# Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider

Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini *

Microsoft Azure and Microsoft Research

## Abstract

Function as a Service (FaaS) has been gaining popularity as a way to deploy computations to serverless backends in the cloud. This paradigm shifts the complexity of allocating and provisioning resources to the cloud provider, which has to provide the illusion of always-available resources (*i.e.*, fast function invocations without cold starts) at the lowest possible resource cost. Doing so requires the provider to deeply understand the characteristics of the FaaS workload. Unfortunately, there has been little to no public information on these characteristics. Thus, in this paper, we first characterize the entire production FaaS workload of Azure Functions. We show for example that most functions are invoked very infrequently, but there is an 8-order-of-magnitude range of invocation frequencies. Using observations from our characterization, we then propose a practical resource management policy that significantly reduces the number of function cold starts, while spending fewer resources than state-of-the-practice policies.

## 1 Introduction

Function as a Service (FaaS) is a software paradigm that is becoming increasingly popular. Multiple cloud providers offer FaaS [5, 17, 21, 28] as the interface to usage-driven, stateless (serverless) backend services. FaaS offers an intuitive, event-based interface for developing cloud-based applications. In contrast with the traditional cloud interface, in FaaS, users do not explicitly provision or configure virtual machines (VMs) or containers. FaaS users do not pay for resources they do not use either. Instead, users simply upload the code of their functions to the cloud; functions get executed when "triggered" or "invoked" by events, such as the receipt of a message (*e.g.*, an HTTP request) or a timer going off. The provider is then responsible for provisioning the needed resources (*e.g.*, a container in which to execute each function), providing high function performance, and billing users just for their actual function executions (*e.g.*, in increments of 100 milliseconds).

Obviously, providers seek to achieve high function performance at the lowest possible resource cost. There are three main aspects to how fast functions can execute and the resources they consume. First, function execution requires having the needed code (*e.g.*, user code, language runtime libraries) in memory. A function can be started quickly when the code is already in memory (warm start) and does not have to be brought in from persistent storage (cold start). Second, keeping the resources required by all functions in memory at all times may be prohibitively expensive for the provider, especially if function executions are short and infrequent. Ideally, the provider wants to give the illusion that all functions are always warm, while spending resources as if they were always cold. Third, functions may have widely varying resource needs and invocation frequencies from multiple triggers. These characteristics severely complicate any attempts to predict invocations for reducing resource usage. For example, the wide range of invocation frequencies suggests that keeping resources in memory may work well for some functions but not others. With respect to triggers, HTTP triggers may produce invocations at irregular intervals that are difficult to predict, whereas timers are regular.

These observations make it clear that providing high function performance at low cost requires a deep understanding of the characteristics of the FaaS workload. Unfortunately, there has been no public information on the characteristics of production workloads. Prior work [3, 15, 24, 25, 27, 44] has focused on either (1) running benchmark functions to assess performance and/or reverse-engineer how providers manage resources; or (2) implementing prototype systems to run benchmark functions. In contrast, what is needed is a comprehensive characterization of the users' *real* FaaS workloads on a *production* platform from the provider's perspective.

**Characterizing production workloads.** To fill this gap, in this paper, we first characterize the entire production FaaS workload of Azure Functions [28]. We characterize the real functions and their trigger types, invocation frequencies and patterns, and resource needs. The characterization produces many interesting observations. For example, it shows that most functions are invoked very infrequently, but the most popular functions are invoked 8 orders of magnitude more frequently than the least popular ones. It also shows that functions exhibit a variety of triggers, producing invocation patterns that are often difficult to predict. In terms of resource needs, the characterization shows a 4x range of function memory usage and that 50% of functions run in less than 1 second.

Researchers can use the distributions of the workload characteristics we study to create realistic traces for their work.

*Shahrad is affiliated with Princeton University, but was at MSR during this work. Laureano and Tresness are now with Facebook and D. E. Shaw.

Alternatively, they can use the sanitized *production traces* we are making available with this paper [31].

**Managing cold-start invocations.** Using observations from our characterization, we also propose a practical resource management policy for reducing the number of cold start executions while consuming no more resources than the large cloud providers' current policies. Specifically, AWS and Azure use a fixed "keep-alive" policy that retains the resources in memory for 10 and 20 minutes after a function execution, respectively [39, 40]. Though this policy is simple and practical, it disregards the functions' actual invocation frequency and patterns, and thus behaves poorly and wastes resources.

In contrast, our policy (1) uses a different keep-alive value for each user's workload, according to its actual invocation frequency and pattern; and (2) enables the provider in many cases to pre-warm a function execution just before its invocation happens (making it a warm start). Our policy leverages a small histogram that keeps track of the recent function inter-invocation times. For workloads that exhibit clear invocation patterns, the histogram makes clear how much keep-alive is beneficial and when the pre-warming should take place. For workloads that do not, our policy reverts back to the fixed keep-alive policy. As the histogram must be small, for any workloads that cannot be captured by the histogram but exhibit predictable invocation patterns, our policy uses time-series analysis to predict when to pre-warm.

We implement our policy in simulation and for the Apache OpenWhisk [34] FaaS platform, both driven with real workload traces. Our simulation results show that the policy significantly reduces the number of function cold starts, while spending fewer resources than the fixed keep-alive policy. Our experimental results show that the policy can be easily implemented in real systems with minimal overheads. In fact, we describe our recent production implementation in Azure Functions in the end of the paper.

**Contributions.** In summary, our main contributions are:

- A detailed characterization of the entire production FaaS workload at a large cloud provider;
- A new policy for reducing the number of cold start function executions at a low resource provisioning cost;
- Extensive simulation and experimental results based on real traces showing the benefits of the policy;
- An overview of our implementation in Azure Functions;
- A large sanitized dataset containing production FaaS traces.

## 2 Background

**Abstraction.** In FaaS, the user uploads code to the cloud, and the provider enables a handle (*e.g.*, a URL) for the code to be run. The choices of which resources to allocate, when to allocate them, and for how long to retain them, still have to be made, but they are shifted to the cloud provider.

**Triggers.** Functions can be invoked in response to several event types, called triggers [6, 29]. For clarity, in this paper we group Azure's many triggers into 7 classes: HTTP, Event, Queue, Timer, Orchestration, Storage, and others. Event triggers include Azure Event Hub and Azure Event Grid, and are used for discrete or serial events, with individual or batch processing. Queue-triggered functions respond to message insertion in a number of message queueing solutions, such as Azure Service Bus and Kafka. Timer triggers are similar to cron jobs, and cause function invocations at pre-determined, regular intervals. We grouped all triggers related to Azure Durable Functions [30] as Orchestration. One can use these triggers to create native, complex function chaining and orchestration. Finally, we grouped database and filesystem triggers as Storage. These fire in response to changes in the underlying data, and include Azure Blob Storage and Redis.

**Applications.** In Azure Functions, functions are logically grouped in applications, *i.e.* an application may encompass multiple functions. The application concept helps organize the software and in packaging. *The application, not the function, is the unit of scheduling and resource allocation*.

**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" [1] for the application, loads all the required runtime and libraries, and calls the function. This process can take a long time relative to the function execution [44]. There are strategies to reduce the time taken by each cold start, such as keeping pre-allocated VMs or containers, instantiated virtual network interfaces [32], or pre-loaded runtimes that can be specialized on-demand [18]. In this paper, we focus on the complementary and orthogonal goal of reducing the number of cold starts.

**Concurrency and elasticity.** A running instance of an application can respond to a configurable number of concurrent invocations of its functions. The number depends on the nature of the function, and its resource needs. Cold starts can also happen if there is a spike in the load to an application, and new instances have to be allocated quickly. Given full-server instances and our real FaaS workload, only a tiny percentage ($<1\%$) of applications would experience this type of cold start. For this reason, we do not consider it in this paper.

**Cold start management policy.** A key aspect of FaaS is the trade-off between reducing cold starts by keeping instances warm, and the resources (*e.g.*, VMs, memory) they need.

Most FaaS providers use a fixed keep-alive policy for all applications, where application instances are kept loaded in memory for a fixed amount of time after a function execution [39, 40]. This is also the case for most open-source implementations (*e.g.*, OpenWhisk uses a 10-minute period).

This policy is simple to implement and maintain, but does not consider the wide variety of application behaviors our characterization unearths. Thus, it can have many cold starts while wasting resources for many applications. Moreover, it is easy to identify by external users, who sometimes invoke their applications frequently enough (perhaps with dummy

[1] In some systems, a worker is a container, but in others it can be a VM.

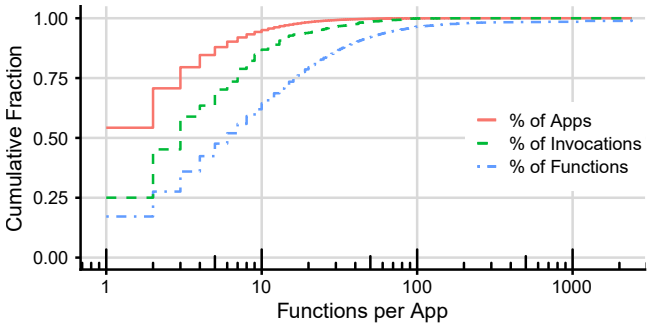



Figure 1: Distribution of the number of functions per app.

invocations) to keep them warm. This practice amplifies the resource waste issue. In this paper, we design a better policy.

## 3 FaaS Workloads

We characterize the FaaS workloads seen by Azure Functions, focusing on characteristics that are intrinsic to the applications and functions (*e.g.*, their arrival pattern), and not on the characteristics that relate to the underlying platform (*e.g.*, where functions are scheduled). Throughout the characterization, we highlight interesting observations and their implications for cold starts and resource management.

### 3.1 Data Collection

We collected data on all function invocations across Azure's entire infrastructure between July 15th and July 28th, 2019. We collected four related data sets:

1. Invocation counts: per function, in 1-minute bins;
2. Trigger per function;
3. Execution time per function: average, minimum, maximum, and count of samples, for each 30-second interval, recorded per worker; and
4. Memory usage per application: sampled every 5 seconds by the runtime and averaged, for each worker, each minute. Average, minimum, maximum, and count of samples, for allocated and resident memory.

With this paper, we are releasing a subset of our traces at https://github.com/Azure/AzurePublicDataset.

**Limitations.** Given the extreme scale of Azure Functions, the invocation counts are binned in 1-minute intervals, *i.e.* our dataset does not allow the precise reconstruction of inter-arrival times that are smaller than one minute. For this paper, this granularity is sufficient.

For the execution time, we also do not have the complete time distribution across all invocations. However, from the many samples of average time, and corresponding counts, we keep a set of weighted percentiles, where the weight of an entry is the number of samples. For example, if we see an average time of 100ms over 45 samples, the resulting percentiles are equivalent to those computed over a distribution where 100ms are replicated 45 times. The quality of the approximation to the true distribution depends on the number of samples in each bin, the smaller the better. We similarly obtain weighted percentiles for memory usage.

| 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.

For confidentiality reasons, we cannot disclose some absolute numbers, such as total number of functions and invocations. Nevertheless, our characterization is useful for understanding a full FaaS workload, and for researchers and practitioners to generate realistic FaaS workloads.

### 3.2 Functions, Applications, and Triggers

**Functions and applications.** Figure 1 shows the CDF of the number of functions per application (top curve). We observe that 54% of the applications only have one function, and 95% of the applications have at most 10 functions. About 0.04% of the applications have more than 100 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 3 functions, and 50% of the functions are part of applications with at most 6 functions. Though we found a weak positive correlation between the number of functions in an application and the median number of invocations of those applications, the number of functions in an application is not a useful signal in resource management.

We took a closer look at the 10 applications with the most functions. Only 4 had more than 1k functions: these, and 3 others, had a pattern of auto-generated function names triggered by timers or HTTP, which suggests that they were being used for large automated testing. Of the remaining 3 applications, two were using Azure Durable Functions for orchestrating multiple functions, and one seems to be an API application, where each function corresponds to one route in a large Web or REST application. We plan to do a broader and more comprehensive study of application patterns in future work.

**Triggers and applications.** Figure 2 shows the fraction of all functions, and all invocations, per type of trigger. HTTP is the most popular in both dimensions. Event triggers correspond to only 2.2% of the functions, but 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%). 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 the relatively low rate they fire in: 95% of the timer-triggered functions in our dataset were triggered at most once per minute, on average.

| Trigger Type | % Apps |
|---|---|
| HTTP (H) | 64.07 |
| Timer (T) | 29.15 |
| Queue (Q) | 23.70 |
| Storage (S) | 6.83 |
| Event (E) | 5.79 |
| Orchestration (O) | 3.09 |
| Others (o) | 6.28 |

(a) Apps with ≥ 1 of each trigger.

| Trigger Types | Fraction of Apps (%) | Cum. Frac. (%) |
|---|---|---|
| H | 43.27 | 43.27 |
| T | 13.36 | 56.63 |
| Q | 9.47 | 66.10 |
| HT | 4.59 | 70.69 |
| HQ | 4.22 | 74.92 |
| E | 3.01 | 77.92 |
| S | 2.80 | 80.73 |
| TQ | 2.57 | 83.30 |
| HTQ | 2.48 | 85.78 |
| Ho | 1.69 | 87.48 |
| HS | 1.05 | 88.53 |
| HO | 1.03 | 89.56 |

(b) Popular trigger combinations.

Figure 3: Trigger types in applications.

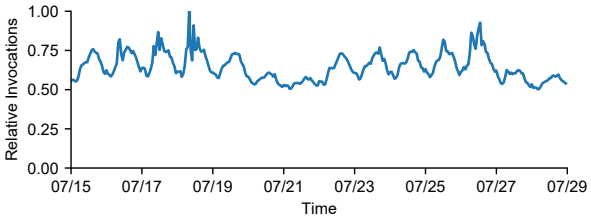


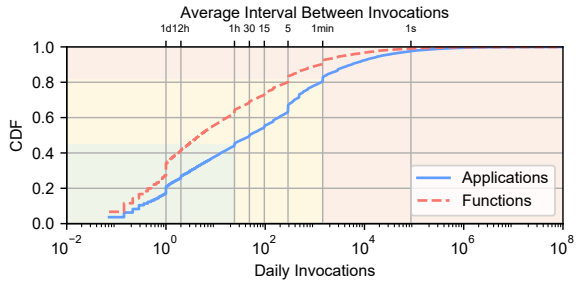

Figure 4: Invocations per hour, normalized to the peak.

Figure 3 shows how applications combine functions with different trigger types. In Figure 3(a), we show the applications with at least one trigger of the given type. We find that 64% of the applications have at least one HTTP trigger, and 29% of the applications have at least one timer trigger. As applications can have multiple triggers, the fractions sum to more than 100%. In Figure 3(b), we partition the applications by their combinations of triggers. 43% of the applications have *only* HTTP triggers, and 13% of the apps have *only* timer triggers. Combining the two tables, we find that 15.8% of the applications have timers and at least one other trigger type. For predicting invocations, as we discuss later, while timers are very predictable, 86% of the applications have either no timers or timers combined with other triggers.

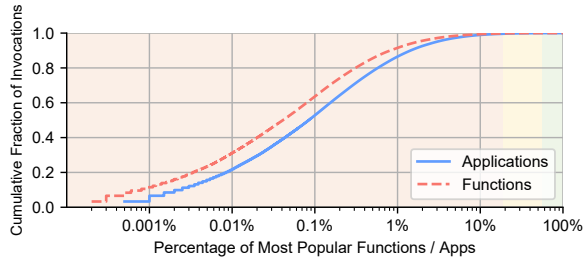
## 3.3 Invocation Patterns

We now look at dynamic function and application invocations. Figure 4 shows the volume of invocations per hour, across the entire platform, relative to the peak hourly load on July 18$^{th}$. There are clear diurnal and weekly patterns (July 20$^{th}$, 21$^{st}$, 27$^{th}$, and 28$^{th}$ are weekend days), and a constant baseline of roughly 50% of the invocations that does not show variation. Though we did not investigate this specifically, there can be several causes, *e.g.* a combination of human and machine-generated traffic, plain high-volume applications, or the overlapping of callers in different time zones.

Figure 5(a) shows the CDF of the average number of invocations per day, for a representative sample of both functions




(a) CDF of daily invocations per function and application, and the corresponding *average* interval between invocations. Shaded regions show applications invoked on average at most once per hour (green, 45% of apps) and at most once per minute (yellow, 81% of apps).




(b) Fraction of total function invocations by the fraction of the most popular functions and applications. Same colors as in Figure 5(a).

Figure 5: Invocations per application and per function for a representative sample of the dataset.

and applications. The invocations for an application are the sum over all its functions. First, we see that the number of invocations per day varies by over 8 orders of magnitude for functions and applications, making the resources the provider has to dedicate to each application also highly variable.

The second observation with strong implications for resource allocation is that the vast majority of applications and functions are invoked, on average, very infrequently. The green- and yellow-shaded areas in the graph show, respectively, that 45% of the applications are invoked once per hour or less on average, and 81% of the applications are invoked once per minute or less on average. This suggests that the cost of keeping these applications warm, relative to their total execution (billable) time, can be prohibitively high.

Figure 5(b) shows the other side of the workload skewness, by looking at the cumulative fraction of invocations due to the most popular functions and applications in the sample. The shaded areas correspond to the same applications as in Figure 5(a). The applications in the orange-shaded area are the 18.6% most popular, those invoked on average at least once per minute. They represent 99.6% of all function invocations.

The invocation rates provide information on the average inter-arrival time (IAT) of function and application invocations, but not on the distribution of these IATs. If the next invocation time of a function can be predicted, the platform can avoid cold starts by pre-warming the application right before it is to be invoked, and save resources by shutting it

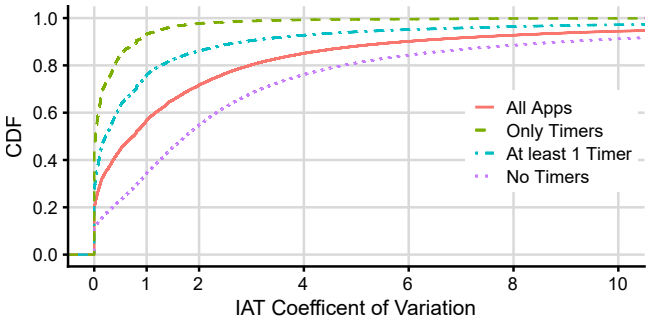



Figure 6: CV of the IATs for subsets of applications.

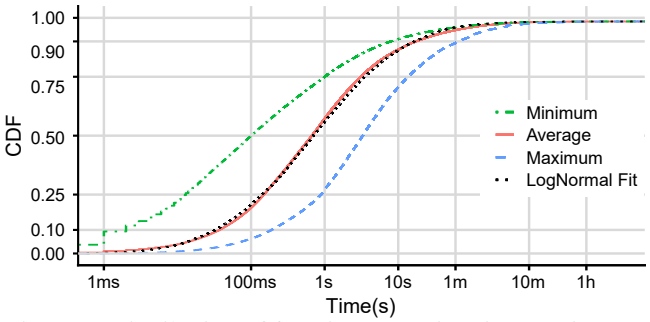



Figure 7: Distribution of function execution times. Min, avg, and max are separate CDFs, and use independent sorting.

down right after execution.

**Inter-arrival time variability.** To gain insight into the IAT distributions of applications, we look at the coefficient of variation (CV) of each application. The CV (standard deviation divided by the mean) provides a measure of the variability in the IATs. We would expect timer-triggered functions to have periodic arrivals, with a CV of 0. Human-generated invocations should approximately follow a Poisson arrival process, with an exponential (memoryless) distribution of IATs [16]. These would ideally yield a CV of 1. CVs greater than 1 suggest significant variability.

Figure 6 shows the distribution of the CV across all applications, as well as for subsets of applications with and without timers. It shows that the real IAT distributions are more complex than the simply periodic or memoryless ones. For example, only ∼50% of the applications with only timer-triggered functions have a CV of 0. Multiple timers with different periods and/or phases will increase the CV. For applications with at least one timer, this fraction is less than 30%, and across all applications the fraction is ∼20%. Interestingly, ∼10% of applications with no timers have CV close to 0, which means they are quite periodic, and should be predictable. This could be due to, for example, external callers (e.g., sensors or IoT devices) that operate periodically. On the other hand, only a small fraction of applications has a CV close to 1, meaning that simple Poisson arrivals are not the norm. These results show that there is a significant fraction of applications that should have fairly predictable IATs, even if they do not have timer triggers. At the same time, these numbers suggest that for many applications predicting IATs is not trivial.

## 3.4 Function Execution Times

Another aspect of the workload is the function execution time, *i.e.* the time functions take to execute *after they are ready to run*. In other words, these numbers do not include the cold start times. Cold start times depend on the infrastructure to a large extent, and have been characterized in other studies [44].

Figure 7 shows the distribution of average, minimum, and maximum execution times of all function executions on July 15th, 2019. The distributions for other days are similar. The graph also shows a very good log-normal fit (via MLE) to the distribution of the averages, with log mean -0.38 and $\sigma$ 2.36. We observe that 50% of the functions execute for less than 1s on average, and 50% of the functions have maximum execution time shorter than ∼3s; 90% of the functions take at most 60s, and 96% of functions take less than 60s on average.

The main implication is that the function execution times are at the same order of magnitude as the cold start times reported for major providers [44]. *This makes avoiding and/or optimizing cold starts extremely important for the overall performance of a FaaS offering.*

Another interesting observation is that, overall, functions in this FaaS workload are very short compared to other cloud workloads. For example, data from Azure [12] shows that 63% of all VM allocations last longer than 15 minutes, and only less than 8% of the VMs last less 5 minutes or less. This implies that FaaS imposes much more stringent requirements on the provider to stand-up resources quickly.

**Idle times.** As we discuss in Section 4, an important aspect of the workload for managing cold starts is idle time (IT), defined as the time between the end of a function's execution and its next invocation. IT relates to IAT and execution time. For most applications, the average execution time is at least 2 orders of magnitude smaller than the average IAT. We verified for the applications in the yellow region in Figure 5(a) – 81% of the applications invoked at most once per minute on average – that indeed the IT and IAT distributions are extremely similar.

**Potential correlations.** Different triggers had average function execution times differing by about 10×, between 200ms and 2s at the median, but all with the same shape for the distributions. One outlier was a class of orchestration functions with median average execution times of ∼30ms, as they simply dispatch and coordinate other functions.

## 3.5 Memory Usage

We finally look at the memory demands of applications. Recall that the application is the unit of memory allocation in the platform we study. Figure 8 shows the memory demand distribution, across all applications running on July 15th, 2019. We present three curves drawn from the memory data: 1st percentile, average, and maximum allocated memory for the application. We also plot a reasonably good Burr distribution fit (with parameters $c = 11.652$, $k = 0.221$, and $\lambda = 107.083$) for the average. Allocated memory is the amount of virtual

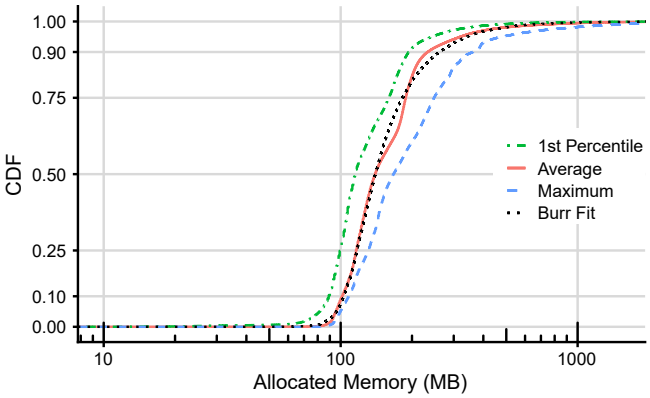


Figure 8: Distribution of allocated memory per application.

memory reserved for the application, and may not necessarily be all resident in physical memory. Here, we use the 1st percentile because there was a problem with the measurement of the minimum, which made that data not usable. Despite the short duration of each function execution, applications tend to remain resident for longer. The distributions for other days in the dataset are very similar.

Looking at the distribution of the maximum allocated memory, 90% of the applications never consume more than 400MB, and 50% of the applications allocate at most 170MB. Overall, there is a 4× variation in the first 90% of applications, meaning that memory is an important factor in warmup, allocation, and keep-alive decisions for FaaS.

**Potential correlations.** We found no strong correlation between invocation frequency and memory allocation or between memory allocation and function execution times.

## 3.6 Main Takeaways

From the point of view of cold starts and resource allocation, we now reiterate our three main observations. First, the vast majority of functions execute on the order of a few seconds – 75% of them have a maximum execution time of 10 seconds – so execution times are on the same order as the time it takes to start functions cold. Thus, it is critical to reduce the number of cold starts or make cold starts substantially faster. Eliminating a cold start is the same as making it infinitely fast.

Second, the vast majority of applications are invoked infrequently – 81% of them average at most one invocation per minute. At the same time, less than 20% of the applications are responsible for 99.6% of all invocations. Thus, it is expensive, in terms of memory footprint, to keep the applications that receive infrequent invocations resident at all times.

Third, many applications show wide variability in their IATs – 40% of them have a CV of their IATs higher than 1 – so the task of predicting the next invocation can be challenging, especially for applications that are invoked infrequently.

# 4 Managing Cold Starts in FaaS

We use insights from our characterization to design an adaptive resource management policy, called *hybrid histogram policy*. The goal is to reduce the number of cold start invocations with minimum resource waste. We refer to a *policy* as a set of rules that govern two parameters *for each application*:

— *Pre-warming window.* The time the policy waits, since the last execution, before it loads the application image expecting the next invocation. A pre-warming window = 0 means that the policy does not unload the application after one of its functions executes. Aggressive pre-warming (a large window) reduces resource usage but may also cause cold starts, in case the next invocation occurs sooner than expected.

— *Keep-alive window.* The time during which an application's image is kept alive after (1) it has been loaded to memory (pre-warming window $\geq 0$) or (2) a function execution (pre-warming window = 0). (Note that our definition for this window differs from the keep-alive parameter in fixed keep-alive policies, which is the same for all applications and only starts at the end of function executions.) Longer windows have the potential to reduce cold starts by increasing the chances of an invocation falling into this window. However, this may also waste resources, *i.e.* leave them idle, in case the next invocation does not happen soon after loading.

A *no-unloading policy* would keep every application image loaded in memory all the time (*i.e.*, infinite keep-alive window and pre-warming window = 0). This policy would get no cold starts but would be too expensive to operate.

## 4.1 Design Challenges

Designing a *practical policy* poses several challenges:

1. **Hard-to-predict invocations.** As Figure 3 shows, many applications are triggered by timers. A timer-aware policy could leverage this information to pre-warm applications right before the next invocation. However, predicting the next invocation is challenging for other triggers.
2. **Heterogeneous applications.** As Figure 5 shows, the 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. A better policy should adapt to each application dynamically.
3. **Applications with infrequent invocations.** Some applications are invoked very infrequently, so an adaptive policy would take some time to learn their invocation patterns. The same applies for applications that it sees for the first time.
4. **Tracking overhead.** Adapting the policy to each application means tracking each application individually. For this reason, the cost to track the information for each application should be small. For example, we need to consider the size of the data structures that will keep this state.
5. **Execution overhead.** Since function executions can be very short (*i.e.*, more than 50% of executions take less than 1 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 for each 10 ms-long execution. Due

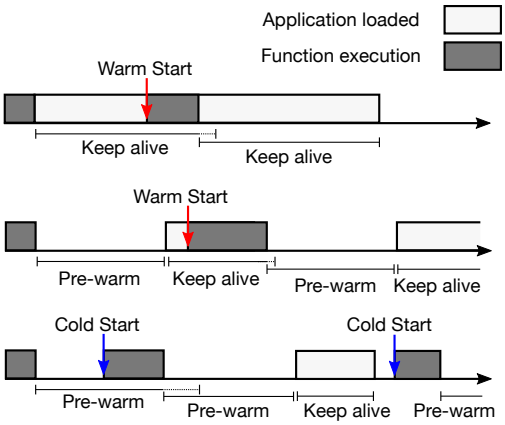



Figure 9: Timelines showing a warm start with keep alives and no pre-warming (top); a warm start following a pre-warm (middle); and two cold starts, before a pre-warm, and after a keep alive (bottom).

to these overheads, expensive prediction techniques, such as time-series analysis, cannot be used for all applications.

## 4.2 Hybrid Histogram Policy

**Overview.** Our hybrid histogram policy addresses all the above challenges. To address challenges #1 and #2, our policy adjusts to the invocation frequencies and patterns of each individual application. It identifies the 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 (after a "pre-warming window" elapses), and keeps it alive for a period (until a "keep-alive window" elapses). The pre-warming window starts after each function execution, and the keep-alive window starts after each pre-warming. If the pre-warming window is 0, we do not unload the application after an execution, and the end of the execution still starts a new keep-alive window. We explain how exactly we compute the length of these windows below.

Figure 9 shows the pre-warming and keep-alive windows in three scenarios. In the top scenario, the pre-warming window is 0, and an invocation that happens before the keep-alive window ends is a warm start. The end of the execution starts a new keep-alive window. In the middle, the next invocation is a warm start, as the application is re-loaded after a pre-warming window. The end of the execution starts a new pre-warming window. In the bottom scenario, there are two cold starts: the first resulting from an invocation arriving before the pre-warming window elapsed, and the second from an invocation arriving after the keep-alive period elapsed.

The policy comprises three main components: (1) a range-limited histogram for capturing each application's "idle" times (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. Figure 10

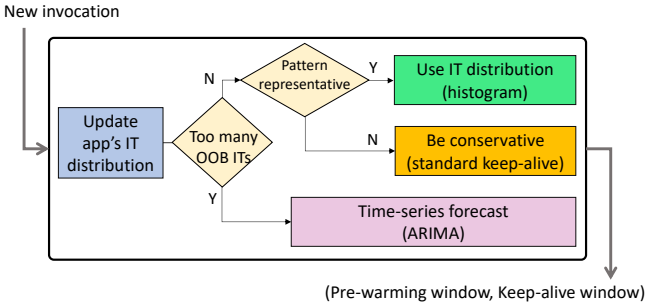



Figure 10: Overview of the hybrid histogram policy.

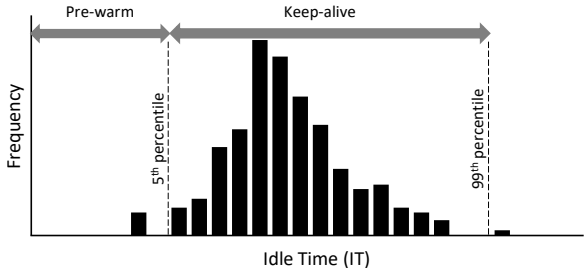



Figure 11: Example application idle time (IT) distribution used to select pre-warming times and keep-alive windows.

overviews our policy and its components. Ultimately, the policy defines the pre-warming and keep-alive windows for each application. Next, we describe each component in turn.

**Range-limited histogram.** To address challenges #4 and #5, the centerpiece of our policy is a compact histogram data structure that tracks the IT distribution for each application. Each entry/bin of the histogram counts the number of ITs of the corresponding length that have occurred. We use 1-minute bins, which strikes a good balance between metadata size and the resolution needed for policy actions. Keep-alive time scales are in orders of minutes for FaaS platforms. We use the same scale for pre-warming. In addition, the histogram tracks ITs of up to a configurable duration (*e.g.*, 4 hours). Any ITs longer than this are considered "out of bounds" (OOBs).

Given the ITs that are within bounds, our policy identifies the head and tail of the IT distribution. We use the head to select the pre-warming window for the application, and the tail to select the keep-alive window. To exclude outliers, we set the head and tail by default to the $5^{th}$- and $99^{th}$-percentiles of the IT distribution. (When one of these percentiles falls within a bin, we "round" it to the next lower value for the head or the next higher value for the tail.) These two configurable thresholds strike a balance between managing cold starts and resource costs. Figure 11 shows the histogram for a sample application, and the head and tail markers. To give the policy a little room for error, our implementation uses a 10% "margin" by default, *i.e.* it reduces the pre-warming window by 10% and increases the keep-alive window by 10%.

Figure 12 shows nine real IT distributions over a week. The three histograms in the left column show cases where both head and tail cutoffs are easy to identify. These distributions produce the ideal situation: long pre-warm windows and short

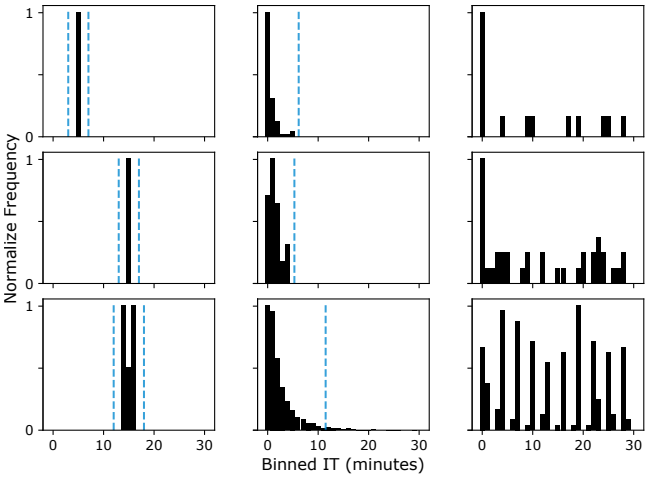



Figure 12: Nine normalized IT distributions from real FaaS workloads over a week.

keep-alive windows. The center cases show no head cutoff as the head marker rounded down to 0. In these cases, the pre-warming window is 0 and the policy does not kill the application after a function execution.

**Standard keep-alive when the pattern is uncertain.** The histogram might not be representative of an application's behavior when (1) it has not observed enough ITs for the application, or (2) when the application is transitioning to a different IT regime (*e.g.*, change from a consistent pattern to an entirely new one). When the histogram is not representative, we revert to a standard keep-alive approach: pre-warming window = 0 and keep-alive window = range of the histogram (*e.g.*, 4 hours). This conservative choice of keep-alive window seeks to minimize the number of cold starts while the histogram is learning a new pattern. Our policy reverts back to using the histogram when it becomes representative (again).

We decide whether a histogram is representative by computing the CV of its bin counts. A histogram that has a single bin with a high count and all others 0 would have a high CV, whereas a histogram where all bins have the same value would have CV = 0. The histogram is most effective in the former case, where there is a large concentration of ITs (left and center of Figure 12). It is not as effective when ITs are spread widely (right of Figure 12). Thus, if the CV is lower than a threshold, we use the standard keep-alive approach. To track the CV efficiently, we use Welford's online algorithm [45].

**Time-series analysis when histogram is not large enough.** A compact histogram cannot represent ITs larger than its range. Thus, applications with very infrequent invocations (challenge #3) may exhibit many out-of-bounds ITs. For these applications, our policy uses time-series analysis to predict the next IT. Specifically, we use ARIMA modeling [11].

With an IT prediction, our policy sets the pre-warm window to elapse just before the next invocation and a short keep-alive window. In more detail, we used the `auto_arima` implementation from the `pmdarima` package [2], which automatically searches for the ARIMA parameters $(p, d, q)$ that produce the best fit. As applications using ARIMA are invoked very infrequently, we update the model for each of them after every invocation. To give the prediction some room more inaccuracy, we include a (configurable) margin of 15%. For example, if the predicted IT is 5 hours, we set the pre-warming window to 4.25 hours (5 hours minus 15%) and the keep-alive window to 1.5 hours (15% of 5 hours in each side of the IT prediction).

**Justification.** Like other FaaS cold start policies, our policy eagerly frees up memory when it is not needed. An alternative would have been to leverage standard (lazy) caching policies, which free up cache space only on-demand. Section 7 explains the differences between these types of policies that justify our approach. Our policy uses a standard keep-alive with a long window, when it does not have accurate IT data about the application, to conservatively prevent cold starts. A shorter window would lower cost but would incur more cold starts. We prefer our approach because it often quickly reduces memory usage greatly, after the histogram becomes active for the application. Instead of using a histogram, we could attempt to predict the next invocation or idle time using time-series analysis or other prediction models. We experimented with some models, including ARIMA, but found them to be inaccurate or excessively expensive for the bulk of invocations. The histogram is accurate, compact, and fast to update. So, we rely on ARIMA only for the applications that cannot be represented with a compact histogram. Producing an ARIMA model is expensive, but can be off the critical path. Moreover, these applications involve only a small percentage of invocations, so computation needs are kept small. Nevertheless, we can easily replace ARIMA with another model.

## 4.3 Implementation in Apache OpenWhisk

We implement our policy in Apache OpenWhisk [34], which is the open-source FaaS platform that powers IBM's Cloud Functions [21]. It is written in Scala.

**OpenWhisk architecture.** Figure 13 shows the architecture of OpenWhisk [35]. It exposes a REST interface (implemented using Nginx) for users to interact with the FaaS platform. A user can create new functions (*actions* in OpenWhisk terminology), submit new invocations (*activations* in OpenWhisk terminology), or query their status. Here, we focus on function invocation and container management. Invocation requests are forwarded to the Controller component, who decides which Invoker should execute each function instance. This logic is implemented in the Load Balancer, which considers the health and available capacity of the Invokers, as well as the history of prior executions. The Controller sends the function invocation request to the selected Invoker via the distributed messaging component (implemented using Kafka). The Invoker receives the invocation request, starts the function in a Docker container, and manages its runtime (including when to stop the container). By default, each Invoker implements a fixed 10-minute keep-alive policy, and informs the

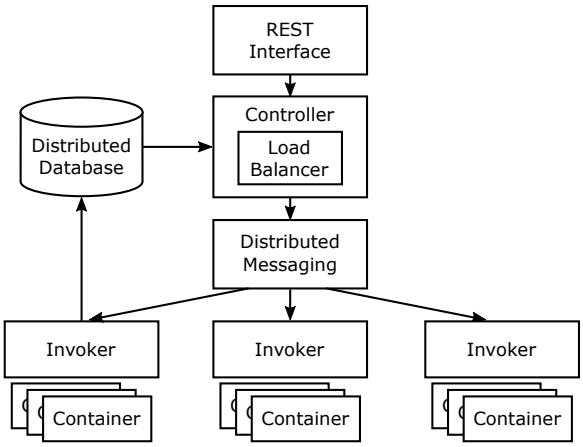



Figure 13: OpenWhisk architecture.

Controller when it unloads a container.

**Implementing our policy.** We modify the following OpenWhisk components to implement the hybrid policy:

1. **Controller:** Since all invocations pass through the Load Balancer, it is the ideal place to manage histograms and other metadata required for the hybrid policy. We add new logic to the Load Balancer to implement the hybrid policy and to update the keep-alive and pre-warm parameters after each invocation. We also modify the Load Balancer to publish the pre-warming messages.
2. **API:** We send the latest keep-alive parameter for a function to the corresponding Invoker alongside the invocation request. To do this, we add a field to the *ActivationMessage* API, specifying the keep-alive duration in minutes.
3. **Invoker:** The Invoker unloads Docker containers that have timed-out in the *ContainerProxy* module. We modify this module to unload containers based on the keep-alive parameter received from *ActivationMessage*.

# 5 Evaluation

## 5.1 Methodology

**Simulator.** Evaluating our policy requires (1) long executions to assess applications with infrequent invocations, and (2) exploring a large space of configurations. To limit the evaluation time, we use simulations. We build a simulator that allows us to compare various policies using real invocation traces.

The simulator generates an array of invocation times for each unique application. It then infers whether each invocation would be a cold start. By default, the first invocation is always assumed to be a cold start. The simulator keeps track of when each application image is loaded and aggregates the wasted memory time for the application, *i.e.* the time when the application's image was kept in memory without actually executing any functions. We conservatively simulate function execution times equal to 0 to quantify the worst-case wasted resource time. We do not have memory usage data for all applications, so we also simulate that applications use the same amount and focus on the wasted memory time.

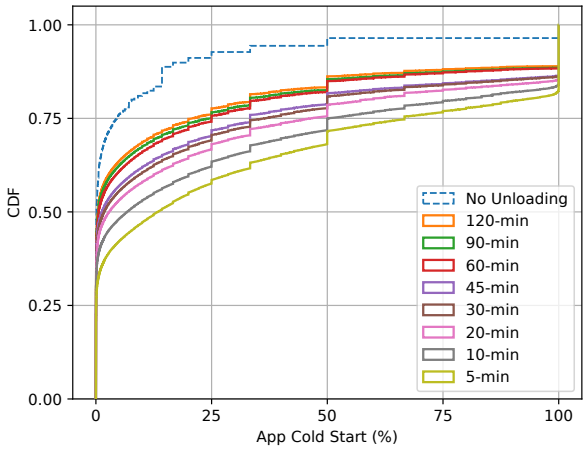



Figure 14: Cold start behavior of the fixed keep-alive policy, as a function of the keep-alive length.

**Real experiments.** To show that our policy can be easily implemented in real systems with minimal overheads, we use our OpenWhisk implementation (Section 4.3). Our setup consists of 19 VMs. One VM with 8 cores and 8GB of memory hosts containers for the Controller and other main components, including Nginx and Kafka. Each of the remaining 18 VMs has 2 cores and 4GB of memory, hosting an Invoker to actually run the functions in Docker containers.

**Workloads.** As input to our simulations, we use the first week of the trace from Section 3. For the real experiments, we use a scaled-down version of the trace. We randomly select applications with mid-range popularity. As we run the full system, we limit each OpenWhisk execution to only 8 hours. As we show in Section 5.3, *the experimental and simulation results show the same trends in both cold start and memory consumption behaviors.*

## 5.2 Simulation Results

**Understanding the fixed keep-alive policy.** We start evaluating the policy used by most providers: the fixed keep-alive policy. We first assess how the length of the keep-alive affects the cold starts. Figure 14 shows the distribution of cold start percentage experienced by all applications for various lengths. For comparison, we also include a *No unloading* policy, which corresponds to each application only experiencing the initial cold start. Even the *No unloading* policy has ∼3.5% of applications with 100% cold starts; these applications have only one invocation in the entire week.

We see significant cold start reduction going from a 10-minute keep-alive to 1-hour. The 75$^{th}$-percentile application experiences cold starts 50.3% of the time for the 10-minute keep-alive. This number goes down to 25% for 1-hour. The cold start improvement is more pronounced in the last quartile of the distribution, since applications with infrequent invocations are those that benefit the most. From now on, we will focus on this metric (*i.e.*, 75$^{th}$-percentile) to report cold starts.

While a longer keep-alive reduces cold starts significantly,

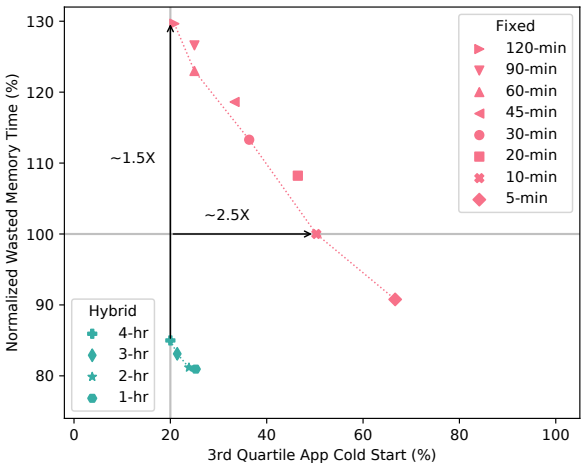



Figure 15: Trade-off between cold starts and wasted memory time for the fixed keep-alive policy and our hybrid policy.

it also increases the resources wasted significantly. The red markers in Figure 15 show the trade-off between cold starts and memory wasted, where we normalize the wasted memory time to the 10-minute keep-alive. The red curve near the red markers approximates the Pareto curve. The figure shows, for example, that a fixed 2-hour keep-alive has almost 30% higher wasted memory time than the 10-minute baseline. An optimal policy would deliver the lowest cold starts with minimum cost. We rely on these Pareto curves to evaluate the policies.

**Impact of using a histogram.** We now start to evaluate our hybrid policy with the impact of the histogram and its range. The green markers in Figure 15 show the cold start percentage and wasted memory time of our histogram for various ranges. The figure shows how our policy reduces the cold starts significantly with lower memory waste. In fact, the 10-minute fixed keep-alive policy involves ∼2.5x more cold starts at the 75$^{th}$-percentile while using the same amount of memory as our histogram with a range of 4 hours. From a different perspective, the fixed 2-hour keep-alive policy provides roughly the same percentage of cold starts as the 4-hour histogram range, but consumes about 50% more resources. Overall, the hybrid policies form a parallel, more optimal Pareto frontier (green curve) than the fixed policies (red curve).

**Impact of the histogram cutoff percentiles.** Our policy uses two cutoff percentiles to exclude outliers in the head and tail of the IT distribution. Figure 16 shows the sensitivity study that we used to determine suitable cutoff values. The figure shows that, by setting the head and tail cutoffs to the 5$^{th}$- and 99$^{th}$-percentiles of the IT distribution (labeled *Hybrid[5,99]* in the figure), the cold start percentage does not degrade noticeably whereas the wasted memory time goes down by 15%, compared to the case with no cutoff (*Hybrid[0,100]*).

**Impact of unloading and pre-warming.** Complementing our adaptive keep-alive with pre-warming allows unloading of an application right after execution and pre-warming right before the next invocation. This reduces the wasted memory time of application images. Figure 17 shows this, where using

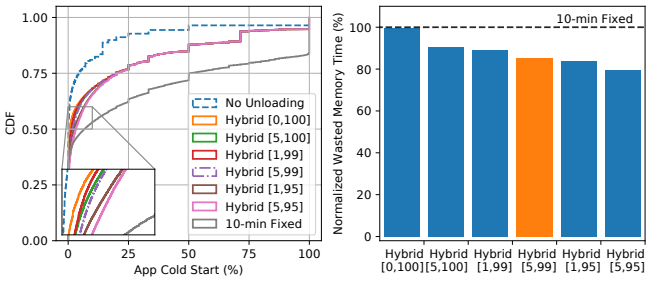



Figure 16: Wasted memory time can be significantly reduced by excluding outliers from the IT distribution.

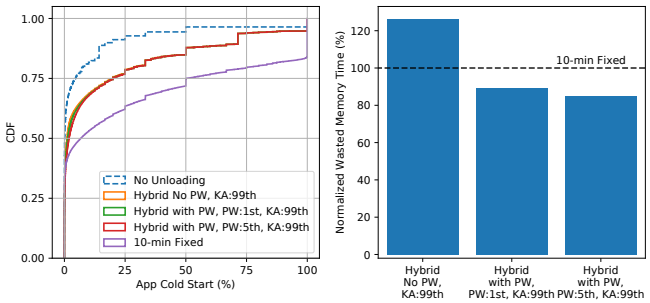



Figure 17: Pre-warming reduces the wasted memory time significantly. The cost is slight increase in cold starts.

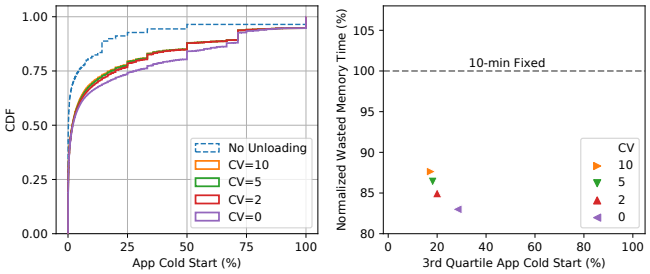



Figure 18: Trade-off between cold starts and memory wasted, as a function of the CV threshold, using a 4-hour range.

similar keep-alive (KA) configurations with and without pre-warming (PW) has significantly different wasted memory time. The cost, however, is adding a small number of cold starts from unexpected invocations. We can control this trade-off by adjusting the histogram head cutoff percentile.

**Impact of checking the histogram representativeness.** Our policy checks whether the histogram is representative before using it. If the histogram is not representative (*i.e.*, the CV of its bin counts is lower than a threshold), it uses a standard keep-alive approach where applications stay loaded for the same length as the histogram range. We study the impact of different CV thresholds in Figure 18. The figure shows the application cold start distributions (left) and the Pareto frontier (right). We see significant gains using a small CV threshold larger than 0. We opt for CV=2 as our default threshold. Increasing the CV further has negligible cold start reduction with higher resource costs.

**Impact of using time-series analysis.** Another feature of our hybrid policy is to use ARIMA modeling for applications

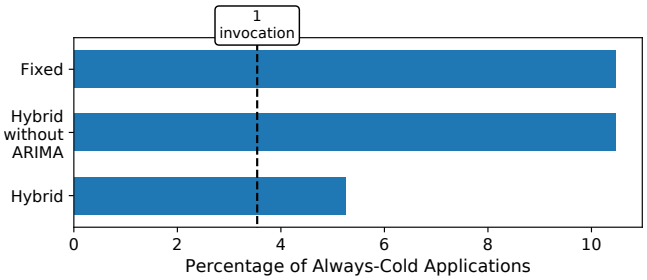



Figure 19: Percentage of applications that always experience cold starts, as a function of policy.

that have many ITs outside the range of the histogram. To evaluate its impact, we now focus on the percentage of applications that show 100% cold starts. Figure 19 shows this percentage when using (1) the fixed keep-alive policy, (2) the hybrid policy without ARIMA, and (3) the full hybrid policy (including ARIMA). All of them use 4 hours for the fixed keep-alive and the histogram range. During the week-long simulation window, 0.64% of invocations were handled by ARIMA, and 9.3% of applications used ARIMA at least once. Using ARIMA reduces 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. This shows that ARIMA provides benefits for applications that cannot benefit from a fixed keep-alive or a histogram-based policy.

**Summary.** Our hybrid policy can reduce the number of cold starts significantly while minimizing the memory cost. We achieve these positive results despite having deliberately designed our policy for simplicity and practicality: (1) histogram bins have a resolution of 1-minute, (2) histograms have a maximum range, (3) they do not require any pre-processing or complicated model updates, and (4) when the histogram does not work well, we resort to simple and effective alternatives.

## 5.3 Experimental results

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 4-hour histogram range. Each experiment ran for 8 hours. During the 8-hour period, there are a total of 12,383 function invocations. We use FaaSProfiler [1, 38] to automate trace replay and result analysis.

Figure 20 compares the cold start behavior of the hybrid and 10-minute fixed keep-alive policies. The significant cold start reductions follow the *same trend as our simulations* (left graph of Figure 16). On average and across the 18 Invoker VMs, the hybrid policy reduced memory consumption of worker containers by 15.6%, which is also *consistent with our simulation results* (right graph of Figure 16). Moreover,

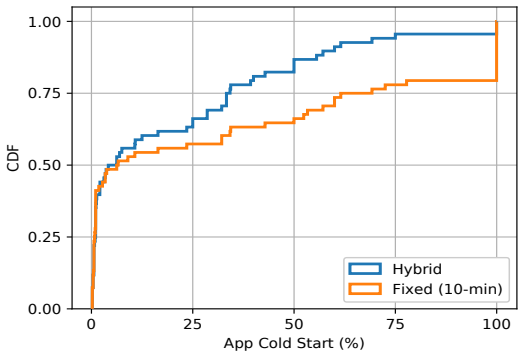



Figure 20: Cold start behavior of fixed keep-alive and hybrid policies in OpenWhisk.

the hybrid policy reduced the average and 99-percentile function execution time 32.5% and 82.4%, respectively. This is due to a secondary effect in OpenWhisk, where the language runtime bootstrap time is eliminated for warm containers.

**Policy overhead.** We measure the (1) additional latency induced by our implementation and (2) the impact of our policy on the scalability of the OpenWhisk controller. The Scala code that implements our policy in the Controller adds an average of only 835.7$\mu s$ ($\sigma = 245.5\mu s$) to the end-to-end latency. This overhead is negligible compared to the existing latency of OpenWhisk components: the (in-memory) language runtime initiation takes $O(10ms)$ and the container initiation takes $O(100ms)$ for cold containers [38]. For the uncommon cases where ARIMA is required (0.7% of invocations), the initial forecast involves building the model, which takes an average of 26.9ms, whereas subsequent forecasts take an average of 5.3ms. Since ARIMA works for applications that would normally experience cold starts, these overheads represent a relatively small cost compared to the cold start overhead.

In terms of scalability, CPU utilization is the limiting factor for the Controller. Our policy adds only a 4-6% higher utilization for a range of benchmarking request rates (10rps to 300rps), compared to OpenWhisk's default policy.

# 6 Production Implementation

We have implemented our policy in Azure Functions for HTTP-triggered applications; its main elements will be rolling out to production in stages starting this month. Here, we overview the implementation.

Azure Functions has a controller that communicates with function-execution workers through HTTP, and a database for persisting system state. The controller gets asynchronous updates from the workers at fixed intervals; we use these to populate the histogram. We keep the histogram in memory (bucket of 240 integers per application, or 960 bytes) and do hourly backups to the database. We start a new histogram per day in the database so we can track changes in application's invocation pattern, and remove histograms older than 2 weeks. We can potentially use these daily histograms in a weighted fashion to give more importance to recent records.

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. Some steps, like JIT of the function code, happen when the actual invocation comes in as the function's code cannot be executed as part of warmup to preserve execution semantics. Each worker maintains the keep-alive duration separately, depending on how long it has been idle for. We make all policy decisions asynchronously, off the critical path to minimize the latency impact on the invocation. This includes updating the in-memory histogram, backing up the histogram to the database, scheduling pre-warming events, and controlling the workers' keep alive intervals.

## 7 Related Work

There is a fast-increasing number of studies on different aspects of serverless computing. The most relevant for our paper are those that characterize FaaS platforms and applications, and those that propose and optimize FaaS serving systems.

**FaaS characterization.** A few studies [7, 15, 24–26, 44] have characterized the main commercial FaaS providers, *but only from the perspective of external users.* They typically reverse-engineer aspects of FaaS offerings, by running benchmark functions to collect various externally visible metrics. Our characterization is orthogonal to these works, as we provide a longitudinal characterization of the entire workload of a large cloud provider from the provider's perspective. *Our characterization is the first of its kind.*

Another class of studies looks at the ways developers are using FaaS offerings, by looking at public application repositories [41]. While valuable, this approach cannot offer insights on the aggregate workload seen by a provider.

**Optimizing FaaS serving.** Another set of relevant work considers optimizing different aspects of FaaS systems. Van Eyk *et al.* [42] identify performance-related challenges, including scheduling policies that minimize cold starts. They also identify the lack of execution traces from real FaaS platforms as a major obstacle to addressing the challenges they identified.

For optimizing each cold start, Mohan *et al.* [32] find that pre-allocating virtual network interfaces that are later bound to new function containers can significantly reduce cold start times. SOCK [33] proposes to optimize the loading of Python functions in OpenLambda by smart caching of sets of libraries, and by using lightweight isolation mechanisms for functions. SAND [3] uses application-level sandboxing to prevent the cold start latency for subsequent function invocations within an application. Azure Functions warms all functions within an application together; thus this was not a concern for us. Replayable Execution [43] proposes checkpointing and sharing of memory among containers to speed up the startup times of a JVM-based FaaS system. Kaffes *et al.* [22] propose a centralized core-granular scheduler. *Our work on reducing the number of cold starts and resource usage by predicting function invocations is orthogonal to these improvements.*

Other studies also use prediction to optimize different aspects. Work in [19, 20] proposes a policy for deciding on function multi-tenancy, based on a predictive model of resource demands of each function. Without discussing design details, EMARS [37] proposes using predictive modeling for allocation of memory to serverless functions. Kesidis [23] proposes to use the prediction of the resource demands of functions to enable the provider to overbook functions on containers. In contrast, we track invocation patterns and use this knowledge to reduce cold starts and memory waste.

**Cache management.** Finally, one might think that the problem of managing cold starts is similar to managing caches of variable-sized objects, such as Web page caches and others [4, 8, 36]. However, there are two fundamental differences. First, FaaS frameworks are often implemented on top of services that charge by the time resources are allocated (*e.g.*, each application is packaged as a container and deployed to a container service). Thus, cold start policies proactively unload applications/functions from memory, instead of waiting for other applications/functions to need the space. Our policy is closest to a class of TTL-based caches where new accesses reset the TTL [9, 10]. These works did not consider temporal prefetching, the equivalent of our pre-warming. Other caching work did consider it, but with capacity-based replacements [46]. Second, most caching algorithms to date have focused on aggregate performance metrics [13, 14], such as the weighted sum or average of per-object miss ratios. In contrast, we tailor our cold start management to each application to maximize individual customer satisfaction.

## 8 Conclusion

In this paper, we characterized the entire production FaaS workload of Azure Functions. The characterization 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. We evaluated the policy using both simulations and a real implementation, and real workload traces. Our results showed that the policy can achieve the same number of cold starts at much lower resource cost, or keep the same resource cost but reduce the number of cold starts significantly. Finally, we overviewed our policy's implementation in Azure Functions. We released sanitized traces from our characterization data at [31].

## Acknowledgements

We would like to thank our shepherd, George Amvrosiadis, and the anonymous reviewers for helping us improve this paper. We also thank Daniel Berger, Bill Bolosky, and Willy Zwaenepoel for their comments on earlier versions of it.

## References

[1] FaaSProfiler. http://parallel.princeton.edu/FaaSProfiler.html.

[2] Pmdarima. https://github.com/alkaline-ml/pmdarima.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. USENIX ATC, 2018.

[4] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A Survey of Web Caching and Prefetching. *International Journal of Advances in Soft Computing and its Applications*, 3(1), 2011.

[5] Amazon. AWS Lambda. https://aws.amazon.com/lambda/.

[6] Amazon. Invoking AWS Lambda Functions. https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html.

[7] Timon Back and Vasilios Andrikopoulos. Using a Microbenchmark to Compare Function as a Service Solutions. ESOCC, 2018.

[8] Abdullah Balamash and Marwan Krunz. An Overview of Web Caching Replacement Algorithms. *IEEE Communications Surveys & Tutorials*, 6(2):44–56, 2004.

[9] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai, and R. Sitaraman. Adaptive TTL-Based Caching for Content Delivery. *IEEE/ACM Transactions on Networking*, 26(3):1063–1077, 2018.

[10] Daniel Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact Analysis of TTL Cache Networks. *Performance Evaluation*, 79:2 – 23, 09 2014.

[11] George EP Box and David A Pierce. Distribution of Residual Autocorrelations in Autoregressive-Integrated Moving Average Time Series Models. *Journal of the American Statistical Association*, 65(332), 1970.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. SOSP, 2017.

[13] Mostafa Dehghan, Laurent Massoulie, Don Towsley, Daniel Sadoc Menasche, and Yong Chiang Tay. A Utility Optimization Approach to Network Cache Design. *IEEE/ACM Transactions on Networking*, 27(3):1013–1027, 2019.

[14] Andrés Ferragut, Ismael Rodríguez, and Fernando Paganini. Optimizing TTL Caches Under Heavy-tailed Demands. *ACM SIGMETRICS Performance Evaluation Review*, 44(1):101–112, 2016.

[15] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. Performance Evaluation of Heterogeneous Cloud Functions. *Concurrency and Computation: Practice and Experience*, 30(23), 2018.

[16] Robert G Gallager. *Stochastic Processes: Theory for Applications*. 2013.

[17] Google. Google Cloud Functions. https://cloud.google.com/functions/.

[18] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless Computation with OpenLambda. HotCloud, 2016.

[19] Mohammad Reza Hoseiny Farahabady, Javid Taheri, Zahir Tari, and Albert Y Zomaya. A Dynamic Resource Controller for a Lambda Architecture. ICPP, 2017.

[20] Mohammad Reza Hoseiny Farahabady, Albert Y Zomaya, and Zahir Tari. A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform. *Transactions on Parallel and Distributed Systems*, 29(7), 2017.

[21] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[22] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized Core-Granular Scheduling for Serverless Functions. SoCC, 2019.

[23] George Kesidis. Temporal Overbooking of Lambda Functions in the Cloud. *arXiv preprint arXiv:1901.09842*, 2019.

[24] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Dominik Ernst, and Daniel Wenzel. Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-Driven Design of Serverless Applications. SAC, 2020.

[25] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of Production Serverless Computing Environments. CLOUD, 2018.

[26] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. IC2E, 2018.

[27] Garrett McGrath and Paul R Brenner. Serverless Computing: Design, Implementation, and Performance. ICDCSW, 2017.

[28] Microsoft. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[29] Microsoft. Azure Functions Triggers and Bindings Concepts. https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings.

[30] Microsoft. What are Durable Functions? https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[31] Microsoft Azure and Microsoft Research. Azure Functions Traces. https://github.com/Azure/AzurePublicDataset.

[32] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. HotCloud, 2019.

[33] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-optimized Containers. USENIX ATC, 2018.

[34] OpenWhisk. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[35] Apache OpenWhisk. How OpenWhisk works. https://github.com/apache/openwhisk/blob/master/docs/about.md.

[36] Stefan Podlipnig and Laszlo Böszörmenyi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, 35(4):374–398, December 2003.

[37] Aakanksha Saha and Sonika Jindal. EMARS: Efficient Management and Allocation of Resources in Serverless. CLOUD, 2018.

[38] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural Implications of Function-as-a-Service Computing. MICRO, 2019.

[39] Mikhail Shilkov. Cold Starts in AWS Lambda. https://mikhail.io/serverless/coldstarts/aws/.

[40] Mikhail Shilkov. Cold Starts in Azure Functions. https://mikhail.io/serverless/coldstarts/azure/.

[41] Josef Spillner. Quantitative Analysis of Cloud Function Evolution in the AWS Serverless Application Repository. *arXiv preprint arXiv:1905.04800*, 2019.

[42] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. ICPE, 2018.

[43] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. EuroSys, 2019.

[44] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. USENIX ATC, 2018.

[45] BP Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3), 1962.

[46] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal Prefetching Without the Off-Chip Metadata. MICRO, 2019.