



FaaS-Utility

Henrique da Conceição Reis dos Santos

Thesis to obtain the Master of Science Degree in

Engenharia Informática e de Computadores

Supervisors: Prof. Luís Manuel Antunes Veiga

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva

Supervisor: Prof. Luís Manuel Antunes Veiga

Members of the Committee: Prof. Rodrigo Fraga Barcelos Paulus Bruno

November 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my parents and my brother for their friendship, encouragement, and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible.

I would also like to deepest acknowledge my dissertation supervisors Prof. Luís Veiga and Prof. José Simão for their invaluable patience and feedback and Prof. Rodrigo Bruno for his comments and insight that made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

Function-as-a-Service (FaaS) is a cloud computing model that allows developers to build and deploy functions without having to worry about the underlying infrastructure. Current challenges such as cold start delay are still being actively studied, which is seen as a delay in setting up the environment where functions are executed, and one of the most significant performance issues. This causes great delays of latency and reduced quality of service to the customer of this model. It is still difficult for users to allocate the right resources, namely CPU and memory, due to the variety of function types, dependencies, and input sizes. Resource allocation errors lead to either under or over-provisioning of functions, which results in persistently low resource usage and significant performance degradation. This thesis presents a novel approach to optimizing the performance of FaaS systems using a utility function that takes into account customer entries. This utility function uses feedback from customers, in the form of preferences and pricing goals, to determine the relative importance of different functions to the overall system. This information is then incorporated into the scheduling process, ensuring that the most customer desired functions receive the necessary resources to perform optimally. This work presents an architecture to successfully implement the new approach into a scheduler in Apache OpenWhisk that uses a utility function that receives customer entries to better determine resource allocation. We also present the evaluation methodology to assess the implementation and analysis of the overall approach performance.

Keywords

Cloud Computing, Resource Scheduling, Function-as-a-Service, Pricing, Utility

Resumo

Função como um serviço (FaaS) é um modelo de computação na nuvem que permite construir e instalar funções sem preocupações sobre a infraestrutura subjacente. Os desafios atuais tais como o atraso no arranque a frio ainda estão em estudo. Este atraso corresponde ao tempo gasto para montar o ambiente de execução da função a ser invocada e corresponde a um dos maiores problemas de desempenho. A grande latência traduz-se numa reduzida qualidade de serviço para o cliente deste tipo de modelos. É ainda difícil reservar antecipadamente os recursos necessário à execução da função, nomeadamente memória e CPU, devido à enorme diversidade de tipos de funções, suas dependências e quantidade de dados de entrada. Imprecisões na reserva de recursos conduzem a que os recursos disponíveis sejam insuficientes ou em excesso, resultando numa subutilização dos recursos e degradação de desempenho. Esta tese apresenta uma nova abordagem para melhorar o desempenho de sistemas FaaS utilizando funções utilitárias que consideram a dimensão dos dados de entrada. Estas funções utilitárias utilizam informação dos clientes, sob a forma de preferências e objetivos de custo, para determinar a importância relativa das diferentes funções no desempenho global do sistema. Esta informação é incorporada no processo de escalonamento procurando que a maioria das funções do cliente receba os recursos adequados a um desempenho ótimo. Este trabalho apresenta uma arquitetura que permite realizar a nova abordagem no escalonamento realizado pelo Apache OpenWhisk que usa funções utilitárias que determinam a melhor reserva de recursos considerando os dados de entrada específicos do cliente. Apresenta-se, igualmente, uma metodologia para a avaliação da realização proposta e a análise do seu desempenho global.

Palavras Chave

Computação na nuvem, Escalonamento de recursos, Função como um serviço, Preço, Utilidades

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Contributions	4
1.3	Document organization	4
2	Related work	5
2.1	Function as a Service	7
2.1.1	Other Cloud Services and FaaS	7
2.1.2	FaaS benefits	8
2.1.3	Use Cases	9
	A – Infrastructural	9
	B – Applications	10
2.1.4	FaaS challenges	11
	A – Cold start delay	11
	B – Resource Allocation	12
	C – Security	12
2.2	Utility	13
2.2.1	Scheduling	14
2.2.2	Pricing	14
	A – Compounded Moore’s Law and beginning expenditures . .	15
	B – Spot Instance	15
	C – Price-at-risk	15
	D – Customer classification and resource consumption status .	16
2.3	Relevant and Related FaaS Systems	16
2.3.1	Amazon AWS	17
2.3.2	Google Cloud	17
2.3.3	Microsoft Azure	17
2.3.4	OpenStack-Cloud	17

2.3.5	Kubeless	18
2.3.6	OpenFaaS	18
2.3.7	Knative	19
2.3.8	Apache OpenWhisk	19
2.3.9	Kubernetes	19
3	Architecture	23
3.1	Apache Openwhisk overview	25
3.2	Scheduler extension	26
3.2.1	Pricing options for the client	27
3.3	Introduction to Apache OpenWhisk's base scheduling system	27
3.3.1	Controller	28
3.3.2	Kafka	30
3.3.3	Invoker	30
3.4	Enhanced Scheduler	32
3.4.1	Scheduling during an over-provisioned state	33
3.4.2	Scheduling during an under-provisioned state	35
4	Implementation	37
4.1	Apache Openwhisk deployment overview	39
4.2	Apache Openwhisk local deployment using Kind	39
4.2.1	Kind deployment lifecycle	39
4.2.2	Integration with Apache Openwhisk Development tools	40
4.2.3	Analysis of Kind for development	41
4.3	Apache Openwhisk Local Deployment using Docker-compose	41
4.3.1	Openwhisk-Devtools Deployment	41
4.3.2	Deployment of Newly updated Container Images	42
4.4	Development of Action-Spreading	43
4.4.1	Main code adjustments	43
4.4.2	Remaining issues	44
4.5	Development of Action priority	45
4.5.1	Priority queue within Kafka	45
4.5.2	Priority queue within Invoker	46
4.5.3	Other possibilities	46
4.5.4	Priority queue conclusion	47
4.6	Utility function management	47

5	Evaluation	49
5.1	FaaS Benchmarks	51
5.2	Metrics	51
5.2.1	Server sided metrics	52
5.3	Evaluation Environment	52
5.3.1	Docker Environment Set-up	53
5.3.2	wsk-cli	55
5.3.3	Curl and Base Evaluation	59
5.3.4	Jmeter and load evaluation	62
5.3.5	Actions used	64
5.3.6	Sub-Enviroments	65
5.3.7	Hardware used	66
5.4	Performance evaluation	67
5.4.1	Test 1, W-A	67
5.4.2	Test 2, C-A	68
5.4.3	Test 3, W-A	69
5.4.4	Test 4, W-A	70
5.4.5	Test 5, W-B	71
5.4.6	Test 6, C-B	71
5.5	Utility Function Evaluation	72
5.6	Analysis and Discussion	75
6	Conclusion	77
6.1	Future work	79
A	Code of Project	87

List of Figures

3.1	Overview of Apache Openwhisk's default architecture	25
3.2	General architecture with newly added scheduler and collector component	26
3.3	A faster response for action A	34
3.4	Four seconds of execution of the priority queue algorithm	36
5.1	Docker setup successfully complete with all containers running	54
5.2	wsk-cli apihost setup example	55
5.3	Example of auth for guest user	55
5.4	Example of not using the -i flag in wsk-cli	56
5.5	Example of creating action fn	56
5.6	Example of updating action fn	56
5.7	Example of creating action fn that already exists	56
5.8	Example of deleting action fn	57
5.9	Example of creating a trigger for fn	57
5.10	Example of rule for action fn	57
5.11	Example of invoking action fn with no input	57
5.12	Example of firing a trigger for action fn	57
5.13	Example of a complete activation result	58
5.14	Example of an activation result for action fn with no input	58
5.15	Example of invoking action fn with parameters	58
5.16	Example of a json file with inputs for action fn	59
5.17	Example of invoking action fn with a file	59
5.18	Example of curl PUT to create action fn	60
5.19	Example of curl PUT to create an already existing action fn	60
5.20	Example of curl PUT to update action fn	61
5.21	Example of curl post for action fn	61
5.22	Example of curl get for the result of invoking action fn	61

5.23 Example of curl post for action fn with blocking and timer	62
5.24 Example of a setup of a Jmeter HTTP Header Manager	63
5.25 Example of a setup of a Jmeter HTTP Authorization Manager	64
5.26 Example of a setup of a Jmeter HTTP request	64
5.27 Docker setup for a cold environment	66
5.28 Cost's behaviour depending on α values	74

List of Tables

2.1	Cloud services and their levels of user control.	8
2.2	Use cases and theirs various studies.	11
2.3	FaaS challenges and theirs various studies.	13
2.4	Characteristics of the various pricing mechanisms.	16
5.1	Comparison between the different possible schedulers	67
5.2	Reduced colds starts evaluation	68
5.3	Evaluation of the scheduling delay present	69
5.4	Control test to check parallelism capabilities of the scheduler	70
5.5	Same executions as test 4 but using hardware B	71
5.6	Same executions as test 2 but using hardware B	72
5.7	Utility evaluation	73

List of Algorithms

3.1	Overview of Apache Openwhisk's scheduling algorithm	32
3.2	Over-provisioned scheduling algorithm	33

Listings

A.1	Publish function, base Openwhisk	87
A.2	Publish function, updated Openwhisk	89
A.3	Schedule function, base Openwhisk	92
A.4	ScheduleAndSpread function, updated Openwhisk	94
A.5	Python code to calculate the number of additional invocations	95
A.6	Setup function for CommonLoadBalancer, updated Openwhisk	96
A.7	ProcessCompletion function for CommonLoadBalancer, updated Openwhisk	97

Acronyms

API	Application Programming Interface
BaaS	Backend-as-a-Service
FaaS	Function-as-a-Service
FCFS	First-Come-First-Serve
FIFO	First-In-First-Out
FTP	File Transfer Protocol
FTPS	File Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure-as-a-Service
JDBC	Java Database Connectivity
JMS	Java Message Service
LDAP	Lightweight Directory Access Protocol
ML	Machine learning
NoSQL	Not Only SQL
PaaS	Platform-as-a-Service
REST	Representational State Transfer
SaaS	Software-as-a-Service
SCP	Secure Copy Protocol
SFTP	SSH File Transfer Protocol
SOAP	Simple Object Access Protocol
URL	Uniform Resource Locator
VM	Virtual Machine

1

Introduction

Contents

1.1 Motivation	3
1.2 Contributions	4
1.3 Document organization	4

Edge computing [1], a development of cloud computing, has benefited from the cheaper cost and improved energy efficiency of lower-end computation and storage equipment that are common at the internet's outer edges. As a result, the edge of the internet is now richer and loaded with numerous resources that are yet mostly untapped.

Although users are initially willing to contribute, the sustainability of these community edge clouds depends on the users' access to interesting, relevant services, which are frequently deployed as virtualized containers, and their ability to get something in return (incentives) for letting others use their hardware [2].

At the same time, more organized and elastic applications, with reduced latency and better resource use, are made possible by serverless computing and the Function-as-a-Service model (also known as FaaS) [3].

1.1 Motivation

Current implementations of the Function-as-a-Service architecture such as Amazon AWS and Microsoft Azure focus deeply on the optimization of systems resources and performance while paying little attention to the individual desires of each customer.

Current scheduling mechanisms [4, 5] attempt to maximize available resources for the least cost, be that cost resource consumption or execution time. Customers tend to wish for execution times to be as low as possible, however, this is in general terms as not all customers are the same when it comes to urgency. One customer might just be requesting a project to be done by the end of the day and has little interest in when it is done in a few minutes or an hour, while another customer might need a request to be done as soon as possible; this information can be leveraged by providers, by employing fewer resources when they are scarce, while reducing the price charged to users [6]. We propose an optimization to the scheduling mechanism in FaaS that will take into account these customer differences in priority as well as provide monetary profits for the provider using our proposal by adjusting the price of the service depending on the priority desired by the customer. This implies that a customer using our system will be provided a few additional options, depending on the server's state, when attempting to request such as monetary discounts for slower execution times or extra monetary costs for his request to be completed promptly. The latter is presented in case the system is saturated and unable to confidently complete customer requests in the initially expected time frame.

While scheduling mechanisms are crucial when resources are limited, we also propose using these to maximize customers' quality of service when the system is not yet saturated (has an abundance of resources available). To achieve this, we propose a scheduling optimization that uses more resources than necessary, when the system has an abundance of resources, to generate faster execution on

repeated requests from a user. More resources than necessary are allocated, however, this comes at a price. Given this, to complement this optimization, we propose a corresponding pricing adjustment for the new total allocated resources. This allows the provider to still be able to offer a fair price to his customers.

We propose a scheduling optimization in the Function-as-a-Service model that receives input from the customer to assist its execution for a more intelligent and focused quality of service.

1.2 Contributions

To accomplish our desired Function-as-a-Service quality of service described above, we set out to do the following objectives:

1. Examine the most cutting-edge FaaS technology in use today to comprehend their key challenges, scheduling integrations, and customer-facing pricing models.
2. Survey current state-of-the-art open-source FaaS technology to determine the best-suited environment to develop and present our proposed scheduler.
3. Design an architecture on the desired open-source FaaS technology that adheres to the requirements set forth by our vision.
4. Develop and implement our proposed architecture in Apache OpenWhisk.
5. Create a structured evaluation methodology to easily assess if our future implementation fulfills our desired requirements.

These objectives are explored and described in the remaining sections of this work.

1.3 Document organization

This document is organized in two main sections, Section 2 and Section 3, followed by an evaluation methodology used and a conclusion, Section 5 and Section 6 respectively. Section 2 is dedicated to the related work in this field, primarily focused on Function-as-a-Service architecture. Following the related work is Section 3 where this document presents our proposed architecture deployed in Apache OpenWhisk outlining and explaining how we wish to implement our desired scheduling extension adhering to the requirements presented in Section 1.1.

2

Related work

Contents

2.1 Function as a Service	7
2.2 Utility	13
2.3 Relevant and Related FaaS Systems	16

This section will discuss the most cutting-edge techniques and technologies currently being used in this field and it is subdivided into three parts: Section 2.1 where a brief introduction to the Function-as-a-Service architecture as well as presenting its benefits, use cases, and challenges; Section 2.2 presents current scheduling and pricing mechanisms used throughout cloud computing; finally Section 2.3 currently used and developing Function-as-a-Service technologies that this work considered and studied.

2.1 Function as a Service

In terms of architectural layers of Cloud Computing, the Cloud is typically considered as numerous Cloud Services [2]. In essence, it relates to who will oversee these Services' many layers, these can be classified as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS), Backend-as-a-Service (BaaS) and finally Function-as-a-Service (FaaS) the main cloud service used of this work.

2.1.1 Other Cloud Services and FaaS

IaaS in the context of cloud computing refers to the management of the hardware and virtualization layer, which includes servers, storage, and networking, by the cloud provider. Applications developed over infrastructure built on top of IaaS are managed by the end user, including virtual instances, operating systems, applications, availability, and scalability. This service is the closest to the user, providing the most amount of control over the system to the user as well as having the lowest transparency [2].

PaaS consists in providing a Service where the cloud provider can offer a platform that controls the OS, availability, scalability, and virtual instances of instances built on top of IaaS. A provided runtime environment can be used if there is no specific runtime environment requirements [2].

SaaS provides complete abstraction of the software and backend. These are full programs that don't require any further effort from the user and may be utilized remotely. However, the restriction is that the organization has no control over the application [7].

BaaS and FaaS are now two additional service models. Both are thought to be serverless, as such BaaS and FaaS are frequently used in conjunction because they share operational characteristics (such as no resource management) [8, 9]. Applications that heavily rely on third-party (cloud-hosted) apps and services to manage the server-side logic and state are referred to as BaaS applications. The client then houses the bulk of the business logic, such applications are frequently referred to as "rich client" applications, including single page applications and mobile apps. Google FireBase is a prime example of BaaS. It is a complete mobile development platform that is hosted in the cloud and has direct client communication capabilities. As a result, there is no server in the way, and all resource and management concerns are handled by the database system [7].

FaaS offers the ability to deploy code (also known as functions) in the cloud and it's the greatest difference from BaaS. As a result, the developer can utilize his own programming without having to handle the hardware itself. An operator of a cloud service platform does not control everything, because the abstraction with FaaS is greater than with PaaS. The provider also manages the data as FaaS must come before PaaS (e.g., the state of the server). Scalability is another significant distinction between FaaS and PaaS. While FaaS scaling is completely transparent, PaaS requires the organization to still consider how to scale. Only the specific functions of the application are now deployed on FaaS [7].

When FaaS is invoked, the request is first authenticated using an authentication method, and FaaS is only triggered when the Invoker is given permission to do so. The code logic provided while deploying the FaaS function is used to activate FaaS for execution. Function instances are terminated once function execution is finished. By storing the state during the execution phase in consistent state resources such as Not Only SQL (NoSQL) databases, parameter stores, etc., the state of the FaaS function can be preserved by external means. This defines the FaaS lifecycle and it provides security as well as ease of use [2].

FaaS is our best prospect for this work due to the user only needing to worry himself with the business logic presented to him as shown in Table 2.1.

Cloud Services	Business Logic	App	Data	Runtime/ OS	Virtualization/ Storage	Examples
On-premise	User	User	User	User	User	Home Computer
IaaS	User	User	User	User	Provider	Apache Cloudstack
PaaS	User	User	User	Provider	Provider	AWS Elastic Beanstalk
BaaS	User	User	Provider	Provider	Provider	Google Firebase
FaaS	User	Provider	Provider	Provider	Provider	Apache OpenWhisk
SaaS	Provider	Provider	Provider	Provider	Provider	Google Workspace

Table 2.1: Cloud services and their levels of user control.

2.1.2 FaaS benefits

FaaS is a fairly straightforward implementation technique for micro-distributed APIs. The user only pays for the period of time that the FaaS was in operation [2].

The application's scalability and availability are not the user's concerns. FaaS is naturally highly available and automatically scalable. This greatly simplifies design architecture. FaaS can scale from a few requests per day to thousands of requests per second, automatically depending on the demand that

is necessary on the API. FaaS is extremely available by nature; even if one event fails, another one will be ready to fulfill the request in a short period of time. Since the FaaS function is only accessible via API to end users, it separates internal cloud resources from them and increases security because backend servers are hidden from view outside of the FaaS function [2].

2.1.3 Use Cases

FaaS can be used for a variety of use cases from Infrastructural where it is used as a middleman to achieve scalability and availability on top of an existing system such as in edge computing. However it can also just as well be used as an application where it can offer a easy and scalable way to resolve requests such as image and video manipulation for more direct results but in machine learning as well for more controlled results.

A – Infrastructural Low latency is frequently needed for use cases like monitoring people's vital signs during emergencies or in daily life [10]. To save lives in the event of a big disaster, paramedic assistance must arrive quickly. User-wearable sensors can offer vital details about a patient's health and assist in establishing a priority list for patient monitoring. Support for low latency is one of the primary forces for **edge computing**. In this situation, a serverless computing framework can handle server, network, load balancing, and scaling operational tasks [1].

There are a number of open-source FaaS frameworks that have been suggested to enable serverless computing on private infrastructure and prevent vendor lock-in. Recent studies have assessed the performance and utility of a few open source serverless frameworks [11, 12], but these studies do not take into account the limitations imposed by an edge-based environment [3].

FaaS is a scalable and flexible event-based programming model so it's a great fit for IoT events and data processing [3]. Consider as an example a connected switch and printer. When the button is pressed it sends an event to a function in the cloud which in turn sends a command to the printer to turn itself on. The three components are easily connected and only the actual function code would need to be provided. Thanks to managed FaaS, this approach also scales from two devices to thousands of devices without any additional configuration [3].

One FaaS solution made specifically for edge situations is tinyFaaS. Edge nodes can be single-board computers with low power or entire data centers, depending on their capabilities. FaaS platforms primarily focus on these larger data centers or clusters of servers, however, tinyFaaS also take into account the more prevalent limited edge nodes, such as single servers or single-board computers. Edge nodes are far more cost-effective in the huge quantities needed for edge computing, even though they have much less computational capacity than a full data center and are sufficient for many use cases [13]. These low-power edge nodes differ from cloud apps, which must scale across multiple cloud computers,

in part because they are monolithic [3].

While scalability and fault tolerance are constrained if a large number of devices approach edge for computation offloading requests, node management is drastically simplified, and platform management overhead is kept to a minimum. One of the key reasons to process data at the edge rather than the cloud is latency, which is a crucial component of the IoT. Alternative messaging protocols like MQTT or CoAP, which tend to be much more resource-efficient, can help to reduce latency [14, 15]. An edge FaaS platform should therefore natively handle such IoT messaging protocols while being able to do without the need for specialized triggers that are specific to cloud applications [3].

B – Applications Basic **image and video processing** that doesn't require a state to be saved for subsequent calls is a good fit for FaaS. Basic image and video operations like resizing, transformation, cropping, image to text conversion, and thumbnail creation can be carried out [2]. Due to its frequent use and ease of data analysis, this case study is frequently used for performance evaluations. For example, in [16] by managing an image resizing case study and in [17] where they considered a more complex image processing pipeline consisting of three functions in the final experiment. The first method generates a thumbnail version of an image by downloading it from its URL; the second function mirrors the image, and the third function converts the image to grayscale.

Application development is quick and reliable when using a distributed architecture based on microservices that can use multiple cutting-edge **programming languages simultaneously** for different modules. With the aid of FaaS, it is possible to create and distribute websites and applications without using backend servers for processing. Due to the built-in feature of autoscaling and the high availability of FaaS, applications and APIs can be scaled automatically. The developer should only concentrate on endpoint integration and processing logic [2].

The benefits of FaaS have triggered a growing interest in how to use it in **Machine learning (ML)**. Recently, research from both academia and industrial communities has focused their attention on the FaaS model for those applications. For instance, in the study [18] it was found that deep neural networks could benefit from the FaaS paradigm since users are allowed to decompose complex model training into multiple functions without managing the server. A novel FaaS architecture for the deployment of neural networks is discussed in [19]. Furthermore, various frameworks have been proposed to deploy machine learning in FaaS environments. For example, SIREN is an asynchronous distributed machine learning framework based on FaaS. AWS also provided one example of ML training in AWS Lambda using SageMaker [20] and AutoGluon [21]. SageMaker is a fully managed service that provides the necessary tools to create, train, and deploy ML models. AutoGluon is an open-source library that automates ML tasks [22].

A lot of efforts have been done to identify the possible ways to deploy FaaS for applications where

scientific computing is crucial. Existing studies, such as [23] demonstrated the feasibility of using the FaaS model for scientific and high-performance computing by presenting various prototypes and their respective measurements. In [23], the authors proposed a high-performance FaaS platform that enables the execution of scientific applications. A prototype for executing the scientific workflows in FaaS environments has been developed and evaluated by [24].

Without the need for intricate cluster-based systems, FaaS can also be activated from a variety of events generated, such as system logs, data events, scalability events, etc. **Event streaming** pipeline, queue, and stores can all be used as inputs for monitoring systems [2]. Table 2.2 shows the use cases and their various studies.

Use cases	Studies
Edge computing	[11–13]
Image and Video Manipulation	[16, 17]
Multi-language Applications	[2]
Auto-scaling highly available Websites and APIs	[2]
Machine Learning	[18, 19, 22]
Scientific Computing	[23, 24]
Event streaming	[2, 24]

Table 2.2: Use cases and theirs various studies.

2.1.4 FaaS challenges

A – Cold start delay The cold start delay, which is seen as a delay in setting up the environment in which functions are executed, is one of the most significant FaaS performance issues [25].

Popular systems most frequently use a pool of warm containers, reuse the containers, and regularly call routines to reduce cold start delay. However, these techniques squander resources like memory, raise costs, and lack knowledge of function invocation trends over time. In other words, while these solutions reduce cold start delay through fixed processes, they are not appropriate for environments with dynamic cloud architecture [26].

Despite the fact that serverless computing reduces some of the major IoT difficulties, these convergent technologies still have unique limits such as cold start time that must be addressed holistically. In the work [26], the authors proposed an intelligent method that chooses the optimum strategy for maintaining the containers' warmth in accordance with the function invocations over time in order to lessen cold start delay and take resource usage into consideration.

While in the work [27], the authors assume that the FaaS platform is a “black box” and use process knowledge to reduce the number of cold starts from a developer perspective. They suggested three methods to lessen the number of cold starts based on indicating the naive approach, the extended

approach, and ultimately the global approach, as well as a lightweight middleware that can be deployed alongside the functions for this purpose.

A straightforward illustration of provider-side cold start optimization is OpenFaaS. For each deployed function, it always maintains a single warm container, according to [28]. However, this only takes into account situations where the increase in arrival rate is smaller than one divided by the typical cold start latency. Cold starts continue for every additional concurrent request [27].

Likewise, Apache OpenWhisk [29] uses so-called “stem cells” which are running containers that use a base image without the function code and its libraries. This reduces the cold start time as containers are already “semi-ready”.

An alternative FaaS platform called SAND combines the functionality of a single application into a single container, preventing cold start buildup. SAND does not require our method, but as a research prototype, it is not yet suitable for production, therefore we must still deal with today’s FaaS services [27].

B – Resource Allocation Due to a variety of function types, dependencies, and input sizes, it is still challenging for users to assign the proper resources, namely CPU and memory. Resource allocation errors cause functions to be either under or over-provisioned, which results in persistently low resource use which generates considerable performance degradation.

Resource managers (RM) for FaaS platforms like Freyr [30] and SmartHarvest [31] optimize resource efficiency by dynamically harvesting free resources from over-provisioned operations and shifting them to under-provisioned services. Spock [32] suggests a cost and SLO-improving FaaS-based Virtual Machine (VM) scaling architecture. The works presented in [5] and [4] both aim to automatically change CPU resources when detecting performance degradation during function executions for FaaS resource management, which helps address the problem of resource over-provisioning.

The CPU resources allotted to functions by existing FaaS systems are typically distributed in proportion to the user-configured memory allocation. Apache OpenWhisk uses the same approach. In particular, the shares option of a newly constructed container for a certain action is set in proportion to the memory value defined for the activity. By doing this, the OS-level scheduler will, in the event of contention, offer actions with bigger memory allocations a higher share of CPU time [17].

C – Security Applications using FaaS raise several security challenges. Applications are vulnerable to a number of security flaws since they are integrated with database services, and back-end cloud services, and are connected through networks and events. For instance, event-data injection occurs when an application receives an unauthorized and untrusted data entry and executes it without checking it first. This kind of injection can target the container’s stored functions’ source code and other confidential information. Denial of service or Denial-of-Wallet attacks can be launched by an attacker due to insecure deployment setup and flawed access control. In order to increase costs or get unauthorized access to

function resources, these attacks take the use of functions having lengthy timeouts. Poisoning the good attacks, which frequently affect libraries and platform code, involves inserting harmful code into a library that numerous programs rely on [22].

To address some of these issues, there are numerous commercial security solutions available [33, 34]. Aqua [33] is a program that continuously checks container images and function's code, to make sure that developers don't add vulnerabilities in a library, embedded secrets (keys and tokens), or permissions. While for instance, Snyk [34] is one of the widely used tools for securing FaaS applications by identifying, addressing, and monitoring any security flaws in open-source dependencies.

Researchers have also addressed security concerns and put forth several solutions [35, 36]. Namely SecLambda is an extensible security framework that [35] proposes for carrying out complex security activities to safeguard a FaaS application and ensure control flow integrity, credential protection, and DoS rate limitation. A workflow-sensitive authorization approach for FaaS apps was created by the authors and published in [36]. It proactively examines the permissions of all workflow functions for external requests. This minimizes the application's attack surface by enabling the program to quickly reject illegitimate requests. Table 2.3 summarizes the FaaS challenges and their various studies.

FaaS challenges	Studies
Cold start delay	[26–28, 37–39]
Resource allocation	[4, 5, 30–32]
Security concerns	[22, 35, 36]

Table 2.3: FaaS challenges and theirs various studies.

2.2 Utility

There is a constant conflict between the provider and the customer throughout the entire product industry. The supplier must work to increase revenue while still enhancing its product for the benefit of the customer. There has been a lot of research done on cloud computing's optimization [17, 40, 41], but this rarely or never considers the potential revenue that these optimizations can provide [42]. We present both sides of the conflict in this section. When it comes to scheduling, the provider can use optimization techniques to improve the customer experience with little to no thought to the financial implications. And pricing is the most recent development in cloud computing pricing methodologies that aim to maximize revenue.

2.2.1 Scheduling

In distributed systems, scheduling is frequently studied to establish a connection between requests and available resources. For clusters [43], clouds [44], and cloud-edge (Fog) systems [45, 46], numerous solutions have been put forth. Load balancing [40], maximizing resource use [47] and energy efficiency [48], minimizing execution costs [49], and maximizing performance are the typical objectives of scheduling [50]. In edge computing, scheduling is necessary when services must be successfully offloaded. Offers scheduling innovations for edge computing that can be used in FaaS systems for this purpose [41]. They provide many approaches that present a fair priority-based scheduling system by taking into account the client and each request.

In the work presented in [17], they offer a cutting-edge scheduling system for FaaS that is QoS-Aware and implemented in Apache OpenWhisk. By adding a Scheduler component, which takes over from the Controller's load balancing function and allows more scheduling policies, they expanded Apache OpenWhisk. In this new design, incoming requests are routed through the Scheduler rather than the Controller in order to be immediately scheduled to the Invokers. This Java-based scheduler, which serves as middleware, is a meaningful inspirational factor in our work. Arrivals and Completions are the two basic events that the Consumer receives. Upon receiving fresh requests, the Controller publishes arrival events, which cause the related activation to enter the Scheduler buffer. In contrast, when activation processing is finished, Invokers publish completion events. The Controller in the standard version of Apache OpenWhisk uses this data, and their Scheduler also makes use of it to monitor the workload of the Invoker. While many of the objectives we hope to attain are illustrated in this study, pricing approaches are missing.

2.2.2 Pricing

The viability of cloud ecosystems is fundamentally dependent on service pricing [42]. Given the size of cloud computing environments, it is essential to offer an energy-conscious cloud architecture in addition to a business strategy with sensible resource pricing and allocation [51]. The bulk of studies places a strong emphasis on lowering overall energy use while paying little attention to other aspects like service pricing and proper cloud service billing [42].

One of the most crucial elements that could draw clients in is the pricing strategy. They consistently seek the best quality of service at the lowest cost. In contrast, cloud service providers strive to increase income while reducing expenses by implementing more modern technologies [52]. For the cloud services they require, different users ask for different quality service classes. Both the requested services and their quality are subject to change over time. Because it lacks the necessary capability to respond to the dynamic changes in service demands and their quality, the fixed price strategy, although simple, is

not a fair technique for both consumers and suppliers. Customers prefer to pay for what they have really used, and service providers prefer to publish a fair pricing structure so that they can bill their clients fairly and be competitive [42].

There are three basic difficulties with pricing models in cloud computing. Users of cloud services are often unable to understand billing events since they take place within the cloud architecture. To do this, a thorough taxonomy that takes into account all significant aspects of pricing schemes is required. The discrepancy between resource utilization and billing time is another issue. Bills are issued far after the use of the resource or service because the billing system is not synchronized with resource consumption. By reducing the processing time, using a suitable pricing model can also reduce the gap that was previously noted [42].

Not least of all, cloud service providers frequently combine or aggregate various events into a single line of code by combining the code of various requests into a single line to be executed. It speeds up the delivery of consumer bills and lowers the computing complexity for cloud providers, but accuracy and fine-grained information in the system are sacrificed. While everyone can agree on a clear fair pricing strategy that both service providers and customers are happy with, fair pricing is a subjective idea [53].

A – Compounded Moore’s Law and beginning expenditures In [54] the primary focus was ensuring that cloud service users received a high level of quality of service by establishing two distinct price restrictions. The upper bound is determined by the compounded Moore’s law, a modified form of the original Moore’s law, while the lower bound serves to cover the beginning expenditures. The variables that make up the initial costs are taken for granted in this analysis. Additionally, neither the cost calculation method nor a model to distinguish between the various parameter categories is disclosed [42].

B – Spot Instance The Spot Instance approach, which was extensively discussed in [55], is one of the realistic attempts to apply dynamic pricing. The actual issues with the application of this strategy are explained in this paper. The considerable price fluctuation of this pricing scheme is one of its drawbacks. Additionally, clients are unable to relate to the many fluctuations that occur in the price and quality of the given services since the pricing mechanism is not transparent to them. Last but not least, because applications may abruptly end in Spot Instances, this approach is unsuitable for real-time, interactive, or applications that require a stable level of quality of service and response time.

C – Price-at-risk The major goal of the work [56] was to address pricing uncertainty for on-demand computing services by providing a Price-At-Risk technique. All of the dynamic conditions described above are taken into account by Price-at-risk. The key issues for which the Price-At-Risk methodology attempts to identify a workable solution are difficulties with demand estimation accuracy and demand price elasticity. Machine learning can be used to tailor pertinent parameters based on the application

and the type of service to address this problem. The problem of coarse-grained granularity by using general formulas could be resolved, and the precision could be improved, by using machine learning algorithms and other cutting-edge technologies [42].

D – Customer classification and resource consumption status In economics, the term “price discrimination” is used to refer to varying prices [57]. To design advanced reservation pricing in Computer Grids, [58] relies on two key pillars: revenue optimization [59] and price discrimination. To assess income performance based on the aforementioned user types, the authors divide users into premium, business, and budget groups. Meanwhile, resource utilization can be classified as peak, off-peak, or saver. Two crucial cloud factors — customer classification and resource consumption status — were carefully taken into account in this study. The primary shortcoming of this research is the coarse granularity of variable rates, as establishing a better pricing strategy for every person requires more precise application parameters as well as service specifications. Table 2.4 summarizes the characteristics of the various pricing mechanism highlighting their advantages and drawbacks.

Study	Pricing method	Advantages	Drawbacks
[54]	Compounded Moore’s law and beginning expenditures	High degree of QoS	Initial cost taken for granted
[58]	Customer classification and resource consumption status	Income performance based on user types	Coarse granularity of variable rates
[55]	Spot Instance	Realistic dynamic pricing	Considerable price fluctuation
[56]	Price-At-Risk	Price elasticity	Lack of accuracy

Table 2.4: Characteristics of the various pricing mechanisms.

For us to create a comprehensive and appropriate FaaS Scheduling that maximizes revenue we must take into account the interests of both the developer (with financial offerings to keep supplying and improving the service) and as well as the user/consumer (with the use the service for his own needs), and not simply the performance of the cloud service.

2.3 Relevant and Related FaaS Systems

One benefit of cloud computing is the vast array of options from which a user can select the one that best suits his needs. This also holds for all developers worldwide, enabling them to support and expand current solutions or even develop new ones. Even though this idea of expanding already existing work occurs much more frequently in open-source projects, it is still essential to have a thorough understanding of what private businesses are providing to foster innovation within the cloud computing industry. We outline the most popular FaaS implementation options in this section, along with open-source options

that we as developers can tailor.

2.3.1 Amazon AWS

Lambda Service [2], a FaaS implementation in AWS, can scale up automatically as needed and can handle from small numbers of requests per day up to thousands per second.

AWS Lambda offers the runtimes for Java Script, Python, Ruby, Java, Go, and .Net as well as other programming languages as a platform for execution.

Support for custom library uploads is now offered for AWS lambda deployment. Numerous AWS services, monitoring events on AWS CloudWatch, and URLs can all be used to trigger Lambda.

2.3.2 Google Cloud

In Google Cloud FaaS, a feature known as “Cloud Function” is included. This feature can scale up automatically as necessary and can handle anywhere between a small number of daily requests and millions of daily demands.

Java Script, Python, and Go Runtime are offered by Cloud Function as Platform.

Support for custom library uploads is not offered for Cloud Function deployment. Cloud Function can be called manually, or it can be triggered by HTTP, Cloud Storage, or Cloud Pub/Sub events [2].

2.3.3 Microsoft Azure

The “Azure” function implementation in Microsoft Azure Cloud FaaS has the ability to scale up automatically when enabled.

As platform options, Azure Function offers PHP, Java, Java Script, PowerShell, C Sharp, and Python Runtime. Support for uploading custom libraries is offered for Azure Function deployment.

With the necessary IAM policies, Azure Function can use auxiliary services like Blob Storage, Cosmos DBs, EventGrid, Event Hub, HTTP, webhook, IoT Hub, Graph, Notification, Queue, Table Storage, Timer, etc [2].

2.3.4 OpenStack-Cloud

Cloud FaaS is implemented in OpenStack using a variety of platforms, including “Apache Whisk”, “Fission”, “IronFunctions”, “Fn Project”, “OpenLambda”, “Kuberless”, and “OpenFaaS.”

Underlying FaaS is implemented in OpenStack using services from Docker and Kubernetes. By developing the appropriate docker image, execution language support can be added as needed. Similarly, RAM and core requirements can be customized according to the docker image implementation for FaaS.

As a result, OpenStack Cloud offers a lot of customization options for microservices-based architecture. FaaS workloads and microservices application containers are typically managed in OpenStack FaaS implementations using Kubernetes. This makes it possible to implement the FaaS paradigm with precise control over memory and processing power.

The benefit of implementing FaaS using OpenStack technologies is that it can be more hardware specific by using the Ironic service. Additionally, specific memory and processing power requirements can be configured when implementing the microservice distributed service architecture. However, public cloud users have restrictions when taking these points into account [2]. These open source technologies are likewise open source, making it simple to access the source code and giving developers everywhere in the globe the tools they need to contribute (including us).

2.3.5 Kubeless

Kubeless is a FaaS framework that is native to Kubernetes. Functions, triggers, and runtime are the three primitives on which the Kubeless programming paradigm is built. The code that will be executed is represented by a function, and an event source is a trigger. Depending on the type of event source, a trigger may be connected to a single function or a collection of related functions.

This platform's key element is a controller, which constantly monitors for changes to function objects and takes the required actions, such as creating or deleting a new function object, as needed. The runtime image used to deploy a function may be explicitly supplied by the user, the image artifact may be generated on the fly, or the function code may be delivered into the associated Kubernetes pod using a pre-built image [11].

2.3.6 OpenFaaS

The fundamental building block of the OpenFaaS programming paradigm is the function. A handler and a function need to be provided by the developer. An API gateway is this platform's primary element. The API gateway interacts with the orchestration engine to offer scaling, metrics collection, and access to the functions (i.e., Kubernetes). Each function is packaged into a Docker container using a command line interface. Every container has a watchdog, which is a webserver that serves as an entry point and calls the function. The Kubernetes Horizontal Pod Scaler (HPA) or the AlertManager component (coupled with Prometheus) are used by OpenFaaS to allow the zero-scale capability where idle functions can be configured to scale down when they haven't received any requests for a period of time [11].

2.3.7 Knative

The Istio and Kubernetes platforms, which offer application (container-based) runtime and sophisticated network routing, serve as the foundation for the Knative framework. As a result, Knative can add CRDs to the Kubernetes platform to support higher levels of abstraction.

Building, Serving, and Eventing are this platform's three key pillars. The Build component is a plug-gable paradigm for building apps (in containers) from source code and is implemented using a Kubernetes CRD. Based on the requests it receives, this component offers scale-to-zero support and leverages Istio for network routing. The essential primitives for consuming and creating events are provided by the eventing component. The implementation of higher-level Application Programming Interface (API) concepts, CLIs, tooling, etc. is left to the discretion of particular vendors since Knative is not a full-featured FaaS platform [11].

2.3.8 Apache OpenWhisk

Actions, Triggers, and Rules are the three primitives on which the Apache OpenWhisk programming paradigm is built. A trigger is a group of events that can be caused by a variety of sources, whereas an action is a stateless function that runs code. A trigger and an action are connected by a rule. A sequence is a lengthier processing pipeline that combines multiple actions from various languages. The orchestration of the dataflow between functions and the language selection is separated by the composition process' polyglot character [11].

This platform's core building blocks are made up of an NGINX webserver, a controller, an Apache Kafka component, an Invoker component, and a CouchDB database for storing user credentials, action information, namespaces, and definitions of actions, triggers, and rules.

The entire system uses the Nginx webserver as a reverse proxy. Each request is authenticated, authorized, and routed by the controller component before control is transferred to the following component. The connection between the controller and Invokers is controlled by the Kafka component. Code from the CouchDB component is copied by the Invoker component and injected into a Docker container. Additionally, this component keeps track of the active Docker containers where actions are running. The outcome of a particular action is saved in the CouchDB component for retrieval once the execution of that action is complete.

2.3.9 Kubernetes

A flexible open-source platform called Kubernetes is used to orchestrate and manage containerized applications. Applications are executed in a cluster using pods, deployments, and services by Kubernetes. In Kubernetes, pods are the smallest deployable pieces of an application. They contain either

a single container or a collection of containers that share an IP address and are running in the same execution environment. One of the Kubernetes objects used to specify how to operate an application container as a pod and regulate the replica count is called a deployment. Services are abstractions that specify access rules and maintain a set of pods in the cluster. A name that is associated with one or more pods can be used to refer to any service established in the cluster. The CoreDNS, DNS server for Kubernetes, resolves the service names. Any DNS request is answered with an IP address by the service discovery tool CoreDNS. It monitors service events and makes any necessary DNS record modifications. When a user creates, modifies, or deletes a service or any of its associated pods, these events are triggered.

For the execution of FaaS applications, Kubernetes provides a number of functions, including auto-scaling, scheduling, load balancing, health checking, and self-healing of containers. One of Kubernetes' key automation features, auto-scaling, helps organizations adapt swiftly to demand spikes. The Horizontal Pod Autoscaler (HPA) is one of the well-known scaling techniques. In accordance with the current resource usage, such as CPU or memory utilization, the HPA is used to automatically scale up and down the number of pods associated with a single application.

The Kube-scheduler uses scheduling as a mechanism to choose the best node for pod placement. When the Kube-scheduler has a pod to deploy, it ensures that the allocated node satisfies all of the pod's unique needs, including those for CPU and memory resources. It begins by selecting the relevant nodes utilizing a set of filters in order to accomplish that. For instance, it makes use of affinity and anti-affinity rules, which are defined by labels and annotations that put restrictions on where pods can be placed. Second, the Kube-scheduler scores every node, giving nodes with higher affinity a higher score and nodes with higher anti-affinity a lower score. The node with the greatest score receives the pod last. The technique of effectively distributing the traffic among various pods of a particular service is known as load balancing. The Kube-proxy component routes the traffic that is sent to a Kubernetes service. By using iptables rules to build a virtual IP for a service, the Kube-proxy by default employs the random selection mode, which directs incoming requests to a service's randomly selected pod. The most adaptable method for exposing services to the outside world is Ingress, which functions as a controller in a dedicated pod and offers routing rules to govern access to the Kubernetes services.

Kubelet continuously checks the health of pods using a straightforward technique to learn more about their current situation. The readiness probes may form the basis of the health check. The health and readiness of the pods are checked using a readiness probe before they may begin accepting traffic. When every container inside a pod is prepared, the pod is deemed ready. A pod gets removed from service load balancers when it is not prepared. The readiness probe can be implemented in three different ways: by an HTTP request, a TCP socket in which the IP and port of the container are checked, and through a user-defined command. Kubernetes uses self-healing, an automated recovery technique,

to make sure the cluster is actually in a healthy state. It includes automatic insertion, automatic restart, and automatic replication. For instance, if a pod fails, Kubernetes restarts a new one. Similarly, if a node goes down, Kubernetes immediately reschedules all the pods from the downed node onto other healthy nodes (which may take up to 5 minutes). Many open-source FaaS frameworks shift the duty of container orchestration functions to Kubernetes and concentrate solely on FaaS features to benefit from the robust Kubernetes infrastructure.

3

Architecture

Contents

3.1	Apache Openwhisk overview	25
3.2	Scheduler extension	26
3.3	Introduction to Apache OpenWhisk's base scheduling system	27
3.4	Enhanced Scheduler	32

In this Chapter, we first present an overview of Apache Openwhisk's systems, more specifically its scheduling methodology, followed by our proposed scheduling extension which is subdivided into two components: during an under-provisioned server state and an over-provisioned server state.

3.1 Apache Openwhisk overview

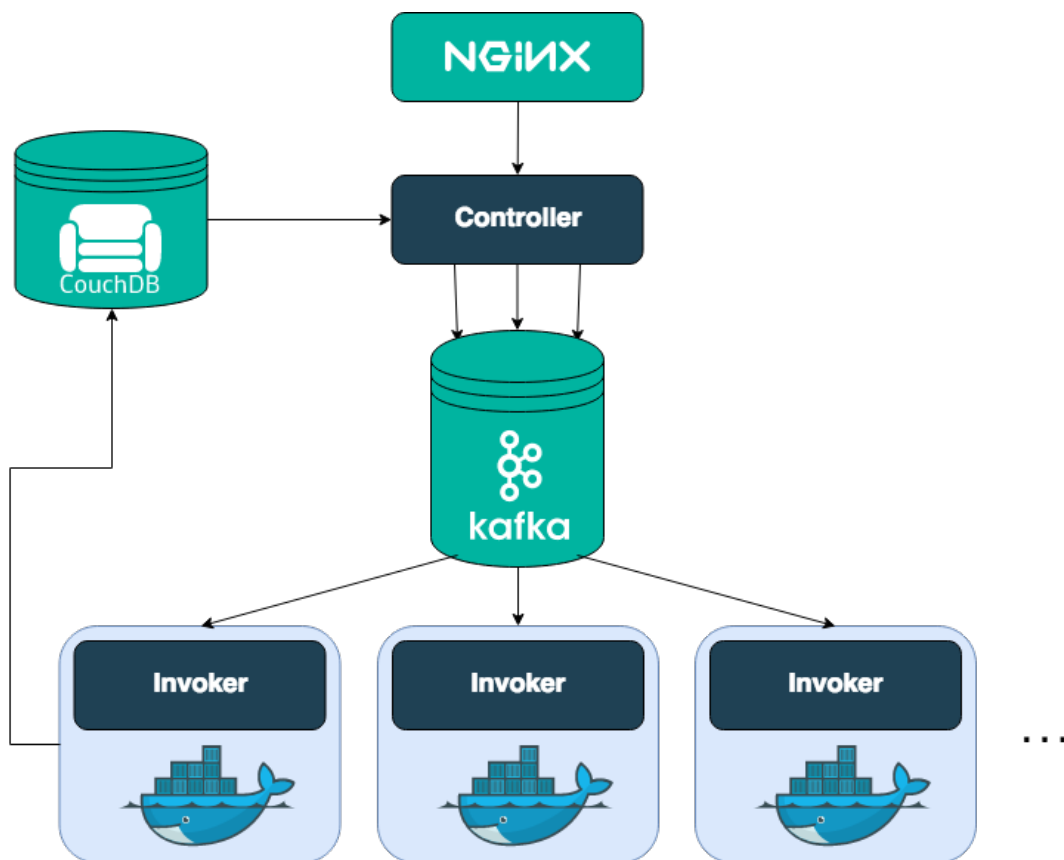


Figure 3.1: Overview of Apache Openwhisk's default architecture

To create new functions, invoke existing ones, and query the outcomes of invocations, Apache OpenWhisk exposes a Representational State Transfer (REST) interface built using NGINX. Users initiate invocations using an interface, which is then transmitted to the Controller. To schedule the function invocation, the Controller chooses an Invoker, which is commonly hosted utilizing virtual machines. Based on (1) a hashing method and (2) information from the Invokers, such as health, available capacity, and infrastructure state, the Load Balancer in the Controller schedules functions invocations. After selecting an Invoker, the Controller delivers the function invocation request to the chosen Invoker via a Kafka-based distributed message broker. After receiving the request, the Invoker uses a Docker container to

carry out the function. Functions are commonly referred to as actions within Apache Openwhisk. The Invoker sends the outcomes to a CouchDB-based Database after the function execution is complete and notifies the Controller of its completion. The Controller then synchronously or asynchronously returns to clients the outcomes of the function executions [60].

3.2 Scheduler extension

In our extended version of the Apache Openwhisk architecture, we will add a newly updated scheduler with all of our requirements for the pricing utility function as well as an updated Collector to allow us to extend the capabilities of warm container creation with no additional overhead. Both of these extra components are shown in Figure 3.2 as the green and blue containers.

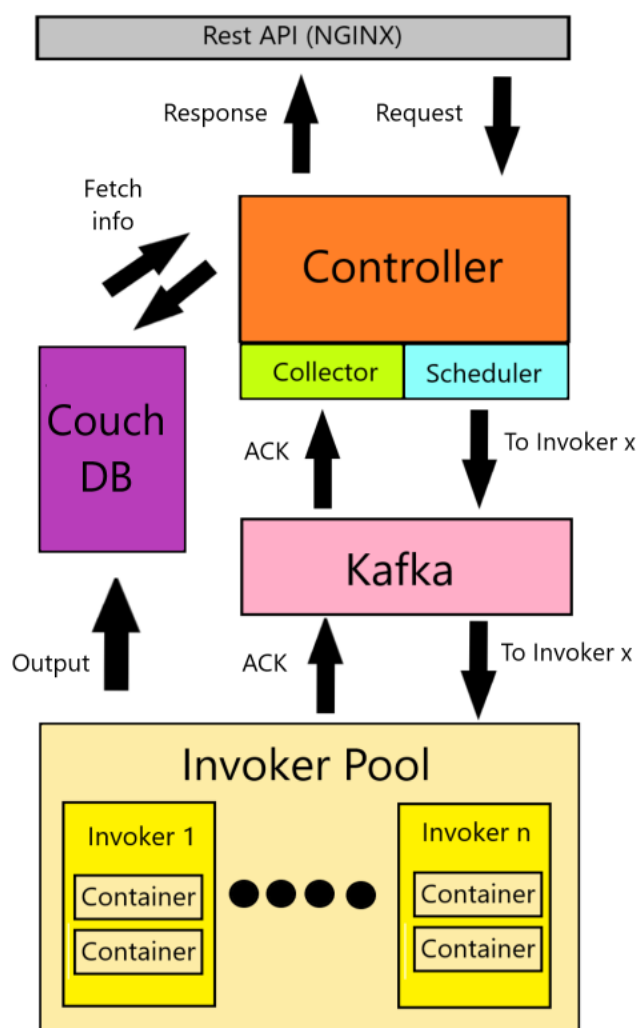


Figure 3.2: General architecture with newly added scheduler and collector component

Firstly the client uses the exposed REST interface built using NGINX to make a request. This request is then forwarded to the controller that receives all the relevant information from the CouchDB for the setup of the activation. It then uses that invocation information with the addition of both our new utility function and the Invoker Pool state to determine the most suited Invoker to schedule the request to. After sending it to the specific Invoker through Kafka it is executed by the containers within it. The completion state is given to CouchDB directly by the Invoker. An activation ACK will be sent back to the controller for it to update the scheduler with additional information for future requests. The enhanced collector will support more advanced information given by the Invoker and Invoker Pool. After the operation has been completed the client will receive the output of the action. We will provide a total of 6 combinations for pricing opportunities, two initial options for the over-provisioned state and three additional ones for the under-provisioned state. The client will be able to choose a combination of the initial and additional one for a more customized experience.

3.2.1 Pricing options for the client

The two initial pricing options will be provided: (1) a **Basic Version** which merely finishes the request with no additional benefits, or (2) a **Premium Version** that completes the request with additional Invokers but the additional resources used for a faster execution of the request will come at a discounted price. The second option is to use the request to create warm containers for this particular client's repeated uses, resulting in future execution times that are quicker. The client will receive all of this information for transparency's sake and encourage continued use.

The three additional different pricing augmentations will be provided if the servers are under provisioned meaning some requests may need to wait in line before being executed. (1) **Standard priority**, which offers no priority when it comes to scheduling requests but still offers the same cost per execution time as when the servers are under-provisioned. (2) **Urgent priority** offers increased request scheduling priority (though not an absolute priority) but at a higher cost for clients who have self-perceived time-critical actions to be performed. An example of a such client is someone who detected a mistake in a database and wishes it to be fixed as soon as possible so that further uses of the database not be compromised. (3) **Reduced priority** which offers, for a reduced price tag, a lower priority in the system for clients that have little interest in the execution delay of the operations, for example, a student that is ahead of schedule for project delivery.

3.3 Introduction to Apache OpenWhisk's base scheduling system

Openwhisk's scheduling is subdivided into three main components: the Controller, Kafka, and Invoker. The primary component of this system is the Controller which is responsible for deciding on

sending the requests (mainly action executions) to the best Invoker possible.

3.3.1 Controller

Firstly the controller communicates with the CouchDB to verify if the user has privileges to execute the request. If it does not, the action request is canceled and logged but the information is not registered for consumption in the load balancer metrics. If the user indeed has privileges to execute such a request then it utilizes a load-balancing algorithm to determine the best Invoker to send the request to.

At first, for every *namespace + action* pair a hash is calculated and then an Invoker is picked based on that hash ($hash \% numInvokers$). The determined index is the so-called *home-invoker*. This is the Invoker where the following progression will start the majority of the time. If this Invoker is healthy and if there is capacity on that Invoker, the request is scheduled for it. If one of these prerequisites is not true, the index is incremented by a *step-size*. The step-sizes available are all coprime numbers smaller than the number of Invokers available (coprime, to minimise collisions while progressing through the Invokers). The *step-size* is picked by the same hash calculated above ($hash \% numStepSizes$). The *home-invoker-index* is now incremented by the *step-size* and the checks (healthy + capacity) are done on the Invoker we land on now. This procedure is repeated until all Invokers have been checked at which point the *overflow* strategy will be employed, which is to choose a healthy Invoker randomly. In a steadily running system, that overflow means that there is no capacity on any Invoker left to schedule the current request. If no Invokers are available or if there are no healthy Invokers in the system, the loadbalancer will return an error stating that no Invokers are available to take any work. Requests are not queued anywhere in this case. An example:

- availableInvokers: 10 (all healthy)
- hash: 13
- homeInvoker: $hash \% availableInvokers = 13 \% 10 = 3$
- stepSizes: 1, 3, 7 (note how 2 and 5 are not part of this because it's not coprime to 10)
- stepSizeIndex: $hash \% numStepSizes = 13 \% 3 = 1$, stepSize = 3

Progression to check the Invokers: 3, 6, 9, 2, 5, 8, 1, 4, 7, 0, done. This heuristic is based on the assumption that the chance to get a warm container is the best on the *home-invoker* and degrades the more steps you make. The hashing makes sure that all load balancers in a cluster will always pick the same home Invoker and do the same progression for a given action. Invoker health is determined via a Kafka-based protocol, where each Invoker *pings* the load balancer every second. If no ping is seen for a defined amount of time, the Invoker is considered "Offline". Moreover, results from all activations are

inspected. If more than 3 out of the last 10 activations contained system errors, the Invoker is considered “Unhealthy”. If an Invoker is unhealthy, no user workload is sent to it, but test actions are sent by the load balancer to check if system errors are still happening. If the *system-error-threshold-count* in the last 10 activations falls below 3, the Invoker is considered “Healthy” again.

- “Offline”: ping missing for more than 10 seconds.
- “Unhealthy”: 3 system-errors or more in the last 10 activations, pings arriving as usual.
- “Healthy”: less than 3 system-errors in the last 10 activations, pings arriving as usual.

The maximum capacity per Invoker is configured using *user-memory*, which is the maximum amount of memory for actions running in parallel on that Invoker. Spare capacity is determined by what the load balancer perceives is scheduled for each Invoker. Upon scheduling, an entry is made to update the books within the controller, and a slot for each MB of the actions memory limit in a semaphore is taken. These slots are only released after the response from the Invoker (active-ack) arrives or after the active-ack timeout. The Semaphore has as many slots as MBs are configured in user-memory.

Known caveats:

- In an overload scenario, activations are queued directly to the Invokers, which makes the active-ack timeout unpredictable, possibly hurting tail latency. Timing out active-acks, in that case, can cause the load balancer to prematurely assign a new load to an overloaded Invoker, which can cause uneven queues.
- The same is true if an Invoker is extraordinarily slow in processing activations. The queue on this Invoker will slowly grow if it gets slow to the point of still sending pings, but handling the load so slowly, that the active-acks time out. The load balancer again will think there is capacity, when there is none.
- The Controller is internally unaware of the real state of the container pool within each Invoker, implying a lack of potential load balancing opportunities depending on the number of warm or pre-warm containers present at a given moment in a desired Invoker.

Now that an Invoker has been chosen, a process that has an invocation setup is started before sending the request. This process consists of generating a unique *activationID* for this given request and storing a completion request inside a *TrieMap*. A *TrieMap*, also known as a Prefix Tree or Radix Tree, is a tree-like data structure used to store a dynamic set of key-value pairs, where the keys are substrings. The term “Trie” comes from the word “retrieval” and is pronounced like “try” (not “tree”). This completion request has a relevant timeout for the request. When the Invoker eventually sends to the controller the request completion this *TrieMap* is used in combination with the *activationID* to check if the request

exists and if it exists, whether it was completed and can be given back to the user. If at any point the setup fails, the request is canceled and the user is informed that the cancellation was not his fault.

3.3.2 Kafka

Assuming the invocation setup was successful, the controller uses a Kafka provider to send the invocation request to the proper Kafka topic relevant to the Invoker. In Kafka, a “topic” is a category or a feed name to which messages are published. Topics are analogous to channels or queues in traditional messaging systems. Producers write data to topics, and consumers read data from topics. Topics in Kafka are highly flexible, allowing different producers and consumers to work independently without direct coupling. This decoupling enables better fault tolerance, as well as the ability to add or remove producers and consumers without affecting other components of the system. These queues are FIFO in processing but can be fully checked by proper use of offsets. Offsets are a critical part of Kafka’s design, as they enable consumer groups to achieve parallelism and fault tolerance while ensuring that messages are not missed or duplicated during the consumption process. They represent the position of the next message that a consumer will read from a particular partition. The consumer groups in Apache Openwhisk’s case are the Invokers.

3.3.3 Invoker

The Invoker is notified by Kafka when a given topic has been updated and proceeds in its extraction and execution. Upon receiving a request to run, it first attempts to schedule it to a warm container of that given action using its name as a reference. Thus meaning if there is a warm container with the same execution code but a different action name it will not be recognized. It’s important for users to take this into consideration when deciding if they want to use an existing action or make a new one. If no warm containers are available then it will attempt to seize a pre-warm container. If this also fails then the Invokers will attempt to create a new cold container for the given action. Firstly it checks if the Invoker has available memory for a new container since the Invoker might only have a few containers and less than its maximum capacity. An example of this situation would be during startup where only a few pre-warm containers are created.

If there is no capacity for the cold container to be created it will remove the least used nonactive containers in order until it has enough capacity for the cold container. If this also fails, in case all of the Invoker containers are in active states it will reschedule it within the Invoker itself and reattempt this operation an additional time after the Invoker completes another activation. This reattempt has priority over new incoming requests, but if the reattempt fails it will be sent back to the controller has an unsuccessful activation.

The Invoker has a sense of priority within incoming requests and will consume them in the order they are given. However, that doesn't mean they will return to the controller in the same order, as certain activations might be faster than others. All parts of activation sequences are treated individually as singular activations with one beginning after the last one is completed and providing no priority to the Invoker upon receiving them.

After the execution is successful or not, it reports to the *controller* for it to update both the user and the CouchDB, updating all metrics related to load balancing and caching data for future similar activations. The containers are paused to reduce resource consumption but are removed after a sufficiently long timeout.

The Apache Openwhisk controllers are responsible for the organization of requests received by the clients. To do so the controllers manage Invokers. Invokers in turn are responsible for the management of the container pools that will deploy the actions. Based on the total number of available pools, as well as the overall number of controllers in the system, Apache OpenWhisk assigns a certain number of Invokers to each controller. When a controller leaves or enters the system, these Invokers are dynamically changed. When a controller receives an action identifies its home Invoker, which is responsible for deploying that specific action on subsequent calls unless the Invoker is unavailable, in which case the action is deployed to another Invoker.

The order in which Invokers are chosen given a specific action is determined by a hash calculation using the action's name and its client. This hash is then combined with the total number of Invokers in the system to choose its home Invoker. As said previously when an action's home Invoker is unable to take any new requests due to capacity or went down due to unforeseen circumstances another Invoker is needed. This choice is made through a combination of the previous Invokers index and the coprimes lower than the total number of Invokers. Coprimes are used to minimize the collisions while progressing through all Invokers. This algorithm creates an Invoker cycle for each action that minimizes their deployment collision, spreading out the actions throughout the system. Given an overloading scenario where all Invokers are unable to receive actions due to a lack of resources, a randomizing function is used to determine the Invoker to which the action is deployed. If all Invokers are unavailable due to being unhealthy then the action deployment is cancelled and the client is notified.

Busy Pool, **Free Pool**, and **Pre-Warm Pool** are the three different types of pools that house containers in these Invokers. The Busy Pool is responsible for running the code for deployed actions, so if it is overloaded with action deployments, it won't be able to run the code for any additional deployed actions. After an action has been deployed, containers are maintained in the free pool; these containers, which are also referred to as *warm containers* for that particular action, are reused if the action with which they are associated is deployed again. Last but not least, the Pre-Warm Pool maintains containers that only require code initialization, making them quicker than newly created containers but slower than the warm

containers of an action making them the second fastest option for an action deployment. The main difference between warm and pre-warm containers is that the former already has the code initialized and only requires the inputs specific for the request, while the latter still has to go through code initialization that while quite fast compared to a cold start is still additional time that leads to a slower total execution time. Given that the Busy pool of a specific Invoker is not saturated, as highlighted in red, it will first try to deploy the action using a container for the specific action in the Free Pool, as highlighted in green. If no container exists it will then try to utilize a Pre-Warm container from the Pre-Warm Pool, as highlighted in blue. Finally, if this attempt also fails; it schedules the Invoker to create a new container and deploy the action. Finally, before any creation of new containers (including using pre-warm containers), the scheduler deletes the least recently used container if the sum of containers in the free pool and busy pool equals the max pool size, as highlighted in purple for the case where there are old containers and brown when none are available exist. Algorithm 3.1 exemplifies in pseudo-code the steps described above with the aforementioned highlighted colors.

Algorithm 3.1 Overview of Apache Openwhisk's scheduling algorithm

```

Action ← A
ActionContainer ← Action
for all Invokers do
  if BusyPoolSize = MaxPoolSize then
    continue
  else if ActionContainer ∈ FreePool then
    FreePool ← FreePool \ ActionContainer
    BusyPool ← BusyPool ∪ ActionContainer
    return
  else if PreWarmPoolSize > 0 then
    PreWarmPool ← PreWarmPool \ PreWarmContainer
    ActionContainer ← PreWarmContainer
    BusyPool ← BusyPool ∪ ActionContainer
  else if FreePoolSize + BusyPoolSize = MaxPoolSize then
    FreePool ← FreePool \ LeastRecentContainer
  else
    ActionContainer ← ColdContainer
    BusyPool ← BusyPool ∪ ActionContainer
  end if
  return
end for
Queue ← Queue ∪ Action

```

3.4 Enhanced Scheduler

Through the combination of our new pricing model using a utility function, as well as the new scheduler, the price presented to the client will be dynamically adjusted based on the additional resources used only if they were used.

3.4.1 Scheduling during an over-provisioned state

The scheduling system will operate as usual if no pricing mechanism is used, or, in other words, if the deployment uses the standard fee for the initial pricing option. If an additional premium fee is requested then the scheduler will attempt to deploy the action to all Invokers, not just the home Invoker. This scheduling modification has two additional benefits for the client: (1) if the action is repeatedly requested, saturating the home Invoker, it allows for a much faster execution following the initial deployment. Clients are further encouraged to use our system repeatedly because doing so will result in faster execution times; (2) the request will be handled by the fastest Invoker at that given time which may not be the home Invoker, while ignoring the overhead of calculating which one it is. The client may customize the deployment to include both versions of the pricing mechanism on a case-by-case basis for each action or trigger. This will allow the client to only include the premium option on specific actions within the deployment.

Algorithm 3.2 Over-provisioned scheduling algorithm

```
Action  $\leftarrow A$ 
ActionContainer  $\leftarrow Action$ 
for all Invokers do
  if BusyPoolSize = MaxPoolSize then
    continue
  else if ActionContainer  $\in$  FreePool then
    FreePool  $\leftarrow FreePool \setminus ActionContainer$ 
    BusyPool  $\leftarrow BusyPool \cup ActionContainer$ 
    return
  else if PreWarmPoolSize > 0 and Invoker = HomeInvoker then
    PreWarmPool  $\leftarrow PreWarmPool \setminus PreWarmContainer$ 
    ActionContainer  $\leftarrow PreWarmContainer$ 
    BusyPool  $\leftarrow BusyPool \cup ActionContainer$ 
  else if FreePoolSize + BusyPoolSize = MaxPoolSize then
    FreePool  $\leftarrow FreePool \setminus LeastRecentContainer$ 
  else
    ActionContainer  $\leftarrow ColdContainer$ 
    BusyPool  $\leftarrow BusyPool \cup ActionContainer$ 
  end if
end for
Queue  $\leftarrow Queue \cup Action$ 
```

Algorithm 3.2 will still queue the action if all Invokers are semi-saturated (the sum of busy and free pool containers is equal to the max pool size), while the original scheduling algorithm will only queue if all Invokers are saturated (busy pool is equal to the max pool size). However, this challenge should rarely arise during an over-provisioned state in which this algorithm is designed for. The main changes made to the algorithm are the ones highlighted in blue and red. Since the purpose of this new functionality is to create new containers we forced the scheduler to ignore pre-warm containers when outside of the action's home Invoker to create warm containers on the other Invokers for future use, as highlighted in

blue. As highlighted in red we adjusted the original algorithm to continue to search for non-fully saturated Invokers instead of simply exiting when the scheduler found a single available Invoker.

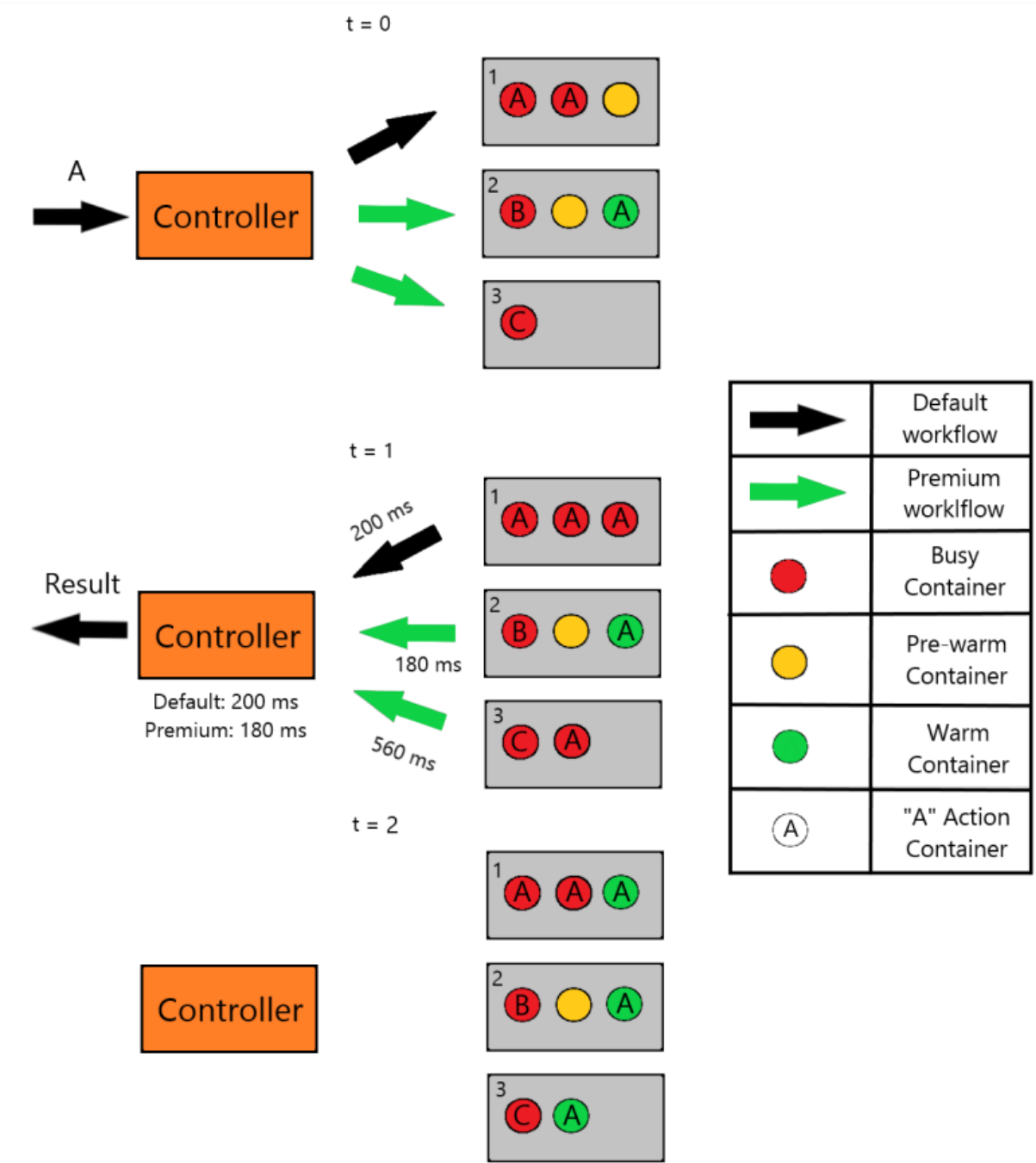


Figure 3.3: A faster response for action A

Figure 3.3 shows an example where at timestamp $t = 0$ the controller receives an invocation for action "A" on an environment with three Invokers each with a MaxPoolSize of three. When receiving

action “A”, the controller finds its action home Invoker, where in this example is Invoker 1, and in default circumstances only sends that action to that Invoker. In the premium version of the operation, the action is also sent to all other Invokers.

When the execution, $t = 1$, within the Invoker is complete each Invoker will send back to the controller the result. The controller will offer the client the fastest response from all the Invokers. In this example, Invoker 2 was the fastest due to having a warm container ready for action “A”. Invoker 1 only had a pre-warm ready to be used which is slightly slower than warm containers. Invoker 3 had no containers ready so it had to create a new action container for action “A” making it a cold start. If the premium fee was not requested the Invoker 2 result would not exist since it’s not the actions “A” home Invoker and Invoker 1 was not full, making the final result 20 ms (0.02 seconds) slower than using the premium compared to the default.

Finally, on the last timestamp $t = 2$ it is shown the final state of the environment after the operation. Due to the premium’s functionality a warm container on Invoker 3 now exists preventing potential future cold starts for action “A” on itself and making the pre-warm container on Invoker 1 also a warm container for even faster future invocations of action “A” just like Invoker 2 had in the default situation. If there is a surge of invocations for action “A” soon, the premium fee is great at preventing cold starts, like those expected in Invoker 3, while making the additional resources cheaper for the client.

All of the results of the multiple executions of the action are received by the controller. The cost of the requested deployment by the client is calculated as a ratio between the cost without the extra Invokers and the total cost of all resources used. Consequently, the cost the client will charge is given by the equation

$$final\ cost = \alpha c + (1 - \alpha) C, \quad (3.1)$$

where α is the ratio of the cost that remains static, c is the cost of the deployment under default conditions, and C is the total cost of all resources used.

This creates a situation where if no additional actions were deployed on other Invokers the final costs are equal to the normal pricing model.

3.4.2 Scheduling during an under-provisioned state

As stated in the final part of Section 3.3, a FIFO priority method is used in case action starts being queued due to the server being saturated, this in turn results in a very low urgency methodology for the clients. This work proposes a more advanced priority-aware system that allows more time-critical situations to be more hastily resolved for an additional cost. It also allows the inverse situation where a client might want a discount if the need arises.

The algorithm is based on a priority value coined by us **aPrio**, standing for absolute priority. If two

actions have the same values of aPrio the FIFO priority will be applied. This aPrio value will be updated every second while the request is in the queue. Given a request's priority ranking of reduced, standard, and urgent the aPrio value will be incremented by $+p_1$, $+p_2$, and $+p_3$, respectively. Figure 3.4 exemplifies four seconds of this algorithm in progress where $p_1 = 1$, $p_2 = 2$ and $p_3 = 5$, and the variable t represents the timestamp used in the system in seconds. Yellow requests are in the queue while red requests are the selected actions for when resources are freed.

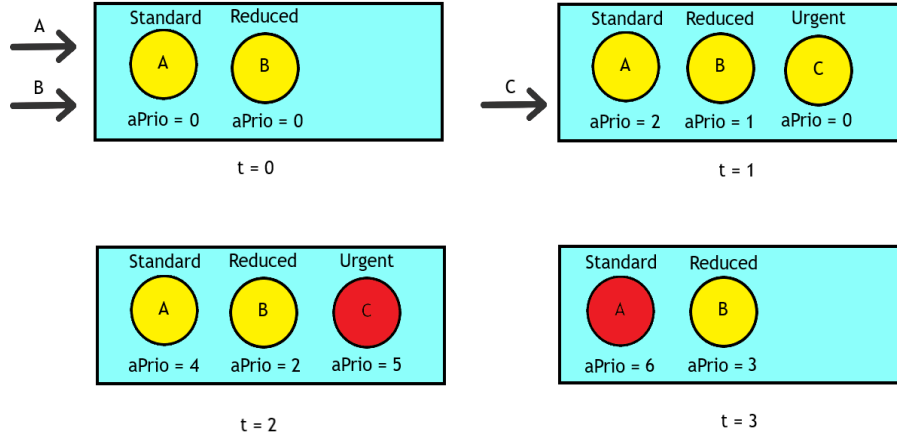


Figure 3.4: Four seconds of execution of the priority queue algorithm

The pricing model utilized is similar to what is offered during the over-provisioned state. The final cost is given by the equation

$$final\ cost = \alpha c + (1 - \alpha) \frac{cp}{p_1}, \quad (3.2)$$

where α is the percentage of cost that remains static, c is the cost of the specific action, p represents the value of the priority system used for the action, and p_1 is the value of the reduced priority system.

4

Implementation

Contents

4.1	Apache Openwhisk deployment overview	39
4.2	Apache Openwhisk local deployment using Kind	39
4.3	Apache Openwhisk Local Deployment using Docker-compose	41
4.4	Development of Action-Spreading	43
4.5	Development of Action priority	45
4.6	Utility function management	47

This Chapter will go over the development steps we took to achieve our realization of the architecture presented in Chapter 3. Starting with an explanation of the two environments we developed in, followed by an in-depth dive into the adjustments and extensions made to the source code of Openwhisk.

4.1 Apache Openwhisk deployment overview

Apache Openwhisk is a combination of various existing components such as CouchDB and NGINX with the unique addition of the Controller and Invoker. For this reason, docker deployments are heavily suggested for its use, given their ease of generation of complex environments and connections between multiple types of components. We started development in a Kubernetes cluster due to it being the most common approach in other works. However, after we took a deeper look into multiple other tools we finalized our development in a Docker-compose environment.

4.2 Apache Openwhisk local deployment using Kind

Our initial attempt at deployment was using a local Kubernetes cluster on a Ubuntu machine. Local deployments allow for greater control and provide options for easier development due to direct access to all related systems instead of requesting logging information from a third party in the case of deployments of Kubernetes clusters in the cloud, like Google and Amazon AWS. These cloud deployments should be considered for proper production deployments instead of development. For the local deployment, Kind was used to generate the nodes related to the Kubernetes cluster. Kind stands for “Kubernetes in Docker”, and it is a lightweight tool that allows you to run a Kubernetes cluster using Docker containers as nodes. It was primarily designed for testing Kubernetes itself but may be used for local development or CI. Kind uses Docker containers as nodes, making it lightweight and fast to set up and tear down clusters. This is especially useful for us who need to spin up multiple clusters quickly for different tests.

4.2.1 Kind deployment lifecycle

Kind by itself only creates the Kubernetes cluster nodes inside the docker meaning the nodes still have no proper connections or sense of order within themselves. For this step, Helm was used. Helm is a package manager for Kubernetes that simplifies and streamlines the deployment and management of applications and services on Kubernetes clusters. It provides an easy way to define, install, upgrade, and uninstall complex Kubernetes applications, often referred to as “charts”. Helm is widely used in the Kubernetes ecosystem due to its numerous advantages. It uses Go templates to create reusable, parameterized manifests. This allows you to define Kubernetes resources once and parameterize them,

making it easier to deploy the same application with different configurations. Helm allows you to manage different versions of your applications using charts and releases. You can easily roll back to a previous version if an update causes issues, providing better control over deployments. Helm also supports managing dependencies between charts. This is particularly useful when deploying applications that rely on other services or components. Overall, Helm simplifies the process of deploying complex applications on Kubernetes by providing a well-structured, version-controlled, and customizable packaging format. It improves the efficiency, reliability, and maintainability of Kubernetes deployments, making it an essential tool for managing applications in a Kubernetes environment.

As for the deployment itself, we built using Kind a cluster with one control plane and three worker nodes. The control plane is the node in which the controller image will be deployed and the three worker nodes are where the Invoker images will be deployed. This information can be checked and altered in the `kind-cluster.yaml` file and it's the central file used by Kind for its deployment. All of the other components such as the Zookeeper, Kafka, CouchDB and NGINX will be deployed as well within the Kubernetes cluster using their latest stable versions publicly available. Helm is then used to properly finish the deployment. The file `mycluster.yaml` is where all the information is related to unique requirements for Helm requested by us, the developers, such as port and hostname locations for the components as well as any flags related to replica limits for Invokers and controllers.

4.2.2 Integration with Apache Openwhisk Development tools

By using Kind, a Helm a base local Apache Openwhisk deployment was successfully deployed and ready to be used. Apache Openwhisk has developed and distributed three useful tools to manage and interact with the cluster. These tools are `wsk-cli`, `wsk-dev` and `wsk-admin`.

The `wsk-cli` abstracts the complexity of interacting with the OpenWhisk platform through simple and intuitive commands. It enables us to create, update, delete, and manage serverless functions and actions effortlessly. We can use the `wsk-cli` to set parameters and pass input data to serverless actions, making it easy to customize the behaviour of functions without modifying the underlying code.

`Wsk-dev` is a tool for developers to safely update the code within Apache Openwhisk. It allows us to isolate testing and deployment of specific parts of the components within Openwhisk. Its main goal is debugging any change made to the source code.

Finally, `wsk-admin` allows fast and easy management of security related to different user privileges. Any security-related issues can be analysed and resolved through `wsk-admin`. While `wsk-cli` is the perfect tool to “toy” around with Openwhisk, any proper tools for testing and data analysis are incompatible with `wsk-cli`, such as Artillery, Locust, Hey, and Jmeter. As such we dove deeper within the inner workings of the `wsk-cli` written in the Go programming language. All Apache Openwhisk operations can be represented through HTTP requests made to the NGINX server interface. An example is (user is named

guest and was properly logged in for wsk-cli):

```
Wsk-cli - wsk action invoke action1
```

```
HTTP - Post request to /api/v1/namespaces/guest/actions/action1
```

This allows for proper workload testing for evaluating Apache Openwhisk using state-of-the-art tools. This allowed initial success when understanding the inner workings of the Apache Openwhisk scheduling system.

4.2.3 Analysis of Kind for development

However, difficulties arose when modifications to the controller were needed. While image alterations with the help of the wsk-dev tool were successful the actual deployment of such image modifications was not so much. While image alteration requests could be made through modifications to the `my-cluster.yaml` file, the newly created images could not be trusted by Helm. This problem arose due to the lack of proper certification required when deploying images using Kind. This issue can be resolved with the use of cloud provider certification or the release of a personal institution-certified website with proper certificates. Both of these solutions would push us much closer to a production-level operation and further away from the development ease required to undertake such a deep understanding of Apache Openwhisk. While initial information pointed us to the use of Kind for the deployment of Apache Openwhisk and all their related components for local deployment, it proved to be more difficult to develop and test components within. Kind is a tool for development in a Kubernetes environment and not a tool to develop the tools within the Kubernetes cluster. It lacked the ease required to swap images of the controller and its redeployment due to security constraints. If our goal was to test and develop a project that used a Kubernetes cluster as support then Kind would be a perfect fit.

4.3 Apache Openwhisk Local Deployment using Docker-compose

As we sought more development-centered tools for Apache Openwhisk. We found the proper dev-tools for Apache Openwhisk, with the help of the git Apache development community.

4.3.1 Openwhisk-Devtools Deployment

The new deployment was directly on Docker-compose where each component of the Openwhisk architecture was a container node. This allows for easy version checking of the images, quick and clean logging information as well as proper port flow management that tools such as Wireshark and other network-related tools require for ease of packet checking. This allows for a much easier and

stronger development environment where we can test and manage changes to the Apache Openwhisk components.

Docker-compose came with its fair share of limitations such as lack of proper Kubernetes availability and scalability. If a node was to fail the system would need to be restarted manually since failure in a node's health check would not prompt its reconstruction. As well as lacking the capability to deploy additional Invoker and controller components mid-execution as resources are needed. Both of these concerns need to be taken into consideration when deploying in a proper production environment. Since our work wants to test and improve the system in both over-provisioned and under-provisioned states both of these concerns can be ignored for the most part. They must still be taken into consideration as any test in a production environment might alter the amount of resources for unrelated reasons to the cluster itself, such as power failures and manual resource allocations. We imitated them as much as possible the Kind deployment as that was never the issue. So we created one controller and three distinct Invokers as well as all other components.

One additional new component was the `Kafka-topics-Ui` for the ease of visualization of Kafka-related issues that might have arisen during development. All of these settings are present in the `docker-compose.yaml` file. One disadvantage of the use of singular containers in docker-compose is the forced use of ports. Whereas in Kubernetes we could combine all Invokers with one singular port. However, due to the reduced scale of our testing and development port availability was never a sufficiently worrisome issue.

4.3.2 Deployment of Newly updated Container Images

Through the use of the devtools provided by the git repository and the available Makefile the deployment of the system was successful. Running the repository was as simple as `make run` as well as using our own new development images as simply running `make pull` after the successful image naming alterations were made to the `docker-compose.yaml`. For new image development and conservation, we utilized a docker assisted image-registry where we made available our images for docker of both new versions of the Invoker and controller components by using the `wsk-dev` tool. The command `wskdev Invoker -b` is used to compile and update the local image of the Invoker component of Apache Openwhisk. A similar command is used for the controller `wskdev controller -b`. To update and push this newly created image to our docker registry we executed the commands

```
docker tag whisk/Invoker localhost:5000/controller:v2
docker push localhost:5000/controller:v2
```

where `localhost:5000` is the registry location. After the system was up and running checking if the new images were being successfully used we both checked the docker interface for their usage as well as checking the logs generated by them in the logging folder determined in the `docker-compose.yaml`

file for each component.

With all the considerations taken into account, we proceeded once again in the understanding of the Apache Openwhisk internal scheduling operations.

4.4 Development of Action-Spreading

Firstly we must fully comprehend what the attempted solution presented in the Architecture chapter entails. The goal is to (1) set up containers for future workloads as well as if possible, and (2) combine the work of all Invokers for an even faster possible execution. Therefore we tackled both of these problems separately starting from the more architecturally taxing, problem (1).

As Invokers are unaware of each other's conditions, we are unable to generate containers at will depending on the states of each other. Invokers also can't create empty warm containers. The best we could do would be to employ more pre-warm containers which are already greatly optimised by Apache Openwhisk. So we should look at the controller as it is aware of some of the Invoker pool's state information. However, as described in the previous section it does not hold all information such as Invoker pool states. Since we assume the global Openwhisk state is in an over-provisioned state we can safely invoke additional actions without taxing already existing actions due to the containers and their executions being isolated ¹. The only overhead generated would be the increased controller message load which is synced as well as Kafka's additional messaging. Kafka however is known to be made to be a high throughput, low latency, and ability to handle large volumes of data, making any increase of workload during an over-provisioned state largely irrelevant, but something to be aware of during evaluation nonetheless.

4.4.1 Main code adjustments

The initial approach would be to alter the algorithm in `ShardingContainerPoolBalancer.scala`, with part of its original code shown in Listing A.1, to not stop at its search at the first available Invoker, but instead, keep searching available Invokers to induce the action. Since our goal is to prepare the action within all possible Invokers, be that in generating ready-to-use warm containers or simply creating cache data for the Invoker our best solution would be to execute the action in all possible Invokers. This would also solve our problem (2) presented in the goals. Since all possible Invokers will attempt to execute the action not simply create a warm container, the fastest container at that moment will return the result to the controller. As shown in the Listing A.1 for the base version of Openwhisk of the publish function within the controller, the hash is calculated in lines 12-14, and the Invoker is chosen through the function

¹During an over-provisioned state Invokers will have an abundance of resources available and due to containers being isolated from each other, creation of additional containers should not result in a degradation of the Invoker's performance

“schedule” present in lines 17-25 where it returns an option where it's None if all Invokers are down, and a tuple with two inputs if there is at least 1 Invoker up. The first input of the tuple is which Invoker was chosen and the second is True if the system was overloaded (an Invoker was chosen at random) or False if not. The boolean is used for metrics purposes and serves no additional purpose for scheduling options.

Afterwards, in lines 49-51 it sets up the activation message and sends it to the Invoker through Kafka. If we are to change the algorithm itself we must dive into the schedule function called above. This function as shown in Listing A.3 the schedule function in the base version of Openwhisk is a tail-recursive function that will use at its tail the *stepsDone* and use it to know if it has cycled through all Invokers. In lines 16-19 it is shown that after it finds an available Invoker, needing to be both a healthy Invoker and the respective semaphore returning true for the required memory, it immediately returns. This code also shows in lines 21-29 that, if all Invokers have been seen, it will forcefully acquire the semaphore and choose a random healthy Invoker, returning afterward.

For our goal to be met we must spread the action to all available Invokers. We must then change the schedule function's return condition to only when all Invokers have been seen and registered all possible Invokers that the action could be scheduled. We also want to maintain the home Invoker metric stable in case our user does not request this additional functionality. To achieve all these conditions we altered in Listing A.3 for schedule. The end result was a new function named *scheduleAndSpread* with the code shown in Listing A.4. We maintained the tail recursion but simply removed the stopping condition and added two additional tail arguments which are the additional Invokers that we want to spread the action to and the homeInvoker named *chosenInvoker* within the code.

To fully ascertain our goal we must also alter the sending operation to take into consideration our extra Invokers “if any”. Considering that the *activationID* is the same for all, the controller knows where to send them back when it receives the response from all Invokers. With this, the action will safely be spread to all Invokers and be executed by such in order to both create additional containers for future use (1) and collect the fastest activation (2), since they all have the same *activationID*.

4.4.2 Remaining issues

While this fully resolves our issues from the client's perspective, there are still issues to be solved from the server's perspective. The system was not made to have multiple acknowledgments with the same *activationID* making it generate soft errors, meaning they don't affect the output that the user receives but generate errors internally. These errors heavily disrupt the logging information and database stability so we must take care to resolve these logging messages.

The acknowledgment processing is made through the use of the *TrieMap* modified during the activation setup. This *TrieMap* assumes the Ids in the *TrieMap*, the *activationID*, are unique and will overwrite

the old setup with the new one. This code is presented in the parent file `CommonLoadBalancer.scala`. To solve the issue we simply need to adjust the code for the `setup` function to take into consideration our new requirements as well as take care of these new possible values in the `processCompletion` function.

For the `setup`, we adjust the value `Invokers` to be the number of `Invokers` currently with this specific activation and simply increment it by 1 if the activation already exists within the *TrieMap*. As for the `processCompletion` we must simply decrement this new value by 1 when receiving a valid *acknowledgementMessage* from the `Invoker` and only remove the activation from the *TrieMap* when no more `Invokers` have this activation. The updated `setup` and `processCompletion` functions are shown in the Listings A.6 and A.7.

We finally have proper action spreading within our new Apache Openwhisk controller, with both goals (1) and (2) realised.

4.5 Development of Action priority

When the system is under-provisioned, we would like to implement action priority as described in our architecture Chapter 3. The main goal is to create a sense of user agency for these situations but provide a priority-induced queue when executing actions. There are two main places where queueing is present in Apache Openwhisk, the `Kafka` and `Invoker` components.

4.5.1 Priority queue within Kafka

`Kafka` itself is a distributed event streaming platform designed to handle real-time data streams. However, `Kafka` is not designed to offer priority when distributing its messages. Unlike other message brokers such as `RabbitMQ` and `ZeroMQ`, `Kafka` follows a First-In-First-Out (FIFO) message processing model, where messages are typically consumed in the order they are received. This comes with multiple advantages to FaaS systems where high availability and fault tolerance are key.

There are solutions to make `Kafka` “support” message priority, such as multiple topics per consumer and partitioning. Generating multiple topics per consumer seems to be the most promising due to Openwhisk already creating topics per `Invoker`. However, this method greatly tarnishes `Kafka`’s speed due to the fault tolerance procedures it takes and it is heavily discouraged unless speed is truly not relevant, which is false in our case. FaaS is meant for event-driven procedures and prides itself on availability, so major hits to the speed of the system would greatly reduce its availability. Partitioning is simply sectioning a topic through multiple consumers, which in our case wouldn’t work since each topic only has one singular `Invoker`.

Substituting `Kafka` for other brokers such as `RabbitMQ` would be a grand undertaking as `Kafka` is one of the core components of Apache Openwhisk and the source code would have to be tremendously

changed but even assuming that would be possible, other brokers that have focus on message complexity which would only be valuable for this circumstance, and we would be losing on the scalability, high throughput and data retention provided by Kafka which are main points for FaaS related systems.

Adjusting the controller to become the queueing system would be disastrous as the controller component is synced and would lack scalability even if additional controllers were created. This makes any avenue to implement priority inside Kafka fruitless if we want to maintain the system as a FaaS system.

4.5.2 Priority queue within Invoker

Another Queuing opportunity is within the Invoker. This queue is used for incoming requests that arrive from Kafka. Adjusting this queue from a simple First-Come-First-Serve (FCFS) queue to a priority queue is a simple task as Scala itself provides a priority queue component. By changing this queue into a priority queue we could safely alter its functionality.

To test and analyze this new extended implementation of the Invoker queue we set up a simple Docker-compose environment with one Invoker only. A stream of 200 simultaneous requests of a very simple action that would just do the sum of 2 integer values. More in-depth information about the environment and the testing process can be found in the Evaluation Chapter 5. We augmented the Invoker logging information to be able to visualize the priority queue itself and not just the number of elements for better analysis. We also check the Kafka logs for its queuing information to ascertain any bottlenecks.

When observing the logs the priority queue of the Invoker would never exceed two items within it, while Kafka had over 100 items in the queue. This observation was also detected when testing the original version of the Invoker queue. This perceived inconsistency is answered through Kafka and the Invoker interaction. When an Invoker has completed an activation as space within it can be used it alerts Kafka, which then Kafka sends the message to the Invoker. This process is made before setting up the return message to the controller. The Invoker will only start processing the next item in the feed/queue after it has successfully sent the message to the controller. This means that the requests in the Invoker queue only exist to more easily parallelize the message retrieval and sending from and to Kafka. Changing its queue to a priority queue would be an easy task but would not even remotely accomplish our goal, as we would be applying an advanced queuing system to a group consisting of 1 or 2 requests instead of the hundreds that Kafka handles.

4.5.3 Other possibilities

A common way to exercise priority in FaaS systems is to not implement them within FaaS but instead exercise a third-party tool to order the HTTP request made to the NGINX server. This allows the FaaS

system to preserve all of its advantages and allows user agency in request management. While this solution is valid for a production environment it is in essence outside of the scope of Apache Openwhisk as it acts outside of its borders. Examples of such tools are Envoy Proxy and HAProxy. Any company seeking to use Apache Openwhisk with priorities for production-focused environments should use this option instead of tarnishing or simply heavily reducing the unique functionalities presented by FaaS systems. However, the use of such tools would be outside of the scope of this work.

4.5.4 Priority queue conclusion

While a priority-aware scheduling system was a valuable and interesting opportunity it brought an immense overhead and a much higher complexity than initially expected during research and Architecture development. As previously explained since Kafka was not originally designed for priority-aware systems and its use is integral for Openwhisk Architecture a great undertaking would need to be done to fully remodel the architecture. As such we will leave the possibility of a priority-aware scheduling system using our utility function (3.2) for possible future work as the topic itself still is a valuable research direction for the FaaS system. As such we decided to instead focus the rest of this document on analyzing the development of the Action-Spreading functionality.

4.6 Utility function management

As described in the architecture Chapter 3, the adjusted scheduling benefits for the user would be described and presented through simple-to-understand utility functions.

We changed both the `primitiveAction.scala` and `ShardingPoolBalancer.scala` files to process the action names uniquely so as to be able to take user interaction into account. The interaction is based on the assumption that if a user wants to call action “fn” without the use of the additional spreading functionality nothing special from the user will have to be made. This keeps the old interactions and tests that a company might have when implementing this new system intact. However, if a user desires to call an action with the additional functionality in mind then he must both create a new action and invoke it with a new name.

Passing a new argument while possible would create additional overhead as it would need to be combined with the HTTP header as a new flag, just like the *blocking* and *overwrite* flags. The new name is simply the addition of “SPREAD_” at the start of the action name. So in the case of “fn” the user would need to input “SPREAD.fn”. This assumes that when creating the action the code placed in the action “fn” is the same as “SPREAD.fn”. The system will remove this prefix when sending to the Invoker to be treated as the same action as the original allowing the creation of relevant warm containers.

This modification is made in the file `PrimitiveAction.scala` as it is where the message sent to the Invoker is made. The prefix “SPREAD_” is simply removed if it exists and added to the message. Had this change not been made then the Invoker would treat this spreading action as an entirely different action due to its name, and making useless warm containers for the relevant action that it would receive in the future. This prefix is also checked `ShardingPoolBalancer.scala` to adjust its execution to either use or not the new functionality. This gives all existing tests that have actions not including the prefix “SPREAD_” with the same outputs as before.

The action spreading utility function can be easily calculated in real-time as logging information can be accessed in real-time and the number of extra Invokers related to a specific activation is easily accessed through a simple `grep` command within the log file. And making an automated system that would do such an operation is simple. We developed this simple program in Python; its code is shown in the Listing A.5. This script relies on the path of the controller logs file so it should be known. If doubt arises on what this path is a simple check of the `docker-compose.yml` on the volumes section of the controller will tell us where it is located. All of the remaining data needed for the utility functions is determined by the company deploying the system and what profits they desire.

This process could be made faster through the use of more advanced tools and algorithms since, for example, we can make the search faster by searching from the tail end first as the log file is chronological. The tool `ag`, also known as “The Silver Searcher”, is a fast and highly efficient text searching tool, primarily designed for searching code files and text content within large codebases, which could be useful to make the search even faster if the log file becomes too big for `grep`.

5

Evaluation

Contents

5.1 FaaS Benchmarks	51
5.2 Metrics	51
5.3 Evaluation Environment	52
5.4 Performance evaluation	67
5.5 Utility Function Evaluation	72
5.6 Analysis and Discussion	75

In this Chapter, we will go into depth about the system goals and the assessed metrics. We will implement the system by deploying Apache Openwhisk on a development environment based on Docker. The base open source code of Apache Openwhisk is extended to the requirements presented by the architecture in Section 3. Data is assumed to be stored locally or on some cloud storage in the same location.

5.1 FaaS Benchmarks

Four diverse FaaS workloads are used in the evaluation of our system those being Sleep functions, File hashing, Video transformation, and Image classification [61]:

Sleep functions are a good FaaS benchmark because it is a simple, low-overhead operation that can be used to measure infrastructural overheads, in our case the scheduling infrastructure, of a FaaS platform.

File hashing is also a good benchmark because it is a relatively simple operation that can be used to test the ability of the system to handle file inputs and outputs.

Video Transformation is a good benchmark for FaaS systems because it exercises many of the key features of the system, such as scalability, concurrency, and performance. Video transformation tasks, such as transcoding, are typically compute-intensive and require parallel processing. This makes them well-suited for testing the ability of the FaaS system to handle high levels of concurrency and scale horizontally.

Image classification is a good FaaS benchmark for our evaluation as well due to it being a complex operation that requires significant computational resources and can be used to test the ability of the system to handle more demanding workloads. Additionally, Image classification is a common use case for FaaS [17], especially in machine learning applications [18, 19], so using it as a benchmark can help to evaluate the system's ability to handle real-world workloads.

5.2 Metrics

Latency, Scheduling delay, and Resource usage are the three main metrics considered to determine the overall success of our system:

Latency is a metric that represents the amount of time it takes for a request to be processed and for a response to be received. It is an important metric for evaluating the performance of a system because it directly measures how long it takes for the system to respond to a user's request.

Systems that have low latency can respond quickly, which can lead to a better user experience. Systems that have high latency may result in slow response times and cause user frustration.

Scheduling delay is a metric that assesses the amount of time that elapses between when a user request is ready to be executed and when it is allowed to run by the scheduler. It is an important metric for evaluating the performance of a system because it measures how well the scheduler can distribute resources and manage the execution of tasks. A low scheduling delay indicates that the scheduler can quickly and efficiently assign resources to tasks, which can lead to better overall system performance. On the other hand, a high scheduling delay can lead to poor resource utilization, decreased system throughput, and increased response times.

Resource usage is a good metric to evaluate FaaS systems because it provides insight into how efficiently the system is utilizing resources such as memory and CPU. By measuring resource usage, one can identify any bottlenecks in the system and make adjustments to improve performance and reduce costs. Additionally, monitoring resource usage can help in identifying and troubleshooting issues such as resource leaks, and it could be combined with information on how effectively applications are making use of the resources allocated to them [62].

5.2.1 Server sided metrics

For Memory consumption and overload management metrics we require the access and analysis of the logs provided by the Openwhisk components. These logs can be found in the file location described in the “volumes” Section of the `docker-compose.yml`. Each service has its logging location and these must be checked to have an understanding of both sides of a request.

These metrics are measured and compared with the Apache OpenWhisk default scheduler.

5.3 Evaluation Environment

This section is subdivided into six subsections. The first subsection is the documentation of how the Environment used for the Evaluation was set up. It is followed by a subsection focused on explaining how this environment can be interacted with. After this, a subsection introduces how the metrics used for Evaluation are retrieved on the basic level. A subsection explaining Jmeter which is the advanced tool used to create and evaluate the tests performed in the following section. The next three subsections detail the various variables used during the tests, these include the actions used, the starting sub-environments of each test, and the two different hardware.

The environment used for evaluation of the newly enhanced Apache Openwhisk scheduling is similar to the one presented during the Implementation Chapter 4. The cluster size was kept low, easily

overloading the system if need be for testing. This means it was the same used for the Implementation Chapter 4, meaning one container for each required component of Openwhisk plus three invokers managed by one controller. The main testing variables are actions used, number of requests, and number of parallel users. Since the core of our work is to offer the user agency within the execution of his actions, we need to simulate different users requesting the server. We assume the server state is fresh at the start of each evaluation. Each test was made in either a cold environment state or a warm environment state. Both of these are more deeply explained in the subsections below. Authentication of the requests required for each evaluation is made by the first request made to the server and is preserved within the cache of the controller needing no additional authentication for the remaining duration of the test.

5.3.1 Docker Environment Set-up

The docker-compose base environment was set up through the standard installation found in the docker website ¹. The Openwhisk environment was deployed through the use of the GitHub repository for the Openwhisk devtools ². The `docker-compose.yml` file was altered from the original found in the dev-tool repository to take into consideration our additional invokers and new image for the controller. This file is subdivided into services and we are interested in the invoker and controller services. In the controller service, we must adjust the `image` parameter to our desired image location. In our case, the image was set up within a docker registry at `localhost:5000`, and the controller image was named `controller` with tag `v2`. Leaving this parameter with `image: localhost:5000/controller:v2` allows docker to start up the container using our newly updated image.

For the addition of two extra invokers, we added two more new services named `invoker2` and `invoker3` with the same information as the already existing `invoker` service except for three parameters, `ports`, `command` and `volumes`. The parameter `port` must be changed to a new unique port within the system, while the `volumes` can be changed to alter the logging location of this specific invoker, to not override the original invoker logging location. Last but not least, the `command` parameter must be changed to include a different invoker `id`. This can be achieved through the change to the flag `--id 1` to `--id 2` and `--id 3` for invokers 2 and 3, respectively.

Any additional invokers can also be added by following these steps as long as `ids` and `ports` are unique. Any removal of invokers can be done simply by deleting or commenting the service within the `docker-compose.yml` file. Before starting up the environment we also adjusted the `Makefile` present in the devtools to include our new image name and tag throughout the file. More specifically in the `pull` and `docker-pull` options, to verify that the image being used by the environment is up-to-date with the one present in the docker registry.

¹<https://docs.docker.com/>

²<https://github.com/apache/openwhisk-devtools>

To start up the environment one must employ the command `make quick-start` to download and set up every container within docker. Any further use can only `make run`. To stop but not destroy the environment the command `make stop` is used. All of these operations are made in the command line and are to execute the existing Makefile within the devtools repository. All images are downloaded including Openwhisk-related ones such as the invoker image and api-gateway (NGINX), these images are `openwhisk/invoke:nightly` and `openwhisk/api-gateway:nightly` respectively. For comparison purposes with the base Openwhisk the controller used was the image `openwhisk/controller:nightly` and is simply adjusting the `docker-compose.yml` controller service back to its original version. After the system is running there can be seen within the docker all the components those being: minio, redis, zookeeper, db (CouchDB), kafka, kafka-rest, kafka-topics-ui, api-gateway, controller, invoker, invoker2 and invoker3. With these containers confirmed running as shown in the Figure 5.1 the system can be used and evaluated.














































Name	Image	Status	Port(s)
 openwhisk	-	Running (12/1)	
 kafka-topics-ui-1 649b61180883 	landoop/kafka-topics-ui:0.9.3	Running	8001:8000 
 kafka-rest-1 a59a85101708 	confluentinc/cp-kafka-rest:3.3.1	Running	
 kafka-1 480f499dfc7d 	wurstmeister/kafka:0.11.0.1	Running	9092:9092 
 db-1 118b3c74b45a 	apache/couchdb:2.3	Running	5984:5984 
 zookeeper-1 f230d16d418e 	zookeeper:3.4	Running	2181:2181  2888:2888  3888:3888 
 redis-1 14cab1e97c20 	redis:2.8	Running	6379:6379 
 minio-1 4d9afda2e211 	minio/minio:RELEASE.2018-07-13T00-0	Running	9001:9000 
 invoker2-1 baedb2e67ec8 	openwhisk/invoke:nightly	Running	8086:8086  9334:9223 
 controller-1 d088f517fe2f 	localhost:5000/controller:v2	Running	2551:2551  8888:8888  9222:9222 
 invoker1-1 e8c467563525 	openwhisk/invoke:nightly	Running	8085:8085  9333:9222 
 invoker3-1 f54baa4d2387 	openwhisk/invoke:nightly	Running	8087:8087  9335:9224  443:443 
 apigateway-1 6fb2f6dbf18e 	openwhisk/apigateway:nightly	Running	80:80  8080:8080 

Figure 5.1: Docker setup successfully complete with all containers running

5.3.2 wsk-cli

To interact with the system `wsk-cli` is used for simple interactions such as creating actions, triggers, and rules as well as invoking singular actions. To use this tool we must first set up its interaction with our system. Firstly we must set up the `apihost` and port through the commands `wsk property set --apihost <hostname>:<port>` as shown in Figure 5.2. The `hostname` and port used for this operation should be the ones defined by the `api-gateway`. This can be checked in the `docker-compose.yaml` file or in the visual docker desktop.

```
henrique@aspireV5:~/directory$ wsk property set --apihost localhost:443
ok: whisk API host set to localhost:443
```

Figure 5.2: wsk-cli apihost setup example

To finalise the initial setup of `wsk-cli` we still need to define the `auth` key for the user that the `wsk-cli` should use. The `auth` key includes the username and password combined, these values refer to the user that is executing the operations. In any production environment, a new user should be created through the use of the `wsk-admin` tool by using the command `wskadmin user create user1` and the output of such command is the `authkey` to be used for further operations. The `auth` key includes the username and password combined, these values refer to the user that is executing the operations. The command `wsk property set --auth <username>:<password>` is used to set up the `auth`. Openwhisk offers a guest user with their respective credentials. Figure 5.3 shows an example of the Openwhisk guest user `auth` setup using the `auth` credentials for the guest user.

```
henrique@aspireV5:~/directory$ wsk property set --auth 23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpXlPkcc0Fqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP
ok: whisk auth set. Run 'wsk property get --auth' to see the new value.
```

Figure 5.3: Example of `auth` for guest user

Now that `wsk-cli` is set up we can start interacting with the system, however, we must always include the flag `-i` to all our commands since this environment operates on a self-signed certificate basis, all commands related to the `wsk-cli` are assumed to have the `-i` flag set. A self-signed certificate is a digital certificate generated and signed by the same entity that it identifies, without involving a third-party Certificate Authority. In simpler terms, when you create a self-signed certificate, you are essentially vouching for your own identity, stating that your server or website is secure and can be trusted. However, this self-asserted trust isn't backed by any external verification. For production environments, it's strongly recommended to use certificates issued by trusted Certificate Authorities. These certificates are verified by third-party organizations that follow industry standards and security best practices. This verification process builds trust with users, as their browsers automatically recognize and accept certificates from these trusted sources without showing warnings. Even in a development environment like ours, it's

important to test our system with encrypted connections (HTTPS) to ensure that they work correctly under secure conditions.

Self-signed certificates provides a way to enable HTTPS in your development environment without the need to purchase or obtain a certificate from a trusted Certificate Authority. If the flag `-i` is not used, an error will emerge as shown in Figure 5.4.

```
henrique@aspireV5:~/directory$ wsk action create fn fn.js
error: Unable to create action 'fn': Put "https://localhost:443/api/v1/namespaces/_/actions/fn?overwrite=false": x509: certificate is not valid for any names, but wanted to match localhost
Run 'wsk --help' for usage.
```

Figure 5.4: Example of not using the `-i` flag in `wsk-cli`

Action creation is done through the use of the command `wsk action create fn fn.js` as shown in Figure 5.5 where `fn` is the name of the action within the Openwhisk server and `fn.js` is the path to the file where the code of the function is located. This example action `fn` is a simple sum of two values received as input.

```
henrique@aspireV5:~/directory$ wsk action create fn fn.js -i
ok: created action fn
```

Figure 5.5: Example of creating action `fn`

Updating an action is made through the operation `wsk action update fn fn.js` to force the override of the action `fn` with the new updated code, as shown in Figure 5.6. If we were to attempt to create an action with `wsk action create fn fn.js` with the updated code an error would be present as shown in Figure 5.7.

```
henrique@aspireV5:~/directory$ wsk action update fn fn_v2.js -i
ok: updated action fn
```

Figure 5.6: Example of updating action `fn`

```
henrique@aspireV5:~/directory$ wsk action create fn fn.js -i
error: Unable to create action 'fn': resource already exists (code 5H5e2SEtJMXoawvwbF5Wo
iCaC1RaRYJK)
Run 'wsk --help' for usage.
```

Figure 5.7: Example of creating action `fn` that already exists

Deleting an action from the server we simply need use the command `wsk action delete fn`, as shown in Figure 5.8.

```
henrique@aspireV5:~/directory$ wsk action delete fn -i
ok: deleted action fn
```

Figure 5.8: Example of deleting action fn

Management of triggers and rules is made through commands `wsk trigger create fn_trigger` and `wsk rule create fn_rule fn_trigger fn -i` in order to create triggers and rules related to the `fn` action. Useful for more advanced utilization of FaaS in production environments, examples of these commands are shown in Figures 5.9 and 5.10.

```
henrique@aspireV5:~/directory$ wsk trigger create fn_trigger -i
ok: created trigger fn_trigger
```

Figure 5.9: Example of creating a trigger for fn

```
henrique@aspireV5:~/directory$ wsk rule create fn_rule fn_trigger fn -i
ok: created rule fn_rule
```

Figure 5.10: Example of rule for action fn

Invoking actions directly is done with the command `wsk action invoke fn` or in case we wish to use an existing trigger the command `wsk trigger fire fn_trigger` is used instead as shown in Figures 5.11 and 5.12, respectively.

```
henrique@aspireV5:~/directory$ wsk action invoke fn -i
ok: invoked /_/fn with id 2d5ef6ecf08947669ef6ecf089c76675
```

Figure 5.11: Example of invoking action fn with no input

```
henrique@aspireV5:~/directory$ wsk trigger fire fn_trigger -i
ok: triggered /_/fn_trigger with id aa62cd74aeeef4f43a2cd74aeeef6f43b5
```

Figure 5.12: Example of firing a trigger for action fn

The Output of these operations is the *activation_id* that is required to check the complete result of the operations through the use of the command `wsk activation get <activation_id>` as demonstrated in Figure 5.13. This shows all information related to the invocation not just the operational result of the action. If one desires just the action result one can also invoke the action with an additional flag `wsk action invoke fn -result` as shown in Figure 5.14, however, this invocation will become blocking. Blocking invocations still have an *activation_id* that can be checked afterward, but their main purpose is to get the result for the action as soon as possible on the `wsk-cli`, allowing us to gauge latency metrics for each invocation. All of these invocations are all without the use of any parameters meaning our action `fn` is constantly returning `"result": null`.

```
henrique@aspireV5:~/directory$ wsk activation get 2d5ef6ecf08947669ef6ecf089c76675 -i
ok: got activation 2d5ef6ecf08947669ef6ecf089c76675
{
  "namespace": "guest",
  "name": "fn",
  "version": "0.0.1",
  "subject": "guest",
  "activationId": "2d5ef6ecf08947669ef6ecf089c76675",
  "start": 1693412680160,
  "end": 1693412680227,
  "duration": 67,
  "statusCode": 0,
  "response": {
    "status": "success",
    "statusCode": 0,
    "success": true,
    "result": {
      "result": null
    }
  }
},
```

Figure 5.13: Example of a complete activation result

```
henrique@aspireV5:~/directory$ wsk activation result 06ee6f0278764f9fae6f0278762f9f58 -i
{
  "result": null
}
```

Figure 5.14: Example of an activation result for action fn with no input

Invocations with function parameters we must include each parameter with the command `--param` `<paramID>` `<paramValue>`. In the case of our example action `fn` where we sum two params named `num1` and `num2` we can complete our invocation with the operation `wsk action invoke fn --param num1 3 --param num2 4` for the result to be 7. Both the invocation and its result are shown in Figure 5.15. If the inputs are too massive or complicated one can also augment the invocation with a file as input instead of passing the parameters in the command line through the flag `-P`. This file should be in JSON format and have all the relevant inputs inserted as demonstrated in Figure 5.16. Combining this file with the invocation we get the command `wsk action invoke fn -P fn_input.json` as shown in Figure 5.17.

```
henrique@aspireV5:~/directory$ wsk action invoke fn --param num1 3 --param num2 4 -i
ok: invoked /_/fn with id 0e1ca9d8d9d246989ca9d8d9d2f69838
henrique@aspireV5:~/directory$ wsk activation result 0e1ca9d8d9d246989ca9d8d9d2f69838 -i
{
  "result": 7
}
```

Figure 5.15: Example of invoking action fn with parameters


```
henrique@aspireV5:~/directory$ cat fn_input.json
{
  "num1": 3,
  "num2": 4
}
```

Figure 5.16: Example of a json file with inputs for action fn

```
henrique@aspireV5:~/directory$ wsk action invoke fn -P fn_input.json -i
ok: invoked /_/fn with id 7fc9234940514d6c89234940515d6c40
henrique@aspireV5:~/directory$ wsk activation result 7fc9234940514d6c89234940515d6c40 -i
{
  "result": 7
}
```

Figure 5.17: Example of invoking action fn with a file

With this a basis for interaction with the system is made, however, data measuring tools cannot use this tool. As such these tools must interact with Openwhisk more directly. Even still this work seeks to evaluate action execution, so action creation will be done using wsk-cli for ease of use and management. The wsk-cli is a powerful tool that abstracts the reality of the interaction between the user and the server. In its most simple terms, HTTP requests are made to the NGINX server or api-gateway in the case of our development. Understanding these requests are key to properly evaluating our system using state-of-the-art tools.

5.3.3 Curl and Base Evaluation

Curl, short for `Client URL`, is a command-line tool and library for transferring data with URLs. It supports a wide range of protocols, including HTTP, HTTPS, FTP, FTPS, SCP, SFTP, LDAP, and more. It is widely used for interacting with web services, Application Programming Interfaces (APIs), and other network resources from the command line or scripts. Curl is straightforward to use and comes on many Unix-like operating systems and there even exists a Windows port nowadays, making it readily available for various tasks.

All of the Openwhisk operations can be made directly to the API gateway instead of the wsk-cli, through the use of REST call to it. REST is an architectural style defining constraints for creating and interacting with web services. RESTful APIs are designed based on these principles to allow communication between different software systems over the Internet. REST calls, also known as HTTP requests, are the methods used to interact with resources exposed by RESTful APIs.

There are several types of REST calls, each serving a specific purpose. The GET method is used to retrieve data from the server. It requests a representation of the specified resource and does not cause any changes on the server. In other words, it's used for reading data. The POST method sends data to the server to create a new resource. It typically includes data in the request body that will be used

to create the resource. It can also be used for submitting data to be processed. The PUT method is used to update or create a resource. It replaces the existing resource with the new data provided in the request. If the resource doesn't exist, it might create a new one depending on system capabilities and settings.

Action creation requests are PUT HTTP requests and can be made with curl by using by structure shown in Figure 5.18. The `-u` flag indicates the username and password for the `<user>` that is requesting the operation. In a development case, we can simply use the user `guest` and the username and password mentioned above. The `-k` flag is equivalent to the `-i` flag used previously where curl will ignore self-signed certificates. The `-d` flag is the code of the function itself and must be sent through JSON format. As such since the `fn.js` file is an executable we use the `exec`, it's a Nodejs file so we use the `nodejs:14` for its `kind`. Note that Openwhisk by default supports a variety of languages and executables, but the version of these languages must match those present within Openwhisk. In this case, Openwhisk is loaded with Nodejs version 14 and the code is compatible with this version of Nodejs so it can be used. The "code" section is simply the full extension of the functions code. Finally the actual path for the request given to curl where `fn` is the name of the action. Just like the `wsk-cli` action creating, it will generate an error if the action already exists as shown in Figure 5.19. To counteract this we update the PATH used for the action `https://localhost/api/v1/namespaces/<user>/actions/fn` by adding the flag `?overwrite=true` for a complete PATH as demonstrated in Figure 5.20. Since bigger functions would clutter the terminal we heavily encourage everyone to use a tool like `wsk-cli` to create actions.

```
henrique@aspireV5:~/directory$ curl -X PUT \  
> -H "Content-Type: application/json" \  
> -u "23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP" \  
> -k \  
> -d '{"exec": {"kind": "nodejs:14", "code": "<fn_code>"}}' \  
> https://localhost/api/v1/namespaces/guest/actions/fn  
{"annotations":[{"key":"provide-api-key","value":false},{key:"exec","value":"nodejs:14"}],  
"exec":{"kind":"nodejs:14","code":"<fn_code>","binary":false},"limits":{"concurrency":1,"logs":10,"memory":256,"timeout":60000},"name":"fn","namespace":"guest","parameters":[],  
"publish":false,"updated":1693415781176,"version":"0.0.1"}
```

Figure 5.18: Example of curl PUT to create action fn

```
henrique@aspireV5:~/directory$ curl -X PUT \  
> -H "Content-Type: application/json" \  
> -u "23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP" \  
> -k \  
> -d '{"exec": {"kind": "nodejs:14", "code": "<fn_code>"}}' \  
> https://localhost/api/v1/namespaces/guest/actions/fn  
{"code":"kDfpqj0U2rcB6PwmC4QzUW9WI7npeP7U","error":"resource already exists"}
```

Figure 5.19: Example of curl PUT to create an already existing action fn

```
henrique@aspireV5:~/directory$ curl -X PUT \
> -H "Content-Type: application/json" \
> -u "23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP" \
> -k \
> -d '{"exec": {"kind": "nodejs:14", "code": "<fn_code>"}}' \
> https://localhost/api/v1/namespaces/guest/actions/fn?overwrite=true
{"annotations":[{"key":"provide-api-key","value":false},{"key":"exec","value":"nodejs:14"}],"exec":{"kind":"nodejs:14","code":"<fn_code>","binary":false},"limits":{"concurrency":1,"logs":10,"memory":256,"timeout":60000},"name":"fn","namespace":"guest","parameters":[],"publish":false,"updated":1693415907670,"version":"0.0.2"}
```

Figure 5.20: Example of curl PUT to update action fn

Action invocation requests are similar and require a POST request as shown in Figure 5.21. The command is very similar to the PUT operation, however, the `-d` flag is used for the parameters that action invocation might require. The response body given by Curl for the POST request solely contains the `activationId` and just as when utilizing `wsk-cli` this ID can be used to get the result of the activation when it is completed. To achieve this in Curl we must do a GET request as shown in Figure 5.22, where the PATH used is:

`https://localhost/api/v1/namespaces/<user>/activation/<activationID>`.

This operation requires the proper authentication so we must include the `-k` and `-u` flags.

```
henrique@aspireV5:~/directory$ curl -X POST \
> -H "Content-Type: application/json" \
> -u "23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP" \
> -k \
> -d '{"num1" : "3", "num2" : "4"}' \
> https://localhost/api/v1/namespaces/guest/actions/fn
{"activationId":"b3f435a11a764ed9b435a11a762ed90a"}
```

Figure 5.21: Example of curl post for action fn

```
henrique@aspireV5:~/directory$ curl -X GET \
> -u "23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP" \
> -k \
> https://localhost/api/v1/namespaces/guest/activations/b3f435a11a764ed9b435a11a762ed90a
{"activationId":"b3f435a11a764ed9b435a11a762ed90a","annotations":[{"key":"path","value":"guest/fn"}, {"key":"waitTime","value":194}, {"key":"kind","value":"nodejs:14"}, {"key":"timeout","value":false}, {"key":"limits","value":{"concurrency":1,"logs":10,"memory":256,"timeout":60000}}],"duration":5,"end":1693416469211,"logs":[],"name":"fn","namespace":"guest","publish":false,"response":{"result":{"result":" 7"},"size":15,"status":"success","success":true},"start":1693416469206,"subject":"guest","version":"0.0.1"}
```

Figure 5.22: Example of curl get for the result of invoking action fn

With this, we can start evaluating Openwhisk by utilizing the `-w` and `-v` flags of curl to analyze request time latency and request information respectively. Note that requests without the `blocking` flag of Openwhisk will not return the proper execution time, as without it the request simply returns

the *activationID* that the user can use to check the activation result when he deems it necessary. For evaluation purposes we want all our requests to be blocking to properly evaluate execution and latency times. The `-w` flag gets the operation total execution time, but we must retrieve it in a readable way so it is common to add a human intuitive text to the flag like, for example, is the actual return value for the use of the flag. The `-v` flag offers a lot of connection-related information such as IPs used and type of connection used like UDP or TCP. This is mostly valuable for testing more advanced deployments where multiple servers are located in different networks. Our local deployment is a very simple network so information from this flag is limited. To achieve this we simply add `?blocking=true` at the end of the request path as Figure 5.23 shows. The response body will contain the same information as the GET operation for the *activationID* of the invocation followed by any additional information provided from the additional flags like `-w` and `-v`.

```
henrique@asptreV5:~/directory$ curl -X POST \
> -H "Content-Type: application/json" \
> -u "23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP" \
> -k \
> -d '{"num1" : "3", "num2" : "4"}' \
> -w "\nResponse time: %{time_total} seconds\n" \
> https://localhost/api/v1/namespaces/guest/actions/fn?blocking=true
{"activationId":"5cb113ac20da47b6b113ac20da67b673","annotations":[{"key":"path","value":"guest/fn"}, {"key":"waitTime","value":139}, {"key":"kind","value":"nodejs:14"}, {"key":"timeout","value":false}, {"key":"limits","value":{"concurrency":1,"logs":10,"memory":256,"timeout":60000}}], "duration":6, "end":1693416632177, "logs":[], "name":"fn", "namespace":"guest", "publish":false, "response":{"result":{"result":" 7 "}, "size":15, "status":"success", "success":true}, "start":1693416632171, "subject":"guest", "version":"0.0.1"}
Response time: 0,179762 seconds
```

Figure 5.23: Example of curl post for action fn with blocking and timer

While Curl allows us to evaluate requests it is not enough as we wish to test the system during multiple types of workload scenarios. To do so we employ a proper evaluation tool such as Apache Jmeter.

5.3.4 Jmeter and load evaluation

JMeter is an open-source performance testing tool designed to test the performance, load, and stress of web applications, APIs, databases, and other network services. It allows you to simulate a large number of users interacting with your application to measure its performance and identify potential bottlenecks or issues under different load conditions. JMeter supports a wide range of protocols and technologies, including HTTP, HTTPS, FTP, JDBC, SOAP, REST, JMS, and more. This makes it suitable for testing various types of applications and services, just like Curl. JMeter can simulate thousands of virtual users concurrently, allowing you to test how your application performs under different levels of load and stress. To fully use Jmeter in our evaluation process we first had to create a testPlan. Each

test can be saved separately and rerun in the future for further data analysis. Within a testPlan our main tool is the threadGroup, allowing large amounts of controller instances of HTTP requests in our case. We can adjust both the number of concurrent threads (users) as well as the number of executions. This will allow us to easily keep track of different test cases. Within the threadGroup we need three main components: HTTP Header Manager, HTTP Authorization Manager, and HTTP request. The HTTP Header Manager must be added and modified to include the content type of the request. In our case, it needs a new Item with the name `Content-type` and value `application/json`.

The HTTP Authorization Manager is where the username and password of our user must be added. Multiple of these authorizations can be added depending on the number of different users that are used in the tests. We simply add a new Item and adjust the username and password to the desired user. For development and evaluation purposes we can use the guest user provided by Openwhisk. The mechanism must also be changed to `BASIC_DIGEST`, the remaining options can be left empty.

Finally the HTTP request, we can add as many entries of this item as we want to different types of requests. The number of these HTTP requests will vary in our case when we desire a different number of action types within a singular test. Optionally we can add Aggregate Graphs, View Results Tree, View Results Table, and Aggregate Report to aid us in our data collection. Aggregates are used to view data of the test such as latencies, percentage error, Throughputs, and medians while Viewers are used to check the results of singular requests to help identify issues and order of consumption. Timers may also be used to add variety to the tests, such as request delays for a random or static amount of time. Within the HTTP request item, we need to properly set it up. The “webserver” section needs to be updated with the servername or IP and the port of the API-gateway container. Assuming the previous curl execution of the action `fn` it would be `localhost` and `443` respectively. For the “requests” section itself the protocol needs to be `https` and the method is `POST` for action invocations, `PUT` for action creations, and `GET` for activation results. The path is the same as the one used for curl without the initial section, meaning `/api/v1/namespaces/<user>/actions/fn?blocking=true`, where `<user>` is the user and can be substituted for `guest` in development or evaluation scenarios. Additionally, if the request needs arguments you can send parameters directly, but it’s not recommended. Instead, we will add JSON file items to the “Send Files With Request” section with our required arguments for the action. With this set up we can properly evaluate our Openwhisk system when it comes to the client side of the requests. Examples of the HTTP Header Manager, HTTP Authorization Manager, and HTTP request are shown in Figures 5.24, 5.25, and 5.26, respectively.

HTTP Header Manager		
Name: HTTP Header Manager		
Comments:		
Headers Stored in the Header Manager		
Name	Value	
Content-type	application/json	

Figure 5.24: Example of a setup of a Jmeter HTTP Header Manager

HTTP Authorization Manager

Name: HTTP Authorization Manager

Comments:

Options

☐ Clear auth on each iteration?

Authorizations Stored in the Authorization Manager

Base URL	Username	Password	Domain	Realm	Mechanism
	23bc46b1-71f6-4ed5-8c54-816aa4f...			BASIC_DIGEST

Figure 5.25: Example of a setup of a Jmeter HTTP Authorization Manager

HTTP Request

Name: HTTP Request

Comments:

Web Server

Server Name or IP: localhost Port Number: 443

Timeouts (milliseconds)

Connect: Response:

HTTP Request

Implementation: Protocol [http]: https Method: POST Content encoding:

Path: /api/v1/namespaces/guest/actions/fn?blocking=true

☐ Redirect Automatically ☒ Follow Redirects ☒ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

Parameters Body Data

Send Parameters With the Request:

Name	Value	Encode?	Include Equals?

Detail Add Add from Clipboard Delete Up Down

Send Files With the Request:

File Path	Parameter Name	MIME Type
/home/henrique/directory/fn_input.json		

Add Browse... Delete

Figure 5.26: Example of a setup of a Jmeter HTTP request

5.3.5 Actions used

For evaluation, we will vary our testing by using a variety of actions. These actions as described in Section 5.1, have varied performance differences and seek to analyze our system in as many ways as possible. For fast and simple actions F1 was used. This action seeks to represent fast trigger executions and is the common staple of FaaS user event systems. It represents operations like File hashing. For sleep type functions F2 and F3 were utilized. These functions simply sleep the system for either 5000 ms (5 seconds) or 10000 ms (10 seconds) and will allow us to accurately detect scheduling delays present within the system if the system is not performing any CPU or Memory operations. F4 was used for CPU-intensive functions to let us know of any overall performance degradation throughout the system. The F4 function used was a recursive Fibonacci series. The Fibonacci series is a sequence of numbers in which each number is the sum of the two preceding ones. It starts with 0 and 1, and then each subsequent number is the sum of the previous two. Here's how the sequence is generated:

1. **Base Cases:** The Fibonacci sequence starts with the first two terms: $F(0) = 0$ and $F(1) = 1$.

These are the base cases of the sequence.

2. **Recurrence Relation:** Each subsequent term is generated by adding the two previous terms. In mathematical terms, if we denote the n th Fibonacci sequence term as $F(n)$, where $n \in \mathbb{N}_0$ is the position of the term in the sequence (starting from 0), then the recurrence relation is:

$$F(n) = F(n - 1) + F(n - 2)$$

For example, $F(2) = F(1) + F(0) = 1 + 0 = 1$, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on.

In our evaluation, the Fibonacci of 42 was used due to it being a high number for the complexity desired for our executing times. This operation heavily simulates image classification workloads due to its computational complexity.

For all actions, there exists the Default, Base, and Spread versions. The Default version is the action on the original version of Openwhisk. The Base version is the same type of action as the Default but it is run on our version of the system with no additional inputs or modifications to the invocation. Both of these should offer the same execution results, in both result and execution time. It is used to measure our system scheduling delay compared to the Default version of Openwhisk. The Spread version is the same action as both the Default and Base version but its invocation requests the use of our newly added functionality. The action result should be the same but the execution time may vary depending on the circumstances. These circumstances are extensively explored during the tests.

All actions are created during the setup of the test and this extra execution time is not considered for the test as it bears no interaction with the modified locations of our newly updated system.

5.3.6 Sub-Enviroments

A set of two sub-environments were made to test our enhanced scheduler. These sub-environments reference the initial state of the system immediately before the execution of a given test.

Cold Sub-enviroment “C”: We sought to evaluate our system as the worst case possible where all currently existing warm containers within the invokers mismatch the invoked action. This will allow us to evaluate our system when handling cold invocations, and how well it successfully warms up the system to generate the best user experience. This was achieved through the mass invocation of a “hello world” action which simply returns “hello user” to the user. The mass invocation comprises 100 parallel invocation calls using JMeter, by setting up a thread group with 100 users and 1 call each. The execution of the tests ignores this environment setup and it's done after all containers within the invokers enter the paused state just as shown in Figure 5.27.

Warm Sub-environment “W”: A fully cold environment it’s not entirely realistic as prewarm and warm containers contribute heavily towards faster request execution times and are the backbone of FaaS systems. As such for the same set of tests as the sub-environment 1, we evaluated our system under a warm environment where only prewarm and warm containers of the action to be invoked were present. In the same way as the sub-environment 1 was achieved the warm environment was made with 100 concurrent calls for the specific action related to the test. Once again this execution time was not taken into consideration during the test. Jmeter was set up with 100 users with one HTTP request each.



















Name	Image	Status	Port(s)	Started
 wsk2_29_guest_hello 6e3c1bc8c516 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk1_33_guest_hello c2d007bc620c 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk3_28_guest_hello 52aa9477aca5 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk3_29_guest_hello b01044d61c42 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk1_34_guest_hello 733d10fc192e 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk3_30_guest_hello 68d03798e5eb 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk2_31_guest_hello 77e63aed65a3 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk1_35_guest_hello 12682976dcad 	openwhisk/action-nodejs-v14:nightly	Paused		
 wsk3_32_guest_hello 136b28fd7e58 	openwhisk/action-nodejs-v14:nightly	Paused		

Figure 5.27: Docker setup for a cold environment

5.3.7 Hardware used

Two different pieces of hardware were used for testing to accurately determine potential system degradation caused by our scheduler.

Hardware A: A laptop with an Intel® Core™ i7-6700HQ CPU @ 2.60GHz processor with 4 physical cores and 8 threads on Ubuntu 20.04.6 LTS 64-bit. Most of the testing was done in this hardware due to its ease of access and testing environment. Important to note that only 1 data bus exists within the hardware meaning all interaction between threads and memory is centralized.

Hardware B: To have access to results closer to a production environment we utilized much stronger tools with access to more CPU cores. This hardware B uses a Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 3192, 8 Physical Core(s), 18 Logical Processor(s).

5.4 Performance evaluation

A total of 6 tests were made to evaluate our newly augmented scheduler. These tests vary in both Sub-environment and hardware. Each test is referred to by the test number, which Sub-Environment it uses followed by which hardware it utilizes, for example, “Test 1, W-A” is test number 1 and uses both a Warm Sub-environment and hardware A. An additional evaluation of our utility function is made to analyze its behavior if applied in our tests. Following this section, an analysis of all of the tests’ unique results is made that more deeply analyses the behavior of the utility function given the outcome of the tests.

5.4.1 Test 1, W-A

Test 1, W-A, seeks to evaluate our system in the same conditions as the base version of Openwhisk. A total of 30 invocations were made non-concurrently with a 2000 ms (2 seconds) between each call to allow the system to reuse the same resources and safely evaluate the scheduling delay. Jmeter was set up with 1 Thread group comprised of 1 user and 30 calls of a single HTTP request of the action. The action used was F1 which is the simple and fast function, as the actual execution time of the action is irrelevant for this test. As we seek to evaluate the system in the same conditions we did not use the additional spreading functionality of our system. The results of this test are shown the Table 5.1 with “Base” referring to the base version of Openwhisk and “Standard” to the updated version without the use of any spreading functionality.

	average latency	median	99% line	variance	extra invoker calls	total time
Default	234	155	2808	479	0	65097
Base	206	134	2538	427	0	64353

Table 5.1: Comparison between the different possible schedulers

As expected the latencies and total execution times are very close to each other. This shows that the systems are very comparable when it comes to scheduling delays under the same conditions. Our updated version shows very slightly better results probably due to the more optimized tail function used for scheduling, but as the results are so similar to one another it could just as likely be a faster cold container creating as the difference between total execution time is close to the same as the difference between 99% lines. However, since the relative difference between 99% lines is very slight it wouldn't

indicate anything of note while the absolute difference would generate considerable changes in the average latencies. The difference between the 99% line and the median is also very large, indicating that only very few invocations resulted in cold starts.

This allows us to safely conclude that we can use our new version of the system without worries of additional overhead if we use it for the same operations we would use the old version of the system. For the remainder of this section the Base version of Openwhisk results will be omitted as they were consistently similar to the Standard operation of our updated version and any conclusions made through comparing our additional functionality and the base version would be the same as comparing it with the Standard operation of our version.

5.4.2 Test 2, C-A

For this test, we evaluate our system during the ideal scenario where the user uses the system lightly before an abundance of similar requests. This work primarily seeks to better the user experience by allowing him to for a reduced cost better prepare the system for incoming overloading scenarios. To achieve this ideal scenario the test is subdivided into two sections.

The first is the start-up section where the user lightly uses our system. This is achieved through the execution of 5 concurrent requests of the action. The second section is the overloading scenario. To achieve such an overload scenario 20 concurrent requests of the same action are made. The second set of requests is made 15000 ms (15 seconds) after the first set of requests is made. Jmeter was set up with two distinct thread groups, one for each set of requests. Where the seconds set has an added scheduling delay of 15 seconds. The second set of requests is the same between tests and only the first set is modified to either use or not our newly implemented spreading functionality.

In this test, we seek to observe a faster total execution time for our version as there are expected to be fewer cold starts for the second set. The results for this test are shown in Table 5.2

	average latency	median	99% line	variance	extra invoker calls	total time
Base F1	5243	4186	9156	2742	0	24507
Spread F1	2216	655	10379	3377	8	17002
Base F2	14410	12413	24158	6153	0	39009
Spread F2	19821	19524	29214	3802	9	37952
Base F4	29720	29720	41610	10077	0	61223
Spread F4	38330	36376	49497	7653	9	66829

Table 5.2: Reduced colds starts evaluation

For action F1 which is a fast execution action we can see great improvements to the total execution time. As we can also observe the difference between 99% and median is much higher for the test made by our additional functionality showing much lower amounts of cold starts while the 99% difference

between the tests remains very similar meaning the cold starts themselves generated largely the same latency.

For action F2 we can see slight improvements but due to the nature of delay functions the total execution time wouldn't change much since the delay of 5000 ms (5 seconds) is higher than the cold container start-up. Still, we can see that the system got quite overloaded and many delay functions were queued up as the average delay was much higher than 5000 ms.

For action F4 we can see major issues in all aspects of the data. This is due to the set-up phase executions exceeding the 15-second startup, making the overloading section actions queue after the set-up section actions, which is not ideal. However, we can still observe that the primary culprit for the additional total execution time is the setup since the variance is much lower and the 99% line is higher.

This test allows us to conclude two aspects. First, if the setup phase of the action is longer than the execution of the action then the system greatly improves cold start delay impacts during follow-up overloading scenarios. Second, due to the hardware used to test this system being weak for such parallel execution, a concurrent operation such as the start-up of the F3 gets greatly slower. This system would have to be deployed in a much higher parallel capacity environment for the concurrency issue to be minimized, as the results of one container execution time got worse if other containers on different invokers were also running, which should be impossible if the system on the scale observed if the containers/invokers were truly isolated and running in completely separate threads since only the connection overhead should be noticed and not scale with the execution time of the action.

5.4.3 Test 3, W-A

This test was made within a warm environment. A total of 30 action invocations were made non-concurrently without made timer between them. This allows us to evaluate if our system is affected by the type of action executed and any bottlenecks or overhead present within it. In Table 5.3 we can observe the result of this test.

	average latency	median	99% line	variance	invoker calls	total time
Base F1	267	195	723	155	0	8140
Spread F1	339	292	699	120	60	10244
Base F4	5655	5579	6358	326	0	169699
Spread F4	8189	7986	9656	754	60	245733
Base F2	5480	5403	6339	216	0	164519
Spread F2	5583	5576	5817	82	60	167596
Base F3	10315	10302	10649	89	0	309507
Spread F3	10681	10541	12417	467	60	320548

Table 5.3: Evaluation of the scheduling delay present

From the results for the tests of F1 and F4, we could see a great difference in total execution time

and average latency. However, this is hard to be given the association to the scheduling overhead of our system since it seems to scale with the execution time of the action performed. As for the cold-starts, 99% lines were much slower for the spreading testing when the invokers were untouched with our new version. We can safely assume that the hardware that the system we tested on was insufficient for such concurrent operations that spread demands. To attest to this, the test 4 will further evaluate this phenomenon.

The results for tests of F2 and F3 we can see much closer values since sleep functions are much more easily made concurrent due to the lack of CPU-intensive operations if a system lacks concurrency hardware. We can also see a lower deviation for the spreading functionality since it is collecting the fastest output of the 3 invokers leading to less variance in invoker performance. This allows us to conclude that in sleep-like functions where the difference between cold starts and warm starts is smaller than the sleep amount the spread functionality neither provides tangible benefits nor considerable overhead.

Also within all results of this test, the number of invoker calls remained the same meaning the system during spreading operations fully utilized all 3 invokers during all 30 invocations. This is the expected result but it still serves as confirmation that the system is working as intended.

5.4.4 Test 4, W-A

As shown in the last test there are some doubts if the lack of performance is due to the scheduling system or simply due to the extra concurrent workload given to the hardware to do. To better test this doubt we set up the same environment as the last test but added a new set of situations where we try to simulate the spreading operation using the standard scheduling system through Jmeter. Since the last test was made with 1 user and 30 requests non-concurrently, we simply adjust the number of users to 3 and that makes it so that we have 30 non-concurrent sets of 3 concurrent invocations. This will accurately simulate the number of extra invocations the spreading functionality does plus keep the total number of invocations made by the system also the same. The values for this test are shown in Table 5.4.

	average latency	median	99% line	variance	invoker calls	total time
Base F1	267	195	723	155	0	8140
Spread F1	339	292	699	120	60	10244
Base F1x3	342	288	724	161	0	10133
Base F4	5655	5579	6358	326	0	169699
Spread F4	8189	7986	9656	754	60	245733
Base F4x3	8032	7863	9555	825	0	239233

Table 5.4: Control test to check parallelism capabilities of the scheduler

As we can safely ascertain from this test, the hardware is unable to take care of concurrent requests truly isolated from each other. The number of concurrent requests in both the spreading tests and 3 user tests are the same making both demonstrate the same values. This also confirms that our scheduling system is comparable in execution time given the same conditions, those being 30 requests of 3 concurrent invocations each. Making our system offer a less expensive version if the user simply wants the extra invocations into creating warm containers and the company providing the service uses our utility function to determine its price.

5.4.5 Test 5, W-B

For this test, we redid test 4 using this new hardware. We sought to find if the increased latency and total execution time found in test 4 were due to the scheduler itself or the hardware being unable to handle such load. Table 5.5 shows the results for this test.

	average latency	median	99% line	variance	invoker calls	total time
Base F1	151	149	159	12	0	11655
Spread F1	167	157	233	98	60	12655
Base F1x3	181	133	337	180	0	14095
Base F4	2880	2857	3309	164	0	84625
Spread F4	3372	3213	3654	215	60	102948
Base F4x3	3780	3733	4109	365	0	114158

Table 5.5: Same executions as test 4 but using hardware B

We can still see that Openwhisk degrades with an abundance of requests even with stronger hardware. However, we can see improvements in the Spread functionality compared to the Base x3. Due to the optimization made where the fastest response from the Invokers is the one the user consumes and the access to more parallelism functionality from the stronger hardware, we can see a much lower variance. Since the Base x3 has to receive requests from all of the responses if an Invoker gets overloaded it will still need to wait for the requests from it while the Spread functionality will simply skip over the additional requests. This test made it clear that the hardware/scale of the environment required to see an improvement for the functionality of retrieving the fastest response from all Invokers to offer benefits is very high. This makes us aware that the system is simply overloaded with parallel requests instead of the scheduler itself generating the overhead.

5.4.6 Test 6, C-B

Finally, we still wish to see if using stronger hardware still offers an improvement for the main attraction of the Spread functionality of setting up warm containers for a future abundance of requests for a

cheaper price. To achieve this we redid test 2 on this stronger hardware and its values are shown in Table 5.6.

	average latency	median	99% line	variance	invoker calls	total time
Base F1	1011	236	2366	906	0	17016
Spread F1	607	169	2459	838	9	15236
Base F2	13429	11422	24163	7956	0	38164
Spread F2	18821	17995	29111	3645	10	36346
Base F4	3922	3409	6197	2269	0	21089
Spread F4	3478	2885	5587	2364	10	20560

Table 5.6: Same executions as test 2 but using hardware B

For action F2 we can see that no substantial differences were found between the two hardware as expected. Maintaining both executions similar since the cold start delay is not very noticeable for a 10-second sleep function, as well as confirming that hardware A while weaker has little issues were executing sleep function in parallel.

For action F1 the improvement was noticeable in absolute values, meaning the actual total execution time was lower for this stronger hardware compared to hardware A, however, the relative improvement was smaller. This is probably due to the warm start and cold start execution time difference being larger in hardware A.

As for action F4 we can observe an improvement in this new hardware. Due to its greater parallel computation capabilities the execution times of the setup phase of the test no longer exceed 15000 ms (15 seconds), allowing the creation of warm containers with no delay on the total execution time. This allows our scheduling system to show better results while when using hardware A we saw a reduced efficiency.

5.5 Utility Function Evaluation

While the performance of the extended scheduler is crucial we must also evaluate and analyze how our utility function and the final cost to the client vary with the use of the Equation (3.1) shown in the architecture Section 3. The primary values our clients are interested in are how much the latency, total execution time, and final cost vary when using the new scheduling option. As for the provider, the extra resources consumed during the operation and the α used for the utility function are the most important factors. The client would generally want higher latency and total time decrease while paying the least amount. As for the provider, he would want the least amount of extra resources and the highest α that the clients would still be paying for the service. Table 5.7 contains all of the previously done relevant tests and evaluations of the above factors. The values of Latency decrease, Total time decrease, Extra

resources, and Cost are relative compared between the Base version of the test and our enhanced schedulers, with the Base version as the absolute value. For example, if our scheduler used a total of 10 seconds to execute and the Base version 20 seconds instead, then there was a two times (2x) decrease in total execution time. This is done for easier comparison between extra resources consumed and improvement. Since using, two times the resources for two times the speed is simpler for clients to understand instead of twice the resources for half the time.

Table 5.7: Utility evaluation

Test	Latency decrease	Total time decrease	Extra resources	α	Cost
2 – F1	2.37x	1.44x	1.32x	0.8	1.06x
				0.6	1.13x
				0.4	1.19x
2 – F2	0.73x	1.03x	1.36x	0.8	1.07x
				0.6	1.14x
				0.4	1.21x
2 – F4	0.76x	0.92x	1.36x	0.8	1.07x
				0.6	1.14x
				0.4	1.21x
3 – F1	0.78x	0.80x	3x	0.8	1.4x
				0.6	1.8x
				0.4	2.2x
3 – F2	0.98x	0.98x	3x	0.8	1.4x
				0.6	1.8x
				0.4	2.2x
6 – F1	1.67x	1.12x	1.36x	0.8	1.06x
				0.6	1.14x
				0.4	1.19x
6 – F2	0.71x	1.05x	1.4x	0.8	1.08x
				0.6	1.16x
				0.4	1.24x
6 – F4	1.13x	1.03x	1.4x	0.8	1.08x
				0.6	1.16x
				0.4	1.24x

We can observe that our new enhanced scheduler must be used with care and awareness as it is only beneficial in certain situations. We can see that in the case of an already warm environment such as test 3, the performance degradation of additional invocations takes quite a heavy toll on the system and only serves to promote worse performance values across the board. However, we can also observe that due to the abundance of additional resources used (3 times the amount) the α determined by the seller can heavily sway the final cost. This will allow unexpected or undesired uses of our scheduler to be mitigated should the client negotiate with the seller allowing a more positive interaction between the two.

We can also see that depending on what hardware is used the benefits can vary. Test 2 uses hardware A while test 6 uses hardware B. F1's benefit is greater in the weaker hardware A. F2 saw no

change between hardware, but we can see the latency decrease being an issue while the total execution time remains largely untouched. On the other hand F4 on hardware A is severely slower in all aspects being mostly a detriment to the use of the enhanced scheduler while in hardware B we can see some improvements. While the latency decrease for test 6 action F4 was a small 1.13 times compared to the real extra resources of 1.4 times more, depending on the α employed by the seller the trade might still be beneficial for the client. For example, if the α value used was 0.6 then the latency decrease would equally match the extra cost while maintaining the overall cost of the resources cheaper since the true cost for the seller would be 1.4 times more. If the α used was 0.8 then it would become beneficial as long as the client did not prioritize total execution time.

This combination of allowing the client to choose between two options and the modification of the α used for the seller always provides a two-way negotiation in the case of a misuse of the functionality. However, if the client is smart then situations such as test 2 action F1 can arise where no matter the α chosen by the seller the increased performance will always outpace the extra cost, making the extra resources effectively cheaper for the client.

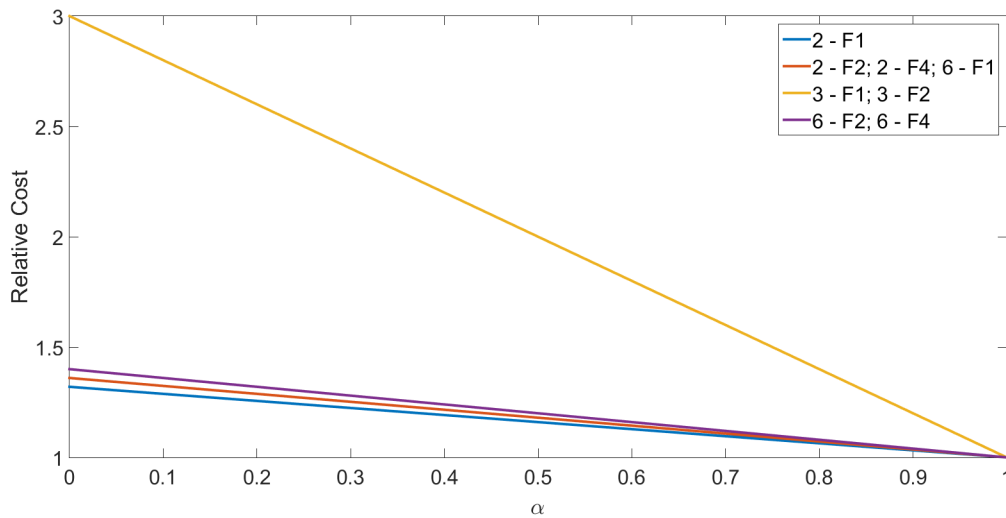


Figure 5.28: Cost's behaviour depending on α values

In Figure 5.28, we can see that the knowledge of the type of environment located within the system can heavily alter how much control the seller has over the final cost of the request. In the cases where the functionality is used in a cold environment, where it is expected to be used, the leverage presented to the seller is reduced thus making the use of the functionality more consistent during its expected environment. In cases where the functionality is misused such as a completely warm environment then the seller who was probably taken by surprise by the amount of additional resources consumed for little to no benefit is allowed a lot more leverage to control the final cost.

5.6 Analysis and Discussion

After all these tests and evaluations, we can observe that the use of our enhanced scheduler needs attention to a lot more factors than just knowledge of the requested action. Knowing the current state of the system is important as our enhanced scheduler mainly provides benefits if an action does not have an already existing warm container within the system. As such only requesting initial requests with the new spread functionality would be expected to provide the most amount of benefits. Clients should also be aware of the hardware in which the action is expected to be executed. Weaker hardware might just be overwhelmed when using CPU-intensive actions and reduce overall performance. However, one can also take advantage of weaker hardware and abuse its value to have greater relative improvements as shown in the use of action F1 in tests 2 and 6.

Our enhanced scheduler requires transparency from the seller for the best quality of service. All of the information of the system state and action spread-ability is easily checked through the use of logs. Lack of transparency or clarity of the system's current state may result in the client paying more for less performance, in the case of a warm environment, leading to less trust from the client.

Our enhanced scheduler is flexible and customizable for both the client and seller. The client can only incorporate the functionality in the actions it deems valuable and be rewarded. The seller may modify the α present in the utility function to customize the balance between the profit margins it desires and client satisfaction.

Finally, our enhanced scheduler easily complements the already existing Apache Openwhisk allowing previous workflows that use the original version to be unaffected in both performance and complexity. Allowing the seller to easily dictate the number of resources available and increase or decrease them.

6

Conclusion

Contents

6.1 Future work	79
---------------------------	----

Our work describe the current state of cloud computing's Function-as-a-Service technology and some of its key benefits and difficulties. To better understand the common customer concerns and desires, and to better assess our requirements, we also examined the cutting-edge scheduling and pricing mechanisms utilized throughout our cloud computing.

We created a scheduler extension architecture that considers user preferences when adjusting scheduling to provide a higher quality of service to the user. Apache Openwhisk was used to implement our solution. For over-provisioned system conditions a new functionality that we named "Action-Spreading" was implemented to allow warm containers to be set up for a reduced cost in preparation for an influx of requests. For an under-provisioned system state originally we intended to implement a priority-aware scheduling extension, however throughout the implementation process it exceeded our expected architectural complexity and we deemed it outside of the scope of this work. Finally, we evaluated our enhanced scheduler through a series of tests.

We concluded that under over-provisioned system conditions, it provided a substantial benefit for the client with a latency decrease of up to 2.37 times for only a maximum of 30% additional cost. We also were able to conclude that should the scheduler be used under unforeseen system conditions it allows for a positive client-seller solution through the use of the proposed utility function management.

6.1 Future work

This work was able to successfully develop an enhanced scheduler that allowed for a better quality of service for FaaS clients by providing more consumer options that would benefit the client mainly during over-provisioned server conditions. However, there were some roadblocks and undesired results discovered throughout the development of this enhanced scheduler.

Some undesired results were ascertained from the "Action-Spreading" functionality such as the lack of performance during a server state where an abundance of warm containers was available. While we were able to observe that the primary bottleneck was hardware performance degradation it was an unsatisfactory result. Perhaps in the future, if we were to be able to have access to more advanced clusters with truly independent machines then we would be able to further test and optimize this scheduler to have our initial desired results.

One of the primary roadblocks was the unexpected complexity of the priority-aware extension we initially desired for our enhanced scheduler. To us, the idea of a priority-aware scheduling system is still a very relevant idea for cloud systems like FaaS even if we were unable to implement it. Since systems like Apache Openwhisk were made with quickness in mind when initially developed they prioritized tools that enhanced such traits, for example, Kafka. The effort required to circumvent the original design of these systems proved to be far greater than initially expected.

Maybe the development of a complete substitute for Kafka and other similar components that don't have priority-aware mechanics in mind specifically for FaaS systems would be a great avenue for future research in this field.

Bibliography

- [1] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *Proceedings of the IEEE World Congress on Services (SERVICES)*, vol. 2642, 2019, pp. 206–211.
- [2] R. Mukundand and R. Bharati, "Function as a service in cloud computing: A survey," *International Journal of Future Generation Communication and Networking*, vol. 13, no. 3, pp. 3291–3297, 2020.
- [3] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *Proceedings of the IEEE International Conference on Fog Computing (ICFC)*, 2020, pp. 17–24.
- [4] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: efficient scheduling and autonomous resource management in serverless environments," in *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 1–10.
- [5] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Automated fine-grained cpu cap control in serverless computing platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2289–2301, 2020.
- [6] J. Simão and L. Veiga, "Partial utility-driven scheduling for flexible SLA and pricing arbitration in clouds," *IEEE Trans. Cloud Comput.*, vol. 4, no. 4, pp. 467–480, 2016. [Online]. Available: <https://doi.org/10.1109/TCC.2014.2372753>
- [7] I. Astrova, A. Koschel, M. Schaaf, S. Klassen, and K. Jdiya, "Serverless, faas and why organizations need them," *Intelligent Decision Technologies*, vol. 15, no. 4, pp. 825–838, 2021.
- [8] M. Roberts, "Serverless architectures," 2018, accessed on 2022-11-15. [Online]. Available: <https://martinfowler.com/articles/serverless.html>
- [9] B. Janakiraman, "Serverless," 2016, accessed on 2022-11-15. [Online]. Available: <https://martinfowler.com/bliki/Serverless.html>

- [10] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.
- [11] S. K. Mohanty, G. Premsankar, and M. di Francesco, "An evaluation of open source serverless computing frameworks," in *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2018, pp. 115–120.
- [12] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," in *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE/ACM, 2018, pp. 161–168.
- [13] D. Bermbach, F. Pallas, D. G. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai, "A research perspective on fog computing," in *Proceedings of the International Conference on Service-Oriented Computing*. Springer, 2018, pp. 198–210.
- [14] Z. Laaroussi, R. Morabito, and T. Taleb, "Service provisioning in vehicular networks through edge and cloud: An empirical analysis," in *Proceedings of the IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, 2018, pp. 1–6.
- [15] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *Proceedings of the IEEE International Systems Engineering Symposium (ISSE)*. IEEE, 2017, pp. 1–7.
- [16] S. Quevedo, F. Merchán, R. Rivadeneira, and F. Dominguez, "Evaluating apache openwhisk - faas," in *IEEE Ecuador Technical Chapters Meeting (ETCM)*. IEEE, 2019, pp. 1–5.
- [17] G. R. Russo, A. Milani, S. Iannucci, and V. Cardellini, "Towards qos-aware function composition scheduling in apache openwhisk," in *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 2022, pp. 693–698.
- [18] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, " λ dnn: Achieving predictable distributed dnn training with serverless architectures," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 450–463, 2021.
- [19] Z. Tu, M. Li, and J. Lin, "Pay-per-request deployment of neural network models using serverless architectures," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. ACL, 2018, pp. 6–10.
- [20] A. SageMaker, accessed on 2022-12-5. [Online]. Available: <https://aws.amazon.com/pm/sagemaker/>

- [21] AutoGluon, accessed on 2022-12-5. [Online]. Available: <https://auto.gluon.ai/stable/index.html>
- [22] Y. Bouizem, "Fault tolerance in faas environments," Ph.D. dissertation, Université Rennes 1, 2022.
- [23] R. Chard, T. J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, and K. Chard, "Serverless supercomputing: high performance function as a service for science," *ArXiv*, vol. abs/1908.04907, 2019.
- [24] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020.
- [25] E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A spec rg cloud group's vision on the performance challenges of faas cloud architectures," in *Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 21–24.
- [26] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: a reinforcement learning approach," *IEEE Internet of Things Journal*, vol. 10, no. 5, pp. 3917–3927, 2023.
- [27] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in faas services," in *Proceedings of the ACM Symposium on Applied Computing*. ACM, 2020, pp. 134–143.
- [28] S. Shillaker, "A provider-friendly serverless framework for latency-critical applications," in *Proceedings of the Eurosys Doctoral Workshop*. ACM, 2018, p. 71.
- [29] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: function composition for serverless computing," in *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2017, pp. 89–103.
- [30] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, "Accelerating serverless computing by harvesting idle resources," in *Proceedings of the ACM Web Conference*. ACM, 2022, pp. 1741–1751.
- [31] Y. Wang, K. Arya, M. Kogias, M. Vanga, A. Bhandari, N. J. Yadwadkar, S. Sen, S. Elnikety, C. Kozyrakis, and R. Bianchini, "Smartharvest: harvesting idle cpus safely and efficiently in the cloud," in *Proceedings of the European Conference on Computer Systems*. ACM, 2021, pp. 1–16.
- [32] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in

- Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 199–208.
- [33] Aqua, accessed on 2022-12-5. [Online]. Available: <https://www.aquasec.com/aqua-cloud-native-security-platform/>
- [34] Snyk, accessed on 2022-12-5. [Online]. Available: <https://snyk.io/>
- [35] D. S. Jegan, L. Wang, S. Bhagat, T. Ristenpart, and M. Swift, “Guarding serverless applications with seclambda,” *arXiv*, vol. 2011.05322, 2020.
- [36] A. Sankaran, P. Datta, and A. Bates, “Workflow integration alleviates identity and access management in serverless computing,” in *Proceedings of the Annual Computer Security Applications Conference*. IEEE, 2020, pp. 496–509.
- [37] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabah, A. Slominski *et al.*, “Serverless computing: current trends and open problems,” in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.
- [38] N. Kratzke, “A brief history of cloud application architectures,” *Applied Sciences*, vol. 8, no. 8, p. 1368, 2018.
- [39] M. Sewak and S. Singh, “Winning in the era of serverless computing and function as a service,” in *Proceedings of the IEEE International Conference for Convergence in Technology (I2CT)*. IEEE, 2018, pp. 1–5.
- [40] L. Lin, P. Li, J. Xiong, and M. Lin, “Distributed and application-aware task scheduling in edge-clouds,” in *Proceedings of the International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*. IEEE, 2018, pp. 165–170.
- [41] A. Madej, N. Wang, N. Athanasopoulos, R. Ranjan, and B. Varghese, “Priority-based fair scheduling in edge computing,” in *Proceedings of the IEEE International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2020, pp. 39–48.
- [42] S. R. Dibaj, L. Sharifi, A. Miri, J. Zhou, and A. Aram, “Cloud computing energy efficiency and fair pricing mechanisms for smart cities,” in *IEEE Electrical Power and Energy Conference (EPEC)*. IEEE, 2018, pp. 1–6.
- [43] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in *Proceedings of the ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

- [44] G. Lee, B. Chun, and H. Katz, "Heterogeneity-aware resource allocation and scheduling in the cloud," in *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. ACM, 2011.
- [45] V. Scoca, A. Aral, I. Brandic, R. De Nicola, and R. B. Uriarte, "Scheduling latency-sensitive applications in edge computing," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*. Springer, 2018, pp. 158–168.
- [46] A. Pires, J. Simão, and L. Veiga, "Distributed and decentralized orchestration of containers on edge clouds," *J. Grid Comput.*, vol. 19, no. 3, p. 36, 2021. [Online]. Available: <https://doi.org/10.1007/s10723-021-09575-x>
- [47] L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4712–4721, 2018.
- [48] S. Mendes, J. Simão, and L. Veiga, "Oversubscribing micro-clouds with energy-aware containers scheduling," in *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*. ACM, 2019, pp. 130–137.
- [49] T. Choudhari, M. Moh, and T.-S. Moh, "Prioritized task scheduling in fog computing," in *Proceedings of the ACM Southeast Conference (ACMSE)*. ACM, 2018, pp. 1–8.
- [50] H. T. T. Binh, T. T. Anh, D. B. Son, P. A. Duc, and B. M. Nguyen, "An evolutionary algorithm for solving task scheduling problem in cloud-fog computing environment," in *Proceedings of the International Symposium on Information and Communication Technology*. ACM, 2018, pp. 397–404.
- [51] L. Sharifi, L. Cerdà-Alabern, F. Freitag, and L. Veiga, "Energy efficient cloud service provisioning: Keeping data center granularity in perspective," *J. Grid Comput.*, vol. 14, no. 2, pp. 299–325, 2016. [Online]. Available: <https://doi.org/10.1007/s10723-015-9358-3>
- [52] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad, "Cloud computing pricing models: a survey," *International Journal of Grid and Distributed Computing*, vol. 6, no. 5, pp. 93–106, 2013.
- [53] L. E. Bolton, L. Warlop, and J. W. Alba, "Consumer perceptions of price (un) fairness," *Journal of Consumer Research*, vol. 29, no. 4, pp. 474–491, 2003.
- [54] B. Sharma, R. K. Thulasiram, P. Thulasiraman, S. K. Garg, and R. Buyya, "Pricing cloud compute commodities: a novel financial economic model," in *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE/ACM, 2012, pp. 451–457.

- [55] K. Song, Y. Yao, and L. Golubchik, "Exploring the profit-reliability trade-off in amazon's spot instance market: a better pricing mechanism," in *Proceedings of the IEEE/ACM International Symposium on Quality of Service (IWQoS)*. IEEE/ACM, 2013, pp. 1–10.
- [56] G. A. Paleologo, "Price-at-risk: A methodology for pricing utility computing services," *IBM Systems Journal*, vol. 43, no. 1, pp. 20–31, 2004.
- [57] B. P. Pashigian, *Price theory and applications*. McGraw-Hill, 1995.
- [58] A. Sulistio, K. H. Kim, and R. Buyya, "Using revenue management to determine pricing of reservations," in *Proceedings of the IEEE International Conference on e-Science and Grid Computing*. IEEE, 2007, pp. 396–405.
- [59] R. L. Phillips, "Pricing and revenue optimization," in *Pricing and Revenue Optimization*. Stanford university press, 2021.
- [60] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, "Faasrank: Learning to schedule functions in serverless platforms," in *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2021, pp. 31–40.
- [61] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a diet," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2020, pp. 45–59.
- [62] J. Simão, S. Esteves, A. Pires, and L. Veiga, "GC-Wise: A self-adaptive approach for memory-performance efficiency in java vms," *Future Gener. Comput. Syst.*, vol. 100, pp. 674–688, 2019. [Online]. Available: <https://doi.org/10.1016/j.future.2019.05.027>



Code of Project

Listing A.1: Publish function, base Openwhisk

```
1  override def publish(action: ExecutableWhiskActionMetadata,  
2  msg: ActivationMessage) (  
3      implicit transid: TransactionId):  
4      Future[Future[Either[ActivationId, WhiskActivation]]] = {  
5          val isBlackboxInvocation = action.exec.pull  
6  
7          val actionType = if (!isBlackboxInvocation) "managed" else "blackbox"  
8  
9          val (invokersToUse, stepSizes) =  
10             if (!isBlackboxInvocation) (  
11                 schedulingState.managedInvokers, schedulingState.managedStepSizes)  
12             else (schedulingState.blackboxInvokers, schedulingState.blackboxStepSizes)  
13
```

```

14     val chosen = if (invokersToUse.nonEmpty) {
15         val hash =
16             ShardingContainerPoolBalancer.generateHash(
17                 msg.user.namespace.name, action.fullyQualifiedName(false))
18         val homeInvoker = hash % invokersToUse.size
19
20         val stepSize = stepSizes(hash % stepSizes.size)
21
22         val invoker: Option[(InvokerInstanceId, Boolean)] =
23             ShardingContainerPoolBalancer.schedule(
24                 action.limits.concurrency.maxConcurrent,
25                 action.fullyQualifiedName(true),
26                 invokersToUse,
27                 schedulingState.invokerSlots,
28                 action.limits.memory.megabytes,
29                 homeInvoker,
30                 stepSize)
31         invoker.foreach {
32             case (_, true) =>
33                 val metric =
34                     if (isBlackboxInvocation)
35                         LoggingMarkers.BLACKBOX.SYSTEM.OVERLOAD
36                     else
37                         LoggingMarkers.MANAGED.SYSTEM.OVERLOAD
38                     MetricEmitter.emitCounterMetric(metric)
39             case _ =>
40                 }
41         invoker.map(_._1)
42     } else {
43         None
44     }
45
46     chosen
47         .map { invoker =>
48             val memoryLimit = action.limits.memory
49             val memoryLimitInfo = if (memoryLimit == MemoryLimit())
50                 {"std" } else { "non-std" }
51             val timeLimit = action.limits.timeout

```

```

52     val timeLimitInfo = if (timeLimit == TimeLimit())
53     { "std" } else { "non-std" }
54     val activationResult = setupActivation(msg, action, invoker)
55     sendActivationToInvoker(messageProducer, msg, invoker).map(
56     - =>activationResult)
57 }
58 .getOrElse {
59     val invokerStates = invokersToUse.foldLeft(
60     Map.empty[InvokerState, Int]) {
61     (agg, curr) =>
62         val count = agg.getOrElse(curr.status, 0) + 1
63         agg + (curr.status -> count)
64     }
65     Future.failed(LoadBalancerException("No invokers available"))
66 }
67 }

```

Listing A.2: Publish function, updated Openwhisk

```

1  override def publish(action: ExecutableWhiskActionMetadata,
2  msg: ActivationMessage) (
3      implicit transid: TransactionId):
4      Future[Future[Either[ActivationId, WhiskActivation]]] = {
5
6      val isBlackboxInvocation = action.exec.pull
7      val actionType = if (!isBlackboxInvocation) "managed" else "blackbox"
8      val (invokersToUse, stepSizes) =
9          if (!isBlackboxInvocation) (
10              schedulingState.managedInvokers, schedulingState.managedStepSizes)
11          else (schedulingState.blackboxInvokers, schedulingState.blackboxStepSizes)
12      var spreadInvokers: Option[List[InvokerInstanceId]] = None
13
14      logging.info(
15          this,
16          s"Action name is ${action.fullyQualified_name(true).name.toString()}")
17      )
18
19      val willSpread: Boolean =

```

```

20     action.fullyQualifiedName(true).name.toString().startsWith("SPREAD_")
21
22     val chosen = if (invokersToUse.nonEmpty) {
23         val hash =
24             ShardingContainerPoolBalancer.generateHash(
25                 msg.user.namespace.name, action.fullyQualifiedName(false))
26
27         val homeInvoker = hash % invokersToUse.size
28
29         val homeInvokerId = invokersToUse(homeInvoker)
30
31         val stepSize = stepSizes(hash % stepSizes.size)
32
33         val invokers: Option[(InvokerInstanceId, Boolean, List[InvokerInstanceId])] =
34             ShardingContainerPoolBalancer.ScheduleAndSpread(
35                 action.limits.concurrency.maxConcurrent,
36                 action.fullyQualifiedName(true),
37                 invokersToUse,
38                 schedulingState.invokerSlots,
39                 action.limits.memory.megabytes,
40                 homeInvoker,
41                 stepSize,
42                 willSpread)
43
44         invokers.foreach {
45             case (_, true, _) =>
46                 val metric =
47                     if (isBlackboxInvocation)
48                         LoggingMarkers.BLACKBOX_SYSTEM_OVERLOAD
49                     else
50                         LoggingMarkers.MANAGED_SYSTEM_OVERLOAD
51                 MetricEmitter.emitCounterMetric(metric)
52             case _ =>
53         }
54         spreadInvokers = invokers.map(_._3)
55         invokers.map(_._1)
56     } else {
57         None

```



```

58     }
59
60
61     val numInvokersSpread: Int = spreadInvokers match {
62         case Some(list) => list.size
63         case None => 0
64     }
65
66     logging.info(
67         this,
68         s"Going to scheduler to ${chosen} and ${spreadInvokers}
69         invokers for a total extra invokers of ${numInvokersSpread}"
70     )
71     spreadInvokers
72         .map { invokerList =>
73             invokerList
74                 .map { invoker =>
75                     val memoryLimit = action.limits.memory
76                     val memoryLimitInfo = if (memoryLimit == MemoryLimit())
77                         "std" else "non-std"
78                     val timeLimit = action.limits.timeout
79                     val timeLimitInfo =
80                         if (timeLimit == TimeLimit()) "std" else "non-std"
81
82                     val activationResult = setupActivation(msg, action, invoker)
83                     sendActivationToInvoker(
84                         messageProducer, msg, invoker).map(_ => activationResult)
85                 }
86         }
87
88
89     chosen
90         .map { invoker =>
91             val memoryLimit = action.limits.memory
92             val memoryLimitInfo =
93                 if (memoryLimit == MemoryLimit()) { "std" } else { "non-std" }
94             val timeLimit = action.limits.timeout
95             val timeLimitInfo =

```

```

96     if (timeLimit == TimeLimit()) { "std" } else { "non-std" }
97     logging.info(
98         this,
99         s"scheduled chosen ${chosen}, activation ${msg.activationId},
100         action '${msg.action.asString}' ($actionType),
101         ns '${msg.user.namespace.name.asString}',
102         mem limit ${memoryLimit.megabytes} MB (${memoryLimitInfo}),
103         time limit ${timeLimit.duration.toMillis} ms
104         (${timeLimitInfo}) to ${invoker}")
105     val activationResult = setupActivation(msg, action, invoker)
106     sendActivationToInvoker(
107         messageProducer, msg, invoker).map(_ => activationResult)
108 }
109 .getOrElse {
110     // report the state of all invokers
111
112     val invokerStates =
113     invokersToUse.foldLeft(Map.empty[InvokerState, Int]) { (agg, curr) =>
114         val count = agg.getOrElse(curr.status, 0) + 1
115         agg + (curr.status -> count)
116     }
117
118     logging.error(
119         this,
120         s"failed to schedule activation ${msg.activationId},
121         action '${msg.action.asString}' ($actionType),
122         ns '${msg.user.namespace.name.asString}'
123         - invokers to use: $invokerStates")
124     Future.failed(LoadBalancerException("No invokers available"))
125 }
126 }

```

Listing A.3: Schedule function, base Openwhisk

```

1 def schedule(
2     maxConcurrent: Int,
3     fqcn: FullyQualifiedEntityName,
4     invokers: IndexedSeq[InvokerHealth],

```

```

5   dispatched: IndexedSeq[NestedSemaphore[FullyQualifiedEntityName]],
6   slots: Int,
7   index: Int,
8   step: Int,
9   stepsDone: Int = 0) (
10  implicit logging: Logging, transId: TransactionId):
11  Option[(InvokerInstanceId, Boolean)] = {
12  val numInvokers = invokers.size
13
14  if (numInvokers > 0) {
15    val invoker = invokers(index)
16    if (invoker.status.isUsable &&
17        dispatched(invoker.id.toInt).tryAcquireConcurrent(
18            fq, maxConcurrent, slots)) {
19      Some(invoker.id, false)
20    } else {
21      if (stepsDone == numInvokers + 1) {
22        val healthyInvokers = invokers.filter(_.status.isUsable)
23        if (healthyInvokers.nonEmpty) {
24          val random =
25            healthyInvokers(ThreadLocalRandom.current().nextInt(
26                healthyInvokers.size)).id
27          dispatched(random.toInt).forceAcquireConcurrent(
28              fq, maxConcurrent, slots)
29          Some(random, true)
30        } else {
31          None
32        }
33      } else {
34        val newIndex = (index + step) % numInvokers
35        schedule(maxConcurrent, fq, invokers, dispatched,
36            slots, newIndex, step, stepsDone + 1)
37      }
38    }
39  } else {
40    None
41  }
42 }

```

Listing A.4: ScheduleAndSpread function, updated Openwhisk

```
1 @tailrec
2 def ScheduleAndSpread(
3     maxConcurrent: Int,
4     fqcn: FullyQualifiedEntityName,
5     invokers: IndexedSeq[InvokerHealth],
6     dispatched: IndexedSeq[NestedSemaphore[FullyQualifiedEntityName]],
7     slots: Int,
8     index: Int,
9     step: Int,
10    homeInvoker: Int,
11    stepsDone: Int = 0,
12    finalList: List[InvokerInstanceId] = Nil,
13    chosenInvoker: Int = -1)(implicit logging: Logging, transId: TransactionId):
14    Option[(InvokerInstanceId, Boolean, List[InvokerInstanceId])] = {
15        val numInvokers = invokers.size
16
17        val invoker = invokers(index)
18
19        if (stepsDone == numInvokers) {
20            if (chosenInvoker == -1){
21                val healthyInvokers = invokers.filter(_.status.isUsable)
22                if (healthyInvokers.nonEmpty) {
23                    // Choose a healthy invoker randomly
24                    val random = healthyInvokers(ThreadLocalRandom.current().nextInt(
25                        healthyInvokers.size)).id
26                    dispatched(random.toInt).forceAcquireConcurrent(
27                        fqcn, maxConcurrent, slots)
28                    Some(random, true, finalList)
29                } else {
30                    None
31                }
32            } else {
33                Some(invokers(chosenInvoker).id, false, finalList.reverse)
34            }
35
36        } else {
37            if (invoker.status.isUsable &&
```

```

38     dispatched(invoker.id.toInt).tryAcquireConcurrent(
39     fqcn, maxConcurrent, slots)) {
40         val newIndex = (index + step) % numInvokers
41         if (chosenInvoker == -1) {
42             ScheduleAndSpread(maxConcurrent, fqcn, invokers, dispatched,
43             slots, newIndex, step, homeInvoker, stepsDone + 1,
44             finalList, invoker.id.toInt)
45         } else {
46             ScheduleAndSpread(maxConcurrent, fqcn, invokers, dispatched,
47             slots, newIndex, step, homeInvoker, stepsDone + 1,
48             invoker.id :: finalList, chosenInvoker)
49         }
50     } else {
51         val newIndex = (index + step) % numInvokers
52         ScheduleAndSpread(maxConcurrent, fqcn, invokers, dispatched,
53         slots, newIndex, step, homeInvoker, stepsDone + 1,
54         finalList, chosenInvoker)
55     }
56 }

```

Listing A.5: Python code to calculate the number of additional invocations

```

1  import re
2
3  filename = '/home/henrique/tmp/openwhisk/controller/logs/controller-
4  local_logs.log' # Replace with the actual file name
5
6  pattern = r'invokers for a total extra invokers of (\d+)'
7
8  total_sum = 0
9
10 with open(filename, 'r') as file:
11     for line in file:
12         match = re.search(pattern, line)
13         if match:
14             y_value = int(match.group(1))
15             total_sum += y_value
16

```

```
17 print (f"Total sum of Y values: {total_sum}")
```

Listing A.6: Setup function for CommonLoadBalancer, updated Openwhisk

```
1  protected def setupActivation(msg: ActivationMessage,
2                                     action: ExecutableWhiskActionMetadata,
3                                     instance: InvokerInstanceId):
4                                     Future[Either[ActivationId, WhiskActivation]] = {
5
6      totalActivations.increment()
7      val isBlackboxInvocation = action.exec.pull
8      val totalActivationMemory =
9          if (isBlackboxInvocation) totalBlackBoxActivationMemory else
10             totalManagedActivationMemory
11      totalActivationMemory.add(action.limits.memory.megabytes)
12
13      activationsPerNamespace.getOrElseUpdate(msg.user.namespace.uuid,
14      new LongAdder()).increment()
15      activationsPerController.getOrElseUpdate(controllerInstance,
16      new LongAdder()).increment()
17      activationsPerInvoker
18          .getOrElseUpdate(InvokerInstanceId(instance.instance,
19          userMemory = 0.MB), new LongAdder())
20          .increment()
21      val completionAckTimeout =
22          calculateCompletionAckTimeout(action.limits.timeout.duration)
23
24      val resultPromise = if (msg.blocking) {
25          activationPromises.getOrElseUpdate(msg.activationId,
26          Promise[Either[ActivationId, WhiskActivation]]()).future
27      } else Future.successful(Left(msg.activationId))
28
29      // Adjusted section
30      if (activationSlots.contains(msg.activationId)) {
31          val existingEntry = activationSlots(msg.activationId)
32          val updatedInstance = existingEntry.invokers + 1
33          val updatedEntry = existingEntry.copy(invokers = updatedInstance)
34          activationSlots.update(msg.activationId, updatedEntry)
```

```

35     } else {
36         activationSlots.getOrElseUpdate(
37             msg.activationId, {
38                 val timeoutHandler =
39                     actorSystem.scheduler.scheduleOnce(completionAckTimeout) {
40                         processCompletion(msg.activationId, msg.transid, forced = true,
41                             isSystemError = false, instance = instance)
42                     }
43                 ActivationEntry(
44                     msg.activationId,
45                     msg.user.namespace.uuid,
46                     1,
47                     action.limits.memory.megabytes.MB,
48                     action.limits.timeout.duration,
49                     action.limits.concurrency.maxConcurrent,
50                     action.fullyQualifiedName(true),
51                     timeoutHandler,
52                     isBlackboxInvocation,
53                     msg.blocking,
54                     controllerInstance)
55             })
56     }
57     resultPromise
58 }

```

Listing A.7: ProcessCompletion function for CommonLoadBalancer, updated Openwhisk

```

1
2  /** 6. Process the completion ack and update the state */
3  protected[loadBalancer] def processCompletion(aid: ActivationId,
4                                              tid: TransactionId,
5                                              forced: Boolean,
6                                              isSystemError: Boolean,
7                                              instance: InstanceId): Unit = {
8
9      val invoker = instance match {
10         case i: InvokerInstanceId => Some(i)
11         case _                     => None

```

```

12     }
13
14     val invocationResult = if (forced) {
15         InvocationFinishedResult.Timeout
16     } else {
17         if (isSystemError) {
18             InvocationFinishedResult.SystemError
19         } else {
20             InvocationFinishedResult.Success
21         }
22     }
23
24     activationSlots.get(aid) match {
25         case Some(entry) =>
26
27             if (entry.invokers > 1) {
28                 val existingEntry = activationSlots(aid)
29                 val updatedInstance = existingEntry.invokers - 1
30                 val updatedEntry = existingEntry.copy(invokers = updatedInstance)
31                 activationSlots.update(aid, updatedEntry)
32             } else {
33                 activationSlots.remove(aid)
34             }
35             totalActivations.decrement()
36             val totalActivationMemory =
37                 if (entry.isBlackbox)
38                     totalBlackBoxActivationMemory else totalManagedActivationMemory
39             totalActivationMemory.add(entry.memoryLimit.toMB * (-1))
40             activationsPerNamespace.get(entry.namespaceId).foreach(_.decrement())
41             activationsPerController.get(entry.controllerId).foreach(_.decrement())
42
43             invoker.foreach{ inv =>
44                 activationsPerInvoker
45                     .get(InvokerInstanceId(inv.instance, userMemory = 0.MB))
46                     .foreach(_.decrement())
47             }
48             invoker.foreach(releaseInvoker(_, entry))
49

```



```

50     if (!forced) {
51         entry.timeoutHandler.cancel()
52         logging.info(this,
53             s"received completion ack for '$aid' from invoker $invoker,
54             system error=$isSystemError") (tid)
55         MetricEmitter.emitCounterMetric (LOADBALANCER_COMPLETION_ACK_REGULAR)
56     } else {
57         activationPromises
58             .remove(aid)
59             .foreach(_.tryFailure(new Throwable(
60                 "no completion or active ack received yet")))
61         val actionType = if (entry.isBlackbox) "blackbox" else "managed"
62         val blockingType = if (entry.isBlocking) "blocking" else "non-blocking"
63         val completionAckTimeout =
64             calculateCompletionAckTimeout(entry.timeLimit)
65         logging.warn(
66             this,
67             s"forced completion ack for '$aid',
68             action '${entry.fullyQualifiedEntityName}' ($actionType),
69             $blockingType, mem limit ${entry.memoryLimit.toMB} MB,
70             time limit ${entry.timeLimit.toMillis} ms,
71             completion ack timeout $completionAckTimeout from $instance") (
72             tid)
73         MetricEmitter.emitCounterMetric (LOADBALANCER_COMPLETION_ACK_FORCED)
74     }
75     invoker.foreach(
76         invokerPool ! InvocationFinishedMessage(_, invocationResult))
77 case None if tid == TransactionId.invokerHealth =>
78     activationSlots.remove(aid)
79     logging.info(this,
80         s"received completion ack for health action on $instance") (tid)
81     MetricEmitter.emitCounterMetric (LOADBALANCER_COMPLETION_ACK_HEALTHCHECK)
82     invoker.foreach(
83         invokerPool ! InvocationFinishedMessage(_, invocationResult))
84 case None if !forced =>
85     activationSlots.remove(aid)
86     logging.warn(
87         this,

```

```

88         s"received completion ack for '$aid' from $instance which has no entry,
89         system error=$isSystemError") (tid)
90
91         MetricEmitter.emitCounterMetric(
92             LOADBALANCER_COMPLETION_ACK_REGULAR_AFTER_FORCED)
93     case None =>
94         activationSlots.remove(aid)
95         logging.debug(this,
96             s"forced completion ack for '$aid' which has no entry") (tid)
97         MetricEmitter.emitCounterMetric(
98             LOADBALANCER_COMPLETION_ACK_FORCED_AFTER_REGULAR)
99     }
100 }

```