Performance Modeling of Metric-Based Serverless Computing Platforms

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

Abstract—Analytical performance models are very effective in ensuring the quality of service and cost of service deployment remain desirable under different conditions and workloads. While various analytical performance models have been proposed for previous paradigms in cloud computing, serverless computing lacks such models that can provide developers with performance guarantees. Besides, most serverless computing platforms still require developers' input to specify the configuration for their deployment that could affect both the performance and cost of their deployment, without providing them with any direct and immediate feedback. In previous studies, we built such performance models for steady-state and transient analysis of scale-per-request serverless computing platforms (e.g., AWS Lambda, Azure Functions, Google Cloud Functions) that could give developers immediate feedback about the quality of service and cost of their deployments. In this work, we aim to develop analytical performance models for latest trend in serverless computing platforms that use concurrency value and the rate of requests per second for autoscaling decisions. Examples of such serverless computing platforms are Knative and Google Cloud Run (a managed Knative service by Google). The proposed performance model can help developers and providers predict the performance and cost of deployments with different configurations which could help them tune the configuration toward the best outcome. We validate the applicability and accuracy of the proposed performance model by extensive real-world experimentation on Knative and show that our performance model is able to accurately predict the steady-state characteristics of a given workload with minimal amount of data collection.

Index Terms—Serverless Computing, Metric-Based Autoscaling, Knative, Google Cloud Run, Performance Modelling, Optimization, Stochastic Processes.

1 Introduction

Serverless computing platforms are the latest paradigm in the cloud computing era that aim to minimize the administration tasks required to deploy a workload to the cloud. They provide developers, software owners, and online services with services like handling system administration tasks, improving resource utilization, usage-based billing, improved energy efficiency, and more straightforward application development [1], [2].

Despite having a much faster startup time compared with VM-based deployments, serverless offerings have shown to lack predictability in key performance metrics. This has rendered them as unacceptable for many customer-facing products [2]. The issue is exacerbated by the fact that current generation of serverless computing platforms are workload-agnostic; i.e., using the same management policies for all types of workload with different needs [3], [4], [5]. This gives us a plethora of possible savings in terms of infrastructure cost and energy consumption while improving the overall performance by adapting the platform to the unique needs of each workload [6].

An accurate performance model like the one suggested in this work can benefit both serverless providers and application developers. Application developers can leverage performance models to predict the quality of service of their application with different configurations and the respective cost implications, helping them select the configurations that fits their needs. They also can use the performance model to find the limitations of their system and plan ahead for large uptakes in the workload intensity. On the other hand, an accurate performance model can help serverless providers perform capacity planning and give application developers an estimate on cost and performance implications of their workload configurations.

A proper performance model for serverless computing platforms should remain tractable while covering a large portion of the system configuration space. In previous studies [7], [8], we designed such performance models for serverless computing platforms that use scale-per-request autoscaling paradigm predicting both transient and steady-state quality of service characteristics. In this work, we aim to develop and evaluate a performance model that captures the unique structure and characteristics of the most recent paradigm in serverless computing platforms which leverage concurrency value [7] and other metrics to drive autoscaling. The most important examples of these serverless computing platforms are Knative and Google Cloud Run (which is a managed Knative offering from Google Cloud Platform).

The analytical performance model presented in this work assumes a Poisson arrival process to address customer-facing open networks which comprise the majority of services which require strong quality of service guarantees. It has been shown that the arrival process can adequately be modelled as a Poisson process when there are a large number of clients with each having a low probability to submit a request at any given time [9], [10], [11], [12], [13].

- *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.2021.xxxxxxx

We impose no restrictions on the service time distribution or service policies by using data-driven techniques that help extract the unique characteristics of a given workload. The presented model in this work is highly scalable and can handle a high degree of parallelization required in large-scale systems. The presented model can help predict the cost and main quality of service indicators for a given workload, e.g., the average response time. In addition, the presented performance model can help developers by predicting the inherent performance-cost tradeoffs for different workload configurations.

The proposed performance model has been validated by extensive experimentation on Knative deployed on our private cloud computing infrastructure and works with any workload that can be deployed as Docker containers and accepts HTTP requests. The development of the model requires a minimal data collection on the target platform to capture the resource needs of the workload and the effect of concurrency on the quality of service metrics.

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 discusses the threats to the validity of our experiments. Section 7 summarizes our findings and concludes the paper.

2 System Description

There is very limited documentation available about the scheduling algorithm used in most serverless computing platforms that use per-request autoscaling [7]. As a result, previous studies have mostly focused on partially reverse engineering these platforms by running experiments on them [3], [4], [14], [15], [16]. However, the most recent trend in serverless computing platforms that use metric-based autoscaling, Knative [17] and Google Cloud Run [18] for example, are primarily open-sourced and thus we can use their source code to develop accurate performance models without speculations[1].

Figure 1 shows an overview of the Knative scale calculation module. As shown, we need to choose a monitored metric that will be used to drive the autoscaling in our deployment. Then, the metric will go through windowing and averaging to generate more stable observed metrics. Using the observed and target values of the used metric, scale evaluator can calculate the new replica count for a given deployment. This process is repeated every few seconds to ensure the system remains stable. In the next sections, we will go through the details of these steps to outline the system modelled by the proposed performance model.

2.1 Metrics

In the metric-based autoscaling approach used in Knative, there are currently two widely available metrics than can be used to drive autoscaling: 1) Concurrency Value (CC) and 2) Requests Per Seconds (RPS) [19]. Any of these metrics can be used as the primary monitored metric and will be compared against the target value for replica count calculations. These metrics will be monitored by the sidecar container injected by Knative to the Kubernetes deployment and are collected every second.

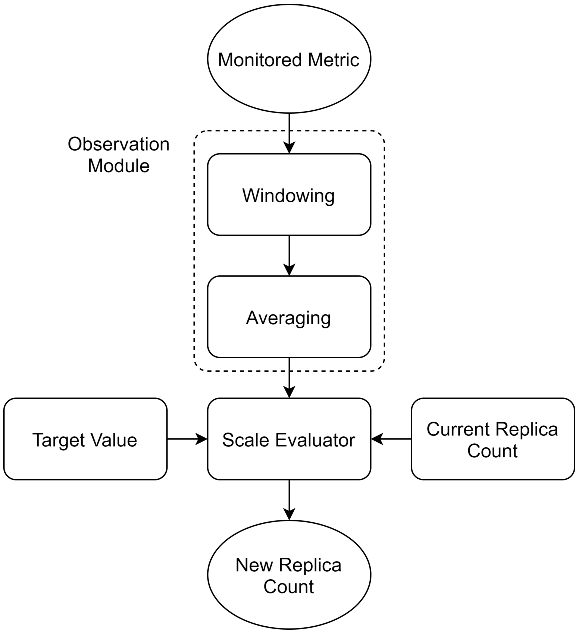

Fig. 1. An overview of the Knative scale calculation module. The resulting new replica count will be applied to the cluster.

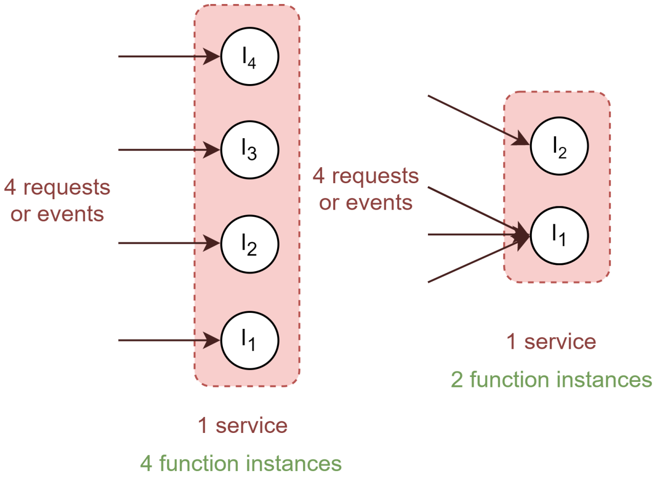

Fig. 2. The effect of the concurrency value on the number of function instances needed. The left service allows a maximum of 1 request per instance, while the right service allows a concurrency value of 3.

2.1.1 *Concurrency Value (CC)*

Unlike most public serverless computing platforms that primary use scale-per-request autoscaling such as AWS Lambda, Google Cloud Functions, Azure Functions, and IBM Cloud Functions, Knative and consequently Google Cloud Run allow several requests to enter the same function instance at the same time. The number of concurrent

[1] Note that metric-based autoscaling precedes serverless computing, but new serverless computing generations use different metrics and measurement methods to drive their autoscaling.

requests being processed by the same container is called the *concurrency value* in Knative documentations. Figure 2 shows the possible effect of concurrency in serverless computing platforms which could lead to fewer function instances. Concurrency value is the default metric used in Knative and is the only metric supported in Google Cloud Run. Thus, we will focus more on this metric throughout this work, but the proposed performance model also works with RPS as the monitored metric. Figure 3 shows an example of how concurrency changes with request arrival and departure in each container. As can be seen, any request arrival results in an increment in the concurrency value and any request departure results in a decrement in the monitored concurrency value.

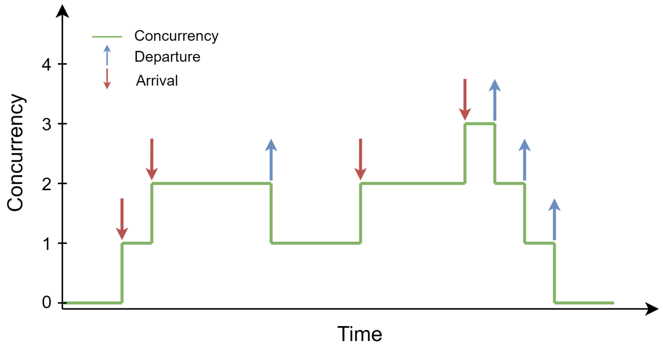

Fig. 3. An example scenario of the change in the container concurrency value. The effect of request arrival and departure on concurrency value is shown over time.

2.1.2 Requests Per Seconds (RPS)

The arrival rate for each container or RPS is another monitored metric supported by Knative. However, at the moment this metric is not being supported by Google Cloud Run. The measurement of this metric is straightforward, the monitoring module monitors the number of requests arriving to each container every second and reports the resulting value.

2.2 Observation Module

The observation module is responsible for collecting monitored metrics from all containers, generating the average values for every second, and calculating the moving average throughout time according to the the *stable window* configuration. The default value of the *stable window* is 60 seconds in Knative.

The output of this module is the *observed value* that will be used for driving scaling decisions. The role of this module is to generate stable observations in order to avoid making premature decisions in the scaling evaluations.

2.3 Scale Evaluator Module

As discussed in Section 2.2, the observation module generates stable averaged measurements from single container measurements of the monitoring module. The *Scale Evaluator* uses these measurements and the user-specified configurations to generate the new replica count ordered by the evaluator in each evaluation using the following equation[2]:

$$NewOrderedReplica = \left\lceil \frac{ObservedValue}{TargetValue} \right\rceil \tag{1}$$

where the *Observed Value* and *Target Value* are values of the chosen monitoring metric by the user, i.e., concurrency or RPS. By default, the Knative autoscaling evaluation takes place every T_{eva} (2 seconds in Knative), setting the new replica target on the Kubernetes deployment.

3 Analytical Model

In Section 2, we outlined the details of the system modelled by our proposed performance model. In this section, we will go through the details of the performance models based on the described system. Our primary focus here is to predict steady-state metrics of a given workload based on the input system configurations.

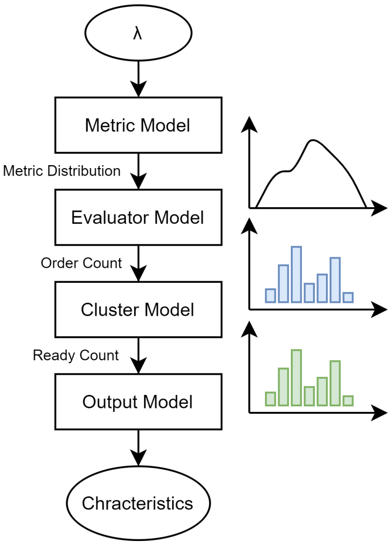

Fig. 4. An overview of the proposed performance model.

Figure 4 shows an overview of the proposed performance model. As can be seen, given the arrival rate, the metric module can use the workload profile to calculate the distribution of the monitored autoscaling metric (i.e., concurrency value or RPS). This step is very important as it captures several important characteristics of a given workload like the amount of work needed for each request and its distribution, along with the deployment configuration like the CPU and memory configuration of the deployment. This is mainly due to the fact that the effect of all of the aforementioned properties is captured in the data achieved from the monitoring module. Given this value, the evaluator model can estimate the probability of setting different values for the replica count of the service deployment. Having calculated the probability of setting the replica count to different values and using the estimated provisioning/deprovisioning rates, we can estimate the probability of seeing different replica counts using the cluster model. Finally, using the ready container replica count and by using the output model, we calculate the steady-state estimates for different characteristics of the deployment.

[2] source: https://github.com/knative/serving/blob/master/pkg/autoscaler/scaling/autoscaler.go. Last accessed 2021-02-01.

TABLE 1
Symbols and their corresponding descriptions.

Symbol	Description
λ	Mean arrival rate of requests
N	Number of function instances
$\overline{N}$	Average replica count
N_{max}	Maximum number of function instances
N_{ord}	Ordered number of function instances
OV	Observed value of the monitored metric
$f_{OV}(\cdot)$	Density function for the observed value
$F_{OV}(\cdot)$	Distribution function for the observed value
TV	Target value for the monitored metric
MM	Metric model
EM	Evaluator model
T_{eva}	Time between consecutive evaluations
Q	The CTMC transition rate matrix
P	The DTMC transition probability matrix
π	The steady-state distribution
μ_{pro}	Mean provisioning service rate
μ_{dep}	Mean deprovisioning service rate
$\overline{RT}$	Mean Service Response Time
$\overline{RT}_N$	Average response time with N containers
RTF	Response Time Function
N_{opr}	Number of overprovisioned instances
N_{upr}	Number of underprovisioned instances
$\overline{C}$	Average concurrency level
C_i	Average concurrency for state number i

In the following subsections, we present the calculation of different parameters in the analytical models using the symbols defined in Table 1; we will elaborate on the details of the aforementioned sub-models.

3.1 Metric Model

As discussed in Section 2.1, there are two main metrics that can be used with this family of serverless computing platforms, namely concurrency (CC) and the arrival rate for each container (RPS). The chosen metric will then be processed by the observation module and will be windowed and averaged to be used in scaling operation. The goal of the metric model is to estimate the distribution of the observed values for a given arrival rate. However, different applications show very different behaviours when processing more than one request.

Processing times, and consequently measured concurrency values, are largely influenced by factors like service policy (whether the application uses First Come First Serve, Processor Sharing, or a combination of both) and its reliance on external service. Intuitively, using a fair load balancer, we can safely assume that the service time and concurrency value for a given workload largely depend only on arrival rate per container, i.e., RPS or λ/N. Also, since the observed metric is being averaged over several containers and over 60 measurements throughout the time, it can safely be assumed to be coming from a Gaussian distribution due to the central limit theorem. Thus, we decided to use data-driven methods to estimate the observed metric average and standard deviation. As a result, we need a few minutes of data collection for a given workload to build our data-driven model before generating predictions. We used 5 minutes of data collection for our experiments and collecting enough data to have at least 100 measurements is suggested to achieve an acceptable accuracy, but gathering more data can always improve the accuracy of the system.

In this step, our goal is to find the function MM that estimates the following:

$$f_{OV}(x) \approx MM(x; \lambda/N) \tag{2}$$

where $f_{OV}(\cdot)$ denotes the observed value density function, MM denotes the metric model, λ denotes the arrival rate, and N represents the number of ready containers in the cluster. Using this distribution, the evaluator model can estimate the number of ordered containers and their probabilities. Note that to develop this model, we are assuming a homogeneous cluster where each container has a similar amount of CPU. We also assume a good performance isolation between containers which is safe assumption due to the high level of performance isolation in the modern managed Knative services like Google Cloud Run.

3.2 Evaluator Model

The evaluator model has been designed to model the behaviour of the *Scale Evaluator* module of the autoscaler. In this model, we use the observed value density function $f_{OV}(\cdot)$ to calculate the probability of different values for the new number of ordered replica count. This module will take the *Target Value* (*TV*), maximum replica count (N_{max}), and other configuration that affect the ordered replica count (e.g. maximum scale up/down rate) into account. We know from the system description that the new ordered replica count in each evaluation is given by the following equation:

$$N_{ord} = \left\lceil \frac{OV}{TV} \right\rceil \tag{3}$$

where N_{ord} is the new number of ordered replica count, OV represents the observed value, and TV is the target value set by the user. Thus, we can calculate the probability of a specific value (i) for N_{ord}:

$$\begin{aligned} Pr\{N_{ord} = i\} &= Pr\{\left\lceil \frac{OV}{TV} \right\rceil = i\} \\ &= Pr\{(i-1) < \frac{OV}{TV} \leq i\} \\ &= Pr\{(i-1) \cdot TV < OV \leq i \cdot TV\} \\ &= F_{OV}(i \cdot TV) - F_{OV}((i-1) \cdot TV) \end{aligned} \tag{4}$$

where $F_{OV}(\cdot)$ is the cumulative density function of the observed value which can be calculated from the metric model using the following:

$$F_{OV}(x) = Pr\{OV \leq x\} = \int_{-\infty}^{x} f_{OV}(x)\, dx \tag{5}$$

Repeating this procedure for any possible number of containers in the range $[1, N_{max}]$, we get the probability of having different values for the number of ordered instance counts in a given deployment.

$$EM(i; f_{OV}) = F_{OV}(i \cdot TV) - F_{OV}((i-1) \cdot TV) \quad (6)$$

where $EM(i; f_{OV})$ is the probability of setting the ordered replica count to i given f_{OV}. These results help us build a complete and accurate cluster model to predict the overall behaviour of our deployment.

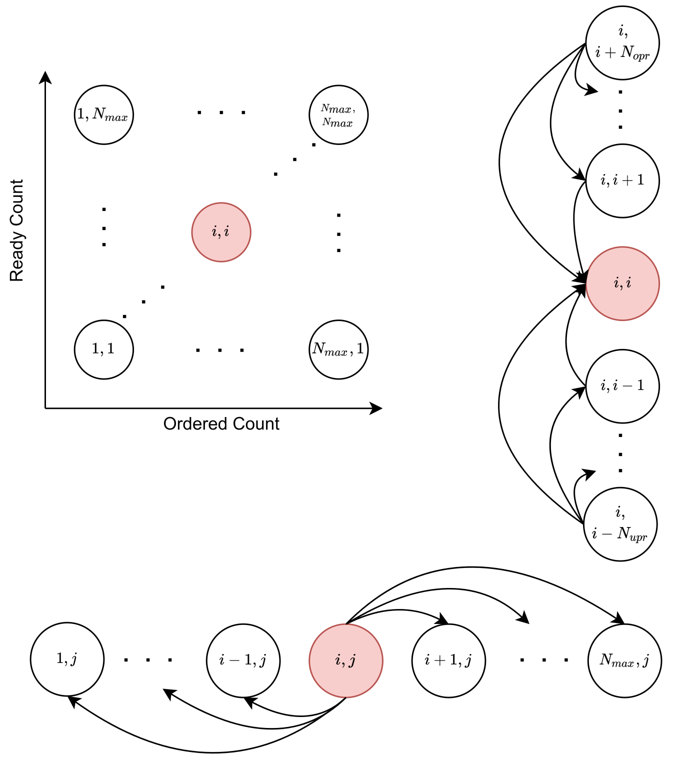

Fig. 5. An overview of the proposed cluster model along with its vertical and horizontal components.

3.3 Cluster Model

In this section, we will detail the design of the proposed Discrete-Time Markov Chain (DTMC) representing the status of our metric-based serverless deployment in the cluster.

Figure 5 shows an overview of the proposed two dimensional DTMC where the x-axis represents the number of containers ordered by the evaluator and the y-axis shows the number of containers that are currently in the *Ready* state and can accept incoming requests. To build the resulting model, we have chosen to evaluate the system at the moment after each evaluation by the scale evaluator. As a result, the newly set order count has not had the chance to affect the system yet and thus gives us the ability to decouple the *single-step* infrastructure effect of provisioning or deprovisioning of containers from the effect of the execution of the evaluator. This is due to the fact that we are modelling a physical system here, and like any other physical systems, configuration changes cannot affect the system instantly and require some time to do so. It is worth noting that our model still captures the relationship between the number of ordered containers and ready containers via vertical transitions (better shown in Figure 6) in consequent steps of the model.

There are two main forces causing the change in the system: 1) change in the order count due to execution of the scale evaluator; and 2) change in the number of deployment containers due to provisioning or deprovisioning of containers. Due to the aforementioned decoupling between these two forces that affect our state, they will be independent and thus any transition probability in these two dimensions can be broken down as the following:

$$P_{(i,j),(i',j')} = P_{i,i'}(j) \times P_{j,j'}(i) \quad (7)$$

where $P_{(i,j),(i',j')}$ is the probability of transitioning to state (i', j') given our current state is (i, j), $P_{i,i'}(j)$ is the probability of transitioning from column i to column i' from row j, and $P_{j,j'}(i)$ is the probability of transitioning from row j to row j' from column i. As can be seen, $P_{i,i'}(j)$ does not depend on j' and $P_{j,j'}(i)$ does not depend on i', which significantly reduces the computational complexity of the overall performance model.

To calculate $P_{i,i'}(j)$, we use the evaluator model developed in the previous section. We assume the result of each scale evaluation is independent from previous evaluations and thus we have:

$$\begin{aligned} f_{OV}(x) &= MM(x; \lambda/j) \\ P_{i,i'}(j) &= EM(i'; f_{OV}) \end{aligned} \quad (8)$$

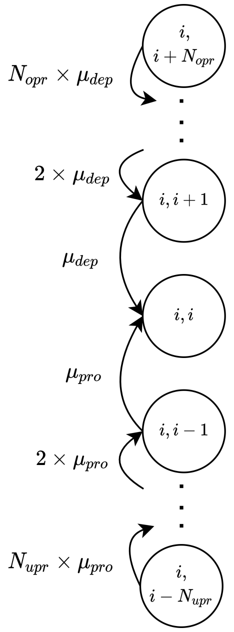

Fig. 6. An overview of the underlying infrastructure CTMC model used in the cluster model. N_{opr} and N_{upr} signify the number of overprovisioned and underprovisioned containers and μ_{pro} and μ_{dep} represent the provisioning and deprovisioning service rates, respectively.

To obtain $P_{j,j'}(i)$, we need to analyze how the infrastructure reacts when provisioning or deprovisioning of containers for a given deployment takes effect. To do so, we use the Continuous-Time Markov Chain (CTMC) model shown in Figure 6 and solve for possible transitions after T_{eva} units of time. In this model, we assume exponentially distributed service times for provisioning/deprovisioning for which the rate is proportional to the amount of the underlying resources. As a result, $P_{j,j'}(i)$ becomes the probability of starting in state (i, j) and provisioning/deprovisioning enough containers to get to j' containers in the cluster after T_{eva} units of time.

To solve the resulting CTMC model, we use the one-step transition rate matrix Q to get the state distribution π'. In this matrix, each element located in row x and column y shows the transition rate at which we transition from state x to state y. Diagonal elements are defined in a way to satisfy $Q_{x,x} = -\sum_{y \neq x} Q_{x,y}$. To solve the resulting CTMC, we have to solve the following equation:

$$\frac{d\pi'}{dt} = \pi' Q \Rightarrow \pi'(t) = \pi'(0)e^{Qt} \tag{9}$$

which can be calculated using the method proposed by Al-Mohy et al. [20].

Using the state distribution π', we can calculate the transition probabilities $P_{j,j'}(i)$ using the following equation:

$$P_{j,j'}(i) = \pi'_{j'}(T_{eva}) \tag{10}$$

Using Equations (8) and (10), we can build the transition probability matrix P for the cluster model shown in Figure 5. To analyze the steady-state behaviour of the system, we need to calculated the limiting probability π_s for any state s where [21]:

$$\pi_s = \lim_{n \to \infty} P^n_{s,s'} \tag{11}$$

where π_s is the probability that chain is in state s, independent of the starting state s'. Using these limiting probabilities, we can calculate the limiting distribution π:

$$\pi = (\pi_1, ..., \pi_M), \sum_{x=1}^{M} \pi_x = 1 \tag{12}$$

where M signifies the total number of states, which is $M = N^2_{max}$ here. It can be shown that the resulting limiting distribution is π if $\pi P = \pi$ and $\sum_{x=1}^{M} \pi_x = 1$. This system of equations can be solved using the method outlined in [22].

After knowing the steady-state probability of being in each state via π, we need to calculate different desired metrics and characteristics of the workload.

3.4 Output Model

In the previous section, we went over the details of cluster model, which is used to calculate the limiting distribution. In this section, we will use the resulting state distribution to calculate metrics of interest in a given Knative deployment. Two of the most important metrics in a given deployment are *average response time* as an indicator for Quality of Service (QoS), *average replica count* as an indicator for cost, and *average concurrency* as a metric used in infrastructure planning like database capacity planning, etc. Here, we will go over the details of calculating each of these metrics.

3.4.1 Average Response Time

Average response time is one of the most widely used metrics to indicate the quality of service for a given deployment in the context of web services.

Intuitively, assuming negligible overhead in the Kubernetes routing mechanism (compared to the request processing time), the average response time for a given workload is only a function of arrival rate per container (λ/N), or in other words the amount of work given to each containers. However, this relationship is highly dependent on the type of workload, its parallel or concurrency features, and the type of workload being used (CPU, I/O, or memory intensive, or a combination of them).

As a result, we have decided to use automated data-driven methods to extract to which extent does the average response time rely on the arrival rate per container and show the result as the following:

$$\overline{RT}_N = \mathit{RTF}(\lambda/N) \tag{13}$$

where $\overline{RT}_N$ is the average response time of the service when we have N containers and RTF shows the response time function, estimated using regression methods from our brief profiling window. To calculate the total average response time, we use the state probabilities calculated:

$$\begin{aligned}\overline{RT} &= \sum_{i=1}^{M} \pi_i \overline{RT}_{N_i} \\ &= \sum_{i=1}^{M} \pi_i \mathit{RTF}(\lambda/N_i)\end{aligned} \tag{14}$$

where M is the number of states, N_i is the number of ready containers in state number i, and π_i is the probability of being in state number i at any time step.

3.4.2 Average Replica Count

Nowadays elite cloud vendors use very complicated and regularly changing pricing schema with multitude of charges for different services and providing a complete pricing model for them is infeasible. However, there are mainly two sets of factors used in calculating the incurred cost of a given deployment in a serverless setting: 1) per-request costs and 2) per-instance cost. For a given arrival rate, the calculation of per-request costs are rather straightforward since we have an estimate of $\lambda \cdot T$ for the number of requests in any given time window with length T. However, calculating per-instance costs relies on the system configurations and characteristics and can vary drastically based on these settings. To provide application developers and operations experts with a tool that helps them understand the tradeoffs of their deployments, we leverage the developed performance model to calculate the average number of running instances in the cluster.

To calculate the average replica count, we can use the state probabilities calculated in previous sections:

$$\overline{N} = \sum_{i=1}^{M} \pi_i N_i \tag{15}$$

where $\overline{N}$ is the average replica count and N_i is the number of ready containers in state number i.

3.4.3 Average Concurrency

The average concurrency level per container is a measure that can help application developers set reasonable resource limits and configurations for a given service as well as tune other services they rely upon, e.g., databases. The average concurrency level ($\overline{C}$) can also be calculated using state probabilities:

$$\overline{C} = \sum_{i=1}^{M} \pi_i C_i \tag{16}$$

where $\overline{C}$ is the overall average concurrency and C_i is the average concurrency for state number i. To get C_i, we can use the metric model for concurrency value:

$$C_i = \int_0^\infty x \cdot MM(x; \lambda/N_i) dx \tag{17}$$

4 Experimental Evaluation

In this section, we introduce our evaluation of the proposed analytical performance model using experimentation on our Knative installation. The code for performing and analyzing the experiments used in this section can be found in our public GitHub repository[3], along with installation and deployment instructions of various workloads used in this study. To the best of authors' knowledge, no other work has proposed a performance model for this type of serverless computing platforms. As a result, our experimental results only include our measurements compared to the proposed performance model.

TABLE 2
Configuration of the VMs in the experiments.

Property	Value
vCPU	4
RAM	8GB
HDD	40GB
Network	1000Mb/s
OS	Ubuntu 20.04
Latency	<1ms

4.1 Experimental Setup

To perform our experiments, we used 4 Virtual Machines (VMs) on the Cybera Cloud [23] with the configuration shown in Table 2. Of the VMs used, 3 joined in a Kubernetes cluster and 1 used as the client. We found the cluster size sufficient for our experiments due to the fact that modern application architectures include several smaller deployments each receiving a portion of the traffic and our approach aims to model these individual deployments. For our cluster, we used Kubernetes version *1.20.0* with Kubernetes client (kubectl) version *1.18.0*. For the client, we used *Python 3.8.5*. To generate client requests based on a Poisson process, we used our in-house workload generation library [4] which is publicly available through PyPi[5]. 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. For all experiments, we performed the experiment in 6 batches totalling one hour for each combination of configurations to get accurate results. Based on the tests on our cluster, we used the estimated values of $\mu_{pro} = 1$ and $\mu_{dep} = 2$ events per second.

4.2 Workloads

To evaluate the proposed performance model, we used workloads in Python and Go programming languages to represent different types of applications. The results for all of these workloads can be found on the project's GitHub repository. To improve the generalizability of the results, these workloads each include several parts designed to dominate one or more resources, and by using different combinations of these workloads, we can represent a large spectrum of different workloads. We also included scripts that automate the process of deployment, load testing, and logging of the results. We present a representative subset of these results here due to space limitation. For *workload 1*, we used the work of Wang et al. [3] written in *Python* with minor modifications and utilizing Flask as the web server. This workload is a combination of CPU intensive and I/O intensive workloads. For *workload 2*, we used a standard and open-source suite of benchmarks implemented by the Knative community in the Go programming language[6].

The regression method is not an integral part of our performance model and thus any regression method with enough accuracy for a given workload can be used. To predict the mean concurrency value based on λ/N for experimental workloads, we used a simple polynomial regression of the following form with no training on the intercept:

$$y = \alpha_1 \cdot x + \alpha_2 \cdot x^2 \tag{18}$$

where α_is are the trained parameters of the model, y represents the output value, and x represent the input to the model (λ/N). This method has a low number of parameters, which increases its interpretability. Besides, its variance is low which enables us to train it accurately using a limited amount of data. It also allows us to control the regression's behaviour in extreme values to make sure it presents sensible values for the model. For example, in very low arrival rates, we know the measured concurrency should approach zero, which is integrated into this model. In our experiments with *workload 1* and *workload 2*, the resulting fit had a Mean Squared Error (MSE) of 0.1004 and 0.004 and R^2 score of 0.9875 and 0.9991, respectively.

Similarly and for the same reasons, we used a polynomial regression but this time with an intercept to get the response time from average arrival rate per container. The resulting function is of the following form:

$$y = \alpha_0 + \alpha_1 \cdot x + \alpha_2 \cdot x^2 \tag{19}$$

[3] https://github.com/pacslab/conc-value-perf-modelling
[4] https://github.com/pacslab/pacswg
[5] https://pypi.org/project/pacswg
[6] For more information, visit https://knative.dev/docs/serving/autoscaling/autoscale-go/

where α_is are the trained parameters of the model, y represents the output, and x represent the input to the model (λ/N). One nice feature that can be enforced with a simpler regression method like the one presented here is that we can control it to approach the service time of the workload when arrival rate per container approaches zero. In our experiments with *workload 1* and *workload 2*, the resulting fit had a Mean Squared Error (MSE) of 0.0380 and $8.5177*10^{-7}$ and R^2 score of 0.8159 and 0.6259, respectively.

4.3 Experimental Results

In this section, we go through our experimental results and their predicted counterparts. To get the results for each point shown in the experimental plots, we ran a test with a specific Poisson arrival process for every single point; we also eliminate the first 5 minutes of the experiment to eliminate the transient effect (i.e., warm-up effect).

Figures 7 and 8 show the measured and predicted average number of containers that are ready to serve incoming requests for different configurations, respectively. Average number of containers are used here as a proxy to deployment cost. Depending on the setup, the deployment cost can be VM-based in a Kubernetes cluster or Pod-based in a Google Cloud Run deployment. However, in both scenarios, the infrastructure costs will be proportional to the average number of containers. Figures 9 and 10 depict the average concurrency value for different configurations measured and predicted, respectively. These values can help the developer set proper configurations for other services that the deployment relies on. For example, the provisioned capacity for most managed database solutions can be set to optimize performance while keeping the costs low. The average response time has been targeted here as an indicator to the deployment Quality of Service (QoS). Figures 11 and 12 outline the measured and predicted average response time for different configurations and arrival rates, respectively. As can be seen, the experimental results shown here are well in tune with the model predictions.

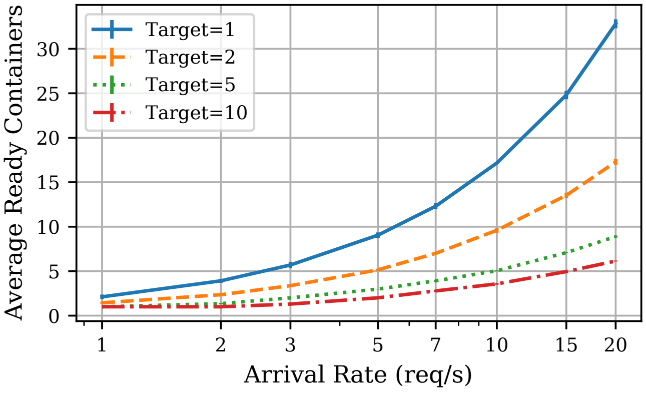

Fig. 7. The measured average number of containers ready to server requests versus the fixed arrival rate for different target concurrency values in our experiments. Note that the x-axis is on a logarithmic scale. The vertical bar shows the 95% confidence intervals which in this case were very small because experiments were long enough to have very accurate results.

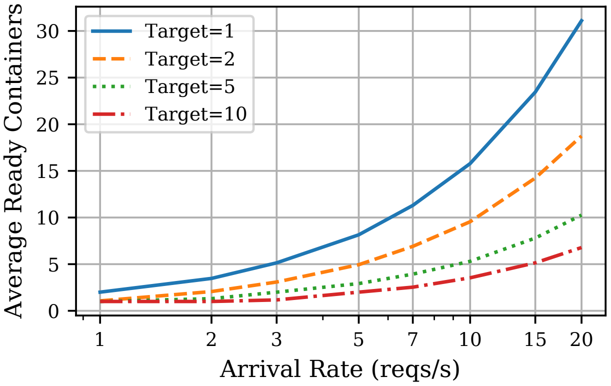

Fig. 8. The predicted average number of containers ready to server requests versus the fixed arrival rate for different target concurrency values. Note that the x-axis is on a logarithmic scale.

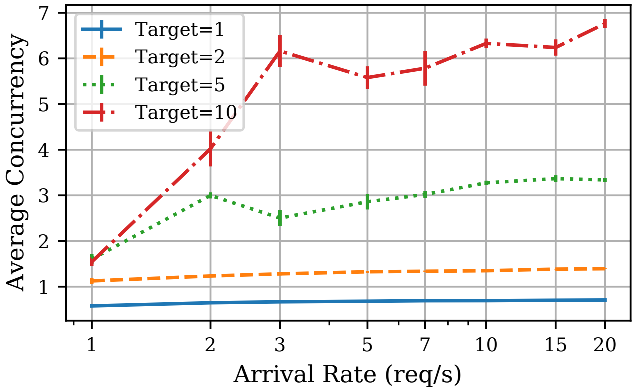

Fig. 9. The measured average concurrency value versus the fixed arrival rate for different target concurrency values in our experiments. Note that the x-axis is on a logarithmic scale. The vertical bar shows the 95% confidence intervals.

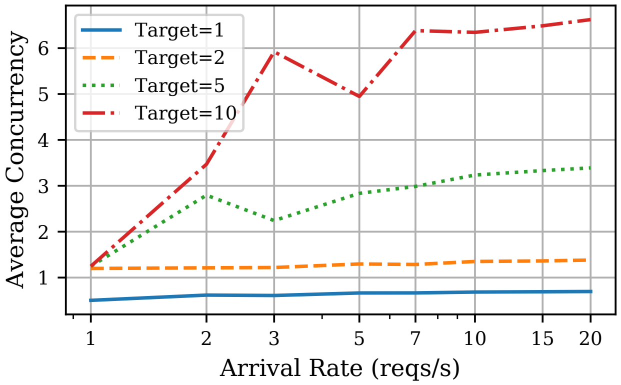

Fig. 10. The predicted average concurrency versus the fixed arrival rate for different target concurrency values. Note that the x-axis is on a logarithmic scale.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3169619

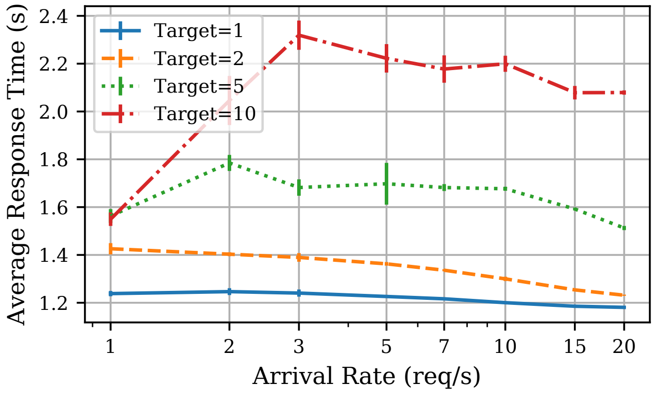

Fig. 11. The measured average response time versus the fixed arrival rate for different target concurrency values in our experiments. Note that the x-axis is on a logarithmic scale. The vertical bar shows the 95% confidence intervals.

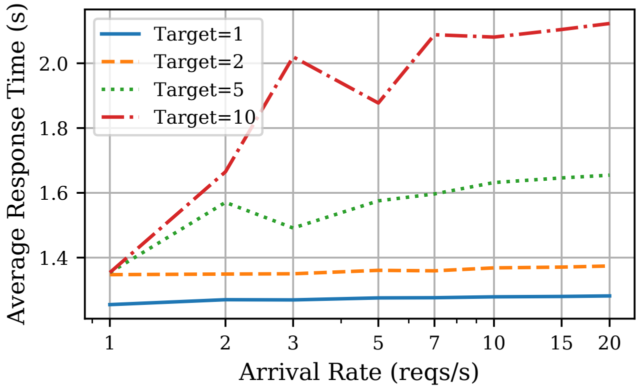

Fig. 12. The predicted average response time versus the fixed arrival rate for different target concurrency values. Note that the x-axis is on a logarithmic scale.

4.4 Discussion

In Section 4.3, we compared the experimental results with the performance model predictions and showed that the effectiveness of the proposed performance model to predict the results of different configurations for metric-based autoscaling in serverless computing platforms. The resulting performance model can be used for any metric-based autoscaling platform as long as they adhere to the system description outlined in Section 2. Examples of serverless computing platform that follow the discussed system description are Google Cloud Run and Knative. To improve the tractability and accuracy of the model while requiring a minimal amount of training data, we chose to use grey-box modelling to integrate our knowledge about the system into the model while allowing the flexibility needed to adapt to different types of workload.

In Figures 7 to 12, we showed the accuracy of the proposed model in predicting key characteristics of the system under different load intensities. By compiling these results, we can create tools that can be leveraged by the developer to optimize their configurations by predicting the effect of a new configuration on the performance and the cost of the system. Figure 13 shows the measured and predicted values for the response time and number of instances. These figures can be used to see the effect of the target value configuration on the cost and QoS simultaneously, which can be beneficial to make a decision about the configuration for a given deployment. As can be seen, the performance model can be consulted by the developer to find the optimal target value configuration in a given system for their specific use case. As different systems have different criteria, finding a globally optimal point for the target value is not possible, but by presenting similar tools, serverless providers can help facilitate a more informed decision by the developers. Figure 14 shows a similar plot but for *workload 2*. As can be seen, the effect of changing the chosen target value on the quality of service varies for different workloads, but the selected regression is able to predict this effect with sufficient accuracy.

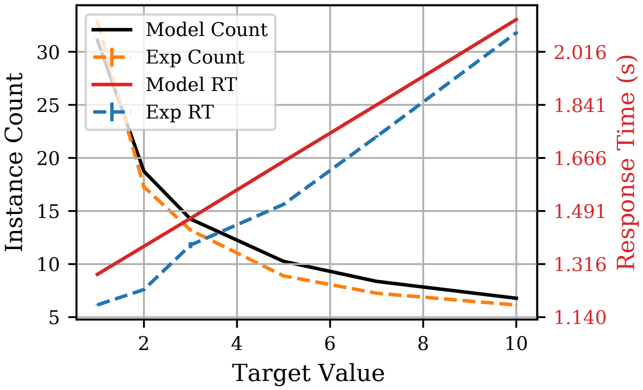

Fig. 13. The effect of changing the target value on the average instance count and average response time measured in experiment and predicted by the proposed model for an arrival rate of 20 requests per second for *workload 1*.

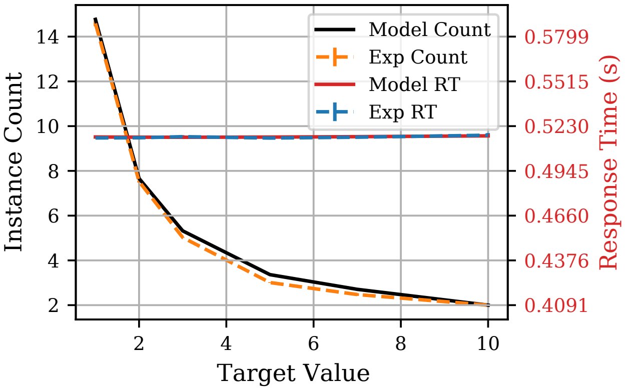

Fig. 14. The effect of changing the target value on the average instance count and average response time measured in experiment and predicted by the proposed performance model for an arrival rate of 20 requests per second for *workload 2*.

5 Related Work

Serverless Computing has attracted a lot of attention from the research community. However, a limited number of research have focused on performance models capturing different challenges and aspects unique to serverless computing platforms. In previous studies, we have developed and evaluated steady-state and transient performance models along with simulators for scale-per-request autoscaling in serverless computing platforms [7], [8], [24]. This work is an effort to present a performance model that captures the complexities of metric-based autoscaling, the newest trend in serverless computing platforms, 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 [25]. Rigorous models have been leveraged to analytically model the performance of various cloud services for IaaS, PaaS, and microservices [6], [26], [27], [28], [29], [30], [31], [32], [33], [34]. In [26], a cloud datacenter 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. Yang et al. [27] modelled the cloud datacenter 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], [28], [29], [30], [34] have proposed monolithic and interactive submodels for IaaS cloud datacenters with enough accuracy and tractability for large-scale cloud datacenters. Qian et al. [31] 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. Lloyd et al. [35] 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 [36], 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. Eismann et al. [37] 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.

Kaviani et al. [38] discusses the effectiveness of several key components of Knative and its contribution to open-source serverless computing platforms. They found the Knative autoscaler highly effective and mature for modern workloads.

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. [39] 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. [40] 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]. In [15], 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. [16] investigated the factors influencing the performance of serverless computing platforms. Bardsley et al. [41] 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. [42] 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. Hellerstein et al. [43] 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. Zheng et al. [44] compared the performance of OpenFaaS, Kubeless, Fission, and Knative and found that the performance of these open-sourced serverless platforms depends on the type of workload, the runtime implementation, and the FaaS system with the optimal set varying case by case.

Li et al. [45] 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. 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. [46] proposed GrandSLAm, an SLA-aware runtime system that aims to improve the SLA guarantees for function-as-a-service workloads and other microservices. Akkus et al. [47] 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. Jia et al. [48] present Nightcore, which is an efficient and scalable serverless computing framework with improved invocation latency overhead and very high invocation rate. To achieve this, they designed improved scheduling modules and introduced concurrency hints to their serverless autoscaler. Balla et al. [49] introduced Libra, an adaptive hybrid vertical/horizontal autoscaler on OpenFaaS trying to outperform both openfaas autoscaler and Kubernetes HPA.

6 Threats to Validity

In this section, we discuss different threats to the validity of our work. We will also go over some of the limiting assumptions that we needed to make for this study to ensure that an interested reader is aware of their implications in the proposed performance model.

In our experiments, we used the average response time as an indicator of the Quality of Service (QoS) and the average instance count as an indicator of costs. These may have an impact on the results obtained if they don't fully align with the user's use case. Analyzing every possible QoS measure and the full billing model of all modern cloud providers is infeasible. We have selected metrics that are commonly used in load testing experiments [50]. Modern cloud-native workloads are also billed based on their provider API usage (e.g., managed machine learning APIs) and Internet traffic. However, we believe these costs mostly depend on the total number of requests served and thus can be calculated without the need of a performance model.

For the presented experiments, we used two workloads in different programming languages, each comprising several configurable benchmarks that stress different resources of the computer and represent different types of workloads. Although experimenting with all types of workloads is not possible, the accuracy of the performance model might differ between different programming languages. Future work should investigate further how the knowledge can be transferred between different programming languages. We also assumed that any external APIs used by the workload have a predictable performance that is not affected by the amount of work applied by the studied workload. This assumption was necessary since no performance model can consider unknown variations in an external API used by the workload.

The accuracy of the proposed model depends on the accuracy of the regression used in our metric and output model. In our experiments, we manually ensured the quality of the resulting fit but didn't fully investigate the extent of this relationship and how much data is required to train a regression model with sufficient accuracy. Future studies should investigate the extent of this relationship and how much training data is needed to ensure results have a predetermined accuracy.

In our experiments, we assumed stationarity for the workloads. This tends to hold true for most workloads, but some workloads might violate this assumption. Especially if the incoming request can drastically affect processing time and the incoming requests change over time, we might see significant model drift. This effect can be mitigated by retraining the metric model over time, but the effect has not been analyzed here and is outside the scope of this study.

Performance experiments in the cloud always have a high degree of uncertainty due to the variable performance perceived in cloud. Using a private academic cloud allowed us to limit the variability of the performance, but results could vary in public clouds on shared (or burstable) CPU configurations. To mitigate this threat, we used recommended practices to obtain and report our experimental results [50].

7 Conclusion

In this work, we proposed and evaluated an accurate and tractable performance model for metric-based autoscaling in serverless computing platforms. We analyzed the implications of different system configurations and workload characteristics of these systems and showed the effectiveness of the proposed model through experimental validation. We also showed how the presented performance model can be used as a tool by application owners for finding the optimal configuration for a given workload under different loads. Serverless providers can also use the proposed model to adopt an adaptive and more sensible defaults for the target value configuration. They can also leverage the performance model to optimize the cost, performance, and energy efficiency of their system according to the real-time arrival rate.

Acknowledgement

This research was enabled in part by support from Sharcnet (www.sharcnet.ca) and Compute Canada (www.computecanada.ca). We would like to thank Cybera, Alberta's not-for-profit technology accelerator, who supports this research through its Rapid Access Cloud services.

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] N. Mahmoudi and H. Khazaei, "Performance Modeling of Serverless Computing Platforms," *IEEE Transactions on Cloud Computing*, pp. 1–15, 2020.

[8] N. Mahmoudi and H. Khazaei, "Temporal Performance Modelling of Serverless Computing Platforms," in *Proceedings of the 6th International Workshop on Serverless Computing*, WOSC '20, pp. 1–6, Association for Computing Machinery, 2020.

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

[10] R. Ghosh, K. S. Trivedi, V. K. Naik, and D. S. Kim, "End-to-End Performability Analysis for Infrastructure-as-a-Service Cloud: An Interacting Stochastic Models Approach," in *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, pp. 125–132, 2010.

[11] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: A model-less and managed inference serving system," 2020.

[12] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, "Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network)," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pp. 1–8, 2018.

[13] W. Li, X. Ma, J. Wu, K. S. Trivedi, X.-L. Huang, and Q. Liu, "Analytical Model and Performance Evaluation of Long-Term Evolution for Vehicle Safety Services," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 3, pp. 1926–1939, 2017.

[14] 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.

[15] 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.

[16] 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.

[17] The Knative Authors, "Knative." https://knative.dev. Last accessed 2021-02-01.

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

[19] The Knative Authors, "Metrics." https://knative.dev/docs/serving/autoscaling/autoscaling-metrics/. Last accessed 2021-02-01.

[20] A. H. Al-Mohy and N. J. Higham, "A New Scaling and Squaring Algorithm for the Matrix Exponential," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 970–989, 2010.

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

[22] Herman Scheepers, "Markov Chain Analysis and Simulation using Python." https://towardsdatascience.com/markov-chain-analysis-and-simulation-using-python-4507cee0b06e. Last accessed 2021-02-15.

[23] Cybera, "Rapid access cloud." https://www.cybera.ca/services/rapid-access-cloud, March 2021.

[24] N. Mahmoudi and H. Khazaei, "SimFaaS: A Performance Simulator for Serverless Computing Platforms," in *International Conference on Cloud Computing and Services Science*, CLOSER '21, pp. 1–11, Springer, 2021.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] H. Khazaei, N. Mahmoudi, C. Barna, and M. Litoiu, "Performance Modeling of Microservice Platforms," *IEEE Transactions on Cloud Computing*, pp. 1–15, 2020.

[35] 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.

[36] 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.

[37] 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.

[38] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards serverless as commodity: a case of knative," in *Proceedings of the 5th International Workshop on Serverless Computing*, pp. 13–18, 2019.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] C. Zheng, N. Kremer-Herman, T. Shaffer, and D. Thain, "Autoscaling high-throughput workloads on container orchestrators," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 142–152, IEEE, 2020.

[45] 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.

[46] 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.

[47] 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.

[48] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," 2021.

[49] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5, IEEE, 2020.

[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 ACM/SPEC International Conference*

on Performance Engineering, ICPE '20, (New York, NY, USA), p. 138–149, Association for Computing Machinery, 2020.

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 graduate 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.