Boston University

OpenBU **http://open.bu.edu**

Boston University Theses & Dissertations Boston University Theses & Dissertations

2023

Orchestrating cloud resources to optimize performance and cost

https://hdl.handle.net/2144/49670

Downloaded from DSpace Repository, DSpace Institution's institutional repository

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

ORCHESTRATING CLOUD RESOURCES TO OPTIMIZE PERFORMANCE AND COST

by

ALI RAZA

B.S., Lahore University of Management Sciences, 2010

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2023

© 2023 by
ALI RAZA
All rights reserved

Approved by

First Reader

Abraham Matta, PhD
Professor of Computer Science

Second Reader

Vatche Isahagian, PhD
Senior Research Scientist and Manager
IBM Research AI

To my parents and my first teacher (my grandfather) ...

امی ، ابو اور دادا جی کے نام ۔۔۔

Acknowledgments

First of all, I want to express my gratitude to my advisor, Abraham Matta, for being an amazing advisor, mentor, and trusted friend throughout my Ph.D. journey. I am forever indebted to him for his constant support and encouragement.

I would also like to extend my appreciation to my collaborators, Nabeel Akhtar and Vatche Isahagian, who were always there to guide me. Their support and expertise played an important role in shaping my research.

I am also indebted to my undergraduate teachers Dr. Fareed Zaffar and Dr. Ihsan Ayyub Qazi for their constant support and mentorship during my undergraduate degree and after. During my time at the New York University – Abu Dhabi, I was fortunate to work with amazing mentors and colleagues Yasir Zaki and Thomas Pötsch. The skills and lessons learned during that time helped me a lot during the course of my Ph.D. degree.

Furthermore, I am blessed to have a wonderful group of friends, both at Boston University and afar, who stood by me every step of my Ph.D. journey. I am always grateful to Bushra Ali, Ahsan Abdullah, Munaf Arshad Qazi, and Aqeel Ahmad Yousfzai for their support and encouragement. Their presence in my life has made all the difference.

I also want to express my appreciation to my roommates, Sarah Williford, Carson Williford, Aashraya Jha, and Max Heldman, for creating a warm and welcoming home away from home for me here. I will always cherish the memories we shared and will miss them dearly.

Lastly, I want to thank my family, especially my parents, and sisters, for being a constant source of joy and encouragement. Their unconditional support and love have been my rock throughout my journey, and I am forever grateful for them.

ORCHESTRATING CLOUD RESOURCES TO OPTIMIZE PERFORMANCE AND COST

ALI RAZA

Boston University, Graduate School of Arts and Sciences, 2023

Major Professor: Abraham Matta, PhD
Professor of Computer Science

ABSTRACT

In the last decade, Function-as-a-Service (FaaS) became one of the popular choices for building and deploying cloud applications. Compared to Infrastructure-as-a-Service (IaaS), FaaS offers an abstraction of backend management, an easy programming model, low cold starts, and a true "pay as you go" pricing model. While efficient and relatively simpler, the cost and performance of an application, when deployed using FaaS, can be adversely affected if not properly managed and configured. Previous approaches have advocated the *limited* use of FaaS while scaling out the virtual machine (VM) based resources to avoid Service-Level-Objective (SLO) violations. However, these approaches miss out on potential long-term cost savings by employing FaaS consistently. Similarly, to manage a FaaS deployment, various machine learning and optimization techniques have been suggested but these techniques either have high costs or fall short as they fail to adapt to the dynamic nature of FaaS platforms. To this end, we present Thrifty, a hybrid approach to leveraging FaaS in conjunction with other cloud services to optimize both cost and performance. Thrifty consists of two main components: 1) LIBRA: a load-balancing framework to utilize IaaS and FaaS resources efficiently. Based on the demand, it decides to use either FaaS, IaaS,

or both to maximize cost savings while meeting the SLOs; 2) xCOSE: a resource configuration and placement technique for FaaS deployments. It addresses the performance variability of FaaS platforms and meets the SLO by adapting the resource configurations with minimal sampling cost. It can configure single- and multi-function (service graph) applications.

We evaluate Thrifty in extensive simulations and on the Amazon Web Services (AWS) cloud platform using real applications. Our evaluations show that consistent and opportunistic usage of FaaS through LIBRA can reduce SLO violations by up to 85% and cost by up to 53% when compared to other approaches to deploying cloud applications. Furthermore, xCOSE has the ability to configure simple and complex FaaS applications with minimal sampling cost.

Contents

1 Introduction **1**

1.1 Cloud Services 2

1.2 Developer's Perspective & Challenges 5

1.2.1 Picking a service 5

1.2.2 Resource and configuration selection 6

1.2.3 Managing an application 6

1.3 Hybrid Clouds 7

1.3.1 Cost and Performance 8

1.3.2 Availability 9

1.3.3 Unified Development Model 9

1.4 Thesis Contributions 10

1.5 Roadmap of thesis 12

2 Background & Related Work **13**

2.1 FaaS 13

2.1.1 Developer's View of FaaS 17

2.1.2 Performance & Cost 18

2.1.3 FaaS Advantages 20

2.2 Related Work 21

2.2.1 FaaSification of Application 22

2.2.2 FaaS Usage 22

2.2.3 Measurement & Benchmarking 23

2.2.4 FaaS Application Management . . . 25
2.2.5 FaaS Usage in Hybrid Cloud . . . 26

3 Thrifty **28**

4 LIBRA Motivation **33**
4.0.1 FaaS Pricing Model . . . 33
4.0.2 IaaS: VM Pricing Model . . . 35
4.0.3 Cost Analysis . . . 35
4.0.4 FaaS + IaaS . . . 38

5 LIBRA Architecture **40**
5.1 Architecture . . . 41
5.1.1 Traffic Monitor . . . 41
5.1.2 Scaling Manager . . . 43
5.1.3 Load Balancer . . . 45
5.1.4 LIBRA Parameters . . . 46
5.2 How to deploy LIBRA? . . . 47

6 LIBRA Evaluation **49**
6.1 Simulation Model . . . 49
6.1.1 Modeling Cloud Services . . . 49
6.1.2 Simulation Parameters . . . 50
6.1.3 Log Traces . . . 51
6.1.4 Resource Provisioning & Deployment Policies . . . 51
6.2 Simulation Results . . . 53
6.2.1 Cost and SLO violations . . . 53
6.2.2 VM Provisioning & Request Distribution . . . 55
6.2.3 VM Uptime & Cost Breakdown . . . 56

6.2.4 Traffic Estimation 56
6.3 AWS Results 57
6.3.1 Implementation Details 58
6.3.2 Cost and SLO Violations 60
6.3.3 VM Uptime & Cost Breakdown 62
6.4 Discussion & Future Work 63
6.4.1 Development Overhead 63
6.4.2 Performance Overhead & LIBRA Scalability 63
6.4.3 Application Availability 63
6.4.4 Leveraging Spot Instances 64

7 xCOSE **66**

7.0.1 Ideal Configurator 67
7.1 Background: COSE 68
7.1.1 Performance Modeler 68
7.1.2 Config Finder 69
7.2 xCOSE 70
7.2.1 Config Finder 70

8 xCOSE Evaluation **73**

8.0.1 Image Processing workflow 73
8.0.2 Personal Protective Equipment 74
8.1 Other Techniques 77
8.1.1 Sizeless 77
8.1.2 Costless 80

9 Conclusions **81**

9.1 Summary 81
9.2 Future Research Directions 82

References 83

Curriculum Vitae 95

List of Tables

2.1 Popular commercial FaaS platforms 14

8.1 xCOSE and other configuration techniques 78

List of Figures

2·1 Performance and cost on AWS Lambda 18

3·1 Queuing Model 29

3·2 Effect SLO and β on optimal α^* 30

3·3 Thrifty Architecture 32

4·1 Cost comparison of Amazon Lambda and EC2 instances for varying average request arrival rate 34

4·2 Hybrid Case 39

5·1 LIBRA architecture 42

6·1 WITS trace and EWMA 52

6·2 LIBRA and other resource provisioning policies 54

6·3 VM provisioning and request distribution of LIBRA. 55

6·4 VM usage and cost breakdown 57

6·5 Cost for different values of ϕ 58

6·6 Performance of LIBRA on AWS 60

6·7 VM usage and cost breakdown 62

7·1 xCOSE Architecture 69

8·1 Image Processing Workflow 73

8·2 Function performance w.r.t. memory allocated 74

8·3 PPE Detection Application 75

8·4 Memory and location w.r.t. SLO 76
8·5 Cost ($) breakdown 77
8·6 xCOSE predicting the execution model 79

List of Abbreviations

AD	....	Active Directory
AIAD	....	Additive Increase Additive Decrease
ALB	....	Application Load Balancer
AWS	....	Amazon Web Services
BMaaS	....	Bare-Metal-as-a-Service
CDF	....	Cumulative Distribution Function
CDN	....	Content Delivery Network
CI/CD	....	Continuous Integration and Continuous Delivery
CIP	....	Cost Indifference Point
CPU	....	Central Processing Unit
CaaS	....	Container-as-a-Service
DAG	....	Directed Acyclic Graph
EC2	....	Elastic Compute Cloud
ECS	....	Elastic Compute Service
EWMA	....	Exponentially Weighted Moving Average
FaaS	....	Function-as-a-Service
GCE	....	Google Compute Engine
GCF	....	Google Cloud Function
HTTP	....	Hyper Text Transfer Protocol
IAM	....	Identity and Access Management
ILP	....	Integer Linear Programming
IaaS	....	Infrastructure-as-a-Service
LB	....	Load Balancer
ML	....	Machine Learning
MLaaS	....	Machine-Learning-as-a-Service
NLP	....	Natural Language Processing
NN	....	Neural Network
PPE	....	Personal Protective Equipment
QoS	....	Quality of Service
RPC	....	Remote Procedure Call
S3	....	Simple Storage Service
SLO	....	Service-Level Objective

SM	....	Scaling Manager
SNS	....	Simple Notification Service
TM	....	Traffic Monitor
VMs	....	Virtual Machines
WITS	....	Waikato Internet Traffic Storage

Chapter 1

Introduction

The advent of virtualization revolutionized the way we run large-scale applications. Originally intended at increasing the utilization of hardware resources, virtualization led to some revolutionary techniques and products, i.e. Virtual Machines (VMs), containers, etc. VMs and containers allow running multiple applications and processes on the same hardware by providing virtual separation among them. Sharing resources can lead to better utilization of overall systems. Through such technologies, the ease of managing multi-tenancy, provisioning, and managing large-scale infrastructure and tenants gave rise to commercial cloud providers. In the last couple of decades, commercial cloud services became a viable option to deploy and manage large-scale cloud applications to the extent that companies rely entirely on these services for their computing needs for example, Netflix relies on Amazon Web Services (AWS) for nearly all of their computing infrastructure [39]. We are truly living in the future of computing that John McCarthy predicted in 1961 [38]:

"Computation may someday be organized as a public utility just as the telephone system is a public utility... Each subscriber needs to pay only for the capacity he actually uses ... Certain subscribers might offer service to other subscribers ... The computer utility could become the basis of a new and important industry." – MIT Technology Review – October 3, 2011

Indeed today, we are living in the future John McCarthy envisioned in 1961 and public clouds have become a critical part of modern computing needs. All the major technology companies such as Amazon, Google, IBM, Microsoft, and Alibaba, to name a few, have introduced their cloud offerings. Public clouds allow a developer to focus on building applications and shipping new features to users without them worrying about the underlying infrastructure management. Public clouds have led the democratization of computing power, where even small companies and individuals can access large-scale computing infrastructure without a large upfront cost. Some of the key reasons for the popularity and adoption of public clouds are cost-effectiveness, scalability, flexibility, and reliability. Public clouds usually charge their user for the precise amount of usage (called *pay as you go*) and invest heavily in the security of their services and user data. They allow easy scaling of resources in case of the application demand increases (often in the order of milliseconds) and guarantee the high availability of their services. Moreover, to deal with the plethora of options available from various cloud providers, we have broker systems (resource orchestrators) [128, 106, 77] that aggregate computing resources from diverse cloud providers for an application to improve its performance and cost of cloud usage.

1.1 Cloud Services

Today, cloud providers offer a diverse range of cloud services to application developers. These services ranging from storage and computing to identity management can cater to all the needs of a cloud application. Some examples of popular cloud services are (i) storage – e.g., Amazon Simple Storage Service (S3), Microsoft Azure Blob Storage, and Google Cloud Storage, which provide scalable, online storage for data and backups, (ii) computing – e.g., Amazon Elastic Compute Cloud (EC2), Microsoft Azure Virtual Machines, Google Compute Engine (GCE), and Alibaba Cloud Elastic

Compute Service (ECS), (iii) platforms to develop and deploy applications – e.g., Amazon Elastic Beanstalk, Microsoft Azure App Service, and Google App Engine, and (iv) security/compliance services – e.g., AWS's Identity and Access Management (IAM), Microsoft Azure Active Directory (AD), and Google Cloud Identity and Access Management (IAM). An application developer depending on her needs can use one or a combination of services to deploy her applications.

While a developer has to choose a combination of all of these services to deploy her application, in this thesis, we will focus on choosing/configuring the computing services for an application[1].

There is a diverse range of computing services offered by public clouds. These services offer varying pricing models, performance, and control to the application developer. An application developer, based on application requirements and budget constraints, chooses one or a combination of these services to run an application. In what follows we will discuss some of the popular types of computing services and their features (pricing, performance, management, and control abstraction).

Infrastructure-as-a-Service (IaaS): In this model, a cloud provider offers on-demand Virtual Machines (VMs) to an application developer. A user can choose various software aspects of the VM such as operating systems, language, etc. A user will be charged for the allocation time of the VM regardless of usage. Cloud providers also provide additional *scaling* services based on various performance metrics, i.e., adding or removing these VMs to improve performance and save cost. During times of low usage, a cloud provider can also decide to offer IaaS resources at heavily discounted prices with no guarantee of availability. These discounted instances are called *Spot Instances*. Examples of IaaS are EC2 from AWS [4], and Compute Engine from Google Cloud [26].

[1] *Henceforth the term "cloud service" or "service" would refer to a viable computing service to deploy/run an application.*

Bare-Metal-as-a-Service (BMaaS): In the BMaaS model, a provider may provide and deploy dedicated hardware infrastructure for the client, offering more flexibility in choosing a network, storage, and compute functionalities. Examples include IBM Cloud Bare Metal Servers [29], and EC2 bare metal from AWS [4].

Container-as-a-Service (CaaS): In the CaaS model, a user does not have to worry about the hardware and software aspects of the compute resources, which are managed by the cloud provider. In CaaS, a user still has to deploy scaling policies based on various performance metrics. Examples of CaaS are Amazon Elastic Container Service (ECS) [2], and Container Services from Microsoft Azure [16].

Function-as-a-Service (FaaS): FaaS[2] has emerged as a new computing service offered by all major cloud providers. In this model, all the infrastructure management, including scaling, is left to the provider. A user develops the application in a high-level language such as Python [105] and she is only charged precisely for the time the application is running, i.e., no idle cost. FaaS examples are AWS Lambda from AWS [4], Azure Functions from Microsoft Azure [18], and Google Cloud Function from Google Cloud [25].

Machine-Learning-as-a-Service (MLaaS): In this model, an organization instead of developing its own ML model, relies on external services. Examples of such services include pre-trained Natural Language Processing (NLP) models, image and video analysis, and speech recognition. MLaaS examples are Amazon SageMaker [12] from AWS, Microsoft Azure's machine learning service [19], and Google Cloud's AI Services [24].

Note that the above is not an exhaustive list of computing services, but is meant to merely provide a synopsis of popular services. In the next sections, we will discuss some of the challenges that a developer faces while employing these computing services

[2]We assume FaaS is a computing service provided by a serverless platform managed by a cloud provider.

to build and deploy her application.

1.2 Developer's Perspective & Challenges

From an application developer's perspective, oftentimes an application can be deployed using any of these services. For example, to deploy a Machine Learning (ML) inference model, a developer can train their own Neural Network (NN) and deploy it using IaaS (with an additional cost of backend management) or employ FaaS (at a higher cost but without the burden of backend management) or altogether offload the computing to an MLaaS and pay per query. Having multiple services and the ability to deploy the application in any of those services raises the natural questions:
How to choose a cloud service(s) to deploy an application to optimize cost and performance? How to configure a cloud service(s) to optimize cost and performance? How to react to changes in demand while still meeting the performance requirements?

In what follows we explain these challenges in detail and discuss previous work and approaches to address these challenges in Chapter 2.

1.2.1 Picking a service

Choosing a cloud service or a combination of services to run a cloud application is the first step for deployment and consequently can affect the application development, performance, and cost of cloud usage. A developer has to take into consideration the service's performance such as cold starts, both monetary and development costs (different services may follow varying application development/programming models) and scalability in the face of variable demand. There have been numerous studies done and measurement studies performed on public clouds to quantify their performance for different kinds of workloads. Moreover, there are benchmarking tools that can help an application developer make a more informed decision about choosing a service. Lastly, the advent of containerized applications has to some extent eliminated the

difference in application development as most cloud services can support and run the containerized application[3], reducing development costs if a developer chooses to change the underlying service running the application.

1.2.2 Resource and configuration selection

Even after choosing a service, a developer has to configure the particular service to achieve the expected service-level objectives (SLOs) and optimize cost. For example, AWS's IaaS service, EC2, offers a wide range of Virtual Machines (VMs) with varying configurations and pricing models. Similarly for FaaS, an application developer has to choose resources such as memory and CPU available for individual functions implementing the application. Previous studies [51, 48] have shown this configuration can affect the performance and cost significantly if not chosen carefully.

1.2.3 Managing an application

There are many aspects of managing a cloud application such as security, database management, continuous integration and continuous delivery (CI/CD), and adapting computing resources to changing demand. As our goal is to optimize computing resources, we will focus on the last challenge of adapting computing resources to changing demand of an application. Computing resources, if not adapted in response to demand, can lead to over/under-utilization of resources and can impact the cost and performance of a cloud application. Even though there are some computing services such as FaaS, which do not need additional configuration from an application developer to scale in the face of changes in demand, other services may require configuration and implementation of additional services to adapt. IaaS is one such example, where a developer may have to implement scaling policies.

[3]Though the deployment model of these containerized applications can differ from one provider to another.

1.3 Hybrid Clouds

There has been an increasing trend of mixing various services from diverse providers (called *hybrid clouds* [4]) to provide the computing infrastructure of an application. We believe hybrid solutions can be a step in the right direction as it allows the developer to pick from a range of services with unique performance and cost features. A hybrid cloud particularly offers the following features:

- **No vendor lock-in:** In the hybrid model, a developer utilizes resources from multiple providers and services, which offers them the flexibility to move their application across vendors easily if one provider is down or becomes expensive.
- **Cost:** A hybrid cloud allows developers to continually pick computing resources across providers/services, which allows them to leverage pricing models as well as discounts and free tiers.
- **Performance:** A key benefit of a hybrid cloud is leveraging the best-in-class performance features of a service. For example, FaaS offers quick provisioning time, and using it in conjunction with IaaS can reduce the SLO violations during the scaling up of IaaS resources as they can take up to minutes to be ready.
- **Availability:** By using a hybrid cloud, a developer can distribute its workloads across different providers/services, and in the event of an outage of one provider/service, the other providers/services can reduce the impact of that downtime.

Hybrid cloud usage can be classified into two broad categories: multi-cloud deployment, and cloud bursting.

[4] Also referred to as *Sky Computing [116]* and *Multi-Cloud [69]* in the literature.

Multi-Cloud Deployment: In this type of hybrid cloud deployment, different parts of an application are deployed in distinct cloud services or infrastructures, such as offloading the machine learning operations to an MLaaS provider and keeping the rest of the application in an IaaS provider. In this scenario, various parts of the application are glued together using APIs. The advantage of this approach is that one can take advantage of the best features of different cloud providers.

Cloud Bursting: In this type of hybrid cloud deployment, the same application is replicated across multiple cloud services or infrastructures, and depending on the demand, it is decided which deployment(s) to utilize. For example, if there is a sudden surge in demand, the application can burst into another cloud service or infrastructure to handle the extra load. This approach offers a more flexible and scalable solution to handle dynamic workloads. Another case of cloud bursting can also be replicating applications across multiple locations, which have different access latencies, and depending on the end-user latency requirements, different deployments can be used.

Both types of hybrid cloud deployments have their advantages and disadvantages, and the choice of deployment depends on the specific requirements of the application and business needs.

To leverage a hybrid cloud, practitioners usually rely on resource orchestrators, which given the computational needs of an application, pricing models of services/resources, and demand for an application, can optimally allocate resources across providers/services. While provisioning resources, an ideal resource orchestrator should address the following aspects:

1.3.1 Cost and Performance

The need for a hybrid cloud arises from the fact that a combination of services can yield either better performance for an application or reduce the cost of cloud usage

or both. For example, container-based services such as CaaS and FaaS have startup delays in the order of milliseconds and can scale up to thousands of application instances quickly to cater to the spikes in demand, thereby reducing SLO violations. Similarly, spot instances are greatly discounted and can lead to cost savings. FaaS can also be a cheaper alternative to IaaS when the demand is low, as in the FaaS pricing model, a cloud provider only charges for the precise amount of usage. An ideal resource orchestrator should have a resource provisioning policy that considers both performance and cost aspects.

1.3.2 Availability

In a hybrid cloud scenario, an application's computing needs are fulfilled by using different services from one or more providers, and these services may have different availability (and reliability) based on the service, provider regions, *etc.* For example, in the AWS case, EC2 instances have availability of at least 99.9%, AWS Lambda provides no such guarantee on availability or performance, and spot instances can be interrupted anytime and provide no guarantee on availability. When using a combination of such services, it is crucial to meet the availability requirements set by the application developer.

1.3.3 Unified Development Model

As different services and providers follow different development models, a hybrid cloud should simplify the development model for an application developer and expose only one development model regardless of the back-end service or provider being employed. It should also aggregate and expose all the performance metrics through one API. This will not only reduce the development cost for the developer but will increase the portability of the application and avoid vendor lock-in.

The primary objective of this thesis is to optimize the cost of cloud usage and improve application performance, without specifically addressing the challenges related to availability and a unified model. We mainly target FaaS and IaaS resources (for cloud bursting) to run an application while meeting SLOs cost-effectively.

1.4 Thesis Contributions

FaaS has recently gained popularity to deploy cloud applications [93, 72, 77, 81, 129, 120]. At its core, FaaS provides an easy and intuitive programming model to develop applications and ease the management and scaling of applications. In addition, FaaS offers certain performance and cost features. Previous approaches [77, 81, 128, 99] leverage FaaS's quick provisioning time feature to improve the performance of cloud applications running over IaaS during scaling events by offloading the *excess* demand to FaaS deployment of the application.

In this thesis, in addition to leveraging FaaS for quick surges in demand (like [77, 81, 128, 99]), we show that FaaS can be a cost-effective option for low-rate demand. We provide a load-balancing framework that combines FaaS's performance (quick provisioning and scalability) and cost-effectiveness (for low-rate demand) with the IaaS's long-term cost savings. Moreover, we present a statistical-learning based approach to configure the FaaS deployment contrary to previous approaches that assume static configurations which can lead to sub-optimal cost or performance.

Thesis Statement: FaaS allows its tenants to rent computing resources on much smaller time granularity (1ms or 100ms) with no idle cost unlike IaaS, where pricing granularity is much higher (1 second or hours), and a user is charged regardless of usage. Moreover, FaaS resources can be provisioned much quicker (in the order of milliseconds) compared to IaaS resources (which can take up to minutes). This particular pricing scheme and performance features of FaaS can be leveraged to optimize both the cost of cloud usage and the performance of a cloud application.

We make the following contributions in this thesis:

- There is an increased trend of mixing various cloud services to run an application to optimize cost or performance or both. Unlike previous work [77, 128, 99, 81] which leverages FaaS opportunistically, we present a detailed economic model of a cloud application when deployed using IaaS and FaaS given the demand. With the help of our analytical analysis, we show that FaaS is a more economical option when applications have low demand, and simultaneously using IaaS in conjunction with FaaS can significantly improve the cost of cloud usage.

- We present the design of our resource orchestrator, named Thrifty, which consists of two main components: 1) LIBRA, and 2) xCOSE.

- Motivated by our economic analysis and various features of FaaS, we designed LIBRA, a load-balancing approach. It utilizes IaaS and FaaS deployment of an application simultaneously. We evaluate LIBRA in simulations and on the AWS family of services. Our evaluation shows that LIBRA achieves more than 85% reduction in Service-Level-Objective (SLO) violations and up to 53% cost savings when compared to other resource provisioning policies. This work along with the economic model has been published at ***IEEE IC2E'21*** and received

the **Best Paper Award** [106].

- The xCOSE component of Thrifty focuses on further improving the cost and performance of FaaS deployment. xCOSE leverages Bayesian Optimisation for intelligent sampling and building the cost/performance model of a FaaS deployment on a given platform. Using this model, it finds the best configurations to optimize both cost and performance. Our work [48] was the first solution in this space. In this thesis, we extend our previous work to accommodate service graph applications and evaluate xCOSE on real multi-function applications on AWS. Our evaluation shows that xCOSE can successfully configure these serverless applications to optimize cost and meet the SLOs. This work has been published at ***IEEE TNSM'23*** [104].

1.5 Roadmap of thesis

The rest of the thesis is organized as follows. In Chapter 2, we give an overview of FaaS and discuss work related to addressing various challenges a developer faces while deploying and managing a cloud application using public clouds, particularly using FaaS. In Chapter 3, we provide a high-level view of Thrifty. In Chapter 4, we present an economic model comparing the cost of cloud usage when deployed using FaaS and IaaS with respect to the demand. In Chapters 5 and 6, we present the design and implementation of LIBRA and its evaluation in simulations and on the AWS family of services. In Chapters 7 and 8, we discuss xCOSE, a resource configuration module for FaaS deployment, and its evaluation on real applications. Chapter 9 concludes this thesis with future work.

Chapter 2

Background & Related Work

2.1 FaaS

Function-as-a-Service (FaaS) emerged as a new paradigm that makes the cloud-based application development model simple and hassle-free. In the FaaS model, an application developer focuses on writing code and producing new features without worrying about infrastructure management, which is left to the cloud provider. FaaS was first introduced by Amazon in 2014 as AWS Lambda [6], and since then, other commercial cloud providers have introduced their FaaS platforms, i.e. Google Cloud Function (GCF) [25] from Google, Azure Function [18] from Microsoft, and IBM Cloud Function [31] from IBM. There are also several open-source projects like Apache OpenWhisk, Knative, OpenLambda, Fission, and others. At the time of the inception of the Internet, applications were built and deployed using dedicated hardware acting as servers, which needed a high degree of maintenance and often lead to under-utilization of resources [63, 64]. Moreover, adding and removing physical resources to scale to varying demands, and debugging an application, was a cumbersome task. Under-utilization of resources and higher cost of maintenance led to the invention of new technologies like virtualization and container-based approaches. These approaches increased resource utilization and made it easy to develop, deploy, and manage applications. Many tools [63, 78, 118, 64] were built to help users orchestrate resources and manage the application. Although virtualization and container-based approaches lead to higher utilization of resources and ease of building applications, developers still

have to manage and scale the underlying infrastructure of an application, i.e. virtual machines (VMs) or containers, despite the availability of a number of approaches that would perform reactive or predictive scaling [70, 108, 53, 98, 91, 128]. To abstract away the complexities of infrastructure management and application scaling, FaaS computing emerged as a new paradigm to build, deploy, and manage cloud applications. The FaaS computing model allows a developer to focus on writing code in a high-level language (as shown in Table 2.1) and producing new features of the application while leaving various logistical aspects like the server configuration, management, and maintenance to the FaaS platform [120].

	AWS Lambda	**Google Cloud Function**	**IBM Cloud Function**	**Microsoft Azure Function**
Memory (MB)	{128 ... 10240}	$128 \times i$ $i \in \{1,2,4,8,16,32\}$	{128 ... 2048}	upto 1536
Runtimes Supported	Node.js 14/12/10, Go 1.x, Ruby 2.7/2.5, Python 3.8/3.7/3.6/2.7, Java 11/8, .NET Core 3.1/2.1, and Custom Runtimes	Go 1.13, Python 3.9/3.8/3.7, Ruby 2.7/2.6, Java 11, Node.js 14/12/10, .NET Core 3.1, and PHP 7.4	Node.js 12, Python 3.7/3.6, Java 8, Swift 4.2, PHP 3.7, Ruby 2.5, Go 1.15, .NET Core 2.2, and Docker	.NET Core 3.1/2.1, .NET Framework 4.8, Node.js 14/12/10/8/6, Java 11/8, PowerShell 7/6, and Python 3.9/3.8/3.7/3.6
Billing	Execution time based on memory	Execution time based on memory & CPU-power	Execution time based on memory	Execution time based on memory used
Billing Interval	1ms	100ms	100ms	1ms
Configurable Resource	memory	memory & CPU-power	memory	n/a
Free Tier	First 1M Executions	First 2M Execution	40,000GB-s	First 1M Executions

Table 2.1: Popular commercial FaaS platforms

FaaS was initially introduced to handle less frequent and background tasks, such as triggering an action when an infrequent update happens to a database. However, the ease of development, deployment, and management of an application and the evolution of commercial and open-source FaaS platforms have intrigued the research community to study the feasibility of the FaaS computing model for a variety of applications [128, 129, 93, 73]. Moreover, there are systems whose aim is to help developers port their applications to a FaaS programming model [65, 115].

In a FaaS computing model, a developer implements the application logic in the form of stateless functions (henceforth referred to as serverless functions) in the

higher-level language. We show various runtimes supported by popular FaaS platforms in Table 2.1. The code is then packaged with its dependencies and submitted to the FaaS platform. A developer can associate different triggers with each function, so that a trigger would cause the execution of the function in a sandbox environment (mostly containers) with specified resources, i.e. memory, CPU power, etc. The output of the serverless function is then returned as the response to the trigger. As serverless functions are stateless, a developer has to rely on external storage (like S3 from AWS), messages (HTTP requests), or platform API [35] to persist any data or share state across function instances. The FaaS computing model is different from traditional dedicated servers or VMs in a way that these functions are launched only when the trigger is activated, while in the traditional model, the application is always running (hence the term “serverless”).

FaaS abstracts away the complexities of server management in two ways. First, a developer, only writes the logic of an application in a high-level language, without worrying about the underlying resources or having to configure servers. Second, in case the demand for an application increases, a FaaS platform scales up the instances of the application without any additional configuration or cost and has the ability to scale back to zero. While FaaS platforms provide typical CPU and memory power to serverless applications, it is their ability to scale quickly (in orders of milliseconds) that gives them a performance advantage over other cloud services. On the contrary, in IaaS, an application developer not only has to specify the additional scaling policies but there can be an additional cost for deploying such autoscaling services and it can take up to minutes to scale up.

In Table 2.1, we show some of the key features provided by popular commercial FaaS platforms[1]. While providing similar services, specific features can vary signifi-

[1] **Features listed on official documentation as of 2/28/2023.**
AWS Lambda: `https://aws.amazon.com/lambda`

cantly from one platform to another. Generally, these platforms only allow memory as a configurable resource for the sandbox environment with the exception of GCF which also allows a developer to specify the CPU power. AWS Lambda allocates CPU in proportion to the memory allocated [9]. IBM Cloud Function seems to have a constant allocation of the CPU share regardless of the memory allocation as increasing memory does not improve runtime significantly [92]. Azure Function does not allow any configurable resource and charges the user based on the execution time and memory consumption [17]. While these platforms initially supported applications written in specific languages, they currently support more languages and custom runtimes, making it possible to run any application using FaaS.

An important feature of the serverless computing model is that serverless platforms follow the "pay as you go" pricing model. This means a user will only pay for the time a serverless function is running. This model charges a user for the execution time of the serverless function based on the resources configured for the function. A user will not be charged for deploying the function or for idle times. Even though all of the cloud providers follow a similar pricing model, the price for the unit time (*Billing Interval*) of execution can vary significantly from one cloud provider to another.

In the serverless computing model, the abstraction of infrastructure management comes at the cost of little to no control over the execution environment (and underlying infrastructure) of the serverless functions. Depending on the platform, a user can control limited configurable parameters, such as memory size, CPU power, and location to get the desired performance.

Azure Functions: `https://azure.microsoft.com/services/functions`
Google Cloud Functions: `https://cloud.google.com/functions`
IBM Cloud Functions: `https://www.ibm.com/cloud/functions`

2.1.1 Developer's View of FaaS

FaaS platforms are largely black boxes for application developers, who submit the code of their application (with a few configurations) and in turn, the code gets executed upon the specified triggers. A user has little to no control over the execution environment, underlying resource provisioning policies, hardware, and isolation. The user has control over limited configurations through which they can control the performance of their serverless application. In what follows we categorize the decisions a developer can make for their serverless applications to get the desired performance or optimize their cost.

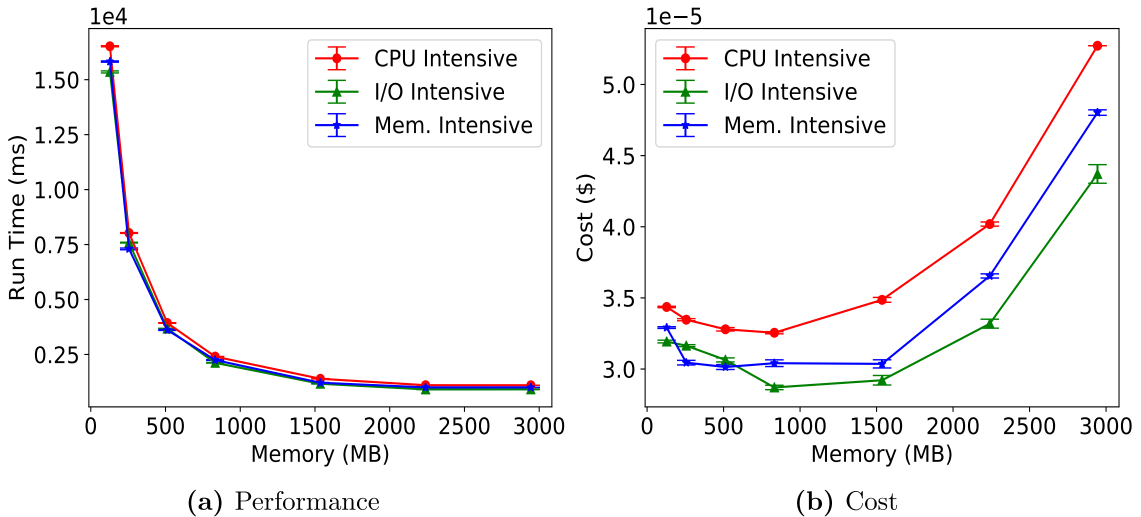
One-Time Decisions: These are the decisions that a developer can make before developing and deploying an application and include selecting the FaaS platform, programming language, and location of deployment. These decisions can be dictated by the features that a FaaS platform offers such as underlying infrastructure, pricing model, elasticity, or performance metrics – for example, certain languages may have lower cold-start latency or the location of deployment can affect the latency to access the application. We believe changing any of these aspects would incur significant development and deployment cost, hence a developer can make such a decision only once in the life cycle of the application.

Online Decisions: A developer has more freedom to change other parameters without a serious effort, including resources (memory, CPU), location, and concurrency limit. These parameters can affect the performance and cost of a serverless application. A developer can employ a more proactive technique to configure her serverless function based on the desired performance metric. Configuring these parameters is also important as serverless platforms provide no Service-Level Objective (SLO), i.e. guarantee on the performance of the serverless function, and a developer's only recourse to get the desired performance is through the careful configuration of these

parameters.

2.1.2 Performance & Cost

In FaaS, the abstraction of infrastructure management comes at the cost of little to no control over the underlying infrastructure and execution environment of serverless functions. A developer can control the performance through a few configurable parameters (*Online Decisions*) such as memory, CPU, and location of deployment. In this section, we review our previous work [49, 104] that studies the effect of these parameters on the performance of cloud applications and the cost of cloud usage.



(a) Performance **(b)** Cost

Figure 2·1: Performance and cost on AWS Lambda

To observe the effect of configurable resources on the performance of a serverless function, we deployed various (I/O-intensive, memory-intensive, and CPU-intensive) functions on AWS Lambda and invoked them with varying resource configurations, *e.g.* memory (the only configurable resource for AWS Lambda [48]). These functions represent the different types of computation (combination of *I/O*-, *CPU*-, *network*- and *memory*-intensive tasks) that a serverless application typically performs. Briefly, we describe these functions as follows: (i) *CPU-intensive:* This is a function that

calculates the trigonometric function *atan* of multiple numbers, hence making it a CPU-heavy function; (ii) *Memory-intensive:* This function applies a filter on a large image. This requires extensive use of memory; (iii) *I/O-intensive:* This function performs multiple I/O related operations on a file, *i.e.,* opening, reading and closing a file; Measurement for each function was repeated 50 times for every memory configuration. Our results showed that AWS Lambda's execution time follows exponential decay with respect to memory (*i.e.*, diminishing return). We show the observed trends in the performance and cost with respect to the resources in Figure 2·1, across all function types. It can be seen that more resources lead to faster execution of the serverless function but the performance gain is limited after a certain point. Note that the performance of I/O-intensive and CPU-intensive functions also improves with more memory allocation. It is because of the fact that AWS Lambda allocated CPU share in proportion to the memory allocated [8]. We observed similar trends for functions deployed over Google Cloud Function and IBM Cloud functions. Note that for IBM Cloud functions, the gain in performance with increasing resources was limited. No significant gain in performance with increasing memory (only configurable resource) suggests that IBM Cloud functions do not allocate CPU share proportional to memory [92]. This observation also confirms previous findings made in [48, 92, 66], which report a similar effect of resources on performance. In addition to resource configurations, AWS Lambda also allows a user to pick the location of deployment for serverless functions, i.e. regions and edge locations [7]. The choice of placement can affect the user-perceived latencies as well as cost (Lambda@Egde is 3x more expensive than the core regions [7]).

In addition to resource and placement configurations, the performance of FaaS deployments can be impacted by various temporal conditions such as cold starts, resource contention, co-location, and arbitrary placement on variable infrastructures [120,

48]. But these factors are beyond a tenant's control and a developer's only option to control performance is through various one-time and online configurations.

Given the effect of resource configurations and placement on the performance and cost of a serverless application, it is critical to configure these parameters properly to meet the SLOs cost-effectively. In this thesis, we address the challenge of configuring serverless applications through our framework xCOSE, which is a statistical learning-based approach and configure both single and multi-function serverless applications at minimal sampling cost and can adapt to any changes in underlying execution model due to resource provisioning policies of the cloud provider, co-location and *etc.*

2.1.3 FaaS Advantages

FaaS in addition to a simple development model and ease of managing an application provides two main performance and cost advantages:

Performance – FaaS platforms execute application code in lightweight containerized environments which can be provisioned in the order of milliseconds [50, 57, 120]. Moreover, FaaS platforms can scale the application instances seamlessly without any additional configurations, unlike IaaS where a developer has to employ scaling services. These performance features have been leveraged by previous works [128, 77, 81, 99] to avoid the SLO violation during the scale-out events of applications deployed over IaaS (as VMs can take up to minutes to start [81, 106]).

Cost – FaaS platforms charge their tenant precisely for the amount of time an application code is executing with a time granularity as low as 1 ms. Also, unlike IaaS where a user is charged for the allocation time of VM resources regardless of the usage, FaaS has no idle cost. This particular pricing model makes FaaS a cost-effective option to run computations with low demand.

The cost-effectiveness of FaaS for smaller computations comes from the fact that FaaS platforms charge users on a much smaller granularity of time (often 1ms or

100ms) while IaaS resources are rented over multiple of larger time units (1 second or hour). Consider FaaS charging for t time units and IaaS for T time units such that $T = kt$, where $k > 1$. Also, consider the cost of FaaS is C_{FaaS} for one unit time and the cost of IaaS per unit time is C_{IaaS}, such that $C_{IaaS} < C_{FaaS}$. If an application runs for ct time units where $1 \leq c < k$, FaaS is more cost-effective than IaaS under the following condition:

$$C_{IaaS} \times T > C_{FaaS} \times ct$$

$$\frac{ct}{T} < \frac{C_{IaaS}}{C_{FaaS}} < 1$$

The above relation essentially provides the viability condition of our approach. It shows that FaaS would be cost-effective if the utilization of the computation ($\frac{ct}{T}$) is less than the ratio of the costs of IaaS and FaaS per unit time ($\frac{C_{IaaS}}{C_{FaaS}}$).

In this thesis, we combine the cost-effectiveness of FaaS with the above-mentioned performance features to not only avoid SLO violations but also reduce the cost in the longer run.

2.2 Related Work

Even though FaaS has been around for only a few years, this field has produced a significant volume of research. This research addresses various aspects of FaaS from benchmarking and improving the performance of various FaaS platforms and applications, porting new applications into a serverless model, to suggesting altogether new serverless platforms. First in Sections 2.2.1, 2.2.2, and 2.2.3, we look at various complementary works to our approach in this thesis. These works can help developers to port legacy applications to the FaaS programming model and find suitable FaaS platforms through benchmarking studies, making usage of Thrifty more viable. Second, in Sections 2.2.4 and 2.2.5 we discuss previous works, which are directly related

to our different components of Thrifty.

2.2.1 FaaSification of Application

FaaS development and computing models significantly differ from traditional IaaS models. Hence, to deploy a legacy application using FaaS, a developer has to translate the application into this unique model. Recently, there have been approaches such as [115, 65, 107] that aim to automate this process for applications written in various languages. As pointed out by Yao et al.[126], these approaches either work for selected parts of the application or fail to leverage some of the key performance benefits offered by FaaS. In particular, these approaches replace a selected part of an application with a Remote Procedure Call (RPC) and deploy the selected part as a serverless function. While helpful to quickly deploy legacy applications using FaaS, these approaches miss taking advantage of the elasticity feature offered by FaaS. We believe that an ideal FaaSification tool should not only consider producing the FaaS counterpart of the application but also leverage the elasticity offered by FaaS platforms. For example, through static and dynamic code analysis, the tool should identify the parts of an application that can be parallelized and generate corresponding serverless functions. An ideal tool for the FaaSification of legacy applications will make the usage of Thrifty easier by significantly reducing the development cost.

2.2.2 FaaS Usage

FaaS is becoming a popular service to deploy cloud applications. Developers have employed FaaS to deploy rather simple DevOps to full production scale applications [83, 67, 111, 23]. We only present here some of the interesting use cases of serverless computing and FaaS.

Malawski et al. [93] show that AWS Lambda and GCF can be used to run scientific workflows. Serverless computing can also be employed to solve various mathematical

and optimization problems [56, 113, 123]. Moreover, on-demand computation and scalability provided by serverless computing can be leveraged by biomedical applications [85, 84, 79]. MArk [128], Spock [77], Cirrus [58] and others [80, 117] explore deploying various machine learning applications using FaaS platforms. The authors in [119, 71] leverage the higher level of parallelism offered by serverless platforms to train machine learning models. FaaS for its on-demand, cost-effective computation power and elasticity has also been explored to deploy stream processing applications [41, 90]. Video processing is one such example, where a user may want to extract useful information from an incoming video stream (video frames), where for each new incoming frame a serverless function can be spawned. Sprocket, ExCamera and others [54, 72, 129] describe the implementation of video processing frameworks using serverless functions. Authors in [101, 60, 100] explore the possibility of using serverless computing for IoT applications and services. Yan et al. [125] use serverless computing to build chatbots. Aditya et al. [47] present a set of general requirements that a cloud computing service must satisfy to effectively host SDN- and NFV-based services. Chaudhry et al. [59] present an approach to improve the Quality of Service (QoS) on the edge by employing virtual network functions using serverless computing.

2.2.3 Measurement & Benchmarking

A significant amount of research work addresses benchmarking and demystifying the various aspect of FaaS platforms such as performance, resource provisioning policies of cloud providers, and the effect of various configurable parameters. These studies and benchmarking tools can help developers to find suitable FaaS offerings for their applications. A detailed survey of these studies and findings is presented in our previous work [105]. In what follows, we briefly discuss some of the interesting findings of these studies.

Cold Start: These studies show that the cold start of serverless applications can be impacted by various one-time and online decisions such as choice of language (implementing the application) [120, 94, 22], cloud provider [120, 88, 97, 94] and code package size [22, 121].

Performance & Cost: As shown by our experimental study (Section 2.1.2), the performance and cost of serverless applications are impacted by the resource configurations of an application. Moreover, studies have also shown that the performance can also be impacted by concurrency [86, 89, 88, 120], co-location [120, 48] and the choice of cloud provider [120, 121].

Elasticity: Elasticity or scalability of an application deployed over a FaaS platform can be impacted by the choice of cloud provider [120, 92] as these providers can be following different resource provisioning policies. These studies [120, 92] show that AWS lambda is best at scaling applications in the face of increased demand. Moreover, elasticity can also be impacted by the configurations [96, 33, 40] and the choice of language for an application [92].

These studies also look at other aspects of FaaS platforms such as the diversity of underlying infrastructure [120, 88], network and I/O throughput [120, 86, 121].

Benchmarking Tools: Many tools have been suggested for benchmarking different aspects of FaaS platforms [75, 114]. FaaSdom [92] is a benchmark suite for FaaS platforms. It supports the current mainstream serverless cloud providers (i.e., AWS, Azure, Google, IBM). Serverless Benchmark Suite (SeBS) [61] is another benchmarking tool that facilitates a developer to benchmark various FaaS platforms with diverse workloads and derive meaningful insights regarding cost and performance. ServerlessBench [127] is an open-source benchmark suite that can help understand characteristic metrics of serverless computing, e.g., communication efficiency, startup latency, stateless overhead, and performance isolation

2.2.4 FaaS Application Management

FaaS platforms largely work as black-box and the performance of serverless applications can be controlled through configurable parameters such as resources (memory and CPU), location, and concurrency limits.

These configurable parameters not only control the performance but can also impact the cost of cloud usage. Much work has been done to address the challenge of configuring serverless applications. Schuler et al. [96] show that the container-level concurrency limit can affect the application's performance. They also suggest an AI-based (reinforcement learning) technique to configure the concurrency limit for Knative. Sizeless [66] is an ML-based approach to configure a serverless application. It trains a multi-target regression model with the profiling data of thousands of synthetic functions. Using this ML model Sizless can predict the resource configurations for serverless applications that would meet the SLO. Similarly, FnCapacitor [82] and others [109, 122] allow users to configure resources for serverless applications to meet the SLO while minimizing the cost. StepConf [122] is a more dynamic approach to configure serverless applications but does not perform function placement. Other approaches to configure resources for serverless functions include local simulations [95] or tracing the underlying infrastructure through logging [62]. Our previous work COSE [48], and AQUATOPE [130], leverage Bayesian Optimization, a statistical learning-based approach to find the best configuration for complex serverless applications consisting of multiple functions. COSE [48] and Costless [68] are two approaches, which in addition to finding the best resource configuration can also perform placement of serverless functions on the edge or core cloud based on the SLO. These previous approaches, either fail to capture the dynamicity of the FaaS execution model, have high costs, or do not work for complex applications.

In this thesis, we extend our previous work COSE [48], which was shown to have

minimal overhead and capture the dynamic nature of the FaaS execution model. COSE [48] can perform resource configuration and placement for simple applications consisting of linear chains of functions. We present xCOSE, which can also perform configuration and placement for complex applications consisting of service graphs. We also evaluate xCOSE using real applications on AWS Lambda.

2.2.5 FaaS Usage in Hybrid Cloud

FaaS for its unique cost and performance features has been extensively studied for hybrid clouds.

Previous works have explored offloading high-scalable parts of an application to FaaS while running the rest of the application on other services. ExCamera [72] and Sprocket [54] present video processing frameworks using multiple service types where part of the application runs in IaaS and utilizes the scalability of FaaS by launching multiple serverless functions to process data streams. To leverage the best-in-class services from different providers, Shamrock [23] presents a use case where AWS and Google services are combined to build an invoicing system. Malawski et al. [93] present a scientific workflow management system that leverages both IaaS and FaaS resources to perform various computations.

Similar to our work in this thesis, FaaS has also been studied for cloud bursting for its performance features. In this case, an application is deployed on both IaaS and FaaS and based on demand it is decided which deployment to use. Spock [77] and MArk [128] suggest the usage of FaaS in conjunction with IaaS resources to cater to the bursty demand and lower SLO violations for Machine Learning applications. Similarly, FEAT [99] and SplitServ [81] also leverage the quick provisioning of FaaS for temporary usage in conjunction with other resources to improve performance. SIRM [102] is a recent work that utilizes MLaaS, fog resources, and serverless functions to serve ML models in order to reduce SLO violations.

We believe that the previous usage of FaaS for cloud bursting fails to take full advantage of FaaS and do not address the following:

- In addition to performance features such as lower cold start and high scalability, FaaS offers cost advantages for low-rate demand. We believe that the cost feature can be combined with the performance features not only to improve performance but also cost.

- Previous approaches assume static configurations for FaaS deployment. FaaS platforms offer no guarantee on SLO, and performance can be impacted by various temporal conditions such as cold-starts, co-location, and placement on variable hardware. Moreover, FaaS deployment can consist of complex workflows such as service graphs. Hence, static configurations can lead to sub-optimal cost or performance, hence either violating SLO or incurring higher cost.

In this thesis, we present our framework Thrifty, which not only utilizes both IaaS and FaaS to optimize the cost and performance of cloud applications but also can configure the FaaS deployment.

Chapter 3

Thrifty

As mentioned in the previous chapters, today's cloud providers offer a plethora of services, and oftentimes an application can be deployed using more than one type of service. To leverage the unique cost and performance features of each service, a developer may choose to replicate the application in multiple services and then based on temporal conditions, dynamically decide which deployment to use (called *cloud bursting*).

In this context of load balancing among multiple deployments, each with unique cost and performance features, this problem can be formally described using queuing theory as distributing demand among different servers. The goal is to distribute the demand among available deployments (servers) while minimizing the cost and meeting the performance requirements of the application. Assume each server is modeled as an M/M/1 queue. A load balancer has to split the application load, with rate λ, among N available deployments (servers) with each receiving $\alpha_i\lambda$ share of the total load (where $0 \leq \alpha_i \leq 1$ and $\sum_{i=1}^{N} \alpha_i = 1$). Each server i has a service rate μ_i and associated cost c_i that represent deployment-specific parameters.

In this thesis, we focus on utilizing two ($N = 2$) popular cloud services, namely IaaS and FaaS, to deploy an application. Consider the model shown in Figure 3·1, where the application has a demand λ (requests per second). Assume the FaaS service is slower than the dedicated VM-based service (IaaS) but is less costly. The FaaS deployment (server) has a service rate of μ (requests per second) and the cost for

serving each request is γc (\$), where $\gamma < 1$. The IaaS deployment (server) has $\beta\mu$ service rate ($\beta > 1$) and associated cost of c (\$). The load balancer distributes requests with FaaS receiving $\alpha\lambda$ requests per second, and IaaS receiving $(1-\alpha)\lambda$ requests per second (where $0 \leq \alpha \leq 1$). To obtain the optimal value for α that minimizes cost, we solve the following optimization problem.

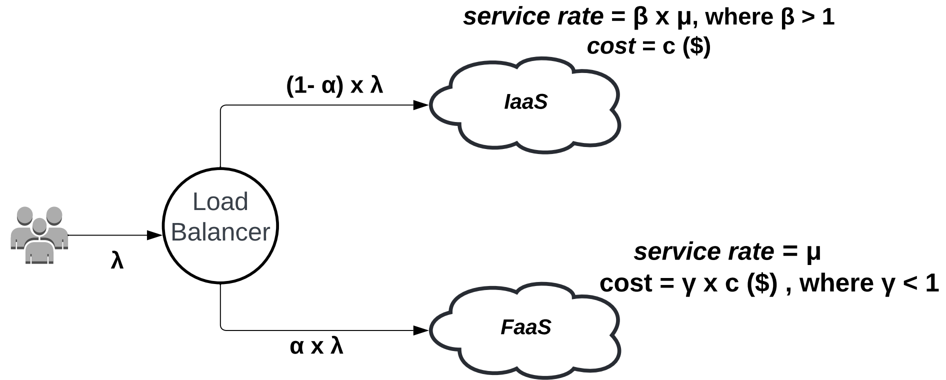
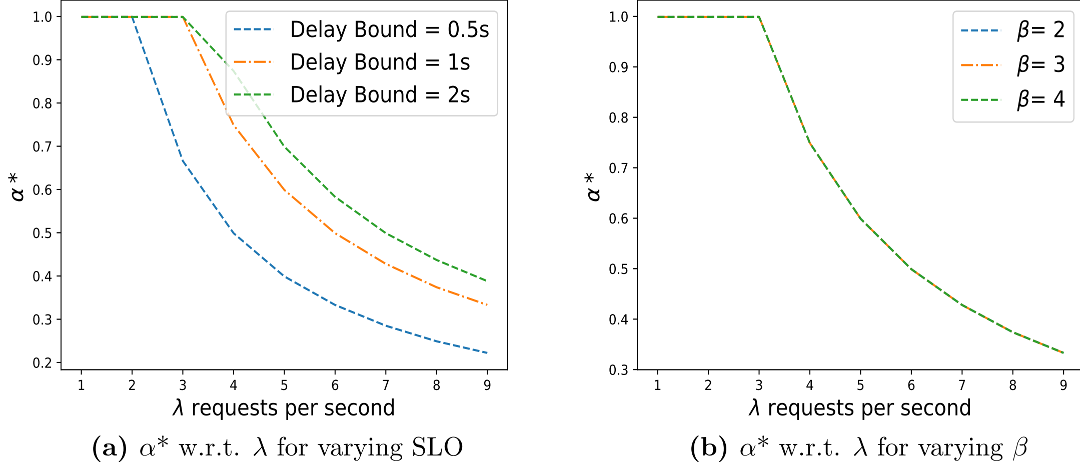


Figure 3·1: Queuing Model

$$
\begin{aligned}
\text{minimize} \quad & \alpha\lambda\gamma c + (1-\alpha)\lambda c \\
\text{subject to} \quad & \frac{1}{\mu - \alpha\lambda} \leq T, \\
& \frac{1}{\beta\mu - (1-\alpha)\lambda} \leq T, \\
& \alpha\lambda < \mu, \\
& (1-\alpha)\lambda < \beta\mu
\end{aligned}
$$

The objective function is the cost to serve demand λ and the constraints ensure that the average delay for each request is below the expected SLO, i.e. delay bound T, and that the system is stable.

Using the above constraints, we can derive the feasible range for the value of α as:



(a) α^* w.r.t. λ for varying SLO **(b)** α^* w.r.t. λ for varying β

Figure 3·2: Effect SLO and β on optimal α^*

$$\max(0, \frac{1 - T\beta\mu + T\lambda}{T\lambda}, \frac{\lambda - \beta\mu}{\lambda}) \leq \alpha \leq \min(1, \frac{T\mu - 1}{T\lambda}, \frac{\lambda}{\mu}) \tag{3.1}$$

Consider the following parameter values: $\mu = 4$, $\gamma = 0.3, \beta = 2, c = 1$, and $T = 1$. We observe the effect of different delay bound (T) and relative speed of the IaaS service (β) on the optimal value of α^* for varying demand λ in Figure 3·2. As it can be seen in Figure 3·2a, the delay bound affects the optimal share of FaaS. As the demand increases, the system would first employ FaaS as its cheaper until it no longer can ensure the SLO and then start using the IaaS deployment by decreasing the value of α^*. As the delay bound becomes more stringent, the system starts using IaaS sooner to meet the SLO. Figure 3·2b shows the effect of β on α^*. While the value of β does not have an apparent effect in this case, it can affect the overall system stability as the system can only serve demand λ such that $\lambda < \mu + \beta\mu$. In the case when λ increases beyond $\mu + \beta\mu$, the system will need to increase its service capacity by adding more resources (called autoscaling).

Despite its apparent simplicity, in real systems, load balancing can be a challenging task, particularly for the following reasons:

- Varying Demand (λ): The demand for an application can vary significantly over time [128, 77, 112]. An optimal resource provisioning and load balancing approach should know patterns of demand for the applications in order to provision the appropriate amount of computing resources to handle it. This is particularly important for IaaS deployments as provisioning VM resources can take up to minutes to start.

- Service Rate ($\mu + \beta\mu$): Ensuring a constant service rate can be a challenging task for reasons including but not limited to co-location, cold-starts, network latency, and resource provisioning policies of the cloud provider [51, 48]. Moreover, the service rate depends on the amount of resources provisioned, i.e. VMs for IaaS deployment and memory/CPU for FaaS deployment.

- The Cost Factor (γ): Different cloud providers and services follow varying pricing models and to estimate the cost correctly, one may have to rely on various analytical [106] and experimental [128] methods, given the application's performance model to accurately estimate the cost of cloud usage. Moreover, as we show in Chapter 4, the resources allocated for each request (both in IaaS and FaaS) can also affect the cost factor.

To address the above challenges, we present Thrifty, a resource provisioning and load-balancing approach for cloud applications. Based on the demand, it uses IaaS, FaaS, or both deployments of an application to improve overall performance and minimize the cost of cloud usage. As shown in Figure 3·3, Thrifty consists of two main components, a load balancing framework called LIBRA and a configuration module for FaaS deployment called xCOSE. It not only efficiently utilizes both IaaS and FaaS but also performs configuration (resources and placement) for the FaaS deployment of the application to further augment the cost savings. In the next chapters, we will discuss these components of Thrifty in detail.

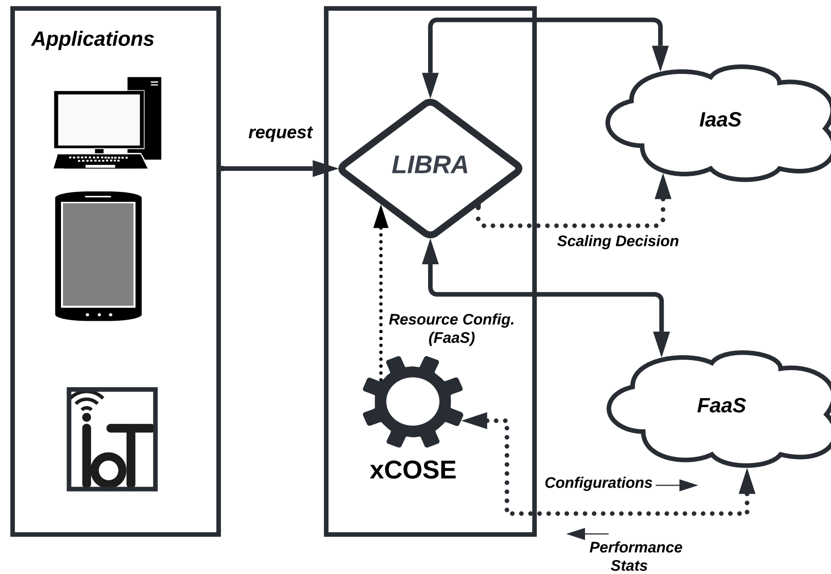


Figure 3·3: Thrifty Architecture

In the next chapter, we present the economic model of applications when deployed using FaaS or IaaS and discuss cost implications with respect to demand.

Chapter 4

LIBRA Motivation

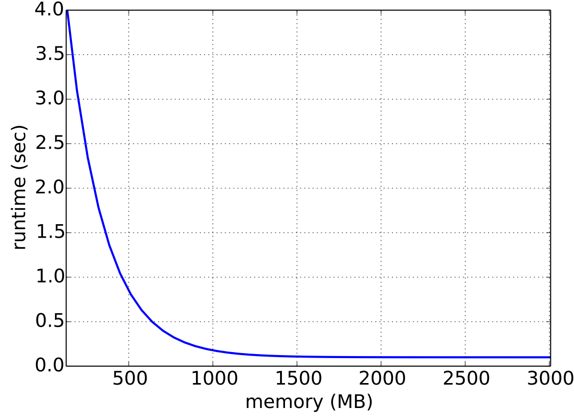
In this chapter, we present an economic (cost) model of an application deployed using either FaaS or IaaS cloud service. We use this model to derive the "Cost Indifference Point" (CIP) as a function of the request arrival rate, where the costs of using IaaS or FaaS for an application are equal. If the application load is below this CIP value, it is more economical to use FaaS. For higher loads, IaaS is more economical.

This analysis helps decide the *Cost Factor* (γ) (mentioned in Chapter 3) and motivates the design of the load-balancing component (LIBRA) of Thrifty, which based on demand, decides whether to use FaaS, IaaS, or both.

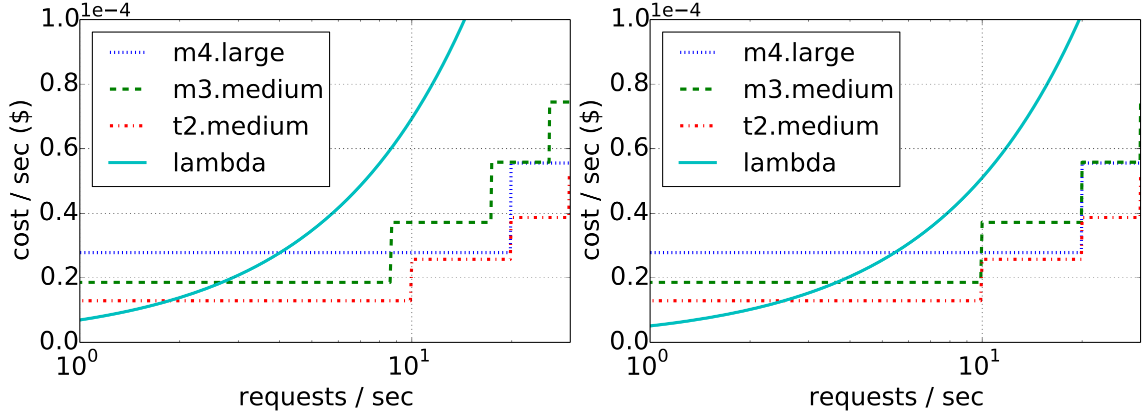
4.0.1 FaaS Pricing Model

FaaS platforms follow a "pay as you go" pricing model where the user is only charged for the execution time of the serverless function based on a particular configuration (*e.g.*, memory in the case of AWS Lambda) [10]. Our previous work [48] studies the effect of configurable resources on the performance of serverless functions deployed over AWS Lambda and shows AWS Lambda's resource and performance relation can be expressed as follows:

$$\begin{aligned} t_f^{FaaS}(m) \approx\ & t_f^{FaaS}(m_{max}) + \\ & \left(t_f^{FaaS}(m_{min}) - t_f^{FaaS}(m_{max})\right)\, e^{-\lambda(m-m_{min})} \end{aligned} \tag{4.1}$$



(a) AWS Lambda Execution Model



(b) Cost at Request Memory=512MB **(c)** Cost at Request Memory=3008MB

Figure 4·1: Cost comparison of Amazon Lambda and EC2 instances for varying average request arrival rate

where $t_f^{FaaS}(m)$ is the execution time of a function f when allocated memory m MB, $t_f^{FaaS}(m_{min})$ is the running time of f at the smallest possible memory configuration ($m_{min} = 128$ MB for AWS Lambda), $t_f^{FaaS}(m_{max})$ is the running time at the largest possible memory configuration ($m_{max} = 10$ GB for AWS Lambda), and λ is a decay constant.

Consider an application, deployed using FaaS, that receives N requests per second, where each request causes the execution of a serverless function f. The usage cost per second can be calculated as follows:

$$cost_{FaaS} = \sum_{i=1}^{N}(t_f^{FaaS}(m) \times C_{FaaS}(p, m) + G_{FaaS}(p)) \tag{4.2}$$

where $C_{FaaS}(p, m)$ is the cost per unit time[1] of executing a serverless function as specified by the serverless platform p for a given configuration m, and $G_{FaaS}(p)$ is the total fixed cost charged by the cloud provider (such as API-gateway cost for AWS Lambda [10]). This cost model also holds for other cloud providers which follow similar pricing models for FaaS, such as IBM Functions, Google Cloud Functions, *etc.*

4.0.2 IaaS: VM Pricing Model

In the IaaS model, a tenant leases a VM with a particular configuration, such as memory, CPU, and storage, to deploy an application. A tenant is charged the cost for the allocated time of the VM (regardless of its utilization). A VM with a particular configuration can only serve a certain number of requests in a given time period while meeting SLO requirements.

If a VM can host at most r_{max} requests per second without violating the SLO, and the IaaS based deployment receives N requests per second, the cost per second can be calculated as follows:

$$cost_{IaaS} = \lceil \frac{N}{r_{max}} \rceil \times C_{vm}(p) \tag{4.3}$$

where $C_{vm}(p)$ is the cost per second[2] of renting a particular VM (vm) from a certain cloud provider p.

4.0.3 Cost Analysis

Using Equations (4.2) and (4.3), we compare the cost of deploying an application using FaaS or IaaS, respectively. We evaluate the cost for varying demand given by

[1] FaaS platforms currently charge for every 1ms or 100ms of execution time, depending on the cloud provider[10, 32, 27, 17].

[2] IaaS resources can be rented on an hourly basis, while a user can also be charged for partial usage (per second)[37, 21, 30].

N, the rate of requests for the application, where each request causes the invocation of application code deployed using FaaS or IaaS.

The execution model of the application/function used is shown in Figure 4·1a and follows an exponential decay in the running time of the function with respect to the amount of resources (memory) allocated [48, 80]. It gives the execution time $t_f^{FaaS}(m)$ of the application for different memory m settings when deployed using FaaS. IaaS based deployment would follow a slightly different execution model as underlying resources can differ from FaaS.

Thus, the execution model of the IaaS based deployment with respect to FaaS can be described as:

$$t_f^{vm}(m) = \tau \times t_f^{FaaS}(m) \tag{4.4}$$

where $t_f^{vm}(m)$ is the execution time of the IaaS based deployment when allocated memory m to each request, and τ is a constant whose value is a real positive number and can vary based on the application and underlying resources. Without loss of generality, we show results where under both IaaS and FaaS, the application follows the same execution model (*i.e.*, $\tau = 1$), and memory is the bottleneck resource in the execution of the function as most FaaS platforms allow only memory as a configurable resource. Other resources, such as CPU, I/O, Network, etc., can also be bottlenecks in the execution of a function. These resources can be substituted here to get a similar analysis. Note that setting τ to values different than 1 does not qualitatively affect the results of our analysis.

Using $t_f^{FaaS}(m)$ and m, we calculate $cost_{FaaS}$ using Equation (4.2), where the costs $C_{FaaS}(p, m)$ and $G_{FaaS}(p)$ are taken from AWS Lambda pricing [10].

For IaaS, the $cost_{IaaS}$ is calculated using Equation (4.3), where r_{max}, the maximum number of requests that a VM with memory M can handle in one second, can

be derived using Little's Law [52]:

$$\frac{M}{m} = r_{max} \times t_f^{vm}(m)$$

$\frac{M}{m}$, the long-term average number of concurrent requests in the system, equals the arrival rate of these requests (r_{max}) times the (average) time that a request spends in the system ($t_f^{vm}(m)$). We thus have:

$$r_{max} = \frac{M}{m} \times \frac{1}{t_f^{vm}(m)} \tag{4.5}$$

We use the AWS Elastic Compute Cloud (EC2) pricing model for different types of EC2 instances (m4.large, m3.medium, and t2.medium). The cost $C_{vm}(p)$ and memory resources M of these instances are specified in EC2 pricing [21]. Figure 4·1b compares the cost of cloud usage when an application is deployed in AWS Lambda or in various instances of EC2 for varying request rate and memory m of 512MB. In real life, these resource configurations are picked keeping in view the SLO. The x-axis is drawn on a logarithmic scale for better readability. We observe that the FaaS model is cost-effective when the request rate is below 4 requests/second for the m4.large EC2 instance (the point where the m4.large and lambda cost curves intersect). This represents the cost-indifference point (CIP) beyond which the IaaS model is cheaper to be used. The CIP is obtained by equating Equations (4.2) and (4.3). Figure 4·1c shows a similar behavior when each request is using memory m of 3008MB.

Though the results shown here are obtained using AWS pricing, the cost model is applicable to other cloud services (*e.g.*, from IBM Cloud Functions and Google Cloud Functions) that follow a similar pricing model. To summarize the key takeaways from our analysis:

- The FaaS model is cheaper to use for low duty-cycle application, *i.e.* when the average request rate N is below the CIP. For higher values of N, IaaS is cheaper.

- The value of CIP depends on the amount of resources used by each request and the type of VM instance. A tenant can find the appropriate resource configuration by profiling the application or using inference approaches, as proposed in [51, 48].

4.0.4 FaaS + IaaS

Demand for an application can significantly vary across certain hours of the day and certain days of the week. Based on our analysis in previous sections, an ideal hybrid load balancing approach will have two main characteristics:

(a) It would continually monitor the demand for an application and when the demand is below CIP, it will only provision FaaS resources to cater to the demand as they are more cost-effective in such a scenario. This is a feature that previous hybrid approaches [77, 128] lack, as they only use FaaS either for transient demand or during scaling out VM resources to avoid SLO violations.

(b) It should employ FaaS consistently for a low-rate and bursty portion of the demand that the system can not serve using IaaS resources within the SLO. This would be beneficial in two ways: first, it will reduce the SLO violations, as sudden spikes in demand would be handled by FaaS, which has negligible cold-start delays and can natively scale out. Second, consistently employing FaaS for a certain portion of the demand can lead to significant cost savings.

To demonstrate the cost saving of such an approach, we leverage the cost analysis in Section 4.0.3. Consider the scenario shown in Figure 4·1b, where an application runs on an EC2 instance of the type *m3.medium* and has a steady demand of 10 requests per second where each request requires 512MB of memory. In Figure 4·2, we compare the cost of serving all the demand or a certain portion of it using FaaS while serving the rest through IaaS. We observe that a hybrid approach, where around 20%

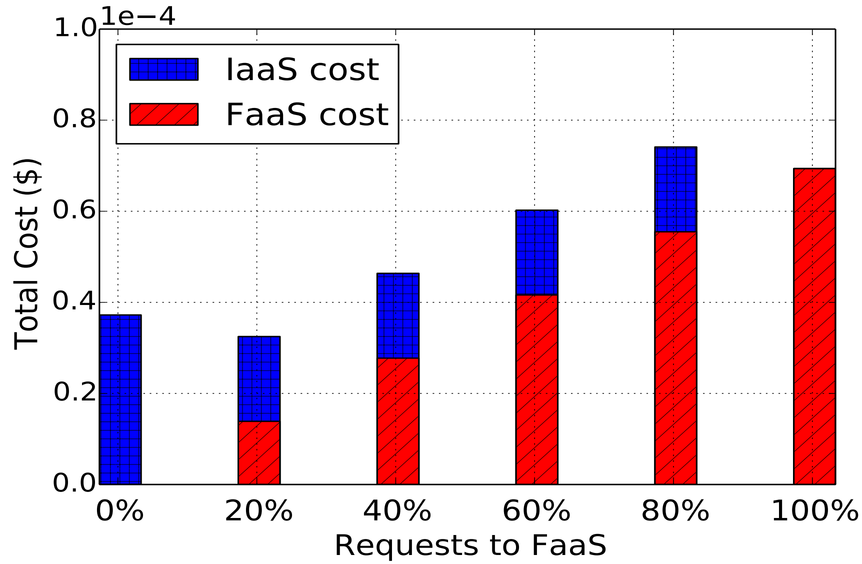


Figure 4·2: Hybrid Case

of requests are served by FaaS and the remaining by IaaS is the most cost-effective as compared to IaaS or FaaS-only scenarios. This is because 20% of the demand is below the CIP for this particular case and is cheaper to be served through FaaS than spinning up a new VM which would be underutilized.

Motivated by the above analysis, we designed our load balancing component LIBRA of Thrifty. In the next chapter, we explain the LIBRA architecture, its implementation, and evaluation.

Chapter 5

LIBRA Architecture

As shown in the previous chapter, the most cost-effective and efficient resource provisioning policy, depending on the demand, should employ FaaS, IaaS, or both to run an application. In this chapter, we present LIBRA, a balanced approach that leverages both IaaS and FaaS. It closely monitors the demand from an application and provisions appropriate VM capacity for the IaaS deployment to handle a portion of the requests while directing the rest to be handled by the FaaS-based deployment of the application. Motivated by our analysis in Chapter 4, the design of LIBRA derives from the following goals:

- Utilize FaaS deployment only if the demand falls below CIP.
- Utilize FaaS for bursty demand to avoid SLO violations, leveraging FaaS's quick provisioning time.
- As explained in Section 4.0.3, a steady rate of traffic below a specific limit (CIP) can be cheaper to serve through serverless functions (FaaS). An efficient load-balancing approach would leverage FaaS for a steady (low) traffic rate *consistently* to reduce the overall cost of cloud usage.

The above goals present load balancing between FaaS and IaaS as a marginal analysis problem [76], where a user can direct a small/bursty portion of the demand to FaaS (additional activity) to be served cost-effectively instead of provisioning new

VMs. This would reduce cost and lead to lower SLO violations as VMs take significantly longer to start (cold starts).

5.1 Architecture

Figure 5·1 gives an overview of our proposed approach. The load balancing across the IaaS and FaaS-based deployment of the application is performed through a *Load Balancer*, which also collects the traffic statistics and shares them with the *Traffic Monitor* using the control plane. Based on the traffic demand, the *Scaling Manager* provisions VM resources for IaaS deployment (*if needed*) and updates the Load Balancer to enforce appropriate forwarding rules. Henceforth, we refer to all three components of LIBRA as LIBRA Gateway (LG). In what follows, we explain each component of LIBRA in detail.

5.1.1 Traffic Monitor

Traffic Monitor's job is to estimate/predict future demand so LIBRA can provision resources if needed and avoid over-provisioning.

Traffic Prediction: It can be viewed as a time series analysis, where given the past history, one estimates the future workload. Most of these workloads exhibit recurring patterns over certain periods of time with anomalous events such as sudden enormous spikes in demand. This area has been extensively studied and there is a number of reactive and predictive mechanisms to predict the future demand for an application such as (rolling-window) linear-regression [128], neural network (LSTMs) [128, 98], autoregressive models [70, 108], and *, etc.*. The choice of the demand prediction mechanism can greatly depend on traffic patterns.

In LIBRA, the Traffic Monitor (TM) continually receives traffic updates from the *Load Balancer* i.e. number of requests received per unit time (*usually second*). Using the historical data of these updates, the TM estimates the future load for the

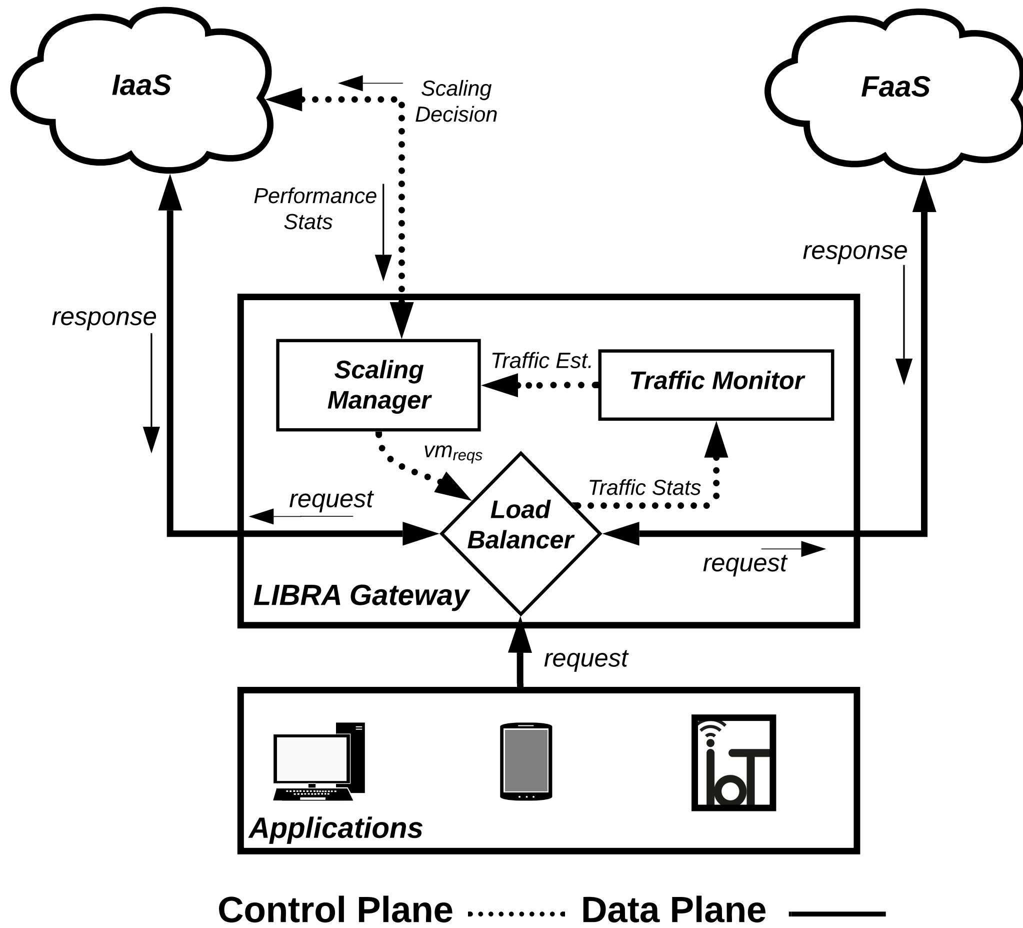


Figure 5·1: LIBRA architecture

application. The forehand knowledge of traffic is critical particularly for the VM-based resources because they can take up to several minutes to start and be ready to serve application traffic. In the current implementation, we introduce the notion of an *epoch* that represents a configurable unit-time (*e.g.*, 1 or 10 seconds or 1 minute). The LB continually reports the number of requests received in an *epoch* to the TM, which uses this information to estimate the future load. Currently, the TM keeps track of the Exponentially Weighted Moving Average (EWMA) and sample deviation of requests received in previous epochs as given in Equations (5.1) and (5.2), where $reqs_{curr}$ is the number of requests received in the current epoch, α and $\beta \in [0, 1]$ are

configurable based on how quickly a user wants the system to react in the face of traffic variations. Our experiments have shown that EWMA and sample deviation of the number of requests track well the traffic variation as shown in Figure 6·1.

$$avg = (1 - \alpha) \times avg + \alpha \times reqs_{curr} \tag{5.1}$$

$$std = (1 - \beta) \times std + \beta \times |reqs_{curr} - avg| \tag{5.2}$$

The TM reports the avg and std values to the *Scaling Manager* every K epochs. The *Scaling Manager* then makes scaling decisions as explained next.

5.1.2 Scaling Manager

The Scaling Manager (SM) is a crucial component of our LIBRA architecture. It periodically receives the traffic statistics from the *Traffic Monitor* and orchestrates resources in the form of VMs for IaaS deployment of the application.

As one of the design goals of LIBRA is to keep the serverless load below a certain threshold (CIP) to avoid overpaying, and send the maximum stable load to provisioned VMs for their cost effectiveness, our SM provisions VM (IaaS) resources that can handle a request rate equal to $(avg + \phi \cdot std)$, with remaining requests directed to run as serverless functions (FaaS). $\phi \in \mathbb{R}$ is a configurable parameter of the LIBRA system *and is discussed later in Section 5.1.4.* Our experiments (cf. Sections 6.1 and 6.3) have shown that any traffic above $(avg + \phi \cdot std)$ is either transient or cheaper to be served by serverless functions. LIBRA provisions VMs cautiously based on the estimated demand given by $(avg + \phi \cdot std)$ so as to avoid either under-provisioning VMs and then suffering from higher startup delays while spinning up additional VMs, or over-provisioning VMs and paying unnecessary cost due to VM under-utilization.

Algorithm 1 describes how the SM procures VM resources for IaaS. In line 1, the function *get_active_vms*() returns the current number of active VMs. Line 2 checks if

Algorithm 1 LIBRA's Scaling Algorithm

Input:
avg, std: EWMA and sample deviation reported by the Traffic Monitor (TM)
VM_{res}: resources available in each VM instance
req_{res}: resources required to serve one application request
req_{time}: average request service time
cip: Cost Indifference Point
ϕ: number of sample deviations beyond average demand
Output:
vm_{reqs} // request rate that provisioned VMs can handle

```
 1: active_vms = get_active_vms()
                  // returns the number of active VMs
 2: if avg < cip then
 3:   remove_vms(active_vms)
                 // removes all VM instances
      vm_reqs = 0
 4:   return
 5: end if
 6: r_max = vm_capacity(VM_res, req_res, req_time)
        // get the maximum number of requests a VM can serve
 7: vm_reqs = avg + φ · std
 8: num_instances = ⌈(vm_reqs/r_max)⌉
 9: vm_diff = num_instances - active_vms
10: if vm_diff > 0 then
11:   add_vms(vm_diff) // adds VM instances
12: else
13:   remove_vms(vm_diff) // removes VM instances
14: end if
```

the average request rate is below the CIP threshold (obtained through cost analysis). If it is, LIBRA shuts down and deallocates all the currently provisioned VMs (line 3), as the current demand can be met cost-effectively using only serverless functions. In line 6, the algorithm calculates the number of requests (r_{max}) that a VM with given resources can serve while meeting the SLO. In the case of homogeneous VM resources and consistent workload for each request, r_{max} can be calculated by using Equation (4.5). In the case of variable workload, adaptive controllers (e.g., PID [74])

can be used to set r_{max}. A proportional–integral–derivative (PID) controller can adapt r_{max} based on the error between the target and measured response/service time.

In lines 7-8, we obtain the number of VM instances that are needed to cater to a demand $vm_{reqs} = avg + \phi \cdot std$, as instantaneous requests beyond that value are considered transient and will be handled by serverless functions. The functions *add_vms* and *remove_vms* (lines 11 and 13) implement the VM-cloud (IaaS) interface to allocate or deallocate VMs to achieve the desired *num_instances* (line 8). The initial provisioning of VMs is performed based on user configurations similar to other autoscaling services [5]. Moreover, if the decision is to add more VMs, the *Scaling Manager* waits until the VMs are in a ready state before sending a vm_{reqs} update to the *Load Balancer*.

5.1.3 Load Balancer

The Load Balancer (LB) act as a reverse proxy for the application and receives requests from the end users and forwards them to the appropriate deployments, either VMs (IaaS) or serverless (FaaS). It also keeps track of the requests received in an epoch and periodically notifies the *Traffic Monitor*. Moreover, whenever the *Scaling Manager* makes a scaling decision, it reports the new value of vm_{reqs} to the *Load Balancer* as the *Scaling Manager* provisions VMs to accommodate a request rate of vm_{reqs}. From queuing theory [55], to ensure stable (predictable) performance and small queuing delays, the request rate to the provisioned VMs should be lower than the service rate given by the VM provisioned rate of vm_{reqs}. This keeps the aggregate utilization of the provisioned VMs below one. Consequently, our LB directs only a fraction ρ of the request rate, *i.e.*, $\rho \cdot vm_{reqs}$ to the VM resources. *This fraction ρ of provisioned VM capacity that can be used to serve requests is a configurable parameter of LIBRA and discussed in detail in Section 5.1.4.*

The LB adopts a forwarding approach that directs requests to VMs (IaaS) first, which has two key benefits: 1) The VMs are already in a ready state and will not incur any cold-start delays, and 2) ready VMs are cheaper compared to FaaS.

5.1.4 LIBRA Parameters

Our LIBRA approach has the following configurable parameters that an administrator can tweak to maximize their gain whether it is performance, cost, or both. We studied the behavior of these parameters in simulation and experimentally, and here we briefly summarize the effect of the following parameters and their recommended settings.

EWMA Weights

The *Traffic Monitor* in LIBRA uses EWMA to monitor the average rate of requests and sample deviation. The weights α and β given to the most recent number of requests observed over the current epoch are configurable parameters. A high weight value can lead to a quick response to a sudden increase in demand, resulting in over-provisioning of VM resources if the increase were transient. On the other hand, a low weight value can lead to a slow response to a sudden increase in demand, resulting in under-provisioning of VM resources if the decrease were persistent, which increases the usage of serverless functions and results in a higher cost.

Scaling Decision Interval

The *Scaling Manager* makes scaling decisions every K epoch. A smaller value of K (*e.g.*, less than the startup delay of VMs) can result in back-to-back scaling decisions without waiting for the system to react and reach equilibrium. A large value of K results in longer intervals between scaling decisions, hence slower adaptation leading to missing potential cost savings. The scaling decision interval should be larger than the startup delay of the VM instances being used. This is because when the *Scaling*

Manager makes the decision to scale out, it waits for the newly added VM instances to be in a ready state before informing the *Load Balancer* of the newly provisioned VM capacity. In our experiments, we set the scaling decision interval to be 3x the VM startup time (cold start).

VM Utilization

As described in Section 5.1.3, the *Load Balancer* only utilizes a certain proportion (ρ) of the provisioned VMs. The goal is to make sure that the VMs serve the requests without SLO violations (cf. Section 5.1.3). In our experiments, we set this parameter ρ to 80%.

Traffic Estimation

To estimate traffic demand, LIBRA uses the EWMA and sample deviation to obtain $(avg + \phi \cdot std)$, where $\phi \in \mathbb{R}$ and its value depends on the fluctuations in demand. In LIBRA, IaaS resources are provisioned for $(avg + \phi \cdot std)$ demand. Hence, a higher value of ϕ will cause more aggressive provisioning of VMs, which can potentially lead to VM under-utilization. On the other hand, a lower value of ϕ can lead to more FaaS usage which can potentially lead to higher cost as serving requests by FaaS is expensive.

5.2 How to deploy LIBRA?

LIBRA can be provided as a value-added service from the cloud provider similar to current load balancers [11, 28] offered by cloud providers. This can allow application developers to use multiple deployments of the applications. LIBRA can also be deployed by developers directly, but this can introduce the extra cost of deploying LG on cloud resources.

In the next chapter, we discuss the implementation details of LIBRA and our evaluation both in simulations and on AWS.

Chapter 6

LIBRA Evaluation

LIBRA closely monitors the demand for an application and consequently provisions VM resources, while the transient spikes and a small portion of the demand are served by FaaS deployment. This approach results in little to no SLO violations while also reducing the cost of cloud usage for the tenant. To evaluate the long-term efficacy of LIBRA, we evaluated [1] our approach in simulations as well as on AWS. We compare the cost of cloud usage, performance, and resource utilization (VM uptime) with other deployment strategies.

6.1 Simulation Model

We modeled various cloud services after Amazon Web Services (AWS) [4]. These include IaaS, FaaS, Load Balancer, and Autoscaler. Using real traces, we evaluated LIBRA against different approaches: VM over-provisioning, FaaS-only, and provisioning of VM resources using the autoscaler.

6.1.1 Modeling Cloud Services

We modeled IaaS and FaaS (and related services) after AWS EC2 and AWS Lambda, respectively.

IaaS: Our modeled IaaS has various resource types to offer for application deployment. Different VM instance types have different cold-start delay depending on

[1] The LIBRA simulator and AWS experiment codes are available at [36].

the size of the instance and resources such as memory. Moreover, our pricing model follows the AWS EC2 pricing model, where users are charged based on partial usage, *i.e.* on seconds basis as specified in [21]. Any instance can host a pre-defined number of requests based on the resources available in the instance and desired SLO. Hosting more requests on an instance can lead to performance degradation and potential SLO violations. The usage cost is calculated according to Equation (4.3).

Load Balancer: Production-ready applications typically use more than one VM. Incoming requests are distributed among them in a Round Robin fashion. If all the VM instances already have a pre-defined number of requests running, any subsequent request is queued and served as soon as any instance can accommodate it.

Autoscaler: Our modeled autoscaler works similarly to Amazon EC2 Auto Scaling [5] and allows users to define auto-scaling policies such as scale-in/scale-out thresholds, scaling groups, and minimum/maximum number of instances. Moreover, our autoscaler can use a threshold on metrics, such as average memory utilization or request count on each instance, to make scaling decisions.

FaaS: To model FaaS, we deployed various types of application functions on Amazon Lambda and found the relationship between the configurable resources (*e.g.*, memory) and execution time as shown by Equation (4.1). Other approaches (*e.g.*, [48],[80]) have reported similar execution patterns. We also use Amazon Lambda's pricing model where, based on the configured resources and execution time, usage cost is calculated according to Equation (4.2).

6.1.2 Simulation Parameters

For our evaluation, we chose the "large" EC2 instance type *m4.large*, which has 8.0 GB of memory and 0.1 dollars per hour cost. We ran multiple instances of type *m4.large* and noted that the provisioning time (cold-start delay) of an instance is about 100 seconds. Each request should have at least 512MB of memory to complete

in one second (SLO) on both IaaS and FaaS [2]. For LIBRA parameters, we used 0.2 as the EWMA weight, 300 seconds as the scaling decision interval (which is 3× the cold-start of the VM instance being used), VM utilization threshold $\rho = 80\%$, and traffic estimation parameter $\phi = 1$.

6.1.3 Log Traces

We used publicly available Waikato Internet Traffic Storage (WITS) [45] traces to evaluate the efficacy of LIBRA. These traces have also been used to evaluate similar approaches [77, 44]. We utilize a 12-hour-long segment from the traces to generate the workload for our simulation. Each request is assumed to have a constant and equal service (execution) time when served on either the IaaS or FaaS-based implementation (when no queuing delay occurs). Our simulator takes into account the cold-start of VMs under IaaS. We also assume that the serverless functions under FaaS have a minimal cold-start delay that would not affect the performance of an application with relatively high popularity as shown in [120]. Figure 6·1 shows the 12-hour snippet of a WITS trace of the number of requests per epoch (second), along with the EWMA. Traces were generated by counting the number of HTTP requests during every second in TCP dumps available at [45]. Despite the high variation of the demand, EWMA tracks the dynamicity of the trace well.

6.1.4 Resource Provisioning & Deployment Policies

As discussed in Section 5.1, LIBRA's main goal is to utilize both FaaS and IaaS to minimize the overall cost while at the same time meeting the performance (execution time) requirement of the application. LIBRA leverages the best of both cloud services: the quick provisioning time of serverless functions and the low cost of provisioned VM resources. We compare LIBRA's balanced approach to the following

[2]For IaaS, a user can specify resources for each container executing the request as documented in [34, 20].

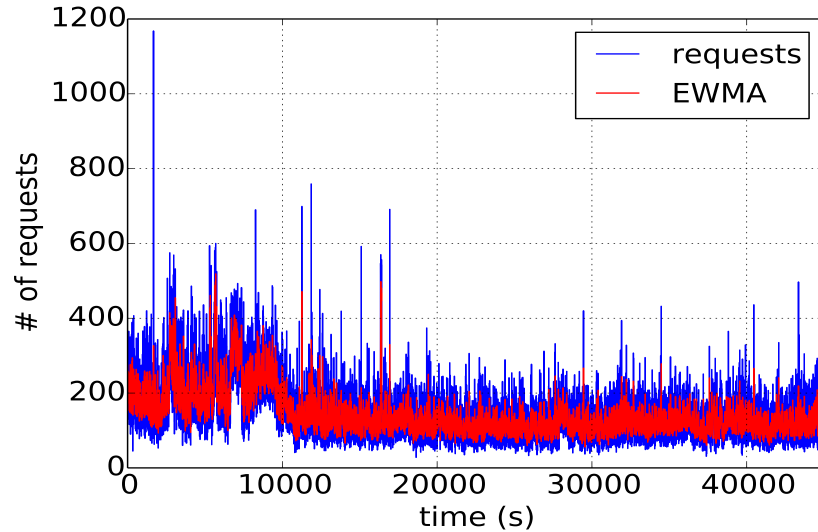


Figure 6·1: WITS trace and EWMA

policies:

Over-provisioning (MAX): Cloud applications have strict SLOs and not meeting their performance constraints can result in a bad user experience and potential loss of revenue. To avoid potential SLO violations, the tenant could opt to over-provision the VM resources. We simulate this scenario by scanning the whole demand trace and provisioning the VM resources based on the *maximum* number of requests received during a second (*i.e.*, the peak rate). While such an approach would avoid SLO violations, allocated VM resources will be underutilized and the client would incur higher costs.

Autoscaling (AUTO): Autoscaling is a popular service provided by major cloud providers. In Autoscaling, the performance of the currently allocated resources (*e.g.*, VMs), is monitored based on some metric. The performance metrics can vary based on the cloud provider, but some examples include memory/CPU utilization or request/connection count on each host. If the metric exceeds a certain threshold, new resources are added to the system to avoid the potential overloading of current resources and subsequent degradation of performance. If the metric falls below a certain

threshold, resources are removed to avoid under-utilization.

Spock: Previous approaches, such as Spock [77] and MArk [128] reduce SLO violations due to VM start-up delays by directing demand to serverless functions while VMs are being provisioned. Unlike LIBRA, Spock-like schemes do *not* consistently and simultaneously use serverless functions to serve transient demand and reduce overall cost. We simulated this by directing the *excess* portion of the demand to FaaS *during* scale-out events.

FaaS only: The application is deployed on a serverless platform and all requests are served by serverless functions.

6.2 Simulation Results

6.2.1 Cost and SLO violations

Figure 6·2 compares the cost and performance of the aforementioned resource provisioning policies with LIBRA. The x-axis represents the different approaches described above, the y-axis (left) represents the incurred cost normalized to that of LIBRA, and the y-axis (right) reports the percentage of SLO violations. LIBRA's cost also includes the cost of the VM instance used to deploy the LIBRA Gateway (cf. Figure 5·1). As expected, over-provisioning (MAX) leads to zero SLO violations but incurs the highest cost. The autoscaling approach (AUTO) reduces cost significantly but introduces significant SLO violations. This is due to the fact that VMs have high cold-start delays, and while a new VM is being set up to share the demand, existing VM instances get saturated, which leads to performance degradation and SLO violations. We note that we have experimented with various thresholds for the utilization metrics used by autoscaling. However, we have observed that either the system performs better at the expense of a much higher cost or the system has a lower cost at the expense of a much worse performance (higher SLO violations). Compared to autoscaling,

Spock reduces SLO violations by more than 35% at about the same cost. Notice that consistent with the original study [77], Spock reduces SLO violations but does not completely eliminate them. LIBRA yields the lowest cost – 15% less cost than autoscaling/Spock and cuts the cost by more than half compared to serverless-only (FaaS) or over-provisioning (MAX). In our simulation, we assume that a serverless function has resources configured correctly so that each request always meets the SLO [48]. Thus, a serverless-only deployment yields zero SLO violations albeit at a higher cost. Similarly, LIBRA yields zero SLO violations. However, LIBRA reduces the overall cost by always directing a portion of the demand to VMs that are provisioned to meet the SLO, while the rest of the demand is directed to serverless functions that are also configured to meet the SLO.

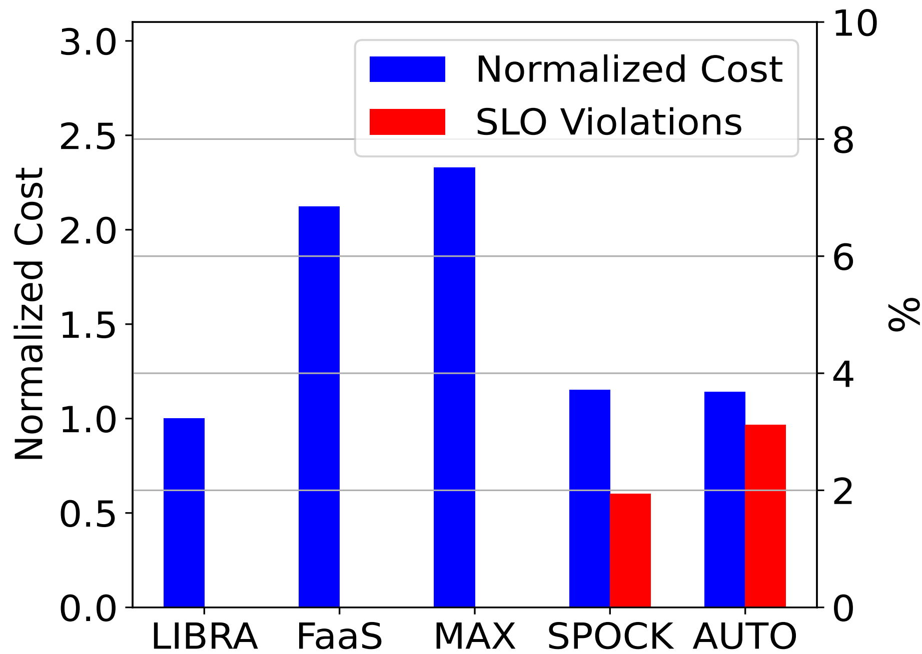
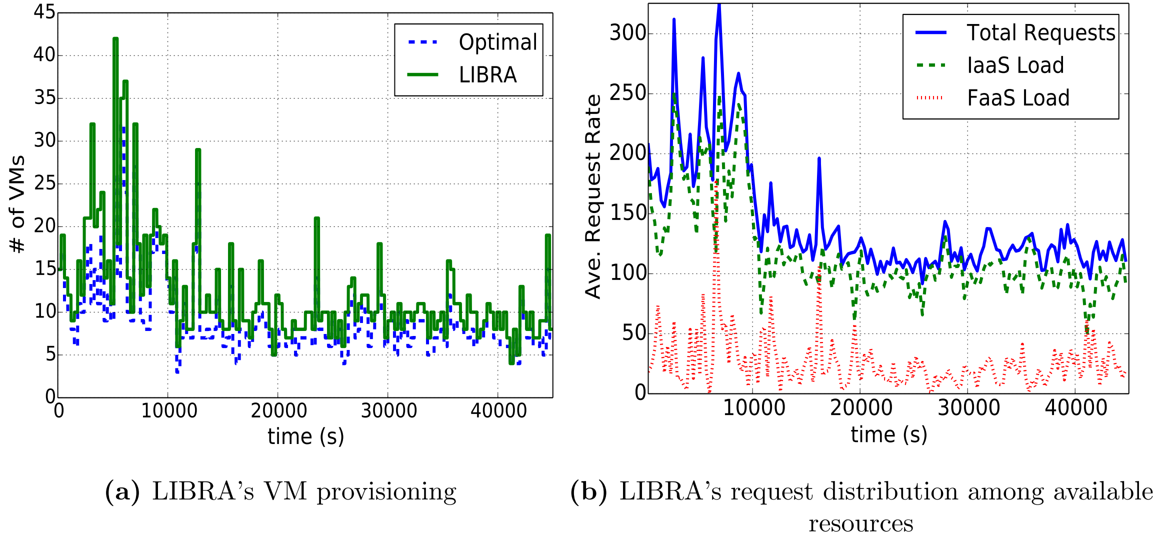


Figure 6·2: LIBRA and other resource provisioning policies



(a) LIBRA's VM provisioning

(b) LIBRA's request distribution among available resources

Figure 6·3: VM provisioning and request distribution of LIBRA.

6.2.2 VM Provisioning & Request Distribution

LIBRA's main goal is to cautiously provision resources in the VM cloud (IaaS) to avoid under/over-utilization while simultaneously serving low-rate and sudden spikes in demand using serverless functions (FaaS). Figure 6·3a shows how LIBRA accurately tracks the incoming load, provisions VM resources, and avoids over-provisioning.

This is observed by the similar behavior of LIBRA in terms of the number of VMs provisioned (green solid curve) throughout the duration of the simulation and the *ideal* (offline) case (blue dashed curve) of provisioning the number of VM instances assuming perfect knowledge of future demand. The points on both curves represent scaling decisions taken every $K = 300$ seconds (cf. Section 5.1.4).

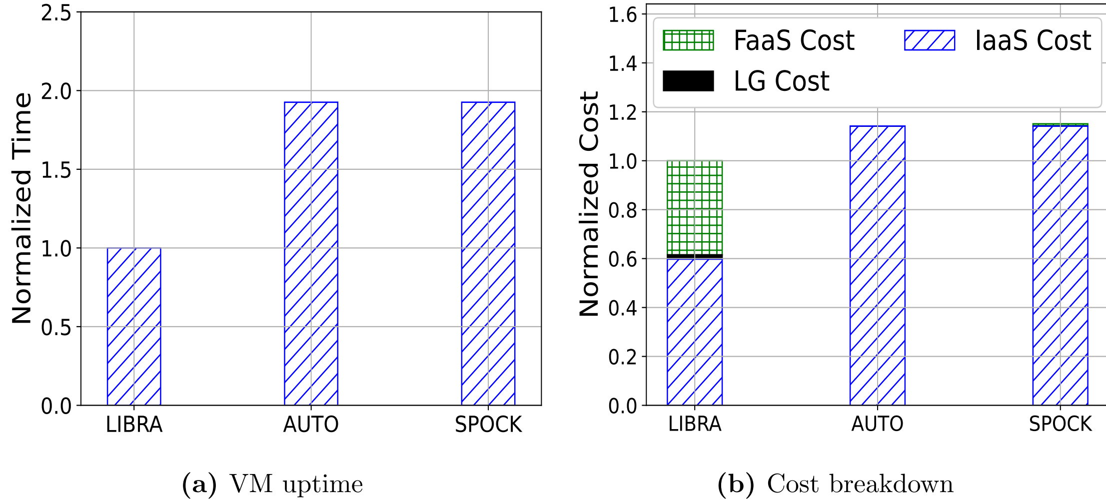
Figure 6·3b shows the average rate of requests for the portion of the demand forwarded to the VM instances (IaaS) and the rest of the demand directed to serverless functions (FaaS) every 300 seconds. We observe that a consistent majority of the load is served by VMs (IaaS) whereas a small amount of the load with temporary peaks is handled by serverless (FaaS).

6.2.3 VM Uptime & Cost Breakdown

In Figure 6·4a, we compare the total uptime of all VMs used to run the application under various approaches. LIBRA cuts the VM uptime by half compared to autoscaling and Spock. This is because (1) LIBRA only scales out when the demand persists for a longer time. LIBRA is able to identify transient demand and avoid reacting to it by using serverless functions (FaaS) rather than adding new VM instances (IaaS); and (2) LIBRA can add any arbitrary number of VMs to the active VMs when scaling out, while autoscaling adds or removes a user-configured number of instances (referred to as "scaling group"). Figure 6·4b compares the cost breakdown across autoscaling, Spock, and LIBRA. Autoscaling has zero FaaS cost since VMs are the only resources used to serve the application demand. Spock employs FaaS when scaling out, only to hide VM startup delay, so the FaaS usage is really small ($\approx 1\%$). LIBRA consistently uses serverless functions to serve a portion of the demand. The FaaS cost contributes around 40% of the total cost in LIBRA's case. Despite higher FaaS cost, the overall cost of LIBRA is smallest. LIBRA intelligently uses FaaS for a portion of the demand that is either below CIP or transient, since the cost of new VM instances for that portion of the demand would have been higher. Note that the cost of LIBRA includes that of the LIBRA Gateway (LG), the added cost of running the LIBRA system.

6.2.4 Traffic Estimation

Recall that LIBRA provisions IaaS resources for a request rate of $vm_{reqs} = (avg + \phi \cdot std)$. The actual request rate directed to the provisioned VMs is $\rho \cdot vm_{reqs}$ ($\rho < 1$), while the remaining requests (in each epoch) are directed to FaaS where they are served within the SLO but at a higher cost. The value of ϕ can be adapted based on the particular fluctuations in demand. Tuning ϕ affects the cost but not the performance of an application. This is because the SLO is met whether LIBRA



(a) VM uptime **(b)** Cost breakdown

Figure 6·4: VM usage and cost breakdown

directs the request to the *provisioned* VMs or to FaaS, however, FaaS is more costly. For the WITS traces used in our evaluation, the effect of different values of ϕ on the cost is shown in Figure 6·5. We observe that a lower value of ϕ leads to more FaaS usage and hence higher overall cost, whereas a higher value of ϕ causes over-provisioning of VMs, which leads to VM under-utilization and hence higher cost. Here, $\phi = 1$ gives the least cost.

6.3 AWS Results

To validate our simulation results from Section 6.1, where LIBRA was shown to be effective in reducing both cloud-usage cost and SLO violations, we implemented LIBRA to perform load balancing for an application deployed on Amazon AWS Cloud, *i.e.* EC2 for IaaS and Lambda for FaaS.

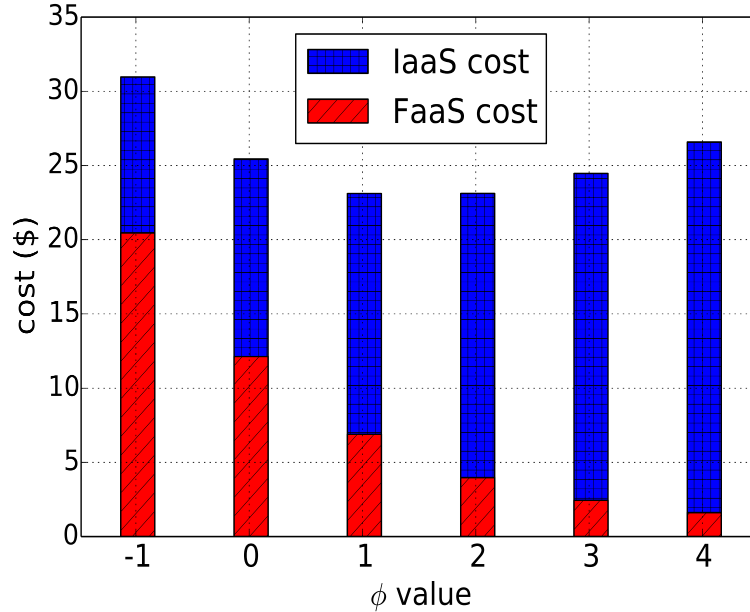


Figure 6·5: Cost for different values of ϕ

6.3.1 Implementation Details

Application

The application used is an image manipulation application written in Python. On Amazon Lambda, we are able to deploy the application as a single function. While we expect similar results for multi-function applications, our choice of single-function application is inspired by many use cases such as ML inference models [77, 128, 80], IoT and computer vision applications [3] which can be usually deployed as a single function. To deploy the application on EC2 VM instances, we used a Python multi-threaded HTTP server library. We ran it on a *t3.medium* [21] EC2 instance type with Ubuntu Server 18.04 LTS operating system, two 2.5 GHz vCPU, and 4 GB memory.

Application Profiling & Lambda Resources

As described in Section 5.1.2, LIBRA's *Scaling Manager* uses the maximum number of requests, r_{max}, that a VM instance can handle, to calculate the required number

of instances. To obtain r_{max} for this evaluation, we deployed our application on a *t3.medium* instance, profiled its performance, and obtained r_{max} that meets the SLO. We take the SLO to be one second of execution time serving a request. Profiling an application on a given VM instance is a one-time task and a developer can perform this prior to production deployment. For FaaS deployment, we invoked the function with various memory configurations and picked the memory setting that gave the least cost while meeting the SLO.

Setup & Implementation

To obtain a consistent network environment for our evaluation on AWS, we deployed an application client on an EC2 instance, which will generate HTTP requests for the application. We compare LIBRA to the same four resource provisioning and deployment strategies described in Section 6.1.4. We needed to modify the implementations of LIBRA, Autoscaling, and Spock, as follows:

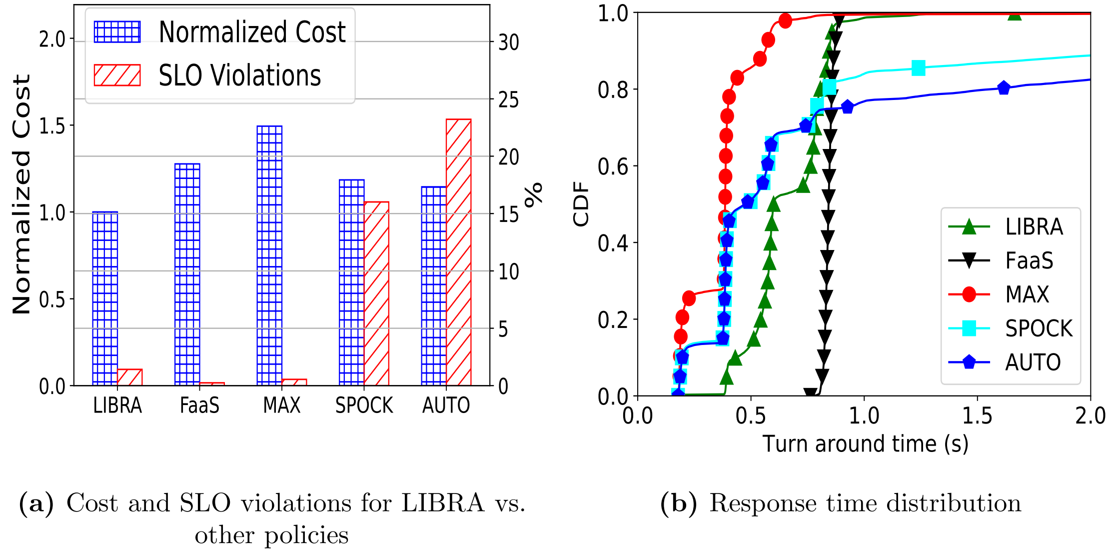
LIBRA on AWS: We deployed the LIBRA Gateway (LG) on an EC2 instance of the type *t2.micro*. The LG distributes requests between IaaS based resources (EC2 VMs) and FaaS (AWS Lambda functions). Within IaaS, we used the AWS Application Load Balancer (ALB) [11] to distribute requests among the active VM instances evenly, *i.e.* in a Round Robin fashion.

Autoscaling on AWS: We used EC2's autoscaling service, which is a threshold-based scaling service. The AWS CloudWatch Alarm was used to monitor *RequestCountPerTarget* metric to make scaling decisions. Again, ALB was used to distribute requests to the VMs.

Spock on AWS: An alarm from AWS CloudWatch triggers the scale-out or scale-in events. When a scale-out event is triggered, new VM instances are provisioned.

During this VM provisioning time, the system sends the extra requests, that cannot be served by the active VM instances, to the FaaS deployment. Once the new VM instances are ready, all requests are forwarded to the VMs. Thus, Spock attempts to use FaaS deployment only to hide VM startup delays[3].

Traces: For our AWS experiments, to reduce real load, we use a scaled-down version of the first 1800 seconds of the WITS trace shown in Figure 6·1. In particular, we reduce the rate of requests by a factor of 16.



(a) Cost and SLO violations for LIBRA vs. other policies

(b) Response time distribution

Figure 6·6: Performance of LIBRA on AWS

6.3.2 Cost and SLO Violations

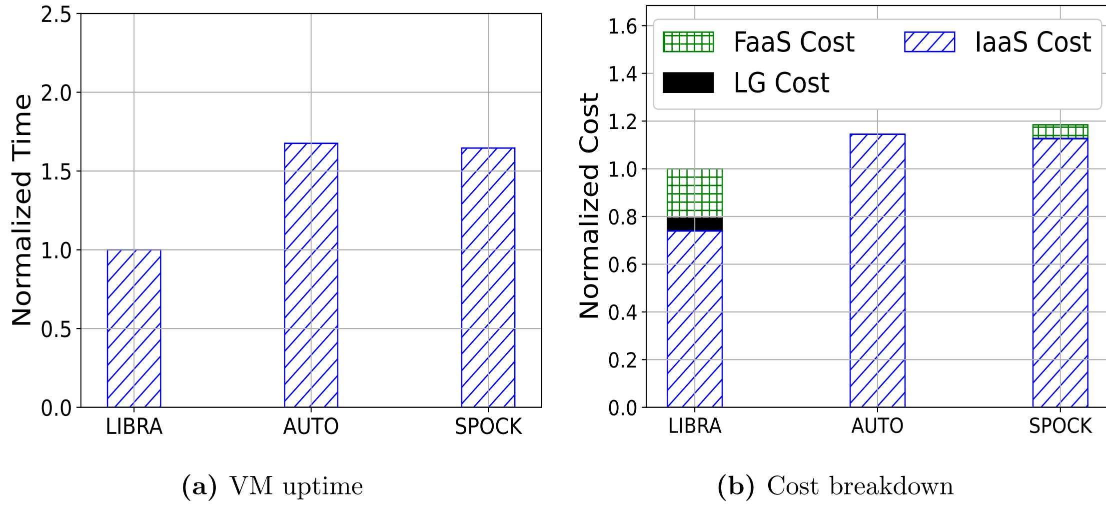
We compare the cost and performance of LIBRA versus other resource provisioning and deployment strategies. The results are consistent with our simulation results. Figure 6·6a shows that LIBRA yields the lowest cost with very low amount of SLO violations. LIBRA reduces the SLO violations (by more than 85%) and cost (up to 20%) when compared to auto-scaling and Spock. LIBRA's cost also includes

[3]Note: Spock [77] was only evaluated in the simulation environment and do not provide implementation details on real cloud provider.

the cost of deploying the LIBRA Gateway on an EC2 instance. Max-provisioning and serverless-only deployment yield the lowest SLO violations but incur up to 50% increase in cost. We observe that FaaS, LIBRA, and max-provisioning, all have a little amount of SLO violations. This is because, unlike our simulation model, in a real setup, factors such as co-location, cold-starts for serverless functions, and underlying resource contention for VMs, can introduce slight variations in the performance of an application. For LIBRA, a lower value for the VM utilization parameter ρ (discussed in Section 5.1.4) can mitigate these SLO violations.

While Spock reduces SLO violations by 40% compared to autoscaling, about 15% of the requests fail to complete within the SLO. This can be explained by Spock's reactive scheme, where scaling out is triggered when VM resources are saturated, resulting in SLO violations. On the other hand, LIBRA avoids saturating the VM instances by directing excess load to serverless functions. If the load is not transient and the demand stays higher for a longer period, LIBRA increases the number of VM instances at the next scaling decision.

Figure 6·6b shows the Cumulative Distribution Function (CDF) of the completion time of each request. The over-provisioning policy gives the best performance in terms of keeping execution time really low, as expected. FaaS has the most consistent performance, *i.e.* the completion time is always between 0.8s to 1s. This is due to the fact that each request is executed in a dedicated sandbox environment with dedicated resources, such as memory, so the chance of performance fluctuation is lower. The performance of a serverless function is primarily affected by cold-start delay [80], which is negligible and can be as low as 10s of milliseconds for serverless functions written in Python [120].



(a) VM uptime **(b)** Cost breakdown

Figure 6·7: VM usage and cost breakdown

6.3.3 VM Uptime & Cost Breakdown

Figure 6·7 confirms the benefit of LIBRA by illustrating the cost breakdown. This is consistent with our simulation results. Figure 6·7a shows the uptime of VM instances used to run the application for Spock, Autoscaling, and LIBRA. LIBRA is able to closely monitor the demand and provision required VM resources without over-provisioning, which results in lower overall VM uptimes and cost. Figure 6·7b shows the cost breakdown of these approaches. LIBRA has the lowest overall cost (including the cost to deploy LG), with the lowest IaaS cost but the highest FaaS cost. LIBRA uses serverless functions more consistently and effectively, *i.e.* for transient demand or portion below CIP, which results in higher usage of FaaS, and lower usage of IaaS.

6.4 Discussion & Future Work

6.4.1 Development Overhead

Employing multi-deployment approaches such as LIBRA can add additional development costs, where the application developer may have to maintain multiple versions of the application, one of each deployment. It can also add to the CI/CD cost. This cost can be significantly reduced by building containerized applications as most services including FaaS and IaaS can support containerized deployments. Moreover, the benefit of using LIBRA, both in terms of cost and performance, can potentially amortize these additional costs.

6.4.2 Performance Overhead & LIBRA Scalability

LIBRA works as a legacy load balancer and directs requests to appropriate resources, introducing overhead no more than legacy load balancers. At the same time, other LIBRA operations, *i.e.* scaling and traffic monitoring, occur in the background and do not impact the real-time processing of requests. The LIBRA Gateway has a small computational footprint, and hence a small cost. Various components of the LIBRA Gateway can be implemented as serverless functions, where scalability is taken care of by the cloud provider. Alternatively, the LIBRA Gateway can be implemented as one service and deployed using VM instances and the application administrator can rely on the cloud provider's scaling services such as AWS auto-scaling.

6.4.3 Application Availability

While in the current version of LIBRA, we are only using the most reliable services i.e. IaaS (AWS provides an SLA of 99.9% availability [13]), and even though FaaS does not provide any such guarantee, it has been found extremely reliable [120]. In case a developer wants to leverage various cheaper IaaS-based resources such as

Spot Instances, the application availability can be impacted if LIBRA's parameters are not configured properly. Spot Instances are greatly discounted IaaS resources, which have variable availability and can be interrupted by the cloud provider with a warning (usually a 2-minute notice [14]). In the next section, we discuss future work for LIBRA, and how to leverage Spot Instances efficiently, while not compromising the overall cost or performance of the application.

6.4.4 Leveraging Spot Instances

Spot instances are greatly discounted IaaS resources, which are offered by most cloud providers during times of low utilization of the resources in the data centers. These resources have been previously leveraged in conjunction with the on-demand resources to minimize the overall cost [124, 87]. We believe that LIBRA can achieve further cost savings if spot instances are also used for the IaaS deployment of the application. In such a scenario, the cost can be estimated as follows:

$$\mathbb{E}(c) = \omega \times d + (1 - \omega) \times f \tag{6.1}$$

Equation 6.1 estimates the expected cost when utilizing spot instances in conjunction with on-demand (full-price) instances. ω is the probability of obtaining a spot instance, while d and f are discounted and full prices of the instances, respectively. Previous studies [103] have reported that the probability of obtaining a VM instance ω can be affected by the region from where the instance is requested and the success probability can be as high as 99%. Similarly, the cost saving is also variable but most spot instances are up to 90% cheaper as compared to on-demand instances [1]. Substituting these values in Equation 6.1, the expected cost is 89% lower than the on-demand instances cost. Usage of spot instances in LIBRA can greatly reduce the cost but introduces the additional challenge of dealing with the untimely interruption

of the spot instance, which in turn can affect the application's availability as well as more usage of FaaS (as the number of provisioned VMs goes down), which may lead to increase in cost. In our future work, we plan to address these challenges, so that the cost savings are maximized while not compromising on the availability of the application.

Chapter 7

xCOSE

To employ FaaS efficiently for a serverless application, a developer has to make certain (one-time and online) decisions (*discussed in Chapter 2*). We discuss the effect of various one-time decisions on performance in our previous work [105]. We believe previous measurement studies and benchmarking tools can help with making these decisions. In this chapter, we focus on online configuration decisions as they are easier to make on the go and can affect the cost and performance of serverless applications as discussed in Chapter 2. FaaS platforms provide users with limited configurable parameters, such as memory, CPU, and location, for a serverless function. In Chapter 2, with the help of our experimental study, we show that these configurable parameters can affect the cost of cloud usage and the performance of serverless applications. As FaaS platforms do not provide any guarantee (SLO) on the performance, configuring the parameters becomes even more crucial to get the desired performance of an application and optimize cost.

Even though there are only a few parameters such as memory, CPU, and location, configuring these parameters for serverless applications cost-effectively while obtaining desired performance can be a challenging task for the following reasons:

Dynamic Execution Model: In the FaaS model, each time, the code of the application is executed on arbitrary resources of the cloud provider's choosing. This can introduce a certain variability in performance even when configured with the same amount of resources. Moreover, other factors such as colocation and cold starts

can introduce certain variability in performance.

Varying Pricing Model: FaaS offerings have evolved over the years and certain providers, such as AWS, allow their users to deploy serverless applications in different regions and locations (edge or core). Different locations have two implications: 1) Latencies: Depending on the end user, the location of deployment can significantly affect the user-perceived latencies as edge deployment would be quicker to access but usually cloud providers do not have many resources available at the edge and can only perform lightweight computations. 2) Cost: Pricing schemes can also vary from location to location. For example, AWS's Lambda@Edge (FaaS at the edge) can cost as much as three times as compared to core resources [7, 10].

Complex Applications: Serverless applications are usually a composition of multiple stateless functions (service graphs or linear chains) and a user has to configure resources and placement for each function individually. In this case, a user can still meet the end-to-end performance requirement of the application by trading off the performance of some of the functions for lower cost or reducing access latencies by deploying certain parts of the applications on the edge.

7.0.1 Ideal Configurator

Considering the above challenges, we believe an ideal configuration and placement strategy should address the following four aspects:

- ***Sampling Cost:*** Sampling the performance of an application for all possible configurations can be expensive as AWS Lambda alone offers tens of thousands of configuration options (memory values, edge or core locations, regions).

- ***Dynamic Adaptation:*** Serverless applications can be running in varying conditions based on the data-center resources and cloud provider's policies. An ideal strategy should adapt according to the situation as one-time configurations

may not always be optimal.

- ***Placement:*** FaaS platforms allow developers to decide the location of deployment for each individual function in an application. These locations can be different clouds (regions), edge, or core -clouds. An ideal approach should also help to find the best placement for each function keeping in view the access latencies for a different location to meet the end-to-end latency requirements.

- ***Service Graphs:*** An ideal approach should be able to find the best configuration for complex applications as optimizing individual functions' performance and cost may not lead to global optimal.

7.1 Background: COSE

Configuring serverless applications can be a challenging task and traditional optimization methods such as Additive Increase Additive Decrease (AIAD) and exhaustive search over possible configurations may not be feasible for sub-optimal performance or added cost [49]. Our previous work COSE [48, 49] addresses the challenge of configuration and placement for serverless applications deployed over FaaS. COSE consists of the following two main components as shown in Figure 7·1.

7.1.1 Performance Modeler

The *Performance Modeler* component of COSE leverages the statistical learning-based technique, Bayesian Optimization, to build a performance and cost model of serverless functions (a serverless application consists of multiple functions) over the different possible resources (memory and CPU) and placement (different regions or edge and core) configurations. Bayesian Optimization is usually used to learn black-box functions. In our case, the function $g(x)$ that we want to learn is the relationship between a serverless function's performance/cost and all possible configurations x,

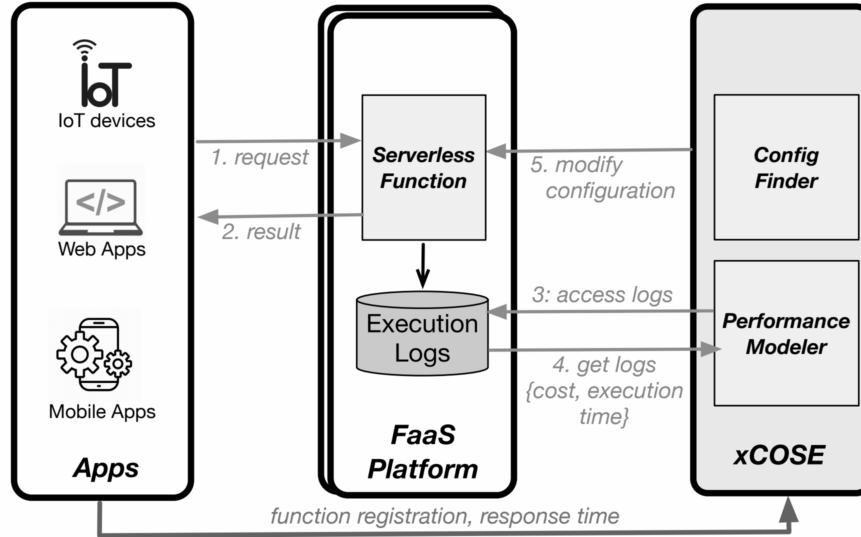


Figure 7·1: xCOSE Architecture

where each x is in the form of (v, m) where $v \in V$ is the set of possible locations and m is the possible amount of resource such as memory, $m \in M$ specified by the FaaS provider. In our previous work [104], we show that the *Performance Modeler* can successfully learn this relation with minimal samples and can adapt to any changes in the underlying execution model. The *Performance Modeler* learns this relationship for each serverless function in an application.

7.1.2 Config Finder

COSE assumes that serverless applications consist of linear chains of functions. The *Performance Modeler* module of COSE, after learning the execution and cost model of each individual function, passes this information to *Config Finder*. It then uses Integer Linear Programming (ILP) to find the best configuration for all functions such that the cost is minimized and the SLO is met.

More details about COSE and its evaluation are presented in our previous work [48, 49].

7.2 xCOSE

In this thesis, we present xCOSE, an extended version of our previous tool COSE. xCOSE employs a updated version of *Config Finder*, which can accommodate service graphs. We make the following additional contributions to COSE [48]:

- Our previous work [48] could only find configurations for applications consisting of linear chains of functions. In reality, applications can also form Directed Acyclic Graphs (DAGs) of functions. In xCOSE, we extend the capabilities of our previous work to find the configuration and placement for such applications.
- We perform more rigorous evaluations of xCOSE using real-life applications on AWS Lambda and Lambda@Edge [7] (FaaS offering at the edge).
- Ee compare xCOSE with some of the state-of-the-art solutions proposed and discuss its advantages.

The xCOSE architecture is similar to COSE as discussed in the previous section (as shown in Figure 7·1) except it uses an updated version of *Config Finder* while using the *Performance Modeler* as discussed in the previous chapter. In what follows, we discuss the updated *Config Finder* component and how it can accommodate serverless applications consisting of service graphs.

7.2.1 Config Finder

The *Performance Modeler*, after learning the $g(x)$ (performance model) for each function f in an application, passes this information to the *Config Finder*. We assume the application forms a directed acyclic graph G. The *Config Finder* then solves this problem of finding suitable configurations in two steps. First, it identifies the set of all possible paths P from the start function in the graph (source) to the end function

(sink)[1]. It then uses Integer Linear Programming (ILP) to find the best configurations for all functions such that the cost is minimized and the SLO is met. *Note that to solve for a single-function application, the service graph degenerates to a single node.*

Integer Linear Programming (ILP)

We assume that for each serverless function f in graph G, we choose a configuration consisting of the cloud provider $v \in V$ and memory $m \in M$, such that the total cost is minimized and for each path, p in P the delay constraint (SLO) D is satisfied.

The objective of the *Config Finder* is to minimize the total price paid for running all the functions of the application. This is given by:

$$minimize \left(\sum_{f \in G} \sum_{x \in C} g^f(x) Y_x^f \right) \tag{7.1}$$

subject to:

1) Every path meets SLO so as to guarantee that the whole application will execute within SLO:

$$\sum_{f \in p} \sum_{x \in C} T^f(x) Y_x^f \leq D \quad \forall p \in P \tag{7.2}$$

where $T^f(x)$ is the predicted end-to-end delay for running serverless function $f \in p$ using configuration $x \in C$.

2) A single configuration $x \in C$ is selected for each serverless function in the application.

$$\sum_{x \in C} Y_x^f = 1 \quad \forall f \in G \tag{7.3}$$

The solution to this problem yields a least-cost feasible solution, *i.e.* the resulting Y_x^f, that gives the configuration x of each serverless function f in the service graph.

[1] This is typically a one-time path computation as the service graph will only change if the developer alters the application.

We evaluated xCOSE in simulations as well as on AWS Lambda. More details about the evaluation of xCOSE in a simulated environment can be found in [104]. In the next chapter, we discuss xCOSE's evaluation on AWS with real applications. We also discuss various related works.

Chapter 8

xCOSE Evaluation

In this chapter, we discuss the evaluation of xCOSE on AWS Lambda using two real applications[1]. We also discuss some of the closely related works to xCOSE and how xCOSE can be easier to use and has less sampling cost, while giving a similar performance.

8.0.1 Image Processing workflow

To evaluate the *Config Finder*'s ability to find the right configurations while minimizing cost for service graph applications, we use the Image Processing workflow [42] shown in Figure 8·1. This application generates a thumbnail of an image uploaded to the S3 database. It first makes sure the image has a face (F1) and not a duplicate (F2) then in parallel, it indexes the face (F3) and creates the thumbnail (F4). The last step is to persist metadata. This application creates four serverless functions implementing the above functionality and was deployed using AWS Step Functions [15].

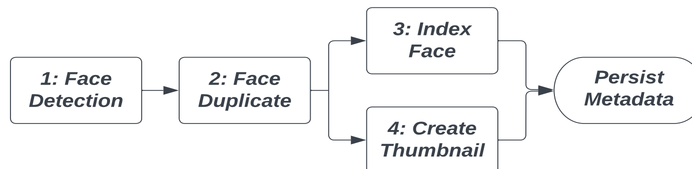


Figure 8·1: Image Processing Workflow

[1]xCOSE code can be found at [46].

Evaluation Results: We ran the Image Processing workflow with various memory configurations of the four serverless functions to obtain each function's behavior with respect to changes in memory allocated and then performed curve fitting to obtain function profiles as shown in Figure 8·2. Note the range on the y-axis is different for each function as they follow different execution behaviors. It can be observed that the minimum achievable SLO is around 1346.91 ms and indeed when we ran our *Config Finder*, we obtained the following memory allocations $\{F1 : 701, F2 : 3000, F3 : 232, F4 : 3000\}$. We also ran our *Config Finder* with other SLOs and the results are shown in the figure – as we relax the SLO, the cost goes down as expected.

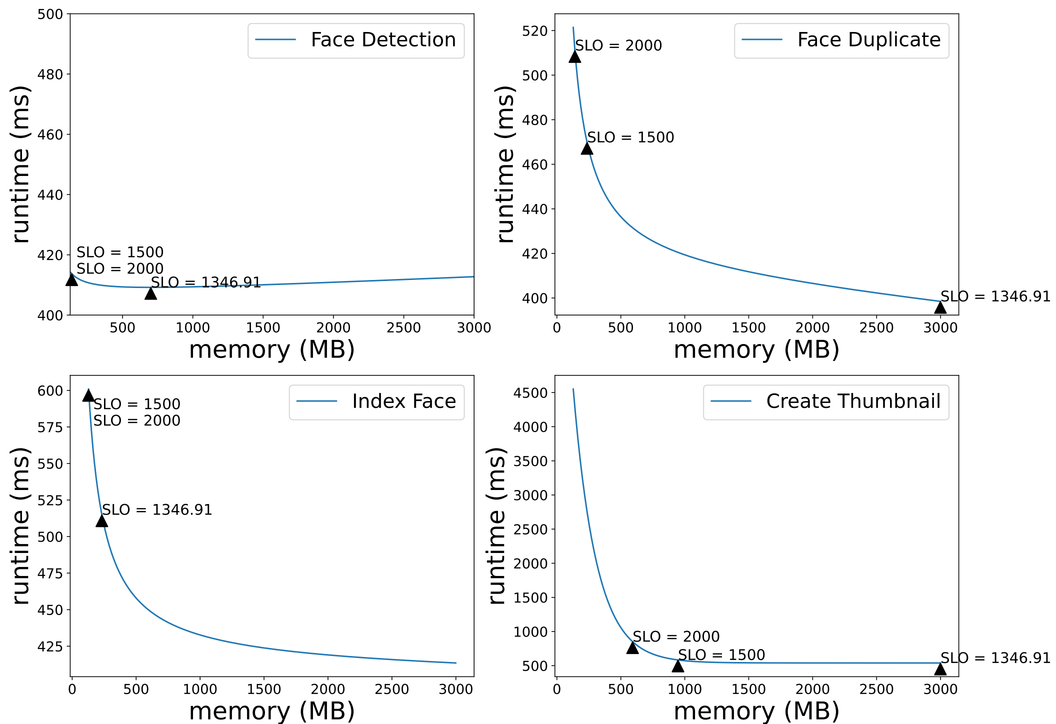


Figure 8·2: Function performance w.r.t. memory allocated

8.0.2 Personal Protective Equipment

We next evaluate xCOSE on a multi-functions Personal Protective Equipment (PPE) detection application, which upon receiving an image from an end device, such as

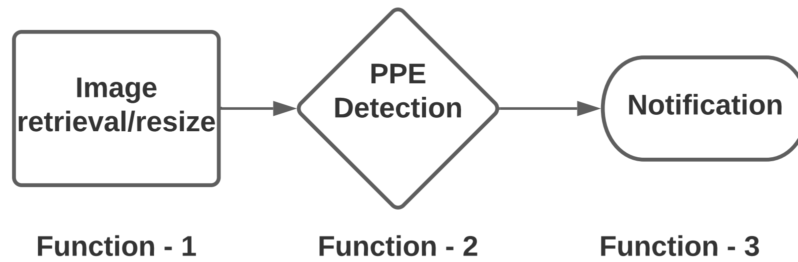


Figure 8·3: PPE Detection Application

a surveillance camera, performs PPE detection, and notifies authorities if anyone violates PPE policy. The application's workflow is shown in Figure 8·3. It consists of three main functions: a) image preprocessing, b) detecting PPE, and c) notification. All three functions were implemented using Python 3.7 and utilize various AWS services such as S3 for storage, *Amazon Rekognition* to perform PPE detection, and *Simple Notification Service* (SNS) for notifications. For edge deployment, we used AWS Lambda@Edge. AWS Lambda@Edge has some key characteristics: i) the cost of running a function on the edge can be 3x the cost of running the function on the core [7], ii) propagation delays are much smaller in the case of Lambda@Edge, and iii) resources at the edge can be limited. In our case, the core region was AWS region *us-east-1*, and the end client was placed on a university campus in *Minneapolis, Minnesota*. Lambda@Edge would execute deployment closer to the user. Note that the current offering of Lambda@Edge supports only certain types of computation (in response to events generated by the Amazon CloudFront content delivery network (CDN)) and we believe cloud providers should enhance their service by providing a more general-purpose computation utility. For our experiment, we executed our functions on the edge using the *origin-request* trigger as this is the only viable trigger type that can configure large memory and intercept the request at the edge.

Evaluation Results: Given the *Performance Modeler* has learned the cost and

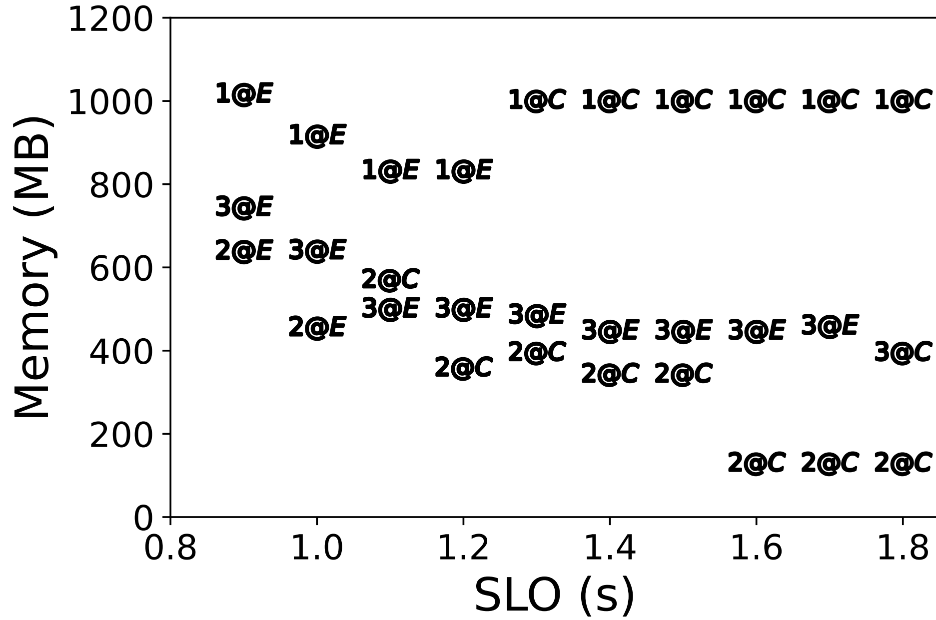


Figure 8·4: Memory and location w.r.t. SLO

performance relations of all the functions, in Figure8·4 we show the memory and location configuration for various SLOs. SLO is measured as the time from the first request is sent and the final response of function-3 is received. Each marker is in the form *function_id@deployed_at* where *function_id* can be 1, 2, or 3 corresponding to Figure 8·3 and *deployed_at* can be edge (E) or core (C). Each column shows the SLO on the x-axis and the corresponding functions' memory configuration and placement on the y-axis. Similar to the simulation results, it can be seen that initially, when the SLO is strict, xCOSE places all the functions on the edge because the edge is closer to the end user (despite being more expensive) but as the SLO is relaxed, xCOSE first reduces the memory allocated to reduce the cost, but as SLO is further relaxed, xCOSE places the functions on the core given the core is cheaper.

Cost & SLO Tradeoff: In xCOSE, SLO dictates the overall cost, and stricter SLO can be met with higher cost. In Figure 8·5, we show the total cost and breakdown of individual function costs. Initially, when the SLO is 0.9s, all the functions are

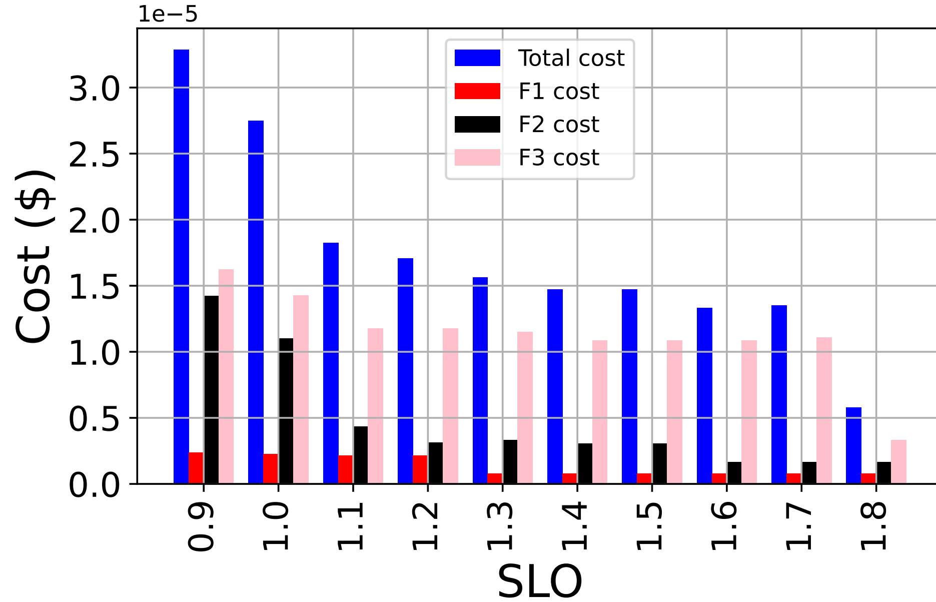


Figure 8·5: Cost ($) breakdown

placed at the edge, as it has less propagation delays, but 3x the cost. When the SLO is relaxed, the cost decreases rapidly because now *Config Finder* can either ask for less resources on the edge or move functions to the core cloud as it is more cost-effective despite higher propagation delays.

8.1 Other Techniques

Finding optimal running configurations for serverless applications is a well-studied topic [66, 68, 110]. We compare xCOSE and other configuration approaches in Table 8.1). As it can be seen that xCOSE has less sampling cost, can adapt to dynamic changes, and can configure resources and placement both for complex applications. Next, we will next discuss two such techniques Sizeless [66] and Costless [68] in detail.

8.1.1 Sizeless

Sizeless [66] is a recent ML-based approach to configure a serverless application. It trains a multi-target regression model with the profiling data of thousands of syn-

System	Config.	Placement	Dynamic	Cost
xCOSE	✓	✓	✓	*Low*
Costless [68]	✓	✓	×	*Low*
Sizeless [66]	✓	×	×	*High*
SLAM [109]	✓	×	✓	*High*
StepConf [122]	✓	×	✓	*Low*

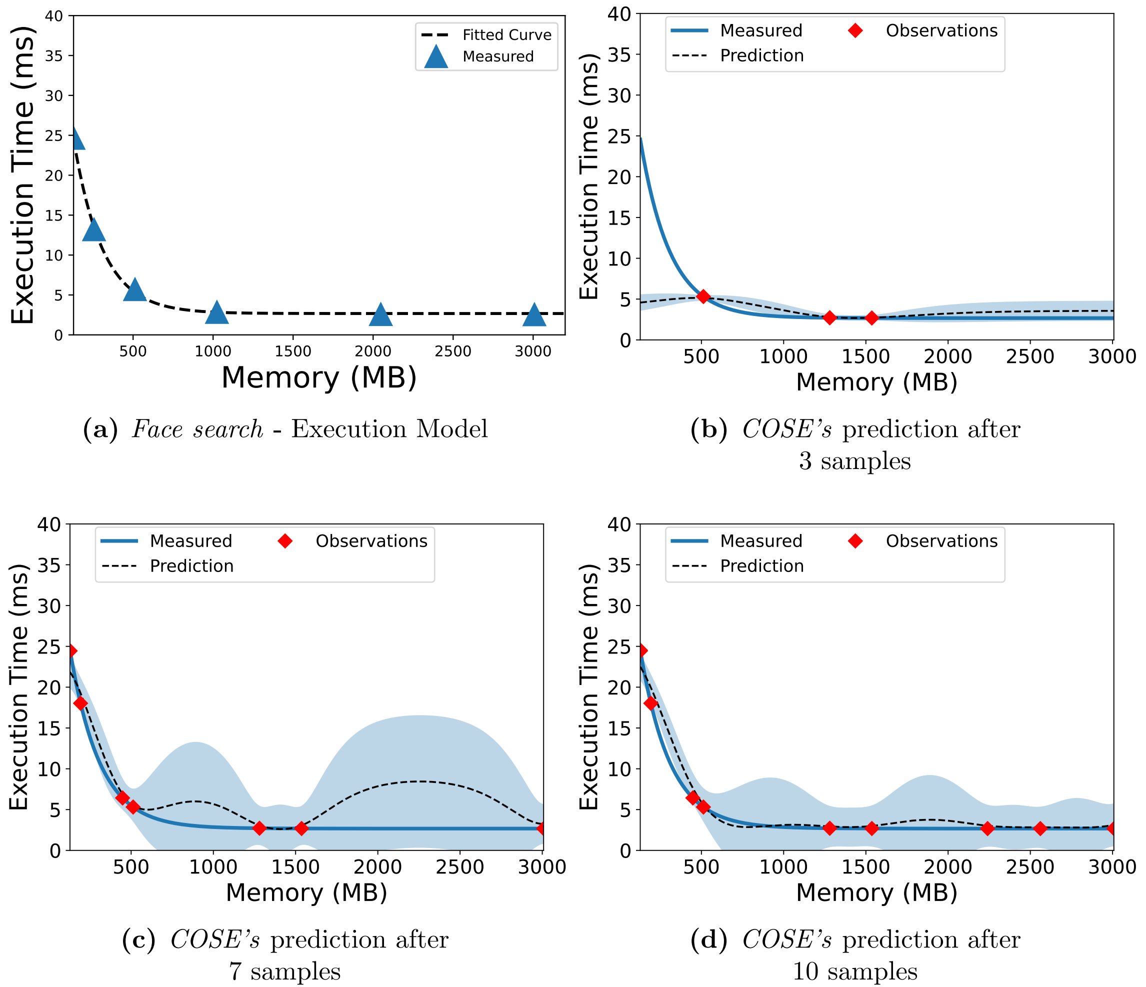
Table 8.1: xCOSE and other configuration techniques

thetic functions. Then using this model, Sizeless can predict the execution model of a serverless application on unseen memory configurations. The code and replication packages of Sizeless have been released, including the measurement data of different applications/functions running on AWS Lambda [43]. To compare xCOSE against Sizeless, we took the measurement data of these functions – Facial Recognition (*FaceSearch*), Airline Booking (*NotifyBooking*), *etc.* – and performed curve fitting on each function to obtain the actual execution model. Then we fed this execution model to xCOSE to see how long it takes for xCOSE to learn the model. In Figure 8·6, we report the performance of xCOSE on one of these functions (*FaceSearch*) – xCOSE performs similarly on other functions. In Figure 8·6a, we show the measurement data from the Sizeless replication package and the curve fit to that data. Figures 8·6b, 8·6c and 8·6d show xCOSE's execution model prediction after 3, 7 and 10 samples. We observe that after 10 samples, xCOSE was able to accurately predict the execution model. In our experiments, xCOSE took anywhere between 6 to 10 samples to predict the execution model with high accuracy. While we do not claim that xCOSE will always take less samples than Sizeless to converge to the execution model of a given function, we believe xCOSE offers certain logistical advantages as discussed next.

Logistical Advantages of xCOSE

xCOSE has certain logistical advantages over Sizeless: 1) xCOSE has the ability to configure applications consisting of multiple serverless functions (*service graphs*) and

meeting the SLO, while Sizeless targets individual function configuration, which may not always lead to the *most* optimal configurations for such applications; 2) Currently, the released Sizeless code only supports applications written in JavaScript, while xCOSE can support any function written in any language as long as the application reports the user-perceived latency; 3) xCOSE does not have any extra cost overhead other than sampling, while Sizeless needs to run thousands of synthetic functions on a serverless platform to generate the training dataset (for offline training); and 4) xCOSE can adapt to changes in the execution model resulting from underlying infrastructure and temporal changes, while Sizeless performs one-time configuration.



(a) *Face search* - Execution Model

(b) *COSE's* prediction after 3 samples

(c) *COSE's* prediction after 7 samples

(d) *COSE's* prediction after 10 samples

Figure 8·6: xCOSE predicting the execution model

8.1.2 Costless

Costless [68] is another technique, which given a serverless application composed of multiple functions, decomposes it across edge and core clouds and performs memory configuration to minimize cost while meeting performance requirements. Costless does not infer/build the execution model of the functions but relies on profiling the application under various memory configurations and then feeds this data to a constrained shortest path optimization solver, similar to the *Config Finder* functionality of COSE. It fundamentally differs from COSE which attempts to build the cost/performance model of the application first and then perform configuration and placement. We believe approaches like Costless can benefit from COSE's *Performance Modeler* functionality to build a more accurate and robust execution model and then feed it to its optimization solver.

Chapter 9

Conclusions

9.1 Summary

In this thesis, we studied the problem of choosing the *right* blend of cloud services to deploy an application while minimizing the cost of cloud usage and meeting the performance requirement. We presented Thrifty, a load-balancing and resource orchestration tool for cloud applications. Thrifty mainly consists of a load balancing approach LIBRA and a resource configuration component for FaaS deployments xCOSE.

LIBRA leverages the cost and performance analysis of IaaS and FaaS deployments and efficiently distributes load between the IaaS and FaaS deployments of an application to lower the cost and meet the performance requirements of an application. We evaluated LIBRA both in a simulation environment as well as on the AWS family of services. Our evaluations show that LIBRA can reduce SLO violations by up to 85% and cost by up to 53%, when compared to other approaches to deploying cloud applications. We also discuss possible enhancements to LIBRA and how further cost savings can be achieved using spot instances.

Our previous work COSE [48], uses a statistical learning technique called Bayesian Optimization and can only configure resources and placement for serverless applications consisting of linear chains of functions. In this thesis, we extend our previous work and present xCOSE, which in addition to linear chains, can also configure resources and placement for applications consisting of complex workflows such as service

graphs. Evaluation of xCOSE on AWS Lambda with real applications shows its efficacy in finding the near-optimal configuration and placement while meeting the SLOs of an application.

9.2 Future Research Directions

In this thesis, our load balancing approach LIBRA only leveraged two cloud services to run an application. In the future, in addition to the current services, we plan to include more cost-effective computing resources such as spot instances to deploy an application. To accommodate this, the *Scaling Manager* component of LIBRA, before adding VMs to the IaaS deployment, should perform an availability analysis of the available spot instances to make sure they meet the application's availability requirement. In addition, we believe that the design goals of LIBRA would inspire future cloud orchestration tools that can leverage multiple services not only within one cloud but across multiple clouds.

We believe xCOSE can be integrated into open-source FaaS platforms such as OpenWhisk to alleviate developers' responsibility to configure resources. If integrated, a developer would be able to only specify the performance requirement (SLO) for an application and the platform should find the best-suited configuration.

References

[1] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/. Access Date: Feb 23, 2023.

[2] Amazon Elastic Container. https://aws.amazon.com/pm/ecs/. Access Date: Feb 11, 2023.

[3] Amazon Rekognition Image. https://github.com/aws-samples/amazon-rekognition-image-for-box-skills. Access Date: Feb 23, 2021.

[4] Amazon Web Services. https://aws.amazon.com/. Access Date: Apr 19, 2022.

[5] AWS Auto Scaling. https://aws.amazon.com/autoscaling/. Access Date: Feb 11, 2023.

[6] AWS Lambda. https://docs.aws.amazon.com/lambda. Access Date: Feb 23, 2021.

[7] AWS Lambda at Edge. https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html. Access Date: Feb 23, 2021.

[8] AWS Lambda CPU-share. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html. Access Date: Feb 23, 2021.

[9] AWS Lambda Memory and CPU. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html. Access Date: May 24, 2021.

[10] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/. Access Date: Feb 14, 2023.

[11] AWS Load Balancer. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html. Access Date: Feb 23, 2023.

[12] AWS Sagemaker. https://aws.amazon.com/sagemaker/. Access Date: May 23, 2021.

[13] AWS Service Level Agreement. https://aws.amazon.com/compute/sla/. Access Date: Feb 23, 2023.

[14] AWS Spot Instances Interruptions. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/prepare-for-interruptions.html. Access Date: Feb 23, 2023.

[15] AWS Step Functions. https://aws.amazon.com/step-functions/. Access Date: Feb 13, 2023.

[16] Azure Containers. https://azure.microsoft.com/en-us/products/category/containers/. Access Date: Feb 11, 2023.

[17] Azure Function Pricing. https://azure.microsoft.com/en-us/pricing/details/functions/. Access Date: Feb 14, 2023.

[18] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Access Date: Feb 23, 2021.

[19] Azure MLaaS. https://azure.microsoft.com/en-us/products/machine-learning/. Access Date: May 23, 2021.

[20] Docker Resources. https://docs.docker.com/config/containers/resource_constraints/. Access Date: Feb 23, 2023.

[21] EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/. Access Date: Feb 14, 2023.

[22] Functions Lifetime. https://mikhail.io/serverless/coldstarts/. Access Date: Feb 27, 2021.

[23] Google + AWS . https://serverless.com/blog/shamrock-transacts-billions/. Access Date: Feb 23, 2021.

[24] Google AI Services. https://cloud.google.com/products/ai/. Access Date: May 23, 2021.

[25] Google Cloud Functions. https://cloud.google.com/functions. Access Date: Feb 23, 2021.

[26] Google Compute Engine. https://cloud.google.com/compute. Access Date: Feb 11, 2023.

[27] Google Function Pricing. https://cloud.google.com/functions/pricing. Access Date: Feb 14, 2023.

[28] Google Load Balancer. https://cloud.google.com/load-balancing.

[29] IBM Bare Metal Servers. https://www.ibm.com/cloud/bare-metal-servers. Access Date: Feb 11, 2023.

[30] IBM Cloud. https://www.ibm.com/cloud/pricing. Access Date: Feb 23, 2021.

[31] IBM Cloud Functions. https://www.ibm.com/cloud/functions. Access Date: Feb 23, 2021.

[32] IBM Function Pricing. https://cloud.ibm.com/functions/learn/pricing. Access Date: Feb 14, 2023.

[33] Kubernetes Knative Concurrency. https://knative.dev/docs/serving/autoscaling/concurrency/. Access Date: Feb 23, 2021.

[34] Kubernetes per Request. https://kubernetes.io/docs/concepts/\\configuration/manage-resources-containers/. Access Date: Feb 23, 2023.

[35] Lambda in Lambda. https://www.sqlshack.com/calling-an-aws-lambda-function-from-another-lambda-function/. Access Date: July 10, 2021.

[36] LIBRA Code. https://github.com/aliraza0337/LIBRA. Access Date: Feb 23, 2023.

[37] Microsoft Azure. https://azure.microsoft.com/en-us/services/virtual-machines/. Access Date: Feb 14, 2023.

[38] MIT Technology Review. https://www.technologyreview.com/2011/10/03/190237/the-cloud-imperative/. Access Date: Dec 23, 2022.

[39] Netflix and AWS. https://aws.amazon.com/solutions/case-studies/netflix/. Access Date: Feb 23, 2023.

[40] OpenWhisk Concurrency. https://github.com/apache/openwhisk/blob/master/docs/concurrency.md. Access Date: Feb 23, 2021.

[41] Serverless for Stream. https://aws.amazon.com/lambda/data-processing/. Access Date: Feb 23, 2021.

[42] Serverless Image Processing. https://www.image-processing.serverlessworkshops.io/. Access Date: Feb 4, 2022.

[43] Sizeless Replication Package. https://codeocean.com/capsule/6066333/tree/v2. Access Date: Feb 23, 2023.

[44] Spock Simulator. https://github.com/jashwantraj92/spock. Access Date: Feb 23, 2023.

[45] WITS Trace. https://wand.net.nz/wits/catalogue.php. Access Date: Feb 23, 2023.

[46] xCOSE Code. https://github.com/akhtarnabeel/COSE-Serverless-Configuration. Access Date: May 11, 2023.

[47] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein. Will serverless computing revolutionize NFV? *Proceedings of the IEEE*, 107(4):667–678, 2019.

[48] N. Akhtar, A. Raza, V. Ishakian, and I. Matta. COSE: configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 129–138, 2020.

[49] Nabeel Akhtar. *Orchestration and management of application functions over virtualized cloud infrastructures*. PhD thesis, Boston University, 2019.

[50] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[51] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[52] A.O. Allen. *Probability, Statistics, and Queueing Theory*. Computer Science and Scientific Computing. Elsevier Science, 1990.

[53] Leonardo Aniello, Silvia Bonomi, Federico Lombardi, Alessandro Zelli, and Roberto Baldoni. An architecture for automatic scaling of replicated services. In *Network Systems (NETYS)*, 8593. Springer, August 2014.

[54] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. ACM.

[55] C. Ayimba, P. Casari, and V. Mancuso. Adaptive Resource Provisioning based on Application State. In *International Conference on Computing, Networking and Communications (ICNC)*, pages 663–668, 2019.

[56] Arda Aytekin and Mikael Johansson. Harnessing the power of serverless runtimes for large-scale optimization. *arXiv preprint arXiv:1901.03161*, 2019.

[57] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[58] J. Carreira, P. Fonseca, Alexey Tumanov, A. Zhang, and R. Katz. Cirrus: a serverless framework for end-to-end ml workflows. *Proceedings of the ACM Symposium on Cloud Computing*, 2019.

[59] S. R. Chaudhry, A. Palade, A. Kazmi, and S. Clarke. Improved QoS at the edge using serverless computing to deploy virtual network functions. *IEEE Internet of Things Journal*, 7(10):10673–10683, 2020.

[60] B. Cheng, J. Fuerst, G. Solmaz, and T. Sanada. Fog function: Serverless fog computing for data intensive iot services. In *2019 IEEE International Conference on Services Computing (SCC)*, pages 28–35, 2019.

[61] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: a serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21. Association for Computing Machinery, 2021.

[62] Robert Cordingly, Wen Shu, and Wes J. Lloyd. Predicting performance and cost of serverless computing functions with SAAF. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 640–649, 2020.

[63] Christina Delimitrou and Christos Kozyrakis. QoS-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.*, 31(4):12:1–12:34, December 2013.

[64] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[65] Serhii Dorodko and Josef Spillner. Selective java code transformation into AWS lambda functions. In *Proceedings of the European Symposium on Serverless*

Computing and Applications, ESSCA@UCC 2018, Zurich, Switzerland, December 21, 2018, volume 2330 of *CEUR Workshop Proceedings*, pages 9–17. CEUR-WS.org, 2018.

[66] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 248–259, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110*, 2021.

[68] T. Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312, 2018.

[69] Yehia Elkhatib. Mapping Cross-Cloud systems: Challenges and opportunities. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[70] W. Fang, Z. Lu, J. Wu, and Z. Cao. RPPS: a novel resource prediction and provisioning scheme in cloud data center. In *2012 IEEE SCC, Honolulu, HI*, 2012.

[71] Lang Feng, Prabhakar Kudva, Dilma Silva, and Jiang Hu. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341, 07 2018.

[72] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[73] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. *arXiv preprint arXiv:1708.08028*, 2017.

[74] Gene F. Franklin, David J. Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice Hall PTR, USA, 4th edition, 2001.

[75] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach. BeFaaS: an application-centric benchmarking framework for faas platforms. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–8, 2021.

[76] Steven A. Greenlaw, Eric Dodge, Cynthia Gamez, Andres Jauregui, Diane Keenan, Dan MacDonald, Amyaz Moledina, Craig Richardson, David Shapiro, and Ralph Sonenshine. *Principles of Economics 2e*. Openstax, 2 edition, 2017.

[77] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208, July 2019.

[78] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[79] Ling-Hong Hung, Dimitar Kumanov, Xingzhi Niu, Wes Lloyd, and Ka Yee Yeung. Rapid RNA sequencing data analysis using serverless computing. *bioRxiv*, 2019.

[80] V. Ishakian, V. Muthusamy, and A. Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, 2018.

[81] Aman Jain, Ata F. Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. SplitServe: efficiently splitting apache spark jobs across faas and iaas. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 236–250, New York, NY, USA, 2020. Association for Computing Machinery.

[82] Anshul Jindal, Mohak Chadha, Shajulin Benedict, and Michael Gerndt. Estimating the capacities of function-as-a-service functions. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '21, New York, NY, USA, 2022. Association for Computing Machinery.

[83] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica,

and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Number UCB/EECS-2019-3, Feb 2019.

[84] Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. Serverless computing provides on-demand high performance computing for biomedical research. *arXiv preprint arXiv:1807.11659*, 2018.

[85] Benjamin D Lee, Michael A Timony, and Pablo Ruiz. DNAvisualization.org: a serverless web tool for DNA sequence visualization. *Nucleic Acids Research*, 47(W1):W20–W25, 06 2019.

[86] H. Lee, K. Satyam, and G. Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450, Los Alamitos, CA, USA, jul 2018. IEEE Computer Society.

[87] Liduo Lin, Li Pan, and Shijun Liu. Backup or not: An online cost optimal algorithm for data analysis jobs using spot instances. *IEEE Access*, 8:144945–144956, 2020.

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

[89] W. Lloyd, Minh Vu, Baojia Zhang, O. David, and G. Leavesley. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 195–200, 2018.

[90] A. Luckow and S. Jha. Performance characterization and modeling of serverless and hpc streaming applications. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 5688–5696, 2019.

[91] A. H. Mahmud, Y. He, and S. Ren. BATS: budget-constrained autoscaling for cloud performance optimization. In *2015 IEEE MASCOTS, Atlanta, GA*, 2015.

[92] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. FaaSdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.

[93] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems*, 11 2017.

[94] J. Manner, M. Endreß, T. Heckel, and G. Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, Dec 2018.

[95] Johannes Manner, Martin Endreβ, Sebastian Bohm, and Guido Wirtz. Optimizing cloud function configuration via local simulations. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 168–178, 2021.

[96] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, 2017.

[97] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[98] Ali Yadavar Nikravesh, Samuel A. Ajila, and Chung-Horng Lung. Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 35–45, 2015.

[99] J. H. Novak, S. K. Kasera, and R. Stutsman. Cloud functions for fast and robust resource auto-scaling. In *2019 11th International Conference on Communication Systems Networks (COMSNETS)*, pages 133–140, Jan 2019.

[100] Pangkaj Paul, John Loane, Fergal Mccaffery, and Gilbert Regan. *A Serverless Architecture for Wireless Body Area Network Applications*, pages 239–254. 10 2019.

[101] Per Persson and Ola Angelsmark. Kappa: Serverless iot deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing*, WoSC '17, page 16–21, New York, NY, USA, 2017. Association for Computing Machinery.

[102] Chetan Phalak, Dheeraj Chahal, and Rekha Singhal. SIRM: cost efficient and slo aware ml prediction on fog-cloud network. In *2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 825–829, 2023.

[103] Thanh-Phuong Pham, Sasko Ristov, and Thomas Fahringer. Performance and behavior characterization of amazon ec2 spot instances. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 73–81, 2018.

[104] Ali Raza, Nabeel Akhtar, Vatche Isahagian, Ibrahim Matta, and Lei Huang. Configuration and placement of serverless applications using statistical learning. *IEEE Transactions on Network and Service Management*, pages 1–1, 2023.

[105] Ali Raza, Ibrahim Matta, Nabeel Akhtar, Vasiliki Kalavri, and Vatche Isahagian. Function-as-a-Service: From An Application Developer's Perspective. *Journal of Systems Research - JSys*, 1(1), 2021/09/19,01:00.

[106] Ali Raza, Zongshun Zhang, Nabeel Akhtar, Vatche Isahagian, and Ibrahim Matta. LIBRA: An Economical Hybrid Approach for Cloud Applications with Strict SLAs. In *IEEE International Conference on Cloud Engineering (IC2E)*, 2021.

[107] Sasko Ristov, Stefan Pedratscher, Jakob Wallnoefer, and Thomas Fahringer. DAF: dependency-aware faasifier for node.js monolithic applications. *IEEE Software*, 38(1):48–53, 2021.

[108] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507, 2011.

[109] Gor Safaryan, Anshul Jindal, Mohak Chadha, and Michael Gerndt. SLAM: SLO-Aware memory optimization for serverless applications. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 30–39, 2022.

[110] Lucia Schuler, Somaya Jamil, and Niklas Kühl. Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 804–811, 2021.

[111] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges and applications. *arXiv preprint arXiv:1911.01296*, 2019.

[112] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[113] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[114] Nikhila Somu, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Panopticon: A comprehensive benchmarking tool for serverless applications. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 144–151, 2020.

[115] Josef Spillner. Transformation of Python Applications into Function-as-a-Service Deployments. *arXiv preprint arXiv:1705.08169*, 2017.

[116] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[117] Zhucheng Tu, Mengping Li, and Jimmy Lin. Pay-per-request deployment of neural network models using serverless architectures. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 6–10, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[118] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[119] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019.

[120] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[121] Jinfeng Wen, Yi Liu, Zhenpeng Chen, Yun Ma, Haoyu Wang, and Xuanzhe Liu. Understanding characteristics of commodity serverless computing platforms, 2021.

[122] Zhaojie Wen, Yishuo Wang, and Fangming Liu. StepConf: SLO-Aware dynamic resource configuration for serverless function workflows. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1868–1877, 2022.

[123] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai. Serverless big data processing using matrix multiplication as example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 358–365, 2018.

[124] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[125] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, MOTA '16, New York, NY, USA, 2016. Association for Computing Machinery.

[126] Shunyu (David) Yao, Muhammad Ali Gulzar, Liqing Zhang, and Ali R. Butt. Towards a serverless bioinformatics cyberinfrastructure pipeline. In *1st Workshop on High Performance Serverless Computing (HiPS '21)*, Virtual Event, Sweden, 2021.

[127] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[128] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[129] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '19, pages 61–66, New York, NY, USA, 2019. ACM.

[130] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 1–14, New York, NY, USA, 2022. Association for Computing Machinery.

Curriculum Vitae