FaaS-Utility

HENRIQUE SANTOS, Instituto Superior Técnico, Lisbon, Portugal

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 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 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.

CCS Concepts: • **Networks → Cloud computing**.

Additional Key Words and Phrases: Resource Scheduling, Function-as-a-Service, Pricing, Utility

1 INTRODUCTION

Edge computing [Palade et al. 2019], 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 [Mukundand and Bharati 2020].

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) [Pfandzelter and Bermbach 2020].

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 [Kim et al. 2020; Suresh et al. 2020] 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 [Simão and Veiga 2016]. 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.

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.

2 RELATED WORK

This section will discuss the most cutting-edge techniques and technologies currently being used in this field.

2.1 Function as a Service

In terms of architectural layers of Cloud Computing, the Cloud is typically considered as numerous Cloud Services [Mukundand and Bharati 2020]. 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.

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 [Mukundand and Bharati 2020].

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 [Mukundand and Bharati 2020].

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 [Astrova et al. 2021].

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) [Janakiraman 2016; Roberts 2018]. 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 [Astrova et al. 2021].

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 [Astrova et al. 2021].

Low latency is frequently needed for use cases like monitoring people’s vital signs during emergencies or in daily life [Nastic et al. 2017]. 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 [Palade et al. 2019].

FaaS is a scalable and flexible event-based programming model so it’s a great fit for IoT events and data processing [Pfandzelter and Bermbach 2020]. 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 [Pfandzelter and Bermbach 2020].

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 [Van Eyk et al. 2018].

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 [Vahidinia et al. 2023].

Although 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 [Vahidinia et al. 2023], 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 to consider resource usage. While in the work [Bermbach et al. 2020], 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.

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 [Lin et al. 2018; Madej et al. 2020; Russo et al. 2022], but this rarely or never considers the potential revenue that these optimizations can provide [Dibaj et al. 2018]. 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 [Schwarzkopf et al. 2013], clouds [Lee et al. 2011], and cloud-edge (Fog) systems [Pires et al. 2021; Scoca et al. 2018], numerous solutions have been put forth. Load balancing [Lin et al. 2018], maximizing resource use [Yin et al. 2018] and energy efficiency [Mendes et al. 2019], minimizing execution costs [Choudhari et al. 2018], and maximizing performance are the typical objectives of scheduling [Binh et al. 2018]. 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 [Madej et al. 2020]. 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 [Russo et al. 2022], 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 [Dibaj et al. 2018]. 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 [Sharifi et al. 2016]. 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 [Dibaj et al. 2018].

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 [Al-Roomi et al. 2013]. 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 [Dibaj et al. 2018].

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 [Dibaj et al. 2018].

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 [Bolton et al. 2003].

3 ARCHITECTURE

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

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 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 [Yu et al. 2021].

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 1 as the green and blue containers.

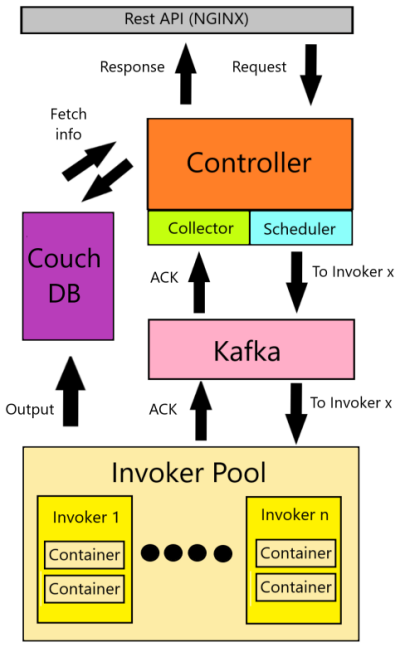

Fig. 1. General architecture with newly added scheduler and Collector component

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.

3.3 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 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.

3.4 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 1 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**
    $PreWarmPool \leftarrow PreWarmPool \setminus PreeWarmContainer$
    $ActionContainer \leftarrow PreWarmContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  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 1 shows the pseudo-code that the new scheduler utilizes. It 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.

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

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

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.

3.5 Scheduling during an under-provisioned state

A First-In-First-Out (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 2 exemplifies four seconds of this algorithm in progress where $p_1 = 1$, $p_2 = 2$ and $p_3 = 5$, and 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.

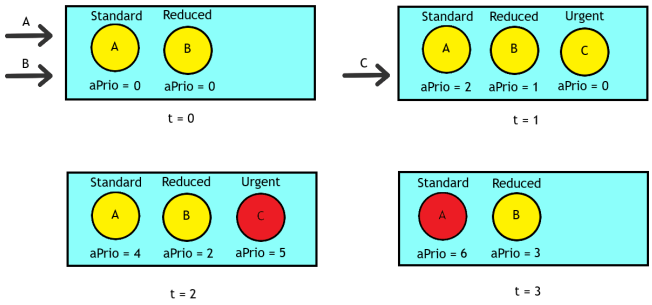

Fig. 2. Four seconds of execution of the priority queue algorithm

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

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

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 IMPLEMENTATION

This Chapter will go over the development steps we took to achieve our realization of the architecture. 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.

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 a Kubernetes deployment as that was never the issue. So we created one controller and three distinct Invokers as well as all other components each in their own containers. 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.

4.2 Development of Action-Spreading

Firstly we must fully comprehend what the attempted solution presented in the Architecture Chapter 3 entails. The goal is to (1) set up containers for future workloads as well as if possible (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.

[1] During 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

The initial approach would be to alter the algorithm in `ShardingContainerPoolBalancer.scala` 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. 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 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*.

4.3 Development of Action priority

When the system is under-provisioned, we would like to implement action priority as described in our architecture. 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.

Kafka 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 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 ApacheOpenwhisk 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.

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.

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.

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 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.

5 EVALUATION

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 [Dukic et al. 2020]:

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 [Russo et al. 2022], especially in machine learning applications [Tu et al. 2018; Xu et al. 2021], 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.

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.

6 EVALUATION ENVIRONMENT

The environment used for evaluation of the newly enhanced Apache Openwhisk scheduling is similar to the one presented during the Implementation Section 4. The cluster size was kept low, easily overloading the system if need be for testing. 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.

We used JMeter 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.

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 or 10000 ms 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. 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.

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.

Warm Sub-enviroment "W": a fully cold enviroment 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.

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).

6.1 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-enviorment 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.

Test 1 (W-A), focused on determining if our scheduler performed similarly without the use of the new Action-Spread functionality. Results showed that our enhanced scheduler performed slightly better under the same circumstances so we could safely use it for previous versions of workloads on Openwhisk.

Test 2 (C-A) and Test 6 (C-B) both seek to evaluate the use case for our functionality. This situation was a cold environment at first followed by the use of our new functionality to set up the warm container finalising with a heavy amount of requests for the specific action. We were able to confirm that our functionality was able to benefit the system in terms of reduced latency, variance, and total execution time for faster execution actions where the cold start delay is more noticeable. We also were able to conclude that hardware can indeed affect the value provided by our functionality as we were only able to see improvements from our F4 function in hardware B.

Test 3 (W-A), 4 (W-A), and Test 5 (W-B) were tested to check the performance of the scheduler during less ideal circumstances, those being in the case where a warm environment already exists for the requested action. We were able to conclude the lack of parallelism potential from the hardware itself was the main bottleneck as the new scheduler would overload the Invokers leading to performance degradation. Test 4 (W-A) specifically focused on confirming this parallelism roadblock by independently doing the same amount of requests as the Action-Spread functionality would do but using the original version of the scheduler. The functionality should not be used for already warm environments as it might lead to unnecessary overloading of the system.

6.2 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. 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 1 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 1. 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 measuse 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.

In Figure 3, 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.

7 CONCLUSION

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

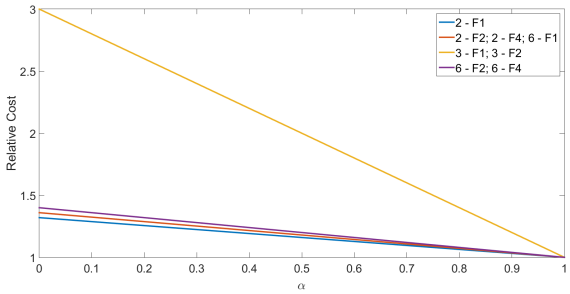

Fig. 3. Cost's behaviour depending on α values

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.

REFERENCES

May Al-Roomi, Shaikha Al-Ebrahim, Sabika Buqrais, and Imtiaz Ahmad. 2013. Cloud computing pricing models: a survey. *International Journal of Grid and Distributed Computing* 6, 5 (2013), 93–106.

Irina Astrova, Arne Koschel, Marc Schaaf, Samuel Klassen, and Kerim Jdiya. 2021. Serverless, FaaS and why organizations need them. *Intelligent Decision Technologies* 15, 4 (2021), 825–838.

David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. Using application knowledge to reduce cold starts in FaaS services. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, New York, NY, United States, 134–143.

Huynh Thi Thanh Binh, Tran The Anh, Do Bao Son, Pham Anh Duc, and Binh Minh Nguyen. 2018. 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, New York, NY, United States, 397–404.

Lisa E Bolton, Luk Warlop, and Joseph W Alba. 2003. Consumer perceptions of price (un) fairness. *Journal of Consumer Research* 29, 4 (2003), 474–491.

Tejaswini Choudhari, Melody Moh, and Teng-Sheng Moh. 2018. Prioritized task scheduling in fog computing. In *Proceedings of the ACM Southeast Conference (ACMSE)*. ACM, New York, NY, United States, 1–8.

S.M. Reza Dibaj, Leila Sharifi, Ali Miri, Jing Zhou, and Azadeh Aram. 2018. Cloud Computing Energy Efficiency and Fair Pricing Mechanisms for Smart Cities. In *IEEE Electrical Power and Energy Conference (EPEC)*. IEEE, New York, NY, United States, 1–6. https://doi.org/10.1109/EPEC.2018.8598406

Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, New York, NY, United States, 45–59.

B. Janakiraman. 2016. Serverless. https://martinfowler.com/bliki/Serverless.html Accessed 15-november-2022.

Young Ki Kim, M Reza HoseinyFarahabady, Young Choon Lee, and Albert Y Zomaya. 2020. Automated fine-grained cpu cap control in serverless computing platform. *IEEE Transactions on Parallel and Distributed Systems* 31, 10 (2020), 2289–2301.

G. Lee, B. Chun, and H. Katz. 2011. Heterogeneity-Aware Resource Allocation and Scheduling in the Cloud. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX Association, Portland, OR, 1–5.

Li Lin, Peng Li, Jinbo Xiong, and Mingwei Lin. 2018. Distributed and application-aware task scheduling in edge-clouds. In *Proceedings of the International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*. IEEE, New York, NY, United States, 165–170.

Arkadiusz Madej, Nan Wang, Nikolaos Athanasopoulos, Rajiv Ranjan, and Blesson Varghese. 2020. Priority-based Fair Scheduling in Edge Computing. In *Proceedings of the IEEE International Conference on Fog and Edge Computing (ICFEC)*. IEEE, New York, NY, United States, 39–48. https://doi.org/10.1109/ICFEC50348.2020.00012

Seŕgio Mendes, José Simão, and Luís Veiga. 2019. Oversubscribing micro-clouds with energy-aware containers scheduling. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*. ACM, New York, NY, United States, 130–137.

Rajeshwar Mukundand and Rajesh Bharati. 2020. Function as a Service in Cloud Computing: A survey. *International Journal of Future Generation Communication and Networking* 13, 3 (2020), 3291–3297.

Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. 2017. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing* 21, 4 (2017), 64–71.

Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. 2019. An evaluation of open source serverless computing frameworks support at the edge. In *Proceedings of the IEEE World Congress on Services (SERVICES)*, Vol. 2642. IEEE, New York, NY, United States, 206–211.

Tobias Pfandzelter and David Bermbach. 2020. tinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *Proceedings of the IEEE International Conference on Fog Computing (ICFC)*. IEEE, New York, NY, United States, 17–24. https://doi.org/10.1109/ICFC49376.2020.00011

André Pires, José Simão, and Luís Veiga. 2021. Distributed and Decentralized Orchestration of Containers on Edge Clouds. *J. Grid Comput.* 19, 3 (2021), 36.

M. Roberts. 2018. Serverless architectures. Availablefrom:https://martinfowler.com/articles/serverless.html

Gabriele Russo Russo, Alfredo Milani, Stefano Iannucci, and Valeria Cardellini. 2022. 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, ieeead, 693–698. https://doi.org/10.1109/PerComWorkshops53856.2022.9767299

Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the ACM European Conference on Computer Systems*. ACM, New York, NY, United States, 351–364.

Vincenzo Scoca, Atakan Aral, Ivona Brandic, Rocco De Nicola, and Rafael Brundo Uriarte. 2018. Scheduling latency-sensitive applications in edge computing. In *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*. Springer, New York, NY, United States, 158–168.

Leila Sharifi, Llorenç Cerdà-Alabern, Felix Freitag, and Luís Veiga. 2016. Energy Efficient Cloud Service Provisioning: Keeping Data Center Granularity in Perspective. *J. Grid Comput.* 14, 2 (2016), 299–325. https://doi.org/10.1007/S10723-015-9358-3

José Simão and Luís Veiga. 2016. Partial Utility-Driven Scheduling for Flexible SLA and Pricing Arbitration in Clouds. *IEEE Trans. Cloud Comput.* 4, 4 (2016), 467–480.

Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. 2020. 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, New York, NY, United States, 1–10.

Zhucheng Tu, Mengping Li, and Jimmy Lin. 2018. 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*. Association for Computational Linguistics, PA, USA, 6–10.

Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. 2023. Mitigating Cold Start Problem in Serverless Computing: a Reinforcement Learning Approach. *IEEE Internet of Things Journal* 10, 5 (2023), 3917–3927.

Erwin Van Eyk, Alexandru Iosup, Cristina L Abad, Johannes Grohmann, and Simon Eismann. 2018. 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, New York, NY, United States, 21–24.

Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. 2021. λDNN: Achieving Predictable Distributed DNN Training With Serverless Architectures. *IEEE Trans. Comput.* 71, 2 (2021), 450–463.

Luxiu Yin, Juan Luo, and Haibo Luo. 2018. Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing. *IEEE Transactions on Industrial Informatics* 14, 10 (2018), 4712–4721.

Hanfei Yu, Athirai A Irissappane, Hao Wang, and Wes J Lloyd. 2021. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, New York, NY, United States, 31–40.