μ: A Bolt-On Approach for Faster and Cheaper Serverless Computing

Diogo Pacheco, João Barreto, Rodrigo Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Abstract

Serverless computing shifts the responsibility of managing and configuring the cloud infrastructure from the user to the provider. However, in the FaaS model, users are still faced with non-trivial resource allocation and management choices, in particular with respect to the way that intra- and inter-function parallelism is managed. As a consequence, deploying a cloud function with an incorrect configuration for either of these two dimensions of parallelism can lead to an unnecessary increase in execution time, cost, or both.

In this paper, we call the attention to the fact that this is one of the final hurdles to realizing the vision behind serverless computing. To overcome it without requiring changes to the existing cloud infrastructure, we propose the design of μ, a system that transparently manages the balance between intra- and inter-function parallelism on behalf of the user. μ is designed as a shim layer to be mostly transparent to the application developer and fully compatible with today's cloud services. Our design allows for running multiple tasks in a single function, using the extra resources that the user is already paying for. This allows for maximizing resource utilization, and leads to cost reductions and performance improvements, without the requiring the user to reason about resource allocation.

ACM Reference Format:

Diogo Pacheco, João Barreto, Rodrigo Rodrigues, INESC-ID, Instituto Superior Técnico, Universidade de Lisboa . 2023. μ: A Bolt-On Approach for Faster and Cheaper Serverless Computing. In *4th Workshop on Resource Disaggregation and Serverless (WORDS '23), October 23, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605181.3626284

1 Introduction

Serverless computing is an extension of cloud computing where the concept of a server is hidden to the user. In this paradigm, the responsibility of managing and configuring the cloud infrastructure is shifted from the user to the provider, making it a simpler option with a lower entry barrier when compared to traditional cloud computing models like IaaS.

FaaS is a serverless solution where the user uploads small pieces of code known as cloud functions or lambdas. These can then be configured to execute on a given trigger. Functions run their code on a temporary and ephemeral virtual machine or container, with the user paying only for the resources used during the execution time of the function. This execution environment has access to a certain number of vCPUs and amount of memory, defined when the function is deployed [3].

In many respects, FaaS embodies the vision underlying serverless computing [16]: the developer only needs to write the code for the application, upload it to the provider and assign triggers. Load balancing is no longer an issue, no setup whatsoever is needed, and scaling is assured by the provider, with the user paying only costs that are proportional to the service load, due to the pay-as-you-go nature of the services.

However, there is still one lingering aspect of resource management that is left for cloud users to handle, which is parallelism. When building and deploying a FaaS application that will execute a set of parallel tasks, the programmer is called to choose the right balance between parallelism *across* multiple function invocations and *inside* each function. As we show later in the paper, a wrong choice in the balance between inter- and intra-function parallelism can lead to a poor utilization of the available resources: in one extreme of the spectrum, if the programmer only exploits inter-function parallelism (with many invocations of a single-threaded function), then the resources allocated to the function may be underutilized and the user may be paying unnecessary monetary costs. In contrast, if the function code is only instantiated once but runs too many threads in parallel, then the resources will be oversubscribed and performance will suffer. Handling this aspect of resource management may be the final roadblock to realizing the vision of serverless computing. This paper aims at establishing a road-map to remove this roadblock with a bolt-on architecture.

To demonstrate that it is possible to relieve the user from the burden of deciding between intra- versus inter-function parallelism, while also maintaining backward compatibility with today's offer of FaaS services, we present the design and a preliminary evaluation of μ, a system that is able to reduce costs and improve performance by automatically leveraging the internal parallelism that is present on FaaS function instances, to the extent that the system allows it. The goal of μ

This work is licensed under a Creative Commons Attribution International 4.0 License.
WORDS '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0250-1/23/10.
https://doi.org/10.1145/3605181.3626284

is to provide a service capable of executing multiple requests, called tasks, on a single function invoked in FaaS, known as a worker. By doing so, μ takes advantage of the ability to have parallelism within a cloud function. Furthermore, to dynamically determine the right balance between inter- and intra-function parallelism, the design of μ includes a resource monitoring component that keeps track of the status of each worker; and a scheduler that uses that information to dispatch incoming task invocations to either an existing under-subscribed worker or a newly-spawned worker.

Finally, we identify key limitations in our preliminary design and discuss the challenges and opportunities to work around them.

We implemented μ as a shim layer around function instances that are deployed on AWS Lambda. The results from our preliminary evaluation using a benchmark cloud function show that μ reduces costs by up to 72.9% while accelerating the execution time by up to 61.2%.

2 Background and related work

From the user's perspective, serverless – and FaaS, in particular – is easier to use than the traditional serverful alternatives (such as IaaS), since it no longer requires a basic understanding of system administration. In FaaS, the user only needs to write the code for the application, upload it to the provider and assign triggers. Each function must also be configured with a certain number of vCPUs and an amount of memory that will be allocated to each invocation of the function. The configuration of these resources varies depending on the specific service being used; e.g., in some providers the number of vCPUs is proportional to the amount of memory chosen by the user, while in others the vCPU allocation is not configurable [3]. However, the programmer is still responsible for choosing the right balance between inter- and intra-function parallelism in order to make the best possible use of the resources that were allocated.

In recent years, the research community has essentially focused on the problems associated with the first dimension of this dichotomy, namely managing inter-function parallelism in FaaS [8, 9, 11, 15, 17, 19].

Inter-function parallelism occurs when multiple instances of one or more functions are called in parallel, and is one of the most fundamental features of FaaS. However, effectively exploiting it raises challenges such as efficiently decomposing the workload into parallel function invocations, sharing data between the parallel function instances, as well as rapidly scaling and scheduling function invocations. Recent proposals such as Wukong [11], gg [9], FIRM [19], Kappa [15], Crucial [8] and NumPyWren [17] try to overcome such challenges for this type of parallelism.

In contrast to inter-function parallelism, intra-function parallelism consists of having parallel computations inside a single function invocation on FaaS. It can be achieved by having the function code spawn multiple threads. Inter-function parallelism can be traded for intra-function parallelism, which enables reducing the number of function invocations (when compared to running the same set of tasks in single-threaded functions, in parallel). This brings two potential advantages. First, since multiple function invocations share the resources of the same container, it can improve resource usage – namely, CPU and memory –, hence reducing costs. Second, it can reduce the occurrences of cold starts and, thus, mitigate the associated performance penalty.

To our knowledge, the benefits of intra-function parallelism in FaaS have only been studied in depth by Kiener et. al. [13]. Their results show that intra-function parallelism can effectively accelerate function execution and improve resource utilization, leading to a cost reduction for the user in commercial FaaS offerings. As of today, for a programmer to take advantage of such benefits, it is his or her responsibility to choose the intra-function parallelism degree and implement such parallelism inside the respective functions. The goal of our work is to overcome what we perceive as being the last hurdle separating us from a vision where the users of serverless computing do not have to reason about any resource management.

In a distinct research avenue, recent proposals rethink the design of FaaS platforms to enable each container to run multiple function instances in parallel, either as processes [4, 14, 18] or threads [12] within a container. Such proposals are able to mitigate many bottlenecks of today's FaaS container-based platforms when faced with bursts of function invocations [18], including the inefficient use of resources as well as cold-start costs. Therefore, their advantages partly overlap with our work. However, they imply re-engineering the FaaS platform, whereas the bolt-on nature of our proposal enables cloud customers to deploy μ today on any FaaS service.

3 μ

In this section, we describe the proposed architecture of μ and its proof-of-concept implementation.

3.1 Architecture

The architecture of μ is outlined in Figure 1. It consists of a centralized coordinator running on an AWS EC2 [7] instance, responsible for the overall orchestration and scheduling of function invocations; a worker function, executed as an instance of FaaS, which acts as an execution environment capable of running several tasks in parallel (intra-function parallelism); and a shim layer API package made available to clients, so they can spawn their own instances of tasks and deploy them on μ. In this architecture, each worker collects its own resource metrics and periodically sends them to the coordinator, to guide its scheduling decisions. The coordinator keeps track of pending tasks that need to be executed in

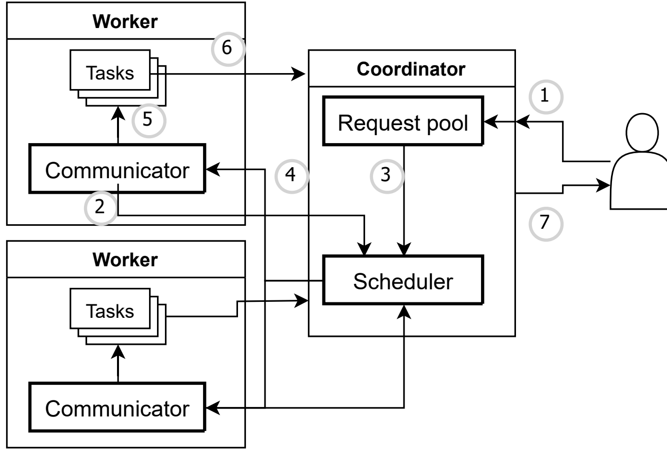

Figure 1. System architecture of μ

a request pool, and schedules them based on the information it maintains about the state of the system.

3.2 Life cycle of a function and the scheduling algorithm of μ

We now explain how functions are invoked in μ and present our scheduling algorithm. Our current design assumes that a worker only executes tasks that correspond to invocations of the same function by the same client. We discuss how this limitation can be lifted in Section 5.

Task invocation requests are sent from the user via its shim layer. Whenever such a request is received by the coordinator (1), it is kept in a request pool. Then, as soon a worker with enough unused resources polls the coordinator (2), the scheduler takes a request from the pool (3) and dispatches it to that worker (4). The worker immediately spawns a new executor thread and assigns it the new task (5). Upon completion, the executor thread submits the results to the coordinator (6) who in turn returns them to the user (7).

The logic of the coordinator is detailed in Algorithm 1, and it describes how μ schedules tasks to workers. The idea is that the coordinator is continuously waiting for poll requests to arrive from one of the workers. When such a request arrives, the coordinator checks if the current duration of the function is still within 80% of the execution timeout. If so, then the coordinator additionally checks if it is possible to schedule a new task to run in that worker, which requires the combination of the following factors: (1) the reported CPU and memory usage are under predefined thresholds, (2) the number of tasks the worker is currently running is smaller than the number of threads it can run, and (3) the request pool is not empty. If all these checks pass, then a new task is assigned to that worker and the response reflects this fact.

In case a new task has not been assigned to this worker, then the coordinator checks if it is possible to signal the worker to shutdown (in case it is no longer running any tasks). Otherwise, the worker can continue its processing without any change.

This logic allows for a smooth shutdown of FaaS instances: when either a worker is nearing the maximum running time provided by the FaaS platform or there are no more tasks to run, then this algorithm identifies the worker as end-of-life and no more tasks are sent to it. Then, upon completion of its tasks, the container can be shut down.

```
begin
  while true do
    req ⟵ server.listen()
    worker ⟵ lookup(req.workerId)
    if now − worker.start ≤
      thr_time × execTimeout then
      if req.currCPU ≤ thr_cpu and
        req.currMem ≤ thr_mem and
        |req.currTasks| < worker.threads − 1
        and requestPool ≠ ∅ then
          task ⟵ requestPool.pop()
          req.connection.send(task)
          continue
      end
    end
    if req.currTasks == ∅ then
      req.connection.send(shutdown)
    end
    else
      req.connection.send(continue)
    end
  end
end
```

Algorithm 1: Pseudo-code of the coordinator. This implements the scheduling algorithm of μ.

3.3 Implementation

We implemented the worker as a generic wrapper function, whose main thread thread starts by registering the new worker with the coordinator, then enters a loop that periodically measures its container's usage of CPU and memory and sends this data to the coordinator via HTTP. If, in response, the coordinator dispatches a new task to this worker, a new thread is spawned to run the code of that task. This wrapper function was developed in golang and makes use of goroutines to handle multi-threading. The coordinator is an HTTP server developed in NodeJS using the Express framework. It contains internal endpoints, to be used by the workers, and external endpoints, to be used by the client. The workers run on AWS Lambda [5] and the coordinator

Memory in MB	Number of cores	Cost in USD ($) per 1ms
2048	2	$0.0000000333
4096	3	$0.0000000667
6144	4	$0.0000001000
8192	5	$0.0000001333
10240	6	$0.0000001667

Table 1. Selected memory configuration with the associated number of cores and cost per 1ms

is hosted on an AWS EC2 [7] instance. We believe, however, that it is straightforward to adapt our implementation to run on any other FaaS platform. The shim layer is currently implemented as an API package for Python, providing an invocation method that can be used instead of the conventional FaaS invocation methods. As such, clients can incorporate this method in their serverless applications to take advantage of μ's capabilities. By spawning new FaaS instances through the shim layer instead of the conventional API, these are then handled through the sequence of steps depicted in Figure 1. This allows the client to take advantage of the intra-function parallelism spawned by μ even when the function's code is single-threaded.

4 Evaluation

In this section we describe a preliminary evaluation of the advantages of using μ to automatically trade inter- for intra-function parallelism. To quantify such advantages, we will consider two main metrics, *cost for the user* and *latency*.

4.1 Methodology

In this study, we use the image thumbnailer benchmark from the SeBS benchmark suite [10], which we ported to the Go language. This benchmark receives as input two AWS S3 bucket names, two AWS S3 file keys, and a width and height for the final image. In this benchmark, a task starts by downloading a 3MB image from the first bucket using the first key. For this, we used the official AWS SDK package for Golang, in the same way that the original benchmark uses the AWS S3 SDK for Python. The next step is to perform the thumbnailing operation on the downloaded image, resizing it to a size of 100px x 100px. The original benchmark used Pillow [2], a popular Python image processing library. In our implementation of the benchmark, we use the Imaging package of golang. The final step is to upload the image to the second bucket using the second key. We again used the AWS S3 SDK for this.

For all tests, our workload consists of a large number of invocations of the above-mentioned task. We developed a script in Python that uses both standard AWS Lambda invocations via the Function URL invocation method as well as our μ shim layer package. As the experimental baseline, which we denote as *conventional FaaS*, we consider the usual approach of invoking the original benchmark code on a standard AWS Lambda instance for each task to be performed in the workload.

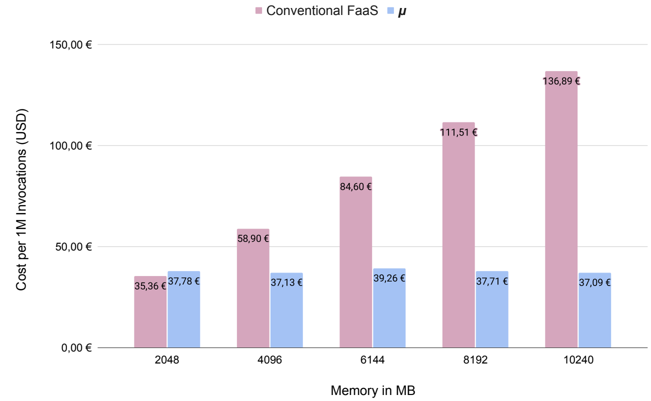

Figure 2. Cost analysis for different memory configurations for a throttled test

AWS Lambda allows for multiple memory configurations. These memory configurations define proportionally how many vCPU cores each worker has access to. For our test runs, we established five different memory configurations to be used. We selected memory configurations that allow us to start at two cores and go all the way up to six cores. The specific memory values, associated number of cores, and cost per 1ms of invocation (according to AWS Lambda [1]) are listed in Table 1. These costs correspond to the configuration we used to deploy functions, namely the x86 architecture running in region eu-west-3 (Paris) under the category of the first six billion invocations.

We used AWS Cloudwatch [6] to measure function execution times and our calculations of the execution costs is based on the official AWS Pricing Table [1].

Cost analysis. One of the key benefits of μ is cost reduction. The idea behind this is that by introducing intra-function parallelism we can drastically reduce the number of (FaaS) function invocations needed to execute all tasks on a given workload and, with that, reduce the total costs for the user.

To evaluate the total costs of μ and compare them with the AWS Lambda baseline, we used a test where we execute a total of N requests while maintaining a cap on the amount of concurrency, namely 8 maximum concurrent requests. In our experiment, we used 100 invocations of the Thumbnailer benchmark as the value of N. We tested this workload with all five different memory configurations and corresponding costs from Table 1.

The results are shown in Figure 2. In this graph, we plot the cost per million invocations on each different memory configuration of a thumbnailer workload using μ, and compare it the AWS baseline. We also depict the number of invocations for each system across every memory configuration in Figure 3.

Comparing both setups, we see a reduction in cost proportional to the increase in the number of cores. In particular, from 4096MB and beyond, each worker of μ has at least two cores to execute tasks in parallel (given that one

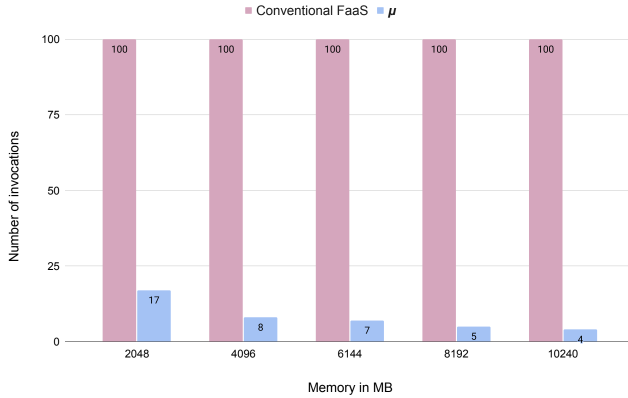

Figure 3. Number of invocations for different memory configurations for a throttled test

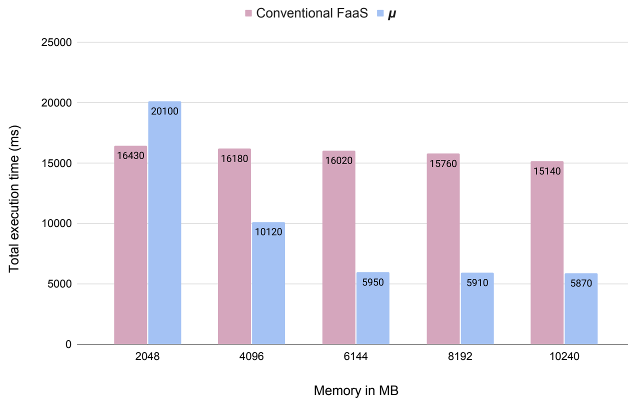

Figure 4. Latency analysis for different memory configurations for a throttled test

core is reserved to resource monitoring and communication). Therefore, for such configurations, μ is able to exploit intra-function parallelism, resulting in a significant cost reduction when compared to the baseline. To better understand where these savings come from, Figure 3 shows that, from 2048 to 4096MB, the number of FaaS function invocations is cut in half, since we double the number of tasks per function instance. This compensates for the increase in cost per millisecond of using more powerful instances, allowing μ to finish computations faster while keeping the cost envelope relatively constant. In contrast, the AWS baseline nearly doubles in cost when going from 2048 to 4096MB because it does not take advantage of the extra processing power to achieve a shorter lifetime, and therefore has to pay the price for the larger instance. This trend is maintained as we increase the memory allocated to the worker and, as a consequence, the number of cores. As a whole, if we factor out the overhead costs, we can see a $1/N$ relation between the cost of the baseline and the cost of our system, where $N = \#Cores - 1$ or, in other words, N is the number of threads free to execute tasks in a given configuration.

Latency analysis. In Figure 4, we compare the time needed to complete 100 invocations on each different memory configuration of a thumbnailer workload using μ with the same workload being processed by native AWS Lambda.

Comparing both setups, we see a significant speedup on higher memory configurations when comparing μ with conventional FaaS. This is explained by the fact that, from 4096MB onward, each worker of μ has at least two cores to execute tasks in parallel. Therefore, for such configurations, μ is able to exploit intra-function parallelism. resulting in significant speedups. As seen in Figure 3, starting at 4096MB, the number of FaaS function invocations is cut in half, since we double the number of tasks per function instance. The reduced number of FaaS invocations results in fewer function start delays and faster response times. This trend stops at 8192MB due to our static policy for spawning workers, which limits the amount of concurrency.

In summary, our preliminary evaluation shows that μ achieves significant cost and latency reductions when compared to a standard AWS Lambda deployment for any memory configuration that provides three or more vCPUs. In particular, as the allocated memory grows, the savings that are attained by μ increase. Regarding latency, we can conclude that this is also significantly improved by μ, due to the fact that it can solve the over-provisioning problem and make better use of the allocated resources when compared to native AWS Lambda.

5 Discussion

While our preliminary design and implementation of μ already shows promising gains in the benchmark evaluated in the previous section, it also leaves the door open to addressing several challenges and opportunities.

Overheads. A drawback of the current implementation of μ, which affects both performance and cost, is the non-negligible amount of overhead. Namely, in our design: μ introduces delays on instance spin-up due to the registration mechanism; it requires I/O operations between each worker and the coordinator, forcing each instance to dedicate a thread to the resource gathering and communication loop and potentially reaching a bottleneck on bandwidth; and the fact that workers fetch tasks based on a polling loop may cause noticeable delays, particularly for short functions. While we intend to refine some of the design choices to improve these aspects, we also envision the possibility of addressing these by directly modifying an existing FaaS platform instead of taking a bolt-on approach. Incorporating our system in a full-fledged FaaS platform can remove the need for a polling mechanism to monitor resources, thus freeing up extra resources and enabling intra-function parallelism even for very short tasks.

Multiplexing cores across tasks for better CPU usage. One of the limitations of our current prototype is that it has

a somewhat constrained division of resources within the worker, namely since it allocates one virtual core to monitoring and communicating with the coordinator, and then assigns one entire core to each task. This can be overcome by modifying the scheduler to exploit configurations where the number of tasks running in a worker may be higher than the number of vCPUs allocated to the function instance. This requires a more sophisticated resource monitoring mechanism than the one currently employed by μ. One other possible direction is to build a better resource usage prediction mechanism. If resource usage for any given request could be more accurately predicted, then our decision-making component could make more informed decisions regarding where and when to dispatch tasks. An ideal solution would be a combination of both approaches, where both the resource estimation component and the resource monitoring component are improved and can provide significantly more accurate data to the decision-making component.

Dynamic instance spawning. Even though μ has the elasticity to dynamically spawn more worker instances as more requests arrive, the threshold that the coordinator uses to decide when more instances are needed is not dynamic. This can also be improved in a similar way to the previous points, namely through improved resource monitoring and resource estimation components, to accurately decide exactly how many instances are needed to keep the response rate in check with the request rate.

Sharing data between tasks. A conceptual difference between μ's tasks and conventional threads within an instance of FaaS is that threads share an address space and communicate through that shared memory. In μ, sharing memory is made difficult by the fact that the scheduler has the option of instantiating either intra- or inter-function parallelism, depending on the current system load.

Given that our current prototype supports only code written in Go, this can be solved seamlessly due to the fact that goroutines share data through channels. The implementation of these channels can thus by modified so that the data is communicated via the network in the case of two tasks that run on different instances of FaaS. However, to extend μ to other languages, we need to resort to other mechanisms akin to far memory.

Heterogeneous workloads. Currently, our prototype only supports task instances that are spawned by the same client and run the same code. Removing the restriction of supporting only one cloud client per worker is difficult in our bolt-on approach, since there is there is an intrinsic need for isolation for performance and security reasons. However, if the same client deploys a pipeline of multiple functions, then it would be possible for our scheduler to leverage this by co-locating tasks with different profiles. In particular, if a given user's pipeline contains some functions that are more CPU-intensive and others that are more I/O-intensive, then co-locating instances of these two classes can lead to a better resource utilization.

6 Conclusion

In this paper we presented μ, a bolt-on system for reducing the cost and improving the performance of FaaS. μ takes advantage of intra-function parallelism in an adaptive fashion, enabling the use of the extra resources that the user is already paying for in each function instance to run more than one task. This can improve resource utilization, thus providing cost reductions, and, in some cases, performance improvements. Our proof-of-concept prototype shows that the system has the potential to improve both performance and cost. Furthermore, μ is an important step towards realizing the vision of serverless computing, where cloud users abstract away all resource management. In the future, we intend to refine our current design, namely to support pipelines of multiple functions, and also consider a native implementation that modifies a FaaS platform.

Acknowledgements

We thank the anonymous reviewers for the feedback. This work was supported by the Fundação para a Ciência e Tecnologia through projects PTDC/CCI-INF/6762/2020 and UID/CEC/50021/2019, and the Center for Responsible AI (Application number C645008882-00000055.PRR).

References

[1] [n. d.]. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/. [Online; accessed 22-May-2023].
[2] [n. d.]. Pillow - Python Imaging Library. https://pypi.org/project/Pillow/. [Online; accessed 22-May-2023].
[3] 2023. *EuroSys '23: Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy). Association for Computing Machinery, New York, NY, USA.
[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.
[5] Amazon. [n. d.]. Amazon Web Services Documentation. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. [Online; accessed 6-May-2022].
[6] AWS. [n. d.]. Amazon Cloudwatch. https://aws.amazon.com/cloudwatch/. [Online; accessed 19-May-2022].
[7] AWS. [n. d.]. Amazon EC2: Elastic Compute Cloud. https://aws.amazon.com/ec2/. [Online; accessed 19-May-2022].
[8] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535
[9] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo

Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 1–15. https://doi.org/10.1145/3419111.3421286

[10] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: a serverless benchmark suite for function-as-a-service computing. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luís Veiga (Eds.). ACM, 64–78. https://doi.org/10.1145/3464298.3476133

[11] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[12] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[13] Michael Kiener, Mohak Chadha, and Michael Gerndt. 2021. Towards Demystifying Intra-Function Parallelism in Serverless Computing. In *WoSC '21: Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021, Virtual Event, Québec City, Canada, 6 December 2021*. ACM, 42–49. https://doi.org/10.1145/3493651.3493672

[14] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[15] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[16] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, João Carreira, Neeraja Jayant Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM* 64, 5 (2021), 76–84. https://doi.org/10.1145/3406011

[17] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless Linear Algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 281–295. https://doi.org/10.1145/3419111.3421287

[18] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. https://doi.org/10.1145/3579371.3589069

[19] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: a programming framework for serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 328–343. https://doi.org/10.1145/3419111.3421277