FaaS-Utility: Tackling FaaS Cold Starts with User-preference and QoS-driven Pricing

Henrique Santos[1], José Simão[1,2], and Luís Veiga[1]*

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
[2] Instituto Superior de Engenharia de Lisboa (ISEL / FIT)

Abstract. This study introduces FaaS-Utility, a novel approach aimed at optimizing Function-as-a-Service (FaaS) systems by addressing the critical issue of cold starts, which significantly impede system performance. By introducing a utility function informed by customer preferences and pricing goals, our methodology prioritizes resource allocation to enhance service quality effectively. We implement this strategy within Apache OpenWhisk, demonstrating its integration into a real-world FaaS platform. Our evaluation reveals that the proposed approach notably improves system performance, particularly in over-provisioned states, by reducing latency up to 2.37 times with a maximum additional cost of only 30%. While our method performs best in cold environments, it also maintains performance when applied in warm settings, offering a balanced solution between client and provider through adaptive pricing.

Keywords: Serverless Computing, Resource Allocation, Performance Optimization, Cold Start Mitigation

1 Introduction

Presently, more sophisticated, dynamic 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) [15]. Current implementations of the Function-as-a-Service architecture such as Amazon AWS Lambda and Microsoft Azure Functions focus deeply on the optimization of systems resources and performance while paying little attention to the individual preferences of each customer.

Current scheduling mechanisms [24, 8, 25, 12, 31] 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

* Work partially developed while as a Visiting Researcher with the Hybrid Cloud Computing Group at IBM Research Europe – Zurich.

information can be leveraged by providers, by employing fewer resources when they are scarce, while reducing the price charged to users [23].

We propose an extension to the scheduling mechanism in FaaS that takes 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 is embodied in 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. This entails that a customer using our system will be provided a few additional options, depending on the server's state, when attempting to issue requests, such as monetary discounts for slower execution times or extra monetary costs for her 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.

The rest of the paper is structured as follows: Section 2 discuss some research on serverless computing, specifically focusing on FaaS, its scalability issues, and the cold start problem. Section 3 elaborates on the architecture of the FaaS-Utility system, highlighting how it incorporates user preferences into resource allocation, alongside the details of the implementation of FaaS-Utility within the Apache OpenWhisk platform. Section 4 discusses the methodology and results of the system evaluation, assessing the effectiveness of our approach in mitigating cold starts and improving performance. Finally, Section 5 concludes the paper with a summary of our findings and potential avenues for future research.

2 Related Work

Cloud computing is structured into various service layers, including traditional models like Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS), as well as the more recent Backend-as-a-Service (BaaS), and Function-as-a-Service (FaaS), with this work focusing on the latter. BaaS and FaaS are both considered serverless, and are frequently used in conjunction because they share operational characteristics (such as no resource management) [9, 17, 7].

FaaS allows developers to deploy code in the cloud without managing hardware, offering greater abstraction compared to PaaS, where the provider manages data and server state. FaaS provides transparent scalability, unlike PaaS, where users must plan how to scale, focusing solely on deploying specific application functions [2]. Low latency is crucial for real-time applications like emergency vital sign monitoring, where quick paramedic response is essential. User-wearable sensors play a key role in health monitoring. Edge computing, driven by the need for low latency, leverages serverless frameworks to manage server operations, network, load balancing, and scaling tasks [13, 14]. 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 [28, 29]. 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 [27].

In the work in [27], the authors proposed an intelligent method that chooses the optimum strategy for maintaining the containers running in accordance with the function invocations over time in order to lessen cold start delay and to consider resource usage. While in the work [3], 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 proposed three strategies to reduce cold starts: the naive approach, the extended approach, and the global approach. Additionally, they introduced a lightweight middleware designed to work in tandem with the functions, aiding in the effective mitigation of cold start occurrences.

While numerous studies focus on cloud computing optimization [18, 11, 12], the potential revenue benefits from these optimizations are often overlooked [4]. We discuss both sides of the aspects. 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. Pricing is one the most relevant issues in cloud computing cost methodologies that aim to maximize revenue. Providers use optimization techniques in scheduling to boost customer experience, sometimes neglecting financial outcomes, while recent advancements in cloud computing pricing strategies aim to optimize revenue.

In distributed systems, scheduling is frequently studied to establish a connection between requests and available resources. For clusters [19], clouds [10], and cloud-edge (Fog) systems [22, 20, 16], numerous solutions have been put forth.

In the work presented in [18], the authors 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 extended 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.

Cloud ecosystems' viability is based on effective service pricing [4] and an energy-aware architecture with sensible resource pricing, although many studies focus more on reducing energy consumption than on pricing and billing strategies [21, 4]. Pricing strategy is pivotal in attracting clients who aim for the highest service quality at minimal costs, while cloud service providers focus on boosting income and cutting costs through advanced technologies [1]. The dynamic nature of service demands and quality necessitates a flexible pricing approach beyond fixed pricing, allowing consumers to pay for actual usage and enabling providers to offer fair, competitive rates [4]. Our system offers improvements over existing models by adopting flexible pricing and user-focused optimization, aiming to better balance quality and cost in cloud computing services.

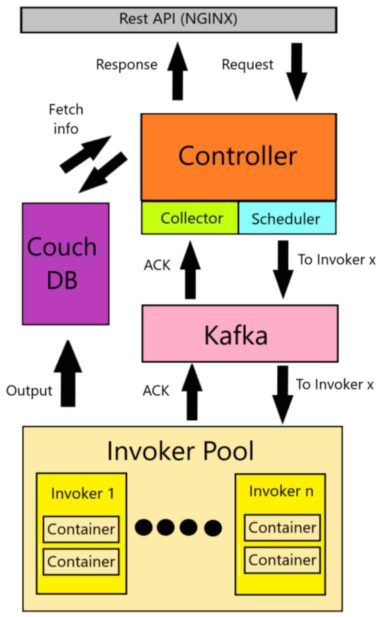

Fig. 1. General architecture with newly added scheduler and Collector component

3 Architecture

In this section, 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: i) during an under-provisioned server state, and ii) an over-provisioned server state. Apache OpenWhisk facilitates the creation, invocation, and outcome querying of functions through a REST interface, using a Controller to assign tasks to a pool of Invokers based on a hashing method and Invoker status [31]. 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 returns to clients the outcomes of the function executions [31].

Scheduler extension. In our extended version of the Apache OpenWhisk architecture, we 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. Clients initiate requests via a REST interface, which are processed by the Controller using information from CouchDB and a utility function to determine

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
$Queue \leftarrow Queue \cup Action$
```

the appropriate Invoker. The Invoker executes the request, and upon completion, updates CouchDB and sends an acknowledgment back to the Controller for future optimization. To accommodate varying system states, a number of pricing options are offered, allowing clients to select from two initial choices in over-provisioned states and three additional ones in under-provisioned states, enabling a tailored service experience.

Pricing options for the client. Two initial pricing options are provided: *Basic* Version which merely finishes the request with no additional benefits; or *Premium* 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. This second option is to use the request to create warm containers for this particular client's repeated uses, resulting in future quicker execution times.

When servers are under-provisioned, leading to potential request queues, three additional pricing tiers are introduced: *Standard priority* maintains normal costs without altering request scheduling priority; *Urgent priority* increases scheduling precedence at a higher cost for time-sensitive tasks; and *Reduced priority* lowers both cost and scheduling priority for users with flexible timelines.

These pricing options cater to different user needs, allowing them to choose based on urgency and cost considerations, like a user needing immediate database error correction versus a student with no immediate project deadlines.

Scheduling during an over-provisioned state. If no special pricing mechanism is applied, scheduling functions normally, but with an added premium: the scheduler will distribute actions across all Invokers to enhance efficiency. This approach benefits clients by ensuring faster execution for repeated requests and optimal Invoker selection for speed, regardless of the Invoker's identity. Clients

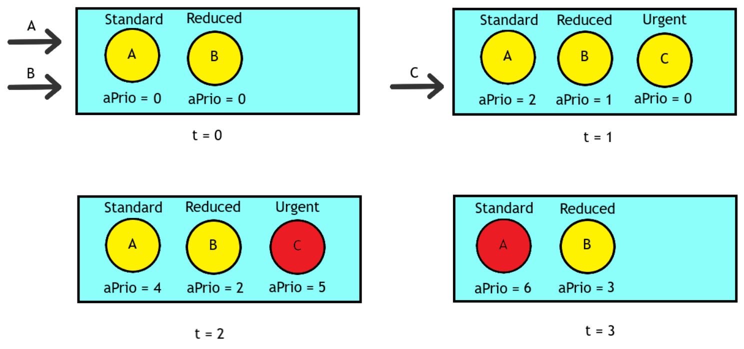

Fig. 2. Four seconds of execution of the priority queue scheme

can also customize their use of the pricing models, applying the *Premium* selectively to specific actions or triggers to optimize cost and performance.

The enhanced scheduler algorithm, detailed in Algorithm 1, introduces a nuanced queuing strategy, triggering action queuing when Invokers are semi-saturated, a more flexible approach compared to the original algorithm's condition of full saturation. This adjustment is particularly effective in over-provisioned states, ensuring that the scheduler optimally utilizes available resources.

Key modifications, marked in the pseudo-code in blue and red, include bypassing pre-warm containers for new container creation on alternative Invokers, and a more exhaustive search for available Invokers rather than settling for the first one found. These enhancements aim to improve resource allocation and system responsiveness by dynamically adapting to the current load on Invokers.

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 be charged is given by:

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

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.

Scheduling during an under-provisioned state. The existing First-In-First-Out (FIFO) priority method, used when server saturation leads to action queuing, offers a low urgency solution for clients. We defend the need for a more advanced priority-aware system that accelerates resolution for time-sensitive requests at an additional cost, also providing options for clients seeking discounts in less urgent situations. A possible scheme (left as future work) can utilize a unique priority value, "aPrio" (absolute priority), which can be adjusted based on the request's urgency level, allowing distinction between requests with identical aPrio values using FIFO. This scheme is visualized in Figure 2, illustrating how different priority levels (p_1, p_2, p_3) could influence the queue management over time. 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:

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

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.

3.1 Implementation Details

The solution proposed for over-provisioned state has two main goals: (1) set up containers for future workloads, and if possible (2) combine the work of all Invokers for an even faster possible execution. Thus, we tackled both problems separately, starting with the more architecturally demanding problem (1).

Invokers in Apache OpenWhisk operate independently without knowledge of each other's conditions, limiting the ability to dynamically create containers based on system-wide states and preventing the generation of empty warm containers. The controller, with some awareness of the Invoker pool's state, becomes a focal point for managing container allocation, particularly in over-provisioned scenarios where additional action invocations don't strain existing resources due to container isolation. Increased workload on the controller and Kafka during over-provisioned states introduces minimal overhead, as Kafka is designed for high throughput and low latency, capable of handling significant data volumes. This system design ensures that resource augmentation in an over-provisioned state does not compromise Invoker performance, though it warrants attention during performance evaluation.

To enhance the algorithm in 'ShardingContainerPoolBalancer.scala' within Apache OpenWhisk, the proposed modification involves extending the search for available Invokers beyond the first one found, aiming to engage multiple Invokers to prepare or execute the action. This approach is designed to ensure the action is spread across all potential Invokers, whether to prepare warm containers or create cache data, addressing a key objective. By executing the action on all available Invokers, the system capitalizes on the quickest response, as the fastest Invoker will deliver the result back to the controller, optimizing performance. To achieve this, the scheduling function must be adjusted so its completion criteria are met only after all Invokers have been considered, ensuring comprehensive action distribution. Additionally, the modification seeks to preserve the stability of the home Invoker metric, allowing users to opt-out of this enhanced functionality if desired.

4 Evaluation

In this section, we will address the system performance and the assessed metrics. We deploy the system with 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.

FaaS Benchmarks. Four FaaS workloads (F1-F4) are used in the evaluation of our system those being: Sleep functions, File hashing, Video transformation, and Image classification, taken from FaaS benchmarking found in the literature [6]:

– F1 - Sleep functions: a simple, low-overhead operation that can be used to measure infrastructural overheads, in our case the scheduling infrastructure, of a FaaS platform.
– F2 - File hashing: a relatively simple operation that can be used to test the ability of the system to handle file inputs and outputs.
– F3 - Video Transformation: it exercises many of the key features of a FaaS 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 to assess system ability to handle high levels of concurrency and scale horizontally.
– F4 - Image classification: 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 of real-world usage in FaaS [18], especially in machine learning applications [30, 26].

Metrics. The performance of our system is evaluated using three key metrics: Latency (time for request processing and response), Scheduling delay (time from request readiness to execution, reflecting the scheduler's efficiency), and Resource usage (assessing how well the system utilizes memory and CPU). To understand memory consumption and system overload, we analyze logs from OpenWhisk components, specified in the docker-compose.yml file, which helps in identifying performance bottlenecks and resource management. These metrics are measured and compared with the Apache OpenWhisk default scheduler.

Evaluation Environment. The evaluation environment for the updated Apache OpenWhisk scheduler involved a minimal cluster setup with one container per OpenWhisk component and three invokers managed by a single *Controller*, designed to stress-test the system. Testing variables included the types of actions, the volume of requests, and the concurrency of users, with each test conducted in either a 'cold' or 'warm' server state to simulate varying user demands and assess the system's responsiveness without needing repeated authentication. JMeter, an open-source tool, was employed to assess the performance of web applications, APIs, and other services by simulating numerous user interactions to identify potential bottlenecks under various load conditions.

The system's evaluation involved actions F1, F2, F3, and F4 to examine various operational aspects, from quick tasks and delay measurements to CPU-intensive processes. Each action was tested in Default, Base, and Spread versions to compare, respectively: i) the original OpenWhisk system, ii) the modified version without enhancements, and iii) the version with new functionalities, enabling a thorough analysis of the system's adaptability and performance in diverse computing environments. During testing, all actions are established in the test setup phase, and the time spent on this initial creation is not included in the test metrics since it does not impact the modified components of the updated system. To evaluate the enhanced scheduler, two distinct sub-environments were crafted, reflecting the system's initial state before each specific test is conducted.

1. Cold Sub-environment *"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 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 is done after all containers within the invokers enter the paused state.

2. Warm Sub-environment *"W"*: a fully cold enviroment is 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 achieved in sub-environment 1, 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 machine representative of a typical low-mid cloud server instance with an Intel® Core™ i7 CPU @ 2.60GHz processor with 4 physical cores and 8 threads on UbuntuLTS 64-bit.
- **Hardware *"B"***: a machine representative of a more capable cloud server instance, but where resource usage optimization is still challenging, to assess the extent of potential gains to be achieved at scale. This hardware B uses a Intel(R) Core(TM) i7 CPU @ 3.20GHz, 6 Physical Core(s), 12 Logical Processor(s).

4.1 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-environment it uses followed by which hardware it utilizes, for example, "Test 1 (W-A)" is test number 1 and uses both a Warm sub-environment and hardware A. The full analysis is described in [5], with detailed results for each action (F1-F4) used; we leave here the main findings.

Test 1 (W-A), focused just on preliminary determining if our Base scheduler performed similarly to Deafult, employing solely F1 only for simplicity. Results show our enhanced scheduler performed slightly better under the same circumstances, so we could 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, and 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 previously more noticeable. We were able to conclude that hardware can indeed affect the value provided by our functionality, as we were only able to see significant improvements for F4 function in more powerful hardware B.

Test 3 (W-A), 4 (W-A), and Test 5 (W-B) 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. Thus, our proposed functionality needs not be used for already ideally warm environments, as it may lead to unnecessary additional overloading of the system.

4.2 Utility Function Evaluation

The enhanced scheduler's performance analysis includes evaluating its impact on client costs and provider resource consumption, focusing on critical metrics like latency reduction, execution time, and the utility function's alpha parameter. Table 4.2 offers a comparative overview, detailing the trade-offs between performance improvements and resource usage, aiding stakeholders in assessing the scheduler's efficiency and cost-effectiveness in a concise manner.

The effectiveness of the enhanced scheduler varies with different scenarios, particularly showing limited benefits in already warm environments like in Test 3, where additional invocations did not mitigate the risk of cold starts and led to performance degradation. However, the introduction of a variable pricing factor, α, influenced by the seller, can adjust the final cost, offering a mechanism to counteract the performance drop or unnecessary resource usage, enabling a

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

Table 1. Utility evaluation

negotiation space between the client and the seller for better interaction and agreement on the service value.

Hardware variations also impact the scheduler's performance benefits, as observed in tests 2 and 6, with different hardware setups. The scheduler's performance varied significantly between hardware types, with certain tests showing better latency improvements on weaker hardware. The adjustment of the α value by the seller can make the resource use more cost-effective for the client, especially if enhanced performance justifies additional resource consumption, highlighting a nuanced trade-off between resource use, performance gain, and cost.

Finally, Figure 3 studies how the system's environmental awareness affects the seller's pricing leverage and the overall cost-effectiveness of the scheduler. The functionality's deployment in its intended cold environment versus an unnecessary warm environment demonstrates the adaptive cost strategy's role in ensuring that the scheduler's use aligns with its designed benefits, thereby optimizing the balance between performance enhancement and resource expenditure.

5 Conclusion

We created a scheduler extension architecture that considers user preferences when adjusting scheduling to provide a higher quality of service to the user. Apache OpenWhisk was used to implement our solution. For over-provisioned system conditions a new functionality that we named "Action-Spreading" was implemented to allow warm containers to be set up for a reduced cost in prepa-

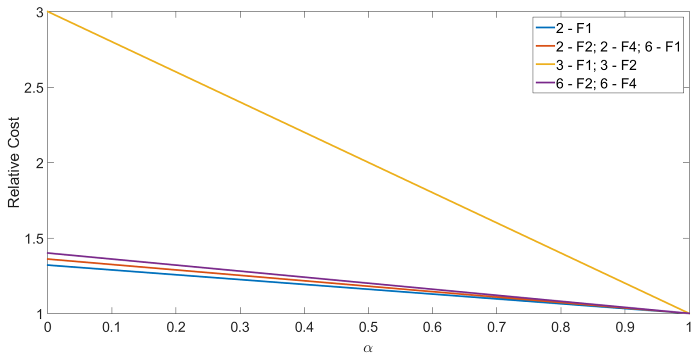

Fig. 3. Cost's behaviour depending on α values

ration for an influx of requests. 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.

As future work, we consider that the development of priority-based queueing extensions for Kafka and other similar components (that do not have priority-aware mechanics in mind), specifically for FaaS systems, would be a great avenue for future research in this field. This would enable enforcing our proposed scheduling approach during an under-provisioned state, by being able to speed up requests from higher priority users or functions, while without hurting the scalability of the controller.

Acknowledgements: This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI:10.54499/UIDB/50021/2020). This work was supported by: "DL 60/2018, de 3-08 - Aquisição necessária para a atividade de I&D do INESC-ID, no âmbito do projeto SmartRetail (C6632206063-00466847)". This work was supported by the CloudStars project, funded by the European Union's Horizon research and innovation program under grant agreement number 101086248.

References

1. M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad. Cloud computing pricing models: a survey. *International Journal of Grid and Distributed Computing*, 6(5):93–106, 2013.
2. I. Astrova, A. Koschel, M. Schaaf, S. Klassen, and K. Jdiya. Serverless, faas and why organizations need them. *Intelligent Decision Technologies*, 15(4):825–838, 2021.

3. D. Bermbach, A.-S. Karakaya, and S. Buchholz. Using application knowledge to reduce cold starts in faas services. In *Proceedings of the ACM Symposium on Applied Computing*, pages 134–143, New York, NY, United States, 2020. ACM.
4. S. R. Dibaj, L. Sharifi, A. Miri, J. Zhou, and A. Aram. Cloud computing energy efficiency and fair pricing mechanisms for smart cities. In *IEEE Electrical Power and Energy Conference (EPEC)*, pages 1–6, New York, NY, United States, 2018. IEEE.
5. H. d. C. R. dos Santos. Faas-utility. Master's thesis, Instituto Superior Técnico, U. Lisboa, 2023.
6. V. Dukic, R. Bruno, A. Singla, and G. Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, New York, NY, United States, 2020. ACM.
7. B. Janakiraman. Serverless, 2016.
8. Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya. Automated fine-grained cpu cap control in serverless computing platform. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2289–2301, 2020.
9. S. Kounev, N. Herbst, C. L. Abad, A. Iosup, I. Foster, P. Shenoy, O. Rana, and A. A. Chien. Serverless computing: What it is, and what it is not? *Commun. ACM*, 66(9):80–92, aug 2023.
10. G. Lee, B. Chun, and H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 1–5, Portland, OR, 2011. USENIX Association.
11. L. Lin, P. Li, J. Xiong, and M. Lin. Distributed and application-aware task scheduling in edge-clouds. In *Proceedings of the International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 165–170, New York, NY, United States, 2018. IEEE.
12. A. Madej, N. Wang, N. Athanasopoulos, R. Ranjan, and B. Varghese. Priority-based fair scheduling in edge computing. In *Proceedings of the IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 39–48, New York, NY, United States, 2020. IEEE.
13. S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.
14. A. Palade, A. Kazmi, and S. Clarke. An evaluation of open source serverless computing frameworks support at the edge. In *Proceedings of the IEEE World Congress on Services (SERVICES)*, volume 2642, pages 206–211, New York, NY, United States, 2019. IEEE.
15. T. Pfandzelter and D. Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *Proceedings of the IEEE International Conference on Fog Computing (ICFC)*, pages 17–24, New York, NY, United States, 2020. IEEE.
16. A. Pires, J. Simão, and L. Veiga. Distributed and decentralized orchestration of containers on edge clouds. *J. Grid Comput.*, 19(3):36, 2021.
17. M. Roberts. Serverless architectures, Nov. 2018.
18. G. R. Russo, A. Milani, S. Iannucci, and V. Cardellini. Towards qos-aware function composition scheduling in apache openwhisk. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 693–698, ieeead, 2022. IEEE.
19. M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the ACM European*

Conference on Computer Systems, pages 351–364, New York, NY, United States, 2013. ACM.

20. V. Scoca, A. Aral, I. Brandic, R. De Nicola, and R. B. Uriarte. Scheduling latency-sensitive applications in edge computing. In *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, pages 158–168, New York, NY, United States, 2018. Springer.
21. L. Sharifi, L. Cerdà-Alabern, F. Freitag, and L. Veiga. Energy efficient cloud service provisioning: Keeping data center granularity in perspective. *J. Grid Comput.*, 14(2):299–325, 2016.
22. J. N. Silva, P. Ferreira, and L. Veiga. Service and resource discovery in cycle-sharing environments with a utility algebra. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–11, 2010.
23. J. Simão and L. Veiga. Partial utility-driven scheduling for flexible SLA and pricing arbitration in clouds. *IEEE Trans. Cloud Comput.*, 4(4):467–480, 2016.
24. A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi. Ensure: efficient scheduling and autonomous resource management in serverless environments. In *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10, New York, NY, United States, 2020. IEEE.
25. A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10, 2020.
26. Z. Tu, M. Li, and J. Lin. Pay-per-request deployment of neural network models using serverless architectures. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 6–10, PA, USA, 2018. Association for Computational Linguistics.
27. P. Vahidinia, B. Farahani, and F. S. Aliee. Mitigating cold start problem in serverless computing: a reinforcement learning approach. *IEEE Internet of Things Journal*, 10(5):3917–3927, 2023.
28. E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann. A spec rg cloud group's vision on the performance challenges of faas cloud architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 21–24, New York, NY, USA, 2018. Association for Computing Machinery.
29. E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann. A spec rg cloud group's vision on the performance challenges of faas cloud architectures. In *Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 21–24, New York, NY, United States, 2018. ACM.
30. F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu. λdnn: Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers*, 71(2):450–463, 2021.
31. H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd. Faasrank: Learning to schedule functions in serverless platforms. In *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 31–40, New York, NY, United States, 2021. IEEE.