Can user facing and background functions coexist in serverless?

Anonymous Author(s)

1 Introduction

A world where cloud compute runs as serverless functions is attractive to developers and providers: developers pay only for what they use, while having access to many resources when needed; and cloud providers can maximize utilization by intelligent scheduling of functions, in contrast to systems in which clients reserve resources which then often lie idle.

Web applications are an example of a workload that could potentially benefit from serverless, due to their bursty load patterns. However, they rarely use use serverless [12, 28, 29]. One reason is that even short functions sometimes suffer long delays under serverless: in a small benchmark on AWS (described in Section 2), we found that total execution times for a simple hello world function that sleeps for 20 ms ranged from 20 to 400ms. Small response time differences can have a large negative impact in interactive applications [8, 14]; the maximum acceptable latency for a user-facing function is closer to 100ms [19].

While a well-known cause of variable latency is cold start, research is putting cold start times of a few milliseconds within reach [24, 27]. If cold start is no longer an obstacle, will serverless be ready to support web applications?

A major remaining problem for latency-sensitive tasks on serverless is the *crowding problem*: under high total load, different tenants' functions compete for resources, and thus can drive up each other's delays. Whether the outcome is acceptable depends on how the provider schedules different tenants' functions.

This paper proposes XX, a serverless scheduler that addresses the crowding problem by ensuring that latency sensitive functions aren't blocked behind background work.

Designing XX faces multiple challenges. One of them is that the scheduler needs a basis on which to compare the importances of different tenants' functions. Each individual tenant can be expected to understand which of its own functions are most in need of low latency, but how to compare these decisions across tenants? To achieve this global comparability, tenants declare a *price class* for each function invocation. The price class is the amount of money the tenant proposes to pay per unit of CPU time. The scheduler can then make scheduling decisions among different tenants' functions by comparing their price classes.

Another challenge is rapid placement of functions in a large cluster of servers. Tracking idle or pre-emptible resources is difficult when both the number of new function invocations and the amount of resources are large.

A third key challenge in designing XX is memory management. The provider faces a tension: high CPU utilization requires packing many functions onto each machine, but not so many that the machine runs out of memory. Current systems avoid memory exhaustion by requiring developers to declare the maximum amount of memory each function will use. However, memory use is difficult to predict and varies across invocations. Instead, XX charges developers based on the amount of memory actually used, and requires no bound to be set. XX thus faces the challenging proposition of blindly placing functions not knowing how much memory they will use, but still needing CPU utilization to be high.

2 Motivation

This section reviews the potential benefits of serverless for interactive workloads such as web applications, and presents evidence that the crowding problem will need to be solved in order to obtain those benefits.

2.1 Web applications are a good fit for serverless

Serverless is well suited to situations where load can vary rapidly and is hard to predict, so that long-term cloud resource reservations would lead to waste during low-load periods. Web applications often fit this profile.

For example, consider a web site with 50,000 requests per day, each request consuming 200 ms of CPU time and 128 MB of memory. The web site is bursty in the sense that it has work only about a fifth of the time. Running this web site on AWS Lambda would cost about $1.60 per month. The cheapest EC2 instance (which reserves compute resources) costs just over $3 per month. The expense-minimizing choice differs depending on load level and degree of burstyness [7, 26], but generally high load variation favors serverless.

Slowly-changing load can be handled by dynamic adjustment to long-term reserved resources (e.g. EC2 instances), but such adjustments can take multiple minutes [1] and are thus not suited to rapid load variation.

2.2 The crowding problem

Cloud providers are motivated to keep their compute hardware as busy as possible, in order to avoid waste and reduce

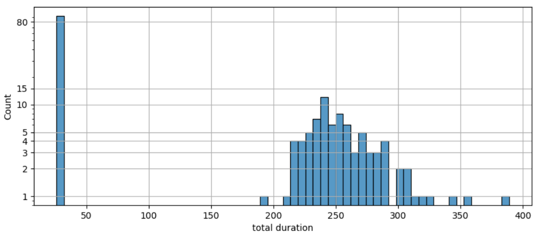

Figure 1: Distribution of end to end durations for a simple function on AWS Lambda. The y axis is log scale.

costs. In a non-reservation system like serverless, if the average load is near capacity (to minimize average waste), the peak load will be above capacity, imposing either queuing or time-sharing delays on functions.

Measurements of cold-start delays in AWS Lambda show evidence of this crowding problem. The experiment launches a test function at intervals randomly spaced between 0 and 10 minutes; the function just sleeps for 20ms and then returns. AWS Xray [2] records end-to-end latencies. The results are in Figure 1. The spike on the left side of the graph reflects warm-start executions, which use a cached container left over from a previous execution. We verify this by having each function look for some state changed in the container by the previous invocation.

The right-hand grouping in Figure 1 reflects invocations that required cold start (creation of a new container). The delays have a wide spread, between ~200 and ~400ms, suggesting that there is more going on than just creation of a new container. We are not able to look in more detail at the reasons for these varying delays in AWS Lambda, but we can examine the designs of other serverless systems to see whether the crowding problem would cause the kind of delay variation in the graph.

The OpenWhisk [21] load balancer chooses a machine to run each new function on and adds a message to a Kafka queue addressed to that machine; the machine pulls invocation messages as resources are freed up by the termination of previous functions [3]. This means that a latency sensitive function might sit in the Kafka queue while another tenant's background function completes: the crowding problem.

Knative [17] similarly queues excess invocations, although it does so via the load balancer, which is also in charge of autoscaling [18]: if the existing pods are fully loaded (with a small, bounded-size queue in front of them), requests are queued separately while the autoscaler starts up more pods. Again, latency sensitive functions potentially wait for someone else's background function completes: the crowding problem.

Hermod [16] is a recent research serverless scheduler, and shows in a simulation that late binding (as Openwhisk and Knative do) performs worse than early binding. Under high load, Hermod places the excess functions on machines anyway (used a least loaded policy) and does Processor Sharing scheduling among them. This means that all are equally slowed down, and no one function has a high delay. This still leads to the crowding problem: now rather than experiencing queuing, latency sensitive functions experience time-sharing delay. Hermod does not address what happens when the machines are out of memory.

None of these schedulers know which functions to prioritize under high load. The way all of the above schedulers avoid the crowding problem is by doing different forms of accounting concurrency: concurrency can be reserved or provisioned for specific functions, and limited for others. This is necessary to ensure that a burst in background tasks doesn't starve the latency sensitive functions. Reserving and provisioning and limiting are, however, conceptually in tension with the goal of serverless, which is to be on-demand and flexible.

3 Using price classes in XX

This section describes XX a scheduler that addresses the crowding problem using price classes and meets the §1 challenges.

3.1 Price classes

XX uses price classes to supplant the current interface, which requires developers to choose an amount of memory per function (which is then tied to a CPU fraction, e.g., 0.2 vCPUs). XX bills memory separately and by use, and the price for memory is the same across all price classes.

Price classes don't imply absolute guarantees about what resources a function receives. The price class is instead a metric to express priority to XX, which XX uses to enforce a favoring of high price class functions. To avoid the developer-side uncertainty of bidding wars, XX exposes a fixed set of price classes (similar to how AWS has different EC2 instance types).

Price classes are a metric that has a number of benefits over resource usage estimations. One is that developers are more likely to have a good sense of what price class a function should have ahead of time, because they know in what context the function will be used. Price classes also remain the same across different invocations, whereas resource needs can be heavily skewed in web applications [16, 23]. And finally, price classes more directly align the interests of the developer with those of the provider, by communicating on the level of what the provider and developer actually care about: money, and latency (as achieved by price classes in the system).

Price classes also allows the provider to provision their datacenters hardware: by looking at the historical overall amount of high price class load, they know a minimum of

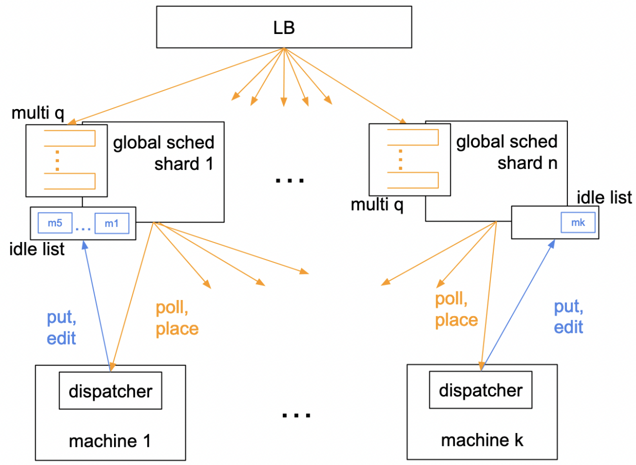

Figure 2: global scheduler shards queue and place functions (in orange), on each machine a dispatcher thread keeps track of memory utilization and writes itself to idle lists (in blue)

how much hardware they need to buy to be able to comfortably fit that load.

3.2 Interface

Developers using XX write function handlers and define triggers just like they would for existing serverless offerings. Triggers assign a price class to a function invocation based on the URL and its arguments. For instance, a simple web application might consist of a home page view that is assigned a higher price class and costs $2\mu¢$ per cpu second, a user profile page view which is assigned a middle-high price class and cost $1.5\mu¢$ per cpu second, and finally an image processing function that can be set to a low price class which costs only $0.5\mu¢$ per cpu second.

Price classes are inherited across call chains: if a high price class function calls a low price class function, that invocation with run with high price class. This inheritance is important in order to avoid priority inversion.

To avoid unexpected costs in the case of for example a DOS attack or a bug, developers also express a monthly budget that they are willing to pay. XX uses this budget as a guideline and throttles invocations or decreases quality of service in the case that usage is not within reason given the expected budget, though it does not guarantee that the budget will not be exceeded by small amounts.

3.3 XX Design

XX has as its goal to enforce the price classes attached to functions, which means it needs to prefer higher price class functions over lower ones, and preempt the latter when necessary. As shown in Figure 2, XX sits behind a load balancer, and consists of: a *distributed global scheduler*, which places new function invocations and has attached an *idle list*, a *dispatcher*, which runs on each machine and communicates with the global scheduler shards, and a *machine scheduler*, which enforces price classes on the machines.

Machine Scheduler. The machine scheduler is a preemptive priority scheduler: it preempts lower price class functions to run higher price class ones. Being unfair and starving low price class functions is desirable in XX, since image processing functions should not interrupt a page view request processing, but vice versa is expected. Within price classes the machine scheduler is first come first served. This design matches Linux' "SCHED FIFO" policy [4].

Idle list. Each global scheduler shard has an idle list, which holds machines that have a significant amount of memory available. In the shards idle list, each machine's entry is associated with the amount of memory available as well as the current amount of functions on the machine. The idle list exists because datacenters are large: polling a small number of machines cannot find something that is a rare occurrence [20]. The idle list allows the rare idle machine to make itself visible to the global scheduler. The idle list also allows the global scheduler to place high price class functions quickly, without incurring the latency overheads of doing polling to find available resources. This design is inspired by join idle queue [20], but defines idleness via memory availability rather than empty queues.

Dispatcher. The dispatcher is in charge of adding itself to a shard's idle list when memory utilization is low. The dispatcher chooses which list to add itself to using power-of-k-choices: it looks at k shards' idle lists and chooses the one with the least other machines in it. If the machine is already on shard i's idle list, but the amount of available memory has changed significantly (either by functions finishing and memory being freed or by memory utilization increasing because of new functions or memory antagonists), the dispatcher will update shard i's idle list.

The dispatcher is also in charge of managing the machine's memory. When memory pressure occurs, the dispatcher uses *price class-based swapping* to move low price class functions off the machine's memory. Having priority scheduling creates an opportunity: because the dispatcher knows that the lowest price class functions will not be run until the high price class functions have all finished, it can swap its memory out knowing it will not be needed soon. The dispatcher swaps the low price class function back in when the memory pressure is gone and the function will be run.

XXcannot bound the amount of swap space required since it doesn't ask functions for a memory bound. In the rare case that XXruns it resorts to killing low-class functions. Providers can estimate the amount of swap space required by looking at memory utilization and since the SSDs necessary for swap space are inexpensive [5] we expect that killing is rare.

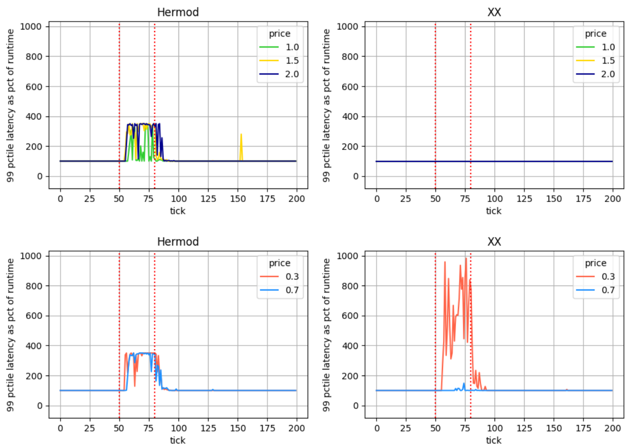

Figure 3: tail latency distribution for Hermod and XX, for high (top) and low (bottom) price class functions. At tick 50 (first red line) the load was increased, and at tick 80 (second red line) it was decreased again

Global Scheduler Shards. Global scheduler shards store the functions waiting to be placed in a multi queue, with one queue per price class. Shards choose what function to place next by looking at each function at the head of each queue, and comparing the ratio of price class to amount of time spent in the queue. This policy ensures that high price class functions don't have to wait as long as low price class functions to be chosen next, but low price class functions will get placed if they have waited for a while.

When placing the chosen function, the shard will first look in its idle list. If the list is not empty, it will choose the machine with the smallest queue length. If there are no machines in the idle list, the shard switches to power-of-k-choices: it polls k machines, and chooses the least loaded machine (by number of functions running).

4 Preliminary Results

In order to provide some evidence that XX's design meets its goals, this section answers two questions: (1) does XXavoid the crowding problem? (§4.2) and (2) is swapping a feasible approach to managing memory of functions? (§4.3)

4.1 Experimental methodology

To answer these questions, we build a simulator in Go[6] in which functions arrive in an open loop. The simulator attaches three properties to each function: runtime, price class, and memory usage. *Function runtime* is chosen by sampling from randomly generated long tailed (pareto) distribution: the length of the tail (α value) is constant, and the minimum value (x_m) is chosen from a normal distribution. This sampling reflects the fact that different functions have different expected runtimes (represented by x_m), and that actual invocation runtimes follow long tailed distributions (represented by the pareto function). *Function price class* is chosen randomly, but weighted: the simulator uses a bimodal weighting across n price class values, each assigned to a fictitious price. Because functions are randomly assigned a price class, runtime and price class are not correlated. *Function memory usage* is chosen randomly between 100MB and 10GB. Over their lifetime, functions increase their memory usage from an initial amount (always 100MB) to their total usage.

The simulator makes a few simplifying assumptions: (1) functions are compute bound, and do not block for I/O; and (2) communication and swap latencies are not simulated.

We simulate running 100 machines with 8 cores each, 4 scheduler shards, and run $k = 3$ for k-choices.

4.2 Does XX avoid the crowding problem?

To show that XX can run high-classes functions quickly even under high load, we compare XX with Hermod, a state-of-the-art research scheduler built specifically for serverles [16]. We simulate Hermod in the best configuration according to the paper: least-loaded load balancing over machines found using power-of-k-choices, combined with early binding and Processor Sharing machine-level scheduling. Because Hermod does not account for memory limits on machines, we ignore memory in this experiment. We also turn off the use of the idle list in XX, so as to be on par with Hermod in placing load.

We run an experiment that starts with a medium load setting, temporarily increase the load, and then return to the baseline load. A strong result for XX would show that it is able to maintain low latency for high price class functions, even under the high load. Figure 3 shows the results. We can see that XX is indeed able to maintain low latencies for the high price class functions, at the cost of increasing the latencies for low price class functions. Hermod spreads the performance degradation across all the different functions equally.

4.3 Is swapping memory of functions feasible?

To answer this question, we count the amount of swap memory that XX uses and which functions XX swaps. We configure the simulator to run XX with limited memory (32GB of RAM per machine). A good result would show: a small spread of memory utilization, that machines start swapping only once memory utilization is high, that the amount of swapping being done is equally spread across machines, and that high-class functions are not impacted by swapping. Figure 4 shows the results. We can see XX swaps only lower price class functions' memory, and that the amount of memory swapped is fairly evenly distributed between all the machines. We can also conclude that with a 500GB SSD, a

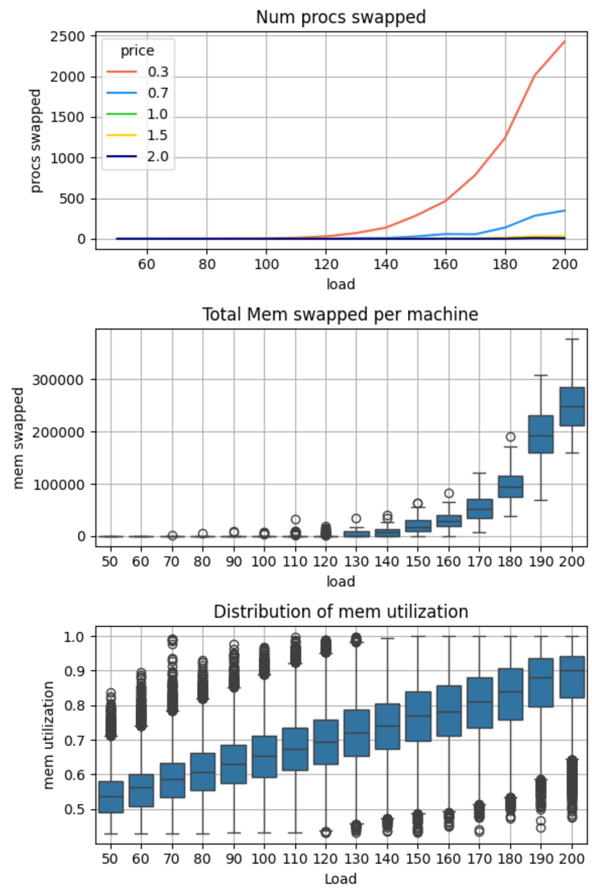

Figure 4: XX's swapping behavior. The amount of memory is in MB

provider would be able to avoid killing while running the datacenter at an average memory utilization of ~90%, at the cost of ~ $30 per machine for swap space [5].

5 Related Work

There is a large literature on scheduling for data centers but none address the crowding problem. Systems like Sparrow[22], Hermod[16], or Kairos[11] improve performance of scheduling in the distributed setting by trying out and using different scheduling policies. Unlike XX, they treat all functions equally.

Like XX, many projects tailor their approach to serverless. Some systems generate information about functions themselves to help placement decisions; for instance ALPS[13], which observes and learns the behaviors of existing functions and then makes scheduling decisions based on those; or Morpheus[15], which learns SLOs from historical runs, and then translates these to recurring reservations. XX instead obtains the price classes directly from the developers and bases its decisions solely on price classes.

Other papers have taken the same approach of obtaining information from the developers. Sequoia[25], for instance, creates a metric of QOS for serverless functions. Unlike XX however, Sequoia does not implement a new scheduler, but is itself a serverless function that manages the invocation sequence of developer's function chains by interposing on the triggers and choosing what to invoke when. This design does not support multi-tenancy.

Allocation Priority Policies (APP)[10] provides a declarative language to express policies. The APP language allows developers to specify custom load balancing decisions, and the scheduler uses the developers' specification to define a mapping of function invocations to workers. XX, on the other hand, does not ask developers to set the load balancing policy, but rather has developers give XX the information it needs to do the load balancing itself.

Serverless orchestration systems like Dirigent [9] are orthogonal to XX: their approaches can be combined to further reduce the latency overheads that functions face.

[fk: scheduling based on money, spot instances]

6 Conclusion

Serverless is in principle a good match for Web applications, allowing developers to pay for the resources needed as load varies without having to reserve servers. Because of the crowding problem, however, existing serverless designs result in latencies that are not tolerable for user-facing functions, even in systems that have fast function start time. We argue for new design based on *price classes* that can keep the latency of high price class functions latencies stable even under high load, making it feasible to run Web applications as functions.

References

[1] URL: https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-default-instance-warmup.html.

[2] URL: https://aws.amazon.com/xray/.

[3] URL: https://github.com/apache/openwhisk/blob/master/docs/about.md.

[4] URL: https://man7.org/linux/man-pages/man7/sched.7.html.

[5] URL: https://www.bestbuy.com/site/pny-cs900-500gb-internal-ssd-sata/6385542.p?skuId=6385542.

[6] URL: https://go.dev/.

[7] Álvaro Alda Rodríguez et al. *Economics of 'Serverless'*. URL: https://www.bbva.com/en/innovation/economics-of-serverless/.

[8] Jake Brutlag. *Speed Matters*. URL: https://research.google/blog/speed-matters/.

[9] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. "Dirigent: Lightweight Serverless Orchestration". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin,

TX, USA: Association for Computing Machinery, 2024, pages 369–384.

[10] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. "Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation". In: *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings*. Dubai, United Arab Emirates: Springer-Verlag, 2020, pages 416–430.

[11] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. "Kairos: Preemptive Data Center Scheduling Without Runtime Estimates". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, pages 135–148.

[12] Jesse Duffield. *My notes from deciding against AWS Lambda*. URL: https://jesseduffield.com/Notes-On-Lambda/.

[13] Yuqi Fu, Ruizhe Shi, Haoliang Wang, Songqing Chen, and Yue Cheng. "ALPS: An Adaptive Learning, Priority OS Scheduler for Serverless Functions". In: *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, July 2024, pages 19–36.

[14] Gigaspaces. *Amazon Found Every 100ms of Latency Cost them 1 Percent in Sales*. URL: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales.

[15] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pages 117–134.

[16] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. "Hermod: principled and practical scheduling for serverless functions". In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22. San Francisco, California: Association for Computing Machinery, 2022, pages 289–305.

[17] *Knative*. URL: https://knative.dev/.

[18] Stavros Kontopoulos. *Demystifying Activator on the data path*. URL: https://knative.dev/blog/articles/demystifying-activator-on-path/.

[19] Greg Linden. *Marissa Mayer at Web 2.0*. URL: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[20] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. "Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services". In: *Performance Evaluation* 68.11 (2011). Special Issue: Performance 2011, pages 1056–1071.

[21] *OpenWhisk*. URL: https://openwhisk.apache.org.

[22] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: distributed, low latency scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pages 69–84.

[23] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pages 205–218.

[24] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. "Unifying serverless and microservice workloads with SigmaOS". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pages 385–402.

[25] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. "Sequoia: enabling quality-of-service in serverless computing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pages 311–327.

[26] Andy Warzon. *AWS Lambda Pricing in Context - A Comparison to EC2*. URL: https://www.trek10.com/blog/lambda-cost.

[27] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. "No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pages 497–517.

[28] *Why or why not use AWS Lambda instead of a web framework for your REST APIs?* URL: https://www.reddit.com/r/Python/comments/1092py3/why_or_why_not_use_aws_lambda_instead_of_a_web/.

[29] *Without saying "it's scalable", please convince me that a serverless architecture is worth it*. URL: https://www.reddit.com/r/aws/comments/yxyyk3/without_saying_its_scalable_please_convince_me/.