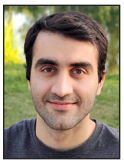
Characterization and Optimization of the Serverless Workload at a Large Cloud Provider

MOHAMMAD SHAHRAD, RODRIGO FONSECA, ÍÑIGO GOIRI, GOHAR CHAUDHRY, AND RICARDO BIANCHINI



Mohammad Shahrad is a Computer Science Lecturer at Princeton University and an incoming Assistant Professor of Electrical and Computer Engineering at the University of British Columbia. He received his PhD from Princeton University and his BSc from Sharif University of Technology. Dr. Shahrad's research aims to improve the efficiency of cloud computing systems through better resource management and enhanced system/architecture integration. mshahrad@ece.ubc.ca

Rodrigo Fonseca is a Principal Researcher at Microsoft Research and an Associate Professor at Brown University's CS Department. He is broadly interested in distributed systems, networking, and operating systems, and his current research involves ways to make cloud computing easier and more applicable for users and more efficient for providers. He holds a PhD from UC Berkeley, a MSc and BSc from Federal University of Minas Gerais, and is the recipient of an NSF CAREER award, an NSDI Test of Time Award, and a 2015 SOSP Best Paper Award. rfonseca@cs.brown.edu

Íñigo Goiri is a Research Software Developer at Microsoft Research. His current research focuses on the efficiency of large scale distributed systems. He holds a PhD from the University Politecnica de Catalunya (UPC). inigog@microsoft.com

Function as a Service (FaaS) has gained tremendous popularity as a way to deploy computations to serverless back ends in the cloud. We performed the first characterization of an entire production FaaS environment (Azure Functions) [1]. Our characterization revealed many unique aspects of serverless workloads compared to traditional cloud applications. Using this deep understanding, we designed a new dynamic resource management policy to improve the performance and reduce the memory footprint of serverless workloads. This new policy is now deployed in production, and our characterization data traces are publicly released for researchers.

Serverless characterization studies before our work can be classified into two main categories: those probing public serverless offerings externally and those looking at ways developers use FaaS offerings by investigating public repositories. These two classes of studies provide valuable information; external probing allows comparing the performance and availability of various FaaS providers using a set of benchmarks, and looking at public FaaS repositories allows finding popular programming trends. However, neither of them can offer insights on the aggregate workload seen by a provider. Only when the entire workload is known can one answer questions such as *"How often do functions get invoked?" "How long do functions execute for?"* or *"How much memory do serverless functions require?"* Answers to such basic questions have major implications for designing various components of serverless systems—from schedulers to virtualization environments to underlying hardware architectures.

We conducted the first detailed characterization of an entire production FaaS workload at a large cloud provider. To do so, we collected data on all function invocations across Microsoft Azure's entire infrastructure between July 15 and July 28, 2019. We invite the reader to read our recent USENIX ATC paper for methodology details and full characterization data [1]. The sanitized traces from a subset of our characterization data are also available publicly at https://github.com/Azure/AzurePublicDataset. In what follows, we summarize some of our characterization insights.

Composition of Applications

In Azure Functions, functions are grouped into applications. The application concept helps organize the software, and the application is the unit of scheduling and resource allocation. As shown in Figure 1, 54% of the applications have only one function, and 95% of the applications have at most 10 functions. The other two curves show the fraction of invocations and functions corresponding to applications with up to a certain number of functions. For example, we see that 50% of the invocations come from applications with at most three functions, and 50% of the functions are part of applications with at most six functions.

Composition of Triggers

Functions can be invoked in response to several event types, called triggers. Figure 2 shows the fraction of all functions and invocations per type of trigger. HTTP is the most popular in both dimensions. Event triggers correspond to only 2.2% of the functions, but they correspond to 24.7% of the invocations due to their automated, and very high, invocation rates. Queue triggers also have proportionally more invocations than functions (33.5% vs. 15.2%).

Gohar Irfan Chaudhry is a Research Software Engineer at Microsoft Research. He is part of the Systems Research Group and is currently working on improving efficiency of serverless infrastructure. Gohar.Irfan@microsoft.com

Ricardo Bianchini received his PhD in computer science from the University of Rochester. He is currently a Distinguished Engineer at Microsoft, where he leads efforts to improve the efficiency of the company's online services and datacenters. He also manages the Systems Research Group at Microsoft Research in Redmond. His main research interests include cloud computing, datacenter efficiency, and leveraging machine learning to improve systems. He is an ACM Fellow and an IEEE Fellow. ricardob@microsoft.com

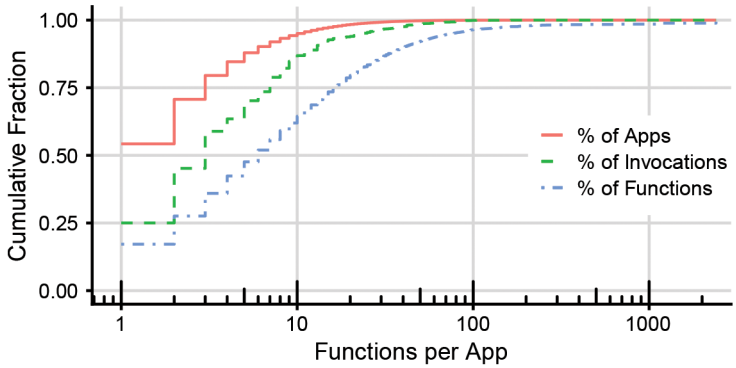


Figure 1: Distribution of function counts per application

The opposite happens with timer triggers. There are many functions triggered by timers (15.6%), but they correspond to only 2% of the invocations, due to their relatively low firing rate: 95% of the timer-triggered functions in our data set were triggered at most once per minute, on average.

Invocation Patterns

We observed that applications are invoked very differently. The number of invocations per day varies by over eight orders of magnitude for different applications. Another observation with strong implications for resource allocation is that the vast majority of applications and functions are invoked, on average, very infrequently: on average, 45% of the applications are invoked once per hour or less frequently, and 81% of the applications are invoked once per minute or less. The other side of this skewness was revealed to us by finding that the top 18.6% most popular applications represent 99.6% of all function invocations. Thus, keeping the applications that receive infrequent invocations resident in memory at all times is expensive.

Function Execution Times

An advantage of the serverless model is that users pay only for their execution time. Figure 3 shows the distribution of average, minimum, and maximum execution times of all function executions on July 15, 2019, which is similar to other days. We observed that 50% of the functions execute for less than 1 sec on average, and 96% of functions take less than 60 sec on average. These short executions in FaaS are unlike virtual machines (VMs). For example, a prior study reported that 63% of all VM allocations on Azure last longer than 15 minutes [2].

Trigger	%Functions	%Invocations
HTTP	55.0	35.9
Queue	15.2	33.5
Event	2.2	24.7
Orchestration	6.9	2.3
Timer	15.6	2.0
Storage	2.8	0.7
Others	2.2	1.0

Figure 2: Functions and invocations per trigger type

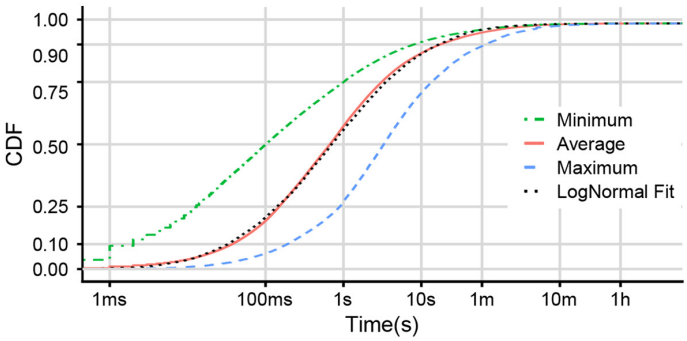


Figure 3: Distribution of function execution times

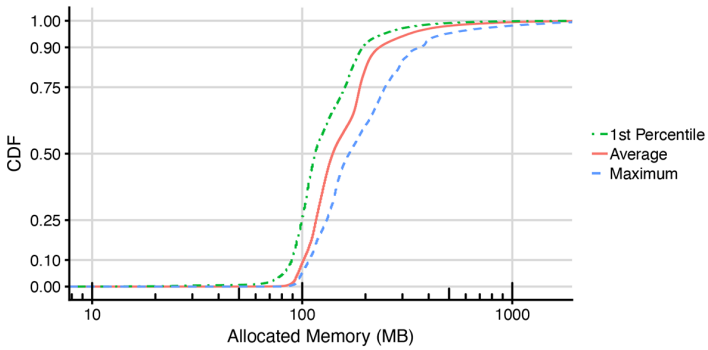


Figure 4: Distribution of allocated memory per application

FaaS applications experience cold starts. A cold start invocation occurs when a function is triggered, but its application is not yet loaded in memory. When this happens, the platform instantiates a worker for the application, loads all the required runtime and libraries, and calls the function. While Figure 3 does not include cold starts, we observed that the execution times from our characterization are the same order of magnitude as the cold start times reported for major providers [3]. Therefore, optimizing cold starts becomes extremely important for the overall performance of a FaaS offering. This can be done either by reducing the cold start latency [4, 5] or by eliminating cold starts. We took the second approach in designing our policy, which we describe later in the article.

Memory Usage

The memory demand of applications on the same day (July 15, 2019) is shown in Figure 4. Looking at the distribution of the maximum allocated memory, 90% of the applications never consume more than 400 MB, and 50% of the applications allocate at most 170 MB. We found no strong correlation between invocation frequency and memory allocation or between memory allocation and function execution times.

Designing a New Adaptive Resource Management Policy

One of our primary goals in understanding workload characteristics was to design better resource management policies. This is because the state-of-the-art in serverless resource management was too simplistic, where each application was kept in memory after function execution for a fixed amount of time. This keep-alive window is 10 minutes for AWS Lambda and IBM Cloud Functions, and was 20 minutes for Azure Functions. Such a policy is too rigid for the wide range of serverless applications. Developers usually circumvent this by creating regular artificial invocations to make sure their applications remain warm in memory. A smart dynamic policy can eliminate such a burden. Additionally, adapting to applications' invocation patterns would mean resources are not kept unused just to keep function images warm without executing them.

There are a few challenges in designing such a policy. As we showed earlier in this article, invocation frequency and pattern vary substantially across applications. A one-size-fits-all fixed policy is certain to be a poor choice for many applications. Adapting the policy to each application means tracking each application individually, and thus the cost to track the information for each application should be small. Finally, since function executions can be very short (i.e., more than 50% of executions take less than one second), running the policy and updating its state need to be fast. This is especially critical considering providers charge users only during their function execution times (e.g., based on CPU, memory). For instance, we cannot take 100 ms to update a policy prediction model for each 10 ms-long execution.

We propose a *hybrid histogram policy* that addresses all the above challenges. It identifies each application's invocation pattern, removes/unloads the application right after each function execution ends, reloads/pre-warms the application right before a potential next invocation, and keeps it alive for a period. The policy does so by capturing the history and predicting next idle times (ITs), defined as the time between the end of a function's execution and its next invocation. Three main components of the *hybrid histogram policy* include: (1) a range-limited histogram for capturing each application's ITs; (2) a standard keep-alive approach for when the histogram is not representative, i.e., there are too few ITs or the IT behavior is changing (again, note that this differs from a fixed keep-alive policy); and (3) a time-series forecast component for when the histogram does not capture most ITs.

Compared to fixed keep-alive policies, *hybrid histogram policies* are closer to optimal. As seen in Figure 5, hybrid policies deliver a significant reduction of unused memory time, while considerably improving the cold start percentage for applications. For instance, a hybrid policy with a four-hour histogram can deliver a 2.5× lower 3rd-quartile cold start percentage and 1.5× less memory time wastage compared to a fixed 10-minute keep-alive policy. Note that there is a tradeoff between cold starts and wasted memory time for both policy families, but hybrid substantially dominates all fixed policies.

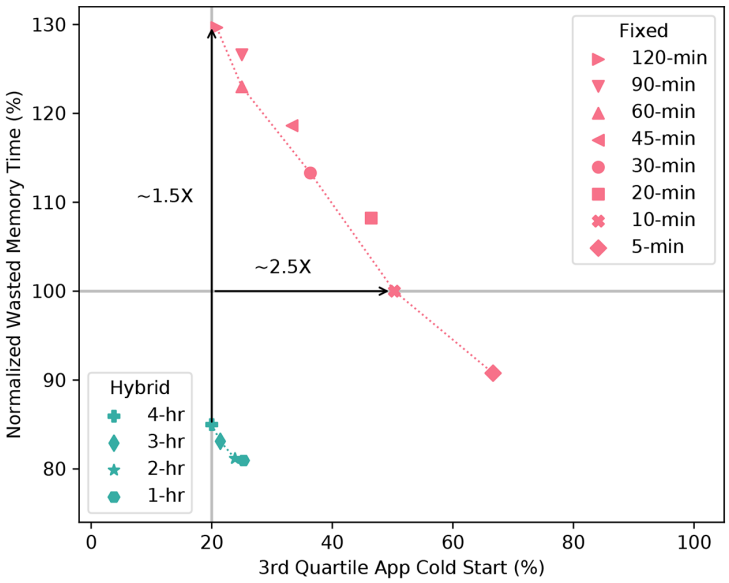


Figure 5: Tradeoff between cold starts and wasted memory time for the fixed keep-alive policy and our hybrid policy

The range-limited histogram at the core of the *hybrid histogram policy* is a lightweight data structure. We use it with a minute-long resolution, which means capturing a four-hour histogram requires an array of length 240. The other two components of the *hybrid histogram policy* complement it to boost performance while maintaining low overhead. Here, we describe some of our design choices and their implications for the policy:

- **Pre-warming to curtail keep-alive values while maintaining low cold starts**: One can eliminate cold starts by just setting the right keep-alive values, but this approach is too costly. Pre-warming allowed us to reduce memory wastage by about 34% compared to using just keep-alives, with a minor cold start increase.
- **Ignoring outlier ITs to deflate keep-alive values**: To exclude outliers of the IT distribution captured by the histogram, we use the 5th- and 99th-percentiles as head and tail cutoffs, respectively. This approach avoided the inflation of keep-alive values and resulted in a ~15% reduction in memory time wastage with a negligible impact of cold start performance of applications.
- **Checking the histogram representativeness to not use it prematurely**: The histogram might not be representative of an application's behavior when it has not observed enough ITs for the application or when the application is transitioning to a different IT regime. We decide whether a histogram is representative by computing the coefficient of variation (CV) of its bin counts and comparing it to a threshold (CV=2). This simple approach improved the 3rd-quartile application cold starts by nearly 49% with only a 3% increase in memory time wastage.
- **Using time-series forecast to eliminate cold starts of infrequent applications**: Using time-series forecast for infrequent applications reduced the percentage of applications that experience 100% cold starts by about 50%, i.e., from 10.5% to 5.2% of all applications. A significant portion of these applications have only one invocation during the entire week, and no predictive model can help them. Excluding these applications, the same reduction becomes 75%, i.e., from 6.9% to 1.7% of all applications.

We implemented our policy in Apache OpenWhisk [6], which is the open-source FaaS platform powering IBM's Cloud Functions. We refer the reader to our paper for implementation details [1]. We ran two experiments with 68 randomly selected mid-range popularity applications from our workload on our 19-VM OpenWhisk deployment: one experiment with the default 10-minute fixed keep-alive policy of OpenWhisk and another with our hybrid policy and a four-hour histogram range. Each experiment ran for eight hours with a total of 12,383 function invocations. We used FaaSProfiler [7] to automate trace replay and result analysis.

Figure 6 compares the cold start distribution of keep-alive and hybrid policies from the simulations (left) and the OpenWhisk prototype (right). As seen, the significant cold start reductions follow similar trends. On average and across the 18 invoker VMs, the hybrid policy reduced memory consumption of worker containers by 15.6%, which was also consistent with our simulation results. Moreover, hybrid policy reduced the average and 99-percentile function execution time 32.5% and 82.4%, respectively, due to a secondary effect in OpenWhisk, where the language runtime bootstrap time is eliminated for warm containers. The price for all of these is an additional 835.7μs latency on average, which is negligible compared to the existing latency of OpenWhisk components: the (in-memory) language runtime initiation takes O(10 ms) and the container initiation takes O(100 ms) for cold containers [7].

After getting promising results from simulations as well as the prototype implementation, we implemented our policy in Azure Functions for HTTP-triggered applications. Its main elements have rolled out to production. We used asynchronous updates from the workers to the Azure Functions controller to populate histograms. We keep the histogram in memory and do hourly backups to the database. We start a new histogram per day in the database so that we can track changes in an application's invocation pattern and remove histograms older than two weeks. When an application changes state from executing to idle, we use the aggregated histogram to compute its pre-warm interval and schedule an event for that time (minus 90 seconds). Pre-warming loads function dependencies and performs JIT where applicable. Each worker maintains the keep-alive duration separately, depending on how long it has been idle. We make all policy decisions asynchronously, off the critical path, to minimize the latency impact on the invocation.

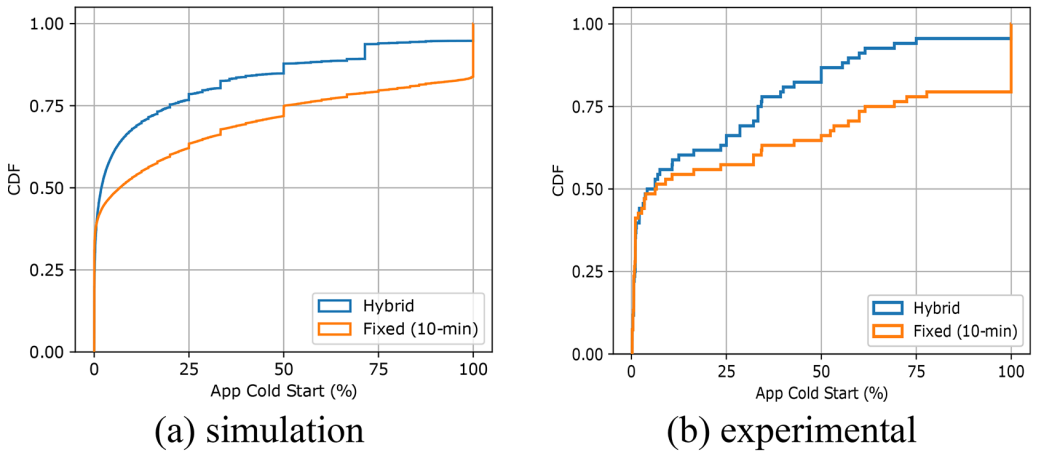


Figure 6: Cold start behavior of fixed keep-alive and hybrid policies in (a) simulation results and (b) experimental results from our OpenWhisk implementation

Conclusion

We characterized the entire production FaaS workload of Azure Functions, which unearthed several key observations for cold start and resource management. Based on them, we proposed a practical policy for reducing the number of cold starts at a low resource cost. The main elements of this policy have rolled out to production. We also released sanitized traces from a subset of our characterization data that is first of its kind. These traces will help researchers design future serverless systems based on realistic workloads and enable new research angles.

References

[1] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pp. 205–218: https://www.usenix.org/system/files/atc20-shahrad.pdf.

[2] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 153–167.

[3] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pp. 133–145: https://www.usenix.org/system/files/conference/atc18/atc18-wang-liang.pdf.

[4] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pp. 55–70: https://www.usenix.org/system/files/conference/atc18/atc18-oakes.pdf.

[5] K. Wang, R. Ho, and P. Wu, "Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment," in *Proceedings of the 14th EuroSys Conference 2019*, pp. 1–16.

[6] Apache OpenWhisk, Open Source Serverless Cloud Platform: https://openwhisk.apache.org/.

[7] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*, pp. 1063–1075.