Optimizing the Serverless Workload at Cloud

Jingyi Yang[1], Siyue Zhang[1], and Ziqi Yin[1]

[1]NTU SCSE.

Abstract

With the prevalence of cloud computing, both individuals and enterprises are increasingly reliant on cloud providers to manage the computing infrastructure. Among the cloud-computing models, Function-as-a-Service has gained popularity over recent years as it completely hides the complexity of managing the server from the user. One key problem in providing FaaS is designing the cold start management policy, *i.e.*, when to unload the application from memory after the function execution finishes. Designing the right cold start management is particularly challenging as one needs to trade off between reducing cold starts and saving memory resources. In this report, we evaluate existing cold start management policies for FaaS through simulation, and propose improvements over the hybrid histogram policy, a recently proposed adaptive policy.

1 Introduction

Function-as-a-Service (FaaS) is a serverless cloud-computing model that enables the user to trigger the application function executions (e.g., HTTP and timer) without the need to build and manage the complex underlying infrastructure, including containers, operating systems, virtual and physical servers. Since being initially offered by the start-up PiCloud in 2010 [1], FaaS has gained unprecedented popularity and been adopted by most of the cloud providers, e.g., AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions. The success of FaaS owes to its obvious advantages in efficiency, scalability, and cost. With FaaS, developers can spend more time on application development and less time on infrastructure management, which results in a much faster development turnaround. Functions could be scaled up or down automatically, independently, and instantaneously based on real-time traffic requirements. Moreover, the user is only required to pay for the resources when the function is running, metered with millisecond accuracy. Due to these advantages, FaaS has been applied in a broad range of use cases, such as the Internet of Things (IoT) [2], chatbot [3], data processing [4], and machine learning [5].

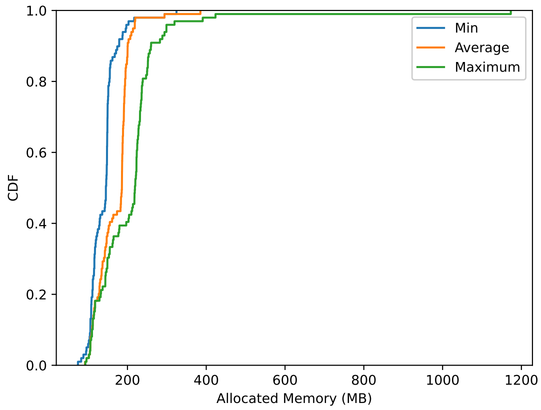
In FaaS, the cloud provider is responsible for executing the application function and provisioning the resources needed. Thus, it is imperative for the provider to achieve high function performance with the least resources consumed. One of the key performance metrics is the function starting time. When the application code is already in the memory, its functions can be launched quickly, which is called a warm start. On the other hand, in the cold start, it takes more time to access the code in the persistent storage and start the function. However, keeping the application code in the memory at all times can be prohibitively expensive, especially for short and infrequent applications. Therefore, the trade-off between performance and resource cost is necessary. A fixed keep-alive policy is typically used by many cloud providers, which retains the applications in the memory for 10 and 20 minutes after execution [6]. To further optimize the trade-off, [7] firstly collected real-life function invocation data across Azure infrastructure for two weeks and characterized the heterogeneous production workloads. An adaptive hybrid histogram-based policy was proposed to balance the function warm start rate and wasted memory time. The policy dynamically regulates the loading and unloading of applications in the memory.

In this serverless computing project, our work and contribution include:

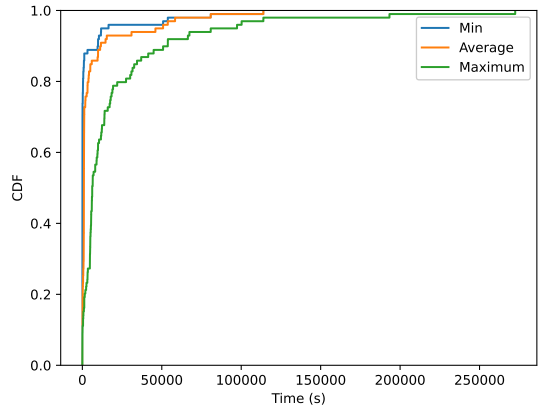
- Exploratory analysis of Azure FaaS workload data
- Generation of realistic function and application traces based on given distributions of invocation time, duration, application memory, etc.
- Simulation and evaluation of fixed keep-alive policy and hybrid histogram policy
- Proposition of improvements in the hybrid histogram policy
- Conclusion and discussion of simulation results

2 FaaS Workload Data Analysis

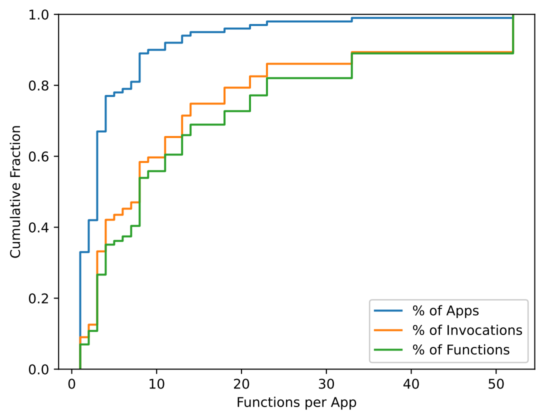
We generate the workload according to the public dataset provided by Azure [8]. The workload consists of a list of functions, where each function is described by the function id, corresponding app id, trigger type, start time, execution duration and memory cost. As the dataset only provides statistics on the distribution of the execution duration and memory cost, we need to generate the execution duration and memory cost for each function that we feed into the simulation. In the following, we introduce our workload generation process in detail. We also analyze the characteristics of the data generated.



(a)



(b)



(c)

Figure 1: (a) Distribution of allocated memory per app. (b) Distribution of function execution time. (c) Distribution of the number of functions per app.

First, we load the function invocations counts data from the dataset. As the data contain the number of invocations in every 1-minute time slot for each function, we generate the function invocations by randomly sampling the start time from the 1-minute interval. As the dataset contains a huge number of function invocations, *e.g.*, day 1 contains over 1000,000,000 total function invocations, we sample a fraction of the application for our experiments. Specifically, we randomly sample 100 applications from the dataset and include all function invocations of the 100 applications in the workload. We conduct the simulation based on the generated workload of the 100 sampled applications.

As discussed, we also need to generate the memory cost and execution duration for each function. It is worth noting that the dataset does not provide the duration time and memory cost directly, but statistics on the distribution of execution duration and memory cost *i.e.*, values at different percentiles. We approximate the cumulative density function (CDF) of the execution duration and memory as a piece-wise linear function based on the distribution statistics. Specifically, for each adjacent pair of percentile values, we use a linear function to approximate the CDF.

To demonstrate the pattern of the data we generate, we plot the graph based on the generation function workload of day 1. The distribution of allocated memory per application of day 1 is shown in Figure 1a. We have similar distribution compared to Figure 8 in the original paper, which indicates the consistency between the data we generate and the original data. The distribution of function execution time is shown in Figure 1b, which shows a similar distribution to Figure 7 in the original paper. And Figure 1c shows the CDF of the number of functions per application which has similar results compared to Figure 1 of the original paper. These data patterns demonstrate that the data we sampled has similar data distribution to the original data, which means high sampling quality.

3 Simulation Designs

To evaluate the performance of different cold start management policies, we build a simulator to simulate the execution of real-world invocation traces. We implement different cold start management policies on top of the simulator and record the key performance metrics for each. In this section, we first go through our simulator and simulation approach. We then discuss in detail different existing cold start management policies. We further propose improvements propose over the existing policies.

3.1 Simulator

We build the simulator to as closely resemble the real-world execution of the functions as possible, while ensuring the efficiency of the simulation process so that we are able to simulate on large-scale datasets. To achieve this, we first sort all function invocations by the start time in chronological order, and then simulate the function invocations and update the system state at the start of each function invocation. Note that this is much more efficient then updating the system state at fixed time intervals. Unlike [7], which considers the worst case of cold start and set all execution duration to 0 for the simulation, our simulator also takes into account the execution duration of each function invocation, so that our simulation results more closely reflects the actual execution.

3.2 Fixed Keep-Alive Policy

The fixed keep-alive policy is adopted by most FaaS providers [6, 9] and open-source FaaS frameworks [10]. It keeps the application loaded in the memory for a fixed amount of time after the function execution finishes, so that follow-up function invocations happening within the keep-alive window can have a warm start.

While the fixed keep-alive help reduce the overall number of cold starts, it has several limitations. **1) High memory waste**: the application is kept loaded in the memory the entire time before the next function invocation, and the memory the application takes up while being idle is wasted. **2) Cold start for infrequent invocation**: the fixed keep-alive policy uses a one-size-fit-all approach where it uses the same keep-alive window for all applications. As a result, applications with infrequent involvement have a high cold start rate as their idle time usually exceeds the keep-alive window.

3.3 Hybrid Histogram Policy

The fixed keep-alive policy could be ineffective when the function is called periodically with a long idle time. The application is kept in the memory after the function execution, but the next invocation comes much later than the end of the fixed keep-alive window, which results in a significant amount of memory waste. To overcome this issue, the hybrid histogram policy unloads the application for a pre-warm window after the function execution and before the start of the keep-alive window. The duration of these two windows is determined by the app's IT distribution. For each application, ITs refer to the time intervals when none of its functions is executed. At the arrival of each invocation, if its application is already in the memory, there is no IT recorded. If the application is not loaded at the moment, the time interval is collected as IT, between the current invocation start time and the last moment when the application was loaded in the memory. ITs are recorded in a list every day, where each list demonstrates an IT distribution. The heterogeneity of IT distribution among applications and changes over time have been observed as in Figure 2.

In each invocation of the simulation, there are three scenarios where the pre-warm window and the keep-alive window are determined differently as shown in Figure 3. **Time-series forecast** scenario is entered when there are many ITs longer than 4 hours, namely out of bounds (OOBs). The auto Autoregressive Integrated Moving

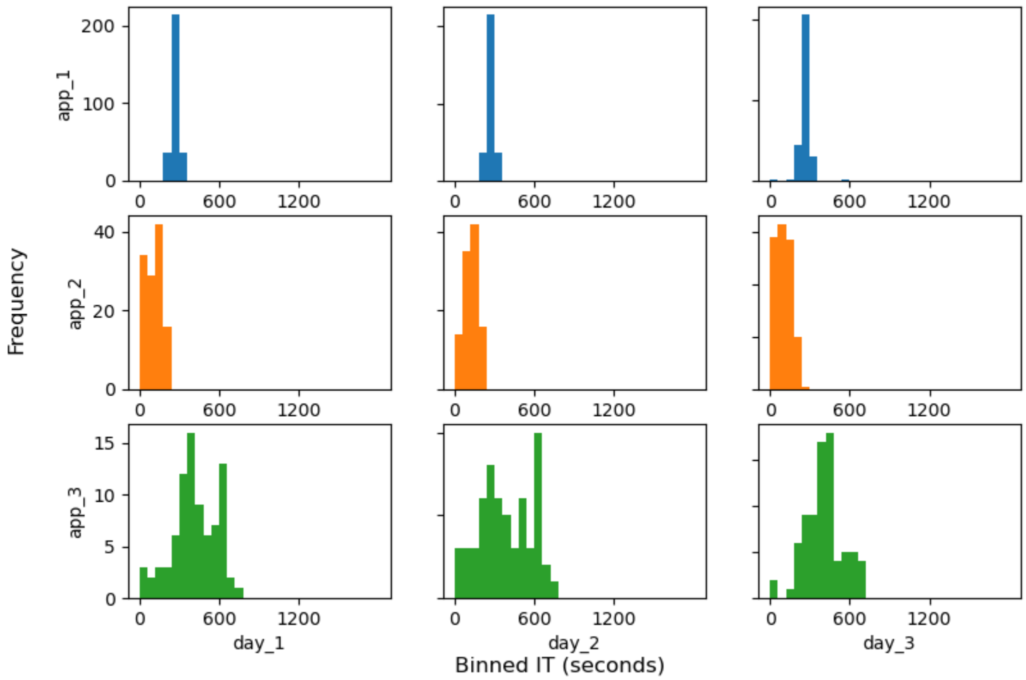


Figure 2: IT distributions of three selected applications in three days.

Average (ARIMA) model is used to forecast the next IT based on all historical ITs. The pre-warm window is set as 85% of the predicted IT and the keep-alive window is 30% of the predicted IT to capture the next invocation. When most ITs are not OOB and the histogram has a representative pattern, **Use IT distribution** scenario is valid. If the histogram has a high coefficient of variation (CV), its pattern is regarded as representative. In this scenario, the pre-warm window is defined as the 5^{th} percentile of IT distribution and the keep-alive window is equal to the difference between the 5^{th} percentile and the 99^{th} percentile as Figure 4. As for the last scenario, **Be conservative**, a standard keep-alive approach is applied when the histogram is not representative. This approach sets the pre-warm window as 0 and the keep-alive window as the range of the histogram. It retains the application in the memory after the execution for a period that is longer than most historical ITs to ensure the invocations have warm starts.

Through dynamically updating the IT distribution histogram, the hybrid histogram policy is able to capture the change in IT distribution over time for each application and adjust the pre-warm window and keep-alive window adaptively. Three scenarios in the hybrid histogram policy accommodate the heterogeneity in the IT

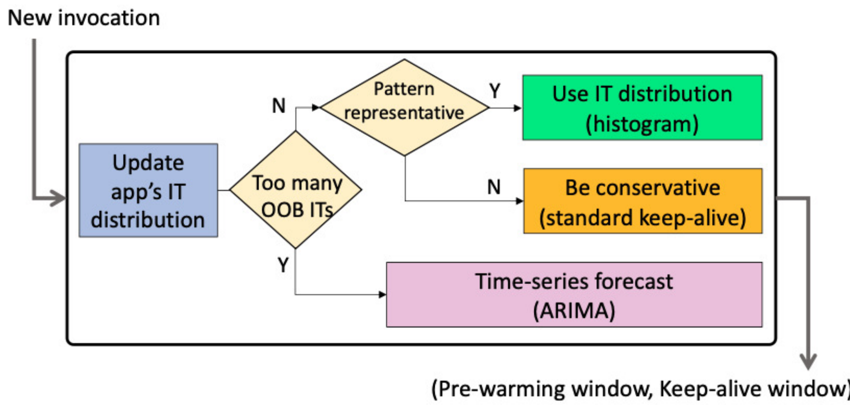


Figure 3: Overview of the hybrid histogram policy. [7]

distributions for diverse applications. Therefore, it is a well-designed policy with balanced trade-offs.

3.4 Proposed Improvements

Trigger-Dependent Histogram

The hybrid histogram policy treats all function invocations of the same application equally, *i.e.*, the same rule is applied to determine the length of the pre-warm window and keep-alive window for all function invocations. However, function invocations of an application have different trigger types, and invocations with different trigger types may have different characteristics. For example, timer invocations and HTTP requests may have completely different idle time distributions. As the type of trigger affects the IT distribution of function invocations, we propose to adjust the cold start management policies based on both the application id and the trigger type. Specifically, we could choose different pre-warm and keep-alive windows for function invocations of different trigger types. For example, HTTP-triggered functions have more variance in the IT distribution, we take the keep-alive window between 10^{th} percentile and the 90^{th} percentile. While timer-triggered functions are more regular, we could take the keep-alive window between 1^{th} percentile and the 99^{th} percentile.

Forecasted Histogram

In the hybrid histogram policy, the pre-warm window and keep-alive window of the current invocation are determined by the collected IT distribution. When there are few ITs in the beginning of histogram collection cycle, the standard keep-alive approach is applied as default. However, this IT distribution might have a representative

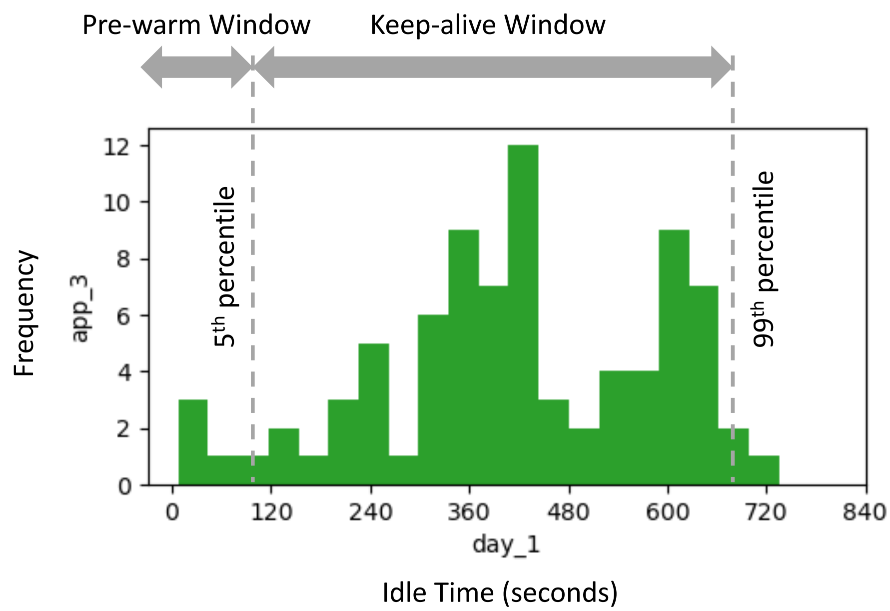


Figure 4: Example application idle time (IT) distribution used to select pre-warming times and keep-alive windows.

pattern after more ITs are collected. Therefore, we propose to use the forecasted histogram in this kind of situation. Based on the historical histograms, we could forecast the percentiles of the complete IT histogram, based on which the pre-warm window and keep-alive window are calculated. As the method is more effective when the IT distribution is changing gradually, we only apply the forecasting when there is no sudden change in historical histograms for early invocations every day.

4 Simulation Results

In this section, we present our simulation results of different cold start management policies. We simulate the function executions using the two cold start management policies and gather key performance metrics. For the fixed keep-alive policy, we use a keep-alive window of 10 minutes. For the hybrid histogram policy, we use combinations of different cut-off percentiles.

Table 1 shows the number of cold start, number of warm start, cold start rate and memory waste time of each policy. For the hybrid histogram policies, the two numbers indicate the two cut-off percentiles. For example, the hybrid[5, 99] policy indicates we set 5 minimum percentiles idle time to be the pre-warm window, set 5-99 percentiles idle time to be the keep-alive time, and give up the 99-100 maximal percentiles idle time. We observe that various hybrid histogram policies achieve competitive performance compared with the keep-alive policy in terms cold start rate. The hybrid [5, 99] policy achieves a 9.94e-5 cold start rate, and the memory wasted time is 7.20e6. In comparison, the fixed keep-alive policy achieves a 3.94e-05 cold start rate and the memory wasted time is 1.09e7. We can see the hybrid[5, 99] policy achieves comparative performance on cold start rate but with much less memory wasted time. In general, for hybrid histogram policy, as the first cut-off percentile increases, the cold start rate increases and the memory waste time decreases. As the second cut-off percentile decreases, the cold start rate increases and the memory waste time decreases.

Table 1: Policy Evaluation

Policy	Cold start	Warm start	Cold start rate	Memory waste time
Fixed Keep-Alive	1,592	4.04e7	3.94e-05	1.09e7
Hybrid [0, 100]	1,975	4.04e7	4.88e-05	9.39e6
Hybrid [5, 100]	3,695	4.04e7	9.13e-05	7.26e6
Hybrid [1, 99]	2,680	4.04e7	6.62e-05	8.15e6
Hybrid [5, 99]	4,021	4.04e7	9.94e-05	7.20e6
Hybrid [1, 95]	4,067	4.04e7	1.00e-04	7.89e6
Hybrid [5, 95]	5,408	4.04e7	1.34e-04	6.94e6

Impact of the histogram cutoff percentiles. To determine the cutoff percentiles of the hybrid histogram policy, we further evaluate the performance of the histogram policy as the cutoff percentile varies. Fig. 5 shows the cumulative density function (CDF) of the application cold start rate of different policies. We observe that the fixed keep-alive policy achieves the lowest cold start rate, as it reduces the cold start rate at the cost of keeping all applications in memory after function execution finishes. For hybrid histogram policies, the cold start rate increases as the first cut-off percentile increases, or as the second cut-off percentile decreases. Fig. 6 shows the normalized memory waste time of different policies, where the memory waste time is normalized by that of the fixed keep-alive policy. We observe that the fixed keep-alive policy has the highest waste memory time, while the waste memory time increases as the first cut-off percentile increases, or as the second cut-off percentile decreases for hybrid histogram policies. From the two figures, we identify that Hybrid [5, 99] has the most suitable cut percentiles, as it achieves a significant reduction in waste memory time without compromising too much on cold start rate.

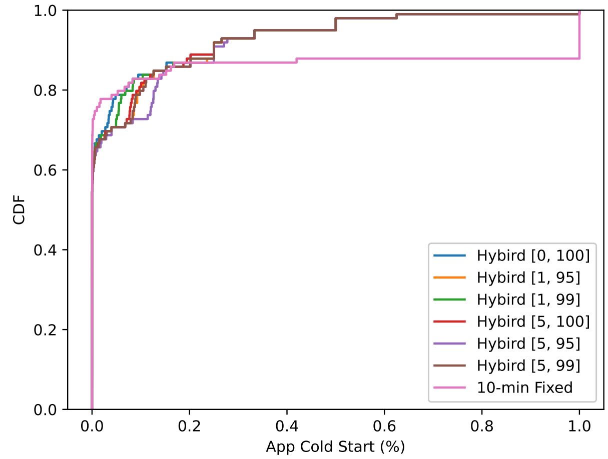


Figure 5: Cumulative density function of app cold start rate of different policies.

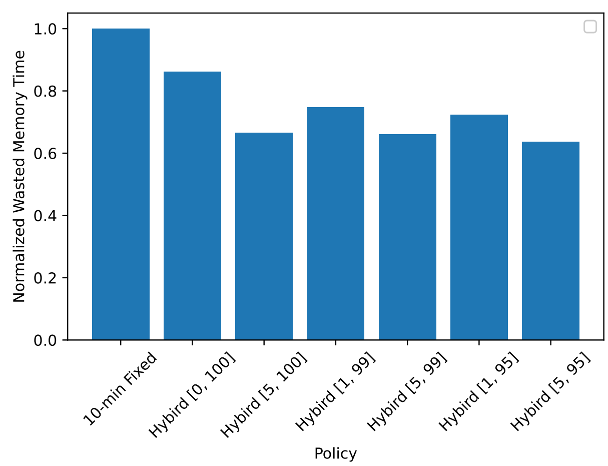


Figure 6: Waste memory time of different policies.

5 Conclusion and Discussion

In this report, we evaluate two existing FaaS cold start management policies, the fixed keep-alive policy, and the hybrid histogram policy. We generate a realistic workload of function execution traces from the public dataset released by Azure, simulate the function executions using the two cold start management policies, and gather key performance metrics. Our experiments demonstrate the superior performance of the hybrid histogram policy over the fixed keep-alive policy. We further propose two improvements over the hybrid histogram policy, Trigger-Dependent Histogram and Forecasted Histogram.

References

[1] L. Rao, *Picloud launches serverless computing platform to the public*, https://techcrunch.com/2010/07/19/picloud-launches-serverless-computing-platform-to-the-public/, 2010.

[2] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "Cspot: Portable, multi-scale functions-as-a-service for iot," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 236–249.

[3] S. Choudary, "Chatbot on serverless/lamba architecture," *Asian Journal of Engineering and Technology Innovation (AJETI)*, p. 190, 2018.

[4] A. Pogiatzis and G. Samakovitis, "An event-driven serverless etl pipeline on aws," *Applied Sciences*, vol. 11, no. 1, p. 191, 2020.

[5] V. Sreekanti *et al.*, "Cloudburst," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2438–2452, Aug. 2020. DOI: 10.14778/3407790.3407836. [Online]. Available: https://doi.org/10.14778%2F3407790.3407836.

[6] M. Shilkov, *Cold starts in aws lambda*, https://mikhail.io/serverless/coldstarts/aws/.

[7] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 205–218, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad.

[8] *Azure function traces*, https://github.com/Azure/AzurePublicDataset, 2021.

[9] M. Shilkov, *Cold starts in azure functions*, https://mikhail.io/serverless/coldstarts/azure/.

[10] OpenWhisk, *Open source serverless cloud platform*, https://openwhisk.apache.org/.