Temporal Performance Modelling of Serverless Computing Platforms

Nima Mahmoudi
Dept. of Electrical and Computer Engineering
University of Alberta
Edmonton, AB, Canada
nmahmoud@ualberta.ca

Hamzeh Khazaei
Dept. of Electrical Engineering and Computer Science
York University
Toronto, ON, Canada
hkh@yorku.ca

Abstract

Analytical performance models have been shown very efficient in analyzing, predicting, and improving the performance of distributed computing systems. However, there is a lack of rigorous analytical models for analyzing the transient behaviour of serverless computing platforms, which is expected to be the dominant computing paradigm in cloud computing. Also, due to its unique characteristics and policies, performance models developed for other systems cannot be directly applied to modelling these systems.

In this work, we propose an analytical performance model that is capable of predicting several key performance metrics for serverless workloads using only their average response time for warm and cold requests. The introduced model uses realistic assumptions, which makes it suitable for online analysis of real-world platforms. We validate the proposed model through extensive experimentation on AWS Lambda. Although we focus primarily on AWS Lambda due to its wide adoption in our experimentation, the proposed model can be leveraged for other public serverless computing platforms with similar auto-scaling policies, e.g., Google Cloud Functions, IBM Cloud Functions, and Azure Functions.

Keywords: performance modelling, serverless computing, serverless, temporal, transient, performance

1 Introduction

Serverless computing is a new paradigm in cloud computing systems in which the cloud service provider manages nearly all of the administrative tasks while dynamically scaling the resources consumed by the application. In this paradigm, the user (application developer) is billed merely for the resources consumed by their application, not the provisioned maximum capacity required by their system. It also enables developers to handle massive changes in their workload volume by providing almost infinitely scalable computing.

This computing paradigm has emerged to address some of the shortcomings in previous generations [7]. Some of the benefits of using serverless computing compared to the previous paradigm in cloud computing are improvements in resource utilization, cost savings, energy efficiency, scalability, and fast and convenient application development. However, the current generation of serverless computing platforms are not fulfilling the potential benefits of adopting this paradigm. For example, although serverless instances have much faster startup times compared to containers or virtual machines, they still are suffering from low and unpredictable performance, offering only best-effort quality guarantees. This has shown to be unacceptable for many consumer-facing applications [7], e.g. hosting websites or latency-sensitive workloads.

The mainstream serverless computing offerings are shown to be non-adaptive and oblivious to the workload being executed on them, with the same managing policies being used for all applications being executed on them [10–12]. This leaves great potentials in performance and infrastructure cost improvements achievable by adapting the system parameters according to each individual workload. In this work, we propose leveraging analytical performance models as a driver for adaptation of the platform to each workload being executed on them. Using analytical performance models, the system can ensure that the key performance metrics for each workload, with its unique characteristics, remain within the acceptable bounds.

In this work, we strive to develop a transient analytical performance model that can capture the unique characteristics of serverless computing platforms and make predictions about the resource requirements and key service quality metrics for the near future. To ensure the applicability of the model through its tractability and fidelity, we validate our model using real-world traces gathered from AWS Lambda. Using the introduced model, serverless providers can create performance-driven and predictive platforms, improving their Quality of Service (QoS) and reducing their wasted computing resources. Application developers would also benefit from such performance models by gaining the necessary

WoSC6, December 7-11 2020, TU Delft, Netherlands

quality of service guarantees enabling them to migrate more workloads into the serverless computing platforms.

The rest of the paper is structured as follows: Section 2 gives a description of the system modelled in this work. Section 3 goes through the details of the presented analytical model. Section 4 outlines the experimental validation of our work using real-world traces from AWS Lambda. Finally, Section 5 concludes the paper and presents our plans for future work.

2 System Description

In this section, we will present a description of the system for which we will develop and validate an analytical performance model. Since there is very little official documentation made publicly available about the operation and management of serverless offerings by cloud vendors, this information needs to be gathered through experimentation and reverse-engineering these offerings. Thus, we decided to use the results of previous work done in this area [3, 5, 8, 12], as well as our own set of experiments to determine several important details about the operation and management of the public serverless computing platforms. In this section, we will go through our findings as a foundation for the proposed analytical model.

In serverless computing platforms, each request (i.e., trigger) is served by a function instance. To achieve simplified billing in which you only pay for the amount of resources that you used instead of the provisioned capacity, there are no provisioned instances for each function of the user by default. Instead, the vendor will spin up new function instances, as they are needed to handle the workload and release the consumed resources when appropriate, providing the application developer with seamless autoscaling.

In current serverless computing platforms, there is no queuing involved for the requests, which increases the responsiveness of the platform. As a result, any request will immediately be assigned to an instance. In case there are no provisioned (warm) instances, the platform will spin up a new one of the function, and then passes the request to the newly created instance. This process, which increases the observed latency, is also referred to as a cold start [5, 8, 12]. However, if the system has warm instances when an arrival occurs, it reuses one of its readily available instances to handle the incoming request. This is commonly referred to as a warm start. So, as defined in this paper, functions instances have three possible states: 1) Initializing; 2) Running; and 3) Idle. The initialization state is referred to when the system is preparing a given function instance to accept requests, including the instance creation and the user initialization code execution. The running state occurs when the instance starts processing a request and continues until a response is sent out to the client. The idle instance signifies warm instances that are not processing any requests and are being kept to enable handling of future surges in the workload.

After processing a request, an instance enters the idle state. In this state, the instance is kept warm until a timeout occurs, causing the system to terminate the instance in order to free up unused resources. We refer to this timeout as the *expiration timeout*. This value is known to be deterministic and fixed for all user functions in current public serverless offerings [9, 10].

The autoscaling described here is referred to as the *scale-per-request* autoscaling pattern hereafter, and the performance model presented in this work only addresses this pattern. We acknowledge that other autoscaling patterns are also available, e.g. Google Cloud Run, but we chose this autoscaling pattern due to its adoption in the mainstream public serverless computing platforms.

When more than one warm instance is available at the time of arrival, the system needs a policy determining the instance that the request will be routed to. We found in our experimentation that the system routes requests to the instances that have been created more recently. This, in turn, maximizes the chance of older instances to be terminated.

Any serverless computing platform has some limitations on the number of concurrently running instances for a given function at any time. This is commonly known as the *maximum concurrency level*. When this limitation is reached, in most cases, the new requests will be rejected with some form of error message, letting them know that the system is not able to handle their request at the moment.

3 Analytical Model

In Section 2, we outlined the operation and management of modern public serverless computing platform with the scale-per-request autoscaling pattern. In this section, we will go through the design and development of a novel temporal analytical performance model to enable the prediction of several important performance metrics through time. Temporal performance models are especially of importance in serverless computing platforms due to their sporadic and highly-dynamic environments for which steady-state performance models could be rendered obsolete to ensure the performance metrics remain within acceptable ranges.

In the process of development of an accurate analytical performance model for serverless computing platforms, an ideal platform can be modelled by a simple $M/G/\infty$ queuing system [6] with Markovian arrival process, general service process, and infinite processing capacity in which every request goes through the same service process and no latency is imposed due to queuing in the system. But unfortunately, modern serverless computing platforms as still far from perfect due to the presence of cold starts, which could be orders of magnitude longer than warm starts, and the limitations on

maximum concurrency level, which could lead to request rejection due to reaching the maximum capacity in the system. In this work, we make modifications to the $M/G/\infty$ queuing system to reflect these changes to build a comprehensive temporal performance model for serverless computing platforms.

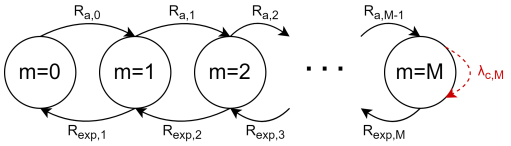

Figure 1. The state transition diagram of the warm pool. The dashed red self-loop shows rejected requests due to insufficient capacity.

In this work, we used the number of instances in the warm pool to represent the state of the system, ranging from 0 to the *maximum concurrency level*. In this encoding, any transition from a state with a lower number of instances to a state with a higher number of instances is the result of a cold start, causing the creation of a new function instance. This only occurs if there are no idle instances in our warm pool, which is modelled here using $M/G/m/m$ queuing systems [6] with m servers and system capacity of m in the pool of warm servers. In this model, the expiration event occurs *expiration timeout* units of time after the instance with the lowest priority (as defined in the system description) has served its last request, leading to the termination of an instance and a transition to a state with a lower number of instances. Figure 1 shows an overview of the presented model.

In the presented state encoding, we have an equivalent queue for the warm pool for each state of the system. In the equivalent queue, we have m servers and m cells in the queue in state m. However, unlike typical queuing systems, unless the maximum concurrency level is reached, the rejected requests end up creating a new instance and becoming a cold start event. To model such a queuing system, we used the M/G/m/m queuing system, which reflects the equivalent Erlang loss system [6] that is formed for each state.

The rest of this section will focus on the calculation of the rates in the presented model, solving the resulting Semi-Markov Process (SMP) model, and extraction of different key performance metrics from the resulting solution. In each state, we only have two possible transitions, a cold start causing a transition to a state with a warm pool with an extra server and an expiration timeout. Thus, we only need to calculate these rates to derive a solution for the model.

3.1 Cold Start Rate

As discussed before, we model each state of the warm pool using $M/G/m/m$ queuing systems. In this modelling method, we assume a rejection by the warm pool to cause a cold start and creation of a new instance, which will later be added to the warm pool. Thus, to calculate the cold start rate in each state, we need to calculate the rate of request rejection in the equivalent $M/G/m/m$ queuing system. We know that state the state probabilities of an $M/G/m/m$ queuing system is equivalent to an $M/M/m/m$ queuing system [4]. Thus, to calculate the probability of a cold start, we can use the blocking probability using the Erlang's B formula [2]:

$$P_{B,m} = B(m, \rho) = \frac{\frac{\rho^m}{m!}}{\sum_{j=0}^{m} \frac{\rho^j}{j!}} \tag{1}$$

where ρ is the offered load, which can be calculated using the following equation:

$$\rho = \lambda / \mu_w \tag{2}$$

where λ is the average arrival rate, and μ_w is the average service rate for warm starts requests. Using this equation, we can calculate the probability of a request being blocked by the warm queue in a pool with m servers. In the case where we have not yet reached the maximum concurrency level (M) where $m < M$, the blocking probability gives us the probability of a cold start (P_{cld}). In case the system has reached the maximum concurrency level ($m = M$), the blocking probability gives us the rejection probability (P_{rej}) where the requests are either served by the warm queue or rejected due to insufficient available resources. Thus, we can calculate these metrics using the state distribution π where π_m is the probability of being in a state with m instances in the warm pool:

$$P_{rej} = P_{B,M}\pi_M \tag{3}$$

and for the cold start probability, we have:

$$P_{cld} = \sum_{m=0}^{M-1} P_{B,m}\pi_m \tag{4}$$

It is also important to calculate the actual arrival rate entering the warm queue ($\lambda_{w,m}$):

$$\lambda_{w,m} = \lambda(1 - P_{B,m}) \tag{5}$$

Also, for the rate of cold start arrivals, we have:

$$\lambda_{c,m} = \lambda P_{B,m} \tag{6}$$

which we will leverage later on to calculate the transition rates in the presented performance model.

3.2 Arrival Rate of Warm Instances

In order to calculate the instance expiration rate in the proposed model, we need to calculate the arrival rate for individual instances in the warm pool. To do so, let's first assume we have a warm pool of m instances shown as $\{I_1, I_2, ..., I_m\}$ where I_1 has the highest priority for new arrivals, and I_m has

the lowest priority. Consider $\lambda_{w,m,n}$ as the arrival rate of the nth instance in the pool.

As defined in eq. (1), $P_{B,n-1}$ is the ratio of arrivals that find $\{I_i; i = 1, 2, ..., n-1\}$ busy, and thus are handled by an instance of lower priority or are rejected. Considering the same pattern for $P_{B,n}$, we can see that:

$$P_{S,n} = P_{B,n-1} - P_{B,n} \tag{7}$$

where $P_{S,n}$ shows the probability of a request being processed by the nth instance in the warm pool with the edge case of $P_{S,0} = 1$. Using this equation, we can calculate the arrival rate for instances in the warm pool:

$$\lambda_{w,m,n} = \lambda_{w,m} P_{S,n} \tag{8}$$

3.3 Instance Expiration Rate

In this section, we aim to calculate the expiration rate of the instances in each state. Using eq. (8), we calculated the arrival rate for each instance of the warm pool. Using this value, we want the rate of expiration, assuming it will happen after *expiration timeout* units of time have passed since the last request. Using the aforementioned rate and considering exponential inter-arrival times, we have the following Probability Density Function (PDF):

$$f_X(x) = \lambda_{w,m,i} \cdot e^{-\lambda_{w,m,i} x} \tag{9}$$

and the following Cumulative Density Function (CDF):

$$F_X(x) = 1 - e^{-\lambda_{w,m,i} x} \tag{10}$$

For each arrival, the probability of it being the last request served by the instance before the expiration of the instance is:

$$P_{lst,m,i} = P\{X > T\} = e^{-\lambda_{w,m,i} T} \tag{11}$$

where $T = T_{exp} + 1/\mu_w$ is the sum of expiration timeout and the average service time of a request. Thus, considering a geometric distribution for each request being the last one served by an instance, we have the average number of arrivals before the expiration event occurs:

$$C_{req,m,i} = \frac{1}{P_{lst,m,i}} \tag{12}$$

Thus, on average, each instance will serve $C_{req,m,i}$ requests with inter-arrivals times shorter than T. Thus, to calculate the expected time an instance is kept warm, we need to calculate the average inter-arrival time assuming they are less than T:

$$\begin{aligned} E[X; X < T] &= \int_0^T x \lambda_{w,m,i} e^{-\lambda_{w,m,i} x} dx \\ &= -T \cdot e^{-\lambda_{w,m,i} T} + \frac{1 - e^{\lambda_{w,m,i} T}}{\lambda_{w,m,i}} \end{aligned} \tag{13}$$

Using this equation, we can calculate the average expected lifespan of an instance in the warm pool:

$$E[LS_{m,i}] = \{(C_{req,m,i} - 1) \cdot E[X; X < T]\} + \frac{1}{\mu_w} + T_{exp} \tag{14}$$

with the following expiration rate for the given instance:

$$R_{exp,m,i} = \frac{1}{E[LS_{m,i}]} \tag{15}$$

giving us the expiration rate of instance I_i in a warm pool of size m. Since any expiration event in the warm pool will cause a transition to a warm pool of size $m-1$, we can calculate the overall expiration rate for a warm pool of size m using the following:

$$R_{exp,m} = \sum_{i=0}^{m} R_{exp,m,i} \tag{16}$$

3.4 Temporal Solution of the Proposed Model

Figure 1 shows an overview of the overall model proposed in this paper. As mentioned before, each state m in the proposed model represents the number of function instances in the warm pool, and M represents the maximum concurrency level. In this section, we will go over the state transition rates, using the rates calculated in sections 3.1 to 3.3. In each state, $R_{a,m}$ is the rate of transitioning to a state with an additional function instance, and $R_{exp,m}$ is the rate of expiring and terminating a function instance, leading to a warm pool of size $m-1$. In eq. (16), we calculated the rate of expiration for each state. To calculate $R_{a,m}$, we first need to consider the fact that a new function is always created using a cold start, but the new server will not be available in the warm pool until it is done processing the initial request. Thus, the rate of adding new instances to the warm pool can be calculated as follows:

$$R_{a,m} = \frac{1}{\frac{1}{\lambda_{c,m}} + \frac{1}{\mu_c}} = \frac{\lambda_{c,m} \cdot \mu_c}{\lambda_{c,m} + \mu_c} \tag{17}$$

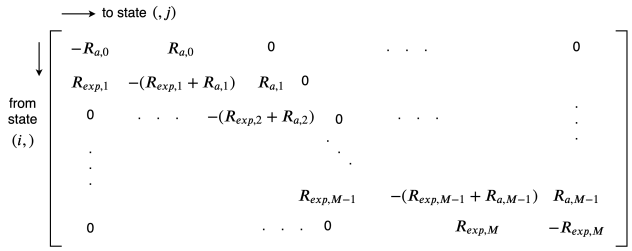

Figure 2. One-step transition rate matrix for the proposed model.

Figure 2 shows the one-step transition matrix Q used to calculate the state distribution π for the proposed SMP. In

this matrix, each element located in row i and column j shows the transition rate at which we transition from state i to state j. Diagonal elements are defined in a way to satisfy $Q_{i,i} = -\sum_{j \neq i} Q_{i,j}$.

To solve the Continuous-Time Markov Chain (CTMC) temporally, we have to solve the following equation:

$$\frac{d\pi}{dt} = \pi Q \Rightarrow \pi(t) = \pi(0)e^{Qt} \tag{18}$$

which can be calculated using the method proposed by Al-Mohy et al. [1] implemented in SciPy[1].

Using π, we can calculate the average number of instances in the warm pool C_w using $C_w = \sum_{m=0}^{M} m\pi_m$. We can also calculate the average number of servers running warm and cold requests in each state using $C_{r,w,m} = RT_w\lambda_{w,m}$ and $C_{r,c,m} = RT_c\lambda_{c,m}$:

$$C_r = \sum_{m=0}^{M} C_{r,w,m}\pi_m + \sum_{m=0}^{M-1} C_{r,c,m}\pi_m \tag{19}$$

We can also calculate the corresponding utilization of the deployed resources, indicating the fraction of time we are using the function instances in our pool:

$$U = \frac{C_{r,w}}{C_w} = \frac{RT_w\lambda(1 - P_{B,m})}{\sum_m m\pi_m} \tag{20}$$

4 Experimental Validation

To analyze the applicability of the proposed temporal performance model, we designed an experiment on AWS Lambda, measuring several performance metrics to compare against the model predictions. Throughout the experiment, we used an oracle predictor (ideal predictor) for the arrival rate changes in time with 1-minute granularity, making prediction 5 minutes into the future. Since the system has no predictions for the first 5 minutes of the experiment, we omitted them from the results. The designed experiment features a one-hour window that has been repeated 10 times to stabilize the results and used the sample average for each reported value. For the experiments, we leveraged the workload designed by Wang et al. [12] with minor modification in the presentation and the returned values publicly available on the project's Github Repository[2]. Our client-side requests caused the execution of a combination of CPU-intensive and I/O-intensive workloads on each function instance.

4.1 Experimental Setup

The workload used in this work has been deployed on AWS Lambda with 128 MB of RAM on the *us-east-1* region. The client machine issuing requests was a VM with 8 vCPUs, 16GB of RAM, and 1000 Mbps connectivity on *removed to keep the authors anonymous* cloud with single-digit milliseconds latency to AWS data center. We leveraged the official *boto3* library to interface with the AWS API making the request directly to the system (instead of using API gateway) to better analyze the behaviour of the AWS Lambda itself. The code performing the experiments, as well as the analysis, is publicly available on the project's Github repository[3].

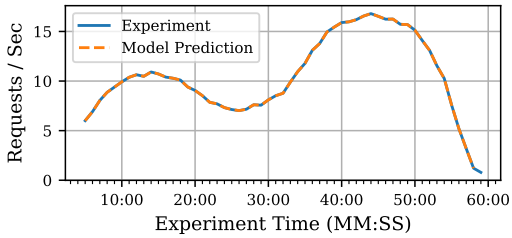

Figure 3. Request arrival rate throughout the experiment, along with the oracle predictions.

4.2 Exogenous Parameters

To be able to perform experiments on a given serverless computing system, we first need to extract system parameters to be fed into the performance model as exogenous parameters. Figure 3 shows the request arrival rate over time designed for this experiment. Since workload prediction is out of the scope of this paper, we used an oracle request rate predictor for our experiments. Another exogenous parameter for the proposed model is the *expiration timeout*, which has been set to 10 minutes for AWS Lambda, also verified by Shikov [9] and Shahrad et al. [10]. The average cold and warm response time (RT_c and RT_w) are set to the average of all requests over all experiment repetitions, which doesn't change over time in modern serverless computing platforms.

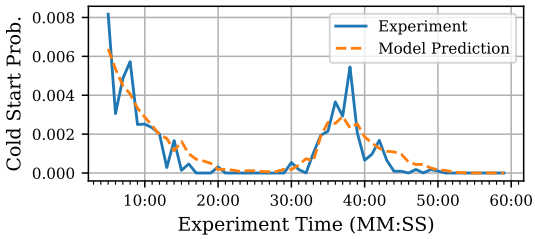

Figure 4. Probability of a cold start occurrence throughout the experiment along with the model predictions.

4.3 Experimental Results

Figure 4 shows the average probability of a cold start over all of the performance experiments. This metric is the most important factor in deciding the quality of service observed by

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.expm.html last accessed Sep-01-2020.

[2] https://github.com/pacslab/serverless-temporal-perf-modeling

[3] https://github.com/pacslab/serverless-temporal-perf-modeling

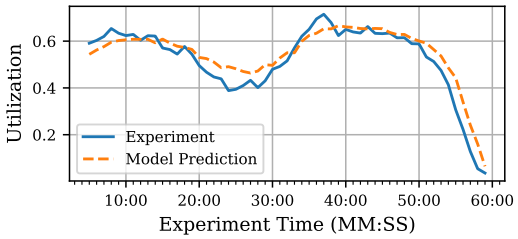

Figure 5. Utilization of resources in the warm pool throughout the experiment compared with the model predictions.

the application user. Since cold starts could be orders of magnitude longer than warm starts, having a large probability of cold start could affect the user experience. In many applications (especially customer-facing applications), it is important to limit the probability of a cold start. Thus, predicting this value is very important to enable migration to serverless computing platforms for many different applications. Figure 5 depicts the average utilization of resources over time in our experiments. Assuming a serverless provider is using some kind of Infrastructure-as-a-Service (IaaS) platform underneath, the number of function instances is proportional to the operational costs incurred by the platform. In this work, the utilization of resources is defined as the average ratio of the time that these instances are being utilized and thus are billed for the application developer. To lower the costs, serverless platforms would want to maximize the average utilization. Using the proposed model, the platform can make predictions for the cost-performance trade-off and decide how to take action accordingly.

5 Conclusion and Future Work

In this paper, we proposed and experimentally validated a temporal analytical performance model for modern serverless computing platforms. The proposed model can predict several key performance indicators needing only request arrival rate, average cold and warm response time, and the system expiration timeout. In the proposed model, we decided to model serverless computing platforms with the scale-per-request autoscaling pattern due to its dominance in mainstream public serverless computing platforms. The presented performance model can be used by application developers to improve the quality of their services by predicting performance metrics and take action accordingly. Serverless providers could also leverage the proposed model to improve their management policy and make their operations smarter and predictive. Several other optimizations (e.g. energy efficiency, response time, cost) are also possible. In summary, the proposed model can help make the serverless computing platforms "workload-aware".

In the future, we plan to build an autonomous middleware for the optimization of a given workload using the proposed performance model. We also aim to extend the model to allow preemptive workload handling, heterogeneous function instances, and a combination of IaaS and FaaS. We also plan to present a performance model for other autoscaling patterns in serverless computing, e.g., Google Cloud Run.

Acknowledgments

This research was enabled in part by support from Sharcnet[4] and Compute Canada[5]. We would also like to thank Amazon for supporting this research by providing us with the education credit to access the Amazon Web Services (AWS).

References

[1] Awad H Al-Mohy and Nicholas J Higham. 2010. A New Scaling and Squaring Algorithm for the Matrix Exponential. *SIAM J. Matrix Anal. Appl.* 31, 3 (2010), 970–989.

[2] Laurence A Baxter. 1992. Probability, statistics, and queueing theory with computer sciences applications.

[3] Diogo Bortolini and Rafael R Obelheiro. 2019. Investigating Performance and Cost in Function-as-a-Service Platforms. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. Springer, 174–185.

[4] SK Bose. 2001. M/G/m/m Loss System. *Prieiga per internet: http://www.iitg. ac. in/skbose/qbook/MGmm_Queue. PDF* (2001).

[5] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance Evaluation of Heterogeneous Cloud Functions. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4792.

[6] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

[7] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).

[8] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 159–169.

[9] Mikhail Shilkov. 2020. Cold Starts in AWS Lambda. https://mikhail.io/serverless/coldstarts/aws/ Last accessed 2020-03-18.

[10] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. *arXiv preprint arXiv:2003.03423* (2020).

[11] Erwin Van Eyk, Alexandru Iosup, Cristina L Abad, Johannes Grohmann, and Simon Eismann. 2018. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 21–24.

[12] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[4] https://www.sharcnet.ca
[5] https://www.computecanada.ca