Defining Service Level Agreements in Serverless Computing

Mohamed Elsakhawy, *The University of Western Ontario*

Supervisor: Bauer, Michael A., *The University of Western Ontario*

Abstract

The emergence of serverless computing has brought significant advancements to the delivery of computing resources to cloud users. With the abstraction of infrastructure, ecosystem, and execution environments, users could focus on their code while relying on the cloud provider to manage the abstracted layers. In addition, desirable features such as autoscaling and high availability became a provider's responsibility and can be adopted by the user's application at no extra overhead.

Despite such advancements, significant challenges must be overcome as applications transition from monolithic stand-alone deployments to the ephemeral and stateless microservice model of serverless computing. These challenges pertain to the uniqueness of the conceptual and implementation models of serverless computing. One of the notable challenges is the complexity of defining Service Level Agreements (SLA) for serverless functions. As the serverless model shifts the administration of resources, ecosystem, and execution layers to the provider, users become mere consumers of the provider's abstracted platform with no insight into its performance. Suboptimal conditions of the abstracted layers are not visible to the end-user who has no means to assess their performance. Thus, SLA in serverless computing must take into consideration the unique abstraction of its model.

This work investigates the Service Level Agreement (SLA) modeling of serverless functions' and serverless chains' executions. We highlight how serverless SLA fundamentally differs from earlier cloud delivery models. We then propose an approach to define SLA for serverless functions by utilizing resource utilization fingerprints for functions' executions and a method to assess if executions adhere to that SLA. We evaluate the approach's accuracy in detecting SLA violations for a broad range of serverless application categories. Our validation results illustrate a high accuracy in detecting SLA violations resulting from resource contentions and provider's ecosystem degradations. We conclude by presenting the empirical validation of our proposed approach, which could detect Execution-SLA violations with accuracy up to 99%.

Keywords

Serverless Computing, FaaS2F, SLA, Execution Performance, VM Placement.

Summary for Lay Audience

Serverless computing has brought significant advancements to the delivery of cloud computing to end-users. By adopting the ephemeral, short-lived microservice model, developers can avoid the administration and operational overheads of previous cloud delivery models and focus on the application's code. Despite such advancement, at its current state, FaaS currently lacks the transparency to allow users to judge the performance of the FaaS platforms. Users have limited insight into the execution performance of their microservices, i.e., serverless functions, and consequently are unable to improve such performance.

This work investigates the factors that impact serverless functions' performance. We demonstrate that end-user choices and providers' ecosystems can significantly influence how a serverless function performs. We propose a framework that users and providers can leverage to assess the performance of serverless functions and define performance guarantees, i.e., SLAs. The framework leverages resource utilization traces of a serverless function to infer its performance and detect degradations that impact the SLA. We apply the framework to serverless applications of different sizes and application categories and examine the accuracy of detecting SLA violations.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Michael Bauer, for his continuous support, invaluable guidance, accommodation, and patience throughout the Ph.D. journey. Prof. Bauer's support, constructive feedback, and highly focused critique were essential in enabling my growth in carrying on independent research. His mentorship, advice, and continuous accommodation in meeting tight deadlines can not be forgotten and are very much appreciated.

Words can not sufficiently express my thanks to my mother, Arifa, and my late father, Fouad, who would have been very proud today. Their upbringing, encouragement, and unconditional support have shaped the person I am today.

I can not be thankful enough to my wife Hanaa for her tremendous support and encouragement throughout the Ph.D. Her emotional support and the support of our two young sons, Ziyad and Taher, have helped me cross the tough times in this journey, to which I am very grateful.

Special thanks also go to my sister, brothers-in-law, father-in-law, and mother-in-law for their support.

Last but not least, I would also like to thank the program committees and the reviewers of IEEE Cloud, Smart-Cloud, Cloud-Summit, ACM-Middleware's WoSC, and ICECCE. Their feedback was extremely valuable and has helped introduce new perspectives to the work's analysis and potential expansions.

Table of Contents

Abstract i

Summary for Lay Audience iv

Acknowledgments (if any) v

Table of Contents vi

List of Tables (where applicable) ix

List of Figures (where applicable) xi

Chapter 1 Cloud Computing 1

1.1 Introduction 1

1.1 Cloud Computing 3

1.2 IaaS Delivery Model 4

1.3 The SaaS and PaaS models 5

1.4 The Function as a Service delivery model 7

Chapter 2 Function as a Service “FaaS” 10

2.1 Introduction 10

2.2 A Serverless Function 11

2.3 Serverless Computing Concepts 12

2.4 Serverless Platforms 14

2.5 A Sample Serverless Function 16

2.6 Serverless computing research directions 21

2.7 Open Problems and Our Research Direction 27

Chapter 3 IaaS Cloud usage behaviors: a pilot study 30

3.1 Introduction 30

3,.2 Formulation 34

3.3 Results and Discussion 41

Chapter 4 SLA in Cloud Computing 43
4.1 Introduction 43
4.2 IaaS and PaaS SLA 43
4.3 SLA in Serverless Computing 44
4.4 The Challenge of FaaS SLA 47
Chapter 5 Serverless Execution Performance 49
5.1 Introduction 49
5.2 Serverless Execution Environments 49
5.3 Impact of Skeleton Container and Execution Engine Builds 53
5.4 Impact of Execution Engine Releases 56
5.4.1 AWS Lambda 57
5.4.2 Google Cloud Functions 59
5.4.3 Azure Cloud Functions 61
5.5 Impact of Ecosystem Performance 62
5.6 Discussion 63
Chapter 6 Serverless Performance Guarantees 65
6.1 Introduction 65
6.2 Performance Guarantees for Serverless Functions 65
6.3 Machine Learning in Cloud Computing 66
6.4 Defining Execution-SLA 67
6.5 FaaS2F 68
6.6 FaaS2F Architecture 69
6.7 Resource Utilization Fingerprints Generation 71
6.8 MLM Classifier Training 73
6.9 Assessing Compliance with Execution-SLA 75

6.10 Problem Formulation 75

6.11 Validation 78

6.12 Number of Training Samples 81

6.13 Conclusion 83

Chapter 7 SLA for Serverless Chains 84

7.1 Introduction 84

7.2 Modeling Serverless Chains Performance 84

7.3 Validation 86

7.3.1 Fixed-size Sequential Serverless chains 86

7.3.2 Variable-size Sequential Serverless chains 89

7.3.3 Choice of the classifier 90

Chapter 8 Conclusions, Limitations, and Future Work 92

8.1 Conclusion 92

8.2 Limitations and Future Work 94

References or Bibliography (if any) 97

Curriculum Vitae 105

List of Tables (where applicable)

Table 1: VM lifetimes 36

Table 2: VM Creation Gaps 37

Table 3: Allocated Disk Distributions 37

Table 4: Allocated vCPU Distributions 38

Table 5: Allocated Memory Distributions 38

Table 6: Underutilized PMs 41

Table 7: Benchmarked Execution Environments 54

Table 8: Execution Duration (in Seconds) 55

Table 9: AWS Lambda Execution Durations 57

Table 10: Google Cloud Functions Execution Durations 59

Table 11: Azure Execution Environments Performance 61

Table 12: Execution Durations per induced latency 63

Table 13: Training and Evaluation Samples 80

Table 14: Classification Accuracy 81

Table 15: Classification Accuracy per Training Sample count 81

Table 16: Chains Classification Accuracy 88

Table 17: Chain-1 Per-Function Classification Accuracy 88

Table 18: Chain-2 Per-Function Classification Accuracy 88

Table 19: Chain3 Classification Accuracy 89

Table 20: Chains' RF-based Classification Accuracy 90

List of Figures (where applicable)

Figure 1: Sample Function High-level diagram 18

Figure 2: Sample Function Table Structures 18

Figure 3: A Sample Serverless Function 20

Figure 4: VM Lifetimes 36

Figure 5: VM Creation Gaps 37

Figure 6: Underutilized PMs Percentage 40

Figure 7: Aspects of FaaS execution performance 45

Figure 8: Execution Durations of Benchmarked Environments 55

Figure 9: AWS Lambda Execution Performance 58

Figure 10: AWS Lambda Billing Durations 59

Figure 11: Google Cloud Functions Execution Performance 60

Figure 12: Google Cloud Functions Billing Durations 60

Figure 13: Microsoft Azure Execution Performance 62

Figure 14: Execution Durations per induced latency 63

Figure 15: FaaS2F Diagram 69

Figure 16: FaaS2F High-level Diagram 69

Figure 17: FaaS2F Implementation Diagram 71

Figure 18: Sample scaled fingerprints 72

Figure 19: CNN Architecture 74

Figure 20: Classification Accuracy per Training Sample count 82

Figure 21: Resource utilization fingerprints stacking 85

Figure 22: Fixed-size (a) and Variable-size (b) chains 86

Figure 23: Fixed-size chains workflows 87

Figure 24: Variable-size (Chain 3) workflow 89

Chapter 1
Cloud Computing

1.1 Introduction

In the past decade, cloud computing has caused several successive shifts in traditional computing. With the wide adoption of the Infrastructure as a Service (IaaS) delivery model, infrastructure components became virtualized, abstracted, and consumed on-demand by users as services. Other delivery models soon followed, such as platform as a service (PaaS) and software as a service (SaaS), further abstracting the platform and the software layers. This development, in turn, drove the relinquishing of the overhead incurred from layer management to cloud providers. The emergence of such delivery models presented many advantages, including limited upfront investment, on-demand provisioning, usage-based billing, and scalability. It also added more challenges for cloud providers to address, such as efficient resource allocation, maintaining tenant isolation, and satisfying the objectives of Service Level Agreements (SLAs).

Function as a service (FaaS), commonly referred to as serverless computing, is a relatively newer cloud delivery model that further abstracts execution runtimes. The model allows users to request code execution on-demand and bills users only when their code is triggered and executed, making way for the advent of second-based and sub-second-based billing. FaaS enables users to focus on code development and reduces the time from code writing to deployment and execution by shifting the management and the administration overheads to the cloud provider. However, as a new programming model, FaaS introduces challenges to the end-users and the cloud providers. End-users are required to adopt the programming model of stateless ephemeral microservices and leverage the serverless provider's ecosystem for providing storage and inter-communication services to their serverless functions. Cloud providers also face novel, system-level, and abstraction-level challenges as they build their platforms to support this new model. One of these challenges is the complexity of defining SLAs in serverless computing. An SLA is a written agreement that defines the provider's contractual obligations toward the end-user. These contractual obligations can specify a broad range of service metrics that the provider must satisfy, such as uptime, network latency, and invocation rate.

This research investigates the challenge of defining performance guarantees in serverless computing stemming from its unique abstraction. Our research examines the following research questions:

- What is serverless performance, and what factors affect a serverless function's execution performance?
- How does serverless computing's unique abstraction differentiate it from earlier cloud delivery models, e.g., IaaS and PaaS
- How can Service Level Agreements be defined and assessed for serverless functions and serverless chains?

Our contributions in this work are:

- A categorical classification of the factors that impact serverless execution performance.
- Introduction of the notion of an *execution-SLA* to specify SLAs for serverless functions execution performance.
- A novel method for specifying *execution-SLAs* in serverless computing by defining execution-performance guarantees in terms of resource utilization fingerprints of a function's execution.
- An approach for cloud providers to extract resource utilization fingerprints for functions' executions and utilize them to define *execution-SLAs*.
- A method for cloud providers and users to validate execution-SLA compliance of functions' executions by comparing their resource utilization fingerprints with fingerprints specified in the *execution-SLA*.
- Novel modeling of sequential serverless chains' execution performance using resource utilization fingerprints of the chain's functions. The modeling is used to define an SLA for the serverless chains' performance.

- An approach to validate compliance with the defined SLAs using a Machine Learning (ML) based classifier.

In the next section, we provide a brief review of cloud computing and describe the evolution of its delivery models.

1.2 Cloud Computing

Cloud computing is a computing model in which resources are provided as a general utility that can be leased by users as and when required [1]. It emerged as a response to the information technology (I.T.) industry requirements for flexible infrastructure provisioning, scalability, consolidation, and isolation. One of the first cloud computing delivery models that arose was IaaS, which abstracts the physical compute, storage, and network elements of infrastructure and enables users to provision virtual instantiations of them. With the introduction of Elastic Compute Cloud (EC2) by Amazon Web Services (AWS) in 2006, users were afforded the capacity to create on-demand virtual machines (VMs), incorporated with user-selected CPU and memory specifications to construct virtual networks and to assign virtual storage to VMs. Other companies joined the commercial cloud market soon after capitalizing on the massive demand for IaaS by the I.T. industry.

Cloud computing cleared the way for realizing several features that the IT industry required, such as low up-front investment in hardware and the ability to scale immediately. Over the years, delivery models for cloud computing have grown, starting with IaaS and moving on to SaaS, PaaS, and recently FaaS. These models have varied at the abstraction level of resource delivery. Their offerings range from delivering dedicated, long-lived user-managed VMs in IaaS to providing shared-tenancy, limited-lifetime, platform-managed function executions in FaaS.

We begin with the theoretical background of resource allocation in IaaS platforms, commonly referred to as the VM placement. We choose to present the literature on this topic because of the dependence of recent delivery models of cloud computing (e.g., FaaS and PaaS) on the IaaS delivery model. The literature on IaaS resource allocation is rich, addressing a broad set of objectives pursued by users and providers. The subsequent sections present the VM placement problem, its classifications, objectives, and an overview of proposed solutions. We then direct attention to the FaaS delivery model, discussing our examination of its advantages, disadvantages, and open research challenges.

1.3 IaaS Delivery Model

In IaaS clouds, compute resources are virtualized and provisioned by cloud users as VMs. Platforms bill users for the resources that they consume and the duration of consumption. Users can either request VM creations on-demand in demand-based clouds or request a reservation in advance in reservation-based clouds. One of the critical functions of a cloud platform is to allocate a VM's requested resources to one of the available physical machines (PMs) in a data center, referred to as VM placement. A placement decision must satisfy the placement objectives specified by a cloud provider, for example, minimizing power consumption, meeting the SLA requirements, and reducing data center operating costs. Based on the time at which a placement decision is taken, the VM placement problem can be classified into static placement and dynamic placement [2].

Static VM placement centers on providing an initial VM-to-PM mapping at the VMs' creation time. A significant shortcoming of static placement is its failure to consider the dynamic nature of IaaS Clouds. Resource allocation and deallocation in IaaS clouds are user-driven and are triggered by the non-deterministic VM creation and termination behaviors of end-users. Moreover, VMs are of varying workloads, and thus, placement decisions need adjustment at runtime. Hence, static placement is, alone, insufficient to maintain the satisfaction of placement objectives at runtime. Dynamic VM placement generates an adaptive mapping of VMs to PMs after the initial placement to ensure the accomplishment of the cloud provider objectives. Dynamic placement is triggered either on a platform-set schedule [3], [4] or in response to changes in the active VMs and PMs composition and utilization of active VMs and PMs [5]–[7]. Dynamic placement decisions are executed through workload adjustment mechanisms. VM migration is a popular workload adjustment mechanism wherein VMs migrate between PMs using hypervisor migration functionalities.

While the IaaS model was a breakthrough, the model had a significant shortcoming in resource utilization efficiency. VMs are allocated resources even when there is little or no demand for the hosted applications, and users are billed for the allocated resources even if they are not actively utilized. Additionally, the model required users to assume administrative and ongoing operational responsibility for their VMs, thus requiring a minimum level of technical expertise to adopt this model.

1.4 The SaaS and PaaS models

As discussed in the preceding section, resources are allocated to users in IaaS clouds as VMs. IaaS cloud users are responsible for the administration and maintenance of the provisioned VMs. Besides, commonly desired functionalities such as high availability and auto-scaling must be configured by the users, thus creating technical and administrative overheads that can potentially affect the adoption of IaaS clouds. Software as a Service "SaaS" delivery model was one of the attempts to address this problem. The model [8] emerged to facilitate the delivery of applications to users by masking the complexity of the underlying infrastructure. A SaaS cloud provider is responsible for administrating and configuring the underlying infrastructure and applications, allowing the end-user to leverage the application on-demand. SaaS excelled in domain-specific software, such as email services and CRM software (e.g., Salesforce).

The model had several disadvantages despite enabling users to employ the software without concerning themselves with the underlying infrastructure complexity. SaaS platforms provide a limited set of software selections to choose from and generally lack the configuration flexibility of an IaaS platform. Users must choose between building complex software environments that fit their requirements via IaaS or leverage a "one size fits all" SaaS platform. To approach this shortcoming, PaaS emerged to provide the utility of creating and deploying custom applications to the cloud, leveraging the broad adoption of containers as a growing standard for application packaging [9]. Docker and Linux containers provide a standardized method of packaging and delivering software [9], [10].

In PaaS platforms, such as Heroku[1] and Amazon Elastic Beanstalk[2], users deploy their code or select a pre-packaged application such as MySQL or NGINX. The platform allocates the requested resources and executes the application in a dedicated container managed by the PaaS platform.

A user specifies the sizing of the resources allocated to their application, and the platform deploys and creates an endpoint to interact with it. A PaaS cloud provider administrates the underlying infrastructure and configures commonly desired features, such as high availability. PaaS cleared the way to the advent of a more extensive set of cloud-hosted applications than SaaS and allowed users to focus on the application layer rather than the infrastructure layer.

The progression from IaaS to PaaS enabled users to take advantage of the features of both models by adopting their workloads to fit each model's design. However, multiple shortcomings remain unaddressed. Amongst them is the resource utilization efficiency challenges where resources are allocated to VMs or containers even with no application demand, and users are billed for these idling resources. Also, these models lack user-transparent techniques for scaling up and down the user's application when the demand change. Users must adopt custom ad-hoc methods to enable their applications to autoscale as needed.

These shortcomings are an implicit consequence of several design choices adopted by the IaaS and PaaS models. Both IaaS and PaaS assume an infinite lifetime of the users' provisioned artifacts (VMs or containers), a constant resource consumption throughout the artifact's lifetime, and a tight coupling of computations and data (i.e., state). They expect the users to develop any necessary methods for scaling up and down the workload as demand changes. The models are specifically tailored to satisfy the requirements of persistent workloads, which are application categories with an expected long lifetime and whose data, i.e., state, must be shared between all the running instances of the application.

[1] Heroku: https://www.heroku.com/

[2] AWS Elastic Beanstalk: https://aws.amazon.com/elasticbeanstalk/

Common examples of these workloads are database engines, web portals, and application servers. These applications wait for incoming requests, process them, and then return to waiting for another incoming request. To scale up and down these applications, data clustering technologies are employed to ensure the state is shared amongst all the running instances of the application. The resource utilization shortcoming of the IaaS and PaaS models results from the assumption that any deployed workload is persistent and thus will require resources throughout its lifetime, even when no demand exists.

While persistent workloads are present in practice, not all workloads exhibit their expected long-lifetime or data coupling specifications. Ephemeral workloads, for example, have short lifetimes and segregate computations from data. Every running instance of an ephemeral workload must only access an unshared subset of the state or access no state at all (stateless). Common examples of ephemeral workloads are short-lived data analytics tasks and stateless microservices. While IaaS and PaaS cloud delivery models can satisfy the persistent subset of cloud workloads, the assumption of the workload persistence results in major resource allocation and scaling disadvantages for ephemeral workloads.

1.5 The Function as a Service delivery model

The Function as a Service (FaaS) delivery model was created to address these shortcomings by targeting ephemeral, short-lived workloads. Users leverage the FaaS model by deploying serverless functions that are executed on-demand. A serverless function is a short-lived stateless microservice that decouples data from computations. This decoupling enables seamlessly scaling up and down microservices with demand as the state (i.e., data) is entirely segregated from computations. Additionally, the on-demand execution of serverless functions improves the resource utilization efficiency than prior delivery models as resources are allocated only when the functions are executed. The FaaS model allows fast adoption by requiring users to only submit code to the cloud platform, where this code (i.e., serverless function) is executed when invoked. Users do not need to worry about provisioning VMs or containers or the underlying system administration or configuration as the serverless providers become responsible for these tasks.

While FaaS brought many advantages for hosting ephemeral workloads in the cloud, the model introduced challenges due to its unique abstraction. As cloud providers assume the majority of the administration overheads in the FaaS model, end-users have limited insight into the factors affecting the performance of their serverless functions. Adding further complexity to this challenge, the FaaS model has widely adopted a pay-upon-execution billing scheme where users are charged based on the execution durations of their functions and the allocated resource. A sub-optimal execution of a serverless function results in longer execution durations and consequently higher incurred costs by the end-user. Thus, it is imperative to create the methodologies that enable users and providers to agree on binding Service Level Agreements (SLAs) in the FaaS model.

SLA in serverless computing has not been thoroughly investigated. Limited attempts have examined defining invocation-rate guarantees [13], scheduling latency guarantees [8], [14], [15], and resource allocation guarantees [16]. To our knowledge, no work has examined execution performance guarantees for serverless functions despite their impact on the financial feasibility and attractiveness of the serverless model. Thus, in this work, we investigate the challenge of defining serverless execution performance guarantees. We develop a methodology and an approach that enables end-users and cloud providers to define and assess compliance with performance guarantees for serverless functions. Our contributions in this work are:

- A categorical classification of the factors that impact serverless execution performance.
- Introduction of the notion of an *execution-SLA* as a means of specifying SLAs for serverless functions execution performance.
- A novel method for specifying *execution-SLAs* in serverless computing by defining execution-performance guarantees in terms of resource utilization fingerprints of a function's execution.
- An approach for cloud providers to extract resource utilization fingerprints for functions’ executions and utilize them to define *execution-SLAs*.

- A method for cloud providers and users to validate execution-SLA compliance of functions’ executions by comparing their resource utilization fingerprints with fingerprints specified in the *execution-SLA*.
- Novel modeling of sequential serverless chains' execution performance using resource utilization fingerprints of the chain's functions. The modeling is used to define an SLA for the serverless chains' performance.
- An approach to validate compliance with the defined SLAs using a Machine Learning (ML) based classifier.

The rest of this thesis is structured as follows:

- Chapter 2 discusses the serverless delivery model, the current research directions, and the open problems.
- In Chapter 3, we present the results of our pilot study into the usage trends of IaaS clouds. The results shed light on the nature of workloads in IaaS clouds and highlight the significant existence of short-lived ephemeral workloads. The results illustrate the challenges facing ephemeral workloads in IaaS clouds and justify the need for the serverless model.
- Chapter 4 discusses the SLA definition in Cloud computing and how defining FaaS SLA fundamentally differs from earlier cloud computing models.
- Chapter 5 examines the factors affecting execution performance in serverless computing and demonstrates how seemingly trivial user-made choices can significantly impact the execution performance of a serverless function.
- Chapter 6 discusses FaaS SLA and presents our framework (FaaS2F) for defining and assessing compliance with performance guarantees for serverless functions.
- Chapter 7 discusses execution performance guarantees for sequential serverless chains and presents the empirical results of utilizing FaaS2F to detect serverless chains' suboptimal executions.
- We finally conclude with Chapter 8, discussing the conclusions, the limitations of our work, and our future directions.

Chapter 2
Function as a Service “FaaS”

2.1 Introduction

FaaS, or serverless computing, represents a paradigm shift [11] and a new programming model of stateless, event-driven microservices. It creates the foundation for programmers to build microservices, commonly known as serverless functions, that are executed when invoked, thus relinquishing the resource-idling that occurs when an application awaits invocations in PaaS and IaaS models. From an operational standpoint, serverless computing shifts the responsibility of resources’ administration and provisioning to the cloud provider, thus enabling developers to focus on the application [11].

To leverage serverless computing, applications are broken down into a set of stateless serverless functions that can be invoked independently. *Serverless Functions* are written in high-level languages, and users define the events that trigger the functions’ execution. The platforms specify these events’ sources and vary based on the provider’s ecosystem. Our experimentation with serverless platforms identified a standardization gap in the commercial and the open-source serverless platforms that can cause vendor lock-ins [12].

AWS Lambda[3], Google Cloud Functions[4], Microsoft Azure functions[5], and IBM Cloud functions[6] emerged to capitalize on the IT industry demand for serverless computing. These platforms support cloud functions written in many high-level languages such as Node.js, Python, Java, Go, and C#. A large number of triggering events such as API calls, File upload, and database record inserts are supported by these platforms. Users are required to specify the memory footprint of the function, and the maximum execution time, commonly referred to as function timeout. Users are charged based on the functions’

[3] AWS Lambda: https://aws.amazon.com/lambda/

[4] Google Cloud Functions: https://cloud.google.com/functions/

[5] Azure Cloud Functions: https://azure.microsoft.com/en-ca/services/functions/

[6] IBM Cloud Functions: https://www.ibm.com/cloud/functions

resources and execution duration; hence, function timeouts are utilized to avoid charges when functions take longer than expected to finish.

As with most industry manifestations, open-source alternatives quickly emerged. Some of these open-source platforms were developed solely for accelerating research in the serverless computing field, such as the OpenLambda project [13]. Others, however, provide production-grade alternatives such as Apache Openwhisk[7], Kubeless[8], Fission[9], OpenFaaS[10], and Knative[11]. These platforms currently provide less feature-set than their commercial counterparts but are expected to proliferate due to their open nature that enables community collaborations on their growth.

In the next section, we review the concept of a serverless function, detailing its composition and distinguishing characteristics. We then present the serverless computing concepts and review current commercial and open-source serverless platforms and their operating principles.

2.2 A Serverless Function

Serverless offerings by the commercial and the open-source platforms use different terms to market their offerings. To standardize, we use the term serverless function to refer to these offerings. AWS Lambda, Azure functions, IBM Cloud functions, and Google Cloud functions are all examples of serverless functions. A serverless function is a stateless, ephemeral, and independent software artifact written in a high-level language and executed by a serverless platform when a user-defined triggering event is satisfied.

Serverless platforms require serverless functions to “operate asynchronously and process one request at a time “ [14]. Inputs and outputs to serverless functions have no specific

[7] Apache Openwhisk: http://openwhisk.apache.org/

[8] Kubleless: https://kubeless.io/

[9] Fission: https://fission.io/

[10] OpenFaaS: https://www.openfaas.com/

[11] Knative: https://knative.dev/

requirements, but they are generally advised to produce and consume JSON objects. In traditional programming, applications are composites of functions written in a programming language [15]. Similarly, serverless functions are written in programming languages supported by the serverless platforms and can collectively construct a complex application through composition and orchestration [16].

2.3 Serverless Computing Concepts

In this section, we provide a review of the main concepts of serverless computing and their relationship to serverless functions.

- Triggering events
  - Serverless functions are event-driven [13], [17]. The functions are executed when a user-defined triggering event is satisfied. Serverless platforms support generic triggering events, such as API calls [14] and platform-specific triggering events. The platform-specific triggering events are closely connected to the platform's ecosystem. For example, AWS Lambdas can be triggered at record-insertions into AWS key-value database "*DynamoDB*".
- Resource allocation
  - Serverless platforms are responsible for allocating the resources required for a serverless function's execution [13]. Users are billed for the allocated resources and the duration of execution. The commercial and the open-source platforms we reviewed, except for Microsoft Azure functions, require users to specify the memory footprint of their functions; CPU allocation is computed in proportion to the allocated memory by the platform. Azure functions' platform computes both the memory and CPU allocations for serverless functions.
- Runtime environments:
  - Serverless functions are executed in isolated runtime environments, sometimes referred to as workers [13]. A serverless platform is responsible for creating and terminating workers based on the function's programming language, allocated

resources, and function timeout. A worker is started by the platform when a serverless function's triggering event is satisfied. Container technologies, such as Docker[12], are a popular choice for runtime environment isolation and thus are leveraged by all the open-source serverless platforms we reviewed. Ideally, when a serverless function's execution completes, the platform terminates its assigned worker to free up resources for other executions. However, it has been acknowledged that creating a new worker takes approximately 1-2 seconds in AWS [13]; this phenomenon is referred to as *cold-start latency*. It can significantly increase a function's response latency, and thus, platforms tend to keep worker containers in a warm state to avoid the cold startup time.

- Limited lifetime:
    - Serverless functions are short-lived and thus are required to finish execution within a limited lifetime. A serverless platform is responsible for terminating a function that exceeds the user-defined maximum lifetime (function timeout).
- Stateless nature:
    - Serverless functions are expected not to hold state between invocations. Subsequent invocations may be scheduled to execute on different workers, and thus state sharing must utilize a separate mechanism. If subsequent invocations are executed on the same worker, the common state may be visible but not guaranteed [13]. The serverless platforms we examined leverage their ecosystems for state sharing. Those mechanisms are, however, inadequate, as later discussed in this work.
- Autoscaling
    - The stateless nature of serverless functions enables computation to scale independently of the storage [11]. A serverless platform scales up and down in

[12] Docker: https://www.docker.com/

response to demand increases and decreases by starting workers and terminating them when they are no longer needed.

- Billing:
    - A user is billed only when their function is executed. The function's allocated resources, and the duration of execution determine the cost. Thus, this model represents an advantage over earlier delivery models that billed users for resource reservations.

A single serverless function cannot implement complex workflows for an extensive application. In traditional programming, applications are composed of many functions that exchange inputs and outputs for implementing complex application logic. Similarly, function composition and orchestration in serverless computing are leveraged to construct large applications from functions.

2.4 Serverless Platforms

Commercial and open-source serverless platforms operate on a common principle of allocating runtime environments for stateless high-level language functions. The platforms vary in their supported language runtimes, feature set, and the platform's ecosystem. In this section, we survey the available commercial and open-source serverless platforms.

2.4.1 Commercial Serverless Platforms

Serverless functions' commercial offerings support a growing number of high-level languages. Amazon's AWS Lambdas can be written in C#, GO, Java, Node.js, Python, and Ruby, in addition to supporting custom runtime environments using shell scripting. Microsoft's Azure functions support JavaScript, .Net, Java, or Python and support custom runtime environments using Docker images. IBM's cloud functions similarly support custom runtime environments using Docker and functions written in PHP, Node.js, Python, Ruby, Swift, and GO. Google's cloud functions do not allow custom runtime environments and support functions written in GO, Python, and Node.js

All the commercial platforms we reviewed, except for Microsoft Azure, require users to specify the allocated memory resources for their cloud functions. Azure dynamically allocates memory as needed at runtime. Functions' timeout must be specified by the user for all the reviewed platforms. All the reviewed platforms support Autoscaling through a configurable triggering metric and concurrency of the runtime environments.

To an outsider's eye, serverless functions' offerings by the commercial platforms depict a substantial similarity. However, the platforms are different in their coding requirements, feature-set, default settings, and supported programming languages [18]. Developers are required to adopt vendor-specific programming styles and platform ecosystems, which may result in vendor-lock-ins. Also, the architectures of the commercial platforms are difficult to review and compare due to the closed nature of these platforms.

2.4.2 Open-Source Serverless Platforms

Open-source serverless platforms closely follow the operating principles of their commercial counterparts. To facilitate their deployment, some of the platforms we surveyed leverage containers to package their core services. Container orchestration engines, such as Kubernetes, are widely adopted to coordinate the platform's deployment and coordinate the creation and the termination of serverless functions' runtime environments.

Our review identified several open-source production-grade serverless platforms that vary in their architecture and their maturity levels. OpenFaaS, Kubeless, Knative, and Fission are examples of these platforms that leverage the Kubernetes orchestration engine. These platforms differ in their feature-set and architecture. However, they provide similar end-user offerings with a broad set of supported programming languages and autoscaling features. Mohanty et al. [12] reviewed the architecture of these platforms and highlighted their distinguishing differences. In addition to supporting containerized deployments, some of the platforms we reviewed support stand-alone installations such as Apache Openwhisk, IronFunctions[13], and Fn[14]. We also surveyed Open Lambda [13] platform specifically designed to accelerate research in serverless computing and is not considered production-ready.

2.5 A Sample Serverless Function

To enhance the understanding, we demonstrate a simple use case of a serverless function used by a retail chain to perform three operations: i) distribute the store-item prices from the head office to the stores, ii) collect sales reports from the stores and report back increases and decreases in daily sales, and iii) generate a daily sales report to head-office. Stores invoke an API call at their opening time to retrieve the item prices and another API call at close to report the quantity of sold items and determine the sales performance. Head-office invokes an API call after stores' closing to generate a sales performance report. At the scale of a small number of stores, the use-case warrants neither the expenses nor the administration overhead of dedicated infrastructure and thus benefit from the serverless model.

We leverage Amazon's Lambda platform and implement a serverless function written in Python. Item prices and daily sales records are stored in two tables in Amazon's key-value database DynamoDB[15]. The daily sales report is generated at the head office's request and is stored as a JSON encoded file in a dedicated AWS S3[16] bucket. The workflow for the application is as follows:

- A head-office manager updates the item prices in the DynamoDB "*ItemPrices*" table before stores' opening.

[13] Ironfunctions: https://open.iron.io/

[14] Fn: https://fnproject.io/

[15] DynamoDB: https://aws.amazon.com/dynamodb/

[16] AWS S3: https://aws.amazon.com/s3/

- At opening time, stores invoke an API call to the serverless function that retrieves the item prices from DynamoDB and serves it over HTTP.

- At stores closing time, stores invoke another API call to report their daily sales count. The function retrieves the stores' previous day's sales count from DynamoDB "*SalesCount*" table and compares it with today's sales. It then responds with "*positive*" or "*negative*" to indicate the increase or decrease of daily sales and saves the new sale count to the table.

- Head-office managers invoke an API call to generate a JSON encoded report that is saved to the S3 bucket.

Using the AWS Lambda platform, we created the serverless function and an endpoint API to invoke it. We designated the "*store_mgmt*" Python function as the handler that is executed when the API is invoked. The API accepts POST requests with JSON encoded input to specify the requested action (get prices, report sales or generate a daily report). We utilize AWS SDK (Boto3) for Python to access *DynamoDB* tables and save the JSON encoded report to the S3 bucket. A high-level diagram of the use case is illustrated in Fig 1.

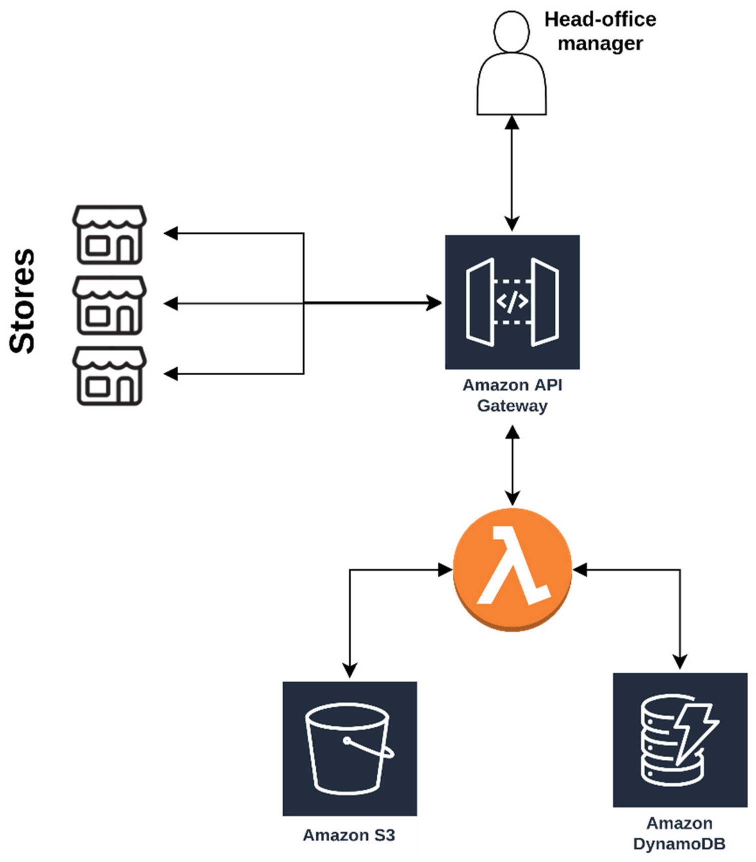
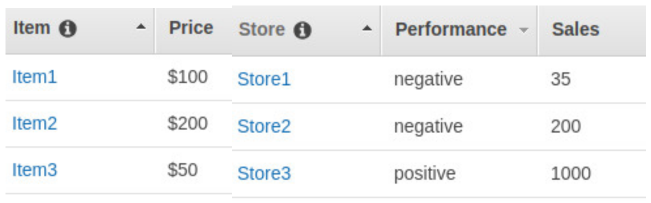


Figure 1: Sample Function High-level diagram

The table structure of *ItemPrices* and *SalesCount* tables is shown in Figure 2



Item	Price
Item1	$100
Item2	$200
Item3	$50

Store	Performance	Sales
Store1	negative	35
Store2	negative	200
Store3	positive	1000

Figure 2: Sample Function Table Structures

The serverless function’s code is illustrated in Figure 3

```
import json
import boto3

def store_mgmt(event, context):
    #Store_mgmt function is the handler that is invoked when the API
    #is called. It then parses the input to determine the action to be executed
    action = event['Action']['Type']
    if action == 'GetPrices':
        return(get_prices())
    elif action == 'ReportSales':
        return(report_sales(event))
    elif action == 'GenerateReport':
        return(generate_s3_report())

def get_prices():
    #Access DynamoDB and retreive the item prices list
    pricestable = 'ItemPrices'
    dynamodb = boto3.resource('dynamodb')
    client = dynamodb.Table(pricestable)
    try:
        priceslist=client.scan()
        priceslist_dict=priceslist['Items']
    except  Exception as e:
        print(str(e))
    return {
        'statusCode': 200,
        'body': json.dumps(priceslist_dict)
    }

def report_sales(event):
    #Parse the input event to retrieve store name and sale count
    storename=event['Action']['StoreName']
    salecount=event['Action']['SaleCount']
    #Access DynamoDB Salescount table to retrieve previous day's sales
    #and compare it to today's sales
    dynamodb = boto3.resource('dynamodb')
    client = dynamodb.Table('SalesCount')
    previoussalerecord=client.get_item(Key={'Store':storename})
    previoussalerecord_dict=previoussalerecord['Item']
    previoussalecount=previoussalerecord_dict['Sales']
    try:

        if (float(salecount) < float(previoussalecount)):
            performance='negative'
        else:
            performance='positive'
        response=client.update_item(
            Key={
                'Store': storename
            },
            UpdateExpression="set Sales = :s, Performance= :p",
            ExpressionAttributeValues={
                ':s': salecount,
                ':p': state,
            },
            ReturnValues="UPDATED_NEW"
        )
    except  Exception as e:
        print(str(e))
    return {
        'statusCode': 200,
        'body': json.dumps(performance)
    }

def generate_s3_report():
    #Retrieve sales performance from the DynamoDB table and generate a json report
    #and place it in the S3 bucket
    s3bucket = 's3-bucket'
    s3 = boto3.client('s3')
    reportfile='sales-report.json'
    dynamodb = boto3.resource('dynamodb')
    client = dynamodb.Table('SalesCount')
    try:
        priceslist=client.scan()
        priceslist_dict=priceslist['Items']
        s3.put_object(Body=json.dumps(priceslist_dict),Bucket=s3bucket, Key=reportfile)
    except  Exception as e:
        print(str(e))
    return {
        'statusCode': 200,
    }
```

Figure 3: A Sample Serverless Function

By examining the use case and its implementation, a few observations can be made:

1. The serverless model enables us to focus solely on the application, leaving the administration of the infrastructure and platform layers to the service provider.

2. As serverless functions are stateless, our function must save its state between executions. Hence, *DynamoDB* tables are utilized to save item prices and sales records.

3. The timeout value we specified for our function is 3 seconds, after which the function is terminated. This timeout value is appropriate for our use case. However, the timeout value must be increased in other use cases with large files

or more extensive computations. AWS supports timeouts values for up to 15 minutes.

4. A fresh execution of the "*generate_s3_report()*" function, where the S3 bucket and DynamoDB tables are accessed, takes 1.9 seconds despite being a simple computation. This latency results from a cold-start of the execution environment and the access to non-local storage (DynamoDB and S3).

5. We were required to adopt AWS SDK (*Boto3*[17]) in our code to access the AWS ecosystem (S3 and DynamoDB), thus introducing the potential shortcoming of vendor lock-ins.

2.6 Serverless computing research directions

Research in serverless computing is relatively new; however, it is gaining increasing attention in academia [12]. Some researchers have investigated the trends and open problems in serverless computing [8], [11]. Jonas et al. [11] examined the limitations of the current serverless platforms and proposed a vision of what serverless computing should become. The researchers specified a pathway defined by a set of challenges to achieve that vision. Baldini et al. [8] presented a survey of serverless platforms, along with their distinguishing characteristics. The researchers illustrated a generic architecture of serverless computing platforms while highlighting the role of an event processing system in such architecture. Their view on the current technology and research challenges facing serverless computing was also discussed in their work.

Sadaqat et al. [19] investigated the core concepts of serverless computing and the adoption benefits and challenges. The researchers presented their view on the future adoption, the market growth, and the vendors of serverless computing. Buyya et al. [20] provided a manifesto identifying the state of research in cloud computing and their vision of Future Generation Cloud Computing. Realizing this vision, the researchers proposed a roadmap

[17] AWS SDK: https://aws.amazon.com/sdk-for-python/

through a set of open challenges that need to be addressed. Economics of cloud computing was identified as one of the key challenges, highlighting the growing interest in computing models that enable sub-second billing.

During our review, we were able to classify the current research into three main directions:

- Serverless platforms research
- Serverless applications research
- Serverless function composition and orchestration

2.6.1 Serverless Platforms Research

This research direction investigates serverless platforms' architecture, service offerings, performance, security, and limitations. Lynn et al. [21] surveyed seven commercial serverless platforms and compared their feature-set. Mohanty et al. [18] investigated the architecture and performance of four open-source serverless platforms and evaluated functions' invocation concurrency in these platforms by measuring the response times and the request processing success ratio. Also, the researchers compared the auto-scaling metrics and the performance of these platforms. Hendrickson et al. [13] discussed Amazon's microservice models, the Lambda model, and presented the OpenLambda platform as an open-source platform to accelerate research on Lambda architectures.

Lee et al. [22] studied concurrent functions' invocations in commercial serverless platforms and compared the performance and the throughput results for the use case of distributed data processing. The researchers also provided a comprehensive review of the platforms' feature-set. Oakes et al. [23] analyzed the cold startup times for Python serverless functions, focusing on low-latency invocations. The researchers analyzed the runtime initialization and containers performance and proposed a special-purpose container system, replacing Docker, to improve latency for Python serverless functions. The specialized container system imports libraries and achieve high steady-state throughput for the runtime environments.

Lloyd et al. [24] investigated the factors affecting serverless functions performance in AWS Lambda and Azure platforms. The researchers identified five factors to evaluate in their study: elasticity, load-balancing, provisioning variation, infrastructure retention, and memory reservation; and examined the platforms for four different states of request serving: provider-cold, VM-cold, container-cold and warm. These states differ in the components of the serverless platform that need to be freshly executed in response to a serverless function's request. They highlighted the role of initialization time in cold start in profoundly impacting the elasticity of deployments in serverless platforms.

Jackson et al. [25] investigated the impact of the language runtime choice on the cost and performance of serverless functions. The paper proposed a Serverless Performance Framework (SPF) that enables consistent metrics gathering from serverless platforms. The researchers concluded that existing serverless platforms are not high-level language agnostic and that the choice of a function's runtime language affects the performance and the cost of its execution. Kim et al. [26] examined resource allocation in serverless platforms to maximize resource utilization and minimize SLA violations. The paper proposed a QoS-aware technique for CPU management in serverless platforms as an alternative to traditional consolidation approaches.

Hypervisor and container runtime performance have also received attention from researchers and industry, especially in the areas of lightweight emulation and security. As the runtimes of the current hypervisors and containers are designed for generic workloads, they exhibit performance and security limitations to sophisticated workloads. Thus, purpose-specific hypervisors and container runtimes emerged to address this shortcoming. Nemu[18] was developed a cloud-specific hypervisor that focuses on improving the emulation performance and reducing the attack surface. Kata containers[19], similarly, address improving the container performance, isolation, and security. Firecracker[20] leverages micro-VMs to enable the isolation and performance improvement of virtualized workloads.

[18] Nemu: https://github.com/intel/nemu

[19] Kata Containers: https://katacontainers.io/

[20] Firecracker: https://firecracker-microvm.github.io/

2.6.2 Serverless Application Research

As serverless computing presents a new programming model, many researchers investigated how existing complex applications can benefit from this new model. One can understand that the mandated stateless nature of serverless functions requires a mindset shift for programmers familiar with traditional programming models. However, it is essential to realize that, as a technological manifest, programmers need enough understanding of the other aspects of serverless computing abstraction to efficiently and securely utilize it.

Jangda et al. [27] pointed to the exposure of low-level operational details as a significant challenge to programmers developing code for serverless platforms. Programmers are required to be aware of how the serverless platforms operate to avoid producing incorrect results, leaking confidential data, or even data loss. The paper points out that "serverless functions are not functions in any typical sense" [27]. To address this gap, the researchers developed a detailed operational semantics of serverless computing. The semantics are designed to allow extensions for key-value store models and serverless function orchestration. With those contributions, the researchers hope that the developed semantics provide formal foundations to bridge the knowledge gap in the adoption of serverless computing by programmers. Similarly, Gabbrielli et al. [28] highlighted the lack of a formal model for serverless computing and proposed a minimalistic model to reason on the new paradigm.

Implementing traditionally stateful applications in serverless platforms has received attention from many researchers. Hong et al. [29] examined the implementation of scalable security services using serverless platforms. The researchers identified six design patterns for serverless architectures and discussed their advantages compared to serverful implementations. Limitations of the existing serverless platforms were also identified, such as resource constraints, event tracking, and security limitations. Jonas et al. [11] implemented five diverse research projects as serverless applications and presented a cost-performance analysis of the serverless implementations versus their serverful counterparts. They concluded that an application's use-case profoundly impacts the cost-performance gains or losses of a serverless implementation. For example, the researchers discovered that a serverless implementation of SQLite database engine incurs an additional 200% more cost per transaction than a serverful application. In comparison, implementation of a serverless MapReduce sorting incurs only 15% extra cost.

Hafeez et al. [30] investigated implementing the well-known publisher/subscriber model using stateless functions in AWS Lambda and Azure functions. The serverless implementation leveraged the decomposition of the traditional broker role into a set of serverless functions that perform subscriber registration and deregistration, creating subscriptions, and submitting publications. The researchers presented a demo utilizing AWS DynamoDB and Azure Table Storage for storage, Amazon Simple Queue Service (SQS), and Azure Service bus to implement messaging queues.

Solving domain-specific problems has also been examined in the literature. Perez et al. [31] proposed a programming model for high throughput file processing serverless applications. The model brings performance advantages to medical image analysis and video analysis applications and relies on customized serverless architectures and containers as the execution environment. Rapid RNA sequencing in serverless platforms was investigated by Hung et al.[32], while Aytekin et al. [33] examined the use of serverless workers for solving optimization problems. The researchers observed performance enhancement in solving regularized logic regression problems. Hussain et al. [34] examined serverless functions for edge real-time sensor data computations at remote sites in the oil and gas industry.

Due to its stateless nature, a serverless function cannot perform adequately in services that rely on in-memory buffers, such as database buffer pools. To improve performance, a user submitting subsequent queries to a serverless database engine must ensure that the same worker serves their requests. Thus, the literature has explored the concept of addressable stateful serverless functions through modifications of existing serverless platforms. Smith [35] proposed utilizing key-based addressable serverless functions to implement a serverless database. The researcher proposed Partitioned Function as a Service (pFaaS) to solve functions' addressing in serverless platforms. A key-based partitioning of serverless platforms enables directing requests to individual serverless workers and thus retaining the applications' state between subsequent function invocations.

2.6.3 Function Composition and Orchestration

Implementing a single serverless function is relatively straightforward; however, complex serverless applications require sophisticated workflows and coordinating input and output exchange between many serverless functions. As with traditional programming models, building modern applications as a coordinated set of functions rather than a monolithic application is often desirable. Thus, serverless functions' composition and orchestration are vital enablers to implement workflows and coordinate input and output exchange for complex serverless applications composed of many serverless functions.

Being crucial for the success and commercialization of a serverless platform, commercial serverless platforms started providing services to enable workflow implementation within their platforms' ecosystem. The orchestration services are offered as stand-alone services that integrate with the platforms' ecosystems. Lopez et al. [15] compared the commercial FaaS composition and orchestration systems offered by Amazon's Step Functions[21], IBM Composer[22], and Azure Durable functions[23] and evaluated them against a set of metrics. The researchers concluded that current orchestration systems vary in maturity, programmability, and simplicity of adoption.

Also, engineering function composition and orchestration services as serverless functions themselves have been investigated by researchers. Researchers at IBM [16] defined three competing constraints for engineering function composition as serverless applications, coining the term "serverless trilemma". In addition, the researchers examined the classes of function composition for the core programming model of Apache OpenWhisk and proposed a framework extension to implement sequential function composition.

[21] AWS Step Functions: https://aws.amazon.com/step-functions/

[22] IBM Composer: https://github.com/ibm-functions/composer

[23] Azure Durable Functions: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations

2.7 Open Problems and Our Research Direction

As an emerging field, the current state of serverless computing imposes many challenges for wide-scale adoption. Several researchers have investigated the limitations and the open problems and provided classifications and potential solution roadmaps. Jonas et al. [11] presented a futuristic view on what serverless computing should become, specifying five classifications of challenges that serverless computing must overcome to reach this vision. They categorized the challenges into abstraction, system, networking, security, and computer architecture challenges. Baldini et al. [8] similarly identified system-level challenges and programming model and DevOps challenges as the two categories of challenges facing current serverless platforms. Their paper also identified some open research problems as unexplored research areas in serverless computing.

In existing serverless platforms, a user can specify the memory allocated to their function. However, other resources are determined by the platform. This, in turn, has introduced resource requirement challenges with existing serverless platforms at the abstraction level[11]. Although it is well understood that the delivery model of serverless computing aims to abstract the cloud resources and thus enable users to focus only on the code, it is imperative that the serverless functions can leverage the resources needed for their code to execute efficiently. Besides, at its current state, serverless functions data dependencies and communication patterns are hidden from the serverless provider. This can lead to suboptimal placement of serverless functions and thus affect their performance. To resolve that, serverless functions must communicate their data dependencies and communication patterns to the serverless platform.

In addition to abstraction level challenges, system, networking, and security challenges facing serverless computing can be summarized as follows:

1. Serverless platforms must provide ephemeral and persistent storage for serverless functions.
2. The platforms must provide means for serverless functions to coordinate and signal one another.

3. Startup times for serverless functions must be minimized, considering the different tasks executed at a function's startup.

4. Serverless platforms must provide means to define and assess compliance with Service Level Agreements (SLA) for serverless functions. Users must be able to verify whether the platforms are abiding with their posted SLAs or not.

5. As applications are decomposed into many serverless functions, serverless platforms should expect and manage significant communication overheads on broadcast, aggregation, and shuffle.

6. As serverless functions from different tenants are expected to cohabit on the same physical host, co-residency attacks need to be investigated and prevented by serverless platforms such as side-channel or rowhammer attacks. Also, platforms need to consider scheduling randomization and physical isolation as part of their security policies.

7. Platforms need to provide granularity in the specification of security policies for serverless functions

8. As serverless functions are ephemeral, their network transmissions do not occur in bulk. Platforms need to provide mechanisms to avoid leaking access patterns and timing information to an attacker monitoring the serverless functions' communication.

The investigation of these research directions is essential for enabling the adoption of the serverless model by the use-cases it was designed to serve. The other cloud delivery models very poorly serve these use-cases, i.e., ephemeral, short-lived workloads. As discussed in section 1.4, the assumption of the workload's infinite lifetime and state persistence of the IaaS and PaaS models can significantly impact the scaling of ephemeral workloads and the overall platform resource utilization efficiency.

Currently, users with ephemeral, short-lived workloads are forced to adopt custom ad-hoc mechanisms for scaling their applications in IaaS or PaaS clouds. Similarly, cloud providers incur significant resource utilization inefficiencies when users deploy ephemeral, short-lived workloads to their cloud platform. To shed light on how this issue is predominant in IaaS clouds, we dedicate the next chapter to examining the usage trends of IaaS clouds. We conduct a pilot study to study the nature of workloads in IaaS clouds and their impact on the resource utilization efficiency of the platforms. The results illustrate a high presence of ephemeral, short-lived workloads in our examined dataset and a significant impact on the resource utilization efficiency of the examined IaaS platforms. The results illustrate the necessity of investigating the directions mentioned above to ensure the maturity of the serverless model for adoption by these workloads.

Chapter 3
IaaS Cloud usage behaviors: a pilot study

This Chapter includes sections from the papers:

- M.Elsakhawy and M.Bauer, "Usage Trends Aware VM Placement in Academic Research Computing Clouds" published in the IEEE International Conference on Cloud Computing (CLOUD '21)

- M.Elsakhawy and M.Bauer, "An Investigation into the Usage-trends of Canada's Research Computing Clouds", published in the IEEE International Conference on Smart Cloud (SmartCloud '19)

3.1 Introduction

As discussed in the previous chapter, IaaS clouds are not well suited to host ephemeral stateless workloads due to the stateful nature of these clouds. The IaaS model's shortcoming results in significant disadvantages for end-users utilizing IaaS clouds to host these kinds of workloads. We perform this pilot study to highlight the importance of developing the serverless model and the necessity of investigating and addressing its limitations, amongst which is the lack of SLA definition and assessment mechanisms. The study sheds light on the shortcomings of hosting ephemeral stateless workloads in IaaS Clouds and how the serverless model could easily act as a superior alternative to hosting these kinds of workloads.

We began our study by exploring the space of publicly published IaaS Clouds datasets. For the dataset to be representative of public IaaS clouds, our criteria for the dataset was i) to contain records representing VM creation and termination behaviors by the cloud users for a sufficiently long duration and ii) the cloud should be serving a large number of users such that the hosted workload categories are sufficiently a diverse sample. The records must specify the VM's allocated resources and their lifespan. To our surprise, we discovered that the landscape of publicly published datasets for IaaS clouds is non-existent. The dataset[24] closest to satisfying our criteria was published by Google's Cluster platform in 2013. A more recent dataset[25] (v3) was published in 2019. Both datasets, however, are not IaaS cloud-specific but are instead computational cluster job records. Thus, these datasets are not helpful in our investigation.

In this absence of public IaaS cloud datasets, we restored to exploring datasets for domain-specific IaaS Clouds, whose data can be shared by the operating institutions. We obtained a dataset representing four public research-computing IaaS clouds in Canada hosted and operated by the Compute Canada Federation[26]. While the clouds are utilized mainly in the research-computing domain, thus affecting the generalizability of our findings, the datasets provided a step forward in understanding the users' behavior in IaaS platforms.

Research computing is a technology sector that provides computing services for research needs. Traditionally, this sector was dominated by shared-access high-performance computing (HPC) clusters with hundreds of thousands of cores and low latency storage. To adjust to a growing demand for isolation and customization, research computing centers worldwide have expanded their offerings to include cloud platforms. IaaS remains a popular delivery model in research computing clouds, providing compute, storage, and network virtualization at its minimalistic offering. Like their commercial counterparts, research computing IaaS providers employ VM placement and reallocation algorithms to minimize energy consumption, maximize resource utilization, and achieve other objectives. Research institutions deploy research-computing IaaS clouds to provide computing capacity for researchers. These platforms generally have a limited user base that gradually grows as more researchers adopt the platform. Users are granted resource quotas for their research projects [36]–[39], and they utilize these quotas to provision the VMs required to support their computations.

[24] Google Cluster dataset: https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md

[25] Google's cluster traces: https://research.google/tools/datasets/google-cluster-workload-traces-2019/

[26] Compute Canada Federation: https://www.computecanada.ca/home/

There are limited literature investigations into the nature of the use-cases of these clouds. A study by Blatecky et al. [40] pointed to three categories of applications in academic research computing clouds that are either cloud-ready or actively running in the Cloud: i) Pleasingly Parallel, i.e., independent highly parallel, workloads; 2) Web portals and Gateways, and 3) SaaS applications. Studies by Taylor et al. [41], [42], Panitkin et al. [43], and Megino et al. [44] have examined the ongoing wide-scale adoption of research computing clouds in the ATLAS experiment [45] by CERN. Posey et al. [46] proposed a management tool to facilitate the transitioning of scientific parallel workloads to virtual clusters in the Cloud. Compute Canada lists specialized virtual clusters and web portals [47] as use cases for its research computing clouds. Toor et al. [48] listed bioinformatic data analysis and portals and scalable processing of ozone data as two representative use-cases to the workloads in the Swedish SSC cloud platform.

While providing insight into the use-cases of the platforms, the studies mostly relied on self-reporting of the IaaS providers or a direct engagement with the platform users. In this work, we formalize the process of understanding the usage trends by following a data analytical approach. We define usage-trends as a broad set of usage metrics that capture aspects of an IaaS platform usage such as resource utilization, VM attributes, user behavior, etc. We utilize a cloud's VM creation and termination records to generate these usage-trends and use these trends to deduce the ephemeral/persistent nature of the use cases of this cloud.

The dataset we obtained consists of VM records for four of Canada's public research computing IaaS clouds deployed between 2014 and 2017. The four clouds are based on the x86 architecture and utilize Openstack as the IaaS framework. Collectively, the platforms offered around 13,000 physical CPU cores on nearly 500 PMs. The dataset contained VM creation and termination records of approximately 1 million VMs provisioned by end-users over four and half years. Users requested access to the platforms leveraging a national resource allocation process and utilized VMs to host a broad range of computational workloads. The defining characteristics of these clouds can be summarized as:

- *IaaS quota-based clouds*: The clouds we examined are quota-based IaaS clouds, where users are allocated a limited quota of vCPUs, RAM, and storage. The cloud platforms utilize “VM Flavors” to define a combination of vCPU count, RAM size, and storage size that a VM can use during its lifetime. users create VMs by selecting one of the available flavors. When a user reaches their quota, they cannot create additional VMs. The resources allocated to a VM are released from the quota when the VM is deleted. Hence, both powered-on and powered-off VMs count towards a user’s quota.
- *Workload characteristics*: Workloads in the clouds utilize VMs for isolation and are user-driven. The workloads' start and end times are non-deterministic and controlled by the user.
- *Cloud access*: Users leverage a cloud portal and a set of API endpoints to provision resources in the clouds.
- *Storage*: CEPH[27] is used as the storage solution. Users are allocated a predefined storage quota to attach to their provisioned VMs.

Openstack[28] stores the cloud’s operational data in a SQL-based backend, configurable by the cloud operator. The VM’s metadata, such as creation and deletion timestamps, is recorded in the dedicated *Instances* table in the Nova[29] database. The dataset comprised exported records of the instances table of the four clouds of approximately 1 million VMs that were created over four and a half years. We eliminated records of VMs’ provisioning failures and duplicated records from the set. The resulting set included records for nearly 983 thousand VMs, with fields specifying their creation and deletion timestamps, creator identifier, and allocated resources. The following section formalizes the main concepts to examine usage-trends in IaaS clouds.

[27] CEPH: https://ceph.io/

[28] Openstack: https://www.openstack.org/

[29] Nova: https://docs.openstack.org/nova/

3,.2 Formulation

For an IaaS cloud S, at a time instant T, U is a set of all cloud users with access to create VMs on S since the inception of S to time instant T. V is a set of all VMs created by U from the inception time of S to time instant T.

$$U = \{u_1, u_2, u_3 \ldots., u_j\} \text{ where } j \in \{1,2, \ldots m\}$$

$$m \text{ is the total number of users from inception}$$

$$\text{to time instant } T$$

$$V = \{v_1, v_2, v_3 \ldots., v_i\} \text{ where } i \in \{1,2, \ldots n\}$$

$$n \text{ is the total number of VMs provisioned in } S \text{ from inception}$$

$$\text{to time instant } T$$

The function that defines the ownership of a VM is $O(v_j)$

$$\forall\, v_j \in V: O(v_j) = u_j \quad \text{where } u_j \text{ is the creator of } VM\ v_j$$

For each user u_j, the set V_j is a subset of V containing only the VMs owned by the user u_j , where

$$\forall\, u_j \in U: V_j \subseteq V$$

$\text{such that } u_j \text{ is the owner of all the VMs in subset } V_j$

VM lifetime L is the duration, in seconds, that the cloud platform S reserves resources for a VM, regardless of those resources being actively utilized or not.

$$\forall\, v_i \in V: L(v_i) = T_d(v_i) - T_c(v_i)$$

Where $L(v_i)$ is the lifetime of VM v_i, $T_d(v_i)$ is the deletion timestamp of VM v_i and $T_c(v_i)$ is the creation timestamp of VM v_i. Based on their lifetime, VMs can be classified into *Ephemeral* and *Persistent* VMs. *Ephemeral* VMs have a short lifetime and host workloads of temporary nature such as short computations or experimental environments. *Persistent* VMs, on the other hand, exhibit a long lifetime and are used for computational workloads of stateful nature, such as web-portals and database engines.

VM creation gap G is the time difference in seconds between two successive VM creations by the same user. *VM creation gap* is an indicator of the frequency of VMs' provisioning by a user. A very short VM creation gap may indicate the rapid creation of VMs by a user to cover the peak load of an application.

$$\forall u_j \in U, \forall v_i \in V_j:$$
$$G(v_i, u_j) = T_c(v_i) - T_c(v_{i-1}) \text{ } where \text{ } v_{i-1} \in V_j$$

Where $G(v_i, u_j)$ is the VM creation gap of VM v_i created by user $u_j$and $T_c(v_i)$ is the creation time of VM v_i and $T_c(v_{i-1})$ is the creation time of VM v_{i-1}.

VM resource allocations R defines the number of virtual CPU cores (vCPU) and the memory size assigned to a VM.

$$\forall v_i \in V: R(v_i) = \{P_i, M_i\}$$

$$where \text{ } P_i is \text{ } the \text{ } number \text{ } of \text{ } allocated \text{ } vCPUs$$

$$and \text{ } M_i is \text{ } the \text{ } size \text{ } of \text{ } allocated \text{ } memory$$

We grouped the examined VMs' lifetimes into three intervals that reflect the degree of persistence of the computational workload. A VM's workload degree of persistence increases with the increase in its lifetime. We define *Highly-Ephemeral VMs* as having a lifetime of less than 1 hour, while *Moderately-Ephemeral VMs* have a lifetime of between 1 hour and one day. *Persistent* VMs have a lifetime of more than one day. The percentiles of VMs' lifetimes in each range are shown in Table 1 and graphically illustrated in Figure 4.

Table 1: VM lifetimes

VM Lifetime	Percentiles
1 hour or less	45.5%
Between 1 hour and a day	52.7%
More than one day	1.9%

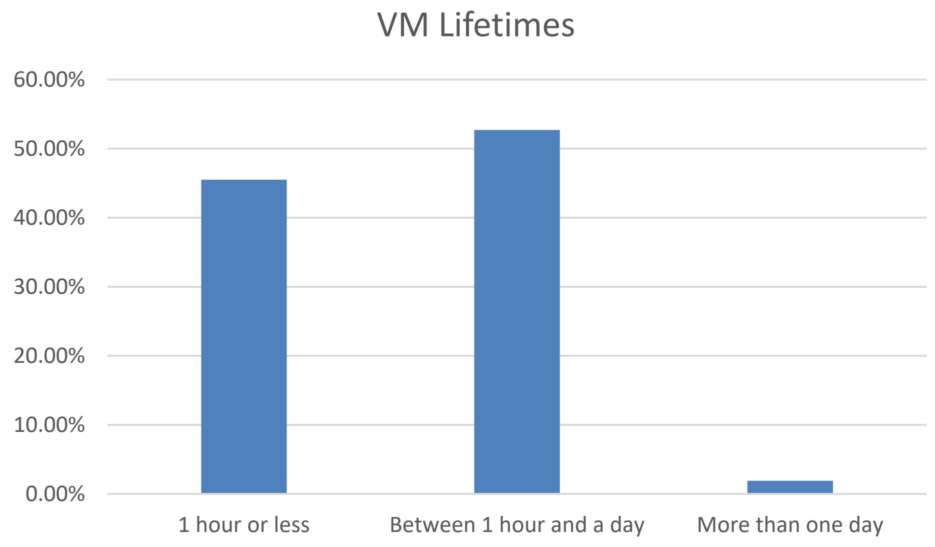


Figure 4: VM Lifetimes

By examining the *VM lifetime* data, it is evident that *Highly-Ephemeral VMs* are of a dominant presence, with 45.5% of the VMs being deleted within 1 hour of creation. In contrast, *Persistent VMs*, with a lifetime of more than one day, accounted for only 1.9% of the surveyed VMs.

Similarly, we grouped the *VMs' creation gap* into three ranges to reflect the frequency by which users provision VMs. *High-rate VMs' provisioning* occurs when VMs are provisioned within 1 minute of one another. This process potentially leverages the API end-points and external schedulers or scripts to provision VMs. *Moderate-rate VMs' provisioning* occurs when VMs are created with a creation gap between 1 minute and one hour. Finally, *Low-rate VMs' provisioning* occurs when VMs are created within more than 1 hour of one another. The percentiles of VMs' creation gaps are shown in Table 2 and graphically illustrated in Figure 5.

Table 2: VM Creation Gaps

VM Creation Gap	Percentiles
1 minute or less	63%
Between 1 minute and 1 hour	34.9%
More than one hour	2.1%

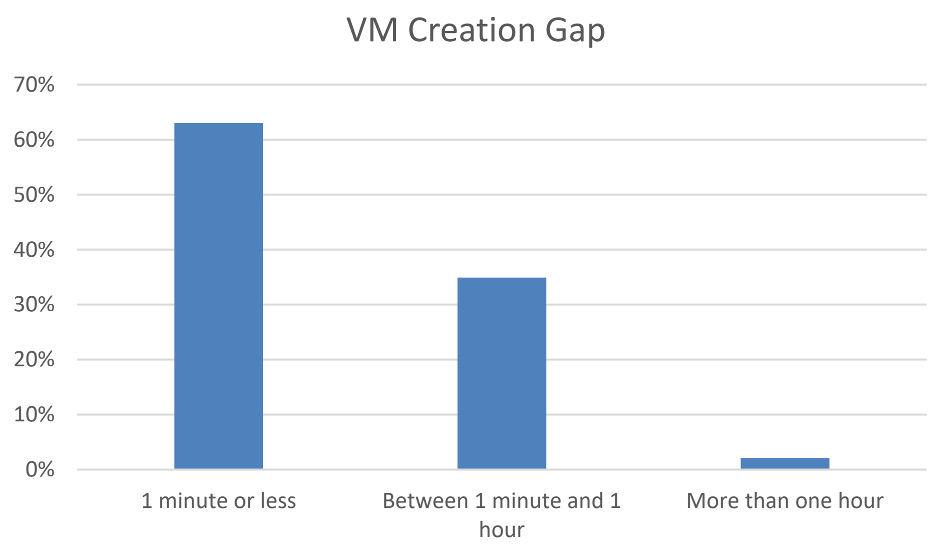


Figure 5: VM Creation Gaps

By examining the *VM creation-gap*, it is evident that *High-rate and Moderate-rate VM provisioning* are of a dominant presence, while *Low-rate VM provisioning* has a minimal presence.

Lastly, we analyzed the *VMs' resource allocations* (Allocated Disk size, RAM size, and vCPU count) in the four clouds and grouped them into ranges starting with memory allocation from 2 Gigabytes to 32 Gigabytes and CPU allocation from 2 vCPUs to 32 vCPUs. The percentiles of every range are shown in Tables 3, 4, and 5.

Table 3: Allocated Disk Distributions

VM Allocated Disk	Percentiles
20 GB or less	86%
More than 20 GB	14%

Table 4: Allocated vCPU Distributions

VM Allocated vCPUs	Percentiles
2 vCPUs or less	22.1%
3 or 4 vCPUs	2.4%
5 to 8 vCPUs	68.2%
9 to 16 vCPUs	7.15%
17 to 32 vCPUs	0.01%
More than 32 vCPUs	0.03%

Table 5: Allocated Memory Distributions

VM Allocated Memory	Percentiles
2GB or less	7.8%
3 or 4 GB	7.7%
5 to 8 GB	7.9%
9 to 16 GB	16.1%
17 to 32 GB	65.4%
More than 32 GB	8.9%

The distributions of the VM lifetimes and VM creation gap metrics show a dominant presence of rapidly-created, short-lived VMs in the four clouds. 45.5% of the examined VMs have a lifetime of one hour or less, while 63% of the VMs have a creation gap of one minute or less. A large number of VMs appear to be provisioned to execute ephemeral computations and are deleted once the computation is complete. While feasible in IaaS clouds, this use-case is highly inefficient from a resource allocation perspective.

As mentioned earlier, IaaS platforms use placement algorithms biased towards workload persistence and do not account for the ephemeral nature of the workloads identified in our dataset examination. To examine the effects of placing these ephemeral workloads on the resource allocation efficiency of the IaaS cloud platforms, we examined the resource utilization of the PMs in the provider's server pool throughout four years of VM placement (Jan 2015 to Jan 2019). Our examination focused on identifying the underutilization of PMs, i.e., when the placed VMs consume 50% or less of their host PM's resource capacity.

For a cloud provider C with a PMs pool P

$$P = \{p_1, p_2, p_3, \dots, p_i\}\ where\ 1 \le i \le m$$

where m is the total number of PMs in P

A PM $p_i \in P$is modeled as a set of resources

$$p_i = \{C_i, M_i, D_i\}$$

Where $C_i$is the number of CPU cores for PM $p_i$and M_i, and D_i are the memory and the disk capacity of PM p_i.

PM p_i can occupy one of two states at a time instant t : i) *active* (powered-on) state and ii) *inactive* (powered-off) state. The Activeness function A is formulated as

$$A(p_i, t) = \begin{cases} 1\ if\ p_i\ is\ active\ at\ time\ instant\ t \\ 0 \qquad\qquad\qquad\qquad\qquad otherwise \end{cases}$$

To minimize the scope of our investigation, we chose to focus on CPU utilization as a single dimension of the PMs' resource utilization. We define the CPU utilization function U of a PM p_i at time instant t as the aggregate count of CPU cores allocated to VMs on p_i at time instant t

$$U(p_i, t) = \sum_1^n cores(v_j) \quad where\ 1 \le j\ \le n$$

$$where\ n\ is\ the\ number\ of\ VMs\ hosted\ on\ p_i\ at\ time\ t$$

A PM p_i with CPU capacity C_i cores, p_i is underutilized at time instant *t* if the aggregate CPU cores allocated to VMs on p_i at time instant *t* is less than a predefined ratio, i.e. C_i/k where k is a positive integer. In this work, we choose $k\ = 2$, where a PM is underutilized if its instantaneous aggregated CPU utilization is less than 50% of its CPU cores capacity, i.e., $<\ C_i/2$. The underutilization function *R* is formulated as:

$$R(p_i, t) = \begin{cases} 1 & if\ U(p_i, t) <\ C_i/2 \\ 0 & otherwise \end{cases}$$

The powered-on (active) time of PM p_i is denoted as $T_{on}(p_i)$, which is equal to the total number of seconds where p_i is powered-on. $T_{on}(p_i)$ can be divided into a set of equal intervals. i.e.

$$T_{on}(p_i) = \{T_1, T_2, \dots T_m\}$$

$$\text{where } \sum_{1}^{m} T_q = T_{on}(p_i).$$

For p_i to be considered underutilized in the interval T_q, it must satisfy the underutilization function $R(p_i, t)$for a minimum of $T_q/2$ seconds in T_q :

$$If \ \sum_{t=0}^{T_q} R(p_i, t) \geq T_q / 2$$
$$PM\ p_i\ is\ considered\ underutilized\ in\ T_q$$

We specify $T_q = 3600\ seconds$, i.e., one hour, to examine the hourly PM underutilization in our dataset. We computed the total hourly utilization per PM and flagged PMs whose resource utilization did not exceed 50% for 1800 seconds (i.e., half-hour) and labeled these PMs as underutilized. The percentage of underutilized PMs to the total number of PMs is plotted in Figure 6. The diagram shows the percentage on an hourly basis (i.e., every hour for the VM placement period of four years).

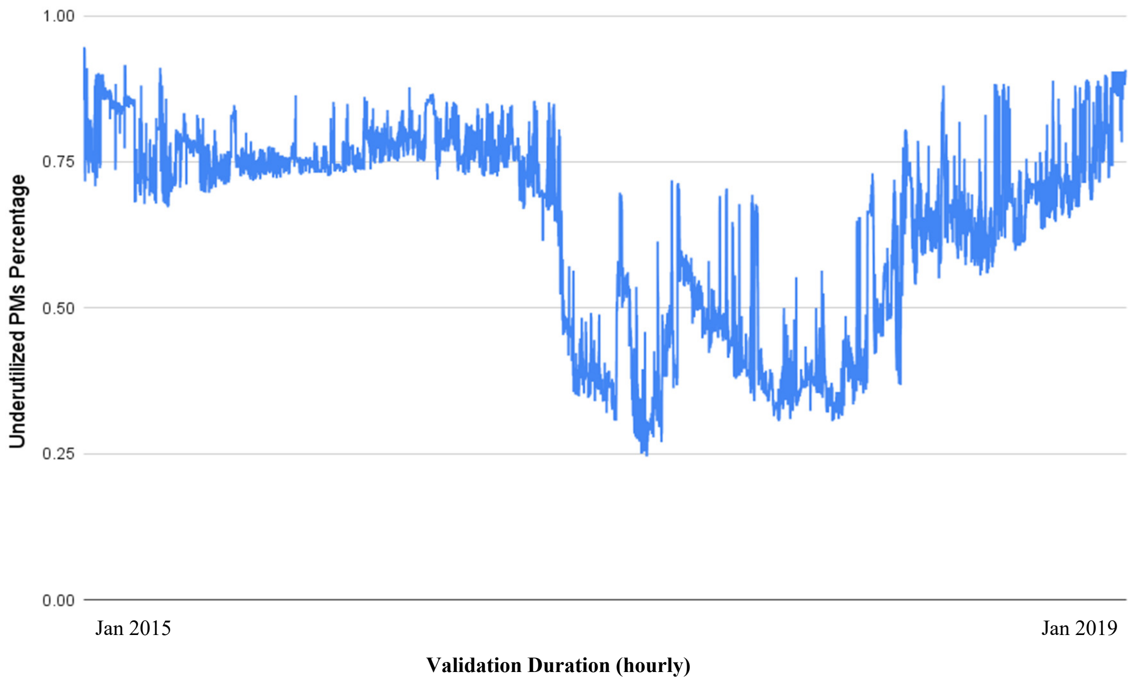


Figure 6: Underutilized PMs Percentage

Table 6: Underutilized PMs

Underutilized PMs	Percentiles
Min	24.7%
Max	94.4%

As shown in Fig. 6 and Table 6, a significant percentage of the PMs in the providers' server pool was underutilized throughout the examined period. The underutilized PMs percentage ranged from 24.7% at best to 94.5% at worst. At best, a quarter of the provider's PMs pool was underutilized throughout the four years period of our examination. These underutilized PMs continued to incur the same energy consumption as fully utilized PMs while their resources were idling. The results demonstrate the visible impact of placing ephemeral workloads, which are the majority of the workloads in our dataset, in IaaS clouds that assume a persistent nature of the workloads. The providers faced higher energy consumption and idling resources while end-users were forced to adopt ad-hoc mechanisms for scaling their ephemeral workloads through *high-rate VM provisioning*. While the providers could have employed dynamic consolidation techniques to reduce the PM underutilization, they would have had to activate such techniques very frequently (potentially hourly), causing massive migrations of many VMs, thus impacting their performance.

3.3 Results and Discussion

The results of our examination [36] highlight that a significant percentage of the workloads in the examined IaaS clouds are ephemeral. The majority of the provisioned VMs had 20GB or fewer storage allocations and a lifetime of one hour or less. Users appear to have adopted their ephemeral workloads to the persistent nature of the IaaS model by using external schedulers that rapidly dispatch VMs for computations and terminate the VMs once the computations are complete. End-users were responsible for coordinating state-sharing (i.e., data) between running VMs to avoid conflicts.

While such ad-hoc techniques were able to retrofit IaaS clouds to the ephemeral workloads, they incurred several disadvantages:

- *High-rate VM provisioning* using external schedulers requires the end-users to have the expertise to adopt these complex technologies. While some users may have sufficient technical knowledge to achieve this, most IaaS users do not.
- By default, resource allocation algorithms in IaaS clouds assume an indeterministic lifetime for provisioned VM. The short lifetime of ephemeral workloads mandates that dynamic consolidation techniques are employed very frequently to ensure that the provider's placement objectives continue to be met at runtime. In addition to the consolidation cost of such techniques that the end-users and the providers incur, this essentially invalidates the initial VM placement decisions for ephemeral workloads as these decisions are expected to change very frequently throughout the VMs lifetime.
- End-users incur unnecessary overheads to their computations, such as the VM's underlying operating system, hypervisor, and network virtualization overheads. These overheads are mandated by the IaaS model and can be reduced by leveraging minimalistic isolation technologies such as containers or isolated runtimes.
- While this was not directly visible in our examined data, features such as autoscaling and HA are left to the end-users to implement. These features also require technical expertise that many users lack.

Our examination results illustrate the importance of developing the serverless model to serve the ephemeral workloads poorly served in the other cloud delivery models. As discussed in Section 2.7, we chose to focus on investigating how performance guarantees can be established in the serverless model. Such guarantees are an existential necessity for both the financial feasibility of the pay-upon-execution model in serverless computing and for establishing the user faith in moving their workloads to serverless platforms. The next chapter discusses SLA in cloud computing and demonstrates how SLA in the serverless model fundamentally differs from earlier cloud computing models. We discuss the challenges facing the definition of FaaS SLA due to its unique abstraction levels. In Chapter 5, we illustrate how seemingly trivial end-user choices can significantly impact the performance of serverless functions.

Chapter 4
SLA in Cloud Computing

4.1 Introduction

Service level agreements (SLAs) specify the contractual obligations of cloud computing providers to their customers. Providers must abide by these agreed-upon SLAs or risk penalties for incompliance. Customers rely on a provider's SLA to guide their providers' choices and level-set their expectations of the cloud platform. While the contractual obligations vary between cloud providers, the SLA specifications generally utilize a standard set of measurable metrics based on the nature of the cloud delivery model. In the coming sections, we discuss SLA in the IaaS and PaaS delivery models and how it fundamentally differs from FaaS SLA.

4.2 IaaS and PaaS SLA

In IaaS and PaaS delivery models, SLA is generally defined in terms of *resource-guarantees and platform uptime*. In resource-guarantees, the provider promises the customer a predetermined set of computational resources with particular specifications and sizes. An SLA violation occurs if a customer's computational entity, i.e., a VM or a container, is not granted its promised resources when requested. IaaS and PaaS users can use various OS or container tools to validate the resources available to their VMs or containers. On the other hand, platform-uptime specifies the minimum duration for the VMs or containers to be available and operational. The provider guarantees that the user's entity (VM/container) is running and that the components in the cloud platform (e.g., networking and storage) are operational and healthy. The platform uptime is commonly assessed monthly, and providers are penalized for downtime durations exceeding that allowed by the SLA.

While resource-guarantees suited the IaaS and PaaS models, the FaaS model abstracts computational resources, making it impossible to define an SLA using resource-guarantees. In essence, FaaS exhibits the following unique characteristics that distinguish its SLA from earlier delivery models:

- FaaS abstracts the resource, ecosystem, and execution environment layers, requiring end-users only to submit code to be executed. With such abstraction, computational resources are hidden from end-users and beyond their control and assessment. Serverless platforms are responsible for selecting the resource allocations and the execution environment configurations required to execute a serverless function optimally.
- The FaaS model segregates computations from data, and thus serverless functions are highly dependent on the provider's ecosystem for data storage, access, and communication. This increases the complexity of assessing SLA violations since many components within the ecosystem contribute to the serverless function's performance.
- The execution duration of serverless functions is expected to vary based on non-SLA impacting factors, such as input payload sizes, the computational workflows, and the nature of the functions' computations. Hence, execution durations cannot be utilized solely to evaluate a function's performance.
- The performance of the resource, ecosystem, and execution environment layers that directly impact a function's execution-performance varies significantly between multiple serverless providers, as Jackson et al. [26] illustrated.

For these reasons, defining SLAs in serverless computing must accommodate the uniqueness of its abstraction level and performance attributes. In the next section, we investigate SLA in serverless computing and propose the notion of an *execution-SLA* as a means of defining execution-performance guarantees for serverless functions.

4.3 SLA in Serverless Computing

Defining execution-performance guarantees is crucial for serverless computing due to their direct financial implications. Users incur costs based on the execution durations of their functions and the consumed resources. Bottlenecks that adversely affect a function's execution-performance result in higher user costs and impact the financial attractiveness of the serverless model. Little work has been done to investigate SLA in serverless computing. Nguyen et al. [49] investigated invocation-rate guarantees, i.e., invocation-SLA, in serverless frameworks as a requirement for real-time applications to leverage serverless functions. The authors, however, did not address guarantees on a function's execution-performance.

While some attempts have been made to investigate serverless Quality of Service (QoS) and Service Level Objectives (SLOs), the works focused only on function scheduling, startup, and data-access latency [50]–[53]and invocation concurrency [12], [22], [49]. These objectives, while important, are only a subset of the performance guarantees for serverless functions. To provide a holistic SLA for FaaS, platforms must define guarantees on the factors that impact a function's performance from invocation to termination. These factors can be grouped as:

- Request servicing latency
- Resource allocation and provisioning latency
- Performance of the execution environment
- Performance of the platform's ecosystem
- Invocation concurrency

The mentioned factors are illustrated in Fig. 7, which follows a serverless function's lifetime (1-4) along with its concurrency (5). In the following sections, we clarify these factors and their role in a function's execution.

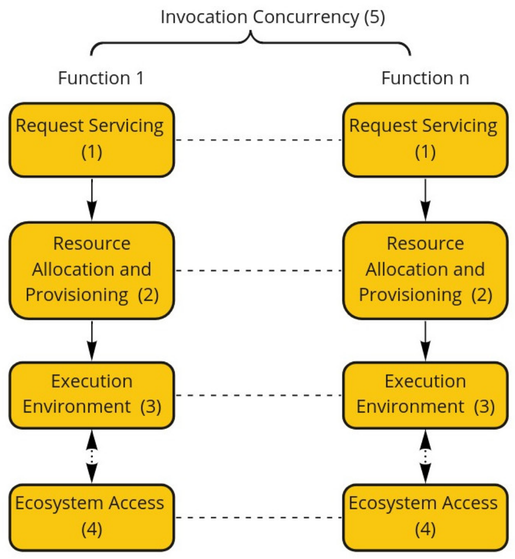

Invocation Concurrency (5)
Function 1
Function n
Request Servicing (1)
Resource Allocation and Provisioning (2)
Execution Environment (3)
Ecosystem Access (4)
Request Servicing (1)
Resource Allocation and Provisioning (2)
Execution Environment (3)
Ecosystem Access (4)

Figure 7: Aspects of FaaS execution performance

4.3.1 Request Servicing Latency.

Serverless functions are event-driven [8] and receive requests through a set of triggering events. These triggering events can be HTTP requests, object store put-events, and database row inserts. The request servicing latency is defined as the FaaS platform's time-consumed to detect an incoming request, acknowledge it, and redirect it to the component responsible for starting the execution procedure.

4.3.2 Resource Allocation and Provisioning Latency

After the request is serviced, the platform must allocate resources to the function's execution environment. A component within the FaaS platform utilizes a placement algorithm to select the Physical Machine (PM) or the Virtual Machine (VM) to allocate the resources on, based on the function's resource requirements, the PM's (or VM's) utilization, and the placement objectives. The time consumed by the algorithm to make the placement decision, start/repurpose an execution environment on the selected PM or VM, and transfer the function's code to the execution environment.

4.3.3 Performance of the Execution Environment

An execution environment provides an isolated process space and the interpreter required to execute the function's code. Docker and other container technologies are widely used in serverless platforms to implement process isolation. The execution environment's performance can be impacted by a broad range of factors such as the choice of containerization technology [23], [53], its configuration, interpreter optimizations, and runtime libraries.

4.3.4 Performance of the Platform's Ecosystem

Serverless functions are stateless microservices that rely on the provider's ecosystem for data storage, access, and communication. The standard ecosystem services, also referred to as Backend as a Service (BaaS) [54], include object storage, databases, and messaging queues. The performance of these services directly impacts a function's performance.

4.3.5 Invocation Concurrency

When the number of concurrent requests increases, the platform must scale up the number of running instances of the function [49]. Platforms must provide guarantees on the maximum number of concurrently running function instances and the rate by which they can scale up these instances.

4.4 The Challenge of FaaS SLA

The ability to measure the abovementioned factors is essential for defining performance guarantees in serverless computing. The request servicing and resource allocation latencies (Sections 4.3.1 and 4.3.2) can be measured by examining each responsible component's time consumed to complete its functionality. Similarly, the invocation concurrency (Section 4.3.5) can be measured by monitoring the number of concurrent execution environments and the rate of scaling up the number of instances [12], [22], [55].

In contrast, measuring the execution environments' performance and the ecosystem's performance (Sections 4.3.3 and 4.3.4) is not a simple task. Determining the metrics that reflect an execution environment's performance is complicated as the configuration, and optimization options for containers and interpreters are endless and vary significantly between different containerization technologies [56], [57], and programming languages interpreters [58]. Similarly, every service of the platform's ecosystem (object storage, messaging queue, etc.) has unique performance metrics specific to its particular functionality. Thus, it is crucial to dedicate efforts to identifying the metrics that can measure the performance of these two abstracted layers.

Classical approaches for defining performance do so in terms of execution durations of running applications. For serverless environments, however, we cannot rely on execution duration as a metric for performance measurement for the following reasons:

- The abstraction of execution environments and ecosystem services:
    - End users have limited insight and control on the choices made by the provider in the execution environments configuration, containerization technology, and the ecosystem services configurations and optimizations.

    Execution duration of a single function on the same serverless platform have shown to exhibit wide variability, as illustrated in [58]

- The dependency of execution durations on the particular function's workflow:
    - The execution duration of functions may depend on many factors, including the input payload type and size as well as the function's computational workflow. Thus executions durations can be impacted by any change in these factors and not only the performance of the execution environment and the ecosystem.

In the next chapter, we illustrate the performance variations of serverless execution environments (sections 5.3 and 5.4) by examining the impact of multiple environmental optimizations on the execution durations of serverless functions. We illustrate that seemingly trivial decisions can significantly affect the execution durations and consequently the associated expenses. We then demonstrate how minor ecosystem performance degradations (section 5.5) can similarly impact a function's execution duration. We later shift our focus in Chapter 6 to examine the specification metrics that can be leveraged to measure a function's execution performance.

Chapter 5
Serverless Execution Performance

This Chapter includes sections from the paper:

- M.Elsakhawy and M.Bauer, "Performance Analysis of Serverless Execution Environments" published in the 3rd International Conference on Electrical, Communication, and Computer Engineering (ICECCE'21)

5.1 Introduction

Serverless platforms bill users for the execution duration of their functions. Commercial platforms use increments of 100ms as a basis to calculate the incurred costs. The longer a user's function takes to execute, the higher costs the user incurs; thus, it is in a user's best interest for their function to execute in the least possible duration. While a user can control the optimization of their function's code, as mentioned in the previous chapter, a broader set of factors impacts the function's execution performance. This chapter examines the impact of execution environments and ecosystem degradations on the execution performance of serverless functions.

5.2 Serverless Execution Environments

Serverless functions are executed in isolated execution environments when a triggering event is satisfied [14]. General-purpose container technologies, such as Docker, are a popular choice by serverless platforms for their portability and wide-scale adoption [23]. These container technologies provide a broad spectrum of configuration choices for building serverless execution environments. Users can select from a growing number of container operating systems, high-level language interpreter versions, and runtime libraries to create a function's execution environment. Such diversity allows tailoring execution environments to satisfy the requirements for every unique serverless function. However, it also introduces the potential lack of uniformity between execution environments.

Commercial serverless platforms attempt to address such nonuniformity in their platforms by offering users a limited set of pre-built execution environments. The pre-built environments provide a limited number of language interpreter choices (e.g., Python, Java, etc.) and versions (Python 3.7, 3.8, etc.). Users can import custom execution environments if their functions require unique runtime libraries or sophisticated customizations. Users with non-unique requirements or with limited technical expertise are encouraged to utilize pre-built execution environments. The nonuniformity of execution environments also impacts opensource serverless platforms. Some platforms, such as Apache OpenWhisk, allow operators to define a set of pre-built execution environments for users to use. Knative users, on the other hand, generally build execution environments embodying their function's code.

As more users adopt the serverless model, it becomes essential to investigate and establish performance guidelines. Users assume that the serverless provider optimizes pre-built execution environments to achieve the serverless function's fastest execution. However, it has been illustrated by Jackson et al. [25] that significant performance variations exist between serverless providers and even within the same provider's execution environments. The authors compared execution durations and costs for a serverless function on AWS and Azure platforms using different interpreters and discovered significant variations in the function's execution durations on both platforms. They highlighted that the language interpreter's choice could significantly impact serverless functions' execution. Lee et al. [22] uncovered performance differences between commercial serverless platforms related to concurrency, I/O throughput, and elasticity. Tariq et al. [52] examined the current scheduling practices in serverless providers and uncovered issues related to inflated application runtimes, function drops, and inefficient allocations. Ginzburg et al. [59] explored performance variations in AWS Lambda and discovered temporal and spatial performance variations consistent enough to exploit. Martins et al. [55] also noted the performance variations in latency, throughput, and computation time between commercial providers. Hellerstein et al. [14] pointed that none of the major serverless providers offer transparent APIs to examine the service levels for their FaaS platform.

The suitability of container technologies has also been studied by Oakes et al. [23]. The authors highlighted that general-purpose containers imposed restrictions on low-latency invocations of functions and proposed a special-purpose container to improve Python serverless functions' latency. Tiwary et al. [53] proposed WebAssembly[30] (WASM) for improving serverless performance.

To investigate execution environments' performance, we first identify the components that form an execution environment as i) a *Skeleton Container*; ii) an *Execution Engine*, and iii) *Runtime libraries*:

- *A Skeleton Container*: provides an isolated namespace for computations, file system operations, and network I/O for the executed function. Lightweight Linux containers are popular skeleton containers for execution environments.
- *An Execution Engine*: provides a language-specific interpreter of the function's code. Python, NodeJS, and Java are some of the popular high-level languages for serverless functions
- *Runtime Libraries*: They are the libraries needed to execute the function's code. Numpy[31] , Scipy[32] , PIL[33] are popular Python libraries for linear algebra and image processing.

These three components' performance directly impacts an execution environment's performance and, consequently, a function's execution performance. We focus on investigating the performance impact of the choice of skeleton container and execution

[30] WASM: https://webassembly.org/

[31] Numpy: https://numpy.org/

[32] SciPy: https://www.scipy.org/

[33] Pillow: https://python-pillow.org/

engine versions and compiler-level optimizations. The rationale for our focus on these specific components is as follows:

- Skeleton containers are, in essence, Linux distributions. Their memory footprint, general optimizations, and default packages impact the execution environment's performance.
- Execution Engines are compiled binaries whose performance is influenced by i) compiler-level optimizations, i.e., *builds*, and ii) version differences, i.e., *releases*.

Our analysis explores the performance variations of skeleton containers, and execution engine builds and releases by focusing on the following two aspects:

- *Execution performance*: We evaluate the speed of executing a serverless function by an execution environment. To quantify that, we leverage the function's execution duration, which is the time consumed by the execution environment to complete one execution of the function's code.
- *Execution consistency*: We evaluate the variability of execution durations of the same execution environment. We measure the standard deviation of a function's execution durations and use that to assess the consistency of executions per execution environment

We divide our investigation is into two parts:

- Section 5.3 focuses on the impact of skeleton containers and execution engine builds on serverless functions' execution performance and consistency. We benchmark seven popular skeleton containers using multiple builds of the Python 3.8.1 execution engine.
- Section 5.4 focuses on the impact of execution engine releases on serverless functions' execution performance and consistency. We benchmark multiple

releases of AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions, pre-built Python execution environments (3.6, 3.7, and 3.8).

5.3 Impact of Skeleton Container and Execution Engine Builds

This section focuses on analyzing the impact of skeleton containers and execution engines' build on a function's execution performance. We benchmark seven popular skeleton containers using multiple builds of the Python execution engine. The choice of Python execution engine was due to the popularity of the Python programming language among software developers. We benchmarked the latest Python release available at the time of investigation (Python 3.8.1). We developed a benchmarking function that performs a sequence of matrix multiplications and additions utilizing the NumPy library. The function is intentionally designed to be input-less and output-less to avoid the interference of I/O latency with the collected statistics.

Four of the benchmarked skeleton containers are based on Alpine Linux[34], while the remaining three containers are general-purpose containers based on Debian[35], Ubuntu[36], and Amazon Linux[37]. The rationale for our choices of the skeleton container and execution engine compiler combinations is as follows:

- Alpine Linux is the default skeleton container for Apache Openwhisk execution environments. Additionally, Debian and Alpine Linux are the skeleton containers of choice for the official Python docker containers on Dockerhub[38] and are

[34] Alpine Linux: https://alpinelinux.org/

[35] Debian: https://www.debian.org/

[36] Ubuntu: https://ubuntu.com/

[37] Amazon Linux AMI: https://aws.amazon.com/amazon-linux-ami/

[38] Python Dockerhub: https://hub.docker.com/_/python

commonly used in the Knative framework. Both containers are widely utilized in opensource serverless frameworks to create execution environments

- Ubuntu is a popular general-purpose container. Users familiar with the Ubuntu OS may be inclined to use Ubuntu to create custom execution environments.
- Amazon Linux is the default skeleton container used in Amazon AWS Lambda, so it was essential to benchmark it as a reference in our performance analysis.
- We used the default versions of GCC and CLANG compilers offered by each skeleton container distribution's repositories. Users are likely to utilize the default compiler that ships with their skeleton container to compile the execution engine. We also used the default optimization options of every compiler, as users are likely to do the same.

The combinations of skeleton containers and compilers that we benchmarked are listed in Table 7.

Table 7: Benchmarked Execution Environments

Skeleton Container	Execution Engine Compiler
Alpine Linux 3.8	GCC 6.4
Alpine Linux 3.9	GCC 8.3
Alpine Linux 3.11	GCC 9.2
Alpine Linux 3.11	CLANG 9
Ubuntu 18.04	GCC 7.4
Amazon Linux	GCC 7.3.1
Debian Linux 10	GCC 8.30

We deployed the execution environments to a Knative cluster with one control node and one worker node. The two nodes' hardware setup is identical, each with 20 cores based on Intel(R) Xeon(R) CPU E5-2660 v3 processor and 128 GB of Memory. We performed 1000 invocations of our benchmarking function and recorded the execution durations. We used Knative directives to ensure execution environments are kept idle between executions to avoid cold-start latency of starting new execution environments. To avoid the interference of co-located applications on the performance of the execution environments, we ensured that the Knative worker node had no running processes during our experimentation. The resulting execution durations are shown in Table 8 and are plotted in Figure 8.

Skeleton Container	Min	Max	Mean	Std. Dev
Alpine Linux 3.8 - GCC 6.4	16.3	26.7	20.4	0.59
Alpine Linux 3.9 - GCC 8.3	15.9	27.3	21.5	0.45
Alpine Linux 3.11 - GCC 9.2	15.9	30.2	26.4	0.52
Alpine Linux 3.11 – CLANG 9	16	29.9	24.5	0.48
Ubuntu 18.04 - GCC 7.4	15.7	22.8	16	0.39
Amazon Linux - GCC 7.3.1	15.7	21.5	16	0.46
Debian Linux 10 - GCC 8.3.0	19	25.8	19.47	0.42

Table 8: Execution Duration (in Seconds)

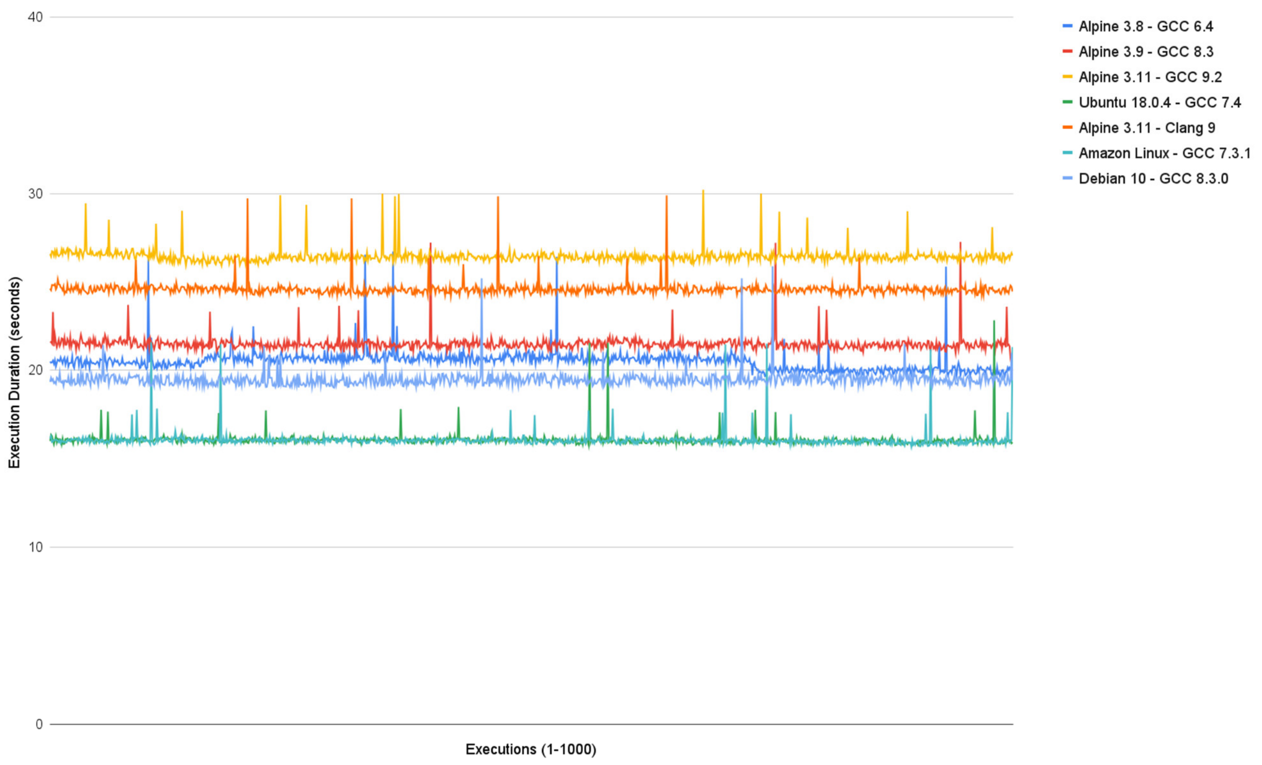


Figure 8: Execution Durations of Benchmarked Environments

The results show significant variations in the execution durations of the benchmarked environments. Amazon and Ubuntu Linux, both utilizing *Python builds* compiled with GCC version 7, outperformed executions using Alpine Linux and Python builds compiled with GCC versions 6, 8, and 9 and CLANG. Debian Linux, with GCC 8.3.0, exhibited fewer performance degradations but remained of lower performance than Amazon and Ubuntu Linux. The results demonstrate that seemingly non-important decisions such as skeleton containers or execution engine builds can result in up to 62% performance degradations. We also found that newer versions of Alpine and GCC appear to under-perform older versions. This behavior can be counter-intuitive to users attempting to improve performance by using the latest releases of skeleton containers and compilers.

The executions of the benchmarked environments were generally consistent. Deviations from the mean execution durations existed but were reasonably limited. We, thus, believe that skeleton containers and execution engine builds do not substantially impact the execution consistency of serverless functions.

5.4 Impact of Execution Engine Releases

This section of the investigation focuses on the impact of execution engine releases on serverless functions' execution performance and execution consistency. While users of open source platforms are likely to build their execution environments, users of commercial platforms are more likely to leverage one of the platform's pre-built execution environments. The pre-built environments offer different releases of execution engines (e.g., Python 3.6 and Python 3.7) that support multiple high-level languages (e.g., NodeJS, Python.etc.).

It was possible to investigate the impact of execution engine release differences using the same benchmarking setup in Section 5.3. However, it is more valuable to perform our analysis on commercial platforms where users frequently decide on execution engine releases. Thus, in this part of the investigation, we benchmark the pre-built Python 3 execution engines offered by the commercial platforms: i) AWS Lambda; ii) Google Cloud Functions; iii) Azure Cloud Functions. We developed a benchmarking function that performs a sequence of factorial calculations for a fixed set of numbers using Python's

Math[39] library. We refrained from using the *NumPy* library since it is not available by default in the benchmarked pre-built execution environments. The function is written in Python 3 and is compatible with the benchmarked execution engines. We invoked the function in the three platforms by defining an external API triggering event in each platform. When the API is called, the function is executed. We used a Linux VM hosted in Compute Canada's Graham Cloud to call the API.

5.4.1 AWS Lambda

We deployed the benchmarking function to the Python 3 execution environments offered by AWS Lambda: i) Python 3.6; ii) Python 3.7; and iii) Python 3.8. We performed 1000 invocations of the function in each execution environment with a gap period of 1 second between invocations. The gap period was chosen to ensure a warm invocation and avoid cold-start latency [12]. We utilized AWS CloudWatch[40] to report the execution durations of the three execution environments. The results are presented in Table 9 and plotted in Figure 9.

Table 9: AWS Lambda Execution Durations

Execution Environment	**Min**	**Max**	**Mean**	**Std. Dev**
Python 3.6	49.3	141.6	85.7	17.9
Python 3.7	75.1	171.6	134.2	9.3
Python 3.8	110.6	215.6	171.9	12.2

[39] Python Math: https://docs.python.org/3/library/math.html

[40] AWS Cloud Watch: https://aws.amazon.com/cloudwatch/

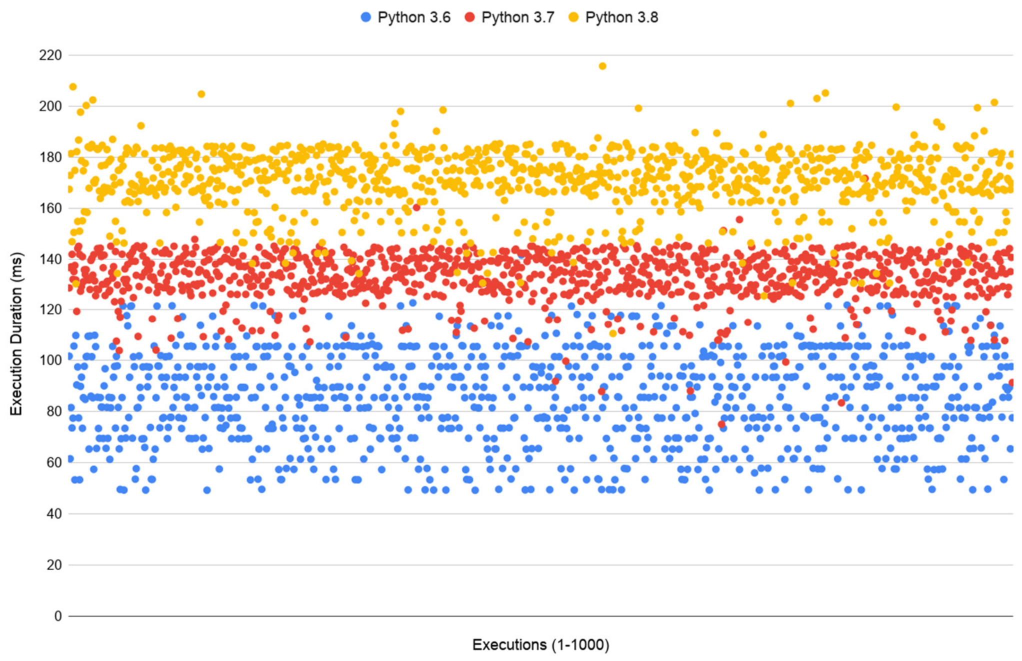


Figure 9: AWS Lambda Execution Performance

The results show performance degradations ranging from 9% to 30% between the execution environments. They also align with our earlier findings that newer releases of execution engines do not necessarily result in better execution performance. Executions utilizing the Python 3.8 execution environments underperformed both Python 3.7 and Python 3.6 environments. The execution consistency was reasonably acceptable, with most of the execution durations having limited deviations from the mean. However, due to the billing practices in AWS that round-up execution durations to the next 100ms step, costs incurred for Python 3.8 and 3.7 executions are significantly higher than the costs for Python 3.6 illustrated in Figure 10.

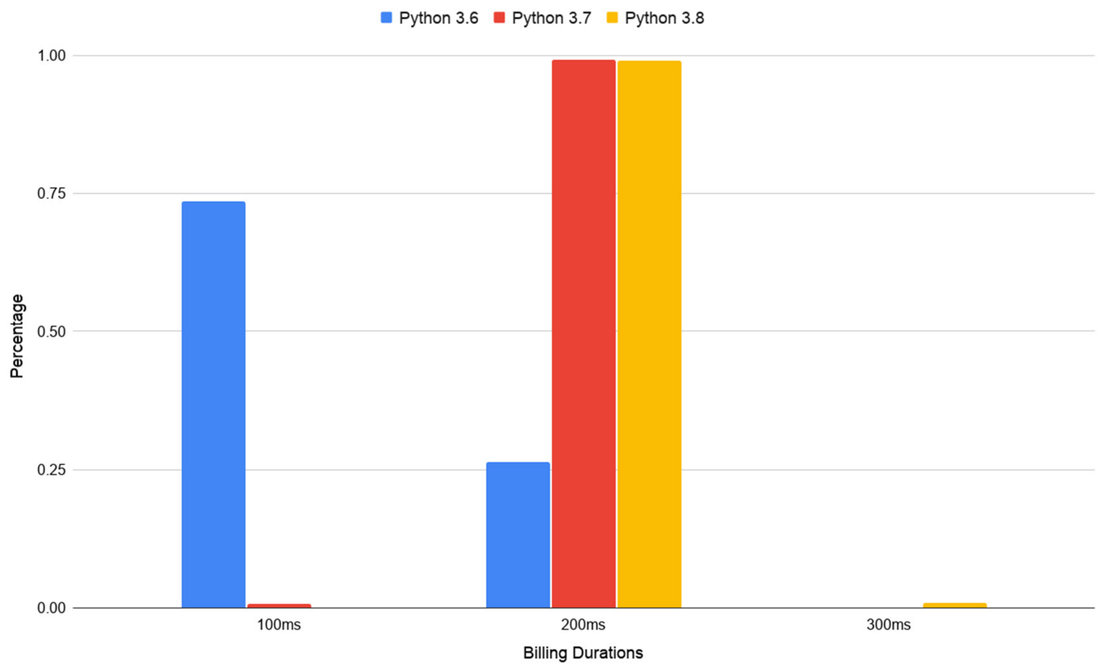


Figure 10: AWS Lambda Billing Durations

5.4.2 Google Cloud Functions

Similarly, we benchmarked the pre-built Python 3 execution environments offered by Google's Cloud Functions platform: i) Python 3.7 and ii) Python 3.8. In contrast with AWS Lambda, Google's Cloud Functions do not offer pre-built Python 3.6 environments. We performed 1000 invocations of the benchmarking function using each of the Python execution environments. We used Google's Cloud Logging[41] to record the execution durations. The results are presented in Table 10 and plotted in Figure 11.

Table 10: Google Cloud Functions Execution Durations

Execution Environment	Min	Max	Mean	Std. Dev
Python 3.7	15	225	69.8	51.4
Python 3.8	15	226	87.5	56.2

[41] Google Cloud Logging: https://cloud.google.com/logging

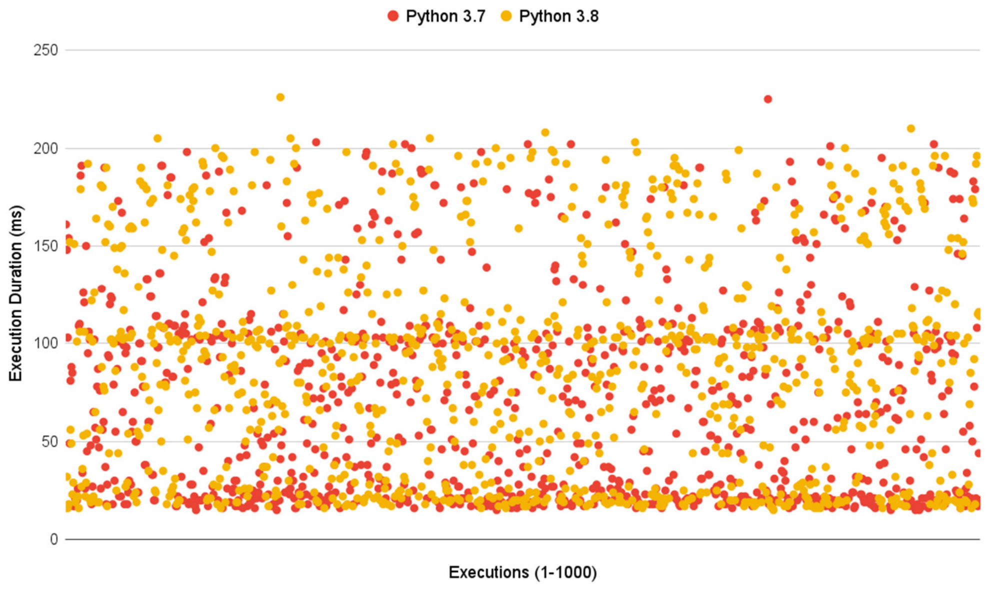


Figure 11: Google Cloud Functions Execution Performance

The results illustrate a limited performance degradation when using Python 3.8 vs. Python 3.7 execution environments. Also, the execution performance in Google's Cloud Functions exhibited high execution inconsistency. To examine its impact on the incurred costs, we plot the execution durations as 100ms steps utilized by Google for billing in Figure 12.

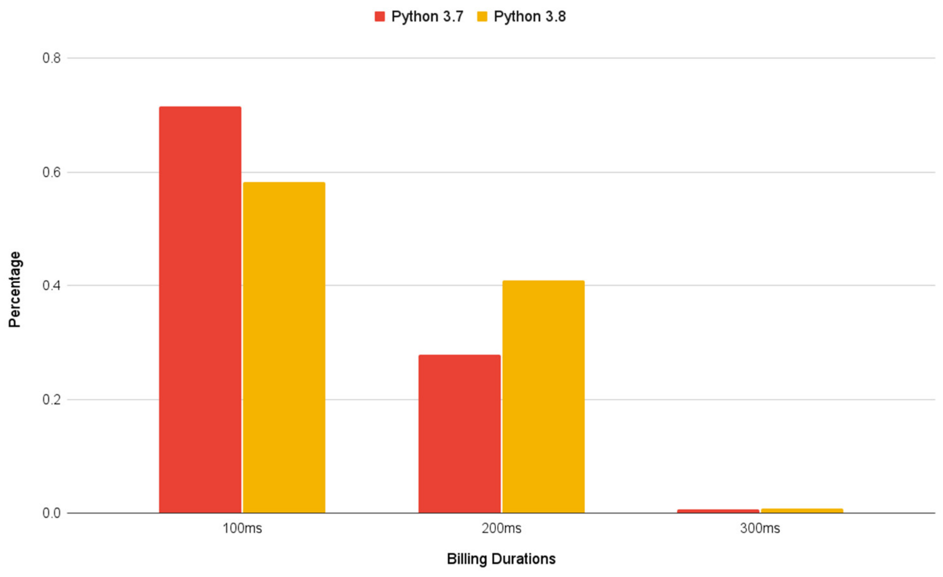


Figure 12: Google Cloud Functions Billing Durations

The results illustrate a visible impact for execution inconsistency, with our benchmarking function incurring three different costs for executions. The inconsistency impact was higher in the Python 3.8 environment, where approximately 42% of the executions were charged for 200ms or more compute resources.

5.4.3 Azure Cloud Functions

Finally, we benchmarked the pre-built Python 3 execution environments offered by Azure Cloud Functions platform: i) Python 3.6; ii) Python 3.7; and iii) Python 3.8. We used the same benchmarking function and performed 1000 invocations on each of the Python execution environments. We used Azure's Monitoring[42] to record the execution durations. The results are presented in Table 11 and plotted in Figure 13.

Table 11: Azure Execution Environments Performance

Execution Environment	**Min**	**Max**	**Mean**	**Std. Dev**
Python 3.6	29.4	300.6	41.6	11.7
Python 3.7	24.8	143.3	32.8	5.76
Python 3.8	29.1	172.7	38.6	7.44

[42] Azure Monitor: https://azure.microsoft.com/en-ca/services/monitor/

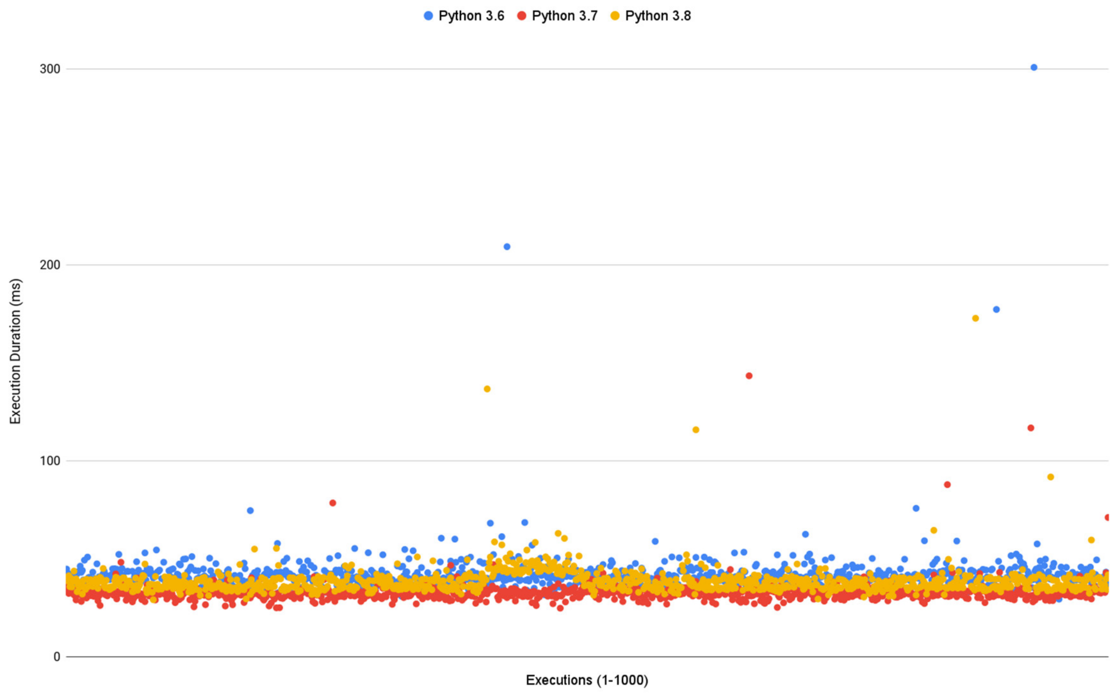


Figure 13: Microsoft Azure Execution Performance

The results show an execution performance variation between the three execution environments ranging from 8% to 28%. The execution durations were reasonably consistent, and the billing durations for 99% of the executions fall into the same billing category (< 100ms).

5.5 Impact of Ecosystem Performance

To demonstrate how minor ecosystem performance degradations can significantly impact the execution duration of a serverless function. We developed a simple object-storage Python function that retrieves a 50MB object from a MinIO bucket and rewrites the object to another bucket. We performed 1000 executions of the function and recorded the execution durations. We then induced a superficial 10ms network latency to the MinIO object storage server and repeated the 1000 executions while recording the execution durations. We used the same skeleton container, execution engine build, and release (Debian Linux 10 – GCC8.3.0 and Python 3.8.1) for all the executions. The purpose of this exercise is to examine how seemingly minor ecosystem degradations can cause a significant increase in execution durations. The execution durations are presented in Table 12 and plotted in Figure 14.

Table 12: Execution Durations per induced latency

Induced latency	Min	Max	Mean	Std. Dev
0 ms	1.7	3,8	1.8	0.13
10 ms	2.1	4.4	2.2	0.18

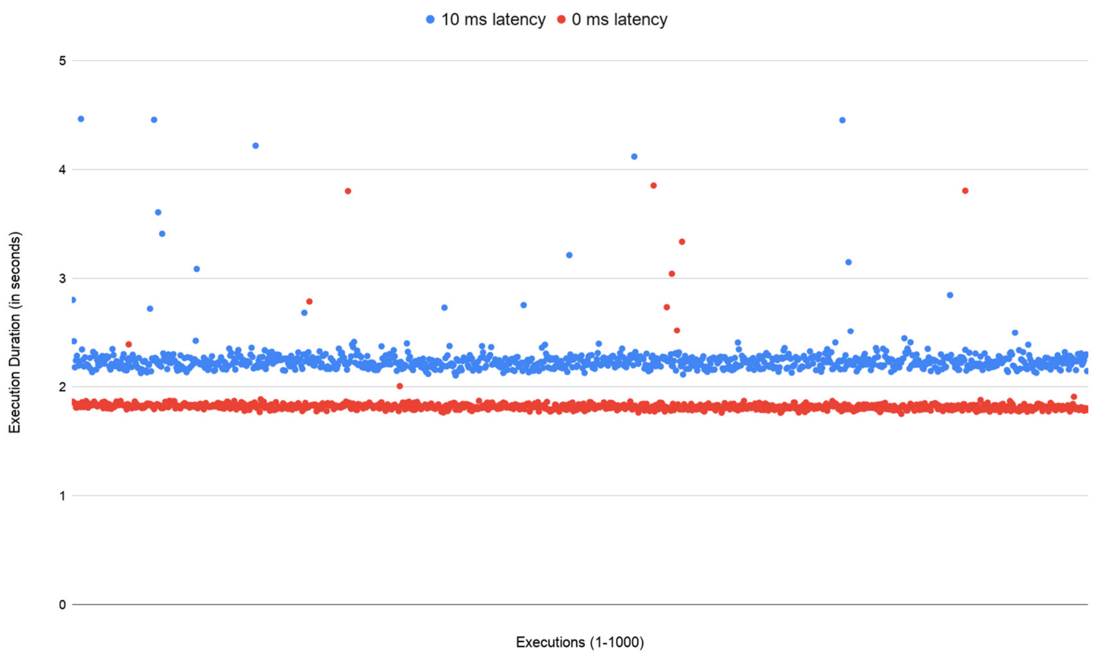


Figure 14: Execution Durations per induced latency

The execution durations show significant degradations (22% on average) when a 10ms induced latency impacts the object storage. The impact may be further compounded in real-life scenarios if the serverless function executes multiple object storage operations.

5.6 Discussion

The results illustrate that the choice of these three execution environment parameters (skeleton containers, execution engine builds, and releases) can significantly impact serverless functions' execution performance. The degree of performance degradation varied by the particular choices of the execution environment. The results also lead us to believe that skeleton containers and execution engine' builds and releases do not substantially impact serverless functions' execution consistency.

While some of the benchmarked execution engine releases demonstrated a high degree of execution inconsistency (e.g., Google's Python environments), we believe that such inconsistency is not a result of execution engine release differences. We instead think such inconsistency may result from the platform's architectural or implementation decisions as it impacted multiple releases of the benchmarked execution engine (Python 3.7 and 3.8 in Google's case).

The results of our investigation illustrate that users must carefully select the optimal execution environments for their functions or risk longer execution durations and higher costs. They also highlight that minor ecosystem degradations can significantly impact the performance of serverless functions. These findings illustrate the necessity of defining execution performance guarantees in serverless computing, where serverless providers are required to satisfy an agreed-upon execution performance or be penalized for violations. Suboptimal execution environments or ecosystem degradations can impact the serverless model's attractiveness and leads to resource utilization inefficiencies and monetary implications. In the next chapter, we investigate the SLA specification metrics in serverless computing. We then present a machine learning (ML) based approach to define and assess compliance with these specification metrics.

Chapter 6
Serverless Performance Guarantees

This chapter includes sections from the paper:

- M.Elsakhawy and M.Bauer, " FaaS2F: A framework for execution-SLA in serverless computing" published in the 2020 IEEE Cloud Summit (CloudSummit'20)"

6.1 Introduction

As illustrated in the previous chapter, serverless functions' executions are highly prone to performance degradations resulting from suboptimal execution environments or ecosystem contentions. Thus, it is crucial to define SLAs that commit the providers to an acceptable level of execution performance. This chapter discusses the current attempts at defining SLA in serverless computing and presents our proposed FaaS SLA Framework.

6.2 Performance Guarantees for Serverless Functions

As a new cloud delivery model, performance guarantees in serverless computing have not been thoroughly investigated. The literature of efforts serverless performance guarantees are limited, with the majority of works focusing on invocation-rate guarantees [13], scheduling latency guarantees [8], [14], [15], and resource allocation guarantees [16]. To our knowledge, no work has examined execution performance guarantees for serverless functions.

A challenge that faces defining execution performance guarantees in serverless computing is its unique abstraction level. The specification-metrics cannot rely on the traditional resource-guarantees as resources are entirely abstracted from the end-users. The challenge is further compounded by the stateless nature of serverless functions that mandates a high dependency on the provider's ecosystem for date access and storage. Thus, the specification-metrics must be generic enough to capture both the execution environment's performance and the ecosystem's performance. In the next section, we present the literature on the utilization of Machine Learning in Cloud Computing. This background is necessary for our proposed solution that is later discussed in this chapter.

6.3 Machine Learning in Cloud Computing

Machine Learning (ML) focuses on teaching computers the ability to predict results or provide classifications for inputs rather than explicitly programming them to do so tasks [60]. In the past few years, significant advancements have been achieved in Neural Network (NN) training and accuracy, which has led to the adoption of ML in a broad set of domains. Some researchers have attempted to utilize ML to address the resource allocation problem in the cloud. Saeed et al. [61] proposed an adaptive backpropagation NN that predicts the priority of submitted jobs to ensure the optimal utilization of resources. Liu et al. [62] proposed Radial Basis Function "RBF" NNs to address resource allocation contentions in cloud environments.

Imam et al. [63] proposed using Time Delay Neural Networks (TDNN) to predict future resource requests by VMs in IaaS clouds. Ray et al. [64] proposed nonlinear autoregressive "NARX" neural networks to reverse engineer job priorities based on the resource allocation of a job. The proposed approach was evaluated on Google's cloud traces. Zhang et al. [65] investigated the use of ML for resource allocation in auction-based clouds. The authors proposed the use of logistic regression for the classification of bids into winning and losing bids. Hemmat et al. [66] investigated leveraging ML for SLA violation detection in task submission in cloud computing. The authors defined an SLA violation as an occurrence of task eviction with no subsequent rescheduling. They examined two learning models and evaluated their proposed approach on Google Cloud Cluster traces.

In this work, we model the problem of FaaS SLA as a classification problem that a broad set of ML classifiers can solve. We model a function's execution using a set of specification metrics that are capable of capturing the performance of the execution environment and the platform's ecosystem (sections 4.3.3 and 4.3.4). We then use these specifications metrics to train a FaaS SLA classifier to detect a function's execution performance based on these metrics' distinct features and classify an execution into SLA-abiding and SLA non-abiding. In the next section, we discuss the specification metrics in detail and the process of defining the SLA.

6.4 Defining Execution-SLA

Serverless functions are ephemeral entities that rely on the provider's ecosystem for long-term data access and retention. A function's execution happens on shared infrastructure with non-dedicated access to the provider's ecosystem, making the execution prone to interference and performance degradations from other neighboring functions. Defining execution-performance guarantees must consider that serverless execution environments do not hold performance guarantees on local computational resources or the provider's ecosystem. Thus, SLAs involving execution-performance must capture the performance guarantees while avoiding the classic resource-based SLA definitions.

We propose the concept of an *execution-SLA* and define it as a binding customer-provider agreement to perform an *SLA-abiding Execution* of a serverless function. An SLA-abiding Execution is an execution that complies with the execution-performance guarantees and occurs when the provider's platform satisfies the following criteria:

- Computational resources (CPU, Memory, and local Disk) are readily available when requested by a serverless function's execution environment. Thus, no resource contentions should occur.
- The provider's ecosystem operates in alignment with its advertised SLA, and serverless functions' execution environments are granted access to the ecosystem when requested.

An SLA-abiding execution is defined in terms of four fingerprints derived from the execution's resource utilization. The fingerprints represent the performance of the layers abstracted by the FaaS model, i.e., resource, ecosystem, and execution environment (Sections 4.3.3 and 4.3.4) as follows:

- CPU & Memory utilization fingerprints are leveraged to model the performance of accessing local resources by a serverless function's execution environment.
- Network Transmit and Receive (Tx/Rx) utilization fingerprints are leveraged to model the performance of accessing the provider's ecosystem, such as long-term data storage or messaging queues, by a function's execution environment.

A serverless function generally exhibits limited variance in its computational workflow; multiple SLA-abiding Executions of the same serverless function are expected to produce resource utilization fingerprints with a high degree of similarity. Thus, significant deviations in resource utilization fingerprints can identify SLA non-abiding executions and, consequently, SLA violations. In this work, we assume a performance homogeneity of the resources at the serverless providers. Performance homogeneity means that the physical servers' computational and network performance in the provider's server pool is identical. This homogeneity applies to performance-impacting factors such as CPU make and models, Memory frequency and network bandwidth, and latency. We also assume the independence of the execution performance on factors external to the serverless provider, such as APIs call to an external entity. The next section introduces Function as a Service SLA Framework (FaaS2F), a framework to define and assess compliance with execution-SLAs for serverless functions.

6.5 FaaS2F

FaaS2F is a modular framework to 1) define execution-SLAs, and b) detect violations of execution-SLAs in serverless functions. FaaS2F uses ML to build a model of execution fingerprints for a specific serverless function. The resulting model can then compare actual executions' fingerprints to the expected execution fingerprints. Using FaaS2F, end-users and cloud providers can define execution-SLAs for functions utilizing the following procedure:

- An end-user submits their function's code to the provider's serverless framework.
- The provider generates an execution environment for the function, executes the function in a *Training Phase*, and generates SLA-abiding and SLA non-abiding resource utilization fingerprints.
- The provider trains four fingerprint-classification NNs and provides the trained NN models to the user.
- The provider moves the function to the *Production Phase* and supplies the function's executions resource utilization fingerprints to the end-user.

- The end-user and the provider utilize the trained NN models to classify executions into SLA-abiding and non-abiding, validating the compliance with the execution-SLA.

The procedure is illustrated in Figure 15

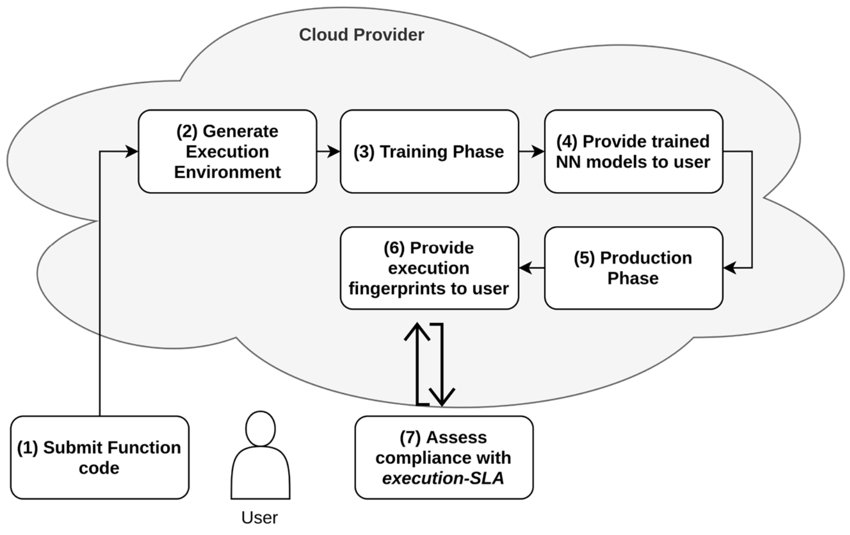

Cloud Provider
(2) Generate Execution Environment
(3) Training Phase
(4) Provide trained NN models to user
(6) Provide execution fingerprints to user
(5) Production Phase
(1) Submit Function code
User
(7) Assess compliance with *execution-SLA*

Figure 15: FaaS2F Diagram

6.6 FaaS2F Architecture

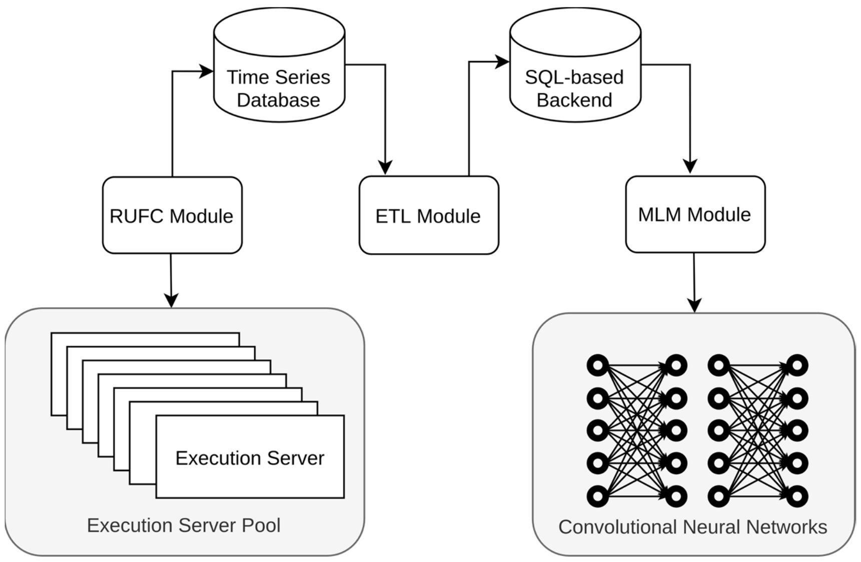


Figure 16: FaaS2F High-level Diagram

Figure 16 illustrates FaaS2F architecture. The Resource Utilization Fingerprint Collection (RUFC) module collects resource utilization metrics for serverless functions' executions and stores them in a time-series database. Time-series databases are a special kind of database better suited for resource metrics collection [67]. The metrics are collected on equally spaced time intervals, starting at the function's invocation and ending with the function's completion. In our validation (Section 6.10), we collect resource utilization derivatives calculated by subtracting the previous resource utilization from the current one and dividing the result by the time difference. We utilize Google's *Cadvisor*[43] to collect the metrics and InfluxDB[44] to store them.

The Extraction, Transformation, and Loading (ETL) module extracts utilization metrics of a function's execution from the time-series database, transforms them into resource utilization fingerprints, and stores them in a SQL-based backend. A resource utilization fingerprint is a vector containing all the resource utilization metrics for a function's execution. The fingerprints are utilized by the Machine Learning Module (MLM) to perform the following functionalities:

- Train a set of Neural Networks (NNs) to classify SLA-abiding executions based on resource utilization fingerprints.
- Assess compliance with execution-SLA by classifying a function's executions into SLA-abiding and non-abiding using the trained NNs.

Our implementation for FaaS2F is written in Python and is integrated with the *Knative*[45] serverless framework. It leverages the commonly used ML libraries: *Keras*[46] and *Tensorflow*[47] for NN training and subsequent classification. We designed the RUFC

[43] Cadvisor: https://github.com/google/cadvisor
[44] InfluxDB: https://www.influxdata.com/
[45] Knative: https://knative.dev/
[46] Keras: https://keras.io/
[47] TensorFlow: https://www.tensorflow.org/

module to be resident on every *Knative* worker node while the ETL and the MLM reside with the *Knative* control plane, as illustrated in Figure 17.

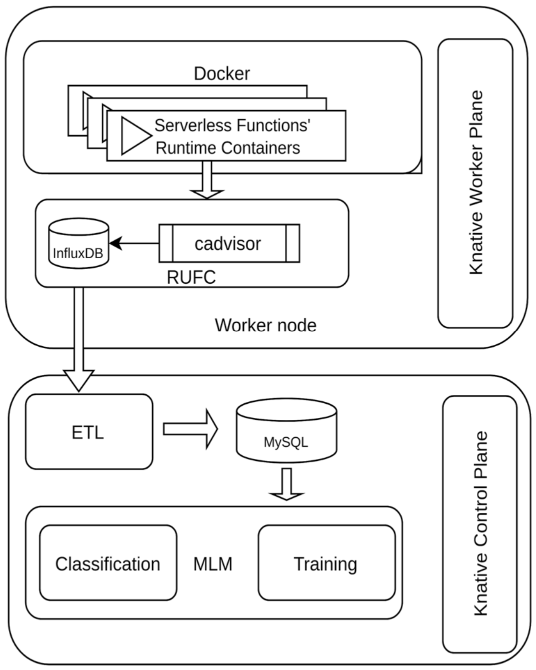


Figure 17: FaaS2F Implementation Diagram

6.7 Resource Utilization Fingerprints Generation

To generate the resource utilization fingerprints that model a function's *SLA-abiding* and *SLA non-abiding* executions, RUFC leverages a *Training Phase* that executes the function with a randomized set of input payloads in a production-replica environment as follows:

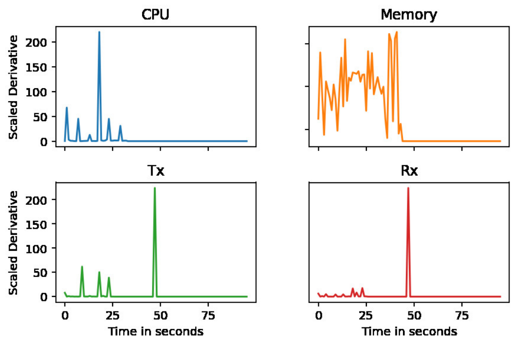
- While modeling *SLA-abiding executions*, the replica environment allocates dedicated local computational resources to the serverless functions, and the platform's ecosystem performance abides with its posted SLAs. Resource utilization fingerprints collected during this phase are labeled *SLA-abiding execution fingerprints.*
- While modeling *SLA non-abiding executions*, the replica environment is subjected to induced resource and ecosystem stresses. Stresses emulate contentions in accessing local resources such as excessive loads on CPU and memory and non-

local resources such as the provider's ecosystem. Ecosystem contentions are emulated by inducing a sustained drop in durable storage IOPS or a sustained latency increase in a message queue service. Resource utilization fingerprints collected during this phase are labeled *SLA non-abiding execution fingerprints.*

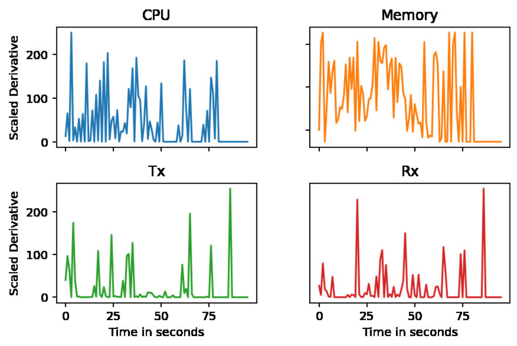
6.7.1 Input Payload Randomizing

Randomizing the input payloads in the training phase is necessary to ensure the collected fingerprints represent generalizations of SLA-abiding and SLA non-abiding executions' utilization fingerprints. To randomize the input payloads during the training phase, end-users must define i) the payload type of a function's input (e.g., numeric, text, etc.) and ii) the payload ranges. The cloud provider then generates a training dataset to use as the input payloads for the function in the training phase. The training dataset consists of random payloads of the same type and within the range defined by the end-user. For example, the cloud provider may generate a set of image files of varying sizes and resolutions as the training dataset for a serverless function that receives an image file as an input.

The training dataset is inputted to the serverless function one at a time. At the same time, resource utilization metrics (CPU, Memory, Rx, and Tx) are monitored and stored in the time-series database. These metrics are collected twice for each input payload: once with *SLA-abiding execution* conditions and another time with *SLA non-abiding execution* conditions. Examples of scaled fingerprints are plotted in Figures 18(a) and 18(b). For illustrative purposes, we use a Min-Max scaler with upper/lower boundaries as 255 and 0 for CPU, Tx, and Rx and 255 to -255 for Memory. We use *RobustScaler* instead in the implementation, as explained in section 6.10.



(a)



(b)

Figure 18: Sample scaled fingerprints

6.8 MLM Classifier Training

The fingerprints generation step is followed by a training step that teaches, via supervised learning, a set of ML models to classify executions into SLA-abiding and SLA non-abiding based on their resource utilization fingerprints. Our empirical validation utilized convolutional neural networks [18] (CNNs) and Random Forest classifiers [68]. We obtained acceptable classification accuracy with CNNs and Random Forst with a relatively small number of training samples. We settled on CNNs as the classifier of choice in our validation; the reasons for this choice and the accuracy of CNNs vs. SVMs vs. RFs are discussed in section 7.3.3. We trained 4 CNNs using the collected resource usage fingerprints. Each CNN utilizes one layer of 4 convolutional filters, followed by two layers of dense neurons. In the training phase, we made the following choices:

- We train a separate classifier for each resource utilization fingerprint. The decision to train four independent CNNs instead of one CNN with the four fingerprints was based on the following rationale:
    - An individual classifier for each fingerprint enables the detection of the resource that caused the SLA violation (i.e., CPU, Memory..etc.). This enables the users and the providers to pinpoint the cause of the SLA violation and take the necessary action to address it.
    - Our experimental validation found that the convergence of NN training using four separate CNNs is achievable with a smaller number of training samples than one CNN with a 4-dimensional fingerprint.
- Our choice of CNN as the NN architecture does not imply CNNs' exclusive superiority over other ML models in execution classification. We have obtained high classification accuracy using RandomForests. The focus of our work is not ML research. We dedicate section 7.3.3 to discuss the aspects of choosing the ML classifier.
- The CNNs produce a binary classification for each execution based on the resource utilization fingerprints, i.e., whether the execution abided or did not abide by the

SLA. Thus, the classification step does not provide the degree of the violation. Future continuations of this work will explore this direction.

Figure 19 illustrates the CNN design in our experimental validation for an input resource utilization fingerprint of dimensions 144 x1. Convolution is applied with a set of 4 filters of a kernel size 2 x 1. The filters are responsible for detecting features in the resource utilization fingerprints. The filter layer is followed by a layer of 256 dense neurons and an output layer with a set of two neurons representing SLA-abiding and SLA non-abiding executions. To improve the classification performance, we use zero-padding for the input vectors. We normalize the resource utilization derivative values by scaling CPU, Mem, Tx, and Rx using Robust Scaling [69]. Our implementation of RUFC using *Cadvisor* collects cumulative resource utilization values for CPU, Tx, and Rx and absolute values for memory. Thus, derivative values for CPU, Tx, and Rx are positive values, while memory derivative values can include negative values.

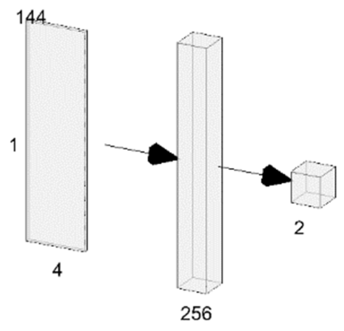


Figure 19: CNN Architecture

To avoid the model's overfitting on our training dataset, we employed a commonly used data augmentation technique that introduces a random horizontal displacement in each training sample. Data augmentation provides several techniques that target the training dataset's quality and size; by applying these techniques, the training dataset is artificially inflated to enhance the generalizability of the classification model [70].

6.9 Assessing Compliance with Execution-SLA

Following the NN training step, the serverless function is deployed to the cloud provider's production environment starting the Production Phase. Similar to the Training Phase, resource utilization metrics of the function's executions are collected. The resource utilization fingerprints are then used as unlabeled inputs to the trained NNs to examine if an execution classifies as SLA-Abiding or SLA non-abiding.

6.10 Problem Formulation

For a serverless function F, the resource utilization for a single execution E with duration T is modeled as the set

$$E = \{CPU, RAM, Tx, Rx\} \qquad (1)$$

CPU is a set of the CPU utilization records for function F, taken at equally spaced time intervals during time period T

$$CPU \; = \{C_0, C_1, \dots, C_t\} \qquad (2)$$

$$where\ C_0\ is\ the\ first\ collected\ CPU\ usage\ metric\ and$$

$$C_t\ is\ the\ lastand\ \ t\ is\ the\ total\ number\ of\ records$$

$collected\ during\ time\ period\ T$

Mem is a set of the memory utilization records for function F, taken at equally spaced time intervals during time period T

$$Mem \; = \{M_0, M_1, \dots, M_t\} \qquad (3)$$

$$where\ M_0\ is\ the\ first\ collected\ Memory\ usage\ metric\ and$$

$$M_t\ is\ the\ last\ \textit{collected metric}$$

$$and\ \ t\ is\ the\ total\ number\ of\ records\ collected$$

$$during\ time\ period\ T$$

Tx is a set of the network transmission utilization records for function F, taken at equally spaced time intervals during period T

$$Tx\ = \{Tx_0, Tx_1, \dots, Tx_t\} \qquad (4)$$

$$where\ Tx_0\ is\ the\ first\ collected\ Tx\ usage\ metric\ and\ Tx_t$$

$$is\ the\ last\ and\ \ t\ is\ the\ total\ number\ of\ records$$

$$collected\ during\ time\ period\ T$$

Rx is a set of the network receiving utilization records for function F, taken at equally spaced time intervals during period T

$$Rx\ = \{Rx_0, Rx_1, \dots, Rx_t\} \qquad (5)$$

$$where\ Rx_0\ is\ the\ first\ collected\ Rx\ usage\ metric\ and$$

$$Rx_t\ is\ the\ last\ t\ is\ the\ total\ number\ ofrecords\ collected\ \ during\ time\ duration\ T$$

During the *Training Phase*, two sets of functions executions are performed, generating resource utilization metrics sets P and N. P is the set of collected resource utilization metrics representing *SLA-abiding* executions, i.e., positive samples, and N is the set of collected metrics representing *SLA non-abiding* executions, i.e., negative samples.

$$P = \{\ E_0,\ E_1,\ E_2, \dots,\ E_n\} \qquad (6)$$

$$where\ \ E_{i\ ,0\ \le\ i\ \le\ n}\ are\ SLA - abiding$$

$$executions\ of\ function\ F$$

and

$$N = \{\ E_0,\ E_1,\ E_2, \dots,\ E_m\} \qquad (7)$$

$$where\ \ E_{j\ ,0\ \le\ j\ \le\ m}\ \ are\ SLA\ non - abiding$$

$$executions\ of\ function\ F$$

CPU, Mem, Tx, Rx records for executions in $P$and N are 1-dimensional vectors such that:

$$CPU_P = \{ CPU_{E_0}, CPU_{E_1}, CPU_{E_2}, \dots, CPU_{E_n} \} \quad (8)$$

$$Mem_P = \{ Mem_{E_0}, Mem_{E_1}, Mem_{E_2}, \dots, Mem_{E_n} \} \quad (9)$$

$$Tx_P = \{ Tx_{E_0}, Tx_{E_1}, Tx_{E_2}, \dots, Tx_{E_n} \} \quad (10)$$

$$Rx_P = \{ Rx_{E_0}, Rx_{E_1}, Rx_{E_2}, \dots, Rx_{E_n} \} \quad (11)$$

where $CPU_{E_i}, Mem_{E_i}, Tx_{E_i}$an Rx_{E_i} are the CPU, Mem, Tx, Rx utilization records for execution $E_{i\,,0 \leq i \leq n}$.

And

$$CPU_N = \{ CPU_{E_0}, CPU_{E_1}, CPU_{E_2}, \dots, CPU_{E_m} \} \quad (12)$$

$$Mem_N = \{ Mem_{E_0}, Mem_{E_1}, Mem_{E_2}, \dots, Mem_{E_m} \} \quad (13)$$

$$Tx_N = \{ Tx_{E_0}, Tx_{E_1}, Tx_{E_2}, \dots, Tx_{E_m} \} \quad (14)$$

$$Rx_N = \{ Rx_{E_0}, Rx_{E_1}, Rx_{E_2}, \dots, Rx_{E_m} \} \quad (15)$$

where $CPU_{E_j}, Mem_{E_j}, Tx_j$ and Rx_{E_j} are the CPU, Mem, Tx, Rx utilization records for execution $E_{j\,,0 \leq j \leq m}$.

The utilization values $CPU, Mem, Tx\ and\ Rx$ are scaled using RobustScaler [71]. The scaled execution records in P and N are used to train 4 CNNs $\{CNN_{CPU}, CNN_{Mem}, CNN_{Tx}, CNN_{Rx}\}$. Each CNN focuses on a single utilization fingerprint. Labels are set as *1* for executions in set P and *0* for executions in set N.

Following the *Training Phase*, the function F is moved to the *Production Phase*. Invocations in this phase are denoted by the set U.

$$U = \{ E_0, E_1, E_2, \dots, E_n \} \quad (16)$$

The following resource utilization records are generated:

$$CPU_U = \{ CPU_{E_0}, CPU_{E_1}, CPU_{E_2}, \dots, CPU_{E_n} \} \quad (17)$$

$$Mem_U = \{ Mem_{E_0}, Mem_{E_1}, Mem_{E_2}, \dots, Mem_{E_n} \} \quad (18)$$

$$Tx_U = \{ Tx_{E_0}, Tx_{E_1}, Tx_{E_2}, \dots, Tx_{E_n} \} \quad (19)$$

$$Rx_U = \{ Rx_{E_0}, Rx_{E_1}, Rx_{E_2}, \dots, Rx_{E_n} \} \quad (20)$$

The trained CNNs $\{CNN_{CPU}, CNN_{Mem}, CNN_{Tx}, CNN_{Rx}\}$ are utilized to classify executions in U into one of the two labels: *1* and *0*, representing *SLA-abiding* and *non-abiding* executions.

6.11 Validation

To validate FaaS2F's accuracy, we developed two serverless functions representing two categories of applications i) Image Processing and ii) High Storage-IOPS:

- Function 1: applies a set of image transformations to an input image utilizing Python's PIL library and saves the output image to a MinIO[48] bucket.

- Function 2: retrieves the list of thousands of objects stored in a MinIO bucket and loops through downloading them to the execution environment ephemeral storage.

We deployed the functions to a Knative serverless framework in a Kubernetes[49] cluster. Our hardware setup consists of two identical physical servers, each with 20 cores based on an Intel Xeon CPU E5-2660 v3 processor and 128 GB of Memory. We use 10Gbit Ethernet connections between the two nodes and the CentOS operating system. The validation setup software components are as follows:

Components on Node 1 of the Kubernetes cluster :

- Kubernetes and Knative control planes.

- FaaS2F's ETL and MLM modules.

[48] MinIO: https://min.io/

[49] Kubernetes: https://kubernetes.io/

- MinIO Object Storage for long-term data retention.

Components on Node 2 of the Kubernetes cluster :

- Execution environment Docker containers for the serverless functions
- FaaS2F RUFC module.
- An Apache web-server[50] to serve input payloads to Function 1.

Function 1 performs a sequence of image transformations to its payload by performing the following procedure:

- Retrieve a random image from an image-dataset served by the Apache webserver. We leverage the University of Oxford flower images-dataset [72] that contains 8114 JPEG-encoded images of flowers commonly present in the U.K. as the input payload to our function. The image sizes range from 12 KB to 112 KB representing a wide range of possible payloads.
- A predefined sequence of color adjustments and image resizing is applied to the retrieved image.
- The output of the transformations is saved to the execution environment as a PNG image.
- The PNG image is committed to the MinIO object-store utilizing the MinIO Python API.

Function 2 utilizes MinIO API to retrieve a list of objects stored in a MinIO storage bucket and iteratively downloads them one at a time to the execution environment's ephemeral storage. We selected these validation functions as representatives to categories identified by Berkley's researchers [11] as computational workflows that stretch today's serverless computing. Thus we consider them ideal candidates for requiring execution-performance guarantees.

[50] Apache Webserver: https://httpd.apache.org/

To train the four CNNs for each function, we generated the training samples by collecting *SLA-abiding* and *SLA non-abiding resource utilization fingerprints* for each function using the guidelines in Section 6.7. We utilized the Linux *stress* utility to induce CPU and Memory stresses on Node 2 and the *tc* utility to induce network latency for MinIO storage access. In our test environment, we collected the SLA abiding fingerprints when our benchmarking serverless functions were the only running functions on the host. Real-live serverless platforms may co-locate many functions on the same physical/virtual server. If fingerprints for defining the SLA are collected in such a situation, then one would expect that SLA-abiding executions' fingerprints in these real-life environments would be similar to those in the training environment. As the fingerprints are only affected by the performance of accessing CPU, Memory, Network transmit and receive, the collected fingerprints will change if the functions' collocation impacts the performance of accessing these resources adversely, thus causing an SLA violation.

Once the fingerprints were collected, we used them to train the CNNs for each function. We then moved to the *Production Phase* and performed a predefined mixture of *SLA-abiding* and *SLA non-abiding executions* of the functions to serve as our evaluation set. The number of samples in the training set and the evaluation set for each function are shown in Table 13. We selected the evaluation samples' count to achieve a minimum of 10 to 1 ratio to the total training samples count.

Table 13: Training and Evaluation Samples

Validation Function	**Number of Samples**		
	Positive Training Samples	**Negative Training Samples**	**Evaluation Samples**
Function 1	1000	1000	25,000
Function 2	400	400	8,000

The resource metrics were collected with a frequency of one reading per second. We specified our training batch size as 50 samples per epoch and leveraged ADAM [73] for training optimization and *Keras ImageDataGenerator* to introduce random horizontal shifts in the input fingerprints. Following the training process, we evaluated the classification accuracy of the trained CNNs using the evaluation set. We measured the correctness of classifying executions into SLA-abiding and SLA non-abiding based on our prior knowledge of the evaluation set. The execution classification accuracies for each function based on the four resource utilization fingerprints (CPU, Memory, Tx, and Rx) are presented in Table 14.

Table 14: Classification Accuracy

Validation Function	Resource Utilization Fingerprint			
	CPU	Memory	Tx	Rx
Function 1	92.8%	82.8%	99.1%	93.4%
Function 2	97.2%	71.7%	99.9%	99.7%

The results of our empirical validation illustrate that FaaS2F is capable of detecting, with high accuracy, sub-optimal executions of serverless functions based on the resource utilization fingerprints of their executions.

6.12 Number of Training Samples

To examine the impact of the number of training samples on the classification accuracy, we repeated the training of FaaS2F for Function 1 with a different number of samples and observed the classification accuracy. The accuracy resulting from each training sample count is presented in Table 15 and illustrated in Fig 20.

Table 15: Classification Accuracy per Training Sample count

Number of Positive/Negative Training Samples	Resource Utilization Fingerprint			
	CPU	Memory	Tx	Rx
300	56.3%	73.6%	98.3%	90%
400	54.3%	82.1%	98.3%	88.9%
500	73.8%	74.7%	98.8%	92.6%
600	64.1%	82.2%	98.8%	90.3%
700	56%	84.6%	99.1%	89.6%
800	70.7%	79.6%	98.7%	93.5%
900	81.1%	82.7%	99.1%	91.9%
1000	92.8%	82.8%	99.1%	93.4%

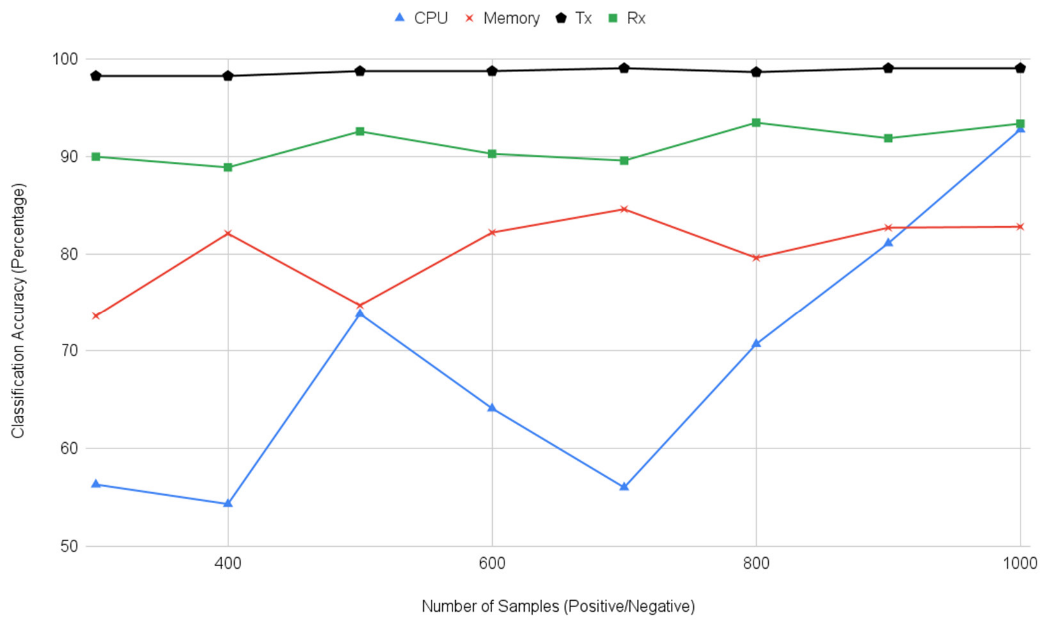


Figure 20: Classification Accuracy per Training Sample count

As observed in the results above, the degree of improvement in accuracy varied per each resource fingerprint. While classification based on CPU fingerprints has improved from 56.3% to 92.8% by increasing the training dataset from 300 to 1000 samples, the accuracy of other fingerprints such as Tx roughly remained unchanged. This leads us to the following conclusions:

- As four distinct and unique datasets, the number of samples needed for an ML model to converge upon training is unique for each of the four fingerprints.
- Similarly, the ML classifier that can produce high accuracy for one fingerprint may produce poor results when applied to another fingerprint. This opens the door to exploring other ML models that address this unique challenge. We touch base on this point later in Section 7.3.3.
- The unique characteristic of the four fingerprints is extendable on a per-function level. This implies that the number of samples for training FaaS2F on one function's dataset can vastly differ from the number of samples for another function. This can also be observed from Table 14's results, where the classification

accuracy for Function 1 was very close to Function 2 despite using 2.5 times the number of samples (1000 vs. 400).

6.13 Conclusion

Our empirical validation for FaaS2F illustrated a high accuracy in detecting sub-optimal executions based on the resource utilization fingerprints of these executions. The results illustrate that resource utilization fingerprints effectively model the execution performance of a serverless function and can accurately reflect degradations resulting from resource contentions or ecosystem bottlenecks. The classification accuracy of each fingerprint is dependent on the degree of fingerprint variation when a sub-optimal execution occurs. This variance is highly correlated to the nature of computations of the serverless functions. For example, CPU-intensive serverless functions (e.g., Function 1) are expected to exhibit high variation in CPU fingerprints between an SLA-abiding execution and an SLA non-abiding execution. Similarly, for network-heavy functions (e.g., Function 2), Tx and Rx fingerprints can be expected to produce high classification accuracy.

While the results of detecting sub-optimal executions for individual functions were promising, applications in real life rarely consist of a single function. Thus, in the next chapter, we investigate the applicability of FaaS2F for composites of serverless functions, i.e., serverless chains.

Chapter 7
SLA for Serverless Chains

This chapter includes sections from the paper:

- M.Elsakhawy and M.Bauer, "SLA for Sequential Serverless Chains: A Machine Learning Approach" published in the 7th International Workshop on Serverless Computing (WoSC'7), part of ACM/IFIP Middleware Conference. "

7.1 Introduction

Serverless platforms require functions to "operate asynchronously and process one request at a time" [14]. In traditional programming, applications are composites of many functions that interact to perform the application's functionality [15]. Similarly, serverless functions can construct larger applications through composition. Serverless chains are constructed by sequentially invoking functions either synchronously from within the functions' code or asynchronously by invoking ecosystem triggers [52]. Composition frameworks are add-on services provided by serverless platforms to coordinate invocations of synchronous functions.

A study by Tariq et al. [52] has found serverless chains to experience mid-chain drops, concurrency limit issues, and burst intolerance in commercial platforms. Another study by Sreekanti et al. [74] uncovered a compounding high latency overhead of AWS STEP functions that becomes intolerable after chaining five functions. A characteristic that makes defining SLA for serverless chains more challenging is their inherently diverse workflows. These workflows can include sequential triggering one function after another or several functions after one function is invoked or other workflows [75]. A single chain can follow multiple workflows in different executions based on the input. We limit our investigation to examining SLAs for sequential synchronous serverless chains, where functions are invoked one after another from within each function's code.

7.2 Modeling Serverless Chains Performance

Two questions naturally arise. The first is how to model a serverless chain's execution performance. The second is whether to define performance guarantees to a sequential serverless chain as a single unit or individually to each of its constructing functions. We leverage our earlier execution-performance modeling [76] using resource utilization fingerprints to define performance guarantees for serverless chains as a single unit. The rationale for our choice is as follows: while it is possible to define performance guarantees for every function forming the chain, this approach quickly becomes infeasible as the chain's number of functions increases. A chain with many functions will incur a high overhead for defining performance guarantees for each of its functions and even a higher overhead assessing compliance with them.

Thus, we treat a serverless chain as a single unit whose execution performance is modeled using its constructing functions' stacked resource utilization fingerprints. We collect the resource utilization metrics for all the chain's constructing functions, stack them, and use them to define one resource utilization fingerprint that models the chain's performance. A sample chain's CPU, Memory, Tx, and Rx fingerprints are shown in Fig 21 (a), along with an illustration of how constructing functions' are stacked to create them in Fig 21 (b).

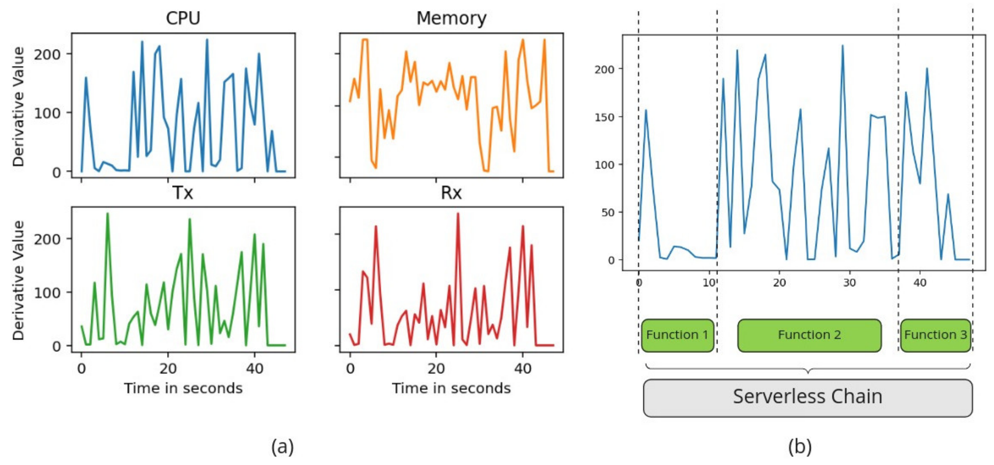


Figure 21: Resource utilization fingerprints stacking

Following the same principles used in FaaS2F, we leverage a Training Phase to generate resource utilization fingerprints representing a chain's SLA-abiding and SLA non-abiding executions. We use the generated fingerprints to train a classifier to detect SLA compliance or non-compliance based on the input resource utilization statistics. We utilize a 4-layer CNN for classification, as discussed in section 6.8.

7.3 Validation

In our validation, we divide sequential serverless chains into two categories based on their sizes:

- *Fixed-size Sequential Serverless Chains*: These chains are composed of a deterministic number of serverless functions executed in sequence when the chain is triggered. Multiple executions of a fixed-size chain result in the same number of functions that are executed. An example for this category of chains is depicted in Fig 22 (a), where the number of functions composing the chain, i.e., the chain's size, is three.

- *Variable-size Sequential Serverless Chains*: These chains are composed of a variable number of functions. The concurrency of the functions forming the chain varies between multiple chain executions resulting in a variable-sized chain. This variance can be dependent on the chain's input or the workflow specifications of the user. For example, the size of the chain in Fig 22 (b) is dependent on the number of concurrent executions of Function 2, which can vary between multiple executions of the chain.

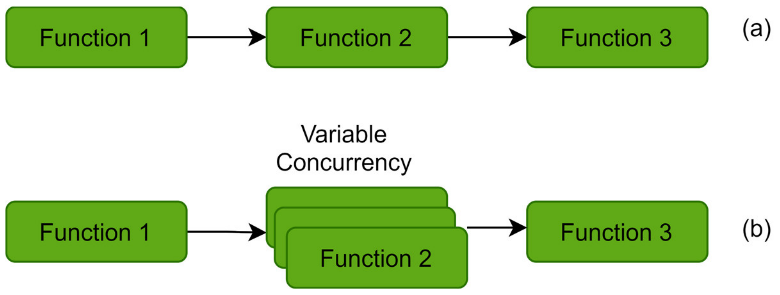


Figure 22: Fixed-size (a) and Variable-size (b) chains

7.3.1 Fixed-size Sequential Serverless chains

We implement two sequential serverless chains of fixed size representing different workloads. Chain 1 is an ML pipeline composed of three functions. The chain receives an image as input and generates a German text representing its contents. We rely on the VGG [77] pre-trained model for object classification and the *py-googletrans* package for translation. Chain 2 is a video processing application composed of six functions. It receives a video file as an input and outputs a text file listing the videos' resolution and codec and a set of randomly transformed screenshots from the video. We leverage the OpenCV[51] python library for video processing and PIL for image transformations. Chains 1 and 2 use MinIO object storage for data input and output and are invoked synchronously using HTTP requests. The two chains' workflows are illustrated in Fig. 23(a) and 23(b).

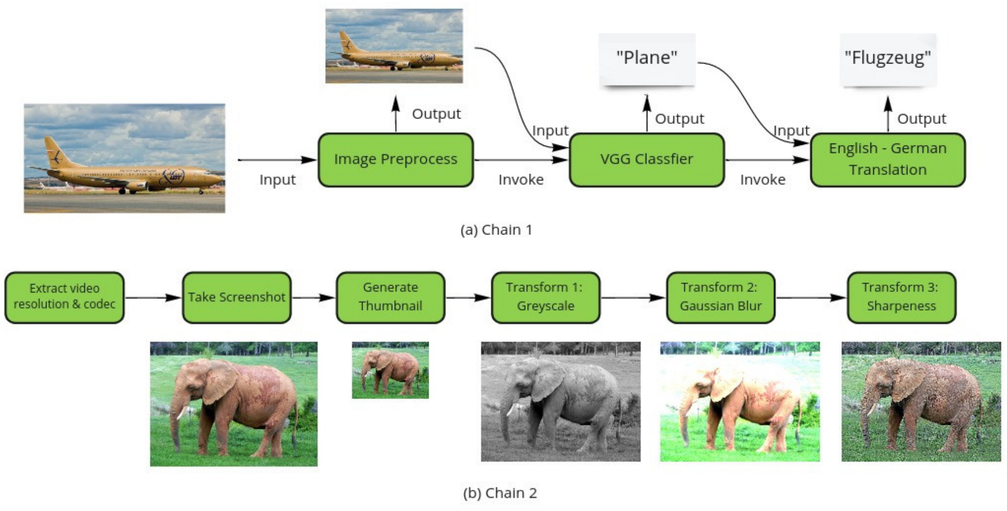


Figure 23: Fixed-size chains workflows

We utilized a subset of MS COCO data-set [78] as input for Chain 1 and a subset of Moments in time data-set [79] as Chain 2's input data-set. We deployed the chains to a two-node *Knative-over-Kubernetes* cluster. One node hosts the cluster's control plane, while the other is a worker node. Our hardware setup consists of two identical physical servers, each with 20 cores based on Intel(R) Xeon(R) CPU E5-2660 v3 processor and 128 GB of Memory. We trained FaaS2F using resource utilization fingerprints of 1000 SLA-abiding and SLA non-abiding executions of each chain. We evaluated the classification accuracy on a validation data-set of 20,000 executions of each chain. We maintain a 20:1

[51] OpenCV: https://opencv.org/

validation-to-training samples ratio. The accuracy of classification per resource utilization fingerprints is shown in Table 16.

Table 16: Chains Classification Accuracy

Validation Chain	Resource Utilization Fingerprint			
	CPU	**Memory**	**Tx**	**Rx**
Chain 1	92.8%	89.4%	96.4%	91.8%
Chain 2	98.4%	94%	97.1%	97.6%

To evaluate our choice of stacking resource utilization fingerprints in Section 7.2, we examined the classification accuracy on a per-function level using each function's individual fingerprints. The results are shown in Tables 17 and 18. The per-function's classification accuracy underperformed that of the stacked fingerprints. Since functions have shorter execution durations, per-function fingerprints have shorter resource utilization records. We believe this causes the per-function fingerprints to be lower resolution than the stacked ones, affecting accuracy [80].

Table 17: Chain-1 Per-Function Classification Accuracy

Validation Function	Resource Utilization Fingerprint			
	CPU	**Memory**	**Tx**	**Rx**
Chain 1 -Function 1	73.1%	83.4%	96%	95.9%
Chain 1 -Function 2	78.2%	85.7%	97.7%	98.7%
Chain 1 -Function 3	78.8%	83.1%	90%	95.8%

Table 18: Chain-2 Per-Function Classification Accuracy

Validation Function	Resource Utilization Fingerprint			
	CPU	**Memory**	**Tx**	**Rx**
Chain 2 -Function 1	69.4%	85.8%	93%	93.9%
Chain 2 -Function 2	58.2%	56.7%	88.5%	92.5%
Chain 2 -Function 3	87.8%	64.9%	91.4%	87.8%
Chain 2 -Function 4	65.4%	79.2%	88.6%	62%
Chain 2 -Function 5	80.1%	72.4%	91.3%	92%
Chain 2 -Function 6	80.7%	83.5%	90.6%	91%

7.3.2 Variable-size Sequential Serverless chains

To examine our approach's accuracy in variable-sized sequential serverless, we implemented Chain 3 as a map-reduce workflow of variable length based on the map-reduce proposed architecture by Alventosa et al. [81]. The chain comprises three distinct functions: i) a coordinator, ii) a set of mappers, and iii)a set of reducers. The chain's objective is to produce a daily AM/PM count of drive records in Microsoft's published T-Drive Taxi trajectory data [82]. The concurrency of the mapper function varies based on the input data size, where a new mapper is allocated for each 1KB chunk of the chain's input. For example, a 10KB input to the chain will result in 10 mapper instances. The chain receives one of the 11 thousand files in Microsoft's dataset as input, splits the file into equally sized 1KB chunks, generates the AM/PM count per chunk in the mapping phase, and then reduces to generate the total AM/PM count in the final phase. Chain 3's workflow is illustrated in Figure 24.

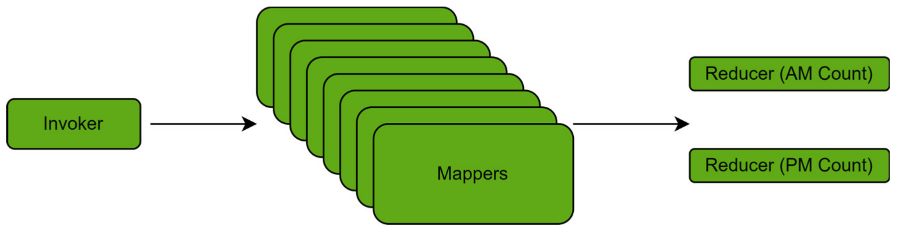


Figure 24: Variable-size (Chain 3) workflow

The number of mappers varies based on the size of the input file and consequently the number of generated 1KB chunks. In our validation, we specified the upper limit of mapper instances to 10 and executed 1000 SLA-abiding and SLA non-abiding executions of the chain. We trained our framework using the produced resource utilization fingerprints and evaluated the classification accuracy on a validation dataset of 20,000 executions of the chain. We maintain a 20:1 validation-to-training samples ratio. The accuracy of classification per resource utilization fingerprint is shown in Table 19.

Table 19: Chain3 Classification Accuracy

Validation Chain	Resource Utilization Fingerprint			
	CPU	Memory	Tx	Rx
Chain 3	79.3%	84.8%	92.7%	82.4%

7.3.3 Choice of the classifier

In this work, we have chosen to utilize CNNs for fingerprint classifications in FaaS2F as this particular NN architecture has an established reputation in the image classification domain. While a broad range of techniques can be used for time series classification, such as autoregressive and hidden Markov models [83]–[85], Dynamic Time Warping (DTW) [86], and Autoencoders [87], CNNs have proven to provide high accuracy end-to-end time series classification solutions [83], [88]. A significant advantage of CNNs is their ability to detect features to classify upon, thus alleviating the need for a human-based feature engineering; that is impractical in the serverless SLA scenario. The convolution layer in CNNs applies a common set of weights to different regions of the input, thus can efficiently detect the presense of classifying features anywhere in the input. While CNNs have proven excellent accuracy in our empirical validation, we hypothesize that the problem at hand is a generic univariate time series classification problem that other classification techniques can solve. Thus, we evaluated Random Forests' accuracy (RFs) accuracy [89], another ML technique commonly used in classification problems to categorize fingerprints into SLA abiding and non-abiding. We trained a generic RF-classifier composed of 100 trees. We evaluated the accuracy of classifying SLA violations in chains 1, 2, and 3 and used the same number of training samples in training and the training duration of both techniques.

Table 20: Chains' RF-based Classification Accuracy

Validation Chain	**Resource Utilization Fingerprint**			
	CPU	**Memory**	**Tx**	**Rx**
Chain 1	99.5%	98.5%	99.3%	99.6%
Chain 2	98.3%	92%	96.6%	96.3%
Chain 3	91.75%	92.3%	91.2%	91.4%

As illustrated in Table 20, RFs generated an excellent accuracy in all of the fingerprints classification. A considerable advantage of using RFs is that they require less training time than CNNs. In our empirical validation, CNNs took approximately 83% more time to train than RFs for the same set of fingerprints. While producing better results than CNNs in classifying our validation fingerprints, we believe that the performance of RFs may not be superior to NNs for every serverless chain's fingerprints. Thus, other ML classification techniques may be used interchangeably depending on the specifics of a chain's fingerprints. The choice is left to the serverless provider to choose an ML classifier that produces an acceptable tradeoff between classification accuracy and training time.

Chapter 8 Conclusions, Limitations, and Future Work

8.1 Conclusion

Since its inception, the FaaS delivery model has been celebrated as a breakthrough in cloud delivery. Leveraging its high abstraction, developers could quickly adopt the model in their applications with no administration overheads. The pay-upon-execution billing model also became favorable as it limited the upfront investment to an absolute minimum. The model's precise tailoring towards ephemeral, short-lived applications makes it attractive to workloads that would otherwise incur higher costs and limited scaling capacity in other cloud delivery models. Despite such advancements, the model's unique abstraction affects the transparency of serverless functions executions and obfuscates the performance-impacting factors from end-users. This transparency is crucial in the serverless model, where execution performance directly impacts incurred costs.

In this work, we examined serverless performance and identified the factors that affect the execution performance in the serverless model. We illustrated how seemingly trivial choices in skeleton containers or execution engines could highly impact the execution performance of serverless functions. We also demonstrated the impact of the providers' ecosystem performance on serverless functions. In the commercial landscape, we examined the variations in execution performance and execution consistency of commercial serverless platforms and how they can impact the users' incurred costs. The results of our examinations led us to perform a deep investigation into defining performance guarantees for serverless functions that can protect users from incurring costs as a result of performance degradations on the provider's end.

We illustrated how the unique abstraction of the serverless model hinders the use of the traditional IaaS or PaaS SLA definition metrics that commonly leverage resource guarantees for SLA definitions. We thus proposed leveraging resource utilization metrics to uniquely model the execution performance of serverless functions. We proposed a framework that utilizes machine learning to detect, with high accuracy, performance variations in serverless functions' executions based on these fingerprints. The framework can be leveraged to define binding customer-provider agreements on execution performance. We extended the framework to define performance guarantees for sequential serverless chains as an integral component for utilizing serverless functions to build larger applications.

Our proposed solution, FaaS2F, can be deployed by cloud providers as an add-on to their platform, thus enabling the articulation of binding SLAs. The framework works natively with the Kubernetes-based serverless deployment, *Knative*, and utilizes Google's *Cadvisor* for resource utilization metrics collection and *InfluxDB* and *MySQL* for metrics storage. Providers can apply customization to the framework to adopt custom metrics-collection agents or other serverless frameworks. Also, other ML classification techniques can be employed to detect SLA violations.

While improving the serverless model, serverless SLAs may only be appropriate in particular scenarios and not for every serverless function. The reason for that is the computational overhead incurred for training the serverless framework, the ongoing monitoring of a function's execution, and detecting SLA violations. These overheads could be computationally comparable to the monitored serverless functions themselves and thus can be more justifiable in some scenarios than others, for example :

1. Functions with an expected large number of executions:
    - The modeling of execution performance and the training of FaaS2F to recognize SLA violations impose the overhead of collecting training fingerprints. In our validation, the number of training fingerprints ranged from a few hundred to one thousand samples. The generation of these fingerprints is justifiable when a serverless function is expected to have a large number of executions (thousands or more). This overhead may not be justifiable for functions with few executions. For example, a function that is executed once daily may not justify defining an SLA for.
2. Functions with longer execution durations:
    - As execution durations translate into incurred costs in the serverless model, SLA definition becomes more beneficial as the execution durations

increase. For example, a serverless function that finishes execution in 1ms will still be charged for 100ms billing duration, i.e., the minimum billing increment, in most commercial serverless platforms. Thus, for such a function, the computational overhead of the SLA framework may not be justifiable. Hence, serverless providers may provide separate server pools, with no SLA guarantees, for functions whose execution durations fall below the minimum billing increment at a much cheaper rate to end-users.

3. Serverless chains:
    - Serverless functions that are parts of chains are better candidates for SLA definitions than stand-alone functions. As a single function's sub-optimal performance can impact the entire chain's performance, SLA for chains can be higher rewarding and more impactful than for stand-alone functions.

8.2 Limitations and Future Work

While the validation results of our proposal are very promising, we believe our work can be extended to address the following limitations:

1. Training data variability
    - In this work, we hypothesized that the samples in FaaS2F's training phase are generalizations of a function's input payload during its production phase. While this assumption has proven to be true in our empirical evaluation datasets[78], [79], [82], we did not provide a mechanism to validate this assumption. In theory, an end-user with malicious intent can execute an ML attack using a crafted training dataset, thus rendering the classification phase unreliable. We hope to examine this further in future continuations of this work.
2. Persistent serverless functions
    - While the serverless model mandated the ephemeral, compute-only nature of serverless functions, some researchers have proposed persistent serverless functions for specific workloads [35]. Ephemeral serverless functions do not hold states between subsequent executions; on the other

hand, persistent functions rely on pre-defined hints to share states between multiple executions. Workflows of persistent serverless functions can thus exhibit high dependency on local filesystems and hence local disk input/output operations and less dependency on network IOPs. While CPU and memory utilization fingerprints may be able to capture some of an execution's performance metrics, local disk IOPs utilization metrics must be incorporated in the modeling of SLA for persistent serverless functions. We plan on examining this point further in future continuations of this work.

3. Serverless chains workflows:
    - We limited the scope of our validation to serverless functions with sequential workflows. Needless to say, the workflows of serverless chains are more diverse and can be of higher complexity as larger applications start to adopt the serverless model. We plan to examine the performance of FaaS2F, and any potential modifications to it, to support more complex workflows in the continuations of this work.
4. Performance improvement guidance:
    - The proposed solution provides a binary classification of executions in SLA-abiding and SLA non-abiding. While this is acceptable for SLA compliance assessment, we hope to extend the framework to guide the serverless providers on methods to improve their platforms based on the historical trends of SLA compliance.
5. Integration with serverless providers
    - As a modular and open-source system, serverless providers can choose how to integrate FaaS2F into their platforms, including collection agents' technological choices, storage technologies, and machine learning classifiers. The customization overhead could present a barrier to some cloud operators who do not have the required technical knowledge to customize the framework to their needs. We hope to extend FaaS2F in the future into an SLA-as-a-Service model that providers can adopt into their platforms and leverage when needed. The SLA-aaS will impose no

overhead of customizing the framework to integrate with the provider’s serverless platform and thus facilitate providers' adoption.

References or Bibliography (if any)

[1] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud Coimputing: state-of-the-art and research challenges," *J Internet Serv, Springer Verlag*, 2010.

[2] Z. Xiao, J. Jiang, Y. Zhu, Z. Ming, S. Zhong, and S. Cai, "A solution of dynamic VMs placement problem for energy consumption optimization based on evolutionary game theory," *J. Syst. Softw.*, vol. 101, pp. 260–272, 2015.

[3] T. Setzer and A. Wolke, "Virtual machine re-assignment considering migration overhead," in *Proceedings of the 2012 IEEE Network Operations and Management Symposium, NOMS 2012*, 2012, pp. 631–634.

[4] A. Wolke, B. Tsend-Ayush, C. Pfeiffer, and M. Bichler, "More than bin packing: Dynamic resource allocation strategies in cloud data centers," *Inf. Syst.*, vol. 52, pp. 83–95, 2015.

[5] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Comput. Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.

[6] A. Beloglazov and R. Buyya, "Energy efficient resource management in virtualized cloud data centers," in *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, 2010, pp. 826–831.

[7] A. Khosravi, A. Nadjaran Toosi, and R. Buyya, "Online virtual machine migration for renewable energy usage maximization in geographically distributed cloud data centers," *Concurr. Comput.* , vol. 29, no. 18, 2017.

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

[9] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Comput.*, vol. 2, no. 3, pp. 24–31, May 2015.

[10] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, 2014, pp. 610–614.

[11] E. Jonas *et al.*, "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv*, vol. abs/1902.0, 2019.

[12] S. K. Mohanty, G. Premsankar, and M. Di Francesco, “An evaluation of open source serverless computing frameworks,” in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2018, vol. 2018-Decem, pp. 115–120.

[13] “Serverless computation with openLambda | Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing.” [Online]. Available: https://dl.acm.org/doi/10.5555/3027041.3027047. [Accessed: 01-Jun-2021].

[14] J. M. Hellerstein *et al.*, “Serverless Computing: One Step Forward, Two Steps Back,” *arXiv*, Dec. 2018.

[15] P. Garcia Lopez, M. Sanchez-Artigas, G. Paris, D. Barcelona Pons, A. Ruiz Ollobarren, and D. Arroyo Pinto, “Comparison of FaaS orchestration systems,” in *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, 2019, pp. 109–114.

[16] I. Baldini *et al.*, “The serverless trilemma: Function composition for serverless computing,” in *Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2017*, 2017, pp. 89–103.

[17] G. C. Fox *et al.*, “First international workshop on serverless computing (WoSC) 2017: Report from workshop and panel on the status of serverless computing and function-as-a-service (FaaS) in industry and research,” *arXiv*, 2017.

[18] S. Mohanty, “Evaluation of Serverless Computing Frameworks Based on Kubernetes,” *Aalto Univ. Sch. Sci. Degree Program. Comput. Sci. Eng.*, 2018.

[19] M. Sadaqat, R. Colomo-Palacios, and L. E. S. Knudsen, “Serverless computing: a multivocal literature review,” *NOKOBIT - Nor. Konf. Organ. bruk av informasjonsteknologi*, vol. 26, no. 1, 2018.

[20] R. Buyya *et al.*, “A manifesto for future generation cloud computing: Research directions for the next decade,” *ACM Comput. Surv.*, vol. 51, no. 5, 2019.

[21] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, “A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms,” in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2017, vol. 2017-Decem, pp. 162–169.

[22] H. Lee, K. Satyam, and G. Fox, “Evaluation of Production Serverless Computing Environments,” in *IEEE International Conference on Cloud Computing, CLOUD*, 2018, vol. 2018-July, pp. 442–450.

[23] E. Oakes *et al.*, “SOCK: Rapid task provisioning with serverless-optimized containers,” in *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*, 2020, pp. 57–69.

[24] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless computing: An investigation of factors influencing microservice performance,” in *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 2018, pp. 159–169.

[25] D. Jackson and G. Clynch, “An investigation of the impact of language runtime on the performance and cost of serverless functions,” in *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, 2019, pp. 154–160.

[26] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, A. Y. Zomaya, and R. Jurdak, “Dynamic Control of CPU Usage in a Lambda Platform,” in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2018, vol. 2018-Septe, pp. 234–244.

[27] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, “Formal Foundations of Serverless Computing,” *arXiv*, 2019.

[28] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro, *No more, no less: A formal model for serverless computing*. 2019.

[29] S. Hong, A. Srivastava, W. Shambrook, and T. Dumitras, “Go serverless: Securing cloud via serverless design patterns,” *10th USENIX Work. Hot Top. Cloud Comput. HotCloud 2018, co-located with USENIX ATC 2018*, 2018.

[30] F. Hafeez, P. Nasirifard, and H. A. Jacobsen, *Demo abstract: A serverless approach to publish/subscribe systems*. New York, NY, USA: ACM, 2018, pp. 9–10.

[31] A. Pérez, M. Caballer, G. Moltó, and A. Calatrava, “A programming model and middleware for high throughput serverless computing applications,” in *Proceedings of the ACM Symposium on Applied Computing*, 2019, vol. Part F1477, pp. 106–113.

[32] L. H. Hung, D. Kumanov, X. Niu, W. Lloyd, and K. Y. Yeung, “Rapid RNA sequencing data analysis using serverless computing,” *bioRxiv*, p. 576199, Jan. 2019.

[33] A. Aytekin and M. Johansson, *Harnessing the power of serverless runtimes for large-scale optimization*. 2019.

[34] R. F. Hussain, M. A. Salehi, and O. Semiari, *Serverless edge computing for green oil and gas industry*. 2019.

[35] J. Schleier-Smith, “Serverless Foundations for Elastic Database Systems,” in *Cidr*, 2019, vol. 41, no. 2014, p. 2017.

[36] M. Elsakhawy and M. Bauer, “An Investigation into the Usage-trends of Canada’s Research Computing Clouds,” in *Proceedings - 4th IEEE International Conference on Smart Cloud, SmartCloud 2019 and 3rd International Symposium on Reinforcement Learning, ISRL 2019*, 2019, pp. 1–6.

[37] “Introduction - CERN OpenStack Private Cloud Guide.” [Online]. Available: https://clouddocs.web.cern.ch/details/quotas.html. [Accessed: 29-Apr-2021].

[38] “Availability and quota | Nectar Research Cloud tutorials.” [Online]. Available: https://tutorials.rc.nectar.org.au/advanced-networking/02-availability-and-quota. [Accessed: 29-Apr-2021].

[39] “Resource Allocation Competitions vs. Rapid Access Service: How to Choose? | Compute Canada.” [Online]. Available: https://www.computecanada.ca/research-portal/accessing-resources/rac-vs-ras/. [Accessed: 29-Apr-2021].

[40] J. Bottum *et al.*, “The Future of Cloud for Academic Research Computing,” *Results an NSF-Supported Work. Entitled “Cloud Forward,”* 2017.

[41] R. P. Taylor *et al.*, “The evolution of cloud computing in ATLAS,” in *Journal of Physics: Conference Series*, 2015, vol. 664, no. 2, p. 022038.

[42] R. P. Taylor *et al.*, “Consolidation of cloud computing in ATLAS,” in *Journal of Physics: Conference Series*, 2017, vol. 898, no. 5, p. 052008.

[43] S. Panitkin *et al.*, “ATLAS cloud R&D,” in *Journal of Physics: Conference Series*, 2014, vol. 513, no. TRACK 6, p. 062037.

[44] F. Harald Barreiro Megino *et al.*, “Exploiting virtualization and cloud computing in ATLAS,” in *Journal of Physics: Conference Series*, 2012, vol. 396, no. PART 3, p. 032011.

[45] G. Aad *et al.*, “The ATLAS experiment at the CERN large hadron collider,” *J. Instrum.*, vol.

3, no. 8, p. 8003, 2008.

[46] B. Posey, C. Gropp, A. Herzog, and A. Apon, “Automated Cluster Provisioning And Workflow. Management for Parallel Scientific Applications in the Cloud,” 2017.

[47] “Compute Canada,” 2016. [Online]. Available: https://www.computecanada.ca/research-portal/national-services/compute-canada-cloud/. [Accessed: 29-Apr-2021].

[48] S. Toor *et al.*, “SNIC Science Cloud (SSC): A national-scale cloud infrastructure for Swedish academia,” in *Proceedings - 13th IEEE International Conference on eScience, eScience 2017*, 2017, pp. 219–227.

[49] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien, “Real-time Serverless: Enabling application performance guarantees,” in *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*, 2019, pp. 1–6.

[50] E. Hunhoff, S. Irshad, V. Thurimella, A. Tariq, and E. Rozner, “Proactive Serverless Function Resource Management,” *WOSC 2020 - Proc. 2020 6th Int. Work. Serverless Comput. Part Middlew. 2020*, pp. 61–66, 2020.

[51] A. Suresh and A. Gandhi, “FNSched: An efficient scheduler for serverless functions,” in *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*, 2019, pp. 19–24.

[52] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, “Sequoia: Enabling quality-of-service in serverless computing,” in *SoCC 2020 - Proceedings of the 2020 ACM Symposium on Cloud Computing*, 2020, pp. 311–327.

[53] M. Tiwary, P. Mishra, S. Jain, and D. Puthal, “Data Aware Web-Assembly Function Placement,” in *The Web Conference 2020 - Companion of the World Wide Web Conference, WWW 2020*, 2020, pp. 4–5.

[54] S. Eismann *et al.*, “Serverless Applications: Why, When, and How?,” *IEEE Softw.*, vol. 38, no. 1, pp. 32–39, 2021.

[55] H. Martins, F. Araujo, and P. R. da Cunha, “Benchmarking Serverless Computing Platforms,” *J. Grid Comput.*, vol. 18, no. 4, pp. 691–709, 2020.

[56] Z. Kozhirbayev and R. O. Sinnott, “A performance comparison of container-based technologies for the Cloud,” *Futur. Gener. Comput. Syst.*, vol. 68, pp. 175–182, Mar. 2017.

[57] V. Tarasov *et al.*, “In search of the ideal storage configuration for docker containers,” in *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017*, 2017, pp. 199–206.

[58] M. Elsakhawy, Mohamed and Bauer, “Performance Analysis of Serverless Execution Environments,” in *3rd International Conference on Electrical, Communication and Computer Engineering*.

[59] S. Ginzburg and M. J. Freedman, “Serverless Isn’t Server-Less: Measuring and Exploiting Resource Variability on Cloud FaaS Platforms,” in *WOSC 2020 - Proceedings of the 2020 6th International Workshop on Serverless Computing, Part of Middleware 2020*, 2020, pp. 43–48.

[60] T. Le Duc, R. G. Leiva, P. Casari, and P. O. Östberg, “Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey,” *ACM Comput. Surv.*, vol. 52, no. 5, 2019.

[61] A. Saeed *et al.*, “An Optimal Utilization of Cloud Resources using Adaptive Back Propagation Neural Network and Multi-Level Priority Queue Scheduling $,” *ISC Int. J. Inf. Secur.*, vol. 11, no. 3, pp. 145–151, Aug. 2019.

[62] F. Liu, P. Wang, and W. Zhang, “Research on Resource Allocation based on RBF Neural Network Under Cloud Computing,” 2017, pp. 698–701.

[63] M. T. Imam, S. F. Miskhat, R. M. Rahman, and M. A. Amin, “Neural network and regression based processor load prediction for efficient scaling of grid and cloud resources,” in *14th International Conference on Computer and Information Technology, ICCIT 2011*, 2011, pp. 333–338.

[64] B. R. Ray and S. Chowdhury, “Reverse Engineering Technique (RET) to Predict Resource Allocation in a Google Cloud System,” in *Proceedings of the 8th International Conference Confluence 2018 on Cloud Computing, Data Science and Engineering, Confluence 2018*, 2018, pp. 688–693.

[65] J. Zhang, N. Xie, X. Zhang, K. Yue, W. Li, and D. Kumar, “Machine learning based resource allocation of cloud computing in auction,” *Comput. Mater. Contin.*, vol. 56, no. 1, pp. 123–135, 2018.

[66] R. A. Hemmat and A. Hafid, “SLA Violation Prediction In Cloud Computing: A Machine Learning Perspective,” *ArXiv*, vol. abs/1611.1, 2016.

[67] C. B. Hauser and S. Wesner, “Reviewing Cloud Monitoring: Towards Cloud Resource Profiling,” in *IEEE International Conference on Cloud Computing, CLOUD*, 2018, vol. 2018-July, pp. 678–685.

[68] A. Liaw and M. Wiener, “Classification and Regression by randomForest,” vol. 2, no. 3, 2002.

[69] R. Vaitheeshwari and V. Sathieshkumar, “Performance analysis of epileptic seizure detection system using neural network approach,” *ICCIDS 2019 - 2nd Int. Conf. Comput. Intell. Data Sci. Proc.*, Feb. 2019.

[70] C. Shorten and T. M. Khoshgoftaar, “A survey on Image Data Augmentation for Deep Learning,” *J. Big Data*, vol. 6, no. 1, pp. 1–48, Dec. 2019.

[71] A. F. Vermeulen, “Unsupervised Learning: Deep Learning,” *Ind. Mach. Learn.*, pp. 225–241, 2020.

[72] M. E. Nilsback and A. Zisserman, “A visual vocabulary for flower classification,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2, pp. 1447–1454, 2006.

[73] D. P. Kingma and J. L. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[74] V. Sreekanti *et al.*, “Cloudburst: Stateful functionsasaservice,” *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2438–2452, 2020.

[75] N. Somu, N. Daw, U. Bellur, and P. Kulkarni, “PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications,” in *2020 International Conference on COMmunication Systems and NETworkS, COMSNETS 2020*, 2020, pp. 144–151.

[76] M. Elsakhawy and M. Bauer, “FaaS2F: A framework for defining execution-sla in serverless computing,” in *Proceedings - 2020 IEEE Cloud Summit, Cloud Summit 2020*, 2020, pp. 58–65.

[77] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[78] T. Y. Lin *et al.*, “Microsoft COCO: Common objects in context,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture*

Notes in Bioinformatics), 2014, vol. 8693 LNCS, no. PART 5, pp. 740–755.

[79] M. Monfort *et al.*, “Moments in time dataset: One million videos for event understanding,” *arXiv*, vol. 42, no. 2, pp. 502–508, Feb. 2018.

[80] Y. Pei, Y. Huang, Q. Zou, X. Zhang, and S. Wang, “Effects of Image Degradation and Degradation Removal to CNN-Based Image Classification,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 4, pp. 1239–1253, Apr. 2021.

[81] V. Giménez-Alventosa, G. Moltó, and M. Caballer, “A framework and a performance assessment for serverless MapReduce on AWS Lambda,” *Futur. Gener. Comput. Syst.*, vol. 97, pp. 259–274, 2019.

[82] J. Yuan *et al.*, “T-Drive: Driving Directions Based on Taxi Trajectories.” 01-Nov-2010.

[83] B. Zhao, H. Lu, S. Chen, J. Liu, and D. Wu, “Convolutional neural networks for time series classification,” *J. Syst. Eng. Electron.*, vol. 28, no. 1, pp. 162–169, Feb. 2017.

[84] M. Corduas and D. Piccolo, “Time series clustering and classification by the autoregressive metric,” *Comput. Stat. Data Anal.*, vol. 52, no. 4, pp. 1860–1872, Jan. 2008.

[85] B. Esmael, A. Arnaout, R. K. Fruhwirth, and G. Thonhauser, “Improving time series classification using Hidden Markov Models,” *Proc. 2012 12th Int. Conf. Hybrid Intell. Syst. HIS 2012*, pp. 502–507, 2012.

[86] M. Müller, “Dynamic Time Warping,” *Inf. Retr. Music Motion*, pp. 69–84, 2007.

[87] T. Kieu, B. Yang, C. Guo, and C. S. Jensen, “Outlier Detection for Time Series with Recurrent Autoencoder Ensembles.”

[88] H. I. Fawaz, · Germain Forestier, J. Weber, · Lhassane Idoumghar, and P.-A. Muller, “Deep learning for time series classification: a review.”

[89] A. Liaw and M. Wiener, “Classification and Regression by randomForest,” vol. 2, no. 3, 2002.

Curriculum Vitae

Name:	Mohamed Elsakhawy
Education:	
2022	University of Western Ontario London, Ontario, Canada Ph.D in Computer Science
2011	University of Western Ontario London, Ontario, Canada M.Sc. in Electrical and Computer Engineering
2007	Alexandria University Alexandria, Egypt B.Sc. in Communications and Electronics Engineering

Publications:

1. M.Elsakhawy and M.Bauer ***"SLA for Sequential Serverless Chains: A Machine Learning Approach ",*** International Workshop on Serverless Computing, Part of ACM/IFIP International Middleware Conference (**WoSC '21**) Accepted, in-press

2. M.Elsakhawy and M.Bauer ***"Usage Trends Aware VM Placement in Academic Research Computing Clouds"***, IEEE International Conference on Cloud Computing, (**CLOUD '21**), Chicago, IL, USA, 2021 pp. 688-697.

3. M. Elsakhawy and M. Bauer, ***"Performance Analysis of Serverless Execution Environments"***, International Conference on Electrical, Communication, and Computer Engineering (**ICECCE '21**), Kuala Lumpur, Malaysia, 2021, pp. 1-6.

4. M. Elsakhawy and M. Bauer, ***"FaaS2F: A Framework for Defining Execution-SLA in Serverless Computing"*** ,IEEE Cloud Summit (**CloudSummit '20**), Harrisburg, PA, USA, 2020, pp. 58-65.

5. M. Elsakhawy and M. Bauer, ***"An Investigation into the Usage-trends of Canada's Research Computing Clouds",*** IEEE International Conference on Smart Cloud (SmartCloud '19), Tokyo, Japan, 2019, pp. 1-6.

6. S. Moursi, M. Elsakhawy , and H. Ghenniwa, ***"Agent oriented media recommender system utilizing Smart multimedia,"*** International Conference on Computer Supported Cooperative Work in Design **(CSCWD '11)**, Laussane, Switzerland, 2011, pp. 437-444.