Performance Modeling of Serverless Computing Platforms

Nima Mahmoudi, *Graduate Student Member, IEEE,* and Hamzeh Khazaei, *Member, IEEE*

Abstract—Analytical performance models have been leveraged extensively to analyze and improve the performance and cost of various cloud computing services. However, in the case of serverless computing, which is projected to be the dominant form of cloud computing in the future, we have not seen analytical performance models to help with the analysis and optimization of such platforms. In this work, we propose an analytical performance model that captures the unique details of serverless computing platforms. The model can be leveraged to improve the quality of service and resource utilization and reduce the operational cost of serverless platforms. Also, the proposed performance model provides a framework that enables serverless platforms to become *workload-aware* and operate differently for different workloads to provide a better trade-off between the cost and performance depending on the user's preferences. The current serverless offerings require the user to have extensive knowledge of the internals of the platform to perform efficient deployments. Using the proposed analytical model, the provider can simplify the deployment process by calculating the performance metrics for users even before physical deployments. We validate the applicability and accuracy of the proposed model by extensive experimentation on AWS Lambda. We show that the proposed model can calculate essential performance metrics such as average response time, probability of cold start, and the average number of function instances in the steady-state. Also, we show how the performance model can be used to tune the serverless platform for each workload, which will result in better performance or lower cost without scarifying the other. The presented model assumes no non-realistic restrictions, so that it offers a high degree of fidelity while maintaining tractability at large scale.

Index Terms—Serverless Computing, Performance Modeling, Optimization, Queuing Theory, Stochastic Processes.

1 Introduction

Serverless computing platforms handle almost every aspect of the system administration tasks needed to deploy a workload on the cloud. They provide users[1] with several potential benefits like handling all of the system administration operations and improving resource utilization, leading to potential operational cost savings, improved energy efficiency, and more straightforward application development [1], [2]. Although cloud functions have a much faster startup (and thus scaling) than traditional VM-based instances, they still show unpredictability in their key performance metrics. This has proven to be unacceptable for many customer-facing products [2].

Current serverless computing offerings are not workload-aware and use the same policies for all functions [3], [4], [5]. This leaves us with untapped potential for savings in infrastructure costs incurred by the provider, energy consumption, and improvements in performance by adapting the platform to different environments. Having an analytical performance model would help application developers as well as serverless operators to perform capacity planning and system study within a couple of seconds without the need for large-scale and expensive experiments. Thus, we decided to investigate the performance of the serverless computing platform and seek potential techniques that can be used to improve the performance. In addition to analyzing the performance of serverless computing platforms, we also identified a method that can be leveraged to enhance the performance of these systems.

Accurate performance modelling of serverless computing platforms can help ensure that the quality of service, performance metrics, and the cost of the workload remains within the acceptable range. It could also benefit providers to help them tune their management for each workload in order to reduce their infrastructure and energy costs [6].

The performance model used to address the performance-related issues in serverless computing platforms should prove to be tractable while covering a vast parameter space of the system. To the best of our knowledge, no such performance model has been introduced for the modern serverless computing platforms. In this work, we strive to develop and evaluate such a model. We used AWS Lambda as an example of a modern serverless computing platform to assess the proposed analytical model.

The analytical model presented in this work assumes a Poisson arrival process to address customer-facing open networks. This is mainly due to the fact that when the number of potential clients is high and each client submits requests with a low probability, the requests' arrival can be adequately modelled as a Poisson process [7]. We impose no restrictions on the service time by considering a generally distributed service time for functions. We consider

- *N. Mahmoudi is with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Alberta, Canada. E-mail: nmahmoud@ualberta.ca*
- *H. Khazaei is with the Department of Electrical Engineering and Computer Science, York University, Toronto, Ontario, Canada. E-mail: hkh@yorku.ca*

Manuscript received April x, 2020; revised August xx, 2020. DOI (identifier): 10.1109/TCC.2020.3033373

1. we use the terms users and application developers interchangeably.

serverless computing platforms with the scale-per-request method of autoscaling utilized by public platforms such as AWS Lambda, Google Cloud Functions, Azure Functions, IBM Cloud Functions, and Apache OpenWhisk [3], [4]. This autoscaling has no queuing involved in the platform, i.e. the request will be serviced by a warm instance of the function[2] if there is any available, or the platform will spin up a new function instance to serve the request. The presented model in this work is highly scalable and can handle a high degree of parallelization common in serverless computing platforms. The model predicts the main system characteristics that can later be used to find important performance metrics of the system. Our analytical model is a tool that can be utilized by both serverless computing platform providers and application developers to predict steady-state performance metrics of the deployed applications. This helps developers decide if a developed workload would comply with their QoS agreements, and if not, how much performance improvement they would need to do so. The performance improvement decided could be achieved either by improving the design, quality of code, or by simply resizing the resource allocated to each instance, which is usually set by changing the allocated memory. Besides, the proposed performance model could be leveraged by the providers by providing developers with more control over the cost-performance trade-off, which is not the same for all functions deployed on a platform. In addition, adaptive platform management could lower infrastructure costs incurred by the provider, enabling them to lower their prices, making their offerings more appealing.

Although the underlying system characteristics are not directly measurable in publicly available serverless computing platforms, we deployed a specialized detective workload that makes it possible for us to extract these values. We validated the presented analytical model by experimentation showing the effectiveness of the model to capture the complexities arising in public serverless platforms.

The remainder of the paper is organized as follows: Section 2 describes the system represented by the analytical performance model proposed in this work. Section 3 outlines the proposed analytical model. In Section 4, we present the experimental validation of the proposed model. In Section 5, we survey the latest related work for serverless computing platforms. Section 6 summarizes our findings and concludes the paper.

2 System Description

There is very little official documentation made publicly available about the scheduling algorithms in public serverless computing platforms. However, a number of previous works have focused on partially reverse engineering this information through the way of experimentations on these platforms [3], [8], [9], [10]. Using the results of such researches and by modifying their code base and thorough experimentation, we have come to a good understanding of how modern serverless frameworks are operated and managed by the service providers. In this work, we plan to use this information to build a tractable, yet accurate, performance model for modern serverless computing platforms.

In serverless computing platforms, computation is done in function instances. These instances are completely managed by the serverless computing platform provider and act as tiny servers for the incoming triggers (requests). To develop a comprehensive analytical performance model for serverless computing platforms, we first need to understand how they work and are managed.

Function Instance States: using the findings of previous studies [3], [8], [11], we identify three states for each function instance: *initializing*, *running*, and *idle*. The *initializing* state happens when the infrastructure is spinning up new instances, which might include setting up new virtual machines, unikernels, or containers to handle the excessive workload. The instance will remain in the *initializing* state until it is able to handle incoming requests. As defined in this work, we also consider *application initializing* which is the time user's code is performing initial tasks like creating database connections, importing libraries or loading a machine learning model from an S3 bucket as a part of the *initializing* state which needs to happen only once for each new instance. Note that the instance cannot accept incoming requests before performing all initialization tasks. It might be worth noting that the *application initializing* state is billed by most providers while the rest of the *initializing* state is not billed. When a request is submitted to the instance, the instance goes into the *running* state. In this state, the request is parsed and processed. The time spent in the *running* state is also billed by the serverless provider. After processing of a request is over, the serverless platform keeps the instances warm for some time to be able to handle later spikes in the workload. In this state, we consider the instance to be in the *idle* state. The user is not charged for an instance that is in the *idle* state.

Cold/Warm start: as defined in previous work [3], [8], [10], we refer to *cold start* request when the request goes through the process of launching a new function instance. For the platform, this could include launching a new virtual machine, deploying a new function, or creating a new instance on an existing virtual machine, which introduces an overhead to the response time experienced by users. In case the platform has an instance in the *idle* state when a new request arrives, it will reuse the existing function instance instead of spinning up a new one. This is commonly known as a *warm start* request. Cold starts could be orders of magnitude longer than warm starts for some applications. Thus, too many cold starts could impact the application's responsiveness and user experience [3]. This is the reason a lot of research in the field of serverless computing has focused on mitigating cold starts [12], [13], [14].

Autoscaling: we have identified three main autoscaling patterns among the mainstream serverless computing platforms: 1) *scale-per-request*; 2) *concurrency value scaling*; 3) *metrics-based scaling*. In *scale-per-request* Function-as-a-Service (FaaS) platforms, when a request comes in, it will be serviced by one of the available idle instances (*warm start*), or the platform will spin up a new instance for that request (*cold start*). Thus, there is no queuing involved in the system, and each cold start causes the creation of a new instance which acts as a tiny server for subsequent

2. in this work, we use server, instance, container, and function interchangeably.

requests. As the load decreases, to scale the number of instances down, the platform also needs to scale the number of instances down. In the *scale-per-request* pattern, as long as requests that are being made to the instance are less than the *expiration threshold* apart, the instance will be kept warm. In other words, for each instance, at any moment in time, if a request has not been received in the last *expiration threshold* units of time, it will be expired and thus terminated by the platform, and the consumed resources will be released. To enable simplified billing, most well-known public serverless computing platforms use this scaling pattern, e.g., AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Apache OpenWhisk, and Azure Functions [3], [5]. As scale-per-request is the dominant scaling technique used by major providers, in this paper, we strive to analytically model this type of serverless platform.

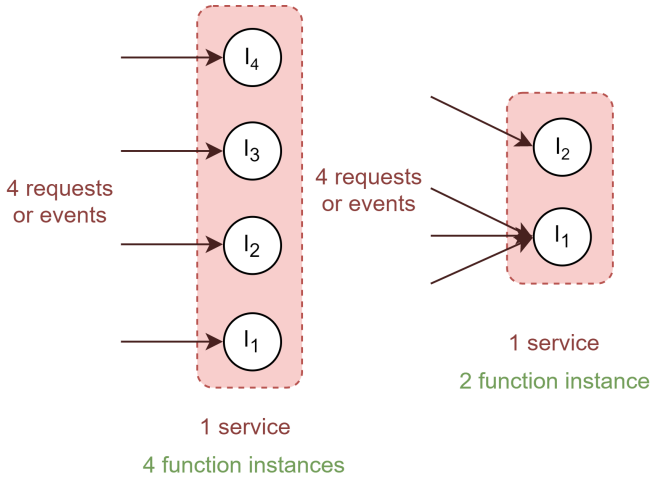

Fig. 1. The effect of the concurrency value on the number of function instances needed. The left service allows a maximum of 1 request per instance, while the right service allows a concurrency value of 3.

In the *concurrency value scaling* pattern [15], function instances can receive multiple requests at the same time. The average and maximum number of requests that can be made concurrently to the same instance can be set via hard and soft limits on the *concurrency value*. Figure 1 shows the effect of hard limit of concurrency value on the autoscaling behaviour of the platform. It is worth noting that the scale-per-request autoscaling pattern might initially appear as a special case of *concurrency value scaling* pattern where concurrency value is set to 1, however, there are fundamental differences between these two autoscaling patterns that led us to classify them into different categories. First, the current generations of concurrency value autoscaling platforms allow for queuing of requests in a shared queue, however there is no queuing involved in scale-per-request autoscaling. In scale-per-request autoscaling pattern, at the time of arrival, an incoming request will either be assigned to an idle instance (warm), or a newly instantiated instance (cold), even if an instance in the warm pool becomes idle while the cold instance is still being instantiated. However, concurrency value autoscaling platforms allow queuing of requests while new instances are being instantiated and allows routing of the requests to an instance only after it has done the initialization and is ready to serve new requests. In addition, autoscaling in scale-per-request is *synchronous* to request arrivals where the creation of an instance (scaling out) happens on arrival of new requests. However, in platforms like Google Cloud Run and Knative which use concurrency value autoscaling, new instances are created *asynchronously* on fixed intervals, e.g. 2 seconds in Knative, and using evaluations of the average of measured concurrency in stable and panic windows [16].

Metrics-based scaling tries to keep metrics like CPU or memory usage within a predefined range. Most on-premises serverless computing platforms work with this pattern due to its simplicity and reliability. Some of the serverless computing platforms that use this pattern are AWS Fargate, Azure Container Instances, OpenFaaS, Kubeless, and Fission.

The analytical model proposed in this work considers only the platforms that use the scale-per-request pattern due to their importance and widespread adoption in mainstream public serverless computing platforms.

Initialization Time: as mentioned earlier, when the platform is spinning up new instances, they will first go into the initialization state. The initialization time is the amount of time it takes since the platform receives a request until the new instance is up and running, and ready to serve the request. The initialization time, as defined here, is comprised of the platform initialization time and the application initialization time. The platform initialization time is the time it takes for the platform to make the function instance ready, whether a unikernel or a container, and the application initialization time is the time it takes for the application to run the initialization code, e.g., connecting to the database.

Response Time: the response time usually includes the queuing time and the service time. Since we are addressing the scale-per-request serverless computing platforms here, there is no queuing involved for the incoming requests. Due to the inherent linear scalability in serverless computing platforms [3], [8], [10], the distribution of the response time does not change over time with different loads. Therefore, we leveraged delay centers [17] in order to analytically model the response time in serverless computing platforms.

Maximum Concurrency Level: every public serverless computing platform has some limitation on the number of function instances that can be spun up and in running state for a single function. This is mainly due to ensuring the availability of the service for others, limiting the number of instances one user can have up and running at the same time. This is mostly known as the *maximum concurrency level*. For example, the default maximum concurrency level for AWS Lambda is 1000 function instances in 2020. When the system reaches the maximum concurrency level, any request that needs to be served by a new instance will receive an error status showing the server is not able to fulfill that request at the moment.

Request Routing: in order to minimize the number of containers that are kept warm and thus to free up system resources, the platform routes requests to new containers, and it will use older containers only if all containers that are created more recently are busy [18]. In other words, the scheduler gives priority to newly instantiated idle instances using priority scheduling according to creation time, i.e.,

TABLE 1
Symbols and their corresponding descriptions.

Symbol	Description
λ	Mean arrival rate of requests
μ_w	Mean warm start service rate
μ_c	Mean cold start service rate
ρ	Offered load
$P_{B,m}$	Blocking probability for a warm pool of m instances
$\lambda_{w,m}$	Actual arrival rate to the warm pool of m instances
$\lambda_{c,m}$	Cold start arrival rate when we have m warm instances
$\lambda_{w,m,i}$	Actual arrival rate to ith instance in the warm pool of m instances
I_i	The ith instance in the warm pool
$P_{S,n}$	Probability of a request being served by the nth instance in the warm pool
$\lambda_{w,m,i}$	The arrival rate for warm instance I_i in a warm pool of m instances
$C_{req,m,i}$	Mean number of requests served by instance i in a warm pool of m instances before being terminated
$P_{lst,m,i}$	Probability of a request being the last one before instance termination
$LS_{m,i}$	Lifespan of the ith server in a warm pool of m instances
$R_{exp,m,i}$	The mean expiration rate of the ith server in a warm pool of m instances
$R_{exp,m}$	The mean total expiration rate in a warm pool of m instances
$R_{a,m}$	Mean transition rate of going from m to $m+1$ servers in the warm pool
Q	The transition rate matrix
π	The steady-state distribution
P_{rej}	Probability of rejection by the system
P_B	Probability of blocking by the warm pool
P_{cld}	Probability of cold start
RT_{avg}	Mean response time
RT_w	Mean warm start response time
RT_c	Mean cold start response time
C_w	The mean number of servers in the warm pool
C_r	Mean number of running instances
$C_{r,w}$	The mean number of servers busy running warm requests
$C_{r,w,m}$	The mean number of servers busy running warm requests in a warm pool of size m
$C_{r,c}$	The mean number of servers busy running cold requests
$C_{r,c,m}$	The mean number of servers busy running cold requests when the warm pool is of size m
C_i	The mean number of idle servers
U	Mean utilization

the newer the instance, the higher the priority. By adopting this approach, the system minimizes the number of requests going to older containers, maximizing their chance of being expired and terminated.

3 Analytical Modelling

Section 2 briefly outlines the scheduling algorithm used for the serverless computing platforms and the one that we consider in this work, i.e., scale-per-request. In this section, we present our analytical performance model based on this scheduling algorithm. Our primary focus is to obtain steady-state metrics of the system based on the system and workload characteristics.

An ideal serverless computing platform should act like an $M/G/\infty$ queuing system (aka delay center) with the same service time distribution for all requests. However, in current serverless computing platforms, the presence of cold start, which could be orders of magnitude longer than a warm start, and limitations on the concurrent number of instances (i.e., servers), shown as *maximum concurrency level*, lead to a more complex performance model. In this work, we impose more restrictions on delay center theory to accurately model the current serverless computing platforms with a high degree of fidelity and tractability.

In the presented model, we leveraged a continuous-time Semi-Markov Process (SMP) where the state number represents the number of instances in the warm instance pool, which is between 0 and *maximum concurrency level*. As shown in Figure 2, adding an instance to the warm instance pool is triggered by a cold start, causing a transition from state i to $i+1$ in our SMP model. In the proposed model, each server is terminated and released after being idle for some time. To calculate the associated transition rates, we model each state of the SMP with an $M/G/m/m$ queuing system. The number of instances (m) can shrink (to the minimum of zero instances in the warm pool) or expand (to the maximum concurrency level) due to the fluctuation in the workload. $M/G/m/m$ queuing systems are appropriate for modelling the warm instance pool since servers are homogeneous, the discipline is non-preemptive FCFS, and there is no priority among incoming requests. Thus, we assume a Poisson arrival process, generally distributed service times, with m warm instances and no extra queuing room beside the server instances. In the following subsections, we present the calculation of different parameters in our analytical model using symbols defined in Table 1.

3.1 Cold Start Rate

As can be seen in Fig. 2, rejection of a request by the warm pool triggers a cold start and thus adds a new function instance to the warm pool to handle subsequent requests. To obtain the rate at which new servers will be instantiated, we need to calculate the probability of a request being rejected by the warm pool. We know that the state probabilities of the *M/G/m/m* loss system are identical to the corresponding Markovian $M/M/m/m$ system with exponentially distributed service times [19]. To calculate the blocking probability for the corresponding $M/M/m/m$ loss system, first, we need to calculate the offered load (ρ) in terms of the arrival rate (λ) and the average service rate (μ_w):

$$\rho = \lambda/\mu_w \tag{1}$$

Then, the Erlang's B formula is obtained as [20]:

$$P_{B,m} = B(m,\rho) = \frac{\frac{\rho^m}{m!}}{\sum_{j=0}^{m} \frac{\rho^j}{j!}} \tag{2}$$

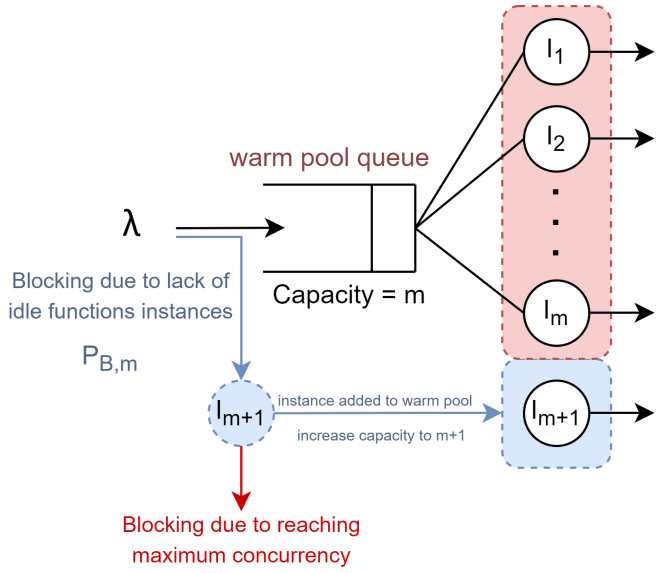

Fig. 2. An overview of the proposed system model using *M/G/m/m* loss systems. In the case of workload fluctuation, m will change during the runtime, just like a delay center. The blue arrows show the path a successful cold start goes through.

This equation gives the probability of a request being rejected (blocked) by the warm pool, assuming there are m warm servers. If m is less than the maximum concurrency level, the request blocked by the warm pool causes a cold start. If the warm pool has reached the maximum concurrency level, any request rejected by the warm pool will be rejected by the platform. We can also calculate the actual arrival rate to the warm pool of m instances ($\lambda_{w,m}$) which is less than λ since some arrivals are being rejected by the warm pool:

$$\lambda_{w,m} = \lambda(1 - P_{B,m}) \tag{3}$$

Using eq. (2), we can derive the rate at which cold starts are happening in the system.

$$\lambda_{c,m} = \lambda P_{B,m} \tag{4}$$

Figure 2 depicts an overview of the proposed model for the rapid scaling up in scale-per-request serverless computing platforms. Using this model, we can calculate the performance metrics of interest in the system.

3.2 Arrival Rate for each Server

To calculate the rate at which servers will be expired and consequently terminated, we first need to calculate the arrival rate for each warm instance. Assuming that we have m instances in warm pool as $\{I_1, I_2, ..., I_m\}$, $\lambda_{w,m,n}$ indicates the arrival rate to instance I_n, where $1 \leq n \leq m$. In our model, we assume the instance I_1 to be the newest server in the system and I_m to be the oldest instance in the system, thus considering the scheduling assumptions laid in Section 2, we can see that $\lambda_{w,m,1} > \lambda_{w,m,2} > ... > \lambda_{w,m,m}$ since the scheduler will first try to route the traffic to instance I_1, then I_2, and it will route traffic to I_m if and only if all other warm instances are currently busy running another request at the time of arrival.

When interpreting $P_{B,n-1}$, as defined in eq. (2), we see that it shows for what ratio of requests, instances $\{I_i; i = 1, 2, ..., n-1\}$ are busy in the warm pool. Thus, $P_{B,n-1}$ of the incoming requests, will either be served by $\{I_i; i = n, n+1, ..., m\}$, or be totally rejected by the system. Similarly, $P_{B,n}$ of the incoming requests, will be served by $\{I_i; i = n+1, n+2, ..., m\}$, or will be rejected by the system due to reaching the maximum capacity. Using these two observations, we can calculate the ratio of requests that are being processed by I_n as:

$$P_{S,n} = P_{B,n-1} - P_{B,n} \tag{5}$$

$P_{S,n}$ shows the probability of a request being served by instance I_n, having $P_{S,0} = 1$. Using this probability, we can calculate the arrival rate for each of instances $\{I_n; 1 \leq n \leq m\}$:

$$\lambda_{w,m,n} = \lambda_{w,m} P_{S,n} \tag{6}$$

3.3 Server Expiration Rate

In eq. (6), we calculated the arrival rate for individual instances in the warm pool. In this section, our goal is to calculate the mean lifespan of instances, considering that they will be expired and subsequently terminated after receiving no requests in *expiration threshold* units of time after processing the last request.

Let's assume the arrival rate $\lambda_{w,m,i}$ for instance I_i with exponential inter-arrival times. Thus, the Probability Density Function (PDF) of inter-arrival time is of the following form:

$$P(X = x) = \lambda_{w,m,i} \cdot e^{-\lambda_{w,m,i} x} \tag{7}$$

And the Cumulative Distribution Function (CDF) will be of the following form:

$$P(X \leq x) = 1 - e^{-\lambda_{w,m,i} x} \tag{8}$$

The probability that a request is the last one before the expiration of the server is equal to the probability that the next inter-arrival time drawn is larger than $T = T_{exp} + 1/\mu_w$, which is equal to:

$$P_{lst,m,i} = P(X \geq T) = e^{-\lambda_{w,m,i} T} \tag{9}$$

Thus, whether or not the request arriving at a server is the last one before the expiry of that server (shown as the *last request* in Figure 3) has a geometric distribution with the probability of $P_{lst,m,i}$ as the distribution parameter. We know that the average number of trials (i.e., arrival of requests) before the server is expired and terminated is:

$$C_{req,m,i} = \frac{1}{P_{lst,m,i}} \tag{10}$$

To see how long $C_{req,m,i}$ requests will keep the server warm, we need the expected inter-arrival time with an arrival rate of $\lambda_{w,m,i}$ which are less than $T = T_{exp} + 1/\mu_w$:

$$
\begin{aligned}
E[X; X < T] &= \int_0^T x\lambda_{w,m,i} e^{-\lambda_{w,m,i}x} dx \\
&= -x \cdot e^{-\lambda_{w,m,i}x}\Big|_0^T + \int_0^T e^{-\lambda_{w,m,i}x} dx \\
&= -T \cdot e^{-\lambda_{w,m,i}T} - \frac{e^{\lambda_{w,m,i}x}}{\lambda_{w,m,i}}\Big|_0^T \\
&= -T \cdot e^{-\lambda_{w,m,i}T} + \frac{1 - e^{\lambda_{w,m,i}T}}{\lambda_{w,m,i}}
\end{aligned} \tag{11}
$$

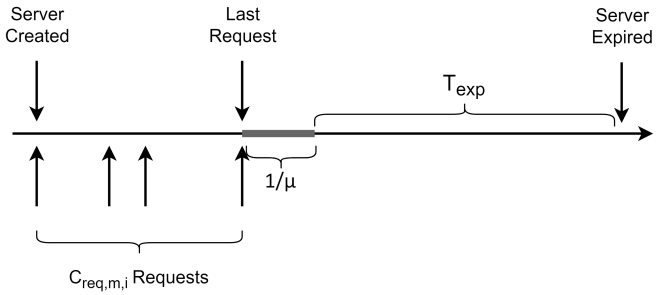

Fig. 3. The server lifespan calculation overview.

Thus, the average lifespan of a server in the warm pool could be calculated as follows (Figure 3):

$$
E[LS_{m,i}] = \{(C_{req,m,i} - 1) \cdot E[X; X < T]\} + \frac{1}{\mu_w} + T_{exp} \tag{12}
$$

where $E[LS_{m,i}]$ denotes the average lifespan of a server in the warm pool.

The expiration rate for servers can be calculated using $E[LS_{m,i}]$:

$$
R_{exp,m,i} = \frac{1}{E[LS_{m,i}]} \tag{13}
$$

which gives us the server expiration rate for I_i. Expiration and terminating any servers in the warm pool will result in having one less server in the pool. Thus, the overall expiration rate for m servers would be the sum of these rates:

$$
R_{exp,m} = \sum_{i=0}^{m} R_{exp,m,i} \tag{14}
$$

This gives the rate at which servers in a pool of m servers will be expired and terminated.

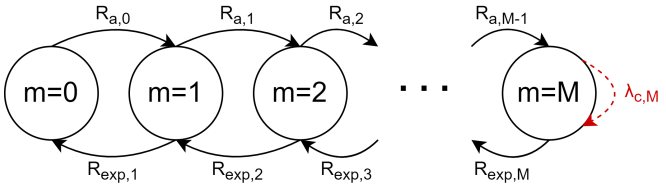

Fig. 4. The state transition diagram of the warm pool in serverless platforms. This is a Semi-Markov process for which we provide a closed-form steady-state solution. The dashed red self-loop shows rejected requests due to insufficient capacity.

3.4 Modelling the Warm Pool

In sections 3.1 to 3.3, we calculated the cold start and server expiration rates in the system. In this section, we model the warm servers pool using a Semi-Markov Process (SMP), for which we derive an exact closed-form steady-state solution. The process is not Markovian since, as can be seen in Figure 3, the lifespan of servers, i.e., the states' holding time, is not clearly exponentially distributed. Figure 4 shows the SMP model where M is the maximum number of servers in the warm pool, also known as *maximum concurrency level*, which is an inherent limitation in all public serverless offerings. In each state, m shows the number of servers in the warm pool. In other words, in each state the warm pool is working like a loss system (i.e., $M/G/m/m$ queue) that can go to another state, i.e., a loss system with one more or less function instance with the rate of $R_{a,m}$ and $R_{exp,m}$, respectively. $\lambda_{c,m}$ and $R_{exp,m}$ indicate the rate of cold start and expiration of a server in a warm server pool of size m, respectively. Also, μ_c is the rate of servicing a cold start request. $1/\lambda_{c,m}$ shows the mean time between two consecutive cold starts. But, when a cold start happens in the system, the server will not be available in the warm pool until the cold start service time has passed. This makes the transition rate of going from m to $m+1$ servers in the warm pool as:

$$
R_{a,m} = \frac{1}{\frac{1}{\lambda_{c,m}} + \frac{1}{\mu_c}} = \frac{\lambda_{c,m} \cdot \mu_c}{\lambda_{c,m} + \mu_c} \tag{15}
$$

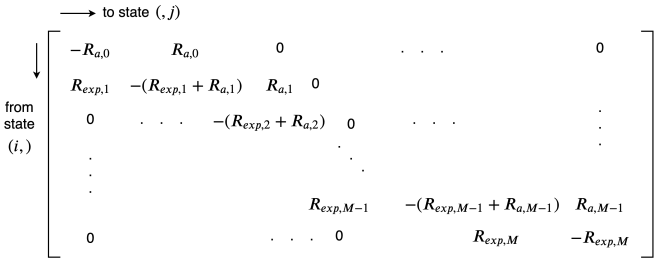

Fig. 5. One-step transition rate matrix for the proposed model.

The one-step transition rate matrix Q can be used to get the limiting distribution π for the SMP. The transition rate matrix used in this work is shown in Figure 5 where rows and columns correspond to the number of servers in the warm pool, starting with zero servers. Each element in the transition rate matrix located in row i and column j shows the rate at which the state transitions from state number i to state number j. Diagonal elements $Q_{i,i}$ are defined such that the following holds:

$$
Q_{i,i} = -\sum_{j \neq i} Q_{i,j} \tag{16}
$$

The steady-state distribution π is the unique solution to the following equation system [21]:

$$
\pi \cdot Q = 0 \text{ and } \sum_{m=0}^{M} \pi_m = 1 \tag{17}
$$

where π_m represents the probability of having m servers in steady-state. Algorithm 1 shows an overview of the proposed analytical model. As shown, after calculating the SMP model parameters for each number of function instances in the warm pool (for each m), we solve the SMP for the equilibrium distribution. Then, we can calculate the steady-state characteristics of interest in the system:

Algorithm 1: Serverless Performance Model Method

Input: λ, μ_w, μ_c, T_{exp}, M
Output: $metrics$

$\rho \leftarrow \lambda/\mu_w$;
$m \leftarrow 0$;
$props \leftarrow$ empty array;
$\lambda_c \leftarrow$ empty array;
$R_{exp} \leftarrow$ empty array;
while $m \leq M$ **do**
  $\lambda_c[m] \leftarrow$ calculate cold start rate;
  $R_{exp}[m] \leftarrow$ calculate expiration rate;
  $prop \leftarrow$ calculate properties for warm pool with m;
  $props[m] \leftarrow prop$;
  $m \leftarrow m + 1$;
end
$Q \leftarrow$ build_transition_rate_matrix(λ_c,μ_c,R_{exp});
$\pi_m \leftarrow$ solve the resulting SMP model using Q;
$metrics \leftarrow$calculate properties using $props$ and π_m;

Probability of Rejection (P_{rej}): as described in the system description, when the system reaches the maximum concurrency level, any request blocked by the warm pool will be rejected by the system. Thus, the probability of rejection for a given request can be calculated as the following:

$$P_{rej} = P_{B,M}\pi_M \tag{18}$$

Probability of Cold Start (P_{cld}): the probability of a cold start happening for each request is an important factor for several reasons, including complying with the Quality-of-Service (QoS) requirements. To calculate this metric, we first need the probability of a request being blocked by the warm pool:

$$P_B = \sum_{m=0}^{M} P_{B,m}\pi_m \tag{19}$$

Now, we can calculate the probability of cold start that may happen for each request, knowing each request blocked by the warm pool can either be a cold start or a rejected request:

$$P_{cld} = P_B - P_{rej} \tag{20}$$

Average Response Time (RT_{avg}): the derivation of the average response time is:

$$RT_{avg} = RT_w(1 - P_B) + RT_c P_{cld} \tag{21}$$

where RT_{avg}, RT_w, and RT_c denote the total average response time and average response time for cold and warm requests in steady-state, respectively. Also, note that $\mu_w = 1/RT_w$ and $\mu_c = 1/RT_c$.

Mean Number of Instances in Warm Pool (C_w): knowing the average number of instances in the warm pool could benefit both the service providers and the users of the serverless computing platform. Users could use this information to set the *provisioned* or *reserved* concurrency levels [22]. Service providers could use this information to modify their system-level settings based on the characteristics of each workload.

The average number of servers in the warm pool C_w can be calculated using π_m since m represents the number of servers in each state:

$$C_w = \sum_{m=0}^{M} m\pi_m \tag{22}$$

Mean Number of Running Instances (C_r): the average number of servers busy running warm requests ($C_{r,w}$) can be calculated using the following:

$$\begin{aligned} C_{r,w,m} &= RT_w\lambda_{w,m} = RT_w\lambda(1 - P_{B,m}) \\ C_{r,w} &= \sum_{m=0}^{M} C_{r,w,m}\pi_m \end{aligned} \tag{23}$$

Similarly, we can calculate the average number of servers busy running cold requests ($C_{r,c}$), considering the fact that requests blocked by the warm pool when reaching maximum concurrency level are rejected, and thus do not count towards the running cold starts:

$$\begin{aligned} C_{r,c,m} &= \begin{cases} 0 & \text{if } m = M \\ RT_c\lambda_{c,m} = RT_c\lambda P_{B,m} & \text{otherwise} \end{cases} \\ C_{r,c} &= \sum_{m=0}^{M-1} C_{r,c,m}\pi_m \end{aligned} \tag{24}$$

Thus, the average number of servers processing user requests could be calculated:

$$C_r = C_{r,w} + C_{r,c} \tag{25}$$

Mean Number of Idle Servers (C_i): as mentioned earlier, the number of idle servers is proportional to the infrastructure overhead of the service provider. This property can be calculated as follows:

$$C_i = C_w - C_{r,w} \tag{26}$$

This equation is derived using the fact that warm instances are either in the *idle* state, meaning they are not processing any requests and are just reserved capacity, or they are in the *busy* state, meaning they are processing a request.

Mean Utilization (U): in this context, the utilization is defined as the ratio of warm instances that are busy processing a request ($C_{r,w}$) over the total instances in the warm pool (C_w). Knowing the average number of running instances, and the average number of instances in the warm pool, we can calculate the average utilization ratio:

$$U = \frac{C_{r,w}}{C_w} = \frac{RT_w\lambda(1 - P_{B,m})}{\sum_m m\pi_m} \tag{27}$$

The utilization metric is especially of importance for service providers since they only charge users for instances that are processing user requests, and thus the rest of the capacity is considered additional costs for them.

3.5 Tractability Analysis

To study the tractability, i.e., scalability of our performance model, we investigate how the complexity of the proposed model grows when various parameters are increased. The number of states in the final Semi-Markov Process model is equal to the maximum concurrency level of the system and grows linearly when increasing the maximum concurrency level. The rate calculations for the SMP model should also prove to be tractable. Using the method outlined in Algorithm 1, we can calculate the time complexity of the analytical model. The expiration rate calculations can be calculated for each state in $O(1)$. Thus, the calculation of expiration rates for the final model grows linearly with the maximum concurrency level. The cold start rate calculation requires the calculation of the Erlang formula, which grows linearly with the number of servers in the state (m). Hence the calculation of all cold start rates can be done in $O(M^2)$, which can be calculated for any scale. Solving the resulting SMP for equilibrium distribution is done in $O(M^3)$, which makes the complexity of the whole process $O(M^3)$.

4 Experimental Validation

In this section, we evaluate our analytical model by way of experimentations on the AWS Lambda serverless platform. All of our experiments were executed for a 28-hour window with 10 minutes of warm-up time in the beginning, during which we don't record any data.

4.1 Experimental Setup

In our AWS Lambda deployment, we used the *Python 3.6* runtime with 128 MB of RAM deployed on *us-east-1* region in order to have the lowest possible latency from our client machine. The workload used in this work was based on the work of Wang et al. [3] with minor modifications and is openly available in our Github repository[3]. For the purpose of experimental validation, we used a combination of CPU intensive and I/O intensive workloads. As the CPU intensive part, the function calculates the multiplication of 1 through $10,000$. The I/O intensive part of the workload includes using dd tool[4] to read and write a file of size 1MB, 5 times for each incoming request. During the experimentation, we have obtained performance metrics and the other parameters such as cold/warm start information, instance id, lifespan, etc., which have been used to guide our analysis.

For the client triggering the deployed function, we used a virtual machine hosted on Compute Canada Arbutus cloud[5] with 8 vCPUs, 16 GB of memory, and 1000 Mbps network connectivity with single-digit milliseconds latency to AWS servers. We used Python for the client's programming language, and the official *boto3* library to communicate with the AWS Lambda API to make the requests (trigger the function) and process the resulting logs for each request with a request-reply pattern. Note that we have not used any intermediary interfaces like AWS Gateway, S3 storage, or message queues to mitigate the effect of their performance fluctuations in our measurements. For load-testing and generating client requests based on a Poisson process, we used our in-house function triggering library[6] which is openly accessible through PyPi[7]. The result is stored in a CSV file and then processed using Pandas, Numpy, Matplotlib, and Seaborn. The dataset, parser, and the code for extraction of system parameters and properties are also publicly available in the project's Github repository[8].

To further improve the reproducibility of our work, we also included a docker image containing the execution runtime of our work which has the required libraries (including our own) pre-installed and ready for use by the research community.

4.2 Parameter Identification

We need to estimate the system characteristics to be used in our model as exogenous parameters. In this section, we discuss our approach to estimating each of these parameters.

Expiration Threshold (T_{exp}): here, our goal is to measure the expiration threshold, which is the amount of time after which inactive servers in the warm pool will be expired and therefore terminated. To measure this parameter, we created an experiment in which we make requests with increasing inter-arrival times until we see a cold start meaning that the system has terminated the server between two consecutive requests. We performed this experiment on AWS lambda with the starting inter-arrival time of 10 seconds, each time increasing it by 10 seconds until we see a cold start. In our experiments, AWS lambda seemed to expire a server exactly after 10 minutes of inactivity (after it has processed its last request). This number did not change in any of our experiments leading us to assume it is a deterministic value. This observation has also been verified in [4], [23].

Average Warm Response Time (RT_w) and Average Cold Response Time (RT_c): to measure the average warm response time and the average cold response time, we used the average of response times measured throughout the experiment.

4.3 Analytical Model Validation

In this section, we outline our methodology for measuring the performance metrics of the system, comparing the results with the predictions of our analytical model.

Probability of Cold Start (P_{cld}): to measure the probability of cold start, we divide the number of requests causing a cold start by the total number of requests made during our experiment. Due to the inherent scarcity of cold starts in most of our experiments, we observed an increased noise in our measurements for the probability of cold start, which lead to us increasing the window for data collection to about 28 hours for each sampled point.

Mean Number of Instances in the Warm Pool (C_w): to measure the mean number of instances in the warm pool, we count the number of unique instances that have responded to the client's requests in the past 10 minutes. We use a

3. https://github.com/pacslab/serverless-performance-modeling/tree/master/deployments
4. https://man7.org/linux/man-pages/man1/dd.1.html
5. https://docs.computecanada.ca/wiki/Cloud_resources
6. https://github.com/pacslab/pacswg
7. https://pypi.org/project/pacswg
8. https://github.com/pacslab/serverless-performance-modeling

unique identifier for each function instance to keep track of their life cycle, as obtained in [3].

Mean Number of Running Instances ($C_{r,w}$): we calculate this metric by observing the system every ten seconds, counting the number of in-flight requests in the system, taking the average as our estimate.

Mean Number of Idle Instances (C_i): this can be measured as the difference between the total average number of instances in the warm pool and the number of instances busy running the requests.

Utilization (U): similar to our model, this is defined as:

$$U = \frac{C_{r,w}}{C_w} \tag{28}$$

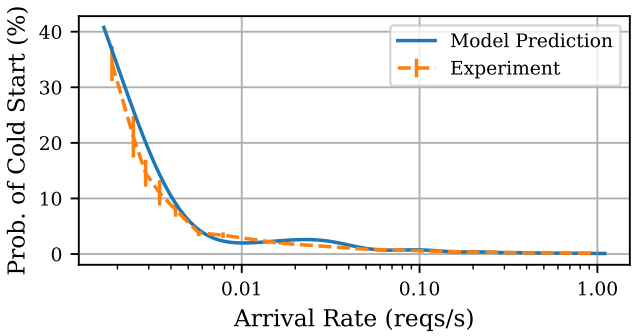

Fig. 6. Probability of cold start against arrival rate. The vertical bars show one standard error around the measured point.

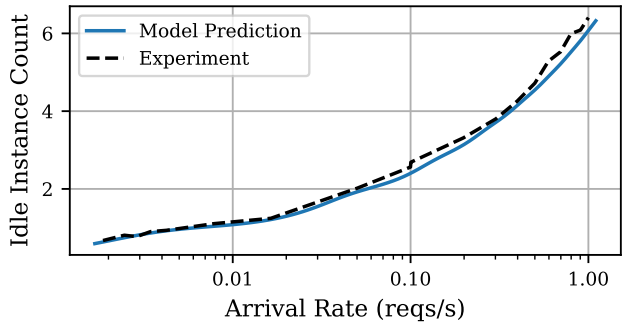

Fig. 7. The number of idle servers against arrival rate.

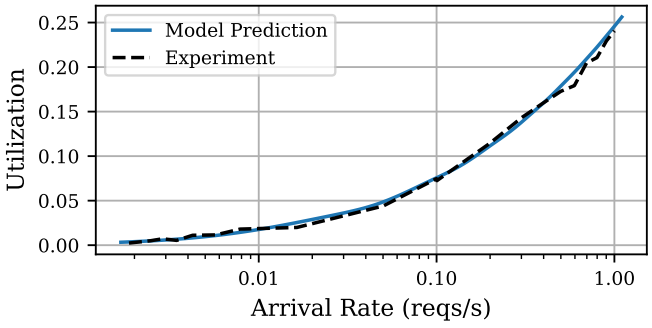

Fig. 8. Utilization against arrival rate.

4.4 Experimental Results

Figures 6 to 8 show the result of our experiments compared with the analytical model results. For each point shown for the experimentation, we ran a test with a Poisson arrival rate with a constant mean for twenty-eight hours with ten minutes of warm-up in the beginning. As can be seen, the analytical performance model results are greatly in tune with the experimental results.

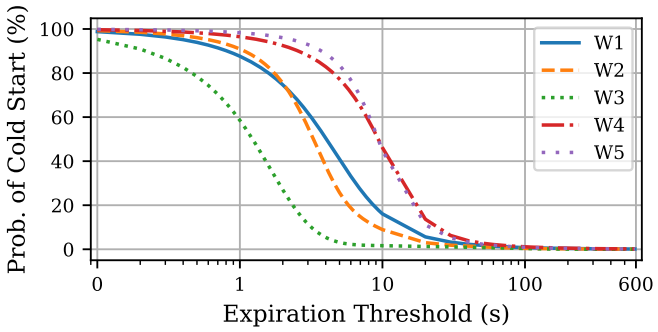

Fig. 9. Cold start probability against the *expiration threshold*. The arrival rate has been set to 1 request per second. The legends denote warm and cold service times. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

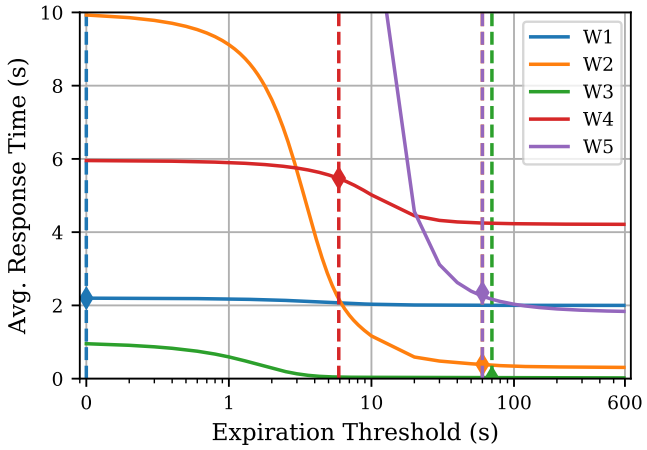

Fig. 10. Average response time against the *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds. The vertical lines show the minimum expiration threshold for which the average response time is at most 30% higher than the average warm start response time.

4.5 Discussion

Section 4.4 outlined the experimental results and their comparison with the analytical performance model. As discussed earlier, these results show the effectiveness, tractability, and fidelity of the model when applied to AWS Lambda [24]. The model proposed in this work can be applied to any serverless computing platform, as long as the management complies with the system description outlined in Section 2. The most important criterion is scale-per-request behaviour (with no queuing). For example, Google

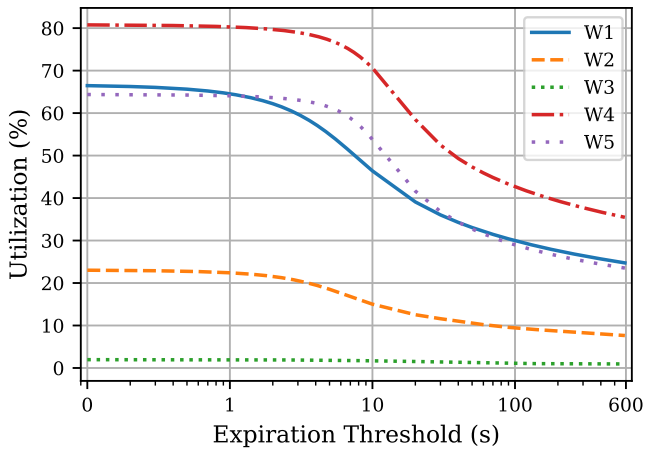

Fig. 11. Utilization against the *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

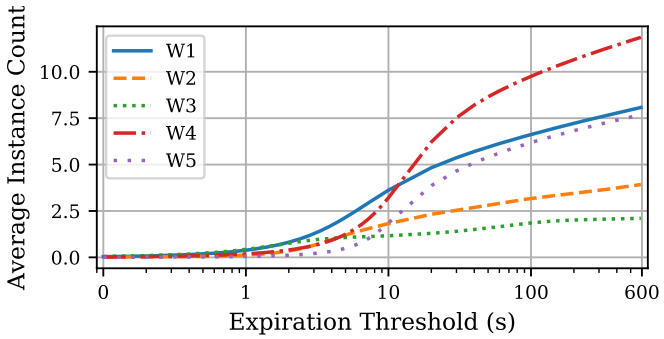

Fig. 12. Average instance count against the *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

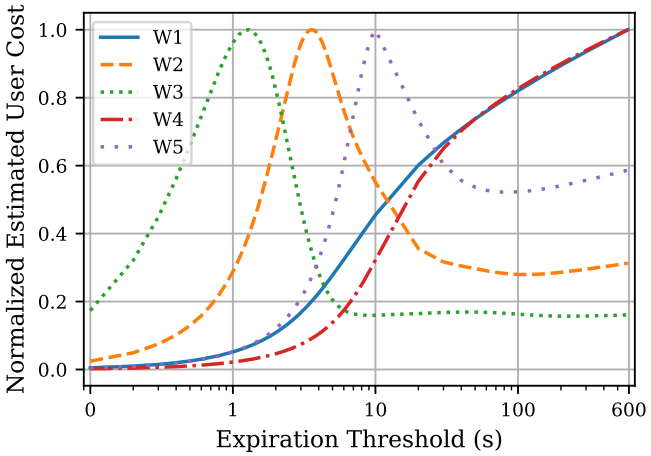

Fig. 13. Average estimated user cost against *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

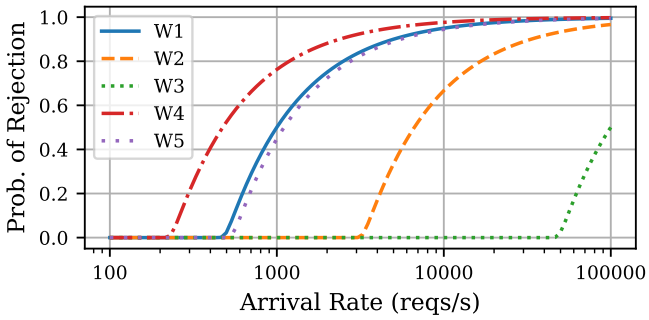

Fig. 14. Probability of rejection against the arrival rate. The expiration threshold has been set to 10 minutes, and the maximum concurrency is 1000. Note that the x-axis is on a logarithmic scale.

Cloud Functions [25], Azure Functions [26], IBM Cloud Functions [27], and Apache OpenWhisk [28] work in a similar fashion, but Google Cloud Run [29], OpenFaaS [30], Kubeless [31], and Fission [32] allow queuing for each server which renders them incompatible with the performance model presented in this work.

In this section, we leverage the presented analytical model to perform what-if analysis and investigate the effect of changing configurations on service quality metrics and infrastructure cost indicators. It is worth mentioning that the analysis presented here have been generated instantly and at no cost using the performance model, which signifies the benefits of a tractable and accurate analytical model.

As mentioned earlier in Section 1, current serverless computing offerings are oblivious to the type of workload that is being executed on them. One way to tune the serverless computing platform to the workload being executed on them is to optimize the *expiration threshold*, after which being idle causes the server to be expired and terminated. Figures 9 to 13 depict the effect of *expiration threshold* on different system characteristics for different workloads with varying warm and cold service times shown in Table 2. As can be seen, the *expiration threshold* has a substantial effect on most system characteristics, where increasing the *expiration threshold* would improve the quality of service, while increasing the infrastructure cost for the serverless platform provider at the same time. Besides, each workload might also have different tolerances for latency. However, as the average response time is the primary quality of service indicator, we desire to drive down the cost and energy consumption as much as possible.

Figure 12 shows the average instance count in the warm pool serving the incoming requests. Assuming the FaaS provider uses an IaaS provider underneath, we consider the infrastructure cost for the provider proportional to the number of instances dedicated to the user's function service. Thus, the average instance cost is our estimate of the provider's cost for serving the same amount of workload (since the arrival rate is kept constant). Assuming the provider will change their pricing proportional to the infrastructure costs, an estimate of the user's cost can be obtained by multiplying the average billed service time by the price per processing time. Figure 13 shows such a normalized estimate for the cost inferred by the user. As can

be seen, different workloads have different behaviour when changing the expiration threshold. For example, consider workload 4, where increasing the expiration threshold from 1 to 600 causes less than 30% improvement in average response time, while it increases the user cost by a factor of 10. However, the same change in workload 3 causes major improvements in average response time while decreasing the user cost by a factor of more than 5. This shows the potential savings that can be unlocked by leveraging our analytical performance model presented and evaluated in this paper.

TABLE 2
A list of workloads analyzed in this study for potential cost and energy savings in the serverless computing platform.

Name	Application	Warm (ms)	Cold (ms)
W1	CPU and Disk Intensive Benchmark [3]	2000	2200
W2	A rang of benchmarks with different configuration [33]	300	10000
W3	Startup test with echo on Apache OpenWhisk [34]	20	1000
W4	Fibonacci calculation on AWS Lambda [14]	4211	5961
W5	Fibonacci calculation on Azure Functions [14]	1809	26681

Figure 11 shows the utilization of the instances in the warm pool for different *expiration thresholds* values. As defined in this study, utilization shows the average ratio of the number of running (billed) instances over all instances in the warm pool. Lower utilization rate causes the creation and maintenance of more instances, which would increase the infrastructure costs. As can be seen in Figure 11, increasing the *expiration threshold* causes utilization to decrease, while for many workloads, as shown in Figures 9 and 10, it wouldn't lead to a noticeable improvement in the quality of service. Considering this effect, we see potentials for substantial savings in infrastructure costs for providers, which could potentially lead to greener computing and emission reductions.

Figure 14 shows the probability of rejection by the platform because of reaching the maximum concurrency level. Such calculations can help the users decide if the serverless computing platform chosen for their workload can handle peaks in arrival requests without the need to perform large-scale and expensive experimentation.

The benefits of our performance model for the serverless providers are two-fold: 1) They can reduce the operational costs by optimizing their management via leveraging analytical performance models, which allows them to decrease the price of their offerings; 2) They can provide users with fine-grain control over the cost-performance trade-off by modifying the *expiration threshold* underneath. This is mainly due to the fact that there is no universal optimal point in the cost-performance trade-off for all workloads. By making accurate predictions, a serverless provider can better optimize their resource usage while improving the experience of application developers and consequently, the end-users. Such degrees of flexibility could also impact the popularity of the platform among developers. Moreover, utilizing the performance model proposed here, serverless computing providers have the chance to incorporate performance-by-design into their management and operation layers.

On the other hand, the presented model could help application developers to decide if a given workload can be deployed on a serverless computing platform while maintaining their desired Quality-of-Service (QoS) guarantees. The only measurement needed to characterize a workload are the average cold and warm start response times, which could be measured in a straightforward manner. The presented model would also help developers come up with appropriate concurrency and memory settings available in public serverless computing platforms.

5 Related Work

Serverless Computing has attracted a lot of attention from the research community. However, to the best of authors' knowledge, no performance model has been proposed that captures different challenges and aspects unique to serverless computing platforms. This work is an effort to present a performance model that captures the complexities of serverless computing and helps us extract several important characteristics of the serverless system. Performance and availability have been listed on the top 10 obstacles towards the adoption of cloud services [35]. Rigorous models have been leveraged to analytically model the performance of various cloud services for IaaS, PaaS, and microservices [6], [36], [37], [38], [39], [40], [41]. In [36], a cloud data center is modelled as a classic open network with a single arrival. Using this modelling, the authors managed to extract the distribution of the response time, assuming interarrival and service times are exponential. Using the response time distribution, the maximum number of tasks and the highest level of service could be derived. [37] models the cloud data center as *M/M/m/m+r* queuing system and derives the distribution of response time. Assuming the periods are independent, the response time is broken down to waiting, service, and execution later on. Khazaei et al. [6], [38], [39], [40] have proposed monolithic and interactive submodels for IaaS cloud data centers with enough accuracy and tractability for large-scale cloud data centers. Qian et al. [41] proposed a model that evaluates the quality of experience in a cloud computing system using a hierarchical model. Their model uses the Erlang loss model and *M/M/m/K* queuing system for outbound bandwidth and response time modelling, respectively. Ataie et al. [42] proposed a hierarchical stochastic model for performance, availability, and power consumption analysis of IaaS clouds. They utilized Stochastic Reward Nets (SRNs) in their proposed model. Instead of a large monolithic analytical model, they developed two approximate SRN models using folding and fixed-point iteration techniques to enable large-scale modelling of the cloud system. Chang et al. [43] proposed a hierarchical stochastic modelling approach for performance modelling of IaaS cloud data centers under a heterogeneous workload. They investigated the effects of variation in job arrival rate, buffer size, maximum vCPU numbers on a PM and VM size distribution on the quality of service metrics. They

also developed closed-form solutions for key performance metrics of the system. Malik et al. [44] used High-Level Petri Nets (HLPNs) for modelling and analysis of VM-based cloud management platforms. They provided a firm mathematical model and analyzed the structural and behavioural properties of the system. Tarplee et al. [45] used statistical programming to find the best set of computing resources to allocate to the workload in IaaS cloud computing environments. Their algorithm models the uncertainty in the computing resources and variability in the tasks in a many-task computing environment. Using their model, reward rate, cost, failure rate, and power consumption can be optimized to compute Pareto fronts. Lloyd et al. [46] developed a cost prediction model for service-oriented applications (SOAs) deployments to the cloud. Their model can be leveraged to find lower hosting costs while offering equal or better performance by using different types and counts of VMs. In [47], the authors proposed and validated an analytical performance model to study the provisioning performance of microservice platforms and PaaS systems operating on top of VM based IaaS. They used the developed model to perform what-if analysis and capacity planning for large-scale microservices. Barrameda et al. [48] proposed a novel statistical cost model for application offloading to cloud computing environments. In their work, each module's cost is modelled as a random variable characterized by its Cumulative Distribution Function (CDF), which is estimated through profiling. They achieved an efficient offloading algorithm based on a dynamic programming formulation. Their method achieved a prediction error of 5 percent with sequential and branching module dependencies. Wu et al. [49] developed a VM launching overhead reference model for cloud bursting. The cloud bursting module is designed to enable private clouds to automatically launch VMs to public clouds when more resources are needed. Their model helps the decision-making process of when and where to launch a VM to maximize the utilization and performance of the system. They verified their model using FermiCloud, a private cloud for scientific workflows. Eismann et al. [50] demonstrated the benefits and challenges that arise in the performance testing of microservices and how to manage the unique complications that arise while doing so.

Due to the fact that there is not much information regarding the management of public serverless offerings, we can only rely on experimentation and speculations to gain insights into the serverless offerings. Wang et al. [3] performed extensive experimentations on the most widely used serverless computing platforms and compiled their findings into insights about how each provider is handling the workload introduced to their systems. Figiela et al. [8] investigated cost, performance, and the life-cycle of an instance in public serverless offerings by deploying a benchmark workload on each of them. Their results shed some light on the management layers of the serverless offerings, as well as depicting the performance implications of different management decisions made by providers.

Research has been done to investigate the performance of serverless computing platforms, but none are offering rigorous analytical models that could be leveraged to optimize the management of the platform. Eyk et al. [51] looked into the performance challenges in current serverless computing platforms. They found the most important challenges hindering the adoption of FaaS to be the sizable computational overhead, unreliable performance, and absence of benchmarks. The introduction of a reliable performance model for FaaS offerings could overcome some of these shortcomings. Kaffes et al. [52] introduced a core-granular and centralized scheduler for serverless computing platforms. The authors argue that serverless computing platforms exhibit unique properties like burstiness, short and variable execution time, statelessness, and single-core execution. In addition, their research shows that current serverless offerings suffer from inefficient scalability, which is also confirmed by Wang et al. [3]. Manner et al. [14] designed a series of experiments to investigate the factors influencing the cold start performance of serverless computing platforms. Their experiments on AWS Lambda and Azure Functions show that factors like the programming language, deployment package size, and memory settings affect the performance on serverless computing platforms. In some settings, the cold start and the warm start had very similar latencies, whereas, in others, the cold start latency could be significantly larger than the warm start latency (e.g., Java on Azure). In [9], Bortolini et al. performed experiments on several different configurations and FaaS providers in order to find the most important factors influencing the performance and cost of current serverless platforms. They found that one of the most important factors for both performance and cost is the programming language used. In addition, they found low predictability of cost as one of the most important drawbacks of serverless computing platforms. Lloyd et al. [10] investigated the factors influencing the performance of serverless computing platforms. They identified four states for the infrastructure in a serverless computing platform: provider cold, VM cold, container cold, and warm. Their results show that the performance of the infrastructure relies heavily upon the state of the system at the time of arrival. Bardsley et al. [53] examined the performance profile of AWS Lambda as an example of a serverless computing platform in a low-latency high-availability context. They found that although the infrastructure is managed by the provider, and it is not visible to the user, the solution architect and the user need a fair understanding of the underlying concepts and infrastructure. Pelle et al. [54] investigated the suitability of serverless computing platforms (AWS Lambda, in particular) for latency-sensitive applications. Thus, the main focus in their research was on delay characteristics of the application. Their findings showed that there are usually several alternatives of similar services with significantly different performance characteristics. They found the difficulty of predicting the application performance for a given task, one of the major drawbacks of current serverless offerings. They also measured the application performance for different loads, which could possibly be calculated using an analytical performance model. Hellerstein et al. [55] addressed the main gaps present in the first-generation serverless computing platforms and the anti-patterns present in them. They showed how current implementations are restricting distributed programming and cloud computing innovations. The issues of no global states and the inability to address the lambda functions directly over the network are some of

these issues. Eyk et al. [5] found the most important issues surrounding the widespread adoption of FaaS to be sizeable overheads, unreliable performance, and new forms of cost-performance trade-off. In their work, they identified six performance-related challenges for the domain of serverless computing and proposed a roadmap for alleviating these challenges. Balla et al. [56] performed extensive experimental studies on language runtimes in open source FaaS. They showed that it is possible to tune some of these runtimes for better performance, but overall, Go programming language results in the best median latency with similar functionality followed by NodeJS and Python.

Li et al. [57] used analytical models that leverage queuing theory to optimize the performance of composite service application jobs by tuning configurations and resource allocations. We believe a similar approach is possible using the presented analytical model for serverless computing platforms. Horovitz et al. [58] used machine learning-based cost and performance optimization to warm-up containers for future requests. Their results show that proactive management of serverless computing platforms could reduce the number of cold starts occurring and thus improve the quality of service. The new paradigm shift toward using serverless computing platforms calls for redesigning the management layer of the cloud computing platforms. To do so, Kannan et al. [59] proposed GrandSLAm, an SLA-aware runtime system that aims to improve the SLA guarantees for function-as-a-service workloads and other microservices. Lin et al. [12] used a pool of warm containers to mitigate cold starts in serverless computing platforms. They showed that even with a warm pool of only one container, we could decrease the number of cold starts by 85%. Utilizing a performance model for the proposed serverless platform, one could gain performance improvements while mitigating the overhead cost introduced to the system. Gunasekaran et al. [60] used AWS Lambda alongside VMs to reduce SLO violations while keeping the cost to a minimum. In the proposed method, they used serverless computing due to its fast autoscaling compared to VMs in order to serve spurious and bursty workloads. Bermbach et al. [13] looked into the use of application knowledge to reduce the number of cold starts in FaaS services. They developed a client-side middleware that analyzes a process and determines the approximate number and time of requests to later functions in the process. On average, they were able to mitigate the number of cold start by 40% in their experiments. Xu et al. [61] proposed an adaptive warm-up strategy as well as an adaptive container pool scaling using a time series prediction model that tries to minimize the cold starts in serverless computing while reducing the waste of container pool based on the function chain model. An analytical model with the level of fidelity presented in this work could be leveraged to optimize the strategies presented in such work with better reliability characteristics. Akkus et al. [34] used application-level sandboxing, and hierarchical message buses to speed up the conventional serverless computing platforms. Their approach proved to lead to lower latency and better resource efficiency as well as more elasticity than current serverless platforms like Apache OpenWhisk.

6 Conclusion

In this work, we presented and evaluated an accurate and tractable analytical performance model suitable for analyzing the performance, utilization and cost of current mainstream serverless computing platforms. We analyzed the performance implications of different system configurations and workload characteristics of the public serverless offerings and showed, through experimentation, that the proposed model could accurately estimate the steady-state performance of various workloads. We also showed that the performance model is scalable, which is critical for evaluating large scale deployments. Serverless users can utilize the presented model to predict the cost and performance of their application and evaluate the effectiveness of FaaS for their workloads. Serverless providers can leverage the presented model to offer an adjustable quality of service and cost. The presented model also allows savings in cost and energy through optimization of the infrastructure for each workload, leading to energy and emission reduction and allowing the realization of green computing.

In summary, the proposed performance model can transform serverless platforms from "workload-agnostic" environments to "workload-aware" adaptive platforms.

Acknowledgement

This research was enabled in part by support from Sharcnet (www.sharcnet.ca) and Compute Canada (www.computecanada.ca). 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] Amazon Web Services Inc., "Serverless Computing." https://aws.amazon.com/serverless/. Last accessed 2019-07-04.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv preprint arXiv:1902.03383*, 2019.

[3] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 133–146, 2018.

[4] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," *arXiv preprint arXiv:2003.03423*, 2020.

[5] E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "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*, pp. 21–24, ACM, 2018.

[6] H. Khazaei, J. Misic, and V. B. Misic, "A Fine-Grained Performance Model of Cloud Computing Centers," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 11, pp. 2138–2147, 2012.

[7] G. Grimmett, G. R. Grimmett, D. Stirzaker, *et al.*, *Probability and Random Processes*. Oxford university press, 2001.

[8] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance Evaluation of Heterogeneous Cloud Functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, p. e4792, 2018.

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

[10] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 159–169, IEEE, 2018.

[11] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, "Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pp. 203–213, 2019.

[12] P.-M. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach," *arXiv preprint arXiv:1903.12221*, 2019.

[13] D. Bermbach, A. S. Karakaya, and S. Buchholz, "Using Application Knowledge to Reduce Cold Starts in FaaS Services," in *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing*, 2020.

[14] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 181–188, IEEE, 2018.

[15] Google Cloud Platform Inc., "Concurrency." https://cloud.google.com/run/docs/about-concurrency. Last accessed 2020-02-13.

[16] The Knative Authors, "Configuring concurrency." https://knative.dev/v0.16-docs/serving/autoscaling/concurrency/. Last accessed 2020-09-03.

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

[18] G. McGrath and P. R. Brenner, "Serverless computing: Design, Implementation, and Performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, IEEE, 2017.

[19] S. Bose, "M/G/m/m Loss System," *Prieiga per internet: http://www.iitg. ac. in/skbose/qbook/MGmm_Queue. PDF*, 2001.

[20] L. A. Baxter, "Probability, statistics, and queueing theory with computer sciences applications," 1992.

[21] W. Whitt, "Continuous-Time Markov Chains," *http://www.columbia.edu/ww2040/6711F13/ CTMCnotes120413.pdf*, 2006.

[22] Amazon Web Services Inc., "Serverless Computing." https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html. Last accessed 2020-02-28.

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

[24] Amazon Web Services Inc., "AWS Lambda." https://aws.amazon.com/lambda/. Last accessed 2020-02-03.

[25] Google Inc., "Cloud Functions." https://cloud.google.com/functions. Last accessed 2020-02-03.

[26] Microsoft Inc., "Azure Functions Serverless Compute." https://azure.microsoft.com/en-us/services/functions/. Last accessed 2020-02-03.

[27] IBM Inc., "IBM Cloud Functions." https://cloud.ibm.com/functions. Last accessed 2020-02-03.

[28] Apache Software Foundation, "OpenWhisk - Open Source Serverless Cloud Platform." https://openwhisk.apache.org/. Last accessed 2020-02-03.

[29] Google Inc., "Cloud Run." https://cloud.google.com/run. Last accessed 2020-02-03.

[30] OpenFaaS Ltd., "OpenFaaS - Serverless Functions Made Simple." https://www.openfaas.com/. Last accessed 2020-02-03.

[31] Kubeless Inc., "Kubeless." https://kubeless.io/. Last accessed 2020-02-03.

[32] Fission Contributors, "Serverless Functions for Kubernetes - Fission." https://fission.io/. Last accessed 2020-02-03.

[33] Robert Vojta, "AWS Journey — API Gateway & Lambda & VPC Performance." https://link.medium.com/PHevHj8ji4. Last accessed 2020-02-19.

[34] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "{SAND}: Towards High-Performance Serverless Computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 923–935, 2018.

[35] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[36] K. Xiong and H. Perros, "Service Performance and Analysis in Cloud Computing," in *2009 Congress on Services-I*, pp. 693–700, IEEE, 2009.

[37] B. Yang, F. Tan, Y.-S. Dai, and S. Guo, "Performance Evaluation of Cloud Service Considering Fault Recovery," in *IEEE International Conference on Cloud Computing*, pp. 571–576, Springer, 2009.

[38] H. Khazaei, J. Misic, and V. B. Misic, "Modelling of Cloud Computing Centers using M/G/m Queues," in *31st International Conference on Distributed Computing Systems Workshops*, pp. 87–92, IEEE, 2011.

[39] H. Khazaei, J. Misic, and V. B. Misic, "Performance Analysis of Cloud Computing Centers using $M/G/m/m+r$ Queuing Systems," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 5, pp. 936–943, 2011.

[40] H. Khazaei, J. Misic, and V. B. Misic, "Performance Analysis of Cloud Centers under Burst Arrivals and Total Rejection Policy," in *IEEE Global Telecommunications Conference-GLOBECOM*, pp. 1–6, IEEE, 2011.

[41] H. Qian, D. Medhi, and K. Trivedi, "A Hierarchical Model to Evaluate Quality of Experience of Online Services Hosted by Cloud Computing," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM) and Workshops*, pp. 105–112, IEEE, 2011.

[42] E. Ataie, R. Entezari-Maleki, L. Rashidi, K. S. Trivedi, D. Ardagna, and A. Movaghar, "Hierarchical Stochastic Models for Performance, Availability, and Power Consumption Analysis of IaaS Clouds," *IEEE Transactions on Cloud Computing*, 2017.

[43] X. Chang, R. Xia, J. K. Muppala, K. S. Trivedi, and J. Liu, "Effective Modeling Approach for IaaS Data Center Performance Analysis under Heterogeneous Workload," *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 991–1003, 2016.

[44] S. U. Malik, S. U. Khan, and S. K. Srinivasan, "Modeling and Analysis of State-of-the-Art VM-Based Cloud Management Platforms," *IEEE Transactions on Cloud Computing*, vol. 1, no. 1, pp. 1–1, 2013.

[45] K. M. Tarplee, A. A. Maciejewski, and H. J. Siegel, "Robust Performance-Based Resource Provisioning using a Steady-State Model for Multi-Objective Stochastic Programming," *IEEE Transactions on Cloud Computing*, 2016.

[46] W. J. Lloyd, S. Pallickara, O. David, M. Arabi, T. Wible, J. Ditty, and K. Rojas, "Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives," *IEEE Transactions on Cloud Computing*, vol. 5, no. 4, pp. 667–680, 2015.

[47] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency Analysis of Provisioning Microservices," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 261–268, IEEE, 2016.

[48] J. Barrameda and N. Samaan, "A Novel Statistical Cost Model and an Algorithm for Efficient Application Offloading to Clouds," *IEEE Transactions on Cloud Computing*, vol. 6, no. 3, pp. 598–611, 2015.

[49] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, K. Chadwick, and S.-Y. Noh, "A Reference Model for Virtual Machine Launching Overhead," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 250–264, 2014.

[50] S. Eismann, C. P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A Performance Tester's Dream or Nightmare?," in *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20)*, 2020.

[51] E. van Eyk and A. Iosup, "Addressing Performance Challenges in Serverless Computing," in *Proc. ICT. OPEN*, 2018.

[52] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized Core-Granular Scheduling for Serverless Functions," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 158–164, 2019.

[53] D. Bardsley, L. Ryan, and J. Howard, "Serverless Performance and Optimization Strategies," in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 19–26, IEEE, 2018.

[54] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, "Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 272–280, IEEE, 2019.

[55] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[56] D. Balla, M. Maliosz, C. Simon, and D. Gehberger, "Tuning Run-

times in Open Source FaaS," in *International Conference on Internet of Vehicles*, pp. 250–266, Springer, 2019.
[57] X. Li, S. Liu, L. Pan, Y. Shi, and X. Meng, "Performance Analysis of Service Clouds Serving Composite Service Application Jobs," in *2018 IEEE International Conference on Web Services (ICWS)*, pp. 227–234, IEEE, 2018.
[58] S. Horovitz, R. Amos, O. Baruch, T. Cohen, T. Oyar, and A. Deri, "FaaStest-Machine Learning Based Cost and Performance FaaS Optimization," in *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pp. 171–186, Springer, 2018.
[59] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–16, 2019.
[60] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 199–208, IEEE, 2019.
[61] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive Function Launching Acceleration in Serverless Computing Platforms," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 9–16, IEEE, 2019.

Nima Mahmoudi received the BS degrees in Electronics and Telecommunications and the MS degree in Digital Electronics from Amirkabir University of Technology, Tehran, Iran in 2014, 2016, and 2017 respectively. He is currently working towards the PhD degree in software engineering and intelligent systems at the University of Alberta, Edmonton, AB, Canada. He is a Research Assistant at the University of Alberta and a visiting Research Assistant in the Performant and Available Computing Systems (PACS) lab at York University, Toronto, ON, Canada. His research interests include serverless computing, cloud computing, performance modelling, applied machine learning, and distributed systems. He is a student member of the IEEE.

Hamzeh Khazaei (Member, IEEE) is an assistant professor in the Department of Electrical Engineering and Computer Science at York University. Previously he was an assistant professor at the University of Alberta, a research associate at the University of Toronto and a research scientist at IBM, respectively. He received his PhD degree in Computer Science from the University of Manitoba, where he extended queuing theory and stochastic processes to accurately model the performance and availability of cloud computing systems. His research interests include performance modelling, cloud computing and engineering distributed systems.