each function invocation.

Figure 2.6 (Bottom) shows that a significant proportion of CPU cycles are taken up by kernel code execution, especially for the under-invoked invocation rate. This is caused by containers regularly pausing and un-pausing and the initialization of language runtime, both making lots of system calls.

All of these issues indicate that running functions that service a web API on a FaaS platform creates an excessive overhead that is in orders of magnitude higher than the actual runtime of the functions, making FaaS currently infeasible for efficiently deploying web APIs. This is problematic for developers as FaaS providers charge by the overall function runtime, which includes overheads like cold-start latency, making FaaS deployment more expensive than necessary [30]. FaaS providers also currently provide no Service-Level Agreements (SLAs) beyond the infrastructure uptime, meaning that they provide no guarantees or upper limits to important runtime characteristics like its cold-start latency, nor provide accurate performance that specifically shows such latencies [3]. It also makes the user experience much longer latencies from these additional overheads. The fact that serverless platforms are already utilised to deploy web APIs (§2.3.1) is likely to be despite these problems, not because of it.

To reduce the cold-start latencies and make FaaS usable for deploying web APIs, it is necessary to find other virtualization mechanisms that are much faster than containers, as pausing and un-pausing containers on a FaaS system are inevitable due to the memory constraints on the physical servers. It is also beneficial to find a way to eliminate or reduce the effects of language runtime initialization, which is shown to significantly affect the cold-start latency of short functions.

It is important to note that another often-cited reason for the problems with deploying a generic application on FaaS platforms involves storage and network latencies with state management through slow cloud storage solutions [8]. While it is an important factor to consider when deploying data-intensive applications on FaaS platforms, for deploying web APIs specifically it does not play a major role as web APIs are RESTful i.e. each function invocations are stateless, meaning that any intermediary states have to be stored on slower storage solutions like relational databases regardless of what architectures are used. Exploring whether introducing intermediary caching solutions for such intermediary states are beyond the scope of this research.

2.4 Software-Based Isolation Mechanisms

Memory Isolation is a critical component of multi-tenant systems like serverless platforms, as multiple users or functions can share the same underlying hardware resources, and it must be ensured that the actions of one tenant do not negatively impact the security of another [32]. Existing virtualisation mechanisms like containers and VMs rely heavily on hardware-based solutions to provide this guarantee at a very low-level, where the guarantees are enforced by kernel-level code or physical hardware that sits at a level below the

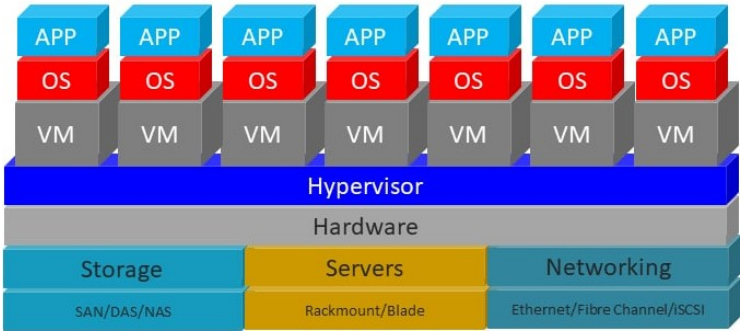


Figure 2.7: Architecture of a Typical Hypervisor System [4]

userspace code being executed [33].

One such method, which is especially popular in multi-tenanted systems, is the use of hypervisors (Figure 2.7). Hypervisor is a software layer that sits below the operating system kernel and directly interfaces with the underlying hardware, responsible for provisioning out the resources of the host machine among the virtual machines and ensuring that each tenant's resources like its system memory are isolated from each other [33]. When a process needs to interact with the hardware, such as reading from and writing to system memory, it invokes a system call, making a request to the OS kernel to execute the privileged operation on its behalf. With a system with VMs and a hypervisor, these system calls are first intercepted by the hypervisor, which verifies it to ensure that the process issuing the system call is not accessing privileged or other process's memory [33]. As this process, known as a 'trap', runs at a privileged level, it involves context switching between different address spaces from userspace-level code to kernel-level code, it induces a significant overhead and longer initialisation times [32], which is especially problematic in serverless systems that require both isolation guarantees and a rapid (de)provisioning of functions.

Due to these reasons, software-based isolation techniques have been proposed as an alternative to hardware-based solutions for memory isolation. As software-based techniques run on a higher level than hardware-based solutions that require kernel-level code executions [32], meaning that the overall execution of code requires far fewer kernel interactions or system traps that—as shown above in §2.3.2—dominate the execution time of serverless functions. One such technique is known as Software-Fault Isolation (SFI). A 'fault' in a system refers to an abnormal condition where a software module behaves unexpectedly or performs undesirable actions, such as trying to access memory it is not allowed to access or attempting to execute invalid instructions [32]. Whereas hardware-based techniques which prevents faults from one module affecting other modules by segregating each module into its own separate address space and require context switching to kernel to handle all memory accesses, SFI instead prevents these faults from affecting other modules by logically segregating each process into its own memory region called 'fault domain' and modify the modules' object code to prevent them from jumping or modifying to an address outside of its fault domain. Exactly how the object codes are modified when using SFI is beyond the scope of this research [32], but allowing all modules to live within a single address space and using userspace-level checks to properly isolate the modules avoids the need for kernel traps and system calls at a slight increase in the number of instructions executed, which as shown before is trivial compared to kernel-level code execution time in serverless functions.

However, software-based isolation techniques like SFI alone cannot provide resource isolation, as these techniques solely focus on memory safety and do not prevent other modules from corrupting resources that are allocated on a per-adderss-space basis [5]. For example, if one module makes a system call to delete files that are needed by another module in the same address space, these techniques will not prevent these actions from

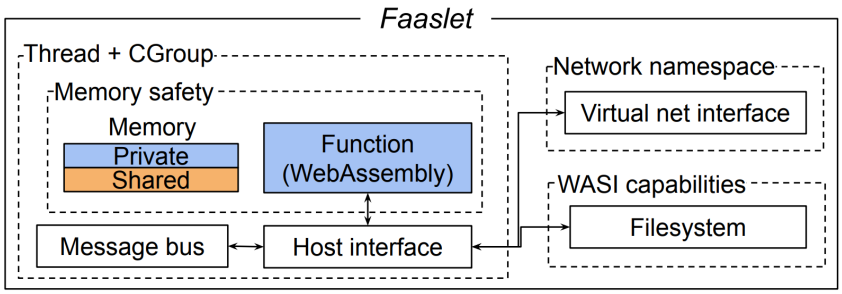


Figure 2.8: *Faaslet* Architecture [5]

corrupting the modules, potentially causing the system as a whole to crash. Therefore, these isolation techniques must be complemented with other techniques to ensure the reliability of serverless systems.

2.5 *Faaslets* and `Faasm`

This section introduces a novel lightweight isolation mechanism for serverless platforms *Faaslets*, along with a serverless runtime `Faasm` that runs serverless functions using *Faaslets*.

2.5.1 *Faaslets*

Shillaker and Pietzuch [5] introduced *Faaslets* as a novel lightweight isolation mechanism based on *WebAssembly* [34], along with `Faasm`, a serverless runtime that uses *Faaslets*. As discussed in previous sections (§2.3.2), containers suffer from cold start latencies of hundreds of milliseconds to seconds, which make FaaS solutions infeasible for deploying web APIs. *Faaslets* use alternative isolation mechanisms, such as the use of WebAssembly runtime, SFI, and standard Linux `cgroups`, to ensure proper isolation of different tenants in a serverless environment while avoiding the use of expensive isolation mechanisms.

Applications run on a WebAssembly runtime offer strong memory safety guarantees by abstracting the memory model into a single linear byte array, providing efficient bounds checking at both compile- and runtime. Runtime checks are implemented as traps, which are implemented as a core part of WebAssembly runtimes [35]. The security guarantees of WebAssembly have been well established in existing literature, such as in formal verification [36], taint tracking [37], and dynamic analysis [38]. One such security guarantee is known as Software-Fault Isolation (SFI) [32], which provides lightweight memory safety through static analysis, instrumentation and runtime traps.

Faaslets rely on SFI for memory safety and isolation guarantees. To isolate and provision CPU accesses, it uses Linux `cgroups` similar to containers. Every function invocation is executed in its thread of the shared runtime process. These threads are each assigned to a `cgroup`, and the Linux's scheduler ensures that they get an equal share of the CPU cycles. This means that many *Faaslet* functions can be executed safely and efficiently on a single machine. It is an improvement over containers with a cold-start latency of less than 10ms, and a memory footprint below 200KB.

To further reduce the cold-start latency, the paper [5] introduces snapshots called *Proto-Faaslets*, which are ahead-of-time pre-initialized *Faaslets* with a snapshot of its memory. *Proto-Faaslets* include the function's stack and heap, function table, stack pointer, and data, as defined in the WebAssembly specification [35]. Due to the simple memory architecture of WebAssembly functions, *Proto-Faaslets* can be restored in hundreds of microseconds. A single *Proto-Faaslet* image can also be used to quickly scale horizontally across multiple hosts. It is used to create new *Faaslet* instances quickly without any fixed over-

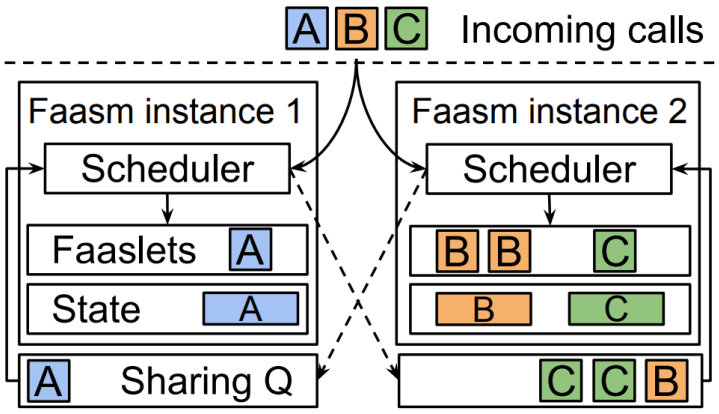


Figure 2.9: `Faasm` Architecture [5]

head, such as the time taken to initialize a language runtime. User-defined initialization code can also be supplied that can be executed before snapshot, which helps reduce the initialization overhead if the function has a fixed initialization code at the beginning. It could also be used to reset *Faaslets* after each function invocation. As *Proto-Faaslets* are a snapshot of a function's initialized state, restoring it guarantees that no data from previous invocations are retained, making it safe for multi-tenant serverless systems. This is normally not safe with container-based platforms, as they cannot ensure that the container's memory is cleared between each invocation.

2.5.2 `Faasm`

`Faasm` is a serverless runtime that uses *Faaslets* to execute functions in its isolated thread on a multi-tenant system on a process within a single address space. `Faasm` is structured in a distributed architecture. There are multiple `Faasm` runtime processes that execute on a pool of servers, and each process manages its dedicated pool of *Faaslet* functions. `Faasm` easily interoperates with other serverless platforms, which can be used to provide the underlying infrastructure, auto-scaling, and user-facing frontends. Then `Faasm` manages the scheduling and execution of *Faaslet* functions. Languages with an LLVM frontend like C++ are natively supported by `Faasm`, as LLVM IR can be compiled into WebAssembly. Interpreted languages like Python can also be supported on `Faasm` by compiling the language runtime to WebAssembly, such as the CPython runtime for Python programs. The `Faasm` runtime translates the applications written in languages like C/C++ and Python into WebAssembly using the LLVM Compiler Toolchain [39].

`Faasm` provides an upload HTTP service that users can upload WebAssembly binaries, after which `Faasm` translates the binary and writes the output object files to a shared object store. It depends on where `Faasm` is hosted, but a Cloud Service Provider's storage solution can be used, such as AWS S3. `Faasm` also provides an invoke HTTP service which is used to invoke the uploaded function. On a cold start of *Faaslet*, it retrieves the object file from the object store and *Proto-Faaslet* and restores the *Faaslet* image.

`Faasm` also supports the execution of Python functions by also accepting Python script files through the upload HTTP service. This is different to native WebAssembly functions, as it accepts a Python source file instead of compiled WebAssembly binary. This is because Python functions are intrepreted on top of a CPython runtime, which in itself is compiled in WebAssembly.

`Faasm` supports a read-global write-local filesystem, which lets functions read files from the shared object store and write to locally cached versions of the files. It is mainly used

to support language runtimes like CPython which needs a filesystem for loading Python library code and storing intermediate Python bytecode. The filesystem is accessed through a UNIX-style API, which itself is internally implemented with the WASI (WebAssembly System Interface) capability-based security model, which provides efficient isolation through unforgeable file handles [40]. This means that using the filesystem does not add to the cold-start latency of *Faaslets*.

Chapter 3

Related Work

3.1 Alternative Serverless Isolation Mechanisms

As hinted in §2.4, many academics and organisations have already recognised the importance of reducing the overhead of virtualisation that comes with containers and virtual machines, especially with cloud systems like serverless platforms. *Faaslet* is far from the only attempt at proposing an alternative isolation mechanism for serverless systems, whether through iterating on the existing Linux- or Windows-based containers and VM-based model, or proposing an entirely new isolation paradigm specifically optimised towards cloud systems.

Projects like Firecracker [41] and Cloud Hypervisor [42] are examples of a 'MicroVM' approach. This approach introduces a specialised Virtual Machine Monitor (also known as a Hypervisor) that only includes the necessary functionalities designed specifically towards running containers in a secure environment with multi-tenant usage. It then provides a minimal Virtual Machine (hence 'MicroVM') that supports running hosts running functions in an isolated environment with a minimal but fully functional guest Linux OS, completely sandboxed away from the underlying server hardware, providing strong security and isolation in a more lightweight environment. This approach has proven successful for existing commercial FaaS systems. For instance, Firecracker, which was developed by the Amazon AWS team, was deployed on the most popular FaaS platform AWS Lambda [41]. However, this approach still uses containers and VMs for resource isolation. This means that, although it does reduce the overhead associated with using VMs, this approach does not modify containers in any way, leading to a still significant overhead associated with using containers, especially for running ephemeral functions like those found in web APIs. Also, as it uses standard containers, it still has the same cold-start problems that dominate the runtime of infrequently invoked functions or functions run in a massively multi-tenant environment where containers are frequently deprovisioned. This solution is appropriate for established commercial FaaS platforms like AWS Lambda that already use containers and VMs for isolation, as introducing a breaking change at the scale of completely replacing containers in a commercial platform is financially infeasible. This research focuses on novel approaches that could show how alternative isolation mechanisms like *Faaslets* can make deploying web APIs on a FaaS system an attractive solution where there are no existing legacy system that needs to be supported.

There are also approaches that attempt to reduce the overhead of containers, usually by increasing the level of trust in users' code and weaken the level of resource isolation i.e. grouping multiple chunks of user code together. For instance, PyWren [43] builds a layer on top of AWS Lambda, where instead of registering functions individually, a single function is registered that acts as a sole runner function with a global key-value store S3 to manage the invocations. This makes it more likely for containers to be reused, as

from the perspective of AWS Lambda's infrastructure, the same single function is invoked repeatedly, ultimately leading to a reduced likelihood of cold-start invocations. Crucial [44] takes a similar approach, but with a custom-built distributed shared memory and a single shared JVM instance to run Java functions on top of. Approaches like this, which essentially reuse container instances to execute multiple functions, do help with reducing the likelihood of experiencing a cold-start overhead, but it not only breaks the isolation guarantees that must be provided by FaaS platforms, it also breaks the fine-grained elastic scaling promised by the serverless vision by tying the scaling of multiple functions together. It also exacerbates other resource overheads from using shared memory for function invocations. Overall, approaches like this are unsuitable for FaaS platforms that expect a multi-tenant usage pattern as it provides insufficient isolation guarantees and scaling flexibility.

There have been a significant number of alternative isolation mechanisms proposed that are tailored towards serverless workloads. Cloudflare Workers [45] achieves software-based isolation by running instances of V8 engines on separate V8 Isolates [46]. This offers similar benefits as *Faaslets* with regards to memory isolation overheads. However, *Faaslets* are more versatile as it natively supports all languages that could be compiled into WebAssembly, whereas for Cloudflare Workers it supports a subset of those languages as it needs to be run on top of V8 engine rather than natively. Krustlets [47] utilises WebAssembly and WASI-based runtimes to run serverless functions similarly to `Faasm`, but it is intended as a drop-in replacement for use in Kubernetes. Other projects propose an entirely novel virtualisation paradigms, such as Dune [48] which is a virtualisation system that interfaces with virtualisation hardware at a per-process abstraction rather than per-machine level, and Light-weight Contexts [49] and Picoprocesses [50] which propose an alternative to existing Linux Processes that could be used in serverless systems as a new virtualisation layer on top of physical hardware. These novel virtualisation paradigms potentially could lead to an improvement in virtualisation overheads, but as they are not completely compatible with existing programs that are run on top of classic Linux Processes, it requires extra effort from users and/or serverless providers to implement into their systems.

3.2 Snapshots and Caches

There have been several research that have specifically focused on the cold-start problem and how to reduce the latency associated with functions experiencing a cold start. Several methods have been proposed involving the use of snapshots or caches, both aiming to provide a quicker way for the workers for the functions to start quicker with less latency.

Snapshotting is a process where, once an instance of a function is fully initialised, its complete state is captured and serialised in some form and stored onto a disk. On further instances of cold start, instead of initialising the function instance from ground up, it uses the saved state to restore the states of the initialised function. Projects like Catalyzer [51] and Firecracker Snapshots [52] are successful examples of implementing snapshots on a serverless FaaS platform, showing significant improvements where Catalyzer for instance achieves orders of magnitude lower startup latency, reaching sub 1ms latency in the best case. Ustiugov et al. [53] made further improvements through recognising that the runtime of a function is dominated by page faults served one by one lazily into the container's memory, as due to the nature of how the memory pages of each function instance are mapped in the underlying host OS, there is virtually no spatial locality in the access of pages on the host hardware, which essentially invalidates all read-ahead prefetching done by the host OS. Combined with the observation that the pages accessed are largely the same across multiple invocations of the same function, REAP (REcord And Prefetch)

mechanism is proposed, which records which pages are loaded by each function invocations and proactively fetches the entire set of pages in a single disk read to eliminate most of page faults, further reducing the cold-start latency by an average of 3.7 times. *Proto-Faaslets* are a continuation in the snapshot approach, where it contains the function's stack, heap, function table, and other metadata [5], and the recovery of the snapshot is made quick by the simple memory model used in the WebAssembly runtime.

Caching is another prominent approach to reduce the cold-start latency, where a set of pre-warmed function instances are in memory and ready to process requests, achieving a trade-off of a higher memory usage for lower average function invocation latencies. For example, in an Android runtime, each app process is always forked from an existing process called Zygote, which is started after the system boot and after common framework code and resources are loaded, enabling most system memory pages that are allocated for those common code can be shared for different processes. It also mmaps static data like Java Runtime code and shared library code to allow further sharing of memory pages across different processes [54]. SOCK [55] uses this Zygote approach to spin up function instances much more quickly. SAND [56] implements a feature where pre-initialised sandboxes could be reused across different function invocations, where a set of long-lived containers can execute multiple functions.

Chapter 4

Design

This research aims to explore whether a web API used on a large-scale could be effectively deployed on a serverless FaaS platform like `Faasm`, which uses alternative resource isolation mechanisms that avoid the use of containers or VMs for (de)provisioning functions. To explore this, a product must be developed that allow a conventional web application that implements a web API to be seamlessly deployed onto `Faasm`.

A successful implementation of this product should be able to demonstrate, by deploying a web API on `Faasm`, a comparable latency to a native deployment of web APIs on non-serverless monolithic systems, showing latencies below an order of magnitude difference. This would be an improvement from existing serverless systems that show an order of magnitude or more latency overhead through the use of heavy hardware-based isolation architecture like containers and VMs.

4.1 Requirements

Currently, `Faasm` natively supports the execution of C++ and Python functions, along with other languages like C and Rust that do not have a language runtime and could be natively compiled to WebAssembly [5]. Of the programming languages available for `Faasm`, Python was chosen as the language to implement the web APIs, both for the abundance of web application frameworks and web APIs available in Python, and the benefits of being able to test the latency overhead of running an interpreted language with a language runtime like Python. Python has two most popular web application frameworks available: Django [57] and Flask [11]. For the purposes of this project, Flask is a better fit, as Flask is a more lightweight framework with more flexibility in how the web application could be structured, whereas Django is known to enforce a more rigid structure to the applications, making porting applications over to it less convenient. Flask also exposes more of its internals to the user that wants to modify its behaviour, which this product will need to do. Ideally, the product should introduce minimal code duplication, as if in the future the product needs to support other web application frameworks, only the Flask-specific interface for the function hijacking would have to be modified, leaving other parts of the product framework-agnostic.

In order to demonstrate that this solution could feasibly be used on existing web applications deployed on a serverless system, care must be taken to ensure that zero modifications on the web application would be necessary. This requirement ensures that the product remains portable for other existing applications, and if it shows significant strides in the aforementioned improvements, it would demonstrate that even existing commercial FaaS platforms could possibly replace the existing container-based isolation mechanisms with software-based isolation mechanisms like *Faaslets*. For this requirement to be satisfied, the product should be compatible with web applications developed in Flask, where

the source code does not have to be modified in any way, and the product will read in the web application object along with other source information to reconstruct a new version of the web application that utilises `Faasm`.

As the web application cannot be directly modified to use `Faasm`, care must be taken to minimise the overhead in the 'critical loop' of the product. A 'critical loop' in this project is the sequence of operations during a HTTP call to the web API, which begins from the Flask app receiving an API call, sending the request data and input to `Faasm`, waiting for `Faasm` to process and return the result, parsing the output, and finally responding back to the original requester with the response. It is vital to introduce a minimal amount of overhead during this process, as it directly contributes to the overall latency of the request handler logic. This means that the amount of actual executed code during web API call should be minimal, but any amount of processing or logic prior to deploying the web application could contain more complex logic and processing.

4.1.1 Limitations

There are certain limitations imposed by `Faasm` and the listed requirements that do need to be addressed.

`Faasm` exposes an HTTP API that other systems can use to interact with it. There are several limitations imposed the use of standard HTTP protocols. `Faasm` uses HTTP/1.1, which is a text-based protocol rather than a binary-based protocol [58], which is inherently less efficient as the input and request data must be formatted as text. This may degrade the performance from the overhead of encoding and decoding text-based data for input parsing, which is included in the critical loop of the application. However, it is likely that this overhead would play an insignificant role as the input data size is very small, since the API functions themselves are ephemeral in nature, which means that the possible inputs would not be large enough to significantly affect the performance of the critical loop. An alternative would be to introduce an RPC-based API endpoint in `Faasm`, which uses standard binary formatting for the input data, along with maintaining a connection between `Faasm` and the product to avoid sending extraneous HTTP header data and re-establishing TCP handshake between the two systems. This approach has not been taken as it involves a significant development effort, although care must be taken to accurately measure the overhead introduced by using a text-based protocol for data transfer between the two systems.

Using HTTP also introduces a minor issue, as any asynchronous invocations would have to work on a polling-based system, where the client must periodically send a request to poll on the `Faasm` server to check whether the function invocations are finished. This introduces a discrete time step overhead between the end of a function execution and the next polling request from the client, along with unnecessary network traffic from the repeated polling requests made. This however is a non-issue in the system, as standard REST API are expected to be invoked in a synchronous manner anyway. If in the future asynchronous invocation needs to be supported, however, `Faasm` would have to introduce a new API endpoint that establishes a bidirectional connection between the client and the server, where the server (`Faasm`) can proactively notify the client that the function execution is finished.

As discussed prior, `Faasm` does support the execution of Python functions by hosting a WebAssembly-compiled CPython runtime as a whole on its worker and running Python code on top of it. This does introduce several restrictions in the design of the product. One restriction it imposes is that the Python code must be uploaded as a script source file that contains a `faasm_main()` function acting as an entry point, meaning that each function in `Faasm` must effectively be a self-contained executable script with proprietary function for handling inputs and outputs to the function. Although local library code could

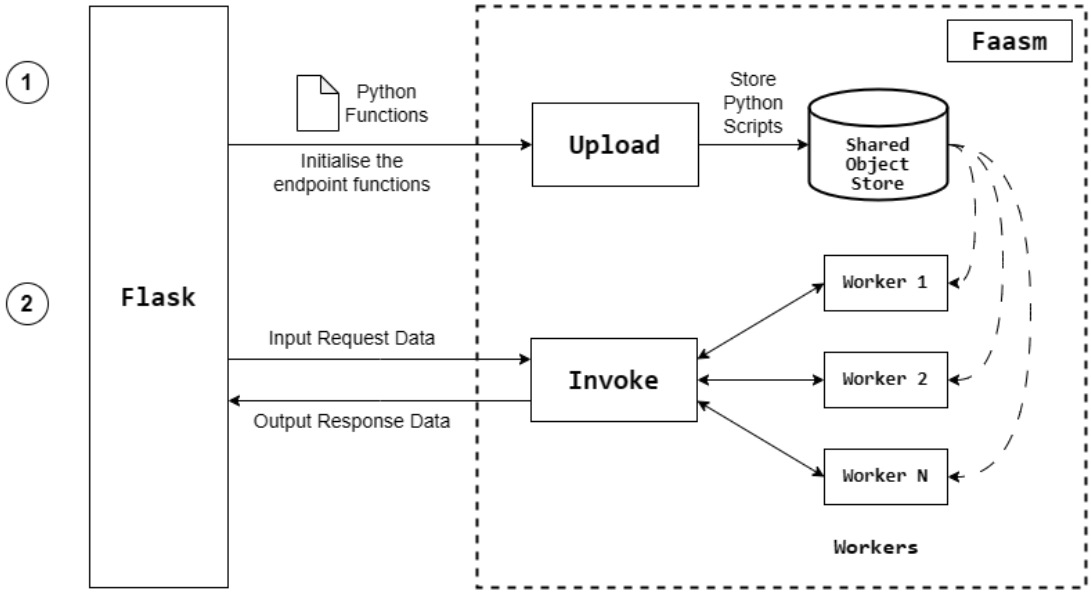


Figure 4.1: Architecture Design of the Proposed Product

be supported through either uploading its source on the `Faasm` file system in appropriate locations or uploading those functions as separate *Faaslet* functions and invoking those functions from the main function, it either introduces complex dependencies on the file system or modification on the script's source code itself, neither of which are necessarily desirable. This, combined with the fact that the script must be uploaded using the HTTP API in text format, means that the uploaded script would be quite large in its size. This, however, is not an issue as uploading the functions are not part of the critical loop.

However, there is a real potential for a latency overhead that results from having a larger script file to execute for each function invocation, as each function invocation request has to run the entire script through the CPython interpreter, including any potential library code which in native execution would be run once on the first import and its contents would be stored as object for later use. As functions are ephemeral in nature, including any library code that could potentially be used, it should not lead to a major increase in latency, but this should be carefully monitored during evaluation to ensure that the overhead stemming from reloading the script for every invocation is not significant enough to have an adverse effect on the overall latency. Potential improvements to this issue are discussed later in §7.1.

Faasm also does introduce a limitation from using a WebAssembly cross-compiled CPython runtime, as the use of third-party libraries like Numpy and Pandas require additional effort to port over, especially if the libraries contain C extension code which would also need to be cross-compiled over. Efforts could have been made to introduce support for the third-party libraries on a case-by-basis, as it would involve processes like cross-compiling the CPython runtime itself. However, it has proven to be unnecessary for this research as it was neither necessary for the web APIs tested nor was a focus of this research.

4.2 System Blueprint

Figure 4.1 visualises the overall system architecture of the proposed product. The new system provides an adapter layer between existing Flask web applications and `Faasm`, where the endpoint functions on the Flask app invokes the *Faaslet* functions on `Faasm`

that execute the same function in a serverless platform. This is effectively the same as deploying individual endpoint functions of a Flask app onto `Faasm` directly, except the Flask app stays in tact, handling the networking logic while delegating the provision and scaling of functions to `Faasm`. The product's operation is divided into two main phases: initialisation and invocation.

The initialisation phase (indicated as (1) in Figure 4.1) of the product occurs after the Flask web application has been set up but before it is deployed and running. The product will modify the app such that it will take the functions that handle the request for each endpoint and upload it to `Faasm` via its upload API. Each endpoint function in the app corresponds to a *Faaslet* function in `Faasm`, packaged in a way that's compatible to be executable with the workers. The functions are stored in the shared object store in `Faasm`, where the functions are stored as Python source files.

The invocation phase (indicated as (2) in Figure 4.1) of the product represents the critical loop of the product, where the end-user of the Flask API sends a request to an endpoint in the API. The product will capture the request sent to the Flask API, intercepts it, and then forwards the request data to `Faasm` via its invoke API. Each invoke request executes a *Faaslet* function that implements the feature of the endpoint function originally in the Flask app. Once an invoke request is made to `Faasm`, it automatically provisions a worker instance to execute the function by initialising the WebAssembly CPython runtime, then fetching the uploaded Python function script from the shared object store, and then executing the fetched script on the loaded CPython runtime, and then the result is returned back to the Flask app, which then forwards the output back to the end-user. Effectively, the Flask app with the product acts as a gateway frontend of the API infrastructure that processes the HTTP requests and forwards the actual execution to `Faasm`, which then handles the provisioning of functions and execution in line with the serverless paradigm.

Chapter 5

Flask-Faasm

This chapter introduces *Flask-Faasm*, an implementation of the product outlined in Chapter 4, which provides a adapter layer to deploy Flask web applications on the `Faasm` serverless runtime.

5.1 Overview

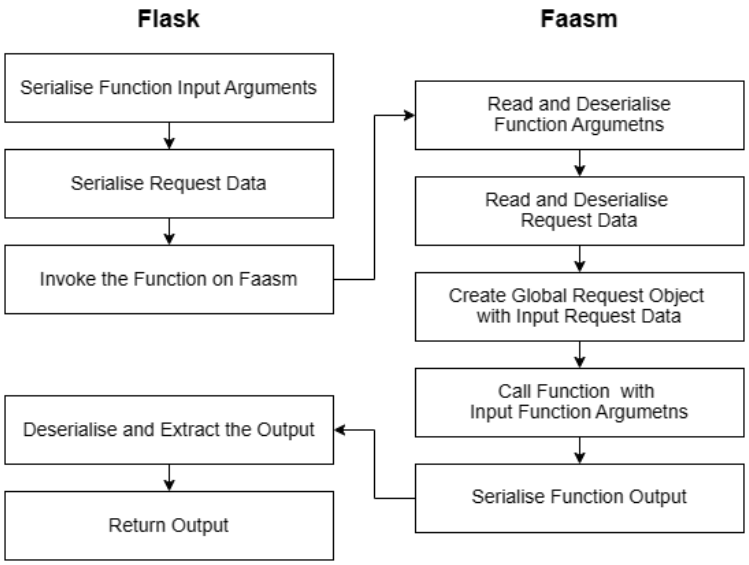


Figure 5.1: Flowchart Demonstrating the Workflow of Invoke Phase

As specified in Figure 4.1, the execution of *Flask-Faasm* can be divided into two phases: upload phase and invoke phase. The upload phase involves the processing and upload of the Flask endpoint functions to `Faasm`, which consists of the following steps:

1. Package the endpoint function to be compatible with `Faasm`.
2. Upload the packaged function to `Faasm` via the upload API.
3. Replace the endpoint function in the Flask app with an entry function that invokes the uploaded function on `Faasm` with the invoke API.

The invoke phase involves the execution of Flask entry function, which in turn invokes the function on `Faasm`. Figure 5.1 shows a typical workflow during the invoke phase, where

Code 5.1: Example Python Echo Script for `Faasm` [Adapted from 6]

```
1  from pyfaasm.core import get_input_len, read_input, write_output
2
3  def echo() -> None:
4      input_len = get_input_len()
5      if input_len == 0:
6          write_output("Nothing to echo")
7          return
8
9      input_data: str = read_input(input_len).decode("utf-8")
10
11     write_output(input_data_str.encode(encoding="utf-8"))
12
13 def faasm_main() -> int:
14     echo()
15     return 0
```

the left-hand side shows the workflow of Flask entry function and the right-hand side shows the workflow of the uploaded function on `Faasm`. The process mainly involves serialisation and deserialisation of the function input arguments, request data, and function outputs.

For both upload and invoke phase, the following questions on Flask and `Faasm` need to be answered in order to understand the mechanism of *Flask-Faasm*:

1. How exactly are Python functions executed in `Faasm`?
2. How can external systems interact with `Faasm` for uploading and invoking functions?
3. How does a Flask app work, and how can the behaviour of endpoint functions be changed without modifying the app itself?
4. How does *Flask-Faasm* create an interface between Flask and `Faasm`?

5.2 Python Function Execution in `Faasm`

5.2.1 Structure of the `Faasm` Python Scripts

Any Python code executed in `Faasm` must have a specific structure for it to be compatible with the WebAssembly-compiled CPython runtime running on a Python *Faaslet*.

Code Listing 5.1 shows an example echo script written to be compatible with `Faasm`. The following explanation may make references to the specific line numbers on this code listing.

All `Faasm` Python functions have a fixed starting point of `faasm_main()` function (**Line 13-15**). In some respects, it acts similarly to a standard C/C++ `main()` function, as it also returns an exit code, but it does not accept command-line arguments in the form of `argc` and `argv`. As mentioned before, all code necessary to execute the function in `Faasm` must effectively be contained in a single script file, including the `faasm_main()` function.

`Faasm` includes a library called `pyfaasm` [59] that provides a Python interface to the `Faasm` host interface [60], which includes support for handling inputs and outputs for the `Faasm` function invocations. Specifically, the `pyfaasm.core` library contains three functions relevant for handling the inputs and outputs:

Line 4 `get_input_len() -> int`: Returns the number of bytes the function input contains. Useful for a call to `read_input()`.

Field	Type	Default	Description
`async`	Boolean	`False`	Whether the function should be executed asynchronously. Always False for *Flask-Faasm*.
`user`	String	`"python"`	`<user>` of function URI. Should be set as `"python"` if `python` field is set as True.
`function`	String	`"py_func"`	`<function>` of function URI. Should be set as `"py_func"` if `python` field is set as True.
`python`	Boolean	`True`	Whether the function to invoke is a Python function or not.
`py_user`	String	n/a	`<user>` of URI if Python function invoked.
`py_func`	String	n/a	`<function>` of URI if Python function invoked.
`input_data`	String	`""`	Base-64 encoded string of function input, to be read by the `read_input()` function.

Table 5.1: JSON Message Fields for the Invoke API of `Faasm`

Line 9 `read_input(input_len: int) -> bytes`: Given the number of bytes to read from the input (usually set to the result of `get_input_len()`), it returns the values read from the function input stream.

As the results are given in bytes, and the function input is likely in string format if the HTTP API is used for invoking the function, it usually needs to be converted back into string via call to the `_.decode(encoding="utf-8")` method.

Line 11 `write_output(output_data: bytes)`: Writes the given output data in bytes to `Faasm` 's output stream.

As the function accepts bytes, if the output data is text, it must be converted to bytes via call to the `_.encode(encoding="utf-8")` method.

5.2.2 `Faasm` HTTP API

As explained in §2.5.2, `Faasm` exposes an HTTP API that enables external systems to interact with it to facilitate the use of `Faasm` in code. `Faasm` exposes two services, each responsible for the creation and execution of *Faaslet* functions: upload service and invoke service.

The upload service allows its clients to upload the function's code to `Faasm` to be stored for later use in invocation. All functions in `Faasm` are addressed by its URI (Uniform Resource Identifier) `"/<user>/<function>"`, where `<user>` indicates a function group the function belongs in and `<function>` indicates a unique identifier for the function within the `<user>` function group. The upload service exposes two endpoints for facilitating the upload of different types of function code. The endpoint `"/f/<user>/<function>"` is used for uploading WebAssembly compiled C++ (or other runtime-free language) code, and the endpoint `"/p/<user>/<function>"` is used for uploading the source code of the Python script file. Both endpoints are necessary for *Flask-Faasm*, as the WebAssembly endpoint is necessary for uploading the CPython runtime (at URI `/python/py_func`) and the Python endpoint is necessary for uploading the API functions written in Python.

The invoke service enables its clients to execute the uploaded functions by specifying the function's URI and the input data to pass along. Table 5.1 shows the fields that need to be specified in the request body for an invocation request. The response from the invoke service request is formatted as follows:

```
1 <output on stdout/stderr>
2 Python call succeeded
```

Code 5.2: Minimal Example of a Flask Web Application with an 'echo' Endpoint

```
1  from flask import Flask, request
2
3  app = Flask(__name__)
4
5  @app.post("/echo")
6  def echo():
7      data = request.get_data()
8      return str(data)
9
10 if __name__ == "__main__":
11     app.run(host="127.0.0.1", port=8080, debug=True)
```

```
3
4
5  <output from write_output() function>
```

Therefore, in order to extract the output of the function generated by the `write_output()` function, a string separation function could be used with the delimiter `"Python call succeeded\n\n"` e.g. `response.text.split(DELIMITER)[1]`.

5.3 Intercepting HTTP Request Handler in Flask

5.3.1 Examining the Structure of Flask Applications

As Flask is a highly modular and customisable web framework, web applications built with Flask have a high degree of flexibility, meaning that the codebase is leaner and more efficient by allowing developers to pick and choose only the specific components necessary to specify the functionalities of the web API. Therefore, simple web APIs developed in Flask contain a minimal amount of code overhead and are very simple in nature.

Code Listing 5.2 shows a minimal web API developed in Flask with an 'echo' endpoint that responds with the input data in the request. The following explanation makes references to the specific line numbers on this code listing.

Line 3 A Flask application is an instance of the `Flask` class, which accepts a name of the package the app belongs in, which is usually set as `__name__` for most applications. This object acts as the central point for the application.

Line 5-6 Each endpoint in the web API are registered by writing a regular Python function that implements the functionality and returns the expected response for the input, and then using a set of decorators provided by the `Flask` app instance to specify the endpoint and HTTP request method for the endpoint.

For instance, the listing shows an example of registering a function for the `"/echo"` endpoint with `POST` method by using the `@app.post("/echo")` decorator on the function `echo()` that implements the echo feature.

Flask provides various decorators for each HTTP method (e.g. `@app.get(...)`, `@app.put(...)`) which are just a shorthand for the main decorator `@app.route("...", method=[METHOD1, METHOD2, ...])`.

Registered endpoint functions are stored internally in the app object in a Python Dictionary named `app.view_functions` which maps endpoint strings (e.g. `"/echo"`) to the function object implementing the functionality (e.g. `echo()` function object).

Code 5.3: Minimal Example of Request Hijacking in Flask

```
1  from flask import Flask
2
3  # Initialise the Flask application.
4  app = Flask(__name__)
5
6  @app.get("/foo")
7  def foo():
8      ...
9  # Further processing here...
10
11 def bar():
12     ...
13
14 # 'Hijack' the reqest by replacing the function entry in `view_fucnctions`.
15 app.view_functions["foo"] = bar
```

Line 7 Request data in Flask is accessed through a global proxy object `request`, which contains various data and metadata regarding the specific request, such as query parameters, JSON body, form data and header values.

For instance, the code listing shows the use of `request.get_data()` function, which returns the text of the request body, which could then be parsed into JSON manually or automatically through the use of other methods.

The usage of this proxy object will be discussed in more detail in §5.4.

Line 10-11 Once the `Flask` object is initialised and endpoint functions are registered, the application can be hosted in various ways, such as one shown in the listing, which shows a simple example of deploying the web API on `localhost` at `http://127.0.0.1:8080` in debug mode. Applications deployed in production must use a more sophisticated platform, which will be discussed in more detail at §6.1.

Flask supports many other features, such as Jinja2 templating engine for dynamically built HTML pages or serving static files, but for the purpose of this research, an understanding of the features listed above is sufficient for understanding the implementation of *Flask-Faasm*.

5.3.2 Hijacking the HTTP Requests to a Flask API

As explained above, Flask applications store the mapping from each endpoint to the function implementing the feature in a dictionary named `view_functions`. When a request is made to the Flask application, the app object queries this dictionary with the provided endpoint in the request to call the function that is registered by the decorator on the functions.

Fortunately, this dictionary is fully accessible to any user that can access the app object, and although Flask officially encourages against modifying this data structure for future flexibility [61], once the application is initialised and functions are registered, the dictionary will not be mutated further during the execution of the app instance. This means that the dictionary can be safely mutated to change what function will be called on receiving the HTTP request from the user on the specific endpoints, giving full freedom over the behaviour of the endpoint functions without having to modify the source of the Flask application in any way.

This process will be referred to as 'hijacking' the requests, as the dictionary is modified after the app is initialised and just before the app is deployed, and the replaced function can have arbitrary behaviour programmable in a Python function, effectively hijacking the behaviour of the endpoint functions after the app was initialised normally. An example of this is provided in Code Listing 5.3.

This undoubtedly requires serious care in ensuring that the function replacing the original endpoint function does not introduce breaking features. A few potential failure scenarios are outlined below, along with some design principles and potential solutions to prevent these failures:

1. The arguments to the functions do not match: If the replacing function has different input signature than the original endpoint function, the resulting API request will fail with `TypeError` raised.

   The replacing function should either have the same function signature as the original function, or it should use `*args` and `**kwargs` variable-length arguments to introduce flexibility in the function signature.

2. The output of the replacing function is not serialised to be compatible with what the HTTP response can support in its body: Arbitrary Python functions can return anything, including Python-specific values like `None` or complex objects like class instances. If the function returns those incompatible objects as return values, the Flask app will raise an exception and fail the request with 500 Internal Server Error.

   Care must be taken to ensure that the replacing function always returns numbers, (unicode) string, or raw bytes. Alternatively, serialisation libraries like `json` could be used to always serialise the return value of the function before actually returning the function, and handle the errors properly in the function itself.

Despite these risks, this insight is valuable as there are no alternative solutions that can dynamically modify the behaviour of the application without modifying the web application or even Flask library itself, which is both a significant development method for either the developer of the web application or the maintainer of products like *Flask-Faasm*, and goes against the requirements of being zero-cost in development outlined before in §4.1.

5.4 Interfacing Flask and `Faasm`

Flask-Faasm interfaces Flask and `Faasm` by hijacking the endpoint functions on the Flask app to invoke the uploaded function on `Faasm`. For each endpoint function, the function's source code is retrieved through the use of `inspect` module, and a function template file, defined in Code Listing 5.4, is used to inject the function's source, which is then uploaded to `Faasm` during the upload phase. The function template is defined such that the main `faasm_main()` function is dedicated to handle the serialisation and deserialisation of input and output data, and the function definition obtained from the Flask app is separated out to its own function, to be invoked separately by the main function. This structure enables high modularity, where the template file only requires the source of each entrypoint function to substitute into the template.

There are some additional information necessary for the interface to be fully functional, explanation on which are listed below. The following explanation may make references to specific lines in the Code Listing 5.4.

Function Inputs and Outputs The function arguments for the endpoint function must now be forwarded from the new endpoint entry function to the new *Faaslet* function. As

Code 5.4: Full Definition of the *Flask-Faasm* Function Template

```
import json
from pyfaasm.core import get_input_len, read_input, write_output


{__imports}


{__request_def}


request = None


{__lib_source}


{__function}


def faasm_main() -> int:
    # Read input data as bytes.
    input_data_bytes = read_input(get_input_len())

    # Parse the input as JSON.
    input_data = json.loads(input_data_bytes)

    # Obtain the function args and kwargs from the input data.
    args = input_data["func_args"]["args"]
    kwargs = input_data["func_args"]["kwargs"]

    # Initialize the global input request object from values of input.
    request_data = input_data["request_data"]
    global request
    request = Request(**request_data)

    # Call the provided function.
    output = {__function_name}(*args, **kwargs)

    # Format the output in JSON.
    output_data = json.dumps(output)

    # Encode the output in bytes and output the function result to Faasm.
    write_output(output_data.encode(encoding="utf-8"))

    return 0
```

discussed above, these arguments must be serialised before it is sent over via HTTP request body, as the *Faaslet* function accepts input with a call to `read_input()` that returns a byte stream of serialised value.

The arguments of Flask functions are used for capturing URL parameters, which are specified in the route definition (e.g. `"/user/<username>"`) and are automatically passed in as an argument to the endpoint function [61]. This means that the function arguments are simple types like integers and string, all of which are JSON serialisable.

Therefore, the function arguments could be correctly serialised and deserialised by capturing the possible inputs using the `*args` and `**kwargs` variable arguments and serialising them into JSON before sending it over to `Faasm`, where the *Faaslet* function can deserialise it (line 28-29) before forwarding the argument over the endpoint function (line 37).

Global Request Proxy Object As explained before in §5.3.1, the request data is accessed by the endpoint functions through the use of the global proxy object `request`. As it is provided as a global variable whose value can only be known during the invoke stage, the request data must be captured and serialised before it is sent over to `Faasm`, where the request object can be reconstructed with the provided data.

Flask-Faasm supports the global `request` object by first defining a new `Request` class with a subset of necessary fields and functions implemented, such as `args`, `form`, `get_json()`, and `method`. Then during the upload phase, the source of the class definition is included with the *Faaslet* function (line 8), and during the invoke phase, the entry endpoint function sends a JSON-serialised version of the request data with the function arguments as input, and the request data is received by the *Faaslet* function and used to initialise the `Request` object as a global variable with name `request` (Line 32-34) which could be then used by the endpoint function identically to the Flask Request proxy object.

There were several alternatives to this approach. One was to add a `request` parameter to the function arguments. This approach would necessitate a change to the endpoint function's source code itself, specifically its function signature to add a new parameter, which requires an unnecessary parsing phase to parse the source code and introduce a potential for subtle bugs with processing the source itself. It would also still require the request data to be transferred and constructed as a `Request` object, which means that overall this leads to little benefit other than avoiding the use of global variables, which the original `request` object already is, and the script is self-contained without a potential for external modules to modify this global variable anyway.

Another alternative approach was to send the original Flask's `Request` object's source code, which does reduce the development efforts to implement a new class that emulates the behaviour of the original `Request` object. However, not only the original `Request` object contains strictly unnecessary metadata for implementing web APIs, unnecessarily increasing the size of the overall script to be executed, it also introduces a complex library dependencies for implementing the `Request` object itself, which in turn would also either have to be added to the script or installed in the WebAssembly CPython runtime on `Faasm`. Introducing a custom-built `Request` class to be used in the script helps with minimising the *Faaslet* function's size, which helps reduce the overall latency of the function execution.

Imports The functions implementing any real-world functionalities inevitably use library code defined in some module. As explained in §4.1.1, although it is possible to add support for individual third-party libraries on a case-by-case basis, *Flask-Faasm* currently does not support the entry functions importing third-party libraries[1]. However, any Python

[1] An exception to this is Flask, as a Flask app necessarily has to import the library. However, any Flask imports will be deleted from the list of import statements in the `Faasm` script as it is unnecessary for the function's implementation itself.

standard libraries are supported as it is included as part of the CPython runtime. It also can support project-internal modules, but they work differently to the standard library imports, which will be explained below.

Python function objects do not have an attribute that indicates which functions/variables/modules are imported in its scope, as it either needs to be dynamically determined on function execution, or the function's source must be parsed to determine which symbols are used and which of those symbols are imported from other modules. As both of these strategies require unnecessarily complex processing, *Flask-Faasm* instead takes another approach if determining which libraries are imported at a per-file basis rather than per-function.

Python's `ast` module provides a visitor class that, given a parsed Python AST (Abstract Syntax Tree) object, performs recursive visits to each node and performs specific actions depending on the type of the node e.g. an expression, symbol, etc. Using this visitor class, an `ImportVisitor` can be defined to perform actions for the two types of import statements: import statements and from-import statements. The two import statements have the following BNF:

```
<import>        ::= import <module> [as <alias>] (, <module> [as <alias>])*
<from-import>   ::= from <module> <import>
```

The visitor parses the AST of the source file that contains each endpoint function, and it finds all import statements in the file, and then stores the module names (along with its aliases) that are used in the module. Once this information is collected by the visitor, the import statements are reconstructed as code, which is then inserted into the function template (Line 5).

As the project-internal library code works differently in *Flask-Faasm*, those import statements are excluded i.e. relative imports of the form `from . import ...` and relative submodule imports of the form `from .submodule import ...`.

Project-Internal Library Code A true support for project-internal library code would involve parsing the endpoint function's code to determine which internal modules are used, then recursively retrieving the source of the internal modules to create a single block of code, similar to how C/C++ handles 'imports' with the preprocessor macro `#include "module.h"`. Although it is certainly possible to implement this feature using the visitor class of the `ast` module, for the purpose of this research, the product does not need to support this feature, as all evaluation will be done on a custom-built web application suited specifically for benchmarks. It involves a substantial development effort to build this feature whilst it not being relevant for the core of the experiments to be run in evaluation.

However, having some separation of library code from the endpoint functions is a desirable feature for the convenience of implementing the endpoint functions. Therefore, there is some restricted form of project-internal module support. To avoid having to parse the function's source code, all internal library code for the endpoints have a fixed structure. All library code for each function must be contained in a folder named `'lib/'`, with the name of each library file being the name of the function e.g. for an endpoint function `echo()` the library file associated with the function would be stored in the file `'lib/echo.py'`. Then, during the upload phase, the template file is loaded with the library code whose location is known and fixed (Line 14).

Chapter 6

Evaluation

This chapter introduces the strategies for evaluating the performance of *Flask-Faasm*, discussing the experimental setup and benchmarks run, along with discussing the results of the experiments to quantitatively and qualitatively evaluate *Flask-Faasm* in relation to a 'native' deployment on a standard WSGI HTTP server.

6.1 Experimental Setup

There are several interacting components for running the experiments.

Benchmark Web API A Flask app implementing a set of benchmark functions is created to be used for quantifying the performance of the endpoint functions on *Flask-Faasm* compared to a 'native' deployment. Details of the APIs are explained below in §6.1.3.

`Faasm` Instance A `Faasm` instance is created on the DoC Research Cluster the experiment is run on. The instance is created using `docker-compose` to create a cluster of nodes in a container each handling various tasks like upload and invoke. It is important to recognise that while the nodes are running on top of container nodes, the *Faaslet* functions themselves do not run on its own containers as each worker instance on a container is not tied to any *Faaslet* functions, meaning that it is just a worker node that is fully independent from provisioning and deprovisioning of the functions.

Web Server Instance for Flask app In addition to the `Faasm` instance, a web server instance should also exist that runs the Flask app, whether acting as a gateway frontend for `Faasm` with *Flask-Faasm*, or running natively on its own.

6.1.1 Web Server Gateway Interface (WSGI)

As defined in PEP 3333 [62], the Web Server Gateway Interface (WSGI) defines a standard interface between Python web applications and web servers. As a protocol, WSGI outlines how web frameworks in Python should structure their web applications to be compatible with WSGI-compliant servers, which makes it more convenient to deploy web applications without requiring setting up and running a dedicated web server. All major Python web application frameworks including Flask implement this interface for their web applications.

Flask web applications intrinsically support a simple development web server that can be launched by calling the `app.run()` method. This simple web server is WSGI-compliant, but it should not be used outside of development environments as it was intentionally designed to not be production-ready, but rather support development-friendly features

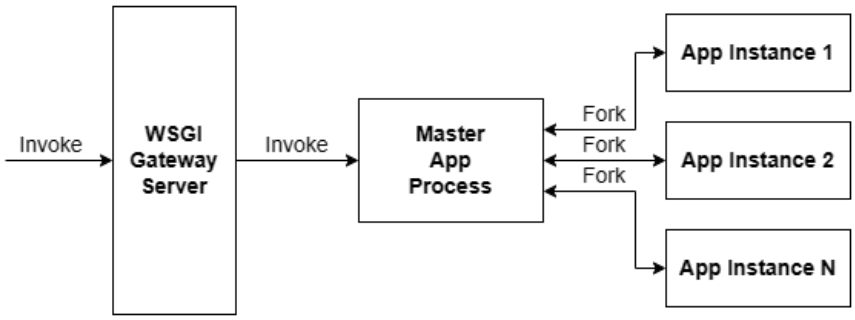


Figure 6.1: Pre-Fork Model Architecture

like hot reloading and single-threaded request handling [61]. Flask's web server lacks necessary features for production deployment, including a production-grade concurrent request handling, which prevents the experiments from producing a proper set of results.

Therefore, a dedicated WSGI web server needs to be used to host the web APIs to run the experiments on. *Gunicorn* [63]–or 'Green Unicorn'–is a commonly used WSGI HTTP server that boasts a high-performance architecture, providing robust concurrency management without sacrificing simplicity and ease of use. It is equipped to handle a significant amount of traffic and is known for its ability to manage multiple worker processes simultaneously. *Gunicorn* employs a pre-fork worker model for managing its worker processes. Figure 6.1 shows how pre-fork worker model works. During initialisation of a server using pre-fork worker model, a master app process is created that pre-initialises the web application by loading the runtime and initialising the app. On invocation request, it hits a WSGI Gateway Server that handles the request by first forking the pre-initialised master app process to create an individual worker process and then running the app on the worker app instance to handle the incoming request. This model is reminiscent to how a monolithic web server works, except some optimisation is made by creating a master app process that pre-initialises the web application instance.

6.1.2 Baseline 'Native' Setup

To contrast with the serverless FaaS setup provided with *Flask-Faasm*, a baseline setup is necessary to show how the performance of the *Flask-Faasm* setup contrasts with a 'native' monolithic deployment. The setup is very similar to the setup used for the *Flask-Faasm* setup, except that the Flask app is unmodified before deployment into *gunicorn*, meaning that the requests are handled directly on the forked app instance, which hosts an entire copy of the web application in its process.

The baseline setup (and the gateway Flask app for the *Flask-Faasm* setup) uses 41 workers, which is recommended by the developers of Gunicorn as it is exactly $2 \times num - cores + 1$, and there are 20 cores in the cluster machine used [64].

6.1.3 Benchmark Web API

The benchmark functions that are deployed to the web API is based on the Python Performance Benchmark Suite [12], often referred to as PyPerformance. PyPerformance provides a diverse set of benchmarks, including low-level micro-benchmark functions, which focus on single operations or functions, and high-level applications, which evaluates the performance of real-world applications and libraries. This diversity that PyPerformance provides allows for a comprehensive evaluation of Python code performance across a variety of tasks.

PyPerformance is also considered to be the standard benchmark suite for testing the performance of Python-related systems and are used in various research, including the original `Faasm` paper [5]. This standardised approach is essential for benchmarking web API and the relevant systems, as it provides a consistent and reliable set of performance metrics to compare with other web systems. Therefore, the benchmark web API will adapt the PyPerformance benchmark suite to obtain a core set of functions that run the benchmark functions, but adapt them so that they are hosted on a Flask web application and can be invoked as an API by its own dedicated endpoint.

A subset of benchmarks provided by PyPerformance will be used, where the subset will show a representative sample of all benchmarks available in PyPerformance. One exception to this are any benchmark functions that depend on external libraries are excluded as `Faasm` currently does not easily support installing external libraries to the WebAssembly compiled CPython runtime. A full list of benchmarks used, along with their explanations are provided in the appendix.

6.2 Experiment 1: Comparing the Latency of *Flask-Faasm* vs. 'Native' Deployment

6.2.1 Motivation

Compared to the execution of a native baseline, *Flask-Faasm* inherently comes with an overhead of running a function in a remote location and associated processes, such as the latency in communication and the time it takes for serialising and deserialising function inputs and outputs, meaning that the operations executed in a *Flask-Faasm* system is a strict superset of the operations executed in a native baseline. However, if running the benchmark functions on *Flask-Faasm* leads to an overhead less than an order of magnitude difference, it would be a considerable improvement over existing serverless systems because, as outlined in §2.3.2, they have a prohibitively expensive overhead that comes from utilising hardware-based isolation solutions like containers and VMs.

6.2.2 Methodology

1. Choose a benchmark function in the benchmark web API.
2. Run the function starting from 1 invocations per second, and repeat for 5 times, capturing the latency of overall function execution each repeated invocations.
3. Test various values of the number of invocations: `[1, 5, 10, 20, 30, 40, 50, 75, 100]`.
4. Obtain the median latency for each invocation per second.
5. Repeat 1-4 for both native execution and *Flask-Faasm*.
6. Repeat 1-5 for all benchmark functions available.
7. Calculate the relative latency increase in *Flask-Faasm* for each benchmark function and invocations per second, and take an arithmetic mean of relative latency for all functions for each invocations per second.

6.2.3 Results

Figure 6.2 shows the relative latency of function execution on *Flask-Faasm* compared to native execution for an increasing number of invocations. The results show that the

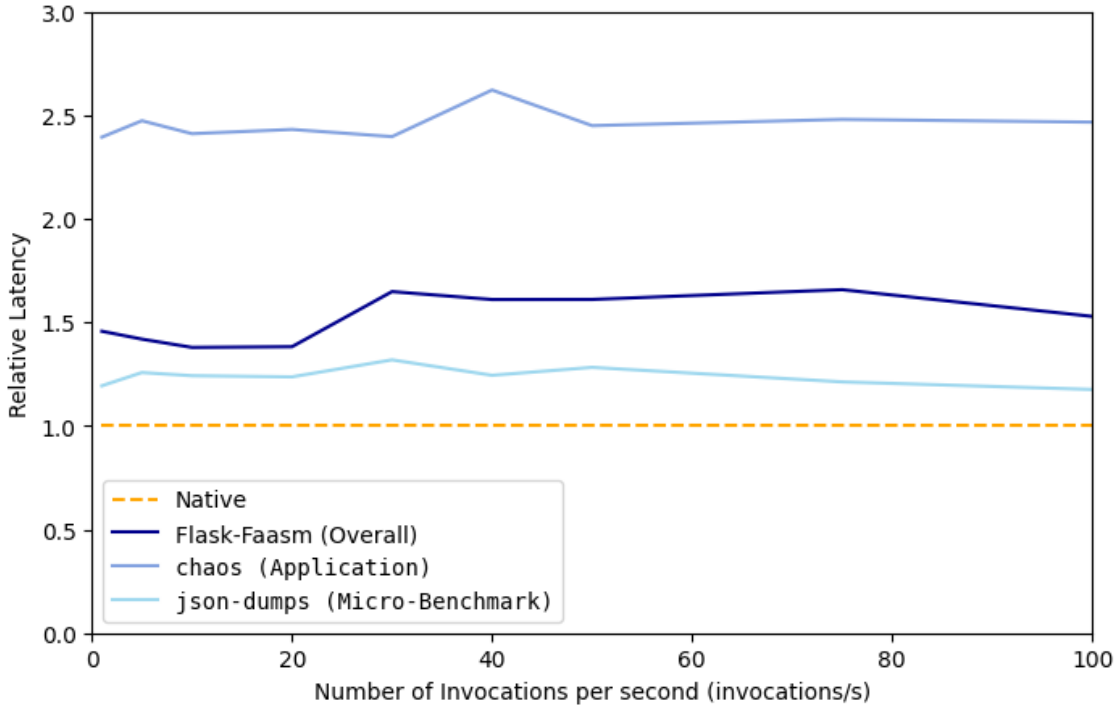


Figure 6.2: Relative Latency of Function Execution in *Flask-Faasm* vs Native Deployment

average latency overhead of *Flask-Faasm* across all benchmarks is around 1.5 times the execution latency of the native execution. The relative latency difference stays near flat, as the pattern of increase in the latency of *Flask-Faasm* and native execution is relatively similar. An average latency overhead of 1.5 times native execution shows that the latency overhead of *Flask-Faasm* overall is very small compared to container-based FaaS solutions that show an order of magnitude or more in latency, even for smaller number of invocations. This shows that the overhead that is associated with forwarding the request to `Faasm` and (de)serialising data is not too significant compared to the actual runtime of the functions. It is important to note that this is an idealised scenario where for *Flask-Faasm* both the Flask gateway app and the `Faasm` instance lives on the same cluster, which means that if the two processes are located in separate servers in distinct regions, the Round-Trip Time (RTT) between the two processes will likely add to the overall relative latency. The experiment was conducted in this fashion to more accurately judge the overhead that directly comes from *Flask-Faasm* and its data serialisation and the overhead coming from using *Faaslet* functions on `Faasm`.

It is interesting to note that the relative latency for 'Application'-type benchmark functions are significantly different to 'Micro-Benchmark'-type functions. Figure 6.2 also shows an example for each type of benchmark functions, where `chaos` is an example of an application-type benchmark function, and `json-dumps` is an example of a micro-benchmark-type benchmark function. This shows that the relative latency for larger applications that have a longer runtime actually has a much more significant relative latency compared to smaller ephemeral functions that have a shorter runtime. This suggests that the latency overhead from dynamically interpreting the uploaded function script on `Faasm` vastly outweighs the latency overhead that comes from input and output data serialisation and deserialisation.

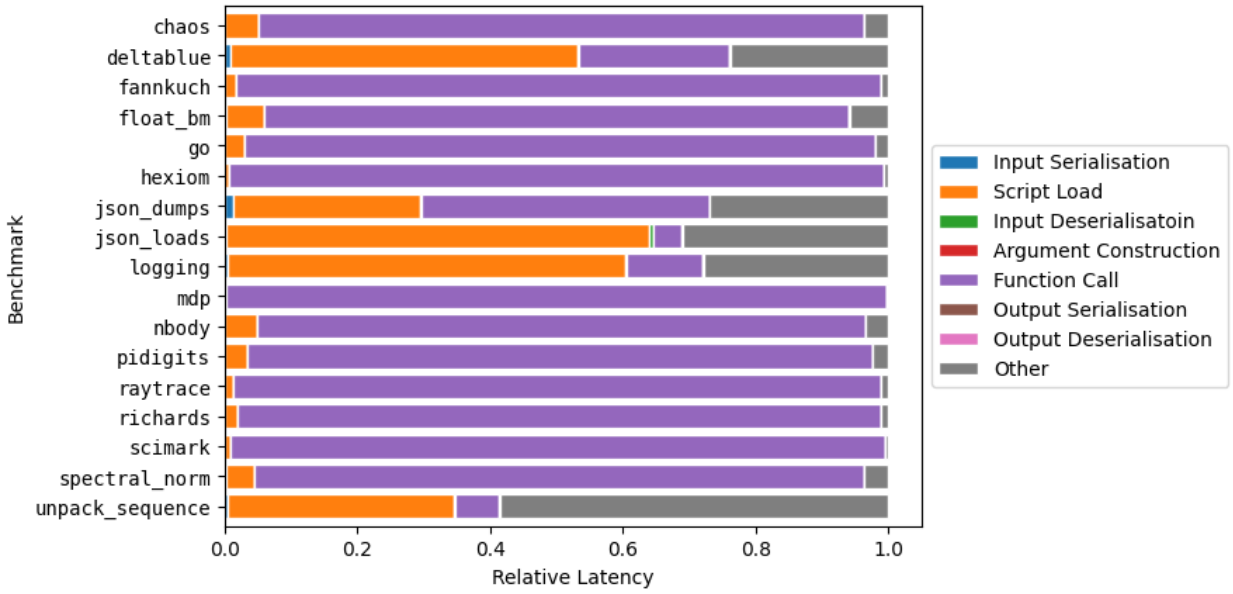


Figure 6.3: Relative Latency Distribution for *Flask-Faasm* Function Calls

6.3 Experiment 2: Examining the Source of Overheads in *Flask-Faasm* Function Execution

6.3.1 Motivation

As outlined above, although *Flask-Faasm* shows a significantly less overhead than existing container-based serverless solutions, it still shows a considerable overhead when compared to a native deployment, which is equivalent to a monolithic solution, especially for fewer invocations per second. This overhead may come from the serialisation overhead, or a lack of pre-initialising or caching the function instances. Through this experiment, it is possible to quantify exactly what causes this latency and figure out where improvements and optimisations could be made.

6.3.2 Methodology

Overall, each benchmark functions is run 10 times to collect the latency measurements. For this experiment, the template script file will contain various calls to the function `perf_counter()`, which measures the current clock time, including time spent on other processes and sleep, in the highest precision available in the system. The calls to `perf_counter()` are placed in various locations, such as at the beginning and end of each script (to measure the time it takes to load the script), and in between the serialisation and deserialisation of input and output data in both Flask entrypoint function and *Faaslet* function on `Faasm`. The times measured on `Faasm` will be given as a return value, and the time differences will be calculated to give a measure of latency.

6.3.3 Results

Figure 6.3 shows the relative distribution of overall end-to-end latency of function execution in *Flask-Faasm* for the benchmark functions. The figure shows that the latency overhead from loading the script into a *Faaslet* dominates the latency (other than function execution time itself) for all functions. The distribution of latency vs function execution time is also highly dependent on the nature of functions, where more ephemeral functions with a very

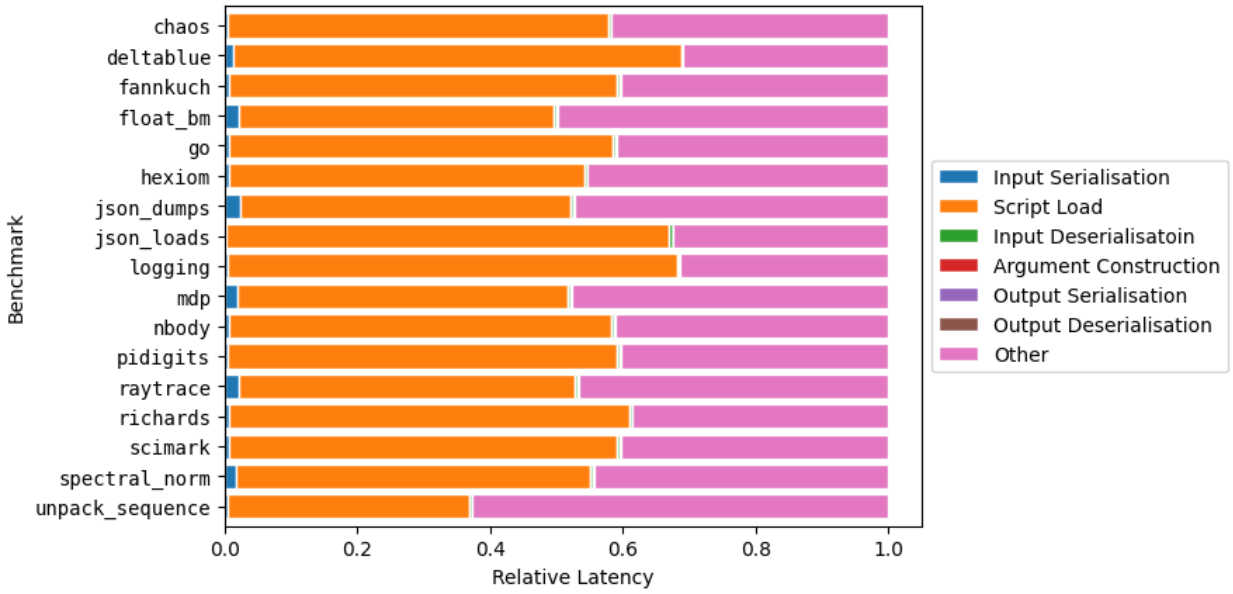


Figure 6.4: Relative Latency Distribution without Function Runtime

short runtime, such as `json_loads` with a runtime of around 50ms, have the script load time take up nearly 50-60% of the overall runtime, whereas larger application-level functions, such as `mdp` with a runtime of around 10s, have its actual function execution time dominate the overall runtime of the functions. It is also clear that the latency from *Flask-Faasm* for the majority of functions take up less than 20% of the overall runtime, which shows that *Flask-Faasm* already demonstrates that it has a significantly less overhead compared to a native execution, when compared to other existing serverless solutions that use hardware-based isolation mechanisms.

To have a more in-depth look at the latency profile, Figure 6.4 is shown where the relative latency is recalculated by excluding the runtime of the function itself. Again it is clear that the script load time dominates the latency for all functions, where for most functions it takes up over 50% of the runtime. It is also evident that the input serialisation also has a small overhead of 1-2%, whereas other serialisation and deserialisation operations have a negligible overhead. This is likely due to the fact that the input arguments, including request data and function arguments are essentially empty, where minimal amount of data is transferred. This may mean that with a larger input, the latency characteristics may change significantly where the serialisation may take up more proportion in the relative latency distribution. However, as most web APIs used to implement the most common applications like e-commerce do not transfer much data from its clients to servers, it is unlikely that the serialisation/deserialisation overhead will pose a significant problem.

It should also be noted that the 'Other' latency also takes up a significant proportion in relative latency. The Other latency constitutes all components outside of *Flask-Faasm*'s control, which may involve function allocation to workers in `Faasm`, the RTT involving the client, Flask, and `Faasm`, and any other potential delays in the overall runtime of the functions themselves. It is likely that there are limited rooms for improvement with the 'Other' category of latency as it is outside of the control of the implementation of *Flask-Faasm* and may involve directly modifying either `Faasm`, Flask library, or the web application itself, which is beyond the scope of this research.

Chapter 7

Conclusion

Overall, this research has successfully demonstrated that it is possible to create an adapter interface between web applications developed using standard web frameworks like Flask and a serverless runtime like `Faasm` with API exposed to interact with it, with zero modifications made to either the web application code or the serverless runtime (Chapter 5). This research also has shown that the overhead in latency caused by *Flask-Faasm* is around 1.5-2 times of a native execution, which is considerably smaller compared to other commercial serverless solutions which utilise the standard hardware-based isolation mechanisms like containers and VMs that are known to exhibit an overhead in orders of magnitude higher than the function's runtime (§6.2).

This research also considers the latency profile of functions on *Flask-Faasm* and identifies which components in the critical loop of the function execution shows the most significant overhead, identifying potential improvements that could be made to the system (§6.3). Although the result shown paints a promising picture of using alternative software-based isolation mechanisms in a multi-tenant serverless system, it also does demonstrate that the latency of the overall function execution is still considerably large especially for ephemeral functions, which constitute the majority of web API functions. The research is also conducted in an idealised environment, where the gateway web application instance and the serverless `Faasm` instance is hosted on the same cluster, and the input size is very small. This, however, should not significantly affect the results, as the reason why the components were placed on the same cluster was to minimise the effect of the latency overhead from the RTT between components which can be highly variable and hard to control, and the input size is usually small for most web APIs implementing common web applications like e-commerce.

7.1 Future Work

There are further opportunities for extending the research, along with opportunities for improving *Flask-Faasm* itself.

Deploy *Flask-Faasm* on IaaS Platforms To test the system in a scenario that more closely resembles real-life, the system, including Flask gateway app, *Flask-Faasm*, and a `Faasm` instance could be deployed on a commercial IaaS platform like AWS or Azure. Commercial IaaS solutions are preferred as most now also provide their own proprietary FaaS serverless platforms, meaning that the test could be performed with existing serverless solutions that use container-based isolation mechanisms by deploying the same benchmark application on *Flask-Faasm* running on IaaS platform and on the IaaS provider's serverless platforms and comparing the latency.

Introduce *Proto-Faaslets* for Pre-Initialising *Faaslet* Functions As outlined in §6.3, the majority of latency overhead comes from the script loading time, which is caused by dynamically interpreting the uploaded script file during the invoke phase. Section §2.5.1 outline how *Faaslets* have a simple memory layout, leading to the development of *Proto-Faaslets* that provide a snapshot of an ahead-of-time pre-initialised *Faaslet* with its language runtime and libraries initialised and ready to run functions. As *Proto-Faaslets* can be initialised in hundreds of microseconds, it will provide a significant improvement to the latency of invoking *Flask-Faasm* functions, as it currently has a latency overhead in the tens of milliseconds for loading the scripts.

The reason why `Faasm` currently is unable to support this is because *Proto-Faaslets* for Python functions are difficult to implement. Currently, Python functions are executed by loading an uploaded Python script file and invoking a centralised 'CPython function' at the URI `/python/py_func`. This makes it hard to create a dedicated snapshot of a function for individual Python functions. A change in `Faasm` could be made, where Python functions also get their own dedicated top-level URI with its function capturing the whole state, including its own CPython runtime.

A New Serialisation Method Although the latency resulting from serialisation is much lower than the latency from script loading, it would still be beneficial to introduce a more efficient form of serialisation than JSON, preferably in a binary format. `Faasm` could be modified to expose a separate RPC (Remote Procedure Call) API to establish a connection between clients and itself, establishing a connection for transferring bytes while also eliminating the overhead of handshakes and transferring unnecessary metadata like HTTP headers.

A promising candidate for serialisation is Google's Protocol Buffers ('Protobufs'). Not only does it provide a compact and standard binary message format, it also provides support for multiple languages including C++ and Python for creating and interpreting Protobufs, making it easy to define a data format for each function. A way to dynamically generate protocol buffer definitions should be implemented, but considering that the request data for all functions remains constant, and how the function signature with type hints provides arguments along with their types, it should not pose too difficult of a challenge to create the system. `Faasm` itself must also be modified to be able to support this feature.

Develop and Deploy a Commercial-Level Web API for Testing *Flask-Faasm* It would also be beneficial to develop a real-life web API for testing the system to more accurately judge the performance overhead associated with real-life workloads. The PyPerformance Benchmark functions do provide a very good overview of the general performance characteristics for various scenarios, but it may not fully represent a realistic workload that web APIs typically provide.

Proper Management of Project-Internal Libraries As outlined in §5.4, although not necessary for the scope of this research, it would be beneficial to provide a proper support for project-internal library code, from parsing the web application's source code to determine which internal modules used, to recursively retrieving the source of all imported project-internal modules to create a single block of library code that can be pasted onto the template function file, similar to how C/C++ handles imports using the `#include` preprocessor macro.

Chapter 8

Ethical Considerations

There are mostly no concerns about the misuse of the results of this research. Beyond a superficial argument about how the following products could technically be deployed on a serverless platform, there is no realistic potential that this research could be used for malevolent/criminal/terrorist abuse, nor involve the development of technologies or the creation of information that, if misapplied, could have severe negative impacts on human rights standards, such as privacy, stigmatization, and discrimination. It also does not involve information on/for the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery.

However, there are some concerns to be made regarding infrastructural vulnerability or cybersecurity that may have the potential for terrorist or criminal abuse. `Faasm` uses software-based isolation mechanisms like *software-fault isloation* (SFI) of *WebAssembly* to isolate the memory between the function invocations [5]. There are serious concerns to be made regarding arbitrary user code execution on multi-tenant systems like a serverless platform, and using software-based isolation mechanisms rather than containers–which do have more concrete safety measures at a kernel level–should be approached with caution as protections at a software level have more potential vulnerabilities [18] and are more easily exploited by malicious parties like terrorists or criminals. However, this is not a major concern for this research as currently there are no studies that suggest that SFI on *WebAssembly* has security vulnerabilities that violate the isolation guarantees required for a serverless system.

As to be discussed below, there are no legal concerns regarding the use or production of goods or information concerning data protection or other legal implications. However, there are nuanced discussions to be made regarding the use and production of software for which there are copyright or licensing implications. As it stands, no software or library has been used that has copyright implications beyond using the names of trademarked products such as AWS and Docker. However, such products and/or other projects do have various open-source licenses attached that may play a role in this research. For instance, this research will make heavy use of Docker containers. Docker containers are notoriously hard to be licensed, as all individual software packaged in the container may have its own license each with its own separate clauses that have to be complied with [65]. Even the most permissive licenses like the MIT license [66] require that the "copyright notice and [the] permission notice shall be included in all copies or substantial portions of the Software," which would have to be complied with. Also, if software licensed under the GNU General Public License [67] were to be used, the resulting product must be licensed under the GPL license (unless the software is licensed under LGPL), and the source code of the product must be available at any time on request. Currently, the product uses no software licensed under GPL, and the source code of the product of this project has been open-sourced regardless, meaning that complying with the GPL license should not pose a

major concern.

This research does not involve any living organisms, including human participants and animals.

This research does not involve the collection and/or processing of existing or newly-generated personal data, including sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction), genetic information, and data useful for tracking or observation of participants. This means that this research is compliant with the GDPR legislated under both the EU and the UK. It is important to note that, although during the evaluation process, some networking data–such as IP address and MAC address–might intentionally or unintentionally have been collected, these data would not be associated with any participants, but rather information associated with servers or container instances. Also, note that there were no participants involved in this research to collect the data from.

This research has no specific focus or relevance with developing or low and/or lower-middle income countries, nor puts any individuals in such countries at risk.

No elements that may cause harm to either nature–including the environment, animals, or plants–or humans–including project staff–have been used for this research.

There are no considerations to be made in regards to whether this research has a potential for a 'dual-use' as defined by the EU. Due to this research's explicit focus on web backend applications on cloud platforms, it is highly unlikely that this research has any potential for affecting military applications or ethics, nor have a potential use case that will require any export licenses from the appropriate jurisdictions. The deployment of web applications on commercially-available serverless platforms has an exclusive civilian focus and the military is highly unlikely to be involved.

Bibliography

[1] Villamizar M, Garcés O, Ochoa L, Castro H, Salamanca L, Verano M, et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. Service Oriented Computing and Applications. 2017;11(2):233-47.

[2] Lynn T, Rosati P, Lejeune A, Emeakaroha V. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE; 2017. p. 162-9.

[3] Shahrad M, Balkind J, Wentzlaff D. Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture; 2019. p. 1063-75.

[4] Fortuna Data. How does a hypervisor work?; 2022. [Online]. Available from: https://www.data-storage.uk/wp-content/uploads/2022/03/hypervisor.jpg [Accessed 4th June 2023].

[5] Shillaker S, Pietzuch P. Faasm: Lightweight isolation for efficient stateful serverless computing. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20); 2020. p. 419-33.

[6] Shillaker S. python/func/python/echo.py at main; 2021. [Online]. Available from: https://github.com/faasm/python/blob/main/func/python/echo.py [Accessed 12th June 2023].

[7] Fowler M, Lewis J. Microservices; 2014. [Online]. Available from: https://martinfowler.com/articles/microservices.html [Accessed 23rd January 2023].

[8] Hellerstein JM, Faleiro J, Gonzalez JE, Schleier-Smith J, Sreekanti V, Tumanov A, et al. Serverless computing: One step forward, two steps back. arXiv preprint arXiv:181203651. 2018.

[9] Amazon Web Services. AWS Lambda Developer Guide; 2023. [Online]. Available from: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html [Accessed 23rd January 2023].

[10] Microsoft. Azure Functions documentation; 2023. [Online]. Available from: https://learn.microsoft.com/en-us/azure/azure-functions/ [Accessed 23rd January 2023].

[11] Pallets. Flask Documentation; 2023. [Online]. Available from: https://flask.palletsprojects.com [Accessed 26th January 2023].

[12] Stinner V. The Python Performance Benchmark Suite; 2017. [Online]. Available from: https://pyperformance.readthedocs.io/ [Accessed 26th January 2023].

[13] Serrano N, Gallardo G, Hernantes J. Infrastructure as a service and cloud technologies. IEEE Software. 2015;32(2):30-6.

[14] Amazon Web Services. What is Amazon EC2?; 2023. [Online]. Available from: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html [Accessed 23rd January 2023].

[15] Hendrickson S, Sturdevant S, Harter T, Venkataramani V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Serverless Computation with {OpenLambda}. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16); 2016. .

[16] Lloyd W, Ramesh S, Chinthalapati S, Ly L, Pallickara S. Serverless computing: An investigation of factors influencing microservice performance. In: 2018 IEEE international conference on cloud engineering (IC2E). IEEE; 2018. p. 159-69.

[17] Rosen R. Resource management: Linux kernel namespaces and cgroups. Haifux, May. 2013;186:70.

[18] McGrath G, Brenner PR. Serverless computing: Design, implementation, and performance. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE; 2017. p. 405-10.

[19] Docker Inc . Docker Reference documentation; 2023. [Online]. Available from: https://docs.docker.com/reference/ [Accessed 23rd January 2023].

[20] Sill A. The design and architecture of microservices. IEEE Cloud Computing. 2016;3(5):76-80.

[21] Cloud Native Computing Foundation. CNCF WG-Serverless Whitepaper v1.0; 2018. [Online]. Available from: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf.

[22] Google. Cloud Functions; 2023. [Online]. Available from: https://cloud.google.com/functions/ [Accessed 23rd January 2023].

[23] IBM. Getting started with IBM Cloud Functions; 2023. [Online]. Available from: https://cloud.ibm.com/docs/openwhisk/index.html [Accessed 23rd January 2023].

[24] Apache Software Foundation. Apache Openwhisk Documentation; 2023. [Online]. Available from: https://openwhisk.apache.org/documentation.html [Accessed 23rd January 2023].

[25] Eismann S, Scheuner J, Van Eyk E, Schwinger M, Grohmann J, Herbst N, et al. The state of serverless applications: Collection, characterization, and community consensus. IEEE Transactions on Software Engineering. 2021;48(10):4152-66.

[26] Red Hat. What is an API?; 2022. [Online]. Available from: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces [Accessed 23rd January 2023].

[27] Red Hat. What is a REST API?; 2020. [Online]. Available from: https://www.redhat.com/en/topics/api/what-is-a-rest-api [Accessed 23rd January 2023].

[28] Amazon Web Services. What is RESTful API?; 2023. [Online]. Available from: https://aws.amazon.com/what-is/restful-api/ [Accessed 23rd January 2023].

[29] Ivan C, Vasile R, Dadarlat V. Serverless computing: An investigation of deployment environments for web apis. Computers. 2019;8(2):50.

[30] Eivy A, Weinman J. Be wary of the economics of "serverless" cloud computing. IEEE Cloud Computing. 2017;4(2):6-12.

[31] Retter M. Serverless Case Study - Netflix; 2020. [Online]. Available from: https://dashbird.io/blog/serverless-case-study-netflix/ [Accessed 23rd January 2023].

[32] Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient software-based fault isolation. In: Proceedings of the fourteenth ACM symposium on Operating systems principles; 1993. p. 203-16.

[33] Bressoud TC, Schneider FB. Hypervisor-Based Fault Tolerance. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. SOSP '95. New York, NY, USA: Association for Computing Machinery; 1995. p. 1–11. Available from: https://doi.org/10.1145/224056.224058.

[34] W3C. WebAssembly; 2023. [Online]. Available from: https://webassembly.org/ [Accessed 23rd January 2023].

[35] W3C. WebAssembly Specification; 2023. [Online]. Available from: https://webassembly.github.io/spec/core/ [Accessed 23rd January 2023].

[36] Watt C. Mechanising and verifying the WebAssembly specification. In: Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs; 2018. p. 53-65.

[37] Fu W, Lin R, Inge D. Taintassembly: Taint-based information flow control tracking for webassembly. arXiv preprint arXiv:180201050. 2018.

[38] Lehmann D, Pradel M. Wasabi: A framework for dynamically analyzing webassembly. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems; 2019. p. 1045-58.

[39] LLVM Team. The LLVM Compiler Infrastructure; 2023. [Online]. Available from: https://llvm.org/ [Accessed 23rd January 2023].

[40] W3C. WASI Design Principles; 2023. [Online]. Available from: https://github.com/WebAssembly/WASI/#capability-based-security [Accessed 23rd January 2023].

[41] Agache A, Brooker M, Iordache A, Liguori A, Neugebauer R, Piwonka P, et al. Firecracker: Lightweight virtualization for serverless applications. In: 17th USENIX symposium on networked systems design and implementation (NSDI 20); 2020. p. 419-34.

[42] Cloud Hypervisor. Cloud Hypervisor: Run Cloud Virtual Machines Securely and Efficiently; 2021. [Online]. Available from: https://www.cloudhypervisor.org/ [Accessed 8th June 2023].

[43] Jonas E, Pu Q, Venkataraman S, Stoica I, Recht B. Occupy the cloud: Distributed computing for the 99%. In: Proceedings of the 2017 symposium on cloud computing; 2017. p. 445-51.

[44] Barcelona-Pons D, Sánchez-Artigas M, París G, Sutra P, García-López P. On the faas track: Building stateful distributed applications with serverless architectures. In: Proceedings of the 20th international middleware conference; 2019. p. 41-54.

[45] Cloudflare. Cloudflare Workers; 2021. [Online]. Available from: https://developers.cloudflare.com/workers/ [Accessed 8th June 2023].

[46] Google. v8::Isolate Class Reference; 2021. [Online]. Available from: https://v8docs.nodesource.com/node-0.8/d5/dda/classv8_1_1_isolate.html [Accessed 8th June 2023].

[47] Desi Labs. Introducing Krustlet, the WebAssembly Kubelet; 2020. [Online]. Available from: https://deislabs.io/posts/introducing-krustlet/ [Accessed 8th June 2023].

[48] Belay A, Bittau A, Mashtizadeh A, Terei D, Mazières D, Kozyrakis C. Dune: Safe user-level access to privileged {CPU} features. In: Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12); 2012. p. 335-48.

[49] Litton J, Vahldiek-Oberwagner A, Elnikety E, Garg D, Bhattacharjee B, Druschel P. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In: OSDI; 2016. p. 49-64.

[50] Howell J, Parno B, Douceur JR. How to Run {POSIX} Apps in a Minimal Picoprocess. In: 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13); 2013. p. 321-32.

[51] Du D, Yu T, Xia Y, Zang B, Yan G, Qin C, et al. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems; 2020. p. 467-81.

[52] Firecracker. Firecracker Snapshotting; 2023. [Online]. Available from: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md/ [Accessed 8th June 2023].

[53] Ustiugov D, Petrov P, Kogias M, Bugnion E, Grot B. Benchmarking, analysis, and optimization of serverless function snapshots. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems; 2021. p. 559-72.

[54] Google. Overview of memory management; 2023. [Online]. Available from: https://developer.android.com/topic/performance/memory-overview#SharingRAM [Accessed 8th June 2023].

[55] Oakes E, Yang L, Zhou D, Houck K, Harter T, Arpaci-Dusseau A, et al. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18); 2018. p. 57-70.

[56] Akkus IE, Chen R, Rimac I, Stein M, Satzke K, Beck A, et al. {SAND}: Towards {High-Performance} Serverless Computing. In: 2018 Usenix Annual Technical Conference (USENIX ATC 18); 2018. p. 923-35.

[57] Django Software Foundation. Django documentation; 2023. [Online]. Available from: https://docs.djangoproject.com [Accessed 26th January 2023].

[58] Corporation M. Evolution of HTTP; 2023. [Online]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP [Accessed 12th June 2023].

[59] Faasm. python/pyfaasm at main; 2023. [Online]. Available from: https://github.com/faasm/python/tree/main/pyfaasm [Accessed 12th June 2023].

[60] Shillaker S. Host interface — Faasm documentation; 2022. [Online]. Available from: https://faasm.readthedocs.io/en/latest/source/host_interface.html#function-input-output-and-chaining [Accessed 12th June 2023].

[61] Pallets. API — Flask Documentation (2.3.x); 2023. [Online]. Available from: https://flask.palletsprojects.com/en/2.3.x/api [Accessed 12th June 2023].

[62] Eby PJ. PEP 3333 – Python Web Server Gateway Interface v1.0.1; 2010. [Online]. Available from: https://peps.python.org/pep-3333/ [Accessed 15th June 2023].

[63] Gunicorn. Gunicorn - Python WSGI HTTP Server for UNIX; 2009. [Online]. Available from: http://gunicorn.org/ [Accessed 15th June 2023].

[64] Gunicorn. Design — Gunicorn 20.1.0 documentation; 2021. [Online]. Available from: https://docs.gunicorn.org/en/stable/design.html#how-many-workers [Accessed 15th June 2023].

[65] amon. Copyright when creating a Dockerfile & image based on a FOSS-licensed project; 2020. [Online]. Available from: https://opensource.stackexchange.com/a/9983 [Accessed 25th January 2023].

[66] MIT. The MIT License; 1998. [Online]. Available from: https://opensource.org/licenses/MIT [Accessed 25th January 2023].

[67] Free Software Foundation. The GNU General Public License v3.0; 2007. [Online]. Available from: https://www.gnu.org/licenses/gpl-3.0.en.html [Accessed 25th January 2023].

Appendix A

Benchmark Functions Adapted from the Python Performance Benchmark Suite

Benchmark	Type	Description
`chaos`	Application	Create chaosgame-like fractals
`deltablue`	Micro-Benchmark	Runs DeltaBlue algorithm, which is a single-directional, incremental, dataflow constraint solver
`fannkuch`	Micro-Benchmark	Runs Fannkuch algorithm, which is a permutation algorithm to generate and test all possible permutations of a sequence
`float`	Micro-Benchmark	Artificial, floating point-heavy benchmark
`go`	Application	Go board game
`hexiom`	Application	Solver of Hexiom board game
`json-dumps`	Micro-Benchmark	Test performance of json deserialization
`json-loads`	Micro-Benchmark	Test performance of json serialisation
`logging`	Micro-Benchmark	Test performance of logging simple messages
`mdp`	Application	Solve a complex Markov Decision Process
`nbody`	Micro-Benchmark	Solve N-Body equations
`pidigits`	Micro-Benchmark	Calculate digits of pi
`Raytrace`	Application	Using a simple raytracer
`Richards`	Micro-Benchmark	Simulates an operating system kernel, carrying out tasks such as scheduling, managing system resources, and processing system requests
`scimark`	Micro-Benchmark	Test various scientific function calculations e.g. sparse matrix multiplication, fast fourier transform
`spectral-n`	Application	Solver for problem 3 in Hundred-Dollar Hundred-Digit Challenge Problems
`unpack-seq`	Micro-Benchmark	Testing performance of unpacking sequence values in Python

Table A.1: Full List of Benchmark Functions Used from PyPerformance