Project - FaaS-Utility (revised)

Henrique da Conceição Reis dos Santos
Number: 92476
Email: henrique.c.r.santos@tecnico.ulisboa.pt

Instituto Superior Técnico

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. Causing great deals 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 a 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 utilizes 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, Function-as-a-Service

1 Introduction

Edge computing [32], 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 [29].

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 know as FaaS) [35].

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. 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.

Current scheduling mechanisms [54,19] 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 [50]. We propose an optimization to the scheduling mechanism 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 state of the server, when attempting to put in a request such as monetary discounts for slower execution times or extra monetary costs for his request to be completed in a timely manner. The latter is presented in case the system is saturated and unable to confidently complete customer requests in the initially expect 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.

1.2 Objectives

In order 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 asses 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 4 and Section 5 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

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 [29]. In essence, it relates to who will oversee these Services' many layers, these can be classified as IaaS (Infrastructure-as-a-Service), PaaS (Platform-as-a-Service), SaaS (Software-as-a-Service), BaaS (Backend-as-a-Service) and finally FaaS (Function-as-a-Service) 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 has to have the lowest transparency [29].

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 [29].

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 [3].

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) [39,17]. 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 [3].

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 [3].

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 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 [29].

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 1.

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 [29].

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 [29].

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.

2.1.3.1 **Infrastructural** Low latency is frequently needed for use cases like monitoring people's vital signs during emergencies or in daily life [31]. 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

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 1. Cloud services and their levels of user control.

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 [32].

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 [28,21], but these studies do not take into account the limitations imposed by an edge-based environment [35].

FaaS is a scalable and flexible event-based programming model so it's a great fit for IoT events and data processing [35]. 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 [35].

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 [8]. These low-power edge nodes differ from cloud apps, which must scale across multiple cloud computers, in part because they are monolithic [35].

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 [22,30]. 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 [35].

2.1.3.2 **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 [29]. Due to its frequent use and ease of data analysis, this case study is frequently used for performance evaluations. For example, in [38] by managing an image resizing case study and in [40] 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 [29].

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 [59] 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 [55]. 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 [41] and AutoGluon [4]. 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 [11].

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 [12] demonstrated the feasibility of using the FaaS model for scientific and high-performance computing by presenting various prototypes and their respective measurements. In [12], 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 [26].

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 [29]. Table 2 shows the use cases and their various studies.

2.1.4 FaaS challenges

2.1.4.1 **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 performance issues [57].

Use cases	Studies
Edge computing	[28,21,8]
Image and Video Manipulation	[38,40]
Multi-language Applications	[29]
Auto-scaling highly available Websites and APIs	[29]
Machine Learning	[59,55,11]
Scientific Computing	[12,26]
Event streaming	[26,29]

Table 2. Use Cases and theirs various studies.

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 [56].

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 [56], 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 [7], 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 [48]. 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 [7].

Likewise, Apache OpenWhisk [6] 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 [7].

2.1.4.2 **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 [62] and SmartHarvest [58] optimize resource efficiency by dynamically harvesting free resources from over-provisioned

operations and shifting them to under-provisioned services. Spock [16] suggests a cost and SLO-improving FaaS-based VM scaling architecture. [19] and [54] 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 [40].

2.1.4.3 **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 [11].

To address some of these issues, there are numerous commercial security solutions available [2,51]. Aqua [2] 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 [51] 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 a number of remedies [18,42]. SecLambda is an extensible security framework that [18] 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 [42]. 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 3 summarizes the FaaS challenges and their various studies.

FaaS challenges	Studies
Cold start delay	[56,5,20,45,7,48]
Resource allocation	[62,58,16,19,54]
Security concerns	[18,42,11]

Table 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 [40,24,25], but this rarely or never considers the potential revenue that these optimizations can provide [14]. 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 [23], and cloud-edge (Fog) systems [37,44], numerous solutions have been put forth. Load balancing [24], maximizing resource use [60] and energy efficiency [27], minimizing execution costs [13], and maximizing performance are the typical objectives of scheduling [9]. 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 [25]. 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 [40], 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 [14]. 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 [46]. 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 [14].

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 [1]. 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 [14].

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 [14].

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 [10].

2.2.2.1 **Compounded Moore's Law and beginning expenditures** In [47] 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 [14].

2.2.2.2 **Spot Instance** The Spot Instance approach, which was extensively discussed in [52], 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.

2.2.2.3 **Price-at-risk** The major goal of the work [33] 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 in order 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 [14].

2.2.2.4 **Customer classification and resource consumption status** In economics, the term "price discrimination" is used to refer to varying prices [34]. To design advanced reservation pricing in Computer Grids, [53] relies on two key pillars: revenue optimization [36] and price discrimination. In order 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 4 summarizes the characteristics of the various pricing mechanism's highlighting their advantages and drawbacks.

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

Table 4. Characteristics of the various pricing mechanism's.

In order 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 of 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 true 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 [29], 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 [29].

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 [29].

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 [29]. 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 [28].

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 [28].

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 is able to add CRDs to the Kubernetes platform in order to support higher levels of abstraction.

Building, Serving, and Eventing are this platform's three key pillars. The Build component is a pluggable 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 API concepts, CLIs, tooling, etc. is left to the discretion of particular vendors since Knative is not a full-featured FaaS platform [28].

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 [28].

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 Kubernets 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 in order to benefit from the robust Kubernetes infrastructure.

3 Architecture

In this section, it will be presented firstly 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

To create new functions, invoke existing ones, and query the outcomes of invocations, Apache OpenWhisk exposes a 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 function 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 the results. The Controller then synchronously or asynchronously returns to users the outcomes of the function executions [61].

3.2 Scheduler extension

Our new scheduler, shown in Figure 1 as the blue container, communicates with Apache OpenWhisk's Controller feeding relevant information such as the state of the server and the invocation's specific requests (mainly the pricing method desired by the user). Given this information, it will calculate a utility function to decide the best server/container to be used and deliver this new information back to the controller allowing it to proceed with the operation. This scheduler is transparent and optional, offering easy compatibility with old systems that utilize Apache OpenWhisk and allowing them to effortlessly deploy this new scheduler at their own pace. Since the scheduling will be mainly performed by our scheduler and it will overwrite the original scheduling function the base Apache OpenWhisk deploys.

The Completion state given by the CouchDB to the Controller will be also given to the scheduler to allow it to update the pricing models offered based on server state feedback. After this information is processed it will be given back to Apache OpenWhisk's Controller. After the operation has been completed the customer will receive the output of the action as well as an update of the server state as well as the pricing model if it was updated (see Section 3.3).

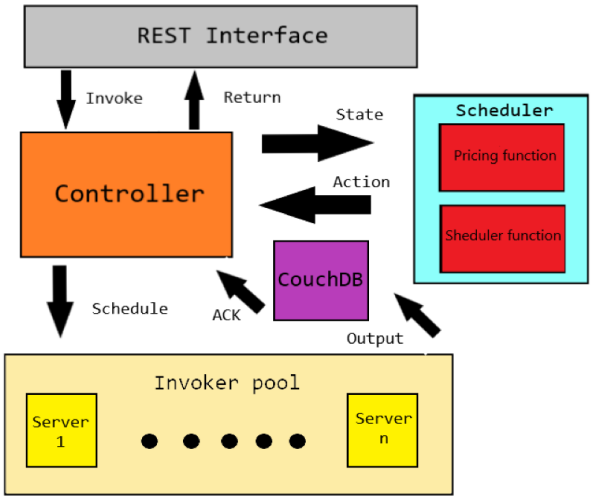

Fig. 1. General architecture with newly added scheduler component

Two pricing options will be provided if the servers are in an over-provisioned state meaning there are no queuing issues with the requests. (1) Merely finishing the request, or (2) finishing the request slower but discounting part of the final cost by the number of extra resources consumed. The second option is to use the request to create warm containers for this particular user's repeated uses, resulting in future execution times that are quicker. The user will receive all of this information for transparency's sake and encourage continued use.

Three different pricing options 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 customers that has self-perceived time-critical actions to be performed. An example of a such customer is someone that detected a mistake in a database and wishes it to be fixed as soon as possible so to furthers uses of the database not be compromised. (3) Reduced priority which offers, for a reduced price tag, a lower priority in the system for customers 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

The Apache Openwhisk controllers are responsible for the organization of requests received by the users. 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 pools that are available 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, it uses a hash function to

identify the action's home Invoker, which is responsible for deploying that specific action on subsequent calls unless the Invoker is full, in which case the action is deployed to another Invoker.

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 stored 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 contains containers that only need code initialization, making them quicker than newly created containers but slower than the particular container of an action.

Upon receiving a new action request the scheduler will first attempt to schedule it in his home Invoker, however, if its Busy pool is saturated it will then attempt to schedule the action over all other Invokers. If all Invokers are saturated the scheduler will queue the action using a FIFO (first-in-first-out) priority method and wait until enough resources are freed. Given that the Busy pool of a specific Invoker is not saturated it will first try to deploy the action using a container for the specific action in the free pool, if no container exists it will then try to utilize a pre-warm container from the pre-warm pool, and 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. The algorithm 1 exemplifies in pseudo-code the steps described above.

3.4 Proposed Scheduling modification

When the available resources by the system is above a threshold the pricing model of our scheduler will inform the controller to update the information given to the customers to swap the pricing model shown to the over-provisioned state and vice-versa. This transition has a grace period so as to not update the model too many times while the resources available are close to the threshold.

3.4.1 Proposed scheduling modification during an over-provisioned state

The scheduling system will operate as usual if no pricing mechanism is used, or, in other words, if the request demands a standard fee. However, the scheduling system will send the action to all Invokers provided with an additional condition to all but the home Invoker if the customer requested the additional pricing mechanism. This additional requirement prevents existing containers in the free pool from being deleted, preventing other requests from needing more time to complete in their respective home Invokers.

In the event that the action is repeatedly requested, saturating the home Invoker, enables much faster execution following the initial deployment. Customers are further encouraged to use our system repeatedly because doing so will result in faster execution times. This updated algorithm 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). This however is a challenge that should rarely if ever arise during an over-provisioned state where this algorithm is designed for.

Algorithm 1 Simplified 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 $FreePoolSize + BusyPoolSize = MaxPoolSize$ **then**
      $FreePool \leftarrow FreePool \setminus LeastRecentContainer$
    end if
  end if
  if $PreWarmPoolSize > 0$ **then**
    $PreWarmPool \leftarrow PreWarmPool \setminus PreeWarmContainer$
    $ActionContainer \leftarrow PreWarmContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  else
    $ActionContainer \leftarrow ColdContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  end if
  return
end for
$Queue \leftarrow Queue \cup Action$

Algorithm 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 $FreePoolSize + BusyPoolSize = MaxPoolSize$ **then**
      if $Invoker \neq HomeInvoker$ **then**
        continue
      else
        $FreePool \leftarrow FreePool \setminus LeastRecentContainer$
      end if
    end if
  end if
  if $PreWarmPoolSize > 0$ **then**
    $PreWarmPool \leftarrow PreWarmPool \setminus PreeWarmContainer$
    $ActionContainer \leftarrow PreWarmContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  else
    $ActionContainer \leftarrow ColdContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  end if
end for
$Queue \leftarrow Queue \cup Action$

All of the results of the multiple executions of the action are received by the controller. The cost of the requested action is calculated as usual but the final cost charged to the customer is a discount of the action cost proportional to the extra resources consumed by the scheduling extension. Consequently, the cost that the user will be charged is given by the equation 1.

$$final\,cost = \alpha\, c + (1 - \alpha)\, \frac{c^2}{C}, \tag{1}$$

where α is the percentage of cost that remains static, c is the cost of the specific action, and C is the total cost of all actions deployed.

This creates a situation where if no additional actions were deployed on other Invokers the final costs are equal to the normal pricing model but as more actions are deployed creating additional delay the final cost is reduced.

3.4.2 Proposed scheduling modification during an under-provisioned state

As stated in the final part of section 3.3, a FIFO priority method is used in case action start being queued due to the server being saturated, this in turn creates a very low urgency methodology for the customers. This work proposes a more advanced priority aware system that allows more time-critical situations that customers might have to be more easily resolved but for a cost as well as the inverse situation if the need arises.

The algorithm is based on a priority value coined 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 2 exemplifies four seconds of this algorithm in progress where $p_1 = 1$, $p_2 = 2$ and $p_3 = 5$.

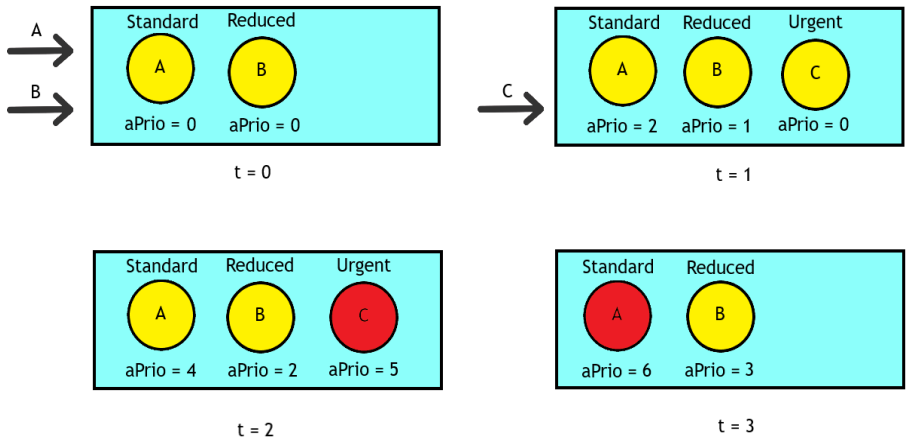

Fig. 2. Four seconds of execution of the priority queue algorithm

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.

The pricing model utilized is similar to what is offered during the over-provisioned state. The final cost is given by the equation 2

$$final\,cost = \alpha\, c + (1 - \alpha)\, \frac{c\, p}{p_1}, \tag{2}$$

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 Evaluation Methodology

In this section, 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 will be extended to the requirements presented by the architecture in Section 3. Data will be assumed to be stored locally or on some cloud storage in the same location.

4.1 FaaS Benchmarks

Four diverse FaaS workloads will be used in the evaluation of our system those being Sleep functions, File hashing, Video Transformation, and Image classification [15]:

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 [40], especially in machine learning applications [59,55], so using it as a benchmark can help to evaluate the system's ability to handle real-world workloads.

4.2 Metrics

Latency, Scheduling delay and Resource Usage will be 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 are able to 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 actually given the opportunity to run by the scheduler. It is an important metric for evaluating the performance of a system because it measures how well the scheduler is able to distribute resources and manage the execution of tasks. A low scheduling delay indicates that the scheduler is able to 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 [49].

These metrics will be measured and compared with the Apache OpenWhisk base scheduler.

5 Conclusion

Our work described the current state of cloud computing's Function-as-a-Service technology, along with some of its key benefits and difficulties. In order 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 takes user preferences into account when adjusting scheduling, to provide a higher quality of service to the user. Better quality of service for the user is part of our suggested architecture, and it applies to both under-provisioned and over-provisioned system states. Apache Openwhisk will be used to implement our suggested solution. Finally, we also proposed a methodology to assess how well the system we implemented performs.

References

1. Al-Roomi, M., Al-Ebrahim, S., Buqrais, S., Ahmad, I.: Cloud computing pricing models: a survey. International Journal of Grid and Distributed Computing 6(5), 93–106 (2013)
2. Aqua: `https://www.aquasec.com/aqua-cloud-native-security-platform/`, accessed 5-december-2022
3. Astrova, I., Koschel, A., Schaaf, M., Klassen, S., Jdiya, K.: Serverless, faas and why organizations need them. Intelligent Decision Technologies 15(4), 825–838 (2021)
4. AutoGluon: `https://auto.gluon.ai/stable/index.html`, accessed 5-december-2022
5. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless computing: current trends and open problems. In: Research Advances in Cloud Computing, pp. 1–20. Springer (2017)
6. Baldini, I., Cheng, P., Fink, S.J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P., Tardieu, O.: 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. pp. 89–103 (2017)

7. Bermbach, D., Karakaya, A.S., Buchholz, S.: Using application knowledge to reduce cold starts in faas services. In: Proceedings of the ACM Symposium on Applied Computing. pp. 134–143 (2020)
8. Bermbach, D., Pallas, F., Pérez, D.G., Plebani, P., Anderson, M., Kat, R., Tai, S.: A research perspective on fog computing. In: Proceedings of the International Conference on Service-Oriented Computing. pp. 198–210. Springer (2018)
9. Binh, H.T.T., Anh, T.T., Son, D.B., Duc, P.A., Nguyen, B.M.: An evolutionary algorithm for solving task scheduling problem in cloud-fog computing environment. In: Proceedings of the International Symposium on Information and Communication Technology. pp. 397–404 (2018)
10. Bolton, L.E., Warlop, L., Alba, J.W.: Consumer perceptions of price (un) fairness. Journal of Consumer Research 29(4), 474–491 (2003)
11. Bouizem, Y.: Fault tolerance in FaaS environments. Ph.D. thesis, Université Rennes 1 (2022)
12. Chard, R., Skluzacek, T.J., Li, Z., Babuji, Y., Woodard, A., Blaiszik, B., Tuecke, S., Foster, I., Chard, K.: Serverless supercomputing: high performance function as a service for science. ArXiv abs/1908.04907 (2019)
13. Choudhari, T., Moh, M., Moh, T.S.: Prioritized task scheduling in fog computing. In: Proceedings of the ACM Southeast Conference (ACMSE). pp. 1–8 (2018)
14. Dibaj, S.R., Sharifi, L., Miri, A., Zhou, J., Aram, A.: Cloud computing energy efficiency and fair pricing mechanisms for smart cities. In: IEEE Electrical Power and Energy Conference (EPEC). pp. 1–6 (2018)
15. Dukic, V., Bruno, R., Singla, A., Alonso, G.: Photons: Lambdas on a diet. In: Proceedings of the 11th ACM Symposium on Cloud Computing. pp. 45–59 (2020)
16. Gunasekaran, J.R., Thinakaran, P., Kandemir, M.T., Urgaonkar, B., Kesidis, G., Das, C.: 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). pp. 199–208 (2019)
17. Janakiraman, B.: Serverless (2016), `https://martinfowler.com/bliki/Serverless.html`, accessed 15-november-2022
18. Jegan, D.S., Wang, L., Bhagat, S., Ristenpart, T., Swift, M.: Guarding serverless applications with seclambda. arXiv preprint arXiv:2011.05322 (2020)
19. Kim, Y.K., HoseinyFarahabady, M.R., Lee, Y.C., Zomaya, A.Y.: Automated fine-grained cpu cap control in serverless computing platform. IEEE Transactions on Parallel and Distributed Systems 31(10), 2289–2301 (2020)
20. Kratzke, N.: A brief history of cloud application architectures. Applied Sciences 8(8), 1368 (2018)
21. Kritikos, K., Skrzypek, P.: A review of serverless frameworks. In: Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). pp. 161–168. IEEE (2018)
22. Laaroussi, Z., Morabito, R., Taleb, T.: 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). pp. 1–6 (2018)
23. Lee, G., Chun, B., Katz, H.: Heterogeneity-aware resource allocation and scheduling in the cloud. In: Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud) (2011)
24. Lin, L., Li, P., Xiong, J., Lin, M.: Distributed and application-aware task scheduling in edge-clouds. In: Proceedings of the International Conference on Mobile Ad-Hoc and Sensor Networks (MSN). pp. 165–170. IEEE (2018)
25. Madej, A., Wang, N., Athanasopoulos, N., Ranjan, R., Varghese, B.: Priority-based fair scheduling in edge computing. In: Proceedings of the IEEE International Conference on Fog and Edge Computing (ICFEC). pp. 39–48 (2020)

26. Malawski, M., Gajek, A., Zima, A., Balis, B., Figiela, K.: Serverless execution of scientific workflows: experiments with hyperflow, aws lambda and google cloud functions. Future Generation Computer Systems 110, 502–514 (2020)
27. Mendes, S., Simão, J., Veiga, L.: Oversubscribing micro-clouds with energy-aware containers scheduling. In: Proceedings of the ACM/SIGAPP Symposium on Applied Computing. pp. 130–137 (2019)
28. Mohanty, S.K., Premsankar, G., di Francesco, M.: An evaluation of open source serverless computing frameworks. In: Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 115–120 (2018)
29. Mukundand, R., Bharati, R.: Function as a service in cloud computing: A survey. International Journal of Future Generation Communication and Networking 13(3), 3291–3297 (2020)
30. Naik, N.: Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: Proceedings of the IEEE International Systems Engineering Symposium (ISSE). pp. 1–7. IEEE (2017)
31. Nastic, S., Rausch, T., Scekic, O., Dustdar, S., Gusev, M., Koteska, B., Kostoska, M., Jakimovski, B., Ristov, S., Prodan, R.: A serverless real-time data analytics platform for edge computing. IEEE Internet Computing 21(4), 64–71 (2017)
32. Palade, A., Kazmi, A., Clarke, S.: An evaluation of open source serverless computing frameworks support at the edge. In: Proceedings of the IEEE World Congress on Services (SERVICES). vol. 2642, pp. 206–211 (2019)
33. Paleologo, G.A.: Price-at-risk: A methodology for pricing utility computing services. IBM Systems Journal 43(1), 20–31 (2004)
34. Pashigian, B.P.: Price theory and applications. McGraw-Hill College (1995)
35. Pfandzelter, T., Bermbach, D.: tinyfaas: A lightweight faas platform for edge environments. In: Proceedings of the IEEE International Conference on Fog Computing (ICFC). pp. 17–24 (2020)
36. Phillips, R.L.: Pricing and revenue optimization. In: Pricing and Revenue Optimization. Stanford university press (2021)
37. Pires, A., Simão, J., Veiga, L.: Distributed and decentralized orchestration of containers on edge clouds. J. Grid Comput. 19(3), 36 (2021)
38. Quevedo, S., Merchán, F., Rivadeneira, R., Dominguez, F.: Evaluating apache openwhisk - faas. In: IEEE Ecuador Technical Chapters Meeting (ETCM). pp. 1–5 (2019)
39. Roberts, M.: Serverless architectures (Nov 2018), `Availablefrom:https://martinfowler.com/articles/serverless.html`
40. Russo, G.R., Milani, A., Iannucci, S., Cardellini, V.: 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). pp. 693–698 (2022)
41. SageMaker, A.: `https://aws.amazon.com/pm/sagemaker/`, accessed 5-december-2022
42. Sankaran, A., Datta, P., Bates, A.: Workflow integration alleviates identity and access management in serverless computing. In: Proceedings of the Annual Computer Security Applications Conference. pp. 496–509 (2020)
43. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the ACM European Conference on Computer Systems. pp. 351–364 (2013)
44. Scoca, V., Aral, A., Brandic, I., De Nicola, R., Uriarte, R.B.: Scheduling latency-sensitive applications in edge computing. In: Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER). pp. 158–168 (2018)
45. Sewak, M., Singh, S.: Winning in the era of serverless computing and function as a service. In: Proceedings of the IEEE International Conference for Convergence in Technology (I2CT). pp. 1–5 (2018)

46. Sharifi, L., Cerdà-Alabern, L., Freitag, F., Veiga, L.: Energy efficient cloud service provisioning: Keeping data center granularity in perspective. J. Grid Comput. 14(2), 299–325 (2016), https://doi.org/10.1007/s10723-015-9358-3
47. Sharma, B., Thulasiram, R.K., Thulasiraman, P., Garg, S.K., Buyya, R.: Pricing cloud compute commodities: a novel financial economic model. In: Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 451–457 (2012)
48. Shillaker, S.: A provider-friendly serverless framework for latency-critical applications. In: Proceedings of the Eurosys Doctoral Workshop. p. 71 (2018)
49. Simão, J., Esteves, S., Pires, A., Veiga, L.: *GC-Wise*: A self-adaptive approach for memory-performance efficiency in java vms. Future Gener. Comput. Syst. 100, 674–688 (2019), https://doi.org/10.1016/j.future.2019.05.027
50. Simão, J., Veiga, L.: Partial utility-driven scheduling for flexible SLA and pricing arbitration in clouds. IEEE Trans. Cloud Comput. 4(4), 467–480 (2016)
51. Snyk: https://snyk.io/, accessed 5- december-2022
52. Song, K., Yao, Y., Golubchik, L.: 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). pp. 1–10 (2013)
53. Sulistio, A., Kim, K.H., Buyya, R.: Using revenue management to determine pricing of reservations. In: Proceedings of the IEEE International Conference on e-Science and Grid Computing. pp. 396–405 (2007)
54. Suresh, A., Somashekar, G., Varadarajan, A., Kakarla, V.R., Upadhyay, H., Gandhi, A.: 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). pp. 1–10 (2020)
55. Tu, Z., Li, M., Lin, J.: 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. pp. 6–10 (2018)
56. Vahidinia, P., Farahani, B., Aliee, F.S.: Mitigating cold start problem in serverless computing: a reinforcement learning approach. IEEE Internet of Things Journal (2022)
57. Van Eyk, E., Iosup, A., Abad, C.L., Grohmann, J., Eismann, S.: 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. pp. 21–24 (2018)
58. Wang, Y., Arya, K., Kogias, M., Vanga, M., Bhandari, A., Yadwadkar, N.J., Sen, S., Elnikety, S., Kozyrakis, C., Bianchini, R.: Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In: Proceedings of the European Conference on Computer Systems. pp. 1–16 (2021)
59. Xu, F., Qin, Y., Chen, L., Zhou, Z., Liu, F.: λdnn: Achieving predictable distributed dnn training with serverless architectures. IEEE Transactions on Computers 71(2), 450–463 (2021)
60. Yin, L., Luo, J., Luo, H.: Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing. IEEE Transactions on Industrial Informatics 14(10), 4712–4721 (2018)
61. Yu, H., Irissappane, A.A., Wang, H., Lloyd, W.J.: Faasrank: Learning to schedule functions in serverless platforms. In: Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). pp. 31–40 (2021)
62. Yu, H., Wang, H., Li, J., Yuan, X., Park, S.J.: Accelerating serverless computing by harvesting idle resources. In: Proceedings of the ACM Web Conference. pp. 1741–1751 (2022)

A Schedule

Tasks		**Duration**
Prototype development	Environment setup	0.5 month
	Code implementation	1.5 months
	Workload setup	1 month
Data evaluation		1 month
Dissertation writing		1 month