Performance Modelling and Optimization of Serverless Computing Platforms

by

Nima Mahmoudi

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

Abstract

During the past 10 years, we have witnessed the proliferation of cloud computing services and their adoption in the industry. This rapid growth has been mainly due to economies of scale, improving resource utilization, infinite computing resources on demand, and pay per use cost model. However, the abundance of cloud computing services and infrastructure has left us with lots of resources to manage, which could surpass the development effort for a product. Several new paradigms have emerged since the introduction of cloud computing services, which have tried to improve the services and address these shortcomings. The most promising paradigm shift for the newly emerging cloud services is serverless computing. As a result, we are currently amid an evolutionary paradigm shift in cloud computing towards serverless platforms. This change is due to several improvements over traditional cloud computing, like handling virtually all of the administrative tasks, improving resource utilization, potential operational cost savings, improved energy efficiency, and more straightforward application development. In addition, serverless computing can enable the rapid development of new cloud-based solutions in our current highly-dynamic environment, enabling new solutions for unforeseen circumstances, e.g., pandemic-era applications. However, the current implementations of serverless computing are far from perfect and have a long way to achieve the full potentials of this paradigm.

This thesis focuses on modelling and improving different aspects of serverless computing platforms. As mentioned above, in serverless computing, a large portion of operational tasks are delegated to the cloud operator, which frees the user from these tasks but leaves the operator to create a management system able to handle almost

any type of workload. The current generations of serverless computing have emerged to sub-optimal management solutions, leading to underutilized infrastructure, exaggerated costs, and unsatisfactory latencies violating most Quality of Service (QoS) guarantees. One of the major reasons behind these shortcomings is the fact that the current serverless computing management systems are workload-agnostic, i.e., they don't adapt to the type of workload being executed on them. Global effort and research is needed to develop management systems capable of running most types of workloads with near-optimal behaviour.

This thesis strives to develop analytical and data-driven models and methods and leverage them to improve the status quo in current serverless computing platforms. We aim to develop several optimization modules using different techniques, complementing each other in different scenarios.

Preface

This thesis is the original work of Nima Mahmoudi. This thesis is organized in paper format following the guidelines for paper-based theses. Parts of this report have been accepted to the peer-reviewed publications listed below.

- Mahmoudi, N. & Khazaei, H. (2021). SimFaaS: A Simulator for Serverless Computing Platforms. The 11th IEEE International Conference on Cloud Computing and Services Science (CLOSER 2021).
- Mahmoudi, N. & Khazaei, H. (2020). Temporal Performance Modelling of Serverless Computing Platforms. The 6th Workshop on Serverless Computing (WoSC6).
- Mahmoudi, N. & Khazaei, H. (2020). Performance Modeling of Serverless Computing Platforms. IEEE Transactions on Cloud Computing (TCC).
- Mahmoudi, N., Lin, C., Khazaei, H., & Litoiu, M. (2019). Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm. 29th Annual International Conference on Computer Science and Software Engineering (pp. 203-213). IBM Corp.

Parts of this report are still undergoing review by the following venues:

- Mahmoudi, N. & Khazaei, H. (2022). Analysis of Metric-Based Autoscaling in Serverless Computing. IEEE Transactions on Cloud Computing (TCC). (major revision)

- Mahmoudi, N. & Khazaei, H. (2022). MLProxy: SLA-Aware Reverse Proxy for Machine Learning Inference Serving on Serverless Computing Platforms. 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2022). (submitted)

Acknowledgements

I would like to thank everyone who has supported and guided me throughout my PhD program. First, I highly appreciate the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant. I would like to express my deep and sincere gratitude to my supervisors, Dr. Hamzeh Khazaei and Dr. Marek Reformat, for providing support and invaluable guidance throughout my research.

Thanks to all my friends and colleagues in the PACS lab who helped me throughout my studies. I would particularly like to thank Changyuan Lin, Alireza Goli, and Mikael Sabuhi, for their assistance with this research.

In addition, I am extremely grateful to my lovely wife, Sara, for her continued support and encouragements when the times got rough. I am also very much thankful to my parents, Rozita and Majid, and my sister, Niki, for their support and guidance throughout my life, and for believing in me, even when I didn't.

This research was enabled in part by support from Sharcnet (www.sharcnet.ca) and Compute Canada (www.computecanada.ca). I would like to thank Cybera, Alberta's not-for-profit technology accelerator, who supported this research through its Rapid Access Cloud services. Parts of this research have also been supported by research credits from Google Cloud Platform (GCP) and Amazon Web Services (AWS).

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are amid an evolutionary paradigm shift in cloud computing towards serverless platforms. This change is due to several improvements over traditional cloud computing, like handling all the system administration operations, improving resource utilization, potential operational cost savings, improved energy efficiency, and more straightforward application development [1, 2]. In this paradigm, the software developers would develop the application through well-defined chunks of code orchestrated in functions. Then, functions will be deployed and run on the cloud infrastructure under predefined conditions and events specified by software owners. This way, the whole runtime management, including provisioning and scaling resources, will be done by the cloud service provider.

In this chapter, we will first describe serverless computing platforms as a new paradigm in cloud computing, then go over more details about the management and container placement of current generations of serverless computing platforms. We then elaborate on the motivation for this thesis and finally wrap up this chapter with the thesis objectives and outline.

1.1 Serverless Computing

Serverless computing includes two overlapping concepts; originally, it was referred to applications that significantly or fully leverage third-party API-based services that

replace core subsets of backend logic and state functionality. Because such APIs are provided as a service that auto-scales and operates transparently, this appears to the developer to be serverless. These types of services have been described as "Backend as a Service" (BaaS). Authentication services (e.g., AWS Cognito and Auth0) and database services (e.g., Firebase and Parse) are prime examples of BaaS [3]. However, serverless can be applied to applications in which the application developers develop the server-side logic, but, as opposed to traditional architectures, it runs in containers that are stateless, event-oriented, ephemeral (may only last for one invocation), and fully managed by a third party. This type of serverless is known as "Function as a Service" (FaaS). It is important to note that although serverless computing is a more generic term which includes both BaaS and FaaS services, it is commonly used interchangeably with FaaS when discussing different execution engines of cloud computing. AWS Lambda functions, Google, Microsoft, IBM and Huawei functions are well-known examples of FaaS [4].

To better understand serverless computing, it is crucial to know other similar technologies and compare their characteristics. There are four primary development and deployment models that application developers may consider when looking for an environment to host their cloud-native applications (Fig. 1.1). All approaches facilitate the deployment of a cloud-native application, but they emphasize different functional and non-functional properties considering their target users and the type of workload. Infrastructure as a service (IaaS) is the most mature one that can be leveraged by software providers to deploy their application in an environment with a high degree of flexibility over the Operating System (OS), frameworks, libraries, programming languages, platform, and runtime management. IaaS does not imply leveraging containers and VMs as the primary deployment unit [5].

Containers-as-a-Service (CaaS) systems maintain full control over the platform so that it has maximum portability. Container orchestration platforms like Kubernetes [6], Mesos [7], and Swarm [8] allow teams to develop and deploy portable ap-

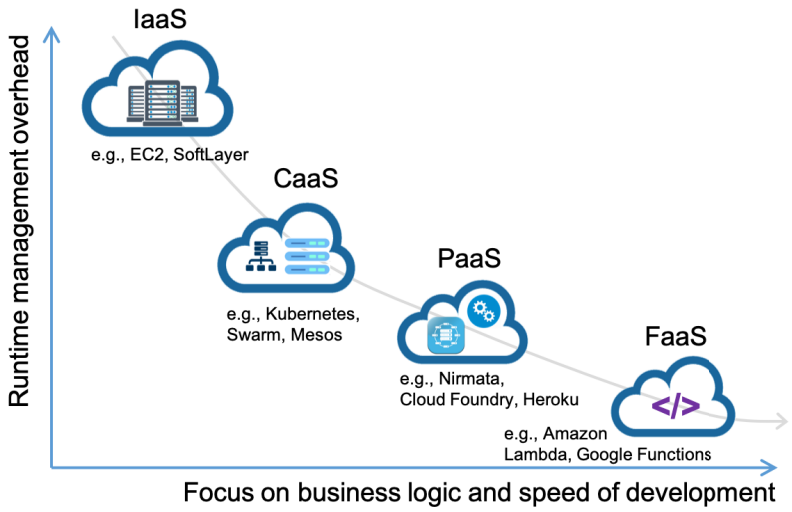


Figure 1.1: Serverless in comparison to other cloud-native application deployments.

plications with flexibility and control over configuration, which can run anywhere without the need to reconfigure and deploy them for different environments.

Platform-as-a-Service (PaaS) paradigm enables teams to deploy and scale applications using a rich set of runtime environments, including a container orchestration engine which has been contextualizing to a catalogue of data, AI, and security services through injection of configuration information into the application, without having to manually configure and manage a container and OS. It is a great fit for existing web apps that have a stable programming model [4].

The paradigm with minimum runtime management overhead is FaaS or serverless in which software developers focus on applications that consist of event-based functions responding to a variety of triggers and let the platform take care of the rest.

In the last three deployment paradigms (i.e., CaaS, PaaS and FaaS), a group of containers comprise the cloud application. Finding the best place to deploy these containers is of great importance as it has a non-trivial impact on the performance/cost of the cloud application. Serverless computing platforms handle almost every aspect of the system administration tasks needed to deploy a workload on the cloud. In the recent years, developers have tried to improve development time, system resiliency,

and scalability by using fine-grained microservices over monolithic architectures. The emerging serverless technologies using FaaS have taken us closer to the *computing as an Internet-based utility* paradigm and have facilitated the accelerated growth of microservices [9].

1.2 Serverless Platform System Description

There is very little official documentation made publicly available about the scheduling and autoscaling algorithms in public serverless computing platforms. However, a number of studies have focused on partially reverse engineering this information using experimentations on these platforms [10–13]. 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 section, we will cover the most important concepts in serverless computing.

Function Instance States: 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). Using the findings of previous studies [10, 11, 14], 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 application developer is not charged for an instance that is in the *idle* state.

Cold/Warm start: as defined in previous studies [10, 11, 13], 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 [10]. This is the reason a lot of researchers in the field of serverless computing have focused on mitigating cold starts [15–17].

Autoscaling: we have identified three main autoscaling patterns among the mainstream serverless computing platforms: 1) *scale-per-request*; 2) *metric-based scaling*; and 3) *resource-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 requests. As the load decreases, to scale the amount of resources 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 [10, 18]. This autoscaling approach is the dominant scaling technique used by major providers.

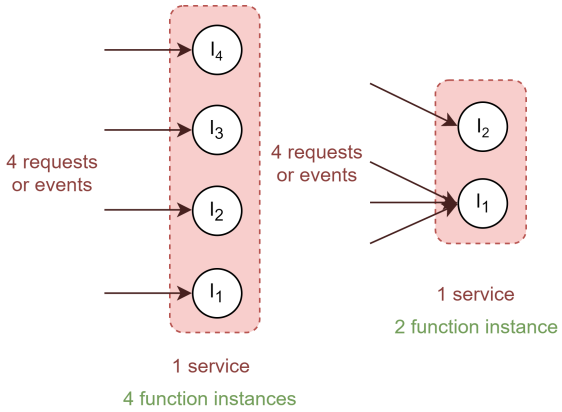


Figure 1.2: The effect of the concurrency value on the number of function instances needed. The left service allows a maximum of 1 request per instance, while the right service allows a concurrency value of 3.

In the *metric-based scaling* pattern [19], 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 1.2 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 metric-based 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 metric-based 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, metric-based 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 metric-based 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 [20].

Resource-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.

Initialization Time: as mentioned earlier, when the platform is spinning up new instances, they will first go into the initialization state. This state might include spinning up new virtual machines, unikernels, or containers. 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, comprises 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 focusing on 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 [10, 11, 13], the distribution of the response time does not change over time with different loads. Therefore, we leveraged delay centers [21] 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 [22]. This means that the scheduler gives priority to newly instantiated idle instances using priority scheduling according to creation time, i.e., 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.

1.3 Motivations and Objectives

Serverless computing platforms provide serverless users[1] with several potential benefits like handling all of the system administration operations and improving resource

[1] we use the terms serverless users and application developers interchangeably.

utilization, leading to potential operational cost savings, improved energy efficiency, and more straightforward application development [1, 2].

Hiding resource management from developers is appealing to some degree, but the resulting opacity hinders adoption for many potential users. For instance, the platforms' ability to deliver on performance [23, 24], security isolation of functions, DDoS resistance [25], as well as the need to understand resource management to improve application performance [26] have shown to be crucial to the adoption of the serverless computing paradigm. There have been some attempts to shed light on the resource management and security [13, 22] of these platforms and the results revealed sub-optimality in many cases. In this section, we discuss the shortcomings and gaps in serverless computing platforms identified by various studies.

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 [10, 18, 27]. 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.

In their study, Wang et al. [10] perform an in-depth and comprehensive evaluation of several aspects of the three most well-known serverless platform providers, namely AWS Lambda, Azure Functions, and Google Cloud Functions (GCF). The evaluated aspects include scalability, resource efficiency, and performance isolation. This work sheds some light on the performance implications of each of these service providers. In this work, the authors discovered several interesting issues regarding the performance isolation and the effect of co-residency of the same type of functions on a VM. They provided measurement-driven approaches to partially reverse-engineer the architecture of these serverless platform providers and uncovered several interesting facts. For example, the AWS Lambda uses binpacking for container placement in

their system, where the placement of the parallel functions seemed to be agnostic to the function code. As discussed in this work, co-residency is undesirable for users wanting several function instances since it causes contention between instances and results in severe performance degradation. The co-residency of functions could cause the CPU utilization rates of the function to fluctuate from 14.1% to 90% with the standard deviation of 16%, which causes a lot of randomness in the observed performance of the function. These results show the potential performance improvements possible through the usage of smarter VM selection algorithms.

Van Eyk et al. [18] identified six performance-related issues in Function-as-a-Service (FaaS) systems and recognized performance isolation as one of the main performance issues with the current FaaS platforms. In this study, they tried to address this issue by improving the utilization of shared resources using the behaviour of the system that is recognized in the profiling stage of their work.

Mcgrath et al. [22] introduced a serverless computing platform based on windows containers and reported the implementation challenges of a serverless computing platform. These challenges include container discovery, life cycle, and container reuse. They performed different tests on their system for SLA-level metrics like latency, throughput, and back-off latency. Although they performed a comprehensive evaluation of their system, they did not exploit the effects of these criteria under diverse workloads (e.g., CPU-intensive, Memory-intensive, and Disk-intensive tasks).

In [28], the authors introduced a package-aware scheduling schema for FaaS functions that tries to pack the functions that need the same library packages on a single machine with some tolerable imbalances, which reduces the network overhead of launching a new instance of a function. Although this seems very interesting, the effect of collocating different functions on the same VM is not investigated.

This research strives to find ways to model, analyze, and optimize the workload execution in serverless computing platforms. The key objectives of this thesis can be summarized as follows:

1. Providing an in-depth analysis in the state of the art of serverless computing platforms and their implications.

2. Development of experimentations and tools enabling research in performance modelling of serverless computing platforms.

3. Design and development of a performance simulator for the mainstream serverless computing platforms.

4. Developing and testing an accurate yet tractable performance model for the mainstream serverless computing platforms.

5. Developing a data-driven optimized scheduler for serverless computing platforms using machine learning techniques.

6. Finding key configurations that can help make current serverless computing platforms workload-aware.

7. Developing an SLA-aware batching algorithm for machine learning inference workloads on managed serverless computing platforms.

1.4 Thesis Outline

The remainder of this document is organized as follows. In Chapter 2, we present our approach to leverage machine learning to develop a data-driven function placement algorithm than can be used to improve the serverless computing platform's throughput.

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 [29]. Chapters 3 and 4 focus on the development of a closed-form analytical performance model for scale-per-request and metric-based serverless computing platforms.

As there are inherent limitations with any performance model, in Chapter 5 we focus on developing a fast and accurate performance simulator for serverless computing platforms to enable both performance researchers and serverless developers to simulate the performance of any desired function with any arrival process.

Throughout experimentation and after in-depth analysis of the literature, we found the serverless adoption to be very low for machine learning inference workloads. Our studies showed that one of the most important reasons for this is the poor performance/cost of this type of workloads on serverless computing platforms. In Chapter 6, we outlined our design for MLProxy, which is an SLA-aware reverse proxy for these types of workoads on serverless computing platforms and can offer improvements both in terms of cost and performance without needing manual intervention.

Finally, Chapter 7 lists our contributions and elaborates on the possible future directions for this research.

Chapter 2

Adaptive Function Placement for Serverless Computing

The main concept behind serverless computing is to build and run applications without the need to manage the servers. It refers to a fine-grained deployment model where applications, comprising one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment. While elite cloud vendors such as Amazon, Google, Microsoft, and IBM are now providing serverless computing services, their approach for the placement of functions, i.e. associated container or sandbox, on servers is oblivious to the workload, which may lead to poor performance and/or higher operational cost for software owners. In this chapter, using statistical machine learning, we design and evaluate an adaptive function placement algorithm that can be used by serverless computing platforms to optimize the performance of running functions while minimizing the operational cost. Given a fixed amount of resources, our smart spread function placement algorithm results in higher performance compared to existing approaches; this will be achieved by maintaining the users' desired quality of service for a longer time, which also prevents premature scaling of the cloud resources. Extensive experimental studies revealed that the proposed adaptive function placement algorithm could easily be adopted by serverless computing providers and integrated into container orchestration platforms without introducing any limiting side effects.

2.1 FaaS and Container Placement

As can be seen in Fig. 2.1, in FaaS, requests may be triggered by an event, schedulers, or users. The API gateway processes the request, finds the corresponding function, and delivers it to the underlying microservice/container platform to be deployed. In most FaaS platforms, such as AWS Lambda and OpenWhisk, the function will be injected into a container, and then the container will be run on a VM or physical machine. If there is not an instance of the requested function, the FaaS platform spins up a new container, injects the function and configures the dependencies, which is known as a *cold start*. Otherwise, another function call will happen inside the already deployed container. A vital question here would be where to place the new container so that we can maintain the desired Service Level Objective (SLO) performance on the one hand and reduce the cost of operation on the other hand? Ideally, the owners of cloud applications aspire to maintain the users' SLO without scaling out the resources, which results in higher cost and increased complexity of their system.

This problem is known as container/function placement, and various solutions have been used. AWS Lambda [30] appears to treat instance placement as a binpacking problem and tries to place a new function instance on an existing active VM to maximize VM memory utilization rates. Container/function placement is unknown in Google Cloud Functions [31]; it appears that Azure Functions [32] does not try to co-locate function instances of the same function on the same VMs, which means it adopted a kind of spread algorithm [10, 13]. The placement algorithm is also unknown for IBM Cloud Functions [33] and Huawei Cloud Functions [34] platforms.

In Docker Swarm [8], the default strategy is *spread* by which the Swarm manager assigns each container to the Swarm node with the most available resources. Swarm also supports *binpack* and *random* strategies. In the binpacking strategy, the manager assigns containers to one Swarm node until it has reached its maximum capacity before assigning them to another one. In the random mode, it assigns containers to

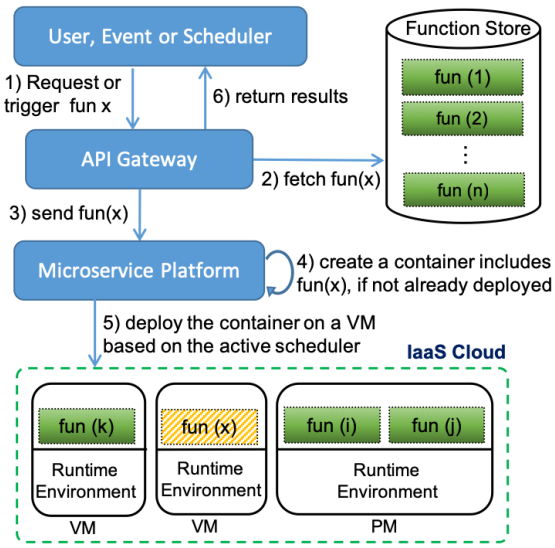


Figure 2.1: Function as a Service (FaaS), a high-level overview.

a random node in the Swarm cluster. Docker swarm also supports filters to tell the Docker Swarm scheduler which nodes to use when creating and running a container. Depending on the filters, it might end up with none, one, or multiple nodes that satisfy the conditions. Kubernetes [35] and Apache Mesos [7], by default, support *spread* strategy for workload placement while they provide an advanced filter and constraint-based scheduling strategies (more or less similar to Docker Swarm) for users.

In summary, all these major container orchestrators treat container/workloads in a black-box manner and use a rigid spread strategy as their best effort to maintain the performance of applications. It would be a great benefit to both FaaS users and provider if there was a more intelligent spread algorithm which provided better performance with the same pool of resources. Our proposed algorithm, hereafter called *smart spread*, strives to fulfill this promise without imposing limiting overhead on the scheduling process. Section 2.2 describes the algorithm in details.

2.2 Smart Spread Algorithm

In this section, we discuss the structure of our proposed smart spread algorithm, a VM selection algorithm used to deploy a new container in serverless computing platforms. Smart Spread algorithm utilizes the performance model powered by machine learning techniques to select the VM that leads to the best performance and cost results for end-users. There exist other common algorithms like binpacking (which sacrifices some level of performance in order to reduce costs), spread (which sacrifices cost trying to balance out all VMs to get the best performance it can), or random placement, but none of these algorithms take the specific characteristics of a workload into account.

In this work, we strive to capture specific characteristics of a workload based on the container's resource utilization on a VM when this container is the only workload being executed. Since there is a huge diversity among workloads in such platforms, we cannot measure the degradation in performance of one container, imposed by co-location with another container. We try to overcome this limitation by developing a predictive performance model that takes any kind of workload and predicts its normalized performance when deployed to a specific VM. Thus the problem of choosing the right VM to deploy the container on is simplified to finding the VM that causes the least degradation in performance of the container. To achieve this, we propose adding a simple profiling step at the time of deployment of a container in the serverless platform, which serves a sample workload specified by the user.

After the profiling step, whenever we need to scale a function, we use the predictive model to find the best performing VM for the function using the resource utilization of different VMs and the workload's profile. Several different data-driven modelling approaches have been investigated for this work (discussed in Section 2.3). Here we chose the artificial neural networks implemented on Tensorflow because of their flexibility, ability to fit nonlinear functions, and low computation costs. We have done comprehensive experimental studies to investigate the effectiveness of this approach.

The details of these implementations and their results are discussed in Section 2.4.

2.3 Machine Learning Models

In this section, a detailed description of our predictive performance model will be laid out.

2.3.1 Features

Features that are being used can be divided into two main categories; those that are reflecting the characteristics of the workload and features that are reflecting the current state of the system. Table 2.1 shows a list of the VM resource utilization features that are used in both features reflecting the characteristics of the workload and features reflecting the current state of the VM. In the former case, we measure these utilization statistics in the profiling phase, where we place the container on a dedicated VM and quantify its characteristics by measuring the utilization and fluctuation of that VM. In the latter case, we quantify the state of a VM before deploying the new container. When these features are collected for the specific function, we normalize the features by removing the mean and scaling them to unit variance.

2.3.2 Performance Normalization

One of our objectives was to design an approach that is capable of working with any kind of workload, independent of their type and complexity. To this end, we need to normalize the throughput to abstract away the specifics of any function type. We used throughput normalization based on the performance of the system when running on an unoccupied VM [36]:

$$tp_{norm} = \frac{tp}{tp_0} \tag{2.1}$$

where tp_{norm} is the normalized throughput, which we intend to predict based on the workload type and the current resource utilization of the VM. tp_0 is the throughput

Table 2.1: Resource utilization statistics used as features (adopted from [37]).

Statistic	**Description**	**Unit**
cpu time	CPU time	ms
cpu usr	CPU time in user mode	ms
cpu krn	CPU time in kernel mode	ms
cpu idle	CPU idle time	ms
contextsw	Number of context switches	count
cpu io wait	CPU time waiting for I/O completion	ms
cpu sint time	CPU time serving soft interrupts	ms
dsr	Disk sector reads	count
dsreads	Number of completed disk reads	count
readtime	Time spent reading from disk	ms
dsw	Disk sector writes	count
dswrites	Number of completed disk writes	count
writetime	Time spent writing to disk	ms
nbs	Network bytes sent	count
nbr	Network bytes received	count
loadavg	Avg # of processes in last min	count
mem used pct	The % of memory currently used	%

of the system in the profiling phase when a container of the function is running on a dedicated VM and tp is the throughput of the system.

2.3.3 Data Collection

In this section, we describe our data collection method for training machine learning algorithms evaluated in our proposed method. To this end, we need to predict the throughput of the system based on the resource utilization statistics of the VM before spinning the container, as well as the characteristics of the workload. To collect

proper data for this problem, our data collection consists of two distinct steps, namely profiling and performance measurement in different scenarios.

In the profiling step of data collection, our main goal is to capture the special needs and characterization of the workload. Therefore, we place the container that needs to be profiled on a dedicated VM and measure several resource utilization statistics listed in Table 2.1 and the throughput on the client's side.

In the second step of data collection, we want to study the effect of VM resource utilization caused by other workloads on the achieved performance of a new container. To do so, we deploy a random number of containers that each generate a random workload to a VM, measure the resource utilization caused by this random workload, and then deploy the new container. Afterward, we observe the achieved performance using Eq. 2.1 and save all the results obtained in our data set for further use by our predictive performance model.

For the model used in our experiments, we collected a total of 183 data points, with 128 data points used as the training set and 55 data points as the test set. All the data and the scripts for training and evaluation purposes can be found in the project Github repository[1].

2.3.4 Regression and Training

In order to build the predictive performance model, we research and examine various machine learning methods to predict the normalized throughput of the serverless platform. Some of these methods include Linear Regression, Support Vector Regression, Decision Tree Regression, Random Forest Regression, and Artificial Neural Networks. Based on the workload characteristics, the performance of the container (i.e., the throughput and response time) of the system changes in a nonlinear fashion. Thus, the performance of the linear models (Linear Regression) is expected to be poor compared to nonlinear methods. This trend was observed in our experiments, which

[1]https://github.com/DDSystemLab/smartspread

Table 2.2: Configuration of the Neural Networks.

Property	**Value**
size	2 layers with 10 and 5 neurons
Activation Function	sigmoid for input and hidden layers, identity function for the output layer
Loss Function	Mean Squared Error
Optimizer	SGD with batch-size of 10 and 1000 epochs of training

showed SVR and Neural Network to have the best performance in terms of accuracy with a slight advantage of Neural Network over SVR. In this work, we used Neural Networks for our experiments because of its agility, prediction speed, generality, and flexibility to fit nonlinear functions. The configuration of the neural networks used can be found in Table 2.2. The code related to the machine learning section and the resulting trained model can be found in the project's Github repository[2].

2.4 Experimental Validation

In order to evaluate the proposed smart spread algorithm, we require full control over the serverless platform for container placement and performance measurements. As a result, we developed a serverless computing platform from scratch based on the architecture depicted in Figure 2.1. Our platform uses Elastic Metric Beats to collect the system information data to be used in the machine learning-based model and Docker for containerization. The codes and scripts developed for the serverless platform can be found in our public Github repository[3]. For our backend cloud platform, we used the Cybera [38] cloud with four nodes working as worker nodes and RabbitMQ as our distributed task queue. The configuration of our worker nodes

[2]https://github.com/DDSystemLab/smartspread-ml
[3]https://github.com/DDSystemLab/smartspread

Table 2.3: Configuration of the VMs in the experiment.

Property	**Value or Include**
vCPU	2
RAM	4GB
HDD	40GB
Network	1Gb/s
OS	Ubuntu 18.04
Python 3.7	numpy, scikit, pandas, tensorflow and keras

is shown in Table 2.3. To manage the Docker containers on workers, we developed a REST API installed on every VM added to the worker fleet to obtain information about containers on nodes (e.g. container count, resource utilization for each, etc.).

We used three different generic workloads to imitate all possible workloads that FaaS might process. Generally, various workloads can be classified as CPU intensive, memory-intensive, or I/O (i.e. disk and network) intensive. To show the effectiveness of the smart spread algorithm, we implemented each type of application (i.e., function) in a dedicated container image to be instantiated on the serverless platform. In each workload, we used a web server that runs a benchmark upon receiving a user request. Table 2.4 summarizes the configuration of these benchmarks and the options that we used in this study.

For the CPU and I/O intensive containers, we used the alpine image, and for the memory-intensive workload, we used the MySQL 5.7 image, all obtained from the Docker hub. Each running container was bound to use only 512 MB of memory, 50% of a CPU core and 7.15 MB/s of disk I/O (half the available disk speed of the VMs achieved through benchmarks), thus making the capacity of each VM to be 7

Table 2.4: Benchmarks used in each application type. Each type has been containerized.

Application Type	**Benchmark**	**Configuration**
CPU Intensive	Sysbench CPU [39]	max-requests=2500 cpu-max-prime=1000
Disk I/O Intensive	FileIO [39]	max-requests=200 file-test-mode=rndrw
Memory-Intensive	OLTP [39, 40]	table-size=10000 table-count=3 max-requests=10

containers. In order to avoid any potential crash, we limited this number to 6, thus making the total number of containers (i.e., capacity) in the system to 24. To trigger the autoscaling, we used a simple pure reactive algorithm. For each function, we measure the total response time for each request, and if it is beyond a predefined threshold, which depends on the function type, new containers of that type will be added to the fleet to maintain the SLO response time.

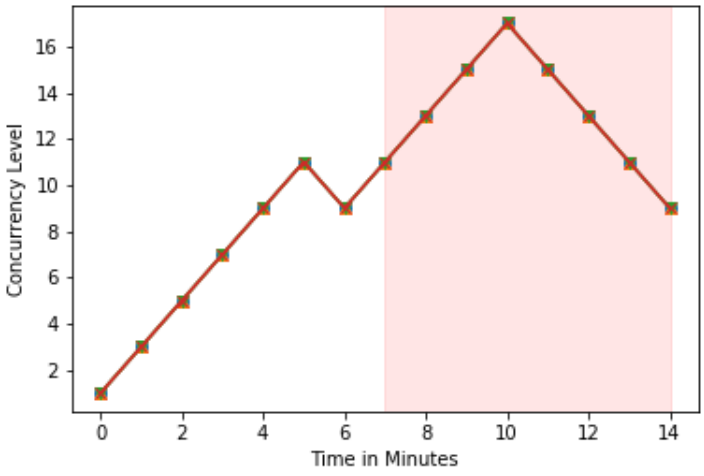


Figure 2.2: Concurrency level sequence used for our tests.

For our experiments, we use the concurrency level test in which each user will make a request, wait for the request to be responded and make another request as soon as it has received the response for the previous one. The sequence used for

the level of concurrency for our tests is shown in Figure 2.2, in which the horizontal axis is the time (minutes) and the vertical axis is the concurrency level (i.e., active users) at each point in time for all workloads. Note that concurrency level changes at the same time for all workloads. Also, note that the number of users has been chosen to be low, with each request requiring more computation to avoid the effect of network congestion. Each time index represents a minute indexed by k at the start of which we start c_k concurrent clients making requests to the server. The results shown in Figures 2.3 to 2.6 are all aggregated results of each minute by taking the average of that time period. To remove the effect of network latency, all tests have been performed from a VM inside the internal network of the servers with single-digit millisecond latency to the server. To get the steady-state throughput and response time at each time interval, we assumed a warmup time of 30 seconds in which the simulated client will make requests to the server, but their results are not recorded and not shown in these figures. As can be seen, the tests have been designed in a way that causes all algorithms to reach the limit of containers that can be used in this system to keep the response time below the desired threshold.

To show that the smart spread algorithm imposes negligible overhead to the serverless computing platform, we performed an analysis to identify the processing time for finding the best performing VM in the pool, as well as the effects of scaling on the overhead. To do so, we performed 100 iterations of VM selection, and the average of results are shown in Table 2.5. All experiments reported have been done on the same VM that was used for the experiments, which has no special-purpose hardware. As can be seen in Table 2.5, in comparison with other proposed algorithms, the calculation time of the smart spread algorithm is larger but still negligible since the lifetime of a container is usually much more than a few seconds [10]. In much larger VM pools, dedicated hardware (GPGPUs) could be utilized to reduce the overhead of the system further.

Figures 2.4 to 2.6 show the average throughput per container, response time, total

throughput, and container count for the Sysbench, FileIO, and OLTP functions, respectively. Figure 2.3 shows the aggregated results of Figures 2.4 to 2.6. Please note that the aggregated throughput per container is the sum of throughput per container for each application type (see Table 2.4) and not the aggregated throughput divided by the aggregated number of containers. This is to avoid favouring the application type with higher throughput in our results. As we mentioned before, we use the response time to trigger the scaling of each function. In the first half of our tests (i.e., light load), since none of the resources in the VMs are congested, we have a linear throughput in the system for all the placement algorithms. However, when we approach the second half of the experiment (i.e., the shaded area in the graph which shows the heavy load region), different approaches tend to give different results. As could be expected, since binpacking sacrifices performance to achieve lower costs, it has the worst throughput in the system. In contrast, since spread sacrifices cost to achieve the best performance it can (without considering the characteristics of the workload), it performs best among the algorithms that discard the workload specifications. For almost all function types, random placement of containers will result in performance measurement in between these two algorithms. However, the proposed smart spread algorithm tries to maximize the output of each function by choosing the VM that gives the best result for that function in that time slot.

Table 2.5: Processing overhead of placement algorithms.

Algorithm	**Our Setting**	**Using k VMs**	**Unit**
Random	0.79	0.79	millisecond
Binpack	0.49	$0.1225 \times k$	millisecond
Spread	0.48	$0.12 \times k$	millisecond
Smart Spread	10.16	$2.54 \times k$	millisecond

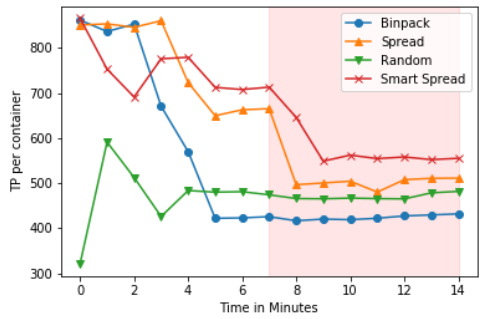
As can be seen in Figures 2.3a and 2.3b, before we reach the limitation on the total number of containers that can be placed in the system, the proposed smart spread algorithm achieves the highest throughput among all algorithms used in this experiment while maintaining the lowest container count. As a result, the proposed algorithm helps reduce the number of required containers (see Figure 2.3d under heavy load), which can be translated into less runtime operational cost for application owners and reduced carbon footprint without sacrificing the performance and end-user satisfaction.

Table 2.6: Average aggregated throughput (Thro) and response time (RT) for each placement algorithm in heavy load region. The relative throughput (Rel Thro) and relative response time (Rel RT) are relative values to smart spread algorithm result.

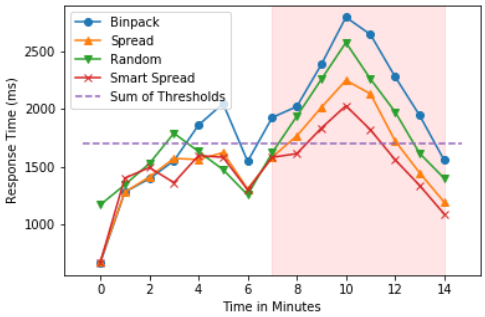
Algorithm	**Thro**	**RT**	**Rel Thro**	**Rel RT**
Binpack	2966.38	2193.39	0.68	1.36
Random	3523.75	1954.76	0.81	1.24
Spread	3934.62	1763.33	0.91	1.10
Smart Spread	4341.62	1608.49	1	1

Table 2.6 shows the average aggregated results of the heavy load region. The aggregated throughput in this table shows the sum of throughputs and the aggregated response time shows the sum of the response times of the functions. The relative throughput and response time are the ratios of the aggregated throughput and response times over the proposed smart spread algorithm. As can be seen in table 2.6, the smart spread function placement improves the throughput and response time between 10-36% and 9-32% respectively when compared to baseline algorithms.

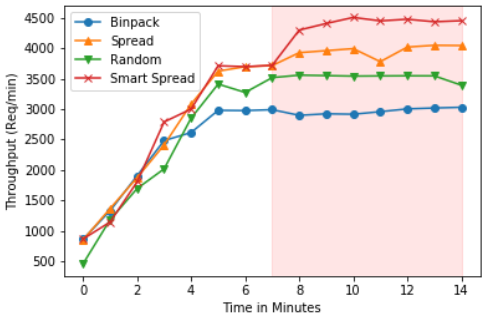
Figure 2.4 shows the results obtained for the Sysbench CPU application. We expect to achieve the best performance of an application when it is the only task being run on the server. But the results for CPU intensive tasks are somewhat counter-intuitive



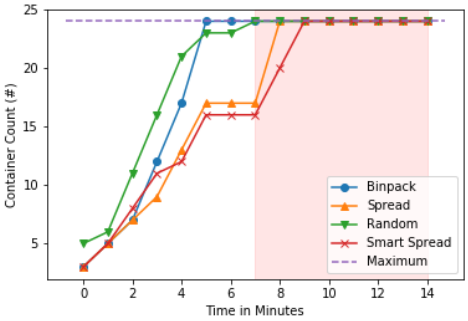
(a) The aggregated throughput per container.



(b) The aggregated response time; the dashed line shows the sum of thresholds used for the functions.



(c) The aggregated throughput.

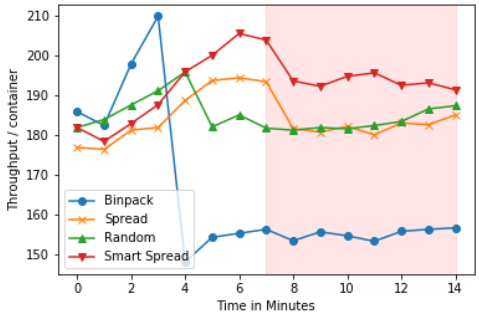


(d) The aggregated number of containers; the dashed line shows the capacity of the system.

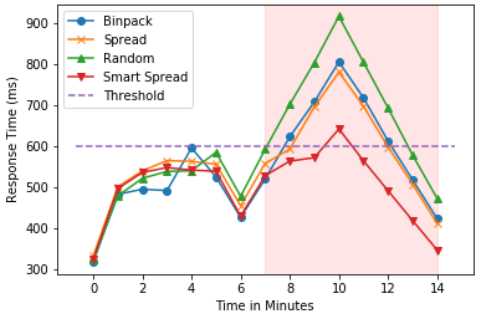
Figure 2.3: The aggregated experimental results for the three functions. Values have been derived by adding the results from Figures 2.4 to 2.6. Note that the aggregated throughput per container is the sum of throughput per container for each application.

since we see a performance boost of up to 13% for each task when more than one CPU intensive task was being run on the VM. This is probably due to the adaptive behaviour of the underlying scheduler that handles CPU time allocation for each VM and the adaptive behaviour of the CPU under heavy load. But as you can see, as the number of containers exceeds 4 (when we reach maximum CPU capacity), we observe a large degradation of performance due to using up the resources available. We observe that this is, to some extent, automatically captured by the smart spread, which leads to the best performance under heavy load.

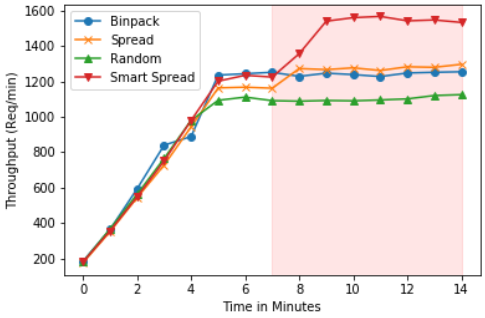
Intuitively, considering how the contention in disk I/O causes performance to drop, an algorithm that works similar to spread should yield the best possible performance



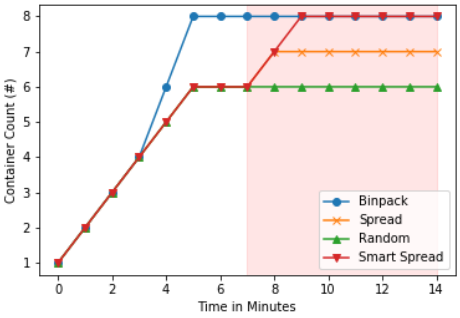
(a) The throughput of the Sysbench CPU image divided by the total number of containers.



(b) The response time of the Sysbench CPU workload (i.e., function). The dashed line shows the threshold used to trigger autoscaling of the function.



(c) The throughput of the Sysbench CPU workload.

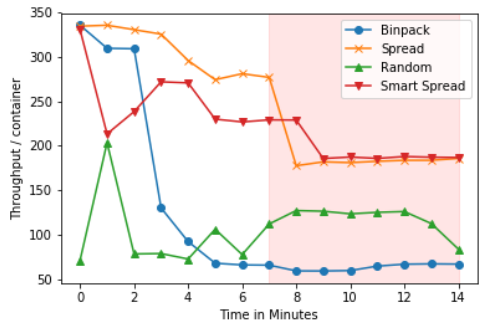


(d) The number of containers created for Sysbench CPU image in each test.

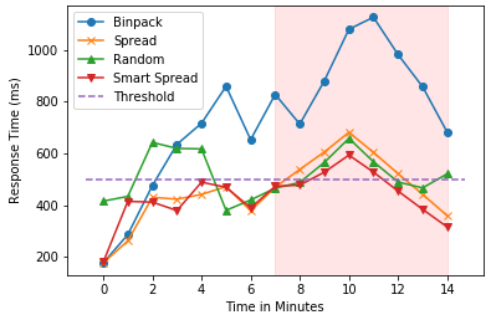
Figure 2.4: Experimental results for the Sysbench which is a CPU intensive workload. The results are obtained when all application types are applied to the platform which leads to different saturation levels for container counts.

for FileIO at all times. We can see in Figure 2.5a that such behaviour is observed between the Random, Binpack, and Spread algorithms. Also, it can be observed that smart spread achieves similar results with a slight advantage under heavy load mainly due to better management of other resource types (CPU and memory).

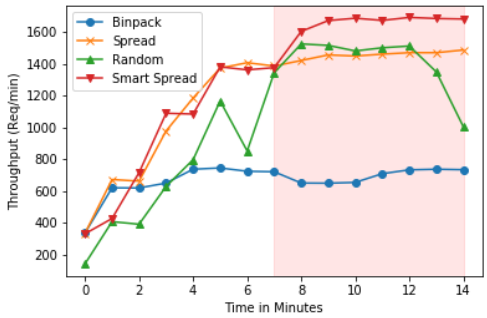
Figure 2.6 shows the results for the OLTP application. Since we ensure that there is always enough memory available to the application in order to prevent the system from crashing, the OLTP benchmark is not influenced much by the selection of the container placement algorithm and thus has little implications for system-wide performance evaluation. Please note that different saturation levels observed for the



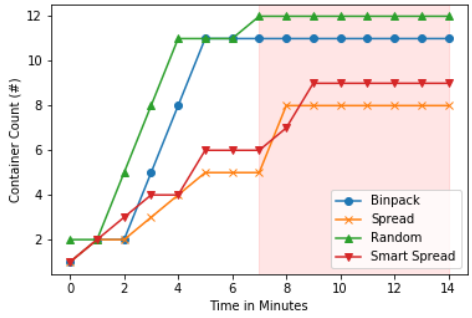
(a) The throughput of the FileIO image divided by the total number of containers.



(b) The response time of the FileIO workload. The dashed line shows the threshold used to trigger auto-scaling of the function.



(c) The throughput of the FileIO workload.

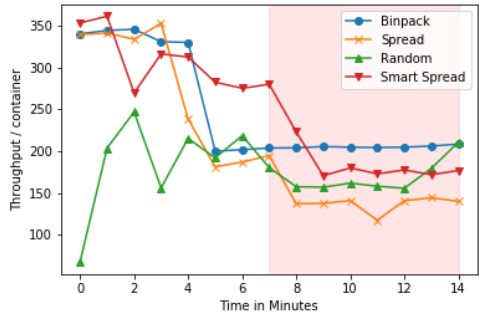


(d) The number of containers created for FileIO workload in each test.

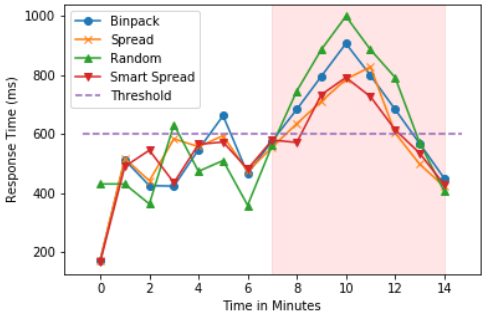
Figure 2.5: Experimental results for the FileIO which is a I/O intensive workload. The results are obtained when all workload types are co-located on the platform which leads to different saturation levels for container counts.

container count for different placement algorithms in Figure 2.6d are due to the fact that all container types are being scaled at the same time for each algorithm. Thus, adding up the saturation levels for Figures 2.4d, 2.5d and 2.6d for each algorithm will be equal to the system capacity (24 containers).

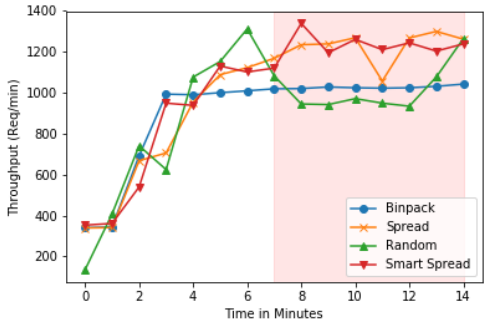
Overall, many manual rules can be found for near-optimal placement of different applications on serverless platforms using the knowledge of the underlying system and thorough experimentation. But for the serverless service providers, the applications appear as black-box models, and thus such rules cannot be found for each application at scale. In addition, as we saw in our experiments, there is no single algorithm that guarantees the best placement at all conditions and for all types of workload without



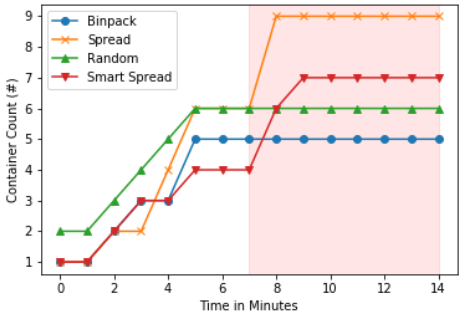
(a) The throughput of the OLTP image divided by the total number of containers.



(b) The response time of the OLTP workload. The dashed line shows the threshold used to trigger auto-scaling.



(c) The throughput of the OLTP workload.



(d) The number of containers created for OLTP workload.

Figure 2.6: Experimental results for the OLTP which is a memory-intensive workload. The results are obtained when all workload types are co-located on the platform which leads to different saturation levels for container counts.

considering the characteristics of the workload. The method presented in this work tries to capture the subtle interaction between the workload and the VM status using black-box performance models and thus can be applied to unknown applications to provide self-optimization without the need for human supervision.

2.5 Related Work

The work in the literature for this research can be divided into 3 main categories: A) workload profiling and other methods of building predictive models for different types of workloads and applications; B) batch job runtime prediction and other research

that try to predict the performance of these types of workloads; and C) works that use machine learning as their primary performance modelling tool for distributed systems.

2.5.1 Workload Profiling and Application Modelling

Many recent studies have focused on workload profiling and application modelling to improve the scheduling algorithms used in workload orchestration platforms. In [41], Lioyd et al. used workload profiling using benchmarks to predict the cost of workloads across different settings, to analyze unique resource requirements of very diverse workloads. Their work assumed a stationary environment of homogeneous VM capacity and initial settings. However, in managing serverless infrastructure, the servers' workload capacity and resource utilization are very dynamic. The authors in [13] investigate the factors influencing microservices. Several factors influence a container's performance like the state, memory limit, concurrent users, and several other factors for commercial platforms. They try to find out the most limiting factors influencing the performance of a container. We found that some of the significant predictors in their work haven't contributed much to the performance predictions in our settings.

Profiling can give us extra information about the microservice that is going to be deployed. One example is giving us an insight into how much the underlying VM is going to be affected by the execution of a container. In [42], the authors developed a Cloud-Scale Java profiler, which helped the developers find out the performance-related problems with their applications as well as giving them insight into the throughput of their system and the resources each microservice is going to require to reach a certain quality of service. In [36], The use of profiling and normalized performance to optimize the workload performance and model the effect of current and co-locating running VMs has been investigated. They proposed an algorithm that increases workload performance while reducing the energy consumption

of the PMs using VM migration. Although they raise very interesting points, their algorithm cannot work under a diverse set of workloads.

Li et al. [43] use analytic models based on queuing theory to optimize performance levels of composite service application jobs by tuning job configurations and resource allocation decisions. In [44], Apte et al. propose a load generator tool aimed especially at multi-tier internet server applications for capacity analysis and profiling. Their work aims at generating a detailed server resource usage profile per request type.

Chen et al. [45] try to predict the performance of enterprise applications based on technologies like COBRA and J2EE using benchmarking and profiling. In their work, they assume a performance model of the application will be built on a machine close to the production environment. The heterogeneous nature of serverless computing platforms is a limiting factor for this method. In addition, they don't consider the variable runtime environment common in serverless computing.

Sadjadi et al. [46] proposed a dynamic execution time prediction using regression on an increasing number of nodes. In their work, they did not consider the possibility of other workloads being executed on the same node, which could affect the performance of an application.

Liu et al. [47] utilized a multi-objective optimization to find an optimal placement of containers. In their work, they consider the variable runtime environment of the nodes but assume only one application is being executed in the cluster. Also, they do not perform any generalization and thus have to perform the optimization for each of the applications separately.

2.5.2 Batch Job Runtime Prediction

Lloyd et al. [37] use Physical Machine (PM) and Virtual Machine (VM) resource utilization statistics to build workload models to predict the task completion time of a workload. This is similar to what we propose for containers in this chapter. They also provide a comprehensive study of the metrics that affect the workload completion

time the most. In [48], the authors try to predict the runtime of batch workloads when running multiple parallel batch jobs on the same machine while sharing resources like CPU and RAM.

In [49], Gribaudo et al. presented a model based on multiformalism and multisolution techniques to predict the overall performance of lambda architecture. They argue that such system could support the design and assessment process in order to optimize the overall throughput of the system while keeping the QoS of the resulting lambda architecture above the required level of agreement.

2.5.3 Machine Learning for Performance Modelling

Many works in the literature have successfully applied machine learning for performance modeling of distributed applications [50–53]. The main reason is that in the increasingly complex systems, the analytical performance modeling is a long and error-prone task which cannot capture the full complexity of the system and the complicated architecture of current systems. In such situations, machine learning techniques can help improve the accuracy of performance modeling while being fast, automated, and able to capture the full complexity of the application and its architecture [53, 54].

Yadwadkar [55] tries to choose the best VM type to optimize the cost for the user while maintaining the target performance required by the user. This work uses workload profiling as well to predict the performance of the system on different VMs but assumes only one application will be running on a VM instance and doesn't take the dynamic nature of running multiple applications on one VM into account. The authors in [56] introduce a closed-loop controller based on a model predictive controller for the lambda architecture. The proposed system does this by considering the resource inference among adjacent lambda functions in the allocation phase. They try to build a decision support mechanism based on machine learning to provide serverless architecture platforms with the tools to use model predictive control frameworks

in their systems. While the methods used in this chapter try to use machine learning to improve serverless architecture, their method needs to collect data and perform the training phase for each workload separately in different settings and this leads to a large number of models and training phases that are imposed on the serverless platform.

In [57], Xu et al. introduce a unified reinforcement learning methodology for tuning the configuration of different VMs to maximize long-term delayed performance reward for the agent. In this work, the authors focused on CPU and memory allocation, while depending on the application, network I/O and disk I/O could have a significant impact on VM or container's performance.

2.6 Conclusion

In this chapter, we proposed and evaluated a new container/function placement algorithm based on Tensorflow neural networks to increase the performance of functions compared to the current placement algorithms being used by FaaS providers. We trained and evaluated fifteen different machine learning approaches for this purpose and formulated the best, called the *smart spread*, in terms of improvement, accuracy, and overhead. To evaluate the smart spread algorithm, we developed a serverless computing platform from scratch and implemented three of the most well-known placement algorithms along with our proposed smart spread algorithm. We carried out extensive experiments in all of which the supremacy of smart spread was shown. It is straightforward to add smart spread to FaaS provider scheduler as well as container orchestration tools such as Docker Swarm, Kubernetes, and Apache Mesos.

Chapter 3

Performance Modelling of Scale-Per-Request Serverless Computing Platforms

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 met-

rics 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.

3.1 Analytical Performance Model

Section 1.2 briefly outlines the scheduling algorithm used for the serverless computing platforms and the one that we consider in this section, 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 3.1, 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 3.1.

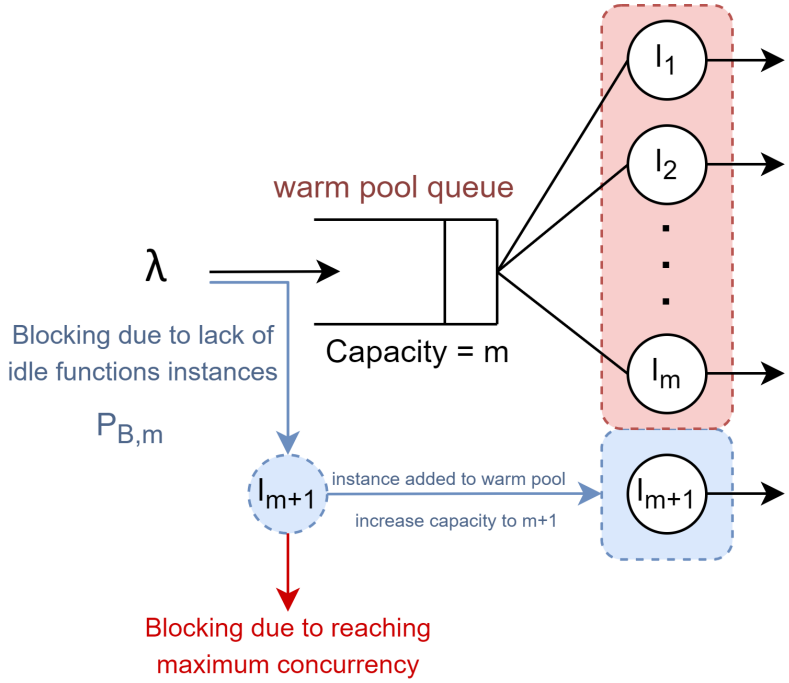


Figure 3.1: An overview of the proposed system model using $M/G/m/m$ loss systems. In the case of workload fluctuation, m will change during the runtime, just like a delay center. The blue arrows show the path a successful cold start goes through.

3.1.1 Cold Start Rate

As can be seen in Fig. 3.1, 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

Table 3.1: 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

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 [58]. 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):

$$\rho = \lambda/\mu_w \tag{3.1}$$

Then, the Erlang's B formula is obtained as [59]:

$$P_{B,m} = B(m, \rho) = \frac{\frac{\rho^m}{m!}}{\sum_{j=0}^{m} \frac{\rho^j}{j!}} \tag{3.2}$$

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:

$$\lambda_{w,m} = \lambda(1 - P_{B,m}) \tag{3.3}$$

Using eq. (3.2), we can derive the rate at which cold starts are happening in the system.

$$\lambda_{c,m} = \lambda P_{B,m} \tag{3.4}$$

Figure 3.1 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.

3.1.2 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 1.2, 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. (3.2), 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:

$$P_{S,n} = P_{B,n-1} - P_{B,n} \tag{3.5}$$

$P_{S,n}$ 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 $\{I_n; 1 \leq n \leq m\}$:

$$\lambda_{w,m,n} = \lambda_{w,m} P_{S,n} \tag{3.6}$$

3.1.3 Server Expiration Rate

In eq. (3.6), 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:

$$P(X = x) = \lambda_{w,m,i} \cdot e^{-\lambda_{w,m,i}x} \tag{3.7}$$

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

$$P(X \leq x) = 1 - e^{-\lambda_{w,m,i}x} \tag{3.8}$$

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:

$$P_{lst,m,i} = P(X \geq T) = e^{-\lambda_{w,m,i}T} \tag{3.9}$$

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 3.2) 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:

$$C_{req,m,i} = \frac{1}{P_{lst,m,i}} \tag{3.10}$$

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 $T = T_{exp} + 1/\mu_w$:

$$
\begin{aligned}
E[X; X < T] &= \int_0^T x\lambda_{w,m,i} e^{-\lambda_{w,m,i}x} dx \\
&= -x \cdot e^{-\lambda_{w,m,i}x}\Big|_0^T + \int_0^T e^{-\lambda_{w,m,i}x} dx \\
&= -T \cdot e^{-\lambda_{w,m,i}T} - \frac{e^{\lambda_{w,m,i}x}}{\lambda_{w,m,i}}\Big|_0^T \\
&= -T \cdot e^{-\lambda_{w,m,i}T} + \frac{1 - e^{\lambda_{w,m,i}T}}{\lambda_{w,m,i}}
\end{aligned}
\tag{3.11}
$$

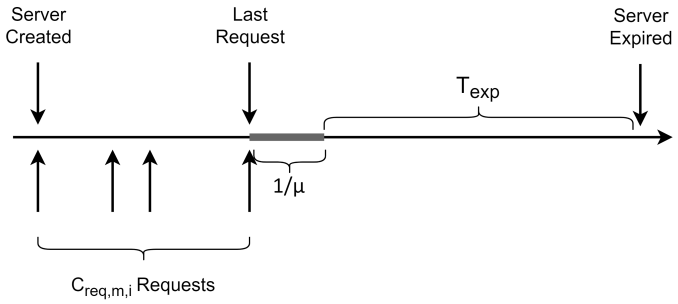


Figure 3.2: The server lifespan calculation overview.

Thus, the average lifespan of a server in the warm pool could be calculated as follows (Figure 3.2):

$$
E[LS_{m,i}] = \{(C_{req,m,i} - 1) \cdot E[X; X < T]\} + \frac{1}{\mu_w} + T_{exp} \tag{3.12}
$$

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 $E[LS_{m,i}]$:

$$
R_{exp,m,i} = \frac{1}{E[LS_{m,i}]} \tag{3.13}
$$

which gives us the server expiration rate for I_i. Expiring 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:

$$
R_{exp,m} = \sum_{i=0}^{m} R_{exp,m,i} \tag{3.14}
$$

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

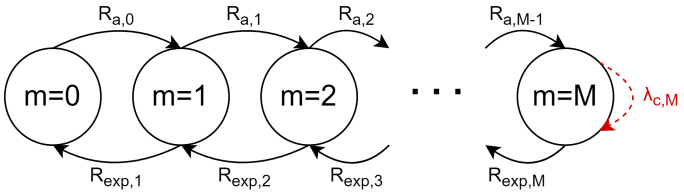


Figure 3.3: The state transition diagram of the warm pool in serverless platforms. This is a Semi-Markov process for which we provide a closed-form steady-state solution. The dashed red self-loop shows rejected requests due to insufficient capacity.

3.1.4 Modelling the Warm Pool

In previous sections, 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 approximate closed-form steady-state solution. The process is not Markovian since, as can be seen in Figure 3.2, the lifespan of servers, i.e., the states' holding time, is clearly not exponentially distributed. Figure 3.3 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:

$$R_{a,m} = \frac{1}{\frac{1}{\lambda_{c,m}} + \frac{1}{\mu_c}} = \frac{\lambda_{c,m} \cdot \mu_c}{\lambda_{c,m} + \mu_c} \tag{3.15}$$

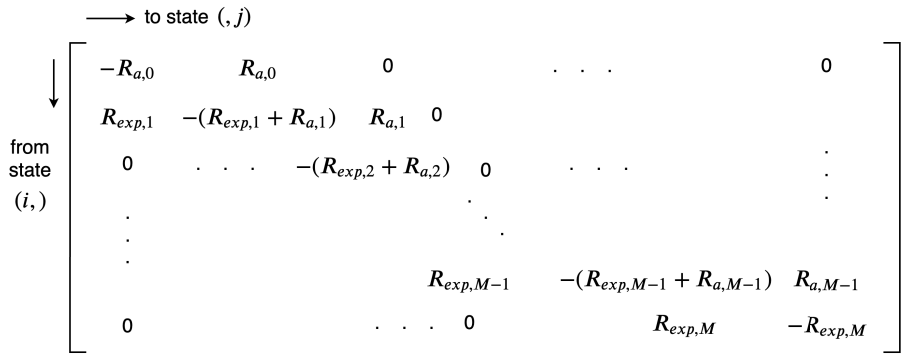


Figure 3.4: One-step transition rate matrix for the proposed model.

3.2 Steady-State Solution

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 3.4 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:

$$Q_{i,i} = -\sum_{j \neq i} Q_{i,j} \tag{3.16}$$

The steady-state distribution π is the unique solution to the following equation system [60]:

$$\pi \cdot Q = 0 \text{ and } \sum_{m=0}^{M} \pi_m = 1 \tag{3.17}$$

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:

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

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;

pool will be rejected by the system. Thus, the probability of rejection for a given request can be calculated as the following:

$$P_{rej} = P_{B,M}\pi_M \tag{3.18}$$

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:

$$P_B = \sum_{m=0}^{M} P_{B,m}\pi_m \tag{3.19}$$

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:

$$P_{cld} = P_B - P_{rej} \tag{3.20}$$

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

$$RT_{avg} = RT_w(1 - P_B) + RT_c P_{cld} \tag{3.21}$$

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 [61]. 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:

$$C_w = \sum_{m=0}^{M} m\pi_m \tag{3.22}$$

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:

$$\begin{aligned} C_{r,w,m} &= RT_w \lambda_{w,m} = RT_w \lambda (1 - P_{B,m}) \\ C_{r,w} &= \sum_{m=0}^{M} C_{r,w,m} \pi_m \end{aligned} \tag{3.23}$$

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:

$$\begin{aligned} C_{r,c,m} &= \begin{cases} 0 & \text{if } m = M \\ RT_c \lambda_{c,m} = RT_c \lambda P_{B,m} & \text{otherwise} \end{cases} \\ C_{r,c} &= \sum_{m=0}^{M-1} C_{r,c,m} \pi_m \end{aligned} \tag{3.24}$$

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

$$C_r = C_{r,w} + C_{r,c} \tag{3.25}$$

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:

$$C_i = C_w - C_{r,w} \tag{3.26}$$

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:

$$U = \frac{C_{r,w}}{C_w} = \frac{RT_w\lambda(1 - P_{B,m})}{\sum_m m\pi_m} \tag{3.27}$$

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.

3.2.1 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 $O(M^3)$.

3.3 Steady-State Solution 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.

3.3.1 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. [10] with minor modifications and is openly available in our Github repository[1]. 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[2] 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

[1] https://github.com/pacslab/serverless-performance-modeling/tree/master/deployments
[2] https://man7.org/linux/man-pages/man1/dd.1.html

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[3] 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[4] which is openly accessible 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[6].

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.

3.3.2 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

[3]https://docs.computecanada.ca/wiki/Cloud_resources
[4]https://github.com/pacslab/pacswg
[5]https://pypi.org/project/pacswg
[6]https://github.com/pacslab/serverless-performance-modeling

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 [27, 62].

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.

3.3.3 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 unique identifier for each function instance to keep track of their life cycle, as obtained in [10].

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:

$$U = \frac{C_{r,w}}{C_w} \tag{3.28}$$

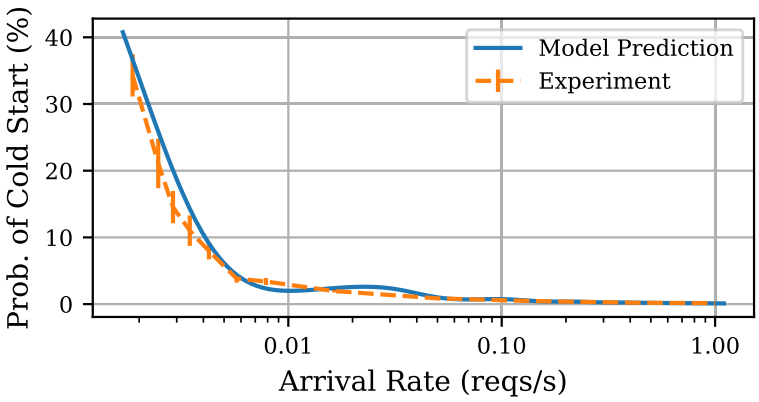


Figure 3.5: Probability of cold start against arrival rate. The vertical bars show one standard error around the measured point.

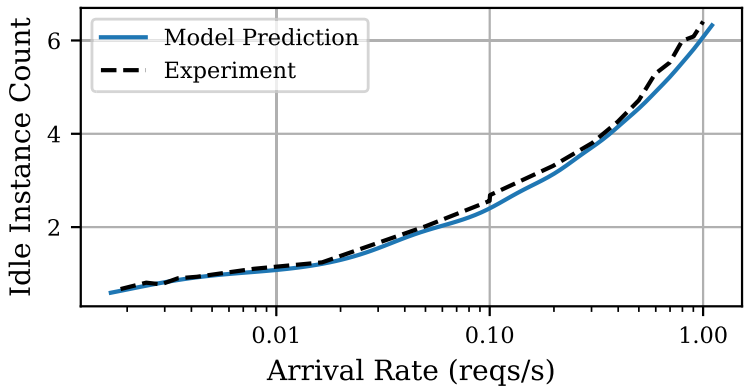


Figure 3.6: The number of idle servers against arrival rate.

3.4 Steady-State Solution Results and Discussion

Figures 3.5 to 3.7 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

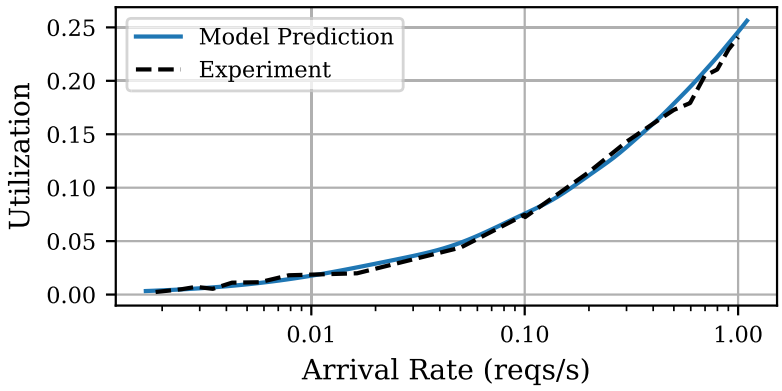


Figure 3.7: Utilization against arrival rate.

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.

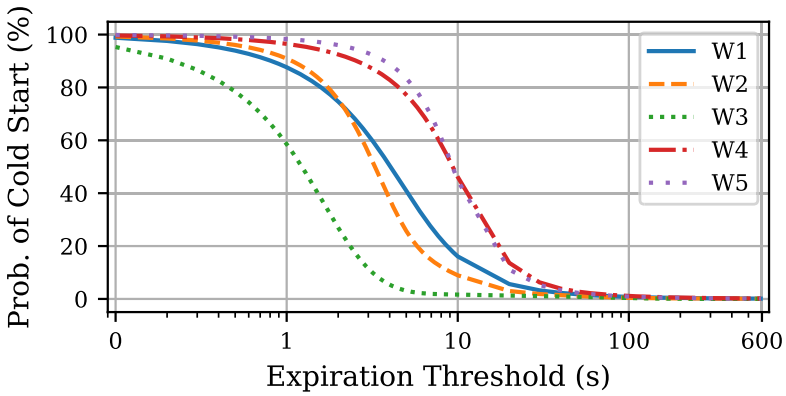


Figure 3.8: Cold start probability against the *expiration threshold*. The arrival rate has been set to 1 request per second. The legends denote warm and cold service times. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

Section 3.4 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 [63]. 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 1.2. The most important criterion is scale-per-request behaviour (with no queuing). For exam-

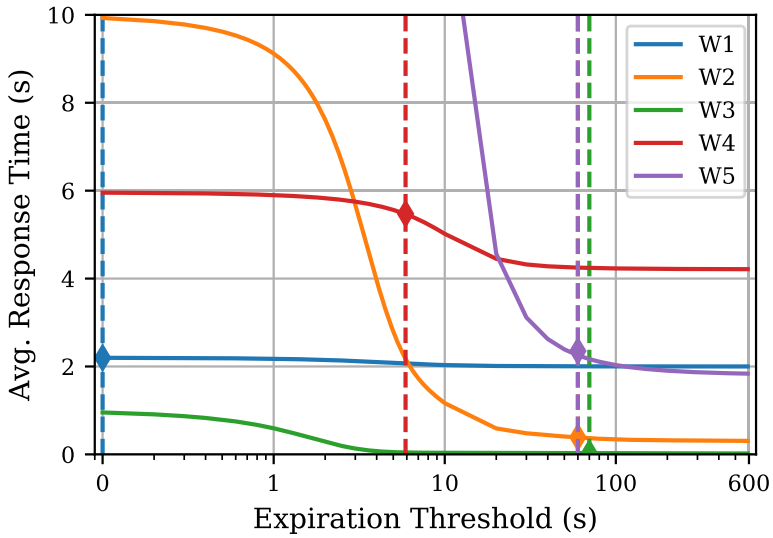


Figure 3.9: Average response time against the *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds. The vertical lines show the minimum expiration threshold for which the average response time is at most 30% higher than the average warm start response time.

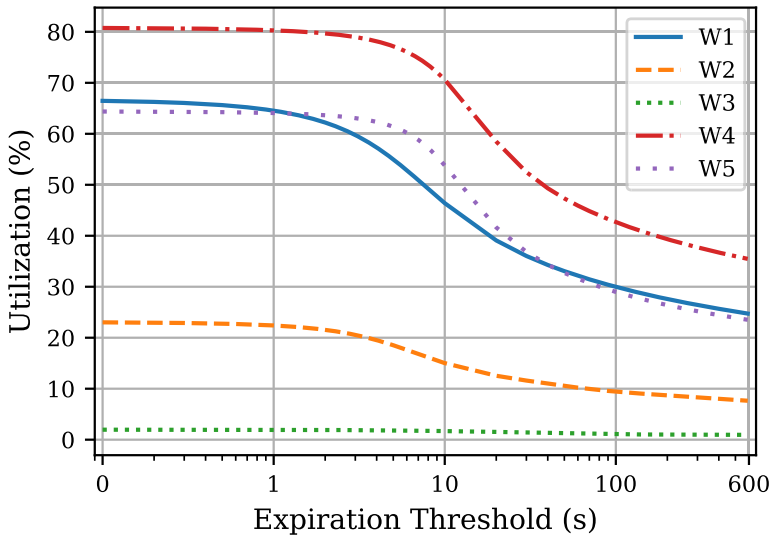


Figure 3.10: Utilization against the *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

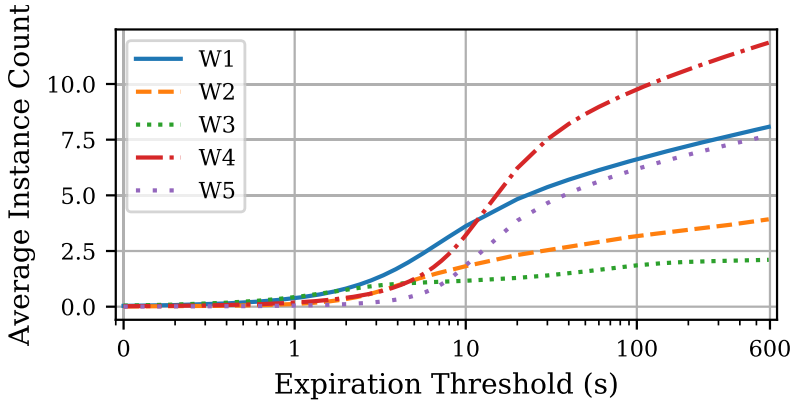


Figure 3.11: Average instance count against the *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

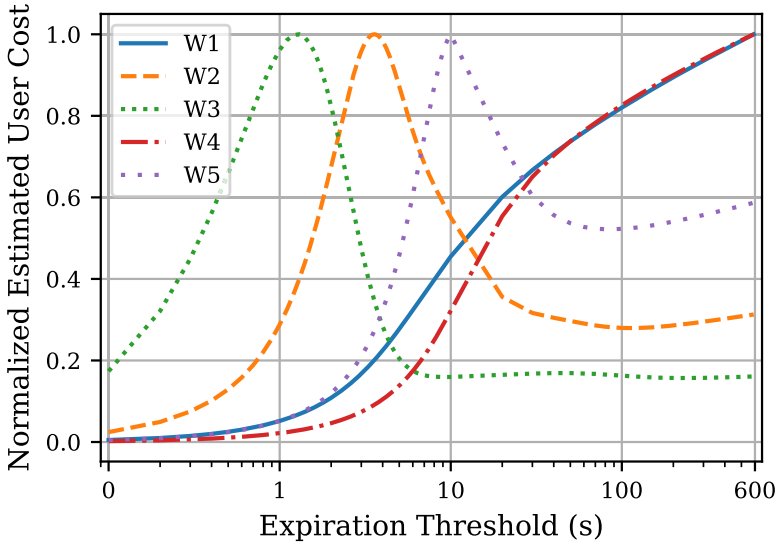


Figure 3.12: Average estimated user cost against *expiration threshold*. The arrival rate has been set to 1 request per second. Note that the x-axis is on a logarithmic scale and changes from 0.1 to 600 seconds.

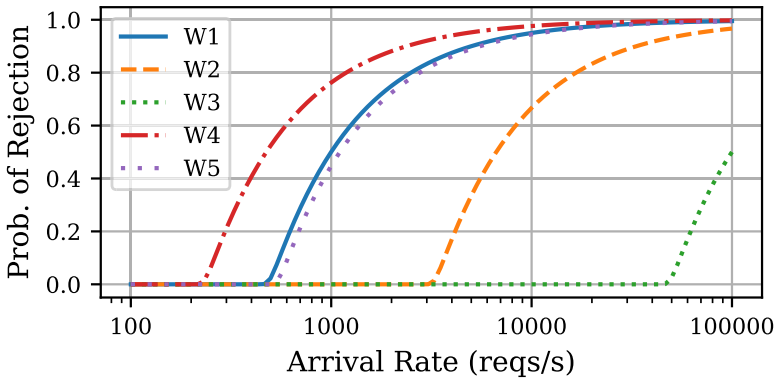


Figure 3.13: Probability of rejection against the arrival rate. The expiration threshold has been set to 10 minutes, and the maximum concurrency is 1000. Note that the x-axis is on a logarithmic scale.

ple, Google Cloud Functions [64], Azure Functions [65], IBM Cloud Functions [66], and Apache OpenWhisk [67] work in a similar fashion, but Google Cloud Run [68], OpenFaaS [69], Kubeless [70], and Fission [71] 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 Chapter 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 3.8 to 3.12 depict the effect of *expiration threshold* on different system characteristics for different workloads with varying warm and cold service times shown in Table 3.2. 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 while keeping an eye on the average response time.

Figure 3.11 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 3.12 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.

Figure 3.10 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 3.10, increasing the *expiration threshold* causes utilization to decrease, while for many workloads, as shown in Figures 3.8 and 3.9, it wouldn't lead to a noticeable improvement in the quality of service. Considering this effect, we see potentials for substantial savings in

Table 3.2: 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 [10]	2000	2200
W2	A rang of benchmarks with different configuration [72]	300	10000
W3	Startup test with echo on Apache OpenWhisk [73]	20	1000
W4	Fibonacci calculation on AWS Lambda [17]	4211	5961
W5	Fibonacci calculation on Azure Functions [17]	1809	26681

infrastructure costs for providers, which could potentially lead to greener computing and emission reductions.

Figure 3.13 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.

3.5 Temporal Solution

Figure 3.4 shows the one-step transition matrix Q used to calculate the state distribution π for the proposed SMP. In this matrix, each element located in row i and column j shows the transition rate at which we transition from state i to state j. Diagonal elements are defined in a way to satisfy the following:

$$Q_{i,i} = -\sum_{j \neq i} Q_{i,j} \tag{3.29}$$

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

$$\frac{d\pi}{dt} = \pi Q \Rightarrow \pi(t) = \pi(0)e^{Qt} \tag{3.30}$$

which can be calculated using the method proposed by Al-Mohy et al. [74] implemented in SciPy[7].

Using π, we can calculate the average number of instances in the warm pool C_w using the following:

$$C_w = \sum_{m=0}^{M} m\pi_m \tag{3.31}$$

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

We can also calculate the average number of servers running warm and cold requests in each state using $C_{r,w,m} = RT_w \lambda_{w,m}$ and $C_{r,c,m} = RT_c \lambda_{c,m}$:

$$C_r = \sum_{m=0}^{M} C_{r,w,m} \pi_m + \sum_{m=0}^{M-1} C_{r,c,m} \pi_m \tag{3.32}$$

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

$$U = \frac{C_{r,w}}{C_w} = \frac{RT_w \lambda (1 - P_{B,m})}{\sum_m m \pi_m} \tag{3.33}$$

3.6 Temporal Solution Experimental Validation

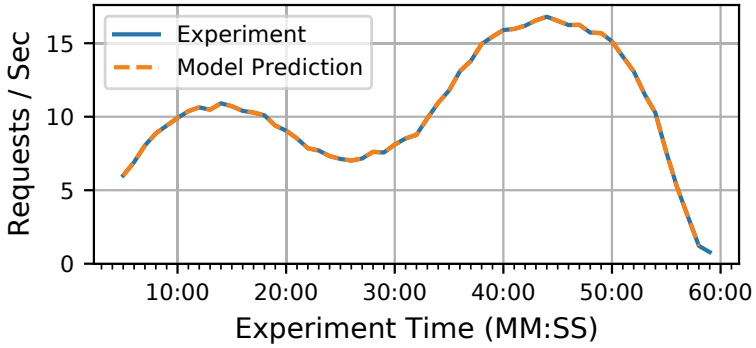


Figure 3.14: Request arrival rate throughout the experiment, along with the oracle predictions.

Figure 3.14 shows the request arrival rate over time designed for this experiment. Since workload prediction is out of the scope of this paper, we used an oracle request rate predictor for our experiments. Another exogenous parameter for the proposed model is the *expiration timeout*, which has been set to 10 minutes for AWS Lambda, also verified by Shikov [62] and Shahrad et al. [27]. The average cold and warm response time (RT_c and RT_w) are set to the average of all requests over all experiment repetitions, which doesn't change over time in modern serverless computing platforms.

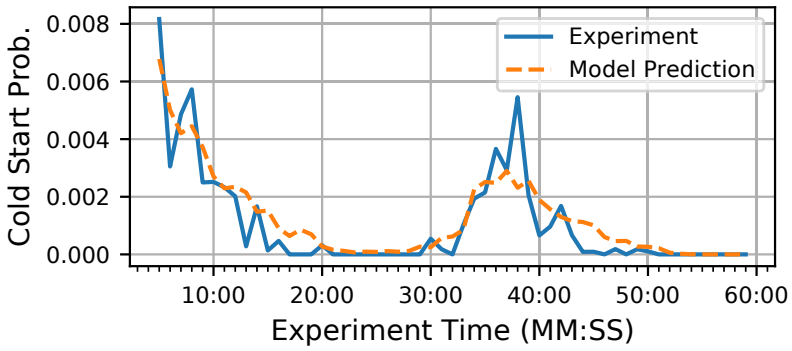


Figure 3.15: Probability of a cold start occurrence throughout the experiment along with the model predictions.

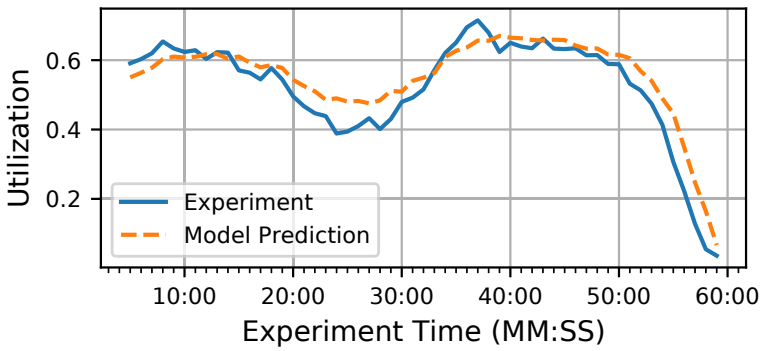


Figure 3.16: Utilization of resources in the warm pool throughout the experiment compared with the model predictions.

3.7 Temporal Solution Results and Discussion

Figure 3.15 shows the average probability of a cold start over all of the performance experiments. This metric is the most important factor in deciding the quality of service observed by the application user. Since cold starts could be orders of magnitude longer than warm starts, having a large probability of cold start could affect the user experience. In many applications (especially customer-facing applications), it is important to limit the probability of a cold start. Thus, predicting this value is very important to enable migration to serverless computing platforms for many different applications. Figure 3.16 depicts the average utilization of resources over time in our experiments. Assuming a serverless provider is using some kind of Infrastructure-as-a-Service (IaaS) platform underneath, the number of function instances is proportional to the operational costs incurred by the platform. In this work, the utilization of resources is defined as the average ratio of the time that these instances are being utilized and thus are billed for the application developer. To lower the costs, serverless platforms would want to maximize the average utilization. Using the proposed model, the platform can make predictions for the cost-performance trade-off and decide how to take action accordingly.

3.8 Related Work

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 [29].

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 tried to develop and evaluate such a model.

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 [75]. Rigorous models have been leveraged to analytically model the performance of various cloud services for IaaS, PaaS, and microservices [29, 76–81]. In [76], 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. [77] 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. [29, 78–80] 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. [81] 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. [82] 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. [83] 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. [84] 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. [85] 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. [86] 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 [5], 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. [87] 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. [88] 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. [89] 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. [10] 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. [11] 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. [90] 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. Singhvi et al. [91] introduced a scalable low-latency serverless platform named Archipelago. Their studies showed that current serverless schedulers are limited in handling very short-lived tasks, tasks with unpredictable arrival patterns, and tasks that require expensive setup of sandboxes. They found that current serverless offerings handle the incoming requests homogeneously. This is while functions have varying latency requirements; e.g., user-facing functions are latency-sensitive while

batch workloads are less sensitive to latency. A performance-aware model like the one proposed in this work could lead to SLA-aware scheduling on serverless computing platforms, reducing the cost and optimizing hardware utilization. Kaffes et al. [92] 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. [10]. Manner et al. [17] 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 [12], 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. [13] 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. [93] 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. [94] 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. [95] 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. [18] 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. [96] 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. [43] used analytical models that leverage queuing theory to optimize the performance of composite service application jobs by tuning configurations and resource allocations. Horovitz et al. [97] 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. [98] 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. [15] 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. [99] 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. [16] 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. [100] 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. [73] 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.

In [101], the authors investigated and quantified the elasticity of current serverless computing platforms for three types of applications, deciding the most suitable

one for each type of workload. Besides, they tried to identify the most influencing performance indicators for each type of application. Ao et al. [102] proposed Sprocket, which is a serverless video processing framework. The proposed system is a parallel data processing framework for video content and targets serverless cloud infrastructures like AWS Lambda as its execution environment. Perez et al. [103] introduced a programming model and a middleware for leveraging serverless computing to execute highly-parallel scientific workloads. This paves the way for efficient, affordable, and high throughput computation. In this work, the authors found maximum memory allocation and maximum execution time to be the most limiting aspects of current serverless computing offerings. They found serverless computing to be ideal for deploying a high number of short-lived tasks. Jackson et al. [104] investigated the implications imposed on the performance of serverless computing platforms by the programming language. They found that using different programming languages could lead to performance challenges in serverless computing platforms. Their results showed that Python is the best performing programming language in AWS Lambda and Azure Functions. Villamizar et al. [105] compared the cost of deploying a web application using monolithic, microservice, and serverless architectures. They found that using serverless computing for web applications could reduce the cost of infrastructure by up to 77% while keeping the performance and response time reasonable under heavy loads and increased number of users.

3.9 Conclusion

In this chapter, 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 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.

Chapter 4

Performance Modeling of Metric-Based Serverless Computing Platforms

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 Chapter 3, 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 chapter, 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 configura-

tions 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.

4.1 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 [10, 18, 27]. 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 [29].

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 [106, 107], 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 [106] 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 [108–112]. 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 this chapter is organized as follows: Section 4.2 describes the system represented by the analytical performance model proposed in this work. Section 4.3 outlines the proposed analytical model. In Section 4.4, we present the experimental validation of the proposed model. In Section 4.5, we survey the latest related work for serverless computing platforms. Section 4.6 discusses the threats to the validity of our experiments. Section 4.7 summarizes our findings and concludes the chapter.

4.2 System Description

There is very limited documentation available about the scheduling algorithm used in most serverless computing platforms that use per-request autoscaling [106]. As a result, previous studies have mostly focused on partially reverse engineering these platforms by running experiments on them [10–13, 27]. However, the most recent trend in serverless computing platforms that use metric-based autoscaling, Knative [113] and Google Cloud Run [68] for example, are primarily open-sourced and thus we can use their source code to develop accurate performance models without speculations[1].

Figure 4.1 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

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

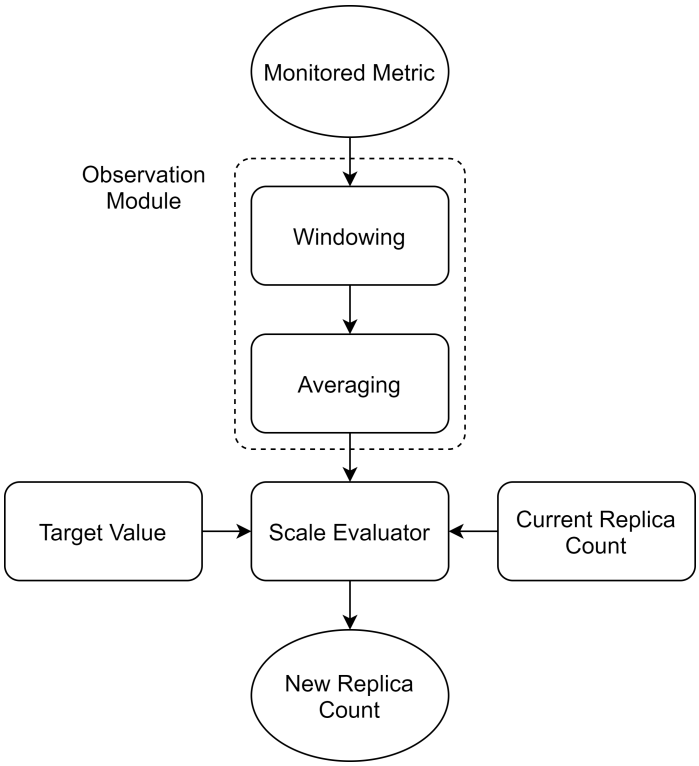


Figure 4.1: An overview of the Knative scale calculation module. The resulting new replica count will be applied to the cluster.

modelled by the proposed performance model.

4.2.1 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) [114]. 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.

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

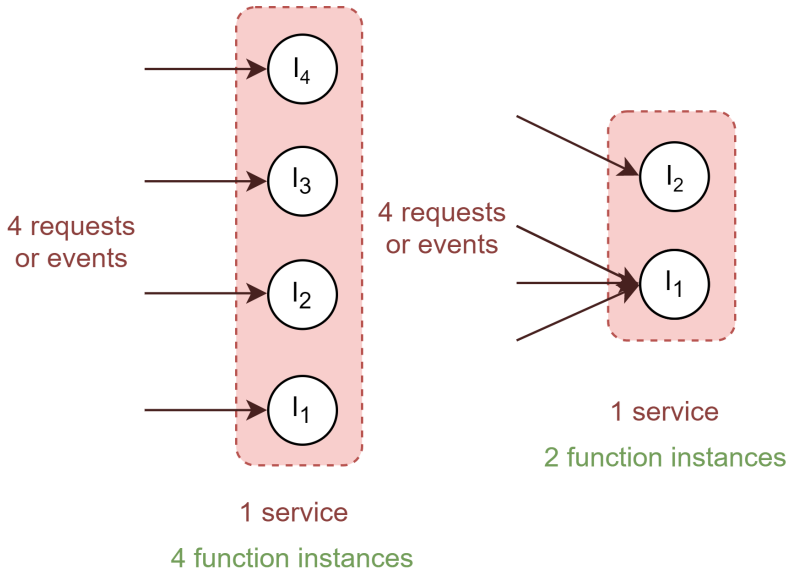


Figure 4.2: The effect of the concurrency value on the number of function instances needed. The left service allows a maximum of 1 request per instance, while the right service allows a concurrency value of 3.

requests to enter the same function instance at the same time. The number of concurrent requests being processed by the same container is called the *concurrency value* in Knative documentations. Figure 4.2 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 4.3 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.

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

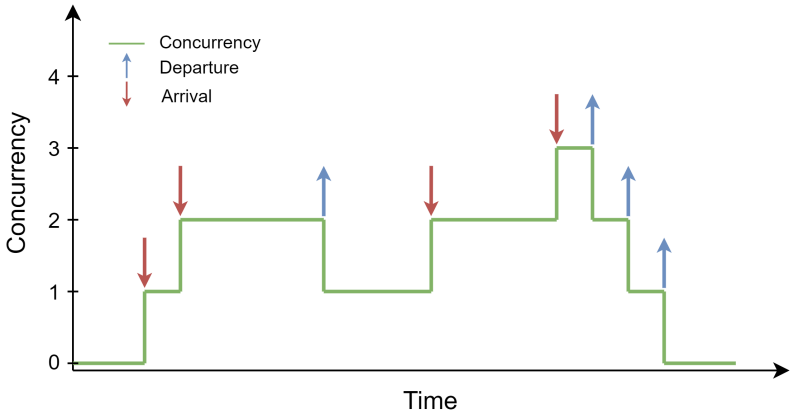


Figure 4.3: An example scenario of the change in the container concurrency value. The effect of request arrival and departure on concurrency value is shown over time.

monitors the number of requests arriving to each container every second and reports the resulting value.

4.2.2 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 *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.

4.2.3 Scale Evaluator Module

As discussed in Section 4.2.2, 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]:

$$NewOrderedReplica = \left\lceil \frac{ObservedValue}{TargetValue} \right\rceil \tag{4.1}$$

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.

4.3 Analytical Model

In Section 4.2, 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.

Figure 4.4 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

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

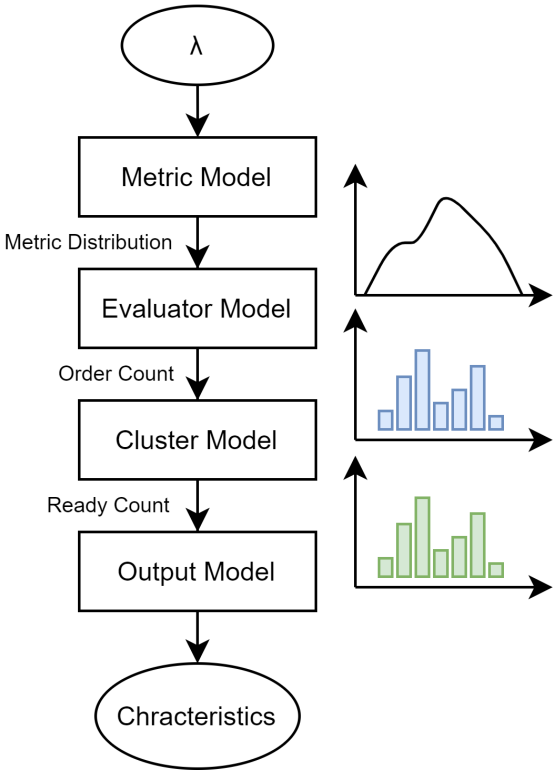


Figure 4.4: An overview of the proposed performance model.

the output model, we calculate the steady-state estimates for different characteristics of the deployment.

In the following subsections, we present the calculation of different parameters in the analytical models using the symbols defined in Table 4.1; we will elaborate on the details of the aforementioned sub-models.

4.3.1 Metric Model

As discussed in Section 4.2.1, 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.

Table 4.1: 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

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:

$$f_{OV}(x) \approx MM(x; \lambda/N) \tag{4.2}$$

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.

4.3.2 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:

$$N_{ord} = \left\lceil \frac{OV}{TV} \right\rceil \tag{4.3}$$

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}:

$$\begin{aligned} Pr\{N_{ord} = i\} &= Pr\{\left\lceil \frac{OV}{TV} \right\rceil = i\} \\ &= Pr\{(i-1) < \frac{OV}{TV} \leq i\} \\ &= Pr\{(i-1) \cdot TV < OV \leq i \cdot TV\} \\ &= F_{OV}(i \cdot TV) - F_{OV}((i-1) \cdot TV) \end{aligned} \tag{4.4}$$

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

$$F_{OV}(x) = Pr\{OV \leq x\} = \int_{-\infty}^{x} f_{OV}(x)\, dx \tag{4.5}$$

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.

$$EM(i; f_{OV}) = F_{OV}(i \cdot TV) - F_{OV}((i-1) \cdot TV) \tag{4.6}$$

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.

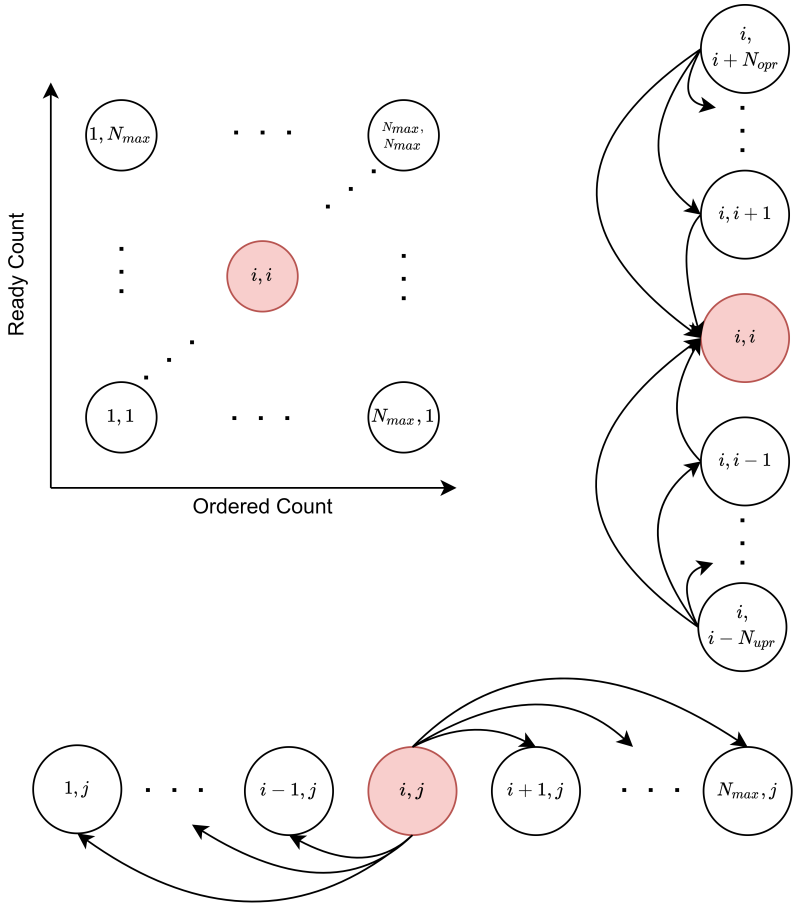


Figure 4.5: An overview of the proposed cluster model along with its vertical and horizontal components.

4.3.3 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 4.5 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 4.6) 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:

$$P_{(i,j),(i',j')} = P_{i,i'}(j) \times P_{j,j'}(i) \tag{4.7}$$

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:

$$
\begin{aligned}
f_{OV}(x) &= MM(x; \lambda/j) \\
P_{i,i'}(j) &= EM(i'; f_{OV})
\end{aligned}
\tag{4.8}
$$

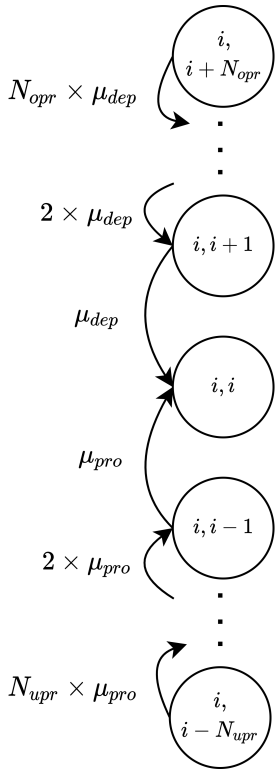


Figure 4.6: An overview of the underlying infrastructure CTMC model used in cluster model. N_{opr} and N_{upr} signify the number of overprovisioned and underprovisioned containers and μ_{pro} and μ_{dep} represent the provisioning and deprovisioning service rates, respectively.

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 4.6 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:

$$\frac{d\pi'}{dt} = \pi' Q \Rightarrow \pi'(t) = \pi'(0) e^{Qt} \tag{4.9}$$

which can be calculated using the method proposed by Al-Mohy et al. [74].

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

$$P_{j,j'}(i) = \pi'_{j'}(T_{eva}) \tag{4.10}$$

Using Equations (4.8) and (4.10), we can build the transition probability matrix P for the cluster model shown in Figure 4.5. To analyze the steady-state behaviour of the system, we need to calculated the limiting probability π_s for any state s where [21]:

$$\pi_s = \lim_{n \to \infty} P^n_{s,s'} \tag{4.11}$$

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:

$$\pi = (\pi_1, ..., \pi_M), \sum_{x=1}^{M} \pi_x = 1 \tag{4.12}$$

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 [115].

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.

4.3.4 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.

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:

$$\overline{RT}_N = RTF(\lambda/N) \tag{4.13}$$

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:

$$\begin{aligned}\overline{RT} &= \sum_{i=1}^{M} \pi_i \overline{RT}_{N_i} \\ &= \sum_{i=1}^{M} \pi_i RTF(\lambda/N_i)\end{aligned} \tag{4.14}$$

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.

Average Replica Count

There are mainly two 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 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:

$$\overline{N} = \sum_{i=1}^{M} \pi_i N_i \tag{4.15}$$

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

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:

$$\overline{C} = \sum_{i=1}^{M} \pi_i C_i \tag{4.16}$$

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:

$$C_i = \int_0^{\infty} x \cdot MM(x; \lambda/N_i) dx \tag{4.17}$$

4.4 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 4.2: Configuration of the VMs in the experiments.

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

[3]https://github.com/pacslab/conc-value-perf-modelling

4.4.1 Experimental Setup

To perform our experiments, we used 4 Virtual Machines (VMs) on the Cybera Cloud [38] with the configuration shown in Table 4.2. 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.

4.4.2 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 re-

[4]https://github.com/pacslab/pacswg
[5]https://pypi.org/project/pacswg

sults. We present a representative subset of these results here due to space limitation. For *workload 1*, we used the work of Wang et al. [10] 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:

$$y = \alpha_1 \cdot x + \alpha_2 \cdot x^2 \tag{4.18}$$

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:

$$y = \alpha_0 + \alpha_1 \cdot x + \alpha_2 \cdot x^2 \tag{4.19}$$

[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.

4.4.3 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 4.7 and 4.8 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 4.9 and 4.10 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 4.11 and 4.12 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.

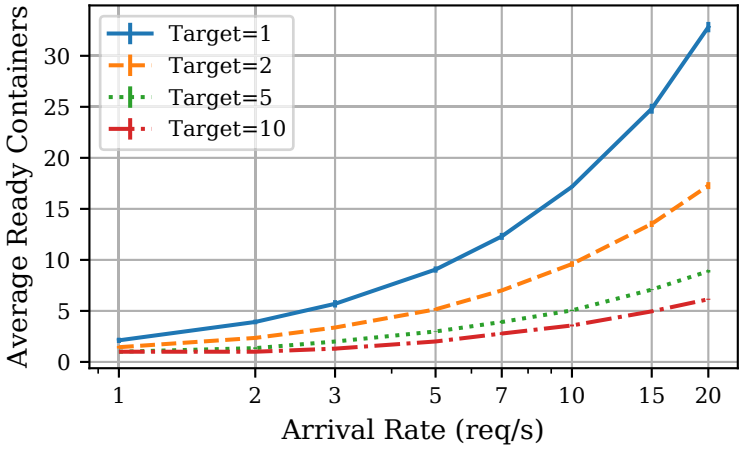


Figure 4.7: The measured average number of containers ready to server requests versus the fixed arrival rate for different target concurrency values in our experiments. Note that the x-axis is on a logarithmic scale. The vertical bar shows the 95% confidence intervals which in this case were very small because experiments were long enough to have very accurate results.

4.4.4 Discussion

In Section 4.4.3, 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 4.2. 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 4.7 to 4.12, 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

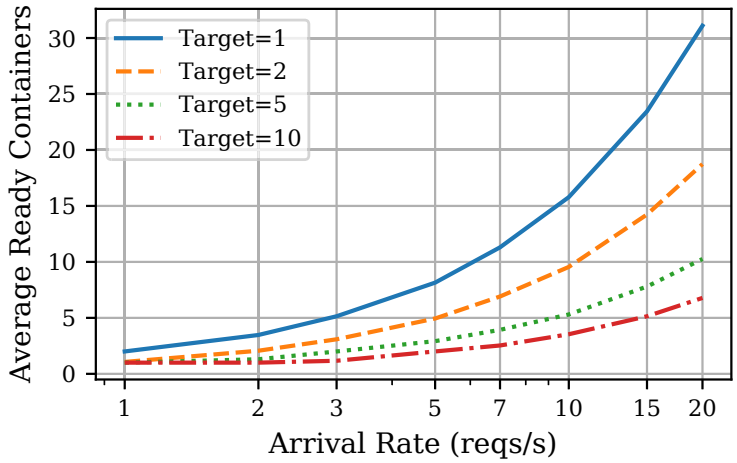


Figure 4.8: The predicted average number of containers ready to server requests versus the fixed arrival rate for different target concurrency values. Note that the x-axis is on a logarithmic scale.

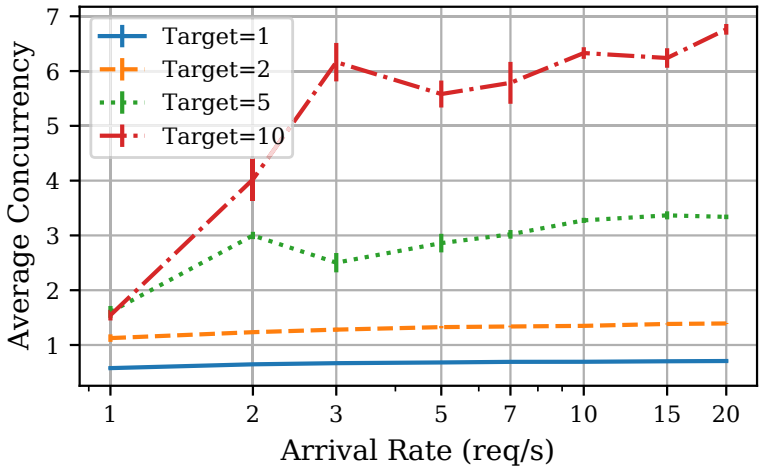


Figure 4.9: The measured average concurrency value versus the fixed arrival rate for different target concurrency values in our experiments. Note that the x-axis is on a logarithmic scale. The vertical bar shows the 95% confidence intervals.

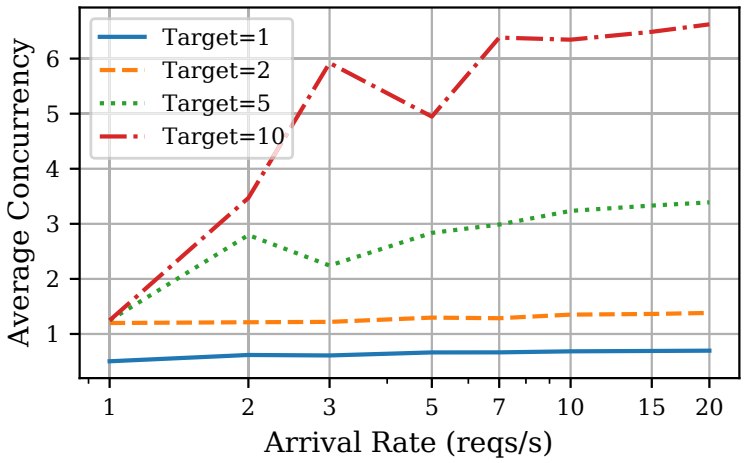


Figure 4.10: The predicted average concurrency versus the fixed arrival rate for different target concurrency values. Note that the x-axis is on a logarithmic scale.

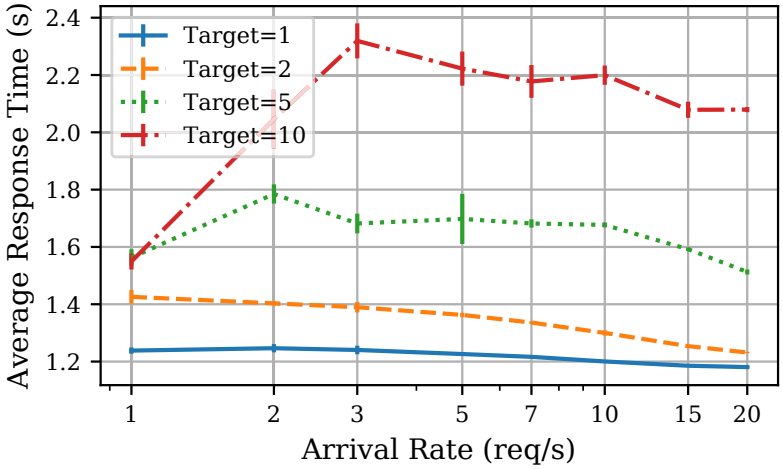


Figure 4.11: The measured average response time versus the fixed arrival rate for different target concurrency values in our experiments. Note that the x-axis is on a logarithmic scale. The vertical bar shows the 95% confidence intervals.

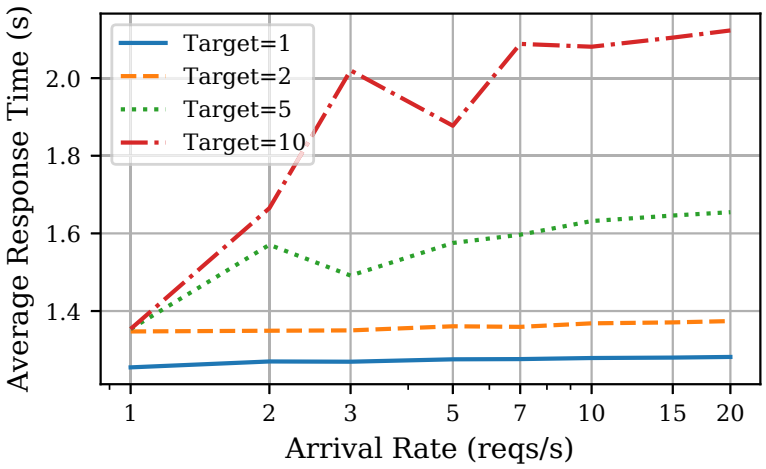


Figure 4.12: The predicted average response time versus the fixed arrival rate for different target concurrency values. Note that the x-axis is on a logarithmic scale.

configurations by predicting the effect of a new configuration on the performance and the cost of the system. Figure 4.13 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 4.14 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.

4.5 Related Work

Serverless Computing has attracted a lot of attention from the research community. However, a limited number of studies have focused on performance models capturing

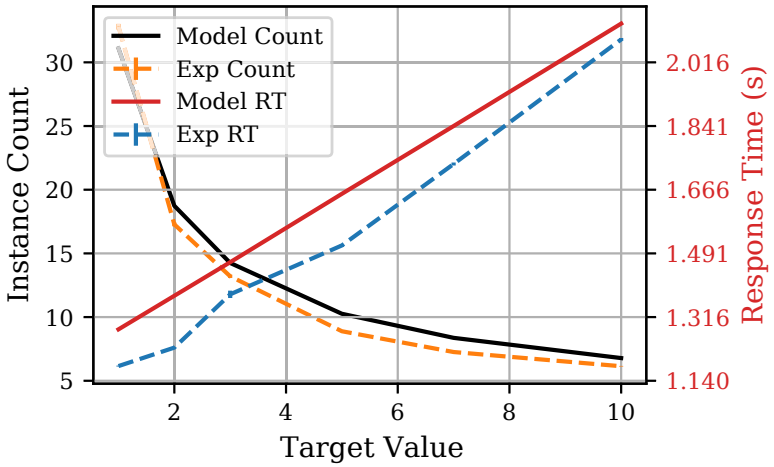


Figure 4.13: The effect of changing the target value on the average instance count and average response time measured in experiment and predicted by the proposed model for an arrival rate of 20 requests per second for *workload 1*.

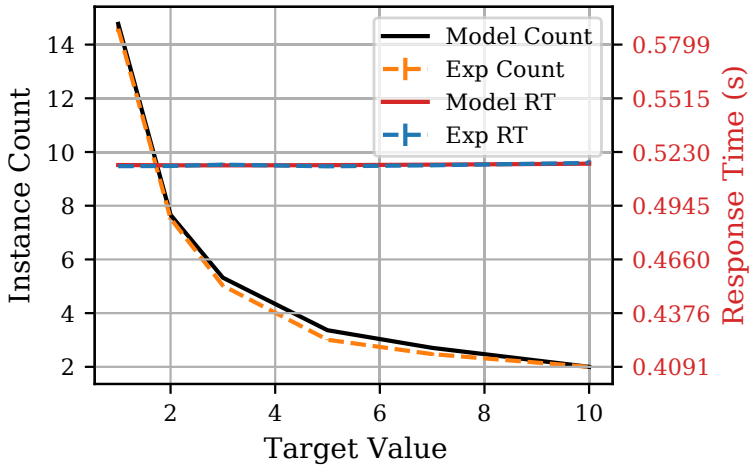


Figure 4.14: The effect of changing the target value on the average instance count and average response time measured in experiment and predicted by the proposed performance model for an arrival rate of 20 requests per second for *workload 2*.

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 [106, 107, 116]. 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 [75]. Rigorous models have been leveraged to analytically model the performance of various cloud services for IaaS, PaaS, and microservices [29, 76–83, 117]. In [76], 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. [77] 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. [29, 78–80, 117] have proposed monolithic and interactive submodels for IaaS cloud datacenters with enough accuracy and tractability for large-scale cloud datacenters. Qian et al. [81] 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. [86] 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 [5], 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. [89] 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. [118] 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. [90] 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. [92] 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. [10]. In [12], 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. [13] investigated the factors influencing the performance of serverless computing platforms. Bardsley et al. [93] 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. [94] 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. [95] 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. [18] 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. [119] 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. [43] 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. [98] 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. [73] 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. [120] 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. [121] introduced Libra, an adaptive hybrid vertical/horizontal autoscaler on OpenFaaS trying to outperform both openfaas autoscaler and Kubernetes HPA.

4.6 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 [122]. 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.

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 (especially on shared CPU configurations). To mitigate this threat, we used recommended practices to obtain and report our experimental results [122].

4.7 Conclusion

In this chapter, 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.

Chapter 5

SimFaaS: A Simulator for Serverless Computing Platforms

Developing accurate and extendable performance models for serverless platforms, also known as Function-as-a-Service (FaaS) platforms, is a very challenging task. Also, implementation and experimentation on real serverless platforms is both costly and time-consuming. However, at the moment, there is no comprehensive simulation tool or framework to be used instead of the real platform. As a result, in this paper, we fill this gap by proposing a simulation platform, called SimFaaS, which assists serverless application developers to develop optimized Function-as-a-Service applications in terms of cost and performance. On the other hand, SimFaaS can be leveraged by FaaS providers to tailor their platforms to be workload-aware so that they can increase profit and quality of service at the same time. Also, serverless platform providers can evaluate new designs, implementations, and deployments on SimFaaS in a timely and cost-efficient manner.

SimFaaS is open-source, well-documented, and publicly available, making it easily usable and extendable to incorporate more use case scenarios in the future. Besides, it provides performance engineers with a set of tools that can calculate several characteristics of serverless platform internal states, which is otherwise hard (mostly impossible) to extract from real platforms. In previous studies, temporal and steady-state performance models for serverless computing platforms have been developed.

However, those models are limited to Markovian processes. We designed SimFaaS as a tool that can help overcome such limitations for performance and cost prediction in serverless computing.

We show how SimFaaS facilitates the prediction of essential performance metrics such as average response time, probability of cold start, and the average number of instances reflecting the infrastructure cost incurred by the serverless computing provider. We evaluate the accuracy and applicability of SimFaaS by comparing the prediction results with real-world traces from Amazon AWS Lambda.

The simulator presented in this work is written in *Python*. The resulting package can easily be installed using pip[1]. The source code is openly accessible on the project Github[2]. The documentation is accessible on Read the Docs[3]. For more information, interested readers can check out our Github repository, which provides links to all of our artifacts as well as easy-to-setup environments, to try out our sample scenarios.

A detailed description of the system simulated in SimFaaS can be found in Section 1.2. The remainder of this chapter is organized as follows: Section 5.1 outlines the design of SimFaaS with the most important design choices and characteristics. Section 5.2 lists some of possible use cases for SimFaaS. In Section 5.3, we present the experimental evaluation of SimFaaS, validating the accuracy of the simulator. Section 5.5 gives a summary of the related work. Finally, Section 5.6 concludes the chapter.

5.1 The Design of SimFaaS

This section discusses the design of the novel Function-as-a-Service (FaaS) platform simulator (SimFaaS) proposed in this work. SimFaaS was created by the authors as a tool for simplifying the process of validating a developed performance model and allowing accurate performance prediction for providers and application developers in

[1]https://pypi.org/project/simfaas/
[2]https://github.com/pacslab/simfaas
[3]https://simfaas.readthedocs.io/en/latest/

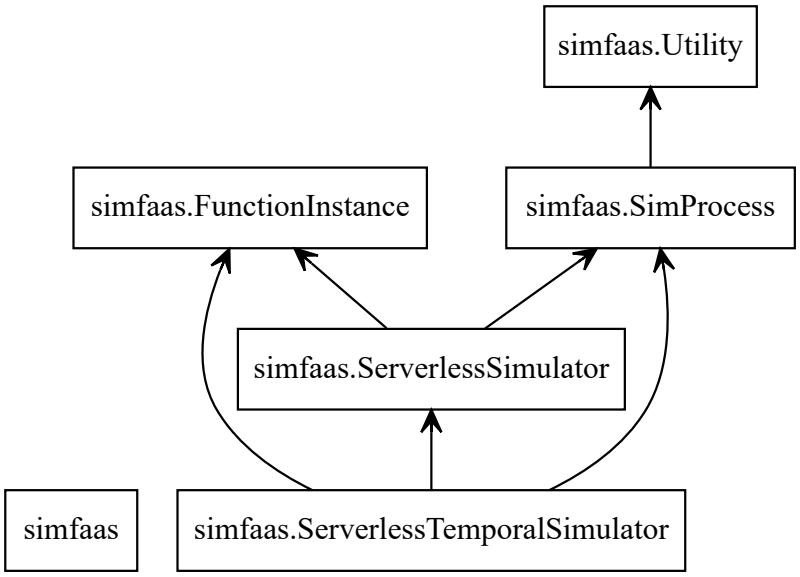


Figure 5.1: The package diagram of SimFaaS.

the absence of one. SimFaaS mainly targets public serverless computing platforms. There are several built-in tools for visualizing, analyzing, and verifying a developed analytical performance model. In addition, we added tools that can accept custom state encoding and generate approximations for Probability Density Functions (PDF) and Cumulative Distribution Functions (CDF) from the simulations, which can help debug several parts of a given analytical performance model.

The proposed simulator can predict several QoS-related metrics accurately like cold start probability, average response time, and the probability of rejection for requests under different load intensities, which helps application developers understand the limits of their system and measure their Service-Level Agreement (SLA) compliance without the need for expensive experiments. In addition, it can predict the average number of running server count and total server count, which helps predict the cost of service for the application developer and the infrastructure cost incurred by the serverless provider, respectively.

Figure 5.1 outlines the package diagram of SimFaaS, showing the dependency between different modules. The *Utility* module provides helper functions for plots and calculations. The *SimProcess* module will help simulate a single process and allows

for comparisons with the optional analytical model provided as a function handle to the module. The *FunctionInstance* module provides the functionality of a single function instance. The main simulation with an empty initial state is provided by the *ServerlessSimulator* module. Finally, *ServerlessTemporalSimulator* module performs simulations similar to *ServerlessSimulator* module, but with added functionality allowing customized initial state and calculation of simulation results in a time-bounded fashion.

5.1.1 Extensibility and Ease of Use

SimFaaS has been developed entirely in Python using an object-oriented design methodology. In order to leverage the tools within the package, the user needs to write a Python application or a Jupyter notebook initializing the classes and providing the input parameters. In addition, the user has the option to extend the functionality in the package by extending the classes and adding their custom functionality. Almost every functionality of classes can be overridden to allow for modification and extensions. For example, the arrival, cold start service, and warm start service processes can be redefined by simply extending the *SimProcess* class. We included deterministic, Gaussian, and Exponential processes as examples of such extensions in the package. Examples of such changes can be found in the several examples we have provided for SimFaaS. In addition, the user can include their analytically produced PDF and CDF functions to be compared against the simulation trace results.

The simulator provides all of the functionality needed for modelling modern scale-per-request serverless computing platforms. However, we created a modular framework that can span future types of computational platforms. To demonstrate this, we extended the *ServerlessSimulator* class to create *ParServerlessSimulator*, which simulates serverless platforms that allow queuing in the function instances but have a scaling algorithm similar to scale-per-request platforms.

5.1.2 Support for Popular Serverless Platforms

SimFaaS includes simulation models able to mimic the most popular public serverless computing platforms like AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Apache OpenWhisk, Azure Functions, and all other platforms with similar autoscaling. We have also performed over one month of experimentation to demonstrate the validity of the simulation results extracted from SimFaaS.

To capture the exogenous parameters needed for an accurate simulation, the following information is needed:

- **Expiration Threshold** which is usually constant for any given public serverless computing platform. According to our experimentations and other works [27, 62], in 2020, this value is 10 minutes for AWS Lambda, Google Cloud Functions, IBM Cloud, and Apache OpenWhisk, and 20 minutes for Azure Functions. For other serverless computing platforms, experimentation is needed by the users. The use of a non-deterministic expiration threshold is also possible by extending the *FunctionInstance* class.

- **The arrival process** which can rather safely be assumed as an exponential for most consumer-facing applications. However, other applications might use a deterministic process, e.g. cron jobs, or other types like batch arrival. The user can use one of our built-in processes or simply define their own.

- **The warm/cold service process** can be extracted by measuring the response time from monitoring the workload response time for cold and warm requests. By default, SimFaaS uses exponential distribution for this process but can be overridden by the user by passing any class that extends the *SimProcess* class.

5.2 Sample Scenarios for Using SimFaaS

In this section, we will go through a few sample use cases for the serverless platform simulator presented in this work. For more details, a comprehensive list of examples can be found in the project Github repository[4].

5.2.1 Steady-State Analysis

In this example, we use the SimFaaS simulator to calculate the steady-state properties of a given workload in scale-per-request serverless computing platforms. In SimFaaS, the workload is only characterized by arrival rate, service time (warm start response time), and the provisioning time (the amount of time to have a cold start instance get ready to serve the request), which are easily accessible through experimentation and any monitoring dashboard. The only information needed to characterize the serverless computing platform is the expiration threshold, which is the amount of time it takes for the platform to expire and recycle the resources of an instance after it has finished processing its last request. This value is usually constant and the same for all users of the serverless computing platform. To run a simple simulation, we can leverage the *ServerlessSimulator* class and run the simulation long enough to minimize the transient effect and let the system achieve the steady-state.

Table 5.1 shows a set of example simulation parameters with the default exponential distribution both for the arrival and service time processes. Note that instead of using exponential distributions, the user can pass a random generator function with a custom distribution to achieve more accurate results for specific applications. As can be seen, the system can produce QoS-related parameters like the probability of cold start or rejection of the request for a given arrival rate, which can help the application developer analyze and find the limits of the system. In addition, the application developer can also use the average number of running servers as an important measure for the cost of their service that can be used for setting different configurations of

[4]https://github.com/pacslab/SimFaaS/tree/master/examples

Table 5.1: An example simulation input and selected output parameters. The output parameters are signified with a leading star.

Parameter	**Value**
Arrival Rate	0.9 req/s
Warm Service Time	1.991 s
Cold Service Time	2.244 s
Expiration Threshold	10 min
Simulation Time	10^6 s
Skip Initial Time	100 s
*Cold Start Probability	0.14 %
*Rejection Probability	0 %
*Average Instance Lifespan	6307.7389 s
*Average Server Count	7.6795
*Average Running Servers	1.7902
*Average Idle Count	5.8893

services that the function relies on, e.g., the database concurrent connection capacity [123]. Besides, information like the average server count can produce an estimate for the infrastructure cost incurred by the serverless provider. The serverless provider can use SimFaaS as a tool to analyze the possible effect of changing parameters like the expiration threshold on their incurred cost and QoS for different scenarios.

Another way the proposed simulator can be leveraged is for extracting information about the system that is not visible to software engineers and developers in public serverless computing platforms like AWS Lambda or Google Cloud Functions. This information could facilitate research for predicting cost, performance, database configurations, or other related parameters. For example, we can find out the distribution of instance counts in the system throughout time in the simulated platform for input parameters shown in Table 5.1, which is shown in Figure 5.2. This information can help researchers develop performance models based on internal states of the system with very good accuracy, which is otherwise not possible in public serverless offer-

ings. To further analyze the reproducibility of our instance count estimation using the parameters in Table 5.1, we ran 10 independent simulations and generated our estimation of average instance count over time for each run. Figure 5.3 shows the average and 95% confidence interval of our estimation over those runs. As can be seen, our estimation converges, showing less than 1% deviation from the mean in the 95% confidence interval.

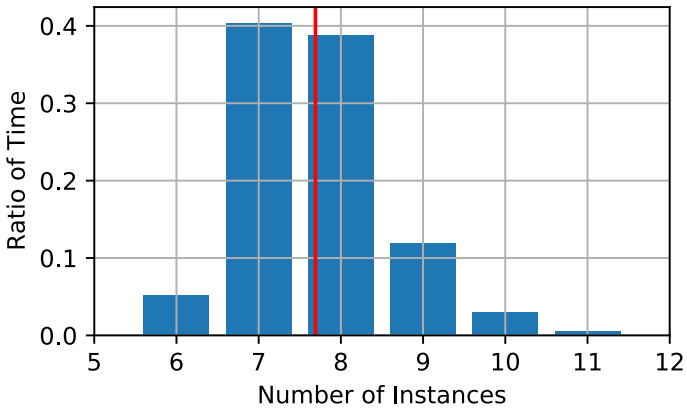


Figure 5.2: The instance count distribution of the simulated process throughout time. The y-axis represents the portion of time in the simulation with a specific number of instances.

5.2.2 Transient Analysis

Although the steady-state analysis of the serverless computing platform's performance can give us the long-term quality of service metrics, the application developer or the serverless provider might be interested in the platform's transient behaviour. A transient analysis simulation can provide insight into the immediate future, facilitating time-bound performance guarantees. Besides, it can help serverless providers ensure the short-term quality of service when trying new designs.

Previous efforts have been made to develop performance models able to provide transient analysis of serverless computing platforms [107]. However, there are inherent limitations to such performance models, like the absence of batch arrival modelling and being limited to Markovian processes. SimFaaS doesn't have such limitations

and can help both application developers and serverless providers gain insight into the transient aspect of the performance of serverless computing platforms.

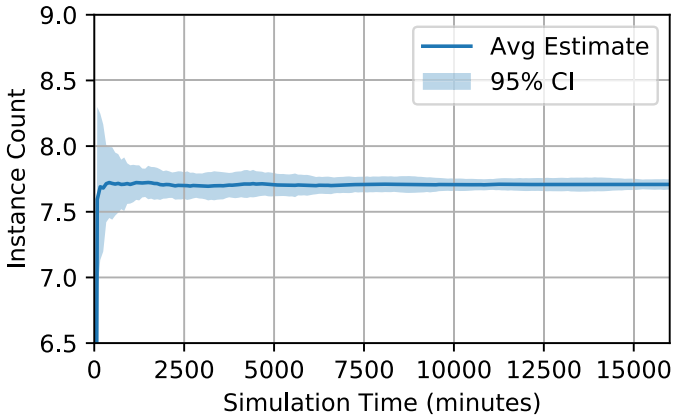


Figure 5.3: The estimated average instance count over time in 10 simulations. The solid line shows the average of simulations and the shaded area shows the 95% Confidence Interval (CI).

5.2.3 What-If Analysis

Due to the inherent highly-dynamic infrastructure of serverless computing platforms, there are very few tools from the performance engineering methodologies and analysis that can be used in the emerging serverless technologies. Because of this inherent lack of tools and resources, serverless computing platforms were forced to use trial and error through implementation to analyze their new designs for making performance and efficiency improvements. There have been previous studies that proposed analytical performance models for serverless computing platforms [106, 107], but these methods have limitations like only supporting Markovian processes, limiting their applicability in a number of scenarios.

One major benefit of having an accurate serverless platform simulator is the ability to perform what-if analysis on different configurations and find the best-performing settings for a given workload. Implementation and experimentation to gather similar data is both time-consuming and costly, while using the proposed simulator makes the data collection much faster and easier. Figure 5.4 shows an example of such an analy-

sis for different values for the *expiration threshold* in the system. Different workloads running on serverless computing platforms might have different performance/cost criteria. Using what-if analysis powered by an accurate simulation platform, one could optimize the configuration for each unique workload. Similar examples can be found in the project examples.

5.2.4 Cost Calculation

Performing cost prediction under different loads in cloud computing is generally a very challenging task. These challenges tend to be exacerbated in the highly dynamic structure of serverless computing platforms. Generally, there is a broad range of possible costs for a given serverless function, including computation, storage, networking, database, or other API-based services like machine learning engines or statistical analysis. However, all charges incurred by serverless functions can be seen as either per-request charges (e.g., external APIs, machine learning, face recognition, network I/O) or runtime charges billed based on execution time (e.g., memory or computation). Per-request charges can be calculated using only the average arrival rate. However, runtime charges may differ under different load intensities due to the difference in cold start probability. Using the proposed simulator, application developers can get an estimate on the cold start probability and the average number of running servers, which are necessary for cost estimation under different load intensities.

In addition to the average running server count, which helps estimate the cost incurred by the application developer, the average total server count is linearly proportional to the infrastructure cost incurred by the serverless provider. Thus, using the proposed simulator, both developer charges and infrastructure charges incurred by the provider can be estimated under different configurations, which can help improve the platform by studying the effect of using different configurations or designs without the need to perform expensive or time-consuming experiments or implementations.

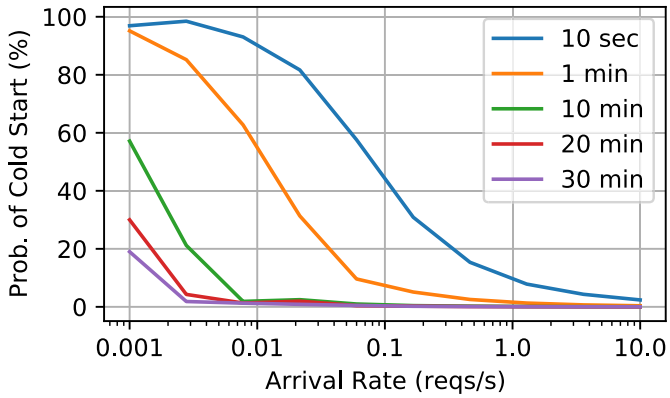


Figure 5.4: Cold start probability against the arrival rate for different values of the expiration threshold for the workload specified in Table 5.1. SimFaaS can ease the process of conducting experiments with several configurations to find the best performing one.

5.3 Experimental Validation

To show the accuracy of the proposed simulator, we performed extensive experimentations on AWS Lambda and showed that the results were in tune with the results from SimFaaS. The experiments are above one month of running benchmark applications on AWS Lambda and are openly accessible on Github[5]. All of our experiments were executed for a 28-hour window with 10 minutes of warm-up time in the beginning, during which we do not record any data. The workload used in this work was based on the work of Wang et al. [10] with minor modifications. Our workload is openly available in our Github repository[6]. For the purpose of experimental validation, we used a combination of CPU intensive and I/O intensive workloads. During the experimentation, we have obtained performance metrics and the other parameters such as cold/warm start information, instance id, lifespan, etc.

5.3.1 Experimental Setup

In our AWS Lambda deployment, we used the *Python 3.6* runtime with 128 MB of RAM deployed on the *us-east-1* region in order to have the lowest possible latency

[5]https://github.com/pacslab/serverless-performance-modeling/tree/master/experiments/results
[6]https://github.com/pacslab/serverless-performance-modeling

from our client. Note that the memory configuration won't affect the accuracy of the simulation as the results depend on the service time distribution, which captures the effect of changing the memory configuration. For the client, we used a virtual machine with 8 vCPUs, 16 GB of memory, and 1000 Mbps network connectivity with single-digit milliseconds latency to AWS servers hosted on Compute Canada Arbutus cloud[7]. We used Python as the programming language and the official *boto3* library to communicate with the AWS Lambda API to make the requests and process the resulting logs for each request. For load-testing and generation of the client requests based on a Poisson process, we used our in-house workload generation library[8], which is openly accessible through PyPi[9]. 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 our analytical model project's Github repository[10].

5.3.2 Parameter Identification

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

Expiration Threshold: here, our goal is to measure the expiration threshold, which is the amount of time after which inactive function instance 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 instance 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

[7]https://docs.computecanada.ca/wiki/Cloud_resources
[8]https://github.com/pacslab/pacswg
[9]https://pypi.org/project/pacswg
[10]https://github.com/pacslab/serverless-performance-modeling

we see a cold start. In our experiments, AWS lambda instances seemed to expire an instance 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 [27, 62].

Average Warm Response Time and Average Cold Response Time: with the definitions provided here, warm response time is the service time of the function, and cold response time includes both provisioning time and service time. To measure the average warm response time and the average cold response time, we used the average of response times measured throughout the experiment.

5.3.3 Simulator Results Validation

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

Probability of Cold Start: 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 led to increasing the window for data collection to about 28 hours for each sampled point.

Mean Number of Instances in the Warm Pool: 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 unique identifier for each function instance to keep track of their life cycle, as obtained in [10].

Mean Number of Running Instances: 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: 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.

Average Wasted Capacity: for this metric, we define the utilized capacity as the ratio of the number of running instances over all instances in the warm pool. Using this definition, the ratio of idle instances over all instances in the warm pool is the wasted portion of capacity provisioned for our workload. Note that this value is very important to the provider as it measures the ratio of the utilized capacity (billed for the application developer) over the deployed capacity (reflecting the infrastructure cost).

5.3.4 Results and Discussion

Figure 5.5 shows the probability of cold start for different arrival rates extracted from the simulation compared with real-world results. As can be seen, the results match the performance metrics extracted from experimentations. The results show an average error of 12.75%, while the standard error of the underlying process for the experiments over 10 runs totalling 28 hours is 10.14% showing the accuracy of the results obtained from the simulation. Figures 5.6 and 5.7 show the average number of instances and the average wasted capacity (in *idle* state) for the simulation and experiments with Mean Absolute Percentage Error (MAPE) of 3.43% and 0.17%, respectively.

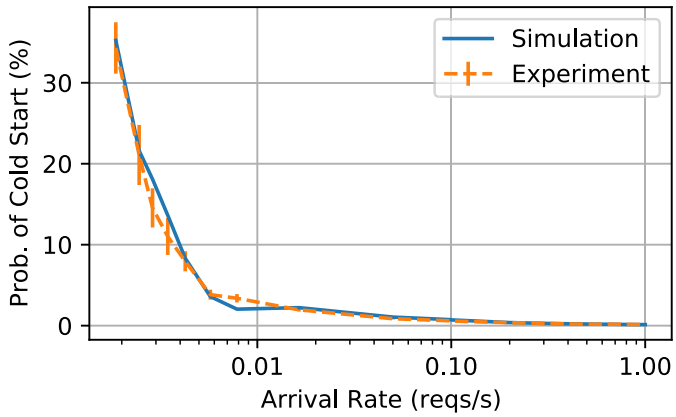


Figure 5.5: Probability of cold start extracted from simulation compared with real-world experimentations on AWS Lambda.

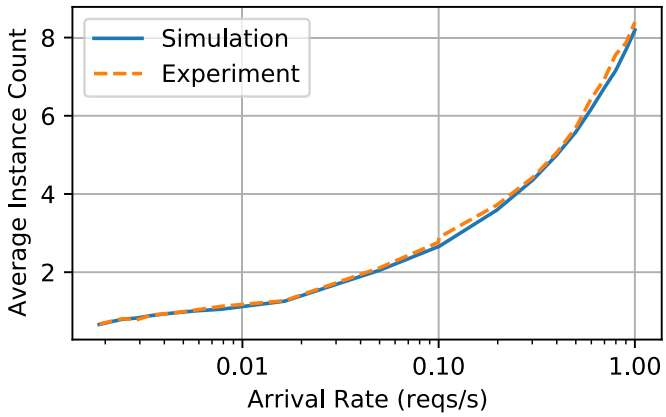


Figure 5.6: The average number of instances extracted from simulation compared with real-world experimentations on AWS Lambda.

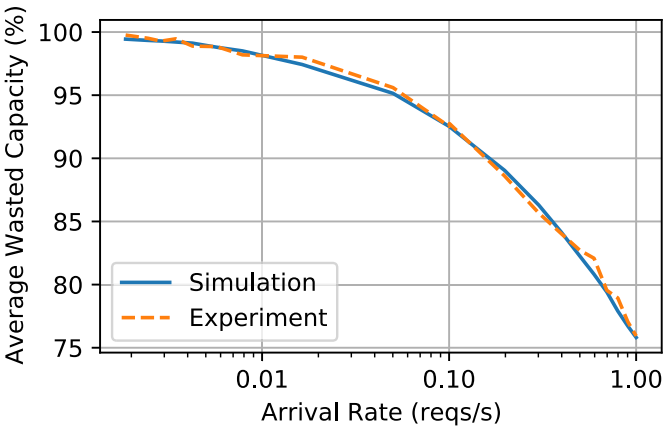


Figure 5.7: Average wasted resources extracted from simulation compared with real-world experimentations on AWS Lambda.

5.4 Limitations and Threats to Validity

In this section, we will discuss some of the threats to validity of our study on SimFaaS and some inherent limitations of this research.

In our experiments, we used average response time as an indicator for the Quality of Service (QoS) and the average instance count as an indicator of costs. These choices might have an impact on the applicability of the study for a specific use case if the metrics don't align with the user's interest. We also used exponential arrival

rate distribution which is the most common arrival rate distribution for consumer-facing online services. However, we acknowledge that other arrival distribution like deterministic cron jobs are also common with serverless computing and the accuracy of the simulator may vary for these configurations. The analysis of other types of arrival rate distributions has been left for future studies.

We used a limited number of benchmarks in our experimentations, however, the accuracy of the simulator may vary for other types of workloads. This effect would be worsened for workloads that work with external APIs and databases, as their performance will depend on these systems. We leave the analysis of a more diverse set of workloads for a future study.

5.5 Related Work

Manner et al. [124] describe the importance of an accurate simulator for Function-as-a-Service (FaaS) products. They mention how scaling, cold starts, function configurations, dependent services, network latency, and other important configurations influence cost-performance trade-off. In their work, they propose a simulation framework for a cost and performance simulator for serverless computing platforms. In this platform, they suggest extracted mean values from experiments as inputs to the performance model in order to calculate different properties.

Boza et al. [125] introduced a model-based simulation for cloud budget planning. In their work, they perform cost simulation for using reserved VMs, on-demand VMs, bid-based VMs, and serverless computing for a similar computing task. In their work, the serverless computing simulation is overly simplistic for performance modelling researchers and lacks several important details. In this work, we focus solely on simulating the performance of serverless computing platforms, but with more details in mind, which seems necessary for the simulator to be leveraged by the performance research community.

Abad et al. [28] mainly considered the problem of scheduling small cloud functions

on serverless computing platforms. As a part of their evaluations, they implemented a SimPy-based simulation for their proposed scheduling method. Although this work shows promise of rather accurate serverless computing simulations, their focus is on scheduling tasks, while ignoring several details of interest for performance modelling. In this work, we strive to fill this gap by providing the performance modelling research with the proper tooling necessary for high-fidelity performance models of serverless computing platforms.

Jeon et al. [126] introduced a CloudSim extension focused on Distributed Function-as-a-Service (DFaaS) on edge devices. Although the DFaaS systems hold a great promise for the future of serverless computing, it doesn't allow simulation for the mainstream serverless computing platforms.

5.6 Conclusion

In this chapter, we presented SimFaaS, a simulator for modern serverless computing platforms with sufficient details to yield very accurate results. We introduced a range of tools available for performance modelling researchers giving them insights and details into several internal properties that are not visible for users in public serverless computing platforms. We reviewed some of the possible use cases of the proposed simulator and showed its accuracy through comparison with real-world traces gathered from running benchmark applications on AWS Lambda.

SimFaaS enables performance modelling researchers with a tool allowing them to develop accurate performance models using the internal state of the system, which cannot be monitored on public serverless computing platforms. Using SimFaaS, both serverless providers and application developers can predict the quality of service, expected infrastructure and incurred cost, amount of wasted resources, and energy consumption without performing lengthy and expensive experimentations. The benefits of using SimFaaS for serverless computing platform providers could be two-fold: 1) They can examine new designs, developments, and deployments in their platforms

by initially validating new ideas on SimFaaS, which will be significantly cheaper in terms of time and cost compared to actual prototyping; 2) They can provide users with fine-grain control over the cost-performance trade-off by modifying the platform parameters (e.g., *expiration threshold*). 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 application developers' experience and consequently the end-users.

Chapter 6

MLProxy: SLA-Aware Reverse Proxy for Machine Learning Inference Serving on Serverless Computing Platforms

Serving machine learning inference workload on the cloud is still one of the key challenges in production machine learning. Optimal configuration of the inference workload to meet Service-Level Agreement (SLA) requirements while optimizing the infrastructure costs is still complicated by the complex interaction between batch configuration, resource configurations, and variations in the arrival process. Serverless computing has emerged in recent years to automate several infrastructure management tasks. Batching has proven to be necessary to improve latency performance and cost-effectiveness of machine learning serving workloads but is yet to be supported out of the box in serverless computing platforms. Our experiments have shown that for many machine learning workloads, batching can hugely improve the efficiency of the system by reducing the processing overhead per request.

In this chapter, we present MLProxy, an adaptive reverse proxy to support efficient machine learning serving workloads on serverless computing platforms. MLProxy supports adaptive batching to ensure SLA compliance while optimizing serverless costs. We performed rigorous experiments on Knative to demonstrate the effectiveness of MLProxy and showed that MLProxy could reduce the cost of serverless deployment

by up to 92% while reducing SLO violations by up to 99% and generalizing across state-of-the-art model serving frameworks.

6.1 Introduction

There are typically three phases when using machine learning models in software applications: 1) Model Design, 2) Model Training, and 3) Model Inference (or Model Serving) [127]. Previous research have found model serving as one of the most challenging stages in the evolution of the use of machine learning components in commercial software systems [128]. Some studies have focused on using different paradigms in cloud computing to improve model serving in terms of latency, throughput, and cost [127, 129–132]. However, these solutions introduce a higher infrastructure management overhead compared to managed services or serverless computing platforms.

Serverless computing proves to be a promising option for the deployment of machine learning serving workloads. Using serverless computing platforms, the developer can only write the code and leave all infrastructure management tasks to the cloud provider. This paradigm can help the developer achieve several non-functional goals like low latency and rapid autoscaling while still being cost-effective. Most serverless providers can now deliver on performance isolation at any scale, which is very important for applications with unpredictable service request patterns. Serverless computing can also reduce the cost of deployment because in this paradigm, the developer is only billed for the actual usage of the resources instead of the provisioned resources. Ease of management is also another trait of using serverless computing platforms.

For this study, we have opted to use Knative[113] on top of Kubernetes[6] as the underlying serverless platform used to perform autoscaling and computations necessary to serve the model inference workload. There are several benefits to using Knative and Kubernetes for serving machine learning inference workloads [133]. One of the major benefits of using Kubernetes and Knative for inference serving is the

ability to use autoscaling provided by serverless computing on hardware accelerators like GPU and TPU. Using Knative also has the benefit of giving the developers the opportunity to either use a managed service (e.g., Google Cloud Run) or a self-hosted version (Knative deployed on the developer's Kubernetes cluster).

Batching has been previously used in other computing paradigms to improve device utilization and cost [129–131, 133]. Batching several requests into a single request can increase the input dimension of the inference, providing great opportunities for parallelization, especially on accelerated hardware. However, there are several challenges that need to be overcome when serving machine learning workloads on serverless computing platforms. Current serverless computing platforms don't support batching natively. As a result, custom middleware is needed to allow batching support. Current serverless computing platforms are oblivious to SLA requirements, while strict SLA requirements are necessary to ensure a good user experience. To ensure the SLA objectives are met during different arrival rates, adaptive parameter tuning is necessary [127].

Many recent studies have focused on optimizing machine learning serving on AWS Lambda by using profiling and prior access to the deployed models or the arrival process [127, 132]. In this work, we strive to build an adaptive system called MLProxy that can function on both managed and self-hosted serverless platforms without prior profiling steps using lightweight adaptive batching. This is necessary for serverless computing platforms with pay-per-use pricing and allows the developed middleware to act as a drop-in replacement for API gateways providing instantaneous improvements over previous methods.

The proposed optimizer 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. Our experiments have shown the effectiveness of MLProxy on several frameworks, including Tensorflow and Tensorflow Serving, BentoML, PyTorch, and Keras.

Serverless computing is predicted to be the future of cloud computing workloads [2]. However, the current implementation of serverless computing platforms has not reached its potential and performs poorly with machine learning inference workloads. In this work, we are trying to introduce an easy to implement, deploy, and adopt API gateway alternative that improves the performance of serverless computing platforms in machine learning inference workloads.

The remainder of the paper is organized as follows: in Section 6.2, we elaborate on the details of MLProxy. In Section 6.3, we present the experimental evaluation of the proposed optimizer. Section 6.4 outlines the experimental results achieved. In Section 6.5, we survey the latest related work in the optimization of machine learning inference workloads. Section 6.6 summarizes our findings and concludes the paper.

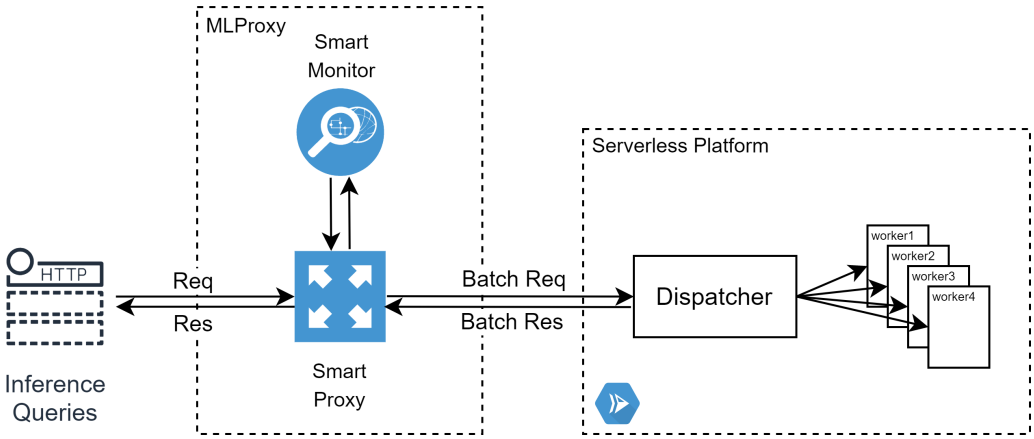


Figure 6.1: MLProxy Overview.

6.2 ML Proxy

Given the importance of batching requests together to improve the utilization and cost of serverless machine learning deployments, and considering the fact that the current generation of API gateways don't support batching out of the box, we strive to design an SLA-aware batch optimization platform for these workloads. To facilitate a smooth transition from the default serverless deployment to our framework, we designed MLProxy as a drop-in replacement for API gateway systems already existent

in every serverless computing platform, and it will provide instant improvements in cost efficiency while making sure that the deployment stays SLA-compliant. In this section, we will go over our design to address these shortcomings.

6.2.1 System Architecture

Figure 6.1 shows an overview of the MLProxy architecture. As can be seen, MLProxy acts as an adaptive reverse proxy that can function as a drop-in replacement for the API Gateway and comprises two modules: 1) Smart Proxy; and 2) Smart Monitor.

The Smart Proxy module is designed to accept incoming HTTP requests, group them using our dynamic batching algorithm, and send them to their respective upstream serverless platform as soon as either the maximum batch size is reached or the timeout has expired.

In order for the Smart Proxy module to be able to function properly, set accurate timeouts for batches, and to facilitate accurate dynamic batching, we needed a smart monitoring system that is tailored to the specific characteristics of machine learning inference workloads with varying batch sizes. To this end, we designed Smart Monitor as a module that works alongside the Smart Proxy module and provides several statistical insights regarding windowed latencies for different batch sizes from the upstream serverless computing platform.

To properly function, MLProxy needs workload configurations, describing the Service Level Objectives (SLO) and endpoints to the upstream serverless platform. Using live data from the monitoring component, the smart proxy component is able to improve the cost and reliability of the system by leveraging dynamic batching.

6.2.2 Monitoring

To properly adjust queue size and timeout configurations, MLProxy needs live monitoring data from the backend serverless computing platform with estimated response times for different queue sizes. To make this possible, the proposed monitoring compo-

nent organizes observed response time values for batch requests made to the upstream serverless computing platform. To ensure we are using the latest latency values, we use a sliding window to only use the latest response time values in our estimations.

To ensure that the platform is complying with the SLO configuration, the monitoring service also logs the end-to-end response time observed by the user using a sliding window. Using this value, the system can make an informed decision about the queuing policy used for the service. The end-to-end response time includes the time spent processing the request in MLProxy, the queuing time, and the response time of the upstream serverless computing platform.

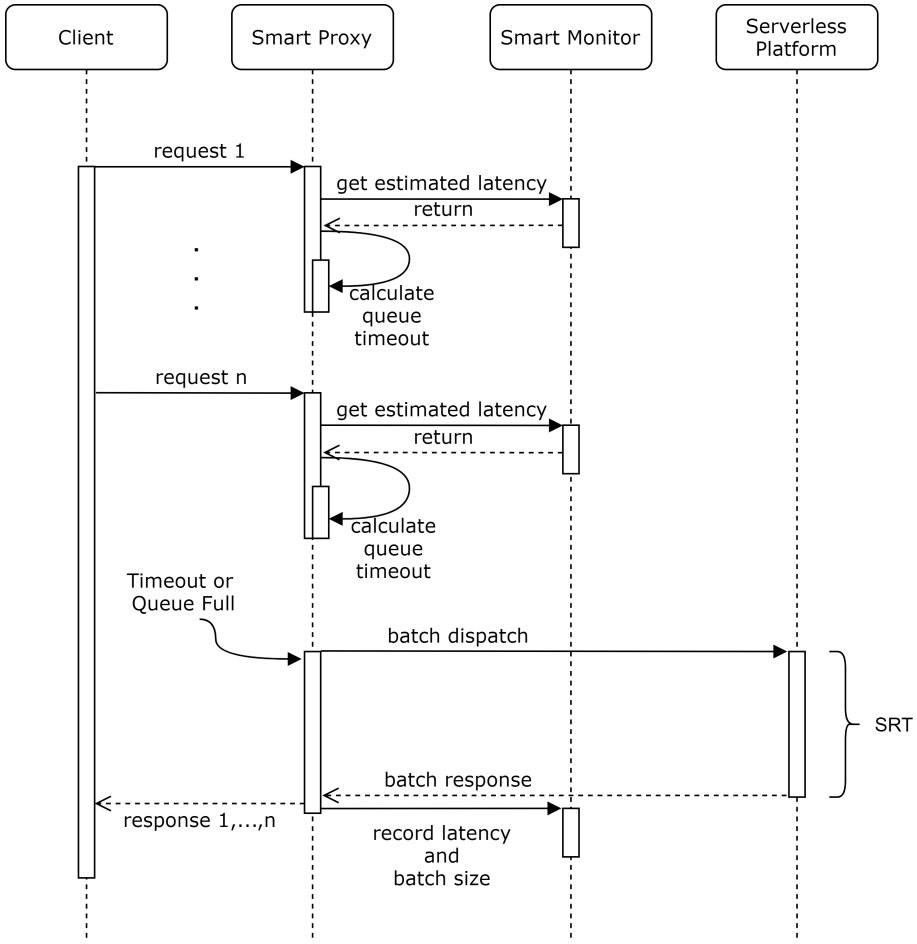


Figure 6.2: UML Sequence Diagram.

6.2.3 Smart Proxy

The Smart Proxy component of MLProxy is responsible for creating batches of requests from individual incoming requests while ensuring end-to-end latency observed by all requests comply with the set SLO configuration.

Figure 6.2 shows the UML sequence diagram of a sample batch of requests created by the Smart Proxy component. Algorithm 2 shows the algorithm used by this component to calculate the timeout and handle the dispatch logic of the batch. When a request is received by the Smart Proxy, it will be added to the current batch of requests. There are two triggers that can cause the current batch of requests to be dispatched to the upstream serverless platform: 1) batch size reaching the maximum batch size; and 2) reaching the queue timeout. The maximum batch size is set by the dynamic batching subsystem and is responsible for reducing the cost of deployment while maintaining the quality of service. The queue timeout is a high-frequency configuration that will be re-evaluated on the arrival of every request, making sure no request exceeds the latency allowed by the SLA.

6.2.4 Dynamic Batch Optimizer

Batching has been studied by previous work in machine learning inference as means to improve the resource utilization and throughput of the system [129–131, 133]. Using a larger value for the batch size, we can achieve a lower deployment cost and higher throughput, but at the expense of causing a higher latency for the service. The dynamic batching subsystem is responsible for choosing a batch size that can improve the performance of the system while ensuring SLA compliance of the system with the current arrival rate observed by the system.

To achieve the set goals of the system, we adopted an additive increase multiplicative decrease method for setting the batch size of the system. In this methodology, by default, we will add a configurable constant amount to the batch size of the system unless we see a violation of a set of goals for the system. In case of a violation, to

Algorithm 2: An overview of the high-frequency queue scheduler.

```
Input: Max_BS — maximum batch size
Input: RT_SLO — response time specified in SLO
Output: metrics
BS ← 0 ;                                          // the size of the current batch
FRT ← reset ;                                     // first request timer
while True do
    wait for new arrival or timeout;
    if new arrival then
        cancel previous timeout;
        if BS=0 then
            FRT ← reset
        end
        BS ← BS + 1;
        P95_est ← 95th percentile serverless latency for BS + 1;
        DTO ← RT_SLO − P95_est;
        TO ← DTO − FRT;
        if BS = Max_BS then
            dispatch current batch to serverless platform;
            BS ← 0;
        else
            set timeout to TO;
        end
    end
    if timeout then
        dispatch current batch to serverless platform;
        BS ← 0;
    end
end
```

ensure the quality of the service remains within the predefined bounds, we will use a multiplicative decrease on the batch size. Algorithm 3 shows an overview of the logic used by this component.

One of the violations that can cause the scheduler to decrease the batch size is the latency of the system. As discussed earlier, the SLA requirements of the workload can be configured by the developer. To ensure the SLA compliance of the workload, we set a lower threshold than SLO (80% of the SLO response time in our experiments) for the system to follow. Whenever the SLO latency of the system goes beyond this threshold, we trigger a decrease in the batch size.

Another violation used to decrease the batch size in the proposed scheduler is when we have too many requests being timed out. This can show that our set batch size

Algorithm 3: An overview of the low-frequency dynamic batch optimizer.

Input: RT — current response time
Input: TO — current ratio of batches being dispatched due to timeout
Input: TO_thresh — batch timeout threshold set by the user
Input: RT_SLO — response time specified in SLO

```
1  inc_step ← 1;
2  dec_mult ← 0.8;
3  Max_BS ← 1;
4  while True do
5  |   RT, TO ← updated monitoring data;
6  |   violation ← False;
7  |   if TO > TO_thresh or RT > RT_SLO then
8  |   |   violation ← True;
9  |   end
10 |   if violation = True then
11 |   |   Max_BS ← Max_BS × dec_mult;
12 |   else
13 |   |   Max_BS ← Max_BS + inc_step;
14 |   end
15 |   wait for 30 seconds;
16 end
```

is too large for the current arrival rate, and as a result, the system might introduce very large batch sizes that can interfere with the functionality of the system.

6.2.5 Queuing Scheduler

Figure 6.2 shows the UML sequence diagram of a typical batch processed in the system. To ensure the incoming requests are being responded to in a timely fashion that complies with the set SLO, we developed a high-frequency queue timeout calculator implemented as a part of our smart proxy module. Our timeout calculator calculates a safe timeout for dispatching requests to the upstream serverless platform in a way that even the oldest request in the queue is responded to before the SLO target latency.

To estimate the overall request latency, we need to estimate the upstream serverless platform's latency. However, the upstream inference latency depends on the batch size that is sent to the platform. As a result, the timeout needs to be calculated on each arrival to adapt to the new queue size (N_q). To create such a model, we use the data gathered by our monitoring service about the 95th percentile of the latency of

the previous requests with a batch size of $N_q + 1$ to ensure we can fulfill the request before the deadline, even with the possible arrival of a new request. To calculate the dispatch timeout, we use the following:

$$DTO = SLO_T - RT95_{N_q+1} \tag{6.1}$$

where DTO is the dispatch timeout, SLO_T is the Service Level Agreement Target, $RT95_{N_q+1}$ is the 95th percentile of a batch request with $N_q + 1$ requests in it, and N_q is the current batch size. However, as the timeout will be re-calculated on each arrival, we will set the next timeout starting from the arrival of the oldest request to make sure all requests will be fulfilled in time:

$$TO = DTO - FRT \tag{6.2}$$

where TO is the resulting timeout and FRT (First Request Timer) is the amount of time since the first request in the queue has arrived. Knowing FRT is important since to avoid surpassing the SLO threshold, we need to consider how long the oldest request in the queue has already waited.

Due to the adaptive nature of the algorithm, DTO might end up as a negative value on each calculation. In such cases, we dispatch the batch to the upstream serverless platform immediately to avoid SLO violations.

6.3 Experimental Evaluation

In this section, we introduce our evaluation of MLProxy using experimentation on our Knative installation. However, please keep in mind that the same methodology can be used on Google Cloud Run, which is a managed Knative offering on the Google Cloud Platform (GCP). The code for performing and analyzing the experiments used in this section can be found in our public GitHub repository[1].

[1]https://github.com/pacslab/serverless-ml-serving

Table 6.1: Configuration of each VM in the experiments.

Property	**Value**
vCPU	8
RAM	30GB
HDD	180GB
Network	1000Mb/s
OS	Ubuntu 20.04
Latency	<1ms

6.3.1 Experimental Setup

To perform our experiments, we used 4 Virtual Machines (VMs) on the Compute Canada Arbutus cloud[2] with the configuration shown in Table 6.1. With this configuration, we were able to utilize 27 vCPU cores for pods running the user's code. For our cluster, we used Kubernetes version *1.20.5* with Kubernetes client (kubectl) version *1.20.0*. For the client, we used *Python 3.8.5*. To generate client requests based on a Poisson process, we used PACSWG workload generation library[3] which is publicly available through PyPi[4]. The result is stored in a CSV file and then processed using Pandas, Numpy, and Matplotlib. The dataset, parser, and the code for extraction of system parameters and properties are also publicly available in the project's GitHub repository.

6.3.2 Workloads

Table 6.2 shows an overview of the workloads used in our experimental studies. As can be seen, we have experimented on a variety of different machine learning tasks with a wide range of complexity levels to see how batching would affect these scenarios.

[2]https://docs.computecanada.ca/wiki/Cloud_resources
[3]https://github.com/pacslab/pacswg
[4]https://pypi.org/project/pacswg

Table 6.2: List of workloads used in our experiments. The docker container for all workloads along with their code and datasets are publicly available on the project's repository. The baseline response time for each service with 1 vCPU and 1 GB of memory is shown in the complexity column.

Name	**Packaging Lib**	**ML Lib**	**Dataset / Model**	**Complexity**
SKLearn Iris	BentoML [134]	Scikit-learn [135]	Scikit-learn [135]	Very Low (8ms)
Keras Toxic Comments	BentoML [134]	Keras / TensorFlow [136]	Jigsaw Toxic Comments [137]	Low (40ms)
ONNX ResNet50	BentoML [134]	ONNX [138]	ONNX Model Zoo [139]	High (201ms)
PyTorch Fashion MNIST	BentoML [134]	PyTorch [140]	TorchVision Fashion MNIST	Medium (125ms)
TFServing MobileNet	TFServing [141]	TensorFlow [142]	MobileNet V1 100x224	Medium (83ms)
TFServing ResNet	TFServing [141]	TensorFlow [142]	ResNet V2 fp32	High (204ms)

6.3.3 Workload Characterization

In this section, we will go through some workload characterizations that help us understand our workloads and how we can find workloads that benefit most from our novel dynamic batching. As the proposed MLProxy uses batching to improve throughput and cost [131], there needs to be *some* benefit in batching requests together for a given workload. Due to fine-grained billing in serverless computing, if the service time scales linearly with the batch size, there is no benefit in batching requests together. However, for workloads with low to medium complexity on CPU and workloads running on accelerated hardware (e.g., GPU or TPU), service time scales sub-linearly with the batch size, and thus we can expect improved throughput and deployment cost for these workloads.

Figures 6.3 and 6.4 show how the response time and time per inference (response time divided by batch size) evolve as we increase the batch size, compared with a batch

size of one. The linear baseline shows a hypothetical workload where the average response time grows linearly with increasing the batch size. The reason behind this baseline is that in a workload where the average response time grows linearly with the baseline, the time per inference remains constant for different batch sizes. Thus, these workloads don't benefit from batching in serverless computing platforms. As can be seen, in many workloads, time per inference is shorter for larger batch sizes due to lower computational overhead per inference. Thus, batching queries together can increase the efficiency and resource utilization of the workload, improving the cost and performance of the deployment.

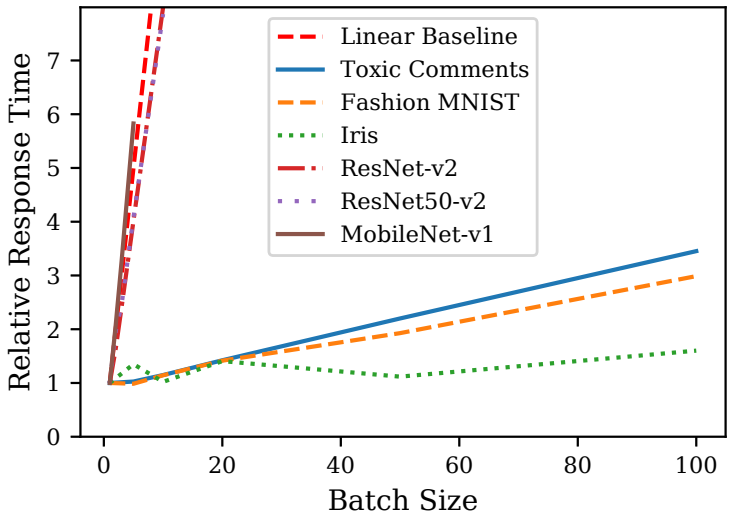


Figure 6.3: The relative response time against the batch size. Relative response time shows how the average response time grows when increasing the batch size. The linear baseline shows the relative response time that grows perfectly linear with the batch size.

6.3.4 Service-Level Objectives (SLOs)

Service-Level Agreements (SLAs) are defined around a specific service and serve the purpose of forming an agreement between the client and the provider of the service, laying out the metrics by which the service is measured and the penalties if the service

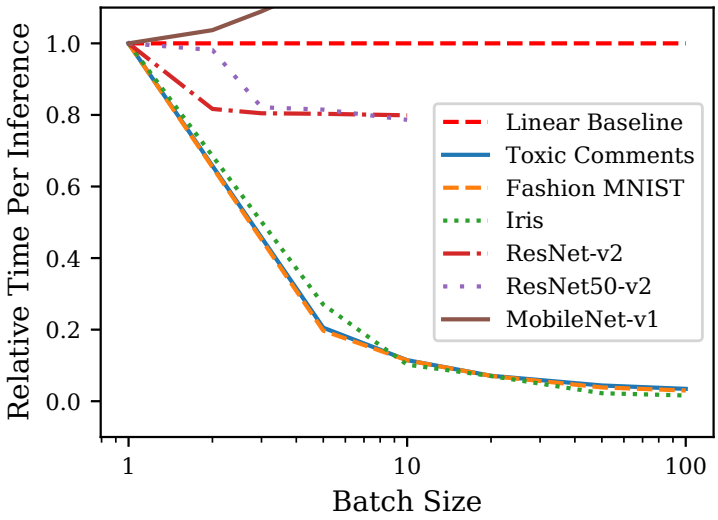


Figure 6.4: The relative average time per inference against the batch size. For many workloads, increasing the batch size results in a reduction in time spent for each inference by reducing the overhead for each query. For a workload with a response time that grows linearly with the batch size, the time per inference remains the same with any batch size.

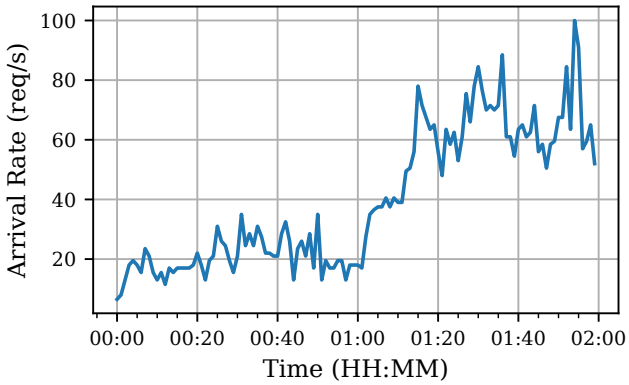
level agreed upon is not reached. Service-Level Objectives (SLOs) serve as the target for a given service metric, e.g., average or 95th percentile response time. SLAs are a more preferred and far more accurate way of representing the needs of the client from the service as they list out the most important criteria for the client.

In this work, our goal is to improve the cost and throughput of serving machine learning workloads on serverless computing platforms while ensuring the agreed-upon level of service is satisfied. For this purpose, we used one of the most common metrics in SLAs, which is the 95th percentile of the response time. However, both the percentile and the threshold used are configurable in our implementation of MLProxy.

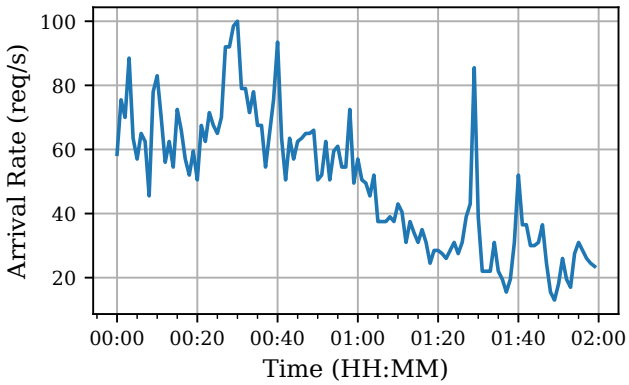
6.3.5 Real-World Traces

To emulate real-world scenarios, we have used the AutoScale real-world traces NLAR T4 and T5 and FIFA World Cup [143]. However, to match the capacity of our cluster and to imitate different load intensities, we scaled the maximum arrival rate for our

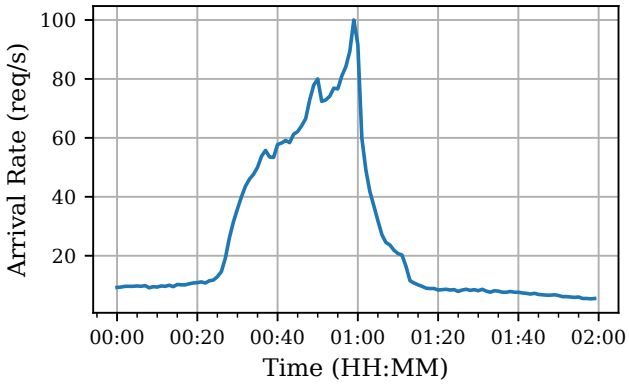
experiments. Figure 6.5 shows these traces scaled to have a maximum arrival rate of 100 requests per second.



(a) AutoScale NLAR T4 Trace.



(b) AutoScale NLAR T5 Trace.



(c) AutoScale FIFA World Cup Trace.

Figure 6.5: The trace patterns used in the experiments [143].

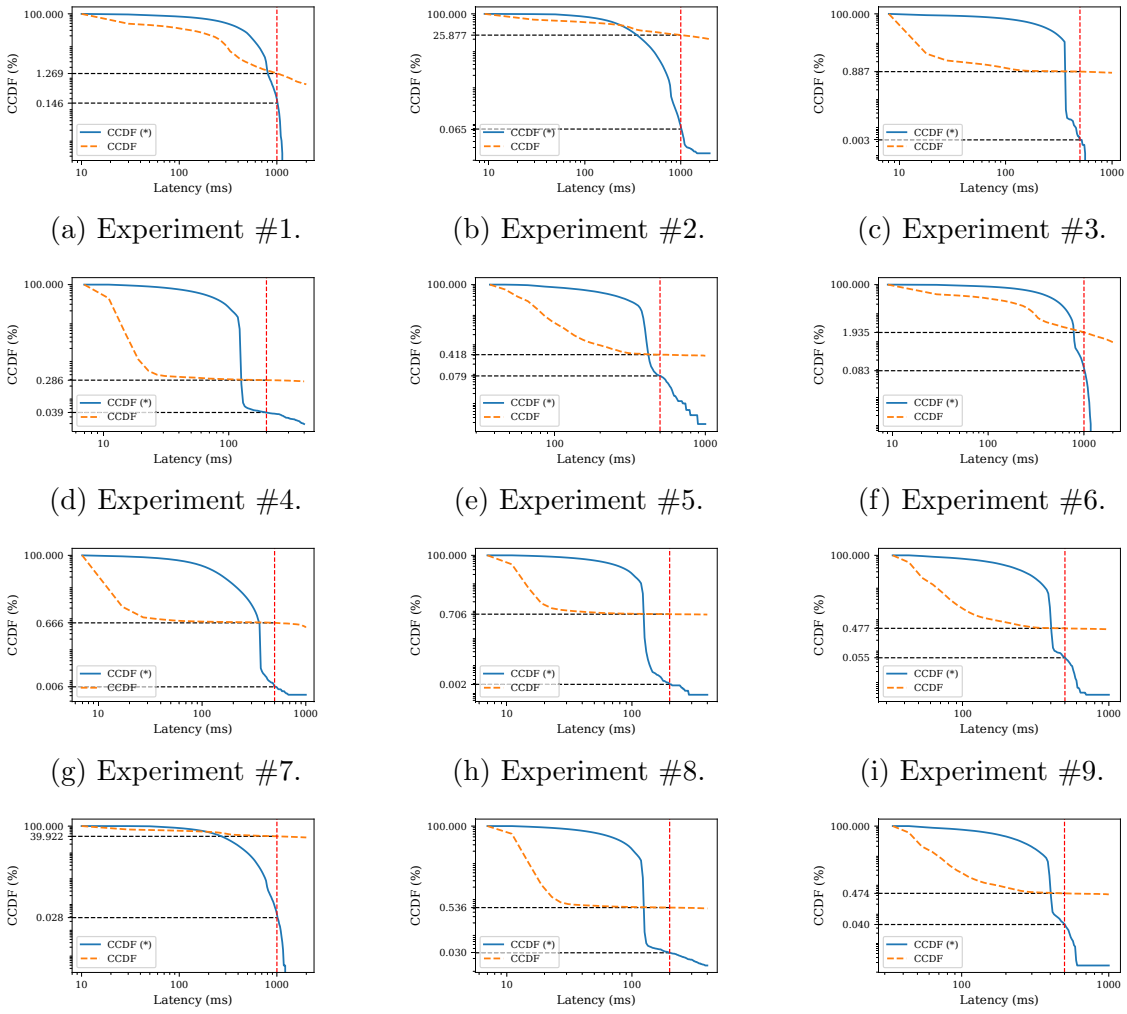
Table 6.3: List of a variety of experimental results. BRT shows the baseline average response time with a concurrency of one and batch size of one and SLO P95 is the 95th percentile response time specified in SLO. Number of containers is shown as the most important deployment cost indicator. Note that the columns specified with an asterisk (*) are the results with MLProxy turned on.

#	**Workload**	**Trace**	**Max RPS**	**BRT (ms)**	**SLO P95 (ms)**	**# of Cont.**	**# of Cont.***	**% of SLO Viol.**	**% of SLO Viol.***	**Avg. BS**
1	Fashion MNIST	WC	30	125	500	2.73	1.00 (↓63.4%)	1.2799	0.1861 (↓85.5%)	4.93
2	Fashion MNIST	WC	100	125	1000	8.75	1.01 (↓88.5%)	26.0048	0.0767 (↓99.7%)	10.93
3	Iris	WC	50	8	500	1.61	1.00 (↓38.1%)	0.8892	0.0033 (↓99.6%)	5.01
4	Iris	WC	185	8	200	1.50	1.01 (↓32.8%)	0.2862	0.0395 (↓86.2%)	6.57
5	Toxic Comments	WC	30	40	500	1.90	1.00 (↓47.2%)	0.4181	0.0811 (↓80.6%)	3.09
6	Fashion MNIST	T5	30	125	500	4.28	1.00 (↓76.6%)	1.9688	0.1002 (↓94.9%)	9.81
7	Iris	T5	185	8	500	3.01	1.00 (↓66.7%)	0.6675	0.0059 (↓99.1%)	18.95
8	Iris	T5	185	8	200	3.01	1.00 (↓66.7%)	0.7064	0.0019 (↓99.7%)	11.00
9	Toxic Comments	T5	50	40	500	3.87	1.00 (↓74.2%)	0.4771	0.0553 (↓74.2%)	7.71
10	Fashion MNIST	T4	100	125	1000	13.34	1.07 (↓92.0%)	39.9915	0.0038 (↓99.9%)	13.34
11	Iris	T4	185	8	200	1.93	1.00 (↓48.3%)	0.5361	0.0295 (↓94.5%)	13.06
12	Toxic Comments	T4	50	40	500	3.12	1.00 (↓67.9%)	0.4737	0.0405 (↓91.4%)	6.12

6.4 Experimental Results and Discussion

In this section, we will go through our experimental results and discuss them in detail. Please note that the codes and scripts used to deploy and experiment with workloads and analyze the results is publicly accessible in the project's GitHub repository[5].

[5]https://github.com/pacslab/serverless-ml-serving



(a) Experiment #1. (b) Experiment #2. (c) Experiment #3.

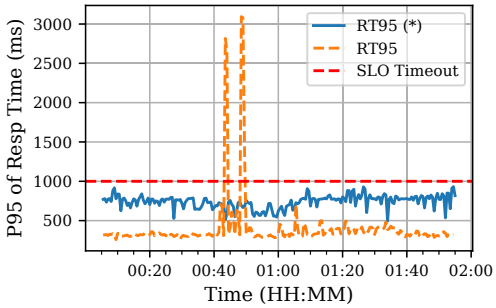
(d) Experiment #4. (e) Experiment #5. (f) Experiment #6.

(g) Experiment #7. (h) Experiment #8. (i) Experiment #9.

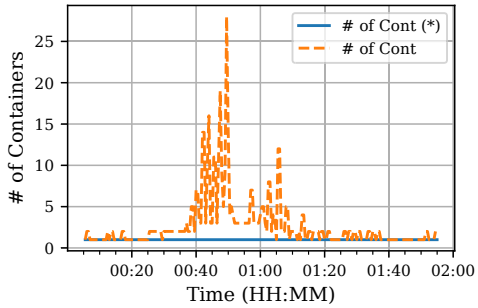
(j) Experiment #10. (k) Experiment #11. (l) Experiment #12.

Figure 6.6: The Complementary CDF (CCDF) results of experiments listed in Table 6.3. The red dashed horizontal line shows the SLO set for the experiment and the vertical bars signify the total SLO miss rate of experiments with and without MLProxy optimizer. Plots marked with an asterisk (*) are the results with MLProxy turned on.

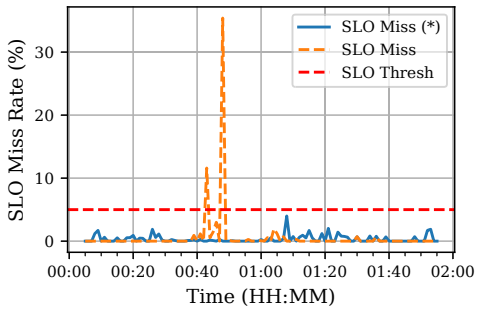
Table 6.3 lists a large number of experiments we have performed on our cluster to determine the efficiency of our method. Out of the workloads listed in Table 6.2, we have only done further experimentation on the ones that would benefit from batching. To further show details of the response times achieved in our experiments with and without using the proposed method, the Complementary Cumulative Density Function (CCDF) plot of the experiments listed in Table 6.3 have been shown in



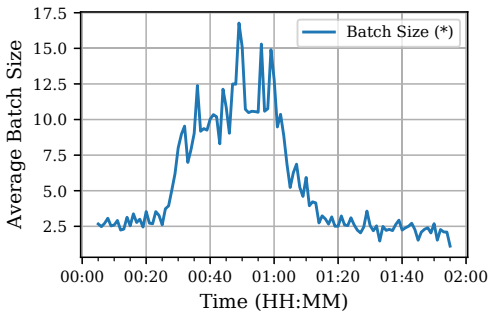
(a) The 95th percentile of the response time throughout the experiment.



(b) Number of containers over time throughout the experiment.



(c) The SLO miss rate over time throughout the experiment.



(d) Average batch size over time throughout the experiment.

Figure 6.7: The experimental results when applying the world cup trace to the Fashion MNIST workload with a scaled maximum arrival rate of 30 requests per second and 95th percentile of response time SLO set to 500ms on the optimizer engine.

Figure 6.6. We have also included other plots about the details of experiment #1 in Figure 6.7.

6.4.1 SLA Compliance

As shown in Table 6.3, MLProxy was able to significantly and consistently reduce the SLA violations in our experiments while reducing the cost of deployment by reducing the number of containers needed. But to understand how MLProxy is able to achieve this, we need to look at Figure 6.7 for more details. Figure 6.7a shows the 95th percentile of response time throughout the experiment. According to the SLA in this experiment, our goal is to keep the 95th percentile of the response time below 1000 ms. As shown in this figure, this value for when we are not using MLProxy is

farther away from the SLO threshold, while MLProxy allows the 95th percentile to stay closer to the SLO threshold to be able to batch requests together.

When the demand for our service (i.e., arrival rate) increases, MLProxy is able to allow a larger maximum batch size (shown in Figure 6.7d) while still ensuring that the service abides by the SLA. By leveraging the larger maximum batch size and due to the improved resource utilization made possible through batching, we can ensure an optimized deployment while maintaining a low SLO violation.

You can see more details about the distribution of the response times in Figure 6.6. As shown, MLProxy increases the latency for a portion of requests that would otherwise be responded to much faster than what is needed according to the SLO. However, this gives MLProxy the flexibility needed to ensure a larger portion of requests is handled before the SLO threshold.

6.4.2 Resource Usage

As can be seen in Table 6.3, MLProxy reduced the number of containers needed to handle incoming requests between $32\% - 92\%$ in our experiments. This improvement is a result of utilizing the existing instances better and with lower overhead for each inference. Due to the complex pricing schemas of Kubernetes offerings, we opted to use the container count as a proxy to represent the deployment cost as it is often proportional to the cost of deployment across different vendors.

It is worth mentioning that the MLProxy deployment used less than $200MB$ of memory and 10% of a single virtual CPU core in all experiments. As a result, the MLProxy deployment introduces negligible overhead and can also be deployed by the cloud provider as an optional module on the API Gateway offerings.

6.4.3 Discussions and Limitations

In previous sections, we overviewed the experimental results of the proposed method. As shown, MLProxy shows significant improvements in terms of both SLO violations

and the cost of deployment. This can be achieved by leveraging the flexibility allowed by SLO. In general, the more flexible the SLO is, the better the improvements would be when using MLProxy. However, these improvements mainly come from batching the requests together when sending them to the upstream serverless platform. As a result, the benefits of MLProxy are bound by the benefits achievable by batching the requests together. If, for a given workload, batching doesn't reduce the overhead per request in a meaningful way, it wouldn't benefit from using the proposed method due to the linear billing nature of serverless computing platforms.

Another effect of MLProxy on the system is reducing the frequency of change in the resources used by a deployment. As a result, less resources will be dedicated to scaling the deployment, and this can help us approach the performance of serverfull deployments while still getting the benefits of serverless platforms.

As shown in Table 6.2, batching can reduce the overhead of the inference per request. Other works in the field have also shown the benefits of batching requests in improving the device utilization and cost [129–131, 133] and these benefits are expected to grow with the introduction of accelerated hardware into the systems. As a result, the methodology proposed here would be even more beneficial in future serverless computing platforms.

While MLProxy addresses many concerns in machine learning inference serving on serverless computing platforms, there are still a few limitations associated with it. As discussed earlier, the benefits of using MLProxy mainly come from the benefits of batching requests together. Thus, for workloads where the overhead reduction is negligible in larger batch sizes, MLProxy can't bring a lot of benefits. This is because in serverless computing platforms, we have fine-grained pricing, and because of this, a single instance running for n request durations would cost the same as n instances running for 1 request duration. Another limitation when using MLProxy is that to be able to achieve better performance by batching requests together, the SLO should be flexible enough to allow a batch of a few requests without causing violations. In

other words, the SLO threshold cannot be smaller than the 95th percentile of the response time of the serverless platform when using a batch of a few requests.

6.5 Related Work

Many studies have proposed methods to improve machine learning inference workloads on serverless computing platforms. In [130], The authors propose InferLine, which is a cost-aware optimizer that tries to find the optimal configuration to maintain a specified tail latency SLO by configuring the hardware type, batch size, and the number of replicas according to the model and arrival process. The approach taken by the authors is very promising but requires extra integration and control on the infrastructure and cannot function on serverless computing platforms. Ali et al. [127] evaluate adaptive batching for inference serving on serverless platforms. They used analytical performance models along with workload profiling and linear regression to find the optimal configuration for a given deployment. Although very promising, their work requires prior access to the deployed model and a profiling step that needs to be updated to be kept up to date. In addition, they do not investigate the performance of existing serverless systems and pure serverless model serving systems while only working with AWS Lambda. Zhu et al. [144] introduced Kelp, a software runtime that strives to isolate high-priority accelerated ML tasks from memory resource interference. They argue that in using accelerated machine learning, contention on host resources can significantly impact the efficiency of the accelerator. They show that their approach can improve the system efficiency by 17%. In [129], the authors propose MArk, a predictive resource autoscaling algorithm aiming to simplify machine learning inference and make it SLA-aware and cost-effective by combining IaaS, Spot Instances, and FaaS. In their design, they allow batching on accelerated deployments that use GPU/TPUs on IaaS to process large bulks of requests and use scale-per-request FaaS (AWS Lambda) as a tool to handle unforeseen surges in requests. In their approach, they were able to achieve an improved tail

latency compared to state of the art in the industry while reducing the cost. The approach proposed in this work requires prior profiling steps that render it inefficient for serverless with pay-per-use pricing systems. Crankshaw et al. [131] proposed adaptive batching for serverfull deployments and were able to reduce the deployment cost and increase the throughput while meeting SLA. Yadwadkar et al. [145] go over some of the challenges that arise for serving machine learning workloads like heterogeneous hardware and software, designing proper user interfaces, and building SLO-driven systems. In their work, the authors try to make a case for managed and model-less inference serving systems. Although non-trivial, our study could be a first step towards fully managed and cost-effective machine learning serving. Chahal et al. [146] used a recommender system as an example ML-based system to compare different deployment strategies on the cloud that result in the desired performance at a minimal cost. In their experiments, they compared serverless and serverfull deployments and found that serverless deployments deliver better performance for bursty workloads and functions with short execution time and low resource requirements. Wu et al. [132] investigated the possibility of using serverless computing for ML serving and found that serverless computing platforms can deliver on ML serving goals: high performance, low cost, and ease of management. They found small memory size, limited running time, and lack of persistent state to be the most limiting factors when deploying ML serving workloads on serverless computing platforms. Benesova et al. [147] explore the possibility of BERT-style text analysis on serverless platforms. Their approximation methods allow the deployment of these models on serverless platforms, eliminating the server management overhead and reducing the deployment costs. Gunasekaran et al. [99] uses serverless computing alongside VM-based autoscaling with predictive and reactive controllers in order to improve SLO while reducing costs for machine learning inference workloads. In their work, they found that when the arrival rate is low, serverless computing platforms could be more cost-efficient that VMs because of their ability to scale to zero. Using our proposed platform, one can benefit

the scale-to-zero capabilities of serverless computing while still having the ability to serve high-traffic workloads. In previous studies, we have developed and evaluated steady-state and transient performance models along with simulators for serverless computing platforms [106, 107, 116] with homogeneous workloads. However, the unique characteristics and challenges in machine learning inference workloads, along with the ever-lasting need for adaptive methods for optimization components, led to the development of MLProxy.

Several of the recent studies investigated different methods to optimize machine learning serving workloads. Romero et al. [110] proposed a model-less and managed inference serving system. In this work, the authors generate model variants using layer fusion or quantization to create models with varying performance/cost. They were able to find the best combination of VMs to serve workloads using only high-level details about the Queries Per Second (QPS), latency requirements, acceptable accuracy, and cost. This approach can help improve serverfull deployments. However, it does not adapt to serverless environments and cannot work with applications where approximate solutions are unacceptable. Lwakatare et al. [148] investigated the challenges faced for developing, deploying, and maintaining ML-based systems at large scales in the industry. They found several challenges according to adaptability, scalability, safety, and privacy. They found the most challenges currently faced to be in relation to adaptability and scalability. In [149], the authors introduce Swayam, an engine for distributed autoscaling to meet SLAs for machine learning inference. To improve resource utilization, the authors allow real-time and batch requests with different levels of SLA to enter the system and they use global state estimation from local data to drive the autoscaling algorithm. In [133], the authors go over the KFServing project that aims to allow scale-to-zero for machine learning serving workloads using Knative. Another benefit of using Kubernetes for serverless inference is found to be the ability to use GPUs while benefiting from autoscaling provided by serverless computing.

There are studies in the literature that attempted to use serverless computing to improve the performance and possibly cost of training machine learning models. Feng et al. [150] investigated the challenges faced when using serverless computing platforms for training machine learning models. They found that serverless computing platforms could be leveraged for hyper-parameter tuning of smaller machine learning models to provide better parallelism. They found the most challenging issues of using serverless computing runtimes for training machine learning models to be their ephemerality, statelessness, and warm-up latency. In [151], Jiang et al. present a comparative study of distributed ML training over FaaS and IaaS. They found that serverless training is only cost-effective with models that have a reduced communication overhead and quick convergence.

Other studies have focused on investigating the challenges and opportunities of different paradigms in cloud computing for inference in machine learning systems. Lwakatare et al. [128] investigated the most challenging aspects of integrating machine learning systems into software products and found dataset assembly and model creation, training, evaluation, and deployment to be the most important ones. Elordi et al. [152] evaluated deployment of Deep Neural Network (DNN) models on a serverless computing platform like AWS Lambda. They used common workloads used in MLPerf [153] and found that increasing the memory of the serverless deployment has a huge impact on the response time but a less dramatic effect on the total throughput of the system. In their experiments, they found the serverless computing platform to be capable of making 51-83 inferences per second, making serverless a suitable deployment point for DNN workloads.

6.6 Conclusion

In this chapter, we presented and evaluated MLProxy which is an adaptive reverse proxy to support efficient and SLA-aware batching for machine learning inference serving types of workloads. We analyzed and evaluated both deployment cost and

performance implications of MLProxy in current serverless computing platforms and we showed the effectiveness of the proposed method through experimentation. We also showed that MLProxy can work across different machine learning libraries. The proposed system can be used by serverless computing providers as a part of their API Gateway offering, significantly improving their system efficiency and competitive advantage. It can also be deployed by the developers in a Kubernetes cluster supporting their serverless deployment with minimal added cost due to the low resource needs of the system.

In summary, the proposed reverse proxy can help developers use the deployed resources in their serverless deployments more efficiently while maintaining the SLA requirements of their machine learning serving system.

Chapter 7

Conclusions, Contributions, and Future Directions

This chapter concludes this thesis with conclusions, a summary of contributions, and some promising paths for future directions.

7.1 Conclusions

In this thesis, our main focus has been on modelling, analyzing, and optimizing the performance of serverless computing algorithms. Throughout the study, we have gone over different aspects of inefficiencies in the current generation of serverless computing platforms and proposed ways to mitigate them.

Chapter 2 outlines the role and importance of function placement algorithms in serverless computing platforms and outlines the design of Smart Spread, a smart function placement algorithm that is workload-aware and offers performance and cost improvements over the previous generations. To allow this, our method uses workload profiling to understand the resource needs of a given function and leverages neural networks to find the best-performing virtual machine for running that workload.

Chapter 3 proposes a novel analytical performance modelling approach that can help predict, analyze, and optimize different aspects of scale-per-request serverless computing platforms. We showed how the proposed model can be leveraged to find the best system configurations for a given workload and to find performance and cost

implications of changes in the system. We also demonstrated that the proposed performance model can handle large deployments by analyzing its tractability. Serverless application developers can use the presented model to predict the cost and performance of their workloads, which improves the predictability of serverless computing platforms. Serverless providers can leverage such performance models to offer better cost-performance tradeoffs to the developers using the platform.

Chapter 4 proposes an analytical performance model for metric-based autoscaling in serverless computing platforms. In this chapter, 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 discussed how metric-based serverless providers can use the proposed model to create adaptive target value configuration systems.

There are inherent limitations to what can be achieved using performance models, e.g., limitations on the arrival and departure processes. To allow the research community, serverless developers, and serverless providers to be able to quickly predict the cost, performance, and limitations of a given deployment when the necessary conditions for an analytical performance model are not met, we introduced a highly accurate and flexible serverless simulator in Chapter 5. In addition to the simulator, SimFaaS, we also developed multiple tools for performance modelling researchers that can give them insights and details about several internal properties of serverless computing platforms that are not measurable by or visible to serverless developers. Since developing analytical performance models is hard and time-consuming, SimFaaS can be used by serverless providers to test out new ideas quickly by customizing certain behaviours of the simulator.

Chapter 6 proposes MLProxy, an adaptive reverse proxy to support efficient and SLA-aware batching for machine learning inference serving types of workloads. Previous studies showed that one of the main reasons that this type of workload hasn't been migrated to serverless computing platforms is their poor cost to performance

ratio in these platforms and batching can help mitigate these inefficiencies. However, static batching needs developer intervention and expertise which is against the serverless computing motto. MLProxy has been designed to allow dynamic and adaptive batching for such use cases and has been shown to improve both the cost and number of SLA violations.

7.2 Contributions

In this dissertation, we aimed at providing tools, methods, models, and technologies that can help improve the performance, cost, efficiency, and overall usability of serverless computing platforms. We began with reviewing the state of the research and industry of serverless computing platforms and found some gaps and shortcomings in the current serverless offerings and proposed methods and models that can help address them. The primary contributions of this work can be listed as follows:

1. Providing in-depth analysis on the latest developments in serverless computing platforms and finding three types of autoscaling in the current generation of serverless computing platforms: 1) scale-per-request; 2) metric-based; and 3) resource-based autoscaling.

2. Design and development of accurate and tractable performance models for serverless computing platforms using scale-per-request and metric-based autoscaling.

3. Development of experimentation methods and tools for steady-state and transient analysis of the performance of serverless computing platforms.

4. Design and development of Smart Spread, a novel container placement algorithm based on machine learning that uses profiling and VM resource usage statistics to find the best performing VM for a given workload.

5. Performing what-if analysis on adaptive expiration threshold for a range of serverless computing workloads.

6. Providing the research community with a dataset of over one month of experimental results on AWS Lambda.

7. Design and development of SimFaaS, a highly-customizable performance simulator aimed at serverless computing platforms with extensibility and ease of use in mind. We have shown how SimFaaS can be leveraged to develop performance models or predict quality and cost characteristics of a given workload when a performance model is not applicable.

8. Design and development of MLProxy, an SLA-aware reverse proxy with adaptive batching for serverless computing platforms. MLProxy was shown to be very effective in reducing the cost of deployment while improving the SLA violations of the deployment. We also showed how MLProxy can be implemented as a part of API gateway by the serverless provider.

7.3 Future Directions

In this section, we will go over some of the possible avenues that can be pursued in future work.

Serverless computing has brought several improvements compared to previous paradigms in cloud computing. Besides, the faster startup times of serverless computing, compared to containers or virtual machines, have enabled faster autoscaling [2]. However, there are several shortcomings in the current generations for large-scale high-throughput applications used by a large number of users, like the limitations on the maximum concurrency achievable [61] and very high costs compared to microservices on workloads with high demand [10, 103, 154, 155]. Due to these limitations, some recent works [99, 154, 156, 157] have focused on exploring hybrid serverless/mi-

croservice deployment of workloads where we leverage the microservices' ability to achieve cost-effectiveness and high throughput while utilizing the serverless computing's fast scaling ability to handle surges in arrivals.

A possible future direction for this research would be to explore automatic hybrid deployment of workloads on serverless computing and microservice platforms while performing smarter workload assignments using analytical and data-driven methods.

Optimization of serverless computing platforms is a non-trivial task due to several contradicting criteria that need to be considered. Data-driven methods using Statistical, Machine Learning (ML), and Reinforcement Learning (RL) methods have been leveraged in recent studies [27, 97, 158–162] to improve several characteristics of many computer systems. In a previous study [14], we improved some key characteristics of a given serverless computing platform by optimizing the function placement algorithm. A direction for future research is leveraging ML and RL-based techniques to improve the management layer of serverless computing platforms, far beyond the status quo, using data-driven innovations that can work autonomously with almost no manual intervention and tuning necessary.

As mentioned in Chapter 1, one of the tasks that are delegated to the provider in the serverless computing paradigm is autoscaling of the resources for a task (or more specifically, a function). Autoscaling is inherently a difficult task, even when done by developers who know the resource requirements and procedures followed by a function. However, it becomes more difficult for the serverless provider as they will need to perform autoscaling in a black-box manner and do it in a way that works properly for all types of workloads. This brings out several new challenges in this task that need to be handled.

In Section 1.2, we talked about how current generations of serverless computing platforms are performing autoscaling. We also talked about the weaknesses associated with those design choices and how they might prove to be inefficient in some cases (depending on the type of workload being executed on them). One of the major

possible future directions of our work is to address these shortcomings by introducing new and adaptive autoscaling algorithms that improve the performance, energy consumption, and resource utilization of the serverless computing platforms. In previous works [163, 164], we leveraged the power of neural networks and other data-driven approaches in conjunction with control theory and graph algorithms to create autonomous autoscalers capable of deriving the best approach to scaling the resources allocated to a specific workload on the cloud. One could build upon these methods to find efficient autoscaling algorithms for serverless computing platforms.

Bibliography

[1] Amazon Web Services Inc., *Serverless Computing*, https://aws.amazon.com/serverless/, Last accessed 2019-07-04. [Online]. Available: https://aws.amazon.com/serverless/.

[2] E. Jonas *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv preprint arXiv:1902.03383*, 2019.

[3] I. Baldini *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, Springer, 2017, pp. 1–20.

[4] H. Yaron, *CNCF Serverless whitepaper*, https://github.com/cncf/wg-serverless/tree/master/whitepaper.

[5] 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)*, IEEE, 2016, pp. 261–268.

[6] Google, Inc., *Manage a cluster of Linux containers as a single system to accelerate Dev and simplify Ops*, http://kubernetes.io/, Oct. 2018. [Online]. Available: http://kubernetes.io/.

[7] Mesosphere, Inc., *Marathon Recipes*, https://mesosphere.github.io, Oct. 2018. [Online]. Available: https://mesosphere.github.io/marathon/docs/recipes.

[8] Docker, *Create a Docker Swarm manager*, https://docs.docker.com/swarm/reference/manage/, Oct. 2018. [Online]. Available: https://docs.docker.com/swarm/reference/manage/.

[9] E. van Eyk *et al.*, "The spec-rg reference architecture for faas: From microservices and containers to serverless platforms," *IEEE Internet Computing*, vol. 23, no. 6, pp. 7–18, 2019. DOI: 10.1109/MIC.2019.2952061.

[10] 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)*, 2018, pp. 133–146.

[11] 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, e4792, 2018.

[12] 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*, Springer, 2019, pp. 174–185.

[13] 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)*, IEEE, 2018, pp. 159–169.

[14] 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*, 2019, pp. 203–213.

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

[16] 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.

[17] 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)*, IEEE, 2018, pp. 181–188.

[18] 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*, ACM, 2018, pp. 21–24.

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

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

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

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

[23] Y. Cui, *How does language, memory and package size affect cold starts of aws lambda?* https://tinyurl.com/ycyd4okl, Oct. 2018. [Online]. Available: https://tinyurl.com/ycyd4okl.

[24] V. Holubiev, *Aws lambda performance issues*, https://serverless.zone/my-accidental-3-5x-speed-increase-of-aws-lambda, Oct. 2018. [Online]. Available: https://serverless.zone/my-accidental-3-5x-speed-increase-of-aws-lambda.

[25] S. Hong, A. Srivastava, W. Shambrook, and T. Dumitras, "Go serverless: Securing cloud via serverless design patterns," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, USENIX Association, 2018.

[26] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[27] M. Shahrad *et al.*, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," *arXiv preprint arXiv:2003.03423*, 2020.

[28] C. L. Abad, E. F. Boza, and E. Van Eyk, "Package-aware scheduling of faas functions," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 101–106.

[29] 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.

[30] Amazon Web Services, *AWS Lambda*, https://aws.amazon.com/lambda, Oct. 2018. [Online]. Available: https://aws.amazon.com/lambda.

[31] Google, Inc., *Google Cloud Functions*, https://cloud.google.com/functions, Oct. 2018. [Online]. Available: https://cloud.google.com/functions.

[32] Microsoft, *Microsoft Azure Functions*, https://azure.microsoft.com/en-ca/services/functions, Oct. 2018. [Online]. Available: https://azure.microsoft.com/en-ca/services/functions.

[33] IBM, *IBM Cloud Functions*, https://console.bluemix.net/openwhisk, Oct. 2018. [Online]. Available: https://console.bluemix.net/openwhisk.

[34] Huawei, *FunctionStage*, https://www.huaweicloud.com/en-us/product/functionstage, Oct. 2018. [Online]. Available: https://www.huaweicloud.com/en-us/product/functionstage.

[35] Google, Inc., *Advanced Scheduling in Kubernetes*, https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes, Oct. 2018. [Online]. Available: https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes.

[36] K. Ye *et al.*, "Profiling-based workload consolidation and migration in virtualized data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 878–890, 2014.

[37] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, "Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds," in *IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, IEEE Computer Society, 2012, pp. 73–80.

[38] Cybera, *Rapid access cloud*, https://www.cybera.ca/services/rapid-access-cloud, Oct. 2018. [Online]. Available: https://www.cybera.ca/services/rapid-access-cloud.

[39] A. Kopytov, "Sysbench manual," *MySQL AB*, pp. 2–3, 2012.

[40] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[41] W. J. Lloyd *et al.*, "Demystifying the clouds: Harnessing resource utilization models for cost effective infrastructure alternatives," *IEEE Transactions on Cloud Computing*, vol. 5, no. 4, pp. 667–680, 2017.

[42] F. Yin *et al.*, "Cloud-scale java profiling at alibaba," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ACM, 2018, pp. 99–100.

[43] 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)*, IEEE, 2018, pp. 227–234.

[44] V. Apte, T. Viswanath, D. Gawali, A. Kommireddy, and A. Gupta, "Autoperf: Automated load testing and resource usage profiling of multi-tier internet applications," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ACM, 2017, pp. 115–126.

[45] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance prediction of component-based applications," *Journal of Systems and Software*, vol. 74, no. 1, pp. 35–43, 2005.

[46] S. M. Sadjadi *et al.*, "A modeling approach for estimating execution time of long-running scientific applications," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 2008, pp. 1–8.

[47] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, "A new container scheduling algorithm based on multi-objective optimization," *Soft Computing*, vol. 22, no. 23, pp. 7741–7752, 2018.

[48] D. Chahal and B. Mathew, "Prowl: Towards predicting the runtime of batch workloads," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ACM, 2018, pp. 199–200.

[49] M. Gribaudo, M. Iacono, and M. Kiran, "A performance modeling framework for lambda architecture based applications," *Future Generation Computer Systems*, vol. 86, pp. 1032–1041, 2018.

[50] C. Witt, M. Bux, W. Gusew, and U. Leser, "Predictive performance modeling for distributed computing using black-box monitoring and machine learning," *arXiv preprint arXiv:1805.11877*, 2018.

[51] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *ACM Sigplan Notices*, ACM, vol. 47, 2012, pp. 3–14.

[52] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, 2007, pp. 249–258.

[53] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *European Conference on Parallel Processing*, Springer, 2005, pp. 196–205.

[54] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "A devops architecture for continuous delivery of containerized big data applications," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, IEEE, 2017.

[55] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *Proceedings of the 2017 Symposium on Cloud Computing*, ACM, 2017, pp. 452–465.

[56] M. R. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari, "A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1442–1455, 2018.

[57] C.-Z. Xu, J. Rao, and X. Bu, "Url: A unified reinforcement learning approach for autonomic cloud management," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 95–105, 2012.

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

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

[60] W. Whitt, "Continuous-Time Markov Chains," *http:// www. columbia. edu/ w̃w2040/ 6711F13/ CTMCnotes120413.pdf*, 2006.

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

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

[63] Amazon Web Services Inc., *AWS Lambda*, https://aws.amazon.com/lambda/, Last accessed 2020-02-03. [Online]. Available: https://aws.amazon.com/lambda/.

[64] Google Inc., *Cloud Functions*, https://cloud.google.com/functions, Last accessed 2020-02-03. [Online]. Available: https://cloud.google.com/functions.

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

[66] IBM Inc., *IBM Cloud Functions*, https://cloud.ibm.com/functions, Last accessed 2020-02-03. [Online]. Available: https://cloud.ibm.com/functions.

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

[68] Google Inc., *Cloud Run*, https://cloud.google.com/run, Last accessed 2020-02-03. [Online]. Available: https://cloud.google.com/run.

[69] OpenFaaS Ltd., *OpenFaaS - Serverless Functions Made Simple*, https://www.openfaas.com/, Last accessed 2020-02-03. [Online]. Available: https://www.openfaas.com/.

[70] Kubeless Inc., *Kubeless*, https://kubeless.io/, Last accessed 2020-02-03. [Online]. Available: https://kubeless.io/.

[71] Fission Contributors, *Serverless Functions for Kubernetes - Fission*, https://fission.io/, Last accessed 2020-02-03. [Online]. Available: https://fission.io/.

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

[73] I. E. Akkus *et al.*, "{SAND}: Towards High-Performance Serverless Computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 923–935.

[74] 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.

[75] M. Armbrust *et al.*, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

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

[77] 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*, Springer, 2009, pp. 571–576.

[78] 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*, IEEE, 2011, pp. 87–92.

[79] 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.

[80] 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*, IEEE, 2011, pp. 1–6.

[81] 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*, IEEE, 2011, pp. 105–112.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] W. J. Lloyd *et al.*, "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.

[87] 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.

[88] H. Wu *et al.*, "A Reference Model for Virtual Machine Launching Overhead," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 250–264, 2014.

[89] 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.

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

[91] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, "Archipelago: A scalable low-latency serverless platform," *arXiv preprint arXiv:1911.09849*, 2019.

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

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

[94] 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)*, IEEE, 2019, pp. 272–280.

[95] J. M. Hellerstein *et al.*, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[96] D. Balla, M. Maliosz, C. Simon, and D. Gehberger, "Tuning Runtimes in Open Source FaaS," in *International Conference on Internet of Vehicles*, Springer, 2019, pp. 250–266.

[97] 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*, Springer, 2018, pp. 171–186.

[98] 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*, 2019, pp. 1–16.

[99] 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)*, IEEE, 2019, pp. 199–208.

[100] 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)*, IEEE, 2019, pp. 9–16.

[101] J. Kuhlenkamp, S. Werner, M. C. Borges, D. Ernst, and D. Wenzel, "Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-driven Design of Serverless Applications," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1576–1585.

[102] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[103] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "A programming model and middleware for high throughput serverless computing applications," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 106–113.

[104] D. Jackson and G. Clynch, "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 154–160.

[105] M. Villamizar *et al.*, "Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures," *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 233–247, 2017.

[106] N. Mahmoudi and H. Khazaei, "Performance Modeling of Serverless Computing Platforms," *IEEE Transactions on Cloud Computing*, pp. 1–15, 2020. DOI: 10.1109/TCC.2020.3033373.

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

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

[109] 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*, 2010, pp. 125–132. DOI: 10.1109/PRDC.2010.30.

[110] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, *Infaas: A model-less and managed inference serving system*, 2020. arXiv: 1905.13348 [cs.DC].

[111] 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)*, 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548070.

[112] 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. DOI: 10.1109/TVT.2016.2580571.

[113] The Knative Authors, *Knative*, https://knative.dev, Last accessed 2021-02-01. [Online]. Available: https://knative.dev.

[114] The Knative Authors, *Metrics*, https://knative.dev/docs/serving/autoscaling/autoscaling-metrics/, Last accessed 2021-02-01. [Online]. Available: https://knative.dev/docs/serving/autoscaling/autoscaling-metrics/.

[115] 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. [Online]. Available: https://towardsdatascience.com/markov-chain-analysis-and-simulation-using-python-4507cee0b06e.

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

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

[118] 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*, 2019, pp. 13–18.

[119] 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)*, IEEE, 2020, pp. 142–152.

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

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

[122] 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*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, 138–149, ISBN: 9781450369916. DOI: 10.1145/3358960.3379124. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3358960.3379124.

[123] Amazon Web Services Inc., *Amazon DynamoDB Read/Write Capacity Mode*, https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html, Last accessed 2020-11-20, 2020. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html.

[124] J. Manner, "Towards performance and cost simulation in function as a service," *Proc. ZEUS (accepted)*, 2019.

[125] E. F. Boza, C. L. Abad, M. Villavicencio, S. Quimba, and J. A. Plaza, "Reserved, on demand or serverless: Model-based simulations for cloud budget planning," in *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, IEEE, 2017, pp. 1–6.

[126] H. Jeon, C. Cho, S. Shin, and S. Yoon, "A cloudsim-extension for simulating distributed functions-as-a-service," in *2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, IEEE, 2019, pp. 386–391.

[127] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00073.

[128] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson, and I. Crnkovic, "A taxonomy of software engineering challenges for machine learning systems: An empirical investigation," in *International Conference on Agile Software Development*, Springer, Cham, 2019, pp. 227–243.

[129] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 1049–1062, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/zhang-chengliang.

[130] D. Crankshaw *et al.*, "Inferline: Latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, Virtual Event, USA: Association for Computing Machinery, 2020, 477–491, ISBN: 9781450381376. DOI: 10.1145/3419111.3421285. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3419111.3421285.

[131] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pp. 613–627, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[132] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi, *Serverless model serving for data science*, 2021. arXiv: 2103.02958 [cs.DC].

[133] C. Cox, D. Sun, E. Tarn, A. Singh, R. Kelkar, and D. Goodwin, *Serverless inferencing on kubernetes*, 2020. arXiv: 2007.07366 [cs.DC].

[134] The BentoML Authors, *Bentoml: Model serving made easy*, https://docs.bentoml.org, Last accessed 2021-07-26. [Online]. Available: https://docs.bentoml.org.

[135] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[136] The Keras Authors, *Keras: The python deep learning api*, https://keras.io, Last accessed 2021-09-10. [Online]. Available: https://keras.io.

[137] The Jigsaw/Conversation AI Team, *Jigsaw multilingual toxic comment classification*, https://www.kaggle.com/c/jigsaw-multilingual-toxic-comment-classification/overview, Last accessed 2021-07-26. [Online]. Available: https://www.kaggle.com/c/jigsaw-multilingual-toxic-comment-classification/overview.

[138] The ONNX Authors, *Open neural network exchange*, https://onnx.ai, Last accessed 2021-09-10. [Online]. Available: https://onnx.ai.

[139] The ONNX Authors, *Onnx model zoo*, https://github.com/onnx/models, Last accessed 2021-09-10. [Online]. Available: https://github.com/onnx/models.

[140] The PyTorch Authors, *Pytorch*, https://pytorch.org, Last accessed 2021-09-10. [Online]. Available: https://pytorch.org.

[141] The Tensorflow Authors, *Tensorflow serving*, https://github.com/tensorflow/serving, Last accessed 2021-09-10. [Online]. Available: https://github.com/tensorflow/serving.

[142] The Tensorflow Authors, *Tensorflow*, https://github.com/tensorflow/tensorflow, Last accessed 2021-09-10. [Online]. Available: https://github.com/tensorflow/tensorflow.

[143] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, Nov. 2012, ISSN: 0734-2071. DOI: 10.1145/2382553.2382556. [Online]. Available: https://doi.org/10.1145/2382553.2382556.

[144] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: Qos for accelerated machine learning systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 172–184. DOI: 10.1109/HPCA.2019.00036.

[145] N. J. Yadwadkar, F. Romero, Q. Li, and C. Kozyrakis, "A case for managed and model-less inference serving," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19, Bertinoro, Italy: Association for Computing Machinery, 2019, 184–191, ISBN: 9781450367271. DOI: 10.1145/3317550.3321443. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3317550.3321443.

[146] D. Chahal, M. Mishra, S. Palepu, and R. Singhal, "Performance and cost comparison of cloud services for deep learning workload," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21, Virtual Event, France: Association for Computing Machinery, 2021, 49–55, ISBN: 9781450383318. DOI: 10.1145/3447545.3451184. [Online]. Available: https://doi.org/10.1145/3447545.3451184.

[147] M. Suppa, K. Benešová, and A. Švec, "Cost-effective deployment of BERT models in serverless environment," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, Online: Association for Computational Linguistics, Jun. 2021, pp. 187–195. DOI: 10.18653/v1/2021.naacl-industry.24. [Online]. Available: https://www.aclweb.org/anthology/2021.naacl-industry.24.

[148] L. E. Lwakatare, A. Raj, I. Crnkovic, J. Bosch, and H. H. Olsson, "Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions," *Information and Software Technology*, vol. 127, p. 106 368, 2020, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106368. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301373.

[149] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: Distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware '17, Las Vegas, Nevada: Association for Computing Machinery, 2017, 109–120, ISBN: 9781450347204. DOI: 10.1145/3135974.3135993. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3135974.3135993.

[150] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 334–341. DOI: 10.1109/CLOUD.2018.00049.

[151] J. Jiang *et al.*, "Towards demystifying serverless machine learning training," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD/PODS '21, Virtual Event, China: Association for Computing Machinery, 2021, 857–871, ISBN: 9781450383431. DOI: 10.1145/3448016.3459240. [Online]. Available: https://doi.org/10.1145/3448016.3459240.

[152] U. Elordi, L. Unzueta, J. Goenetxea, S. Sanchez-Carballido, I. Arganda-Carreras, and O. Otaegui, "Benchmarking deep neural network inference performance on serverless environments with mlperf," *IEEE Software*, vol. 38, no. 1, pp. 81–87, 2021. DOI: 10.1109/MS.2020.3030199.

[153] V. J. Reddi *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459. DOI: 10.1109/ISCA45697.2020.00045.

[154] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," *arXiv preprint arXiv:2006.03720*, 2020.

[155] A. Eivy and J. Weinman, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[156] Z. Li *et al.*, "Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2020, pp. 399–408.

[157] A. Reuter, T. Back, and V. Andrikopoulos, "Cost efficiency under mixed serverless and serverful deployments," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2020, pp. 242–245.

[158] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev, *Sizeless: Predicting the optimal size of serverless functions*, 2020. arXiv: 2010.15162 [cs.DC].

[159] S. Kehrer, D. Zietlow, J. Scheffold, and W. Blochinger, "Self-tuning serverless task farming using proactive elasticity control," *Cluster Computing*, pp. 1–19, 2020.

[160] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, IEEE, 2020, pp. 129–138.

[161] L. Schuler, S. Jamil, and N. Kühl, *Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments*, 2020. arXiv: 2005.14410 [cs.DC].

[162] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2020.

[163] M. Sabuhi, N. Mahmoudi, and H. Khazaei, "Optimizing the Performance of Cloud Software Systems using Adaptive PID-Controllers," *ACM Transactions on Autonomous and Adaptive Systems*, pp. 1–27, 2021.

[164] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian, "A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications," in *International Conference on Cloud Computing and Services Science*, ser. CLOSER '21, Online: Springer, 2021, pp. 1–12.