Imperial College of Science, Technology and Medicine
Department of Computing

Model-based Resource Management for Fine-grained Services

Alim Ul Gias

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy of Imperial College London
February 2022

Statement of Originality

I declare that this thesis is written by myself and the work presented here is my own, unless stated otherwise.

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

IEEE Notice

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Imperial College London's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Abstract

The emergence of DevOps has changed the way modern distributed software systems are developed. Architectures decomposed in fine-grained services, such as microservices or function-as-a-service (FaaS), are now widespread across many organizations. From a resource management perspective, although the systems built with such architectures have many benefits, there are still research challenges that need further attention. In this study, we have focused on three such challenges, each concerning a specific system resource: compute, memory, or storage. Firstly, we focus on scaling the capacity of microservices at runtime. Here, the challenge is to design an autoscaler that can decide between vertical and horizontal scaling options to distribute the CPU capacity. Secondly, we focus on estimating the required capacity of an on-premises FaaS platform such that the service level agreements (SLAs) for function response times are satisfied. The challenge here is to address the cold start dilemma, i.e., that a cold start delays a function response but reduces the memory consumption. Thus, we must find a limit of cold starts such that the memory-consumption remains in-check while satisfying the SLAs. Finally, we focus on the storage management for distributed tracing targeted at microservices. The volume of such traces generated in a data center can be in the scale of tens of terabytes per day, but only a small fraction of these traces is useful for troubleshooting. The objective then is to sample only the useful traces. The key to addressing all these challenges is first, modeling the dynamics concerning the resources and subsequently, leveraging the model in a resource controller. To address the first challenge, we have developed an autoscaler ATOM that leverages layered queueing network (LQN) models to take its scaling decisions. Our experiment, with a real-life application, shows that ATOM produces 30-37% better results than the baseline autoscalers. For the second challenge, we have developed COCOA, a cold start aware capacity planner. COCOA utilizes M/M/k setup and LQN models to assess the cold start scenario and estimate the required capacity. We show with simulation that COCOA can reduce over-provisioning by over 70% compared to the availability aware approaches. Finally, addressing the third challenge, we propose SampleHST, a trace sampler that works under a storage budget constraint. SampleHST relies on either bag of words or graph-based models to represent a trace and groups similar traces using online clustering to perform sampling. We have evaluated the performance of SampleHST using data from both literature and production, which shows it produces 1.2× to 19× better results than the state-of-the-art.

Acknowledgements

First, I express my gratitude to my supervisor Giuliano Casale for his continuous support throughout the course of this PhD program. When I started the PhD, many of the subject-matters were new to me. Giuliano was always patient in clearing up all my confusions. His guidance has made my journey, over the past four years, as smooth as possible.

I am grateful to the Commonwealth Scholarship Commission for funding my PhD program. If not for them, I might not have come to the UK for a PhD. They have also arranged many social events for us, which helped me a lot to improve my soft skills.

I was lucky to co-author a paper with Professor Murray Woodside. I cannot thank him enough for the feedback he provided while writing the paper. Queueing network is an integral part of this thesis and I learnt many critical aspects of it from him. I also want to thank Professor Giuseppe Serazzi, who was the PhD supervisor of Giuliano, for his time to read my papers and suggesting how to improve them. His feedback has greatly shaped my ideas for this thesis.

Over the course of this PhD program, I was an active member of the SPEC RG DevOps Performance Working Group. I am particularly grateful to André van Hoorn, the chair of the group, who has always given me excellent suggestions to improve my work. I was also lucky to present part of my research in a summer school session led by him.

I want to express my gratitude to my group mates Ahmad and Lulai for always helping me out when I was struggling with a difficult problem. Ahmad helped me to set up the initial testbed for my research. Lulai gave me suggestions on how to formulate some of the research problems for this work. I am also grateful to José, who only joined our Lab in my final-year, yet managed to provide valuable feedback on my latest work. I also thank Gabriele, who spent the summer of 2019 in our Lab, for his suggestions on workload design for the experiments.

I am grateful to all the members of our research group, QORE Lab, for always providing constructive feedback to my work. We also have some wonderful memories together from all our group lunches, dinners and parties. They will always be close to my heart.

And last but not least, I am grateful to my family for always being supportive of me. They all played their part, in the best way possible, so that I can complete my work. Thanks to them I can finally submit this thesis.

Dedication

To my wife Rubaida, who is always there for me, in good “times” (results) and in bad.

Contents

Abstract **iii**

Acknowledgements **v**

Dedication **vii**

1 Introduction **1**

1.1 Motivation and Objectives 1

1.2 Contributions 6

1.3 Publications 8

1.4 Thesis Organization 10

2 Background **11**

2.1 Introduction 11

2.2 Microservices and Containers 12

2.2.1 Overview 12

2.2.2 Related Work 16

2.3 Serverless and Function-as-a-Service 19

2.3.1 Overview . . . 19
2.3.2 Related Work . . . 21
2.4 Distributed Tracing . . . 23
2.4.1 Overview . . . 23
2.4.2 Related Work . . . 26
2.5 Layered Queueing Network . . . 27
2.6 Summary . . . 32

3 Capacity Scaling at Runtime 33

3.1 Introduction . . . 33
3.2 Motivating Example . . . 35
3.2.1 Running Case . . . 36
3.2.2 Comparing scaling rules . . . 37
3.3 Performance Model . . . 38
3.3.1 LQN Modeling . . . 39
3.3.2 Service Demand Estimation . . . 41
3.3.3 Validation . . . 43
3.4 ATOM: Autoscaling Microservices . . . 44
3.4.1 Context and Architecture . . . 45
3.4.2 Problem Statement . . . 47
3.4.3 Optimization Method . . . 50
3.5 Evaluation . . . 53

3.5.1 Experimental Setup 53

3.5.2 Results 55

3.6 Summary 63

4 Cold Start Aware Capacity Planning 64

4.1 Introduction 64

4.2 Similarity between FaaS and TTL caches 66

4.2.1 Analogy 67

4.2.2 Example 68

4.2.3 Observations 69

4.3 Modeling Cold Starts in FaaS 70

4.3.1 Estimating Cold Start Probabilities 70

4.3.2 Predicting Response Time 73

4.3.3 Model Validation 77

4.4 Capacity Planning 79

4.4.1 Overview 79

4.4.2 Problem Statement 81

4.4.3 Optimal Strategy Generation 82

4.5 Evaluation 84

4.5.1 Experimental Setup 85

4.5.2 Results 85

4.6 Summary 89

5 Storage Management for Distributed Tracing 90

5.1 Introduction . . . 90

5.2 Issues with Tail-based Sampling . . . 93

5.2.1 Trace Anomaly Definition . . . 93

5.2.2 Evaluation Criteria . . . 94

5.2.3 Demonstration . . . 95

5.3 Anomaly Detection from Trace Stream . . . 97

5.3.1 Bag of Words Model for Point Anomalies . . . 97

5.3.2 Evaluation of Anomaly Detection Methods . . . 98

5.3.3 Modifications to Half Space Trees . . . 99

5.4 Sampling with Storage Budget Constraints . . . 106

5.4.1 Cluster based Sampling . . . 106

5.4.2 SampleHST Clustering . . . 109

5.4.3 SampleHST Controller . . . 117

5.5 Evaluation considering Point Anomalies . . . 125

5.5.1 Experimental Setup . . . 125

5.5.2 Results . . . 127

5.6 Extension for Collective Anomalies . . . 132

5.6.1 Graph Model for Collective Anomalies . . . 132

5.6.2 Anomaly Detection . . . 135

5.6.3 Sampling Performance . . . 136

5.7 Summary 139

6 Conclusion **140**

6.1 Summary of Thesis Achievements 140

6.2 Future Work 142

Bibliography **145**

A FaasSim: Function-as-a-Service Simulator **165**

List of Tables

3.1 Two cases where the front-end microservice is the bottleneck 37

3.2 Workload patterns for model validation 43

3.3 Percent error between model and system measurement 44

3.4 Feature and microservice specific TPS and utilization from model and measurement for workload 1 with 3000 users 44

3.5 Relevant notation for the optimization problem 50

3.6 Auxiliary functions for Algorithm 3.1 51

3.7 Docker swarm server config. for performance evaluation 54

3.8 Workloads considered in evaluating the autoscalers 55

4.1 Simulation parameters for model validation 78

4.2 Percent error in estimating the response time of each function–Across all the Zipf parameters in Table 4.1 79

4.3 Notations used in Algorithm 4.1 83

4.4 Auxiliary functions for Algorithm 4.1 84

4.5 Comparing the predicted memory capacity of different approaches - averaged across the Zipf parameters from Table 4.1 86

5.1 Results of different anomaly detection methods on the production dataset 99

5.2 Approximate counting results on the production data set 106

5.3 Auxiliary functions used in Algorithm 5.1 117

5.4 Auxiliary functions used in Algorithm 5.2 125

5.5 Performance of SampleHST on the Production data with different clustering bandwidths 129

5.6 Sampling results for different budget on the Production data with exact and approximate counting 129

5.7 Performance of SampleHST on the Production data considering hyper-rectangles with 5% storage budget 131

5.8 Comparing the performance of different methods on the Production data 131

5.9 Comparing the anomaly detection results of three methods on the DSB dataset . 136

5.10 Performance of SampleHST on the Compose Post API of the DSB dataset with different clustering bandwidth 137

5.11 Sampling results for different budget on the Compose Post API with exact and approximate counting 137

5.12 Sampling performance on the Compose Post API considering hyper-rectangles with 5% storage budget 138

5.13 Comparing the performance of different methods on the Compose Post API of the DSB dataset. 138

5.14 Comparing the performance of different methods on the User Timeline API of the DSB dataset. 139

List of Figures

1.1 A bird's eye view of the thesis contributions from a resource management perspective 8

2.1 Mapping of the background concepts with the contributing chapters 12

2.2 Comparing the architecture of VMs and Containers 14

2.3 Illustration of the serverless execution model 19

2.4 An illustration of the distributed tracing concepts 24

2.5 Illustrating the distributed trace aggregation mechanism based on timeouts . . . 25

2.6 Different components of a Layered Queueing Network (LQN), illustrated with a simple model 29

3.1 The architecture of the sock shop application 36

3.2 Effect of vertical and horizontal scaling over the front-end microservice in two different cases 38

3.3 Different user interaction scenarios considered in the LQN model 39

3.4 An LQN model of the Sock Shop application 40

3.5 Service demand estimation using two different techniques 42

3.6 Utilization of different servers according to model estimation and system measurement for workload pattern 1 and 3 45

3.7 Overview of ATOM's components and their interactions 46

3.8 Performance comparison of ATOM with its conservative versions: ATOM-S and ATOM-T 56

3.9 The effect over time on the TPS by ATOM and baseline autoscalers under different workload 57

3.10 Comparison of the autoscalers for different elasticity and performance metrics with the increase of concurrent users 58

3.11 Comparison of the autoscalers for different elasticity and performance metrics with change in request mix 58

3.12 The over time CPU capacity allocated by ATOM and baseline autoscalers under different workload 59

3.13 The demand vs. supply in CPU capacity for 3 microservices, involved in bottleneck shifting, after each scaling action by UV and ATOM 61

3.14 Performance comparison of UV and ATOM with change in the monitoring window size 61

3.15 TPS yielded by UV and ATOM with $N = 3000$ 62

3.16 TPS yielded by the scalers with high burstiness in workload 62

4.1 Evaluation of an availability-aware approach in estimating the capacity of a FaaS platform and ensuring the SLA for response time 69

4.2 The M/M/1/setup/delayedoff model for a function 71

4.3 An LQN model for a FaaS platform 75

4.4 The mean, 95^{th} percentile and the maximum modeling error in estimating the response time of each function 78

4.5 An overview of the COCOA approach 80

4.6 Comparing the memory capacity predicted by COCOA and the availability-ware approaches for $\lambda = 0.5$ and $\eta = 1.0$ 86

4.7 Comparing the runtime memory consumption obtained from the analytical approximation and simulation for $\lambda = 0.5$ and $\eta = 1.0$ 87

4.8 Illustrating that COCOA can meet the maximum memory demand for different percentage of memory consumption while the functions are idle 87

4.9 Response time and hit rate distribution of 128 functions with a target SLA of 2 seconds for $\lambda = 0.5$ and $\eta = 1.0$ 88

4.10 Response time and hit rate distribution of 128 functions with a target SLA of 1.5 seconds for $\lambda = 0.8$ and $\eta = 1.0$ 88

5.1 Comparing the scaled value of HTTP 200 code counts with and without applying the transformation function f 101

5.2 Approximate counting methods: counting from 0 to 2^{15} 104

5.3 Illustration of trace sampling based on trace-cluster association 107

5.4 The production trace plotted using the mass-based properties 108

5.5 An overview of the SampleHST architecture 109

5.6 Illustrating the kernel region of the cluster 114

5.7 Demonstrating the cluster removal process 114

5.8 Demonstrating cluster merging process 115

5.9 Illustrating the distance based cluster ranking concept 120

5.10 Illustrating the rare and populous region of the clustering plane for trace sampling 122

5.11 Illustrating the output of the SampleHST clustering algorithm 128

5.12 Comparing the mass scores, with exact and approximate counting, for the production data 130

5.13 Comparing the clusters generated by SampleHST with equal and unequal clustering bandwidth 130

5.14 Time complexity of FGSD algorithm with respect to graph sizes 133

5.15 Examples of directed graphs of traces and the adjacency matrix of their undirected version 134

5.16 Examples of non-isomorphic graphs of traces of length four 135

A.1 The architecture of the FaaS platform considered by FaasSim 166

Chapter 1

Introduction

1.1 Motivation and Objectives

Fine-grained service architectures, such as the microservices architecture [Thö15] and function-as-a-service (FaaS) [VETT+18], are now at the core of DevOps [EGHS16] - a widely adopted development and operational approach in the software industry. This is because, in conjunction with container based technologies [Ber14, And15], microservices and FaaS ease the process of frequent software releases, which is a major aspect of DevOps. The microservices architecture is a software integration pattern evolving from the Service Oriented Architecture (SOA) [PZA+17]. A microservices-based system includes a collection of fine-grained services, hence each of the services is referred as a microservice. A key benefit of microservices is an individual microservice can be deployed and managed independently, which opens the possibility of individual resource management. FaaS is another realization of a fine-grained service architecture, which is built on the serverless execution model [VETT+18]. Typically, a function in a FaaS platform constitutes a business logic that is triggered in response to an event. Although these functions can be loosely defined as microservices, we focus throughout on a strict definition where they are regarded as being different, for example due to the common requirement for functions of being stateless. Moreover, we emphasize individual functions and their underpinning platform as opposed to other use cases such as FaaS workflows.

The combination of a fine-grained system architecture and container-based deployment platform has several benefits:

- Each of the services is lightweight in nature allowing a fine-grained resource allocation. For an instance, instead of allocating a single CPU core per service, half a CPU core can be allocated. Controlling capacity at this granularity reduces resource wastage compared to traditional virtualization mechanisms.

- Since the services are light-weight, they generally have a quick start-up time. This opens-up the possibility of only loading a service when there is a continuous demand rather than always keeping it loaded in the memory.

- Since each of the services are deployed and managed independently, they are not necessarily placed in a single server. Furthermore, as all these services can be scaled individually, they can have multiple replicas spanning across multiple servers, making a highly decentralized architecture.

As these properties are useful in practice, there has been a significant growth of use cases for applying containers, microservices and serverless FaaS, however mechanisms to handle the underpinning resources are still a subject of research investigation. In particular, different resource management challenges exist that can benefit from scientific investigation. We group some key challenges based on the key system resource they emphasize:

- **Compute:** Allocating a fraction of CPU capacity may increase resource utilization but it poses a dilemma when there is an increase in the current workload of the system. In such a case, the CPU capacity needs to be scaled automatically. Typically there are two possible choices - increase the CPU capacity of the container. which is referred as vertical scaling, or add another replica of the service, which is referred as horizontal scaling. This problem is more relevant when the services are inter-dependent, i.e., for the microservices architecture, as the service communication pattern also have an effect over the scaling decision. This is similar to the well studied problem in queueing theory - one

fast CPU versus several slow CPUs [HB13]. This problem is applicable for microservices as small fractions of CPU capacity can be allocated and the capacity can be changed both horizontally and vertically.

- **Memory:** Removing a function from the memory reduces the memory consumption but this adds a latency delay for the upcoming service request. This is known as the cold start problem [WLZ+18]. Although the functions are usually light-weight and have a low start-up overhead, there can be cases when these cold start delays affect the function response time significantly. This poses a dilemma regarding installation of an on-premises FaaS platform. If a function is kept in the memory indefinitely, thus reducing cold starts, it increases the required memory capacity. On the other hand, allowing cold starts will affect the service-level-agreements (SLAs) for response time. To solve this problem, the dynamics among the SLAs, cold starts and memory consumption need to be assessed. Striking a balance among these factors will ultimately reduce the required memory capacity while satisfying the SLAs.

- **Storage:** Deploying and scaling the microservices independently, spanning across multiple servers, creates a problem from the perspective of maintenance. In particular, keeping the logs of a request workflow becomes difficult. Distributed tracing is an approach developed primarily to aid in this scenario. It profiles the user requests spanning across multiple microservices. For a data center hosting microservices in the scale of tens of thousands, the volume of traces generated could easily be in the scale of terabytes per day. To store this traces for future troubleshooting requires a large storage capacity. However, only a small fraction of these traces are helpful for such purposes. This opens up the possibility of reducing storage requirement by saving only the interesting traces in terms of, for example, identifying QoS violations. The challenge is to identify those interesting traces and sample according to a storage budget.

All the challenges discussed above may be narrowed down, in essence, to two main tasks. The first task involves assessing the dynamics of different factors that affect the resources. For example, considering the scaling and cold start scenario, the initial challenge is to design a

mechanism that can estimate different performance metrics, e.g., throughput and utilization, under various system configurations. For this, we can leverage a queueing network [BGdMT06] based model, which is often used to predict such metrics. For the distributed tracing scenario, the initial challenge is to find a suitable trace representation and identify the traces that are interesting or anomalous. A possible way for trace representation could be considering them as a document and use a representation for document classification [Seb02]. If the trace contains a cause-effect like pattern, we can also use a graph-based representation. The second task involves designing a resource manager that will utilize the system insights, generated using the models, to take necessary management actions. For the scaling scenario, the controller will decide the scaling actions based on the estimated performance metrics. Similarly, the controller for the cold start scenario will suggest the required capacity to setup a FaaS platform. The controller for distributed tracing will limit the number of traces to be sampled, based on the information about which traces are anomalous.

Based on the above discussions, we can define three primary research objectives for this thesis, which are are as follows.

- **RO1. Auto-Scaling the Compute Capacity:** The first objective is to develop a method that will scale the CPU capacity of microservices at runtime. To be more specific, the scaler should be able to assess different scaling options and suggest the optimal one. This can be further divided into two sub-goals.
  - **RO1.1. Modeling Microservices:** The first goal will be to model a microservices application using Layered Queueing Networks (LQNs) [WNPM95, FAOW+08], which is already shown to be effective to model modern distributed systems [TMW10, SD11]. The model should incorporate the system dynamics of the application e.g., workload, shared resources, multiple application layers and provide different performance metrics, such as throughput and utilization, as the output.
  - **RO1.2. Optimal Scaling:** The second goal is to provide optimal scaling decisions for microservices. In particular, there should be a controller that, using the LQN

model, can assess different combinations of vertical and horizontal scaling configuration. Based on this assessment, it should select an optimal configuration for the current workload and eventually apply that on the system.

- **RO2. Capacity Planning considering Cold Starts:** The second goal is develop a capacity planning method that considers the trade-off between latency delay and conservation of primary memory due to cold starts. The method should identify a limit of cold starts such that the SLAs are satisfied, while reasonably limiting the memory consumption. This could be divided into the following steps.
  - **RO2.1. Modeling Cold Starts:** The first challenge is to model the cold start scenario in a Function-as-a-Service (FaaS) platform. To be particular, the model should be able to estimate the response time considering a varying degree of function cold start delays and service times.
  - **RO2.2. Capacity Planning:** Once we have a model of the cold start scenario, it needs to incorporated in a capacity planner. For a FaaS-based system to be deployed, with given function cold start delays and service times, the planner should provide a deployment configuration that includes the required CPU and memory capacity that satisfy the SLAs for response time.
- **RO3. Trace Sampling with Storage Constraint:** The final objective is focused on saving distributed traces with a storage budget. Here, we aim to develop a trace sampler, with a limitation on the percentage of traces it can sample, that primarily focuses to save the interesting or anomalous traces. This involves two major challenges.
  - **RO3.1. Anomaly Detection from Trace Stream:** The first challenge is to differentiate the interesting or anomalous traces from the normal ones. This will initially require a trace model and after that a classifier that will detect the anomalies in an unsupervised manner.
  - **RO3.2. Trace Sampler:** The second challenge is to transform this anomaly detection process to a trace sampler. This requires to characterize the traces such the the

anomalous and normal traces form distinct groups and depending on the budget, the traces will be sampled.

1.2 Contributions

The contributions of this thesis is summarized as follows.

- To address our first research objective, **RO1**, we have developed ATOM, a model-driven autoscaler tailored for microservices. We first demonstrate that if a service can be both vertically and horizontally scaled, the workload characteristics need to be taken into account to decide which one of the two scaling actions is preferable. ATOM incorporates this by leveraging layered queueing network models. It instantiates and solves, at run-time, such models of the application for different scaling configurations. Computational optimization is used to dynamically control the number of replicas for each microservice and its associated container CPU share, overall achieving a fine-grained control of the application capacity at run-time. Experimental results indicate that for heavy workloads ATOM offers around 30-37% higher throughput than baseline model-agnostic controllers based on simple static rules. We also find that model-driven reasoning reduces the number of actions needed to scale the system as it reduces the number of bottleneck shifts that we observe with model-agnostic controllers.

- To address our second research objective, **RO2**, we propose a novel approach, COCOA, to size an on-premises FaaS platform. We have initially investigated the similarity of this problem with the hit rate improvement problem in TTL caches and concluded that solutions for TTL cache, although potentially applicable, lead to over-provisioning in FaaS. COCOA addresses this over-provisioning problem. It uses a queueing-based approach to assess the effect of cold starts on the function response times and sets the function idle periods to values such that the cold starts remain tolerable from the perspective of service level agreements (SLAs). Furthermore, it considers different memory consumption values depending on whether the function is idle or in execution. Based on this assessment,

COCOA estimates the required CPU and memory capacity to serve the expected workload. Using an event-driven FaaS simulator, *FaasSim*, we have developed, we show that COCOA can reduce over-provisioning by over 70% in some workloads, while satisfying the SLAs for response time.

- To address our final research objective, **RO3**, we propose SampleHST, a novel trace sampler for distributed tracing with a storage budget constraint. SampleHST is tailored to work on a stream of trace data and taking the sampling decision in an unsupervised manner. Depending on the definition of interesting traces, SampleHST uses either a Bag of Words (BoW) or a Graph model to represent a trace. This representation is then passed to a forest of Half Space Trees (HSTs) [TTL11, TZLT13] for characterizing the trace with a mass score, which indicates the observed frequency of traces of similar types. These mass scores are used to cluster the traces using an online method based on a variant of the mean shift algorithm. Finally, the association of each trace with a cluster, where certain clusters are prioritized over the others, is used to take the sampling decision. We have compared the performance of SampleHST with recently suggested methods using data reported in literature and produced in a data center. Depending on the properties of the experiment, we set the evaluation criteria using either precision, recall or F1-Score. The experiments demonstrate that SampleHST produces 1.2× to 19× better results than the recent methods.

Although these contributions address a wide range of research challenges, in principle, they can be considered as a sequence of resource management tasks, such as monitor, analyze, plan and execute, which work on a shared knowledge-base. Most of these tasks are required for all the contributions *i.e.*, ATOM, COCOA and SampleHST. For example, all of them need to analyze a wide range of data by leveraging artefacts like a queueing model or the trace structure. In the case of ATOM and COCOA, the analysis involves assessing the scaling scenarios and the effect of cold starts. In the case of SampleHST, the analysis phase focuses on trace characterization such that the traces indicate anomalous system behaviors. We present the thesis contributions from this resource management perspective in Fig. 1.1. Note that, all

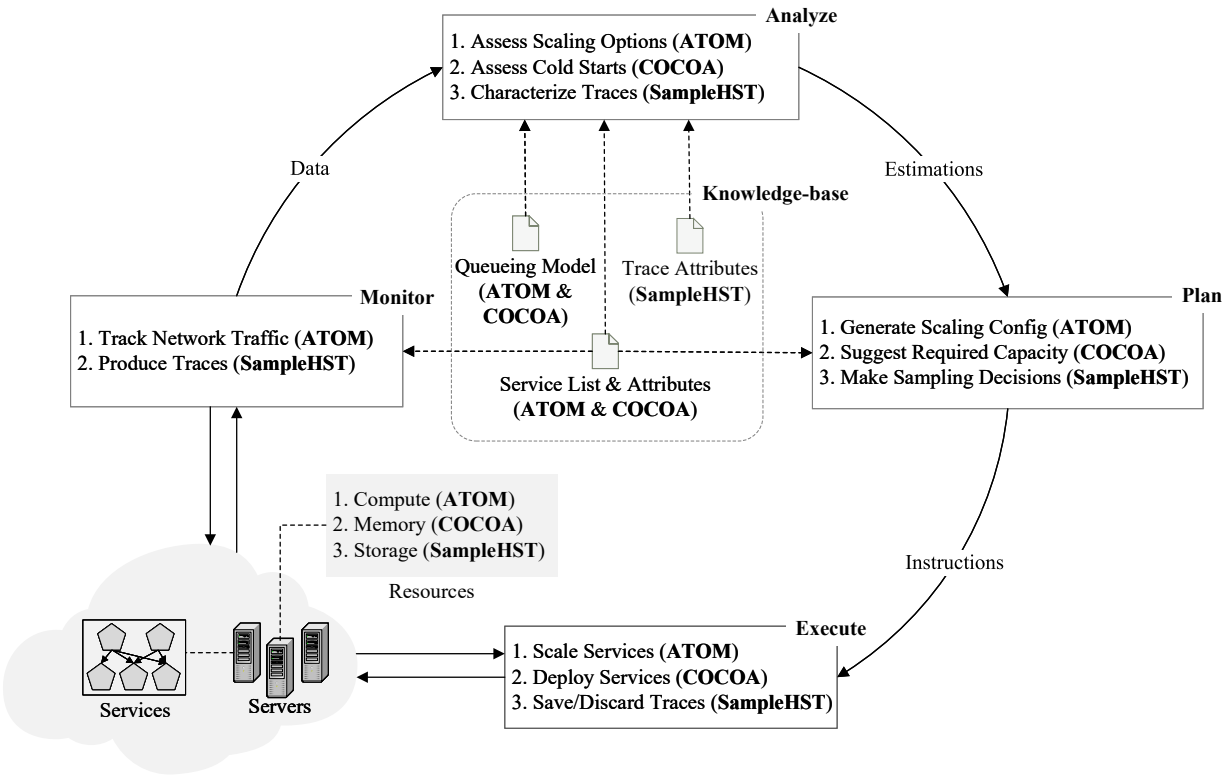


Figure 1.1: A bird's eye view of the thesis contributions from a resource management perspective. The contributions *i.e.*, ATOM, COCOA and SampleHST are presented as a sequence of different management tasks such as Monitor, Analyze, Plan and Execute. The focus of each contribution is on one of the following resources: Compute, Memory and Storage. The tasks leverage different artefacts such as queueing models, service attributes, trace attributes, etc. to achieve their objective.

the tasks are not associated with all the contributions. Here, COCOA is not associated with the monitor task as it focuses on capacity planning, rather than runtime management, which does not require an active monitoring component.

1.3 Publications

Over the course of writing this thesis, we have published some of our research contributions. We present them here.

[GCW19] A. U. Gias, G. Casale and M. Woodside **ATOM: Model-Driven Autoscaling for Microservices**. In Proceedings of 39th International Conference on Distributed Computing Systems (ICDCS), 2019. ©IEEE.

In this paper, leveraging layered queueing networks (LQNs), we have developed an autoscaler for microservices, namely ATOM. I contributed to majority of the paper including modeling a demo microservices application with the LQN, validation of the LQN model against multiple workloads, designing the autoscaler that utilizes the model to take scaling decisions and experimental validation considering baseline rule-based autoscalers. We discuss these contributions in Chapter 3.

[GC20] A. U. Gias and G. Casale. **COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms**. In Proceedings of 28th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, (MASCOTS), 2020. ©IEEE.

In this paper, we present COCOA, a capacity planning method for on-premises FaaS platforms. Like before, I have contributed to majority of the paper. My contributions include modeling the cold start scenario using M/M/k setup and LQN models, validating the cold start model using simulation, incorporating such models in a capacity planner to reduce resource overprovisioning and experimental validation considering availability-aware approaches. We discuss these contributions in Chapter 4.

[AGW+21] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, and A. Filieri. **Quality-Aware DevOps Research: Where Do We Stand?**. IEEE Access 9, 2021. ©IEEE.

This paper is a collaborative work of the QORE Lab[1]. In this paper, we discussed the state-of-the-art of the recent DevOps researches focusing on the Quality of Service (QoS) aspect. We categorized the research themes, based on different stages of DevOps and also presented the open challenges. I contributed to the runtime service management section of the paper that discusses research challenges regarding monitoring microservices, container placement strategies and autoscaling microservices. Part of my contribution is incorporated in Chapter 2.

In addition, we are on the process of preparing a paper submission that includes the contributions from Chapter 5.

[1]QORE – Quality of Service Research Lab: `https://qore.doc.ic.ac.uk/`

1.4 Thesis Organization

The organization of the thesis is as follows.

- In **Chapter 2**, we present the necessary background concepts for this thesis. The chapter also includes a mapping of those concepts with each of the contributing chapters.
- In **Chapter 3** we address our first research objective **RO1**. Here, we introduce ATOM, a model-driven autoscaler for systems built with the microservices architecture.
- In **Chapter 4** we address our second research objective **RO2**. Here, we present COCOA, a cold start aware capacity planner for an on-premises FaaS platform.
- In **Chapter 5**, we address our final research objective **RO3**. Here, we introduce SampleHST, a sampler for distributed traces generated by microservices in a data center.

Lastly, we present the concluding remarks and future research directions in **Chapter 6**.

Chapter 2

Background

2.1 Introduction

The background concepts we present in this chapter are important to understand the contributions of this thesis. Here, we discuss the subject-matters which are closely related to the research contexts already presented in Chapter 1. These include microservices and containers, serverless and Function-as-a-Service (FaaS) platforms, and distributed tracing. It is necessary to know about these concepts to understand the works we present in Chapter 3-5. In addition, we also introduce the concepts of Layered Queueing Network (LQN), which is an integral component of our contributions in Chapter 3 and 4. We present the mapping of these concepts with our three contributing chapters in Fig. 2.1.

The structure of the sections are as follows. In Sections 2.2-2.4, we discuss the concepts related to Chapters 3-5. Firstly, in those sections, we provide key insights on the subject-matters related to this work. In Section 2.2 and 2.3, the discussions are from the perspective of system performance, since it is the focus of our resource controllers in Chapter 3 and 4. In Section 2.4, we discuss the concepts necessary to understand the contribution in Chapter 5, e.g., different trace structure used in distributed tracing and possible trace aggregation mechanism at runtime. These concepts are not part of our solution, but necessary to understand it. Secondly, we present the current state-of-the-art of the particular research problem, we are addressing, on

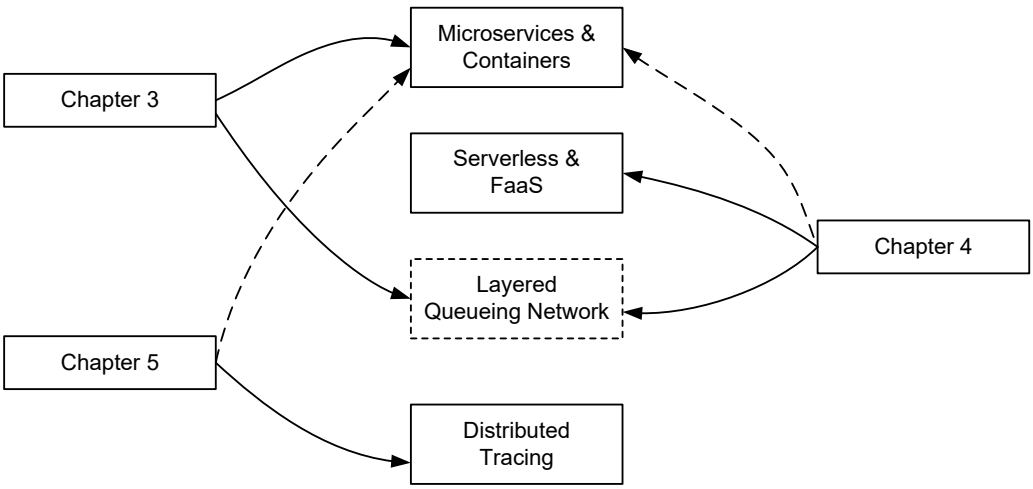


Figure 2.1: Mapping of the background concepts with the contributing chapters. A box with solid lines indicates primary domain of the chapters whereas a box with dotted lines indicates a supporting concept. A dotted arrow indicates a loose connection.

that subject-matter. We also highlight the shortcomings of those works that motivated us to propose new solutions. The purpose of Section 2.5 is different compared to these three sections. It works as a primer to LQN, which is key to understand the resource controllers, namely ATOM and COCOA, we present in Chapter 3 and 4 respectively. The section presents the necessity of LQN-based modeling abstractions, the components of LQN and our options on model solvers depending on the problem context.

2.2 Microservices and Containers

2.2.1 Overview

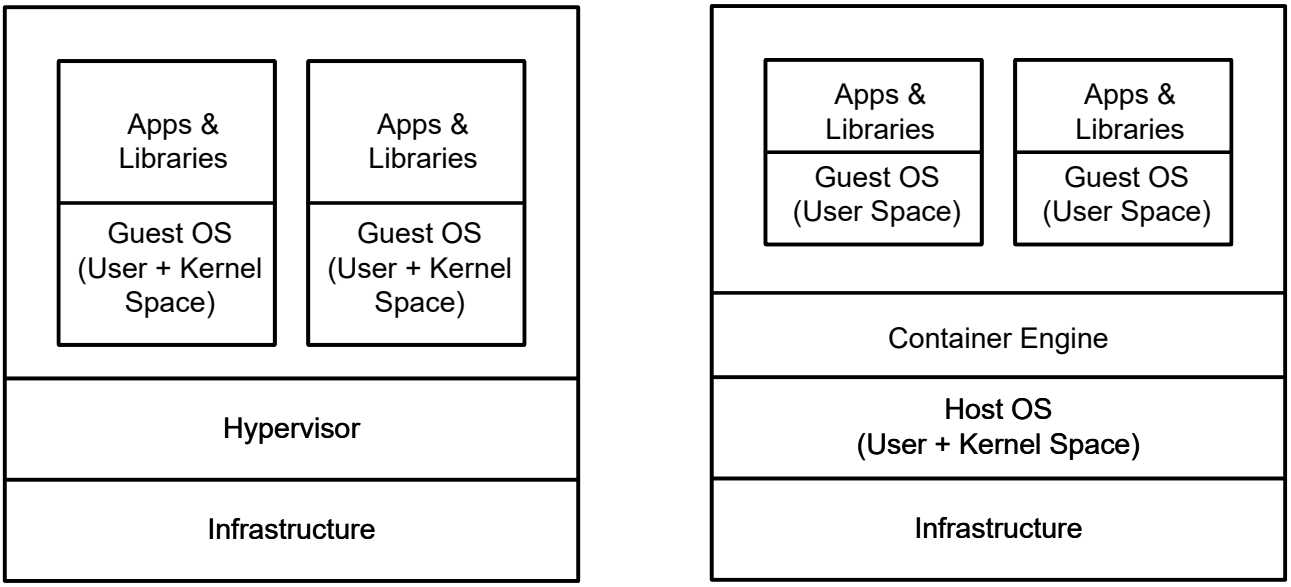
A microservice is a service with finer functional granularity than traditional SOA services [Zim17, BG20]. A system consisting of microservices should follow an architectural style commonly known as microservices architecture. Adopting microservices architecture requires that the software development team follows some development standards, which are not particularly emphasized for SOA. These include, for example, light-weight communication protocol among services and separate data store for each service. Since the microservices architecture complements DevOps [BHJ16, CP21], it is becoming more popular among organizations that follow

such an approach for software development. The DevOps approach focuses on short development life cycles. Such a strategy is aided by microservices architecture since its services are finely grained and have its own data store, which enables small independent teams to develop and deploy it quickly.

In addition to supporting DevOps, microservices have shown indications of improving the system performance compared to monolithic applications [HS17]. Due to such benefits, large companies like Amazon and Netflix are widely using microservices for their software stack [Thö15]. A wide variety of options to develop microservices has further boosted its popularity. A microservice can be implemented with typical programming languages: Java, C#, Python, etc. For communication among the services, the RESTful API is a commonly used option. To store different service data, both relational (MySQL, Oracle, SQL Server, etc.) and NoSQL databases (MongoDB, Cassandra, Redis, etc.) can be used.

To deploy the microservices, container technology is one of the widely used platforms. Containers are a light-weight executable instance of a program with all its runtime dependencies package together. Containers use the Operating System (OS) level virtualization feature of an OS. Due to this feature, the OS kernel can allow multiple isolated user spaces in the virtual memory. Without this feature, the OS would have divided the virtual memory in two segments: a single kernel space and a single user space. However, with OS level virtualization, multiple instances of user space is possible. Each containers run on separate user spaces. The programs within the containers are not aware of the other user spaces and to them the container itself is the real computer. This containerization scheme is completely different from the VM paradigm where each VM has its own OS and the VMs run over a hypervisor. We illustrate this difference in architecture in Fig. 2.2.

The Linux Containers (LXC) [Ber14, Jai20] is one of the early distribution for containerization that was standardized. It is dependent on the containment capability, *i.e.*, isolating processes with different sets of resources, of Linux kernels. At present, Docker [And15, HIB20] is the most prevalent containerization platform, while Kubernetes [BGO+16, Kay20] is the most common container orchestrator. These platforms provide more flexibility in deploying applications, by



(a) VM (b) Container

Figure 2.2: Comparing the architecture of VMs and Containers

bundling all its runtime dependencies together. It requires only a configuration file which mentions all those dependencies. This makes the container platforms ideal for using with DevOps. Besides, it has another potential benefit: recent researches have indicated that such technology can utilize resources such as compute, memory and storage faster and more efficiently than the VM paradigm [CLL19].

Although microservices architecture has shown indications of improving the performance of large scale systems [HS17], recently researchers have identified several performance engineering challenges specific to microservices architecture [HvHK+17]. These challenges should be addressed to improve the performance, scalability and elasticity of a microservices ecosystem. Some of the major challenges include, devising strategies for fine grained monitoring, container level anomaly detection, aligning performance regression testing with containerized microservices, finding appropriate modeling abstractions for microservices and incorporating the container-based infrastructure in performance models.

It is seen that many of these challenges involve the incorporation of container-based platforms. This is because the synergy of containers and microservices can significantly improve the performance of the system. Researchers have already started working on performance evaluation

of containers. In their work [FFRR15], Felter *et al.* have experimentally illustrated that, in majority of the cases, the performance of container deployments are equal or better than virtual machine deployments. They have used popular software like Redis server and MySQL database for their evaluation and presented their analysis from the perspective of latency, concurrency, throughput, utilization, etc. Khazaei *et al.* [KBBML16] and Medel *et al.* [MRBA16] have separately modeled the container provisioning scheme of two different platforms: IaaS Cloud and Kubernetes, using Continuous Time Markov Chain (CTMC) and Object Nets. These works can be used for the capacity planning and resource management of container-based platforms. In addition, Medel *et al.* [MRBA16] provide useful insights of placing CPU, I/O and Network I/O intensive applications in containers. Recently, researchers have shifted their focus to evaluate the performance of containers in IoT-related platforms like Fog computing [PVM+19]. This work, based on their evaluation of multiple policies, aims to suggest best container migration policy under different circumstances. However, all these works are more concerned with the containers and did not incorporate the issues with fine grained microservices and their communications.

To the best of our knowledge, Amaral *et al.* [APC+15] are the first to evaluate the performance of microservices which are deployed in containers. They evaluated both the CPU and network performance considering different architectural alternatives. Although such works provides important guidance to developers, further investigation is required that assesses complex aspects of microservices applications, e.g., replication, container placement and fine-grained resource allocation, to provide a full picture of the performance issues to expect from containerized microservices applications.

Our observation is that there are still research gaps in the domain of performance engineering of containerized microservices. Such works are particularly important for resource management tasks, which is the focus of this thesis. We particularly need to develop performance estimation techniques, based on a model of the system, that can help in taking different resource management decisions. In particular, such techniques should quickly provide performance estimations under various system configuration and workload so that a resource manager can allocate resources based on those estimations. This is an important part of our contribution in Chapter

3, where the focus is on containerized microservices.

Since our first research objective, **RO1**, is autoscaling microservices, we now discuss similar studies from the literature in the next subsection.

2.2.2 Related Work

In principle, since microservices are cloud-native applications, the autoscalers for cloud applications [QCB18] should work on them. However, reviewing the literature we find that such autoscalers cannot realize the full potential of microservices. We first discuss these autoscalers and their limitations. The autoscalers are developed using a wide range of approaches e.g., meta-heuristic algorithms [FFH13, CB17], application profiling [GHBRK12, QCB16] and analytical modeling [LCW+09, ZBYC11, GDK+14, BLF+18]. The meta-heuristic approaches rely on methods like a genetic algorithm [FFH13] or ant colony optimization [CB17] to search a sample space of scaling configurations and suggest the optimal configuration. Application profiling based approaches focus on establishing a relation between the QoS, workload intensity and amount of required software and hardware resources. This relation is then used for dynamic resource provisioning.

The analytical methods focus on using a performance model to gain performance insights of the system. These performance insights are commonly leveraged to minimize a cost function for autoscaling [LCW+09, ZBYC11]. For the performance model, abstractions like LQN [LCW+09, BLF+18] or a traditional queueing network is used [ZBYC11, GDK+14]. Such analytical methods mostly address the issue regarding the amount of required resource share and allocating that share either horizontally or vertically. However, recent researches [ITT18, GDK+18] have highlighted that it is important to make an assessment combining both horizontal and vertical scaling before deciding a scaling strategy. This is because, depending on the workload and service characteristics, vertical scaling may often be the better choice than horizontal scaling.

The overall issue with these approaches are that their capacity estimation technique is not orchestrated for microservices. It has been already suggested that for autoscaling microser-

vices, it is important to consider container level metrics [HvHK+17] and message queue metrics [GLHR18]. This is particularly applicable to profiling and meta-heuristic based techniques as they provide poor estimates if appropriate metrics are not used. The analytical methods also need to incorporate the properties of microservices, like its fractional CPU share, and assess whether the common performance assumptions regarding vertical and horizontal scaling holds for them.

We now discuss the autoscalers targeted at microservices. Researchers have proposed different rule based autoscalers for microservices [FDN16, TBB+17, KRP+17, GLHR18]. Most of these scalers [FDN16, TBB+17, KRP+17] consider simple container level metrics, like CPU and memory usage, rather than exploring new dimensions with metrics like container start-up time [HvHK+17]. Gotin *et al.* [GLHR18] focused on that aspect and used different conditions of the message queues as a measure of system state. Considering such metrics makes these works applicable to microservices. However, these scalers only consider horizontal scaling though vertical scaling can outperform horizontal scaling depending on the nature of the workload [GDK+18].

In recent years, researchers are proposing solutions that consider both vertical and horizontal scaling. Kwan *et al.* [KWJM19] proposed an autoscaler, HyScale, that focuses on the performance trade-offs between horizontal and vertical scaling. HyScale's principle is to scale vertically if the resources are available. If not, it performs horizontal scaling. Rossi *et al.* [RNC19] also emphasize the use of both horizontal and vertical scaling. However, in contrast to Kwan *et al.* [KWJM19], they adopted a model-based approach. It is based on a Reinforcement Learning model that relies on approximations (state transition probabilities and the associated costs) from monitoring data. They realized the so-called Elastic Docker Swarm (EDS) by integrating their method with Docker Swarm. Although these works, consider both vertical and horizontal scaling, they do not consider an important aspect: the bottleneck shift problem [BLV+19]. This problem occurs when there is long chain of microservices, and resolving the bottleneck in lower level microservices introduces bottlenecks in the upper levels.

Bauer *et al.* [BLV+19] presented an autoscaler Chamulteon that focuses on the bottleneck shift problem. It is based on queueing models which are used to forecast system performance and

taking a coordinated scaling actions. Chamulteon also includes a workload forecasting component, which makes it proactive. Barna *et al.* [BKFL17] proposes an Autonomic Management System (AMS) based on Layered Queueing Network (LQN) that inherently offers coordinated autoscaling, addressing bottleneck shifting. However, Chamulteon and AMS both do not consider vertical scaling. Rossi *et al.* [RCP20] proposed a hierarchical autoscaler me-Kube, aiming to decentralize the autoscaler components. Such decentralization makes the autoscaler more scalable when deployed in a large cluster. However, regardless of using queueing models, me-Kube does not address bottleneck shifting as it models each microservice separately rather than the whole application.

FIRM is another autoscaler for microservices presented by Qiu *et al.* [QBJ+20]. It focuses on fine grain resource management of microservices considering resources like cache, network bandwidth, CPU, memory, etc. However, unlike most of the approaches, it opted for a model-free method. Their approach relies on a combination of support vector machine and Reinforcement Learning to identify and allocate resources to bottleneck microservices. They consider both vertical and horizontal scaling and the bottleneck shift problem is addressed using service dependency graphs. However, the performance of FIRM, unlike the model-based analytical methods discussed so far, is largely dependent on the volume and quality of the collected data. In addition, such a method can potentially be slow to react to a sudden change in the workload pattern due to a lack of data.

Reviewing the state-of-the-art shows that there are still research gaps when we consider autoscalers for microservices. The experimental or data-driven approaches, in general, work in a "black-box" manner but require large volume of data for better estimation of scaling strategies. The analytical method in particular are advantageous since they require less data to develop. Such methods work in a "white-box" manner as they require information like system architecture and workflows to construct a model, which are more accessible from a private cloud perspective. An issue with such methods is the model they consider does not allow to incorporate scaling configurations that include a combination of vertical and horizontal scaling, which are better suited for microservices. In Chapter 3, we address these limitations of the current autoscalers for microservices.

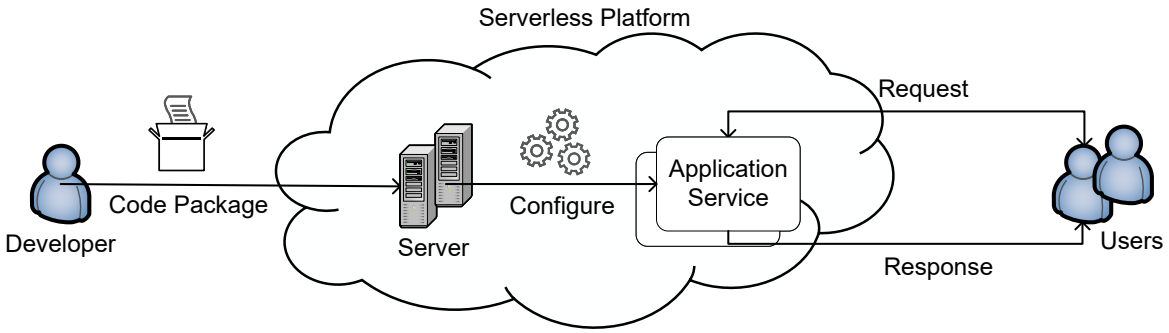


Figure 2.3: Illustration of the serverless execution model

2.3 Serverless and Function-as-a-Service

2.3.1 Overview

Serverless computing [BCC+17] is a cloud-based application execution model that removes the burden of managing the computational resources from the software developer, hence the name serverless. In such an execution model, the service provider manages the computing resources required by the applications deployed on their platform. The amount of resource allocated to an application is proportional to its workload. That is if the workload increases, more resource is allocated to the application. In the case when there is no demand for the application, no resource is allocated to it. This allows an easy implementation of the pay-as-you-go billing model where the customer is primarily[1] charged in terms of transactions. To avail the serverless service, developers need to package their code, in a format defined by the service provider, and simply deploy it using the interface provided by them. For packaging the code, the containerization technology is widely used. An illustration of the serverless execution model is provided in Fig. 2.3.

There are certain application scenarios that align perfectly with the serverless model, for example applications with event-driven workloads. Such applications are often developed using the function-as-a-service (FaaS) architecture and deployed in the serverless platform. In FaaS, developers usually encapsulate the business logic corresponding to an event in a single execution unit, hence the term function is used. This function is triggered in the serverless platform,

[1]The base price per transaction can depend on the amount of required computing resources

when a request for it is received. Although, there is no restriction on a function calling another, this is not regarded as a best practice [Edu19]. This is because it raises problems like increasing the cost and issues with runtime scaling.

We refer to the combination of serverless and FaaS as the FaaS platform. Amazon Web Services (AWS) was the first to provide such a service publicly, namely AWS Lambda. All the major cloud vendors now provide FaaS support. There has also been growing demand to build on-premises or private FaaS platforms. Researchers suggested that such platforms are better suited to scientific applications [PRN+19]. Regardless of the nature of the applications, such an on-premises platform is always helpful from the perspective of privacy and security. To install an on-premise FaaS platform, many open source FaaS frameworks have been developed e.g., OpenFaaS, OpenWhisk and OpenLambda. All these frameworks includes containerization technology, like Docker, to support their functionality.

A major trend in the serverless research community is optimizing the performance of the FaaS platform, which is also important for resource management. The major focus on this aspect is on the scientific applications. An example of such recent works is the CRUCIAL method [BPSAP+19], which facilitates efficient running of machine learning algorithms. It allows developers to manage globally shared states of the algorithm and synchronize across functions, which is a necessity for concurrent stateful applications. Klimovic *et al.* [KWS+18] focused on another aspect - temporary data storage - for function communication during data analytics. They have developed a service named Pocket that dynamically allocates temporary storage resources with reduced cost and high I/O performance. Researchers have also been working to identify different performance related research challenges for the overall serverless architecture [VEIST17]. An interesting challenge they identified is provision for objectively quantifying the performance of FaaS platforms according to the user need. Such a quantification will allow users to do a fair comparison and chose the cost effective option. To realize this, researchers need to emphasize on developing cross-platform performance benchmarks. A recent study confirms that there are still gaps in this domain of research [SL20].

Reviewing the literature, our observation is that, although progress has been made on opti-

mizing FaaS platforms, there is still a shortage of model-based analytical methods providing performance estimates of such platforms. To address this shortage, in Chapter 4, we have used modeling techniques similar to the ones we consider for containerized microservices. This is because, each FaaS function is essentially an individual microservice, likely to be deployed in a container. The key difference here is from the perspective of system architecture, which can be addressed with suitable modifications in the model.

In the next subsection, we discuss the state-of-the-art on capacity planning for FaaS platforms considering cold starts. We particularly focus on the works involving cost, resource management or cold starts as they are more relevant to our second research objective **RO2**.

2.3.2 Related Work

From the perspective of cost, researchers have focused on various FaaS issues. Eismann *et al.* [EGVE+20] presented a technique that predicts the cost of function workflows. They first utilize a combination of neural network and mixture of distributions to predict the response time and output parameter distribution of the functions. These distributions are used in a Monte Carlo simulation to predict the cost of the workflow. Elgamal [Elg18] focused on the similar problem for AWS Lambda but took a different approach. They identify that the cost of these workflows can be reduced by fusing multiple functions in one and placing only the suitable functions in the edge devices. They devised an optimization algorithm that assesses the fusion and placement options and suggests a cost-effective solution. Mahajan *et al.* [MFMR19] have investigated an interesting scenario where the users aim to minimize the cost while the service providers aim to maximize their profit. For this investigation, they have utilized the service cost models along with queueing and game theory. Their study provides insights on how both the user and provider can benefit from such scenarios.

From the perspective of resource management, researchers have mainly focused on runtime CPU allocation considering the QoS [HLZT17, KHL+18]. HoseinyFarahabady *et al.* [HLZT17] proposed a runtime resource controller for a serverless platform that aims to utilize resources while satisfying the QoS requirements in SLA. Their method relies on solving an optimization

problem continuously at runtime with multiple cost functions concerning resource utilization, QoS violations and switching cost. Similar to this work, Kim *et al.* [KHL+18] also focused on ensuring the QoS. They considered the cases with a abrupt surges in the system workload and propose a CPU capping method for such cases. The method basically monitors the system performance and keeps track of the QoS violations. Based on these observations, it sets the CPU cap of a QoS class and assigns the functions to appropriate working threads.

Wang *et al.* [WLZ+18] and Lloyd *et al.* [LRC+18] are among the firsts to investigate function latency considering cold and warm states. Wang *et al.* [WLZ+18] reported many interesting findings regarding cold starts like its impact on function latency significantly increases depending on the function's programming language. They also observed a wide variance in the median cold start times across popular FaaS platforms. Lloyd *et al.* investigated multiple cold start scenarios and found that the performance can even degrade up to 15×. In recent works, researchers are also proposing different solutions to this problem. In a separate work, Lloyd *et al.* [LVZ+18] have addressed cold starts from the end user perspective and mitigated it by periodically sending low cost service requests. Bermbach *et al.* [BKB20] proposed a method developers can use to provision containers in advance by leveraging function composition knowledge. However, these approaches are not applicable from the perspective of service providers.

The server-side solutions to cold start primarily rely on setting up resources beforehand. For example, Mohan *et al.* [MSD+19] have proposed to pre-initialize resources, like networking elements, and associate them with function containers as required. These function containers are saved in a stateless manner in a container pool and re-used later. Shahrad *et al.* [SFG+20] propose a window based approach to load or unload functions by analyzing their invocation patterns. The length of these windows are decided using the distribution of function idle times i.e. the amount of time a function remain idle without receiving a request.

Reviewing the state-of-the art shows that though there are works focusing on cold starts, the aim in such cases has been to mitigate cold starts at runtime rather than incorporating them during capacity planning. None of these works focused on modeling the cold start scenario to

estimate the memory consumption and response time values of the functions. Such estimations could eventually be used for capacity planning. In Chapter 4, we address these limitations.

2.4 Distributed Tracing

2.4.1 Overview

Distributed tracing is a method to track a request as it moves through a distributed environment [Shk19]. It is also known as distributed request tracing. We want to keep track of such requests for future troubleshooting. In principle, this method is applicable to any distributed systems but it is tailored primarily for systems built with the microservices architecture. This is because the traditional tracing mechanism in a web server is incompatible with microservices and thus it was a necessity to build a new method. In a traditional web server, the request is tracked with a thread Id, which is extracted from the log messages. The workflow for a particular request can be found by extracting the logs associated with that thread Id. However, in the case of microservices, for a particular request, not only can a request move to another thread, it can also move to another process running in a separate server. Thus to track such requests, we need to persist a unique request Id rather than a thread Id, throughout the end-to-end path of a request. To implement distributed tracing, developers need to instrument the microservice codes. The goal of this instrumentation is to allow the generation of a unique request Id, which passed along the microservices involved in that request. This process of passing the request Id is often referred as context propagation because it preserves the context to which a service is invoked. The log messages generated by the microservices include that unique request Id, which is used to group the log messages and extract the request workflow.

The structure of log messages generated from distributed tracing will depend on what standard we are following. When following the OpenTracing[2] standard, which is widely adopted in industry, each microservice will contribute a span. The request Id in a span is referred as the trace Id. A span includes the meta-data regarding a unit of work. Companies can apply changes

[2]The OpenTracing project - `https://opentracing.io/`

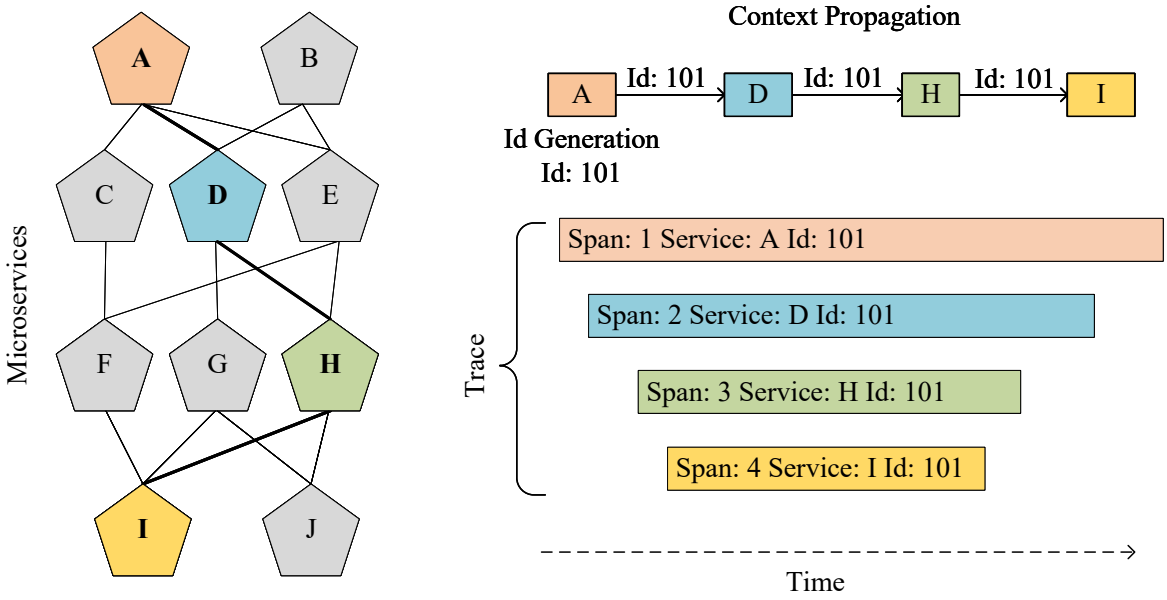


Figure 2.4: An illustration of the distributed tracing concepts. We see a group of microservices and a request that flows from service $A \rightarrow D \rightarrow H \rightarrow I$. A trace Id is generated when the request reaches first service A and the Id is propagated to the next services. The span generated from each service is grouped using this Id.

to the structure of a span but it will ultimately contain a set of property-value pairs and the logs. For example, a property can be what is the HTTP return code or the timestamp when the request is first received. The logs can be any debugging information that has been requested. For further fine-grained information researchers have also proposed other tracing format like the X-Trace [FPKS07]. In the case of OpenTracing the unit of work covers a microservice whereas in the case of X-Trace a unit covers an event inside a microservice, which is referred as a report. This allows a fine-grained analysis of the traces. For a report, the request Id is defined by the task Id property. We illustrate these distributed tracing concepts, considering the OpenTracing standard, in Fig. 2.4.

As mentioned before, a distributed trace needs to be aggregated based on its request Id of the individual components like spans or reports. This aggregation mechanism might need adaptation if we want to perform a run time analysis. Let us assume a trace consists of multiple spans. However, while aggregating, we will not know in advance how many spans a trace will have. The solution to this problem, in principle is as follows. We need to keep track of the trace-span association and assign a timeout value to each of the traces. When a trace expires, its construction (composition of all the associated spans) at that instant will be

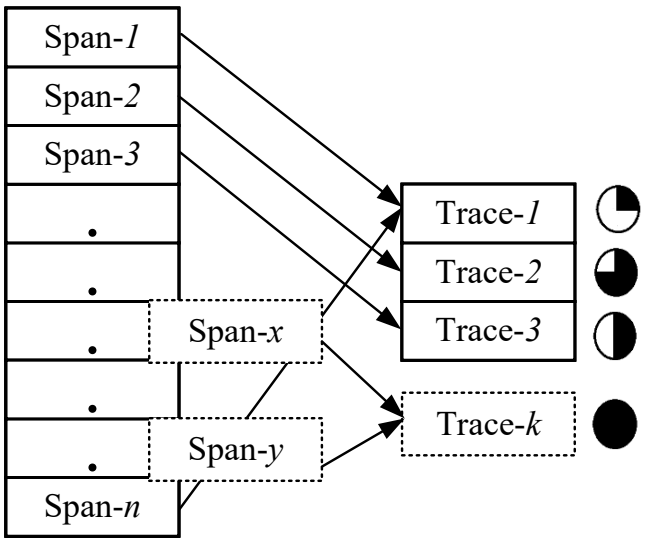


Figure 2.5: Illustrating the distributed trace aggregation mechanism based on timeouts. The circles beside each trace represents the timeout status.

considered as complete. The completed traces will be then placed in a separate buffer. This process is illustrated in Fig. 2.5. On the other hand, if a new span appears before trace expiry, the timeout of the trace will be reset to the original value. The timeout value for the trace should not be a fixed value. It needs to be configured depending on customer needs or any application policy.

Distributed tracing has been widely adopted for troubleshooting of microservices. A common use of it is for anomaly detection [NCK19b] from the traces, where the goal is to identify anomalous service execution paths. This can be used for various purposes e.g., identification of faults. Researchers have also leveraged such traces for performance debugging [GZH+19] i.e., predicting QoS violations and the source of performance unpredictability. There has also been work focusing on privacy risk detection [GAMKF20]. Here, the authors use distributed system traces to identify service paths that can potentially leak personal user data. To perform all these analysis, the trace data typically needs to be stored. This is significant problem for a data center that hosts microservices in the scale of thousands. Data centers like this can produce tens of terabytes of trace data per day [YHC21], requiring a huge storage capacity. A major challenge in distributed tracing is to reduce this storage requirement by sampling the traces that can be potentially useful for further analysis [LCPAM19].

Since our final research objective, **RO3**, is designing a trace sampler for distributed tracing,

we discuss the relevant works on this domain and their limitations in the next subsection.

2.4.2 Related Work

The first step of designing a trace sampler is to be able to differentiate between normal and anomalous traces. There have been many works on anomaly detection from microservices using their generated traces. Wang *et al.* [WZXG20] utilized a call tree extracted from the traces to identify anomalies. Their method initially clusters these call trees to define baseline call patterns. These patterns are then used to detect workflow anomalies in a trace. A similar work in this direction considers is presented by Zuo *et al.* [ZWM+20]. Here the anomaly detector utilizes a temporal model which is learnt from the request execution patterns. Nedelkoski *et al.* in two separate works [NCK19a, NCK19b] investigated the accuracy of deep learning based methods to detect anomalies. In their first work [NCK19a], they focused on the response time anomalies whereas in their second work [NCK19b], the focus was on integrating both the causal relationships and the response time to detect the anomalies. Bogatinovski *et al.* [BNCK20] also used a neural network based method. Their method relies on checking whether a particular span is in the top-K predicted span for that particular position in the trace. Although all these works show promising results for anomaly detection, none of them considered our distributed sampling scenario i.e., they did not present how to transform this anomaly detection result to a sampling decision. This transformation is difficult because we may want to sample normal traces and need to maintain a balance between normal and anomalous traces while sampling.

To the best of our knowledge, there are only a few researches focusing on sampling anomalous traces from microservices. Las-Casas *et al.* [LCMGF18] proposed a sampler based on a hierarchical clustering method PERCH [KMKM17]. They demonstrated that their method can achieve representative sampling, meaning equal share for both normal and anomalous traces. However, being a clustering method, the work should have considered the curse of the data dimensionality [ZSK12], which is not discussed. In addition, this method also requires batch processing but it is not often supported in cloud data centers.

The issue of batch processing is solved by Sifter [LCPAM19]. It takes a sampling decision trace

by trace on the fly. The core concept of Sifter is as follows. Training a neural network with traces should generate a varying degree of training loss based on trace familiarity. A loss can be translated into a sampling probability, which should be inversely proportional to familiarity. The issue with such an approach is even though the probability of an anomalous trace is higher than the normal traces, if the magnitude of that probability is closer to 0 than 1, that trace will not be sampled. As a result, many of the anomalous traces can go unnoticed. This problem is solved in recently proposed sampler named Sieve [HCY+21]. It uses a threshold to first separate the anomalous traces and then amplify the sampling probability. However, the problem with this method is the choice of threshold is critical, particularly when the normal and anomalous traces are not very different.

A common issue with these samplers is they do not bias the sampling decision based on the magnitude of the budget with respect to the fraction of anomalies. Here, a budget is the fraction of traces that the sampler is expected to sample. Although the discussed samplers bias the sampling decision towards smaller trace groups, it will only achieve representative sampling. Such a bias is not sufficient to capture a high proportion of the anomalies when the budget is lower than the fraction of anomalies. Capturing all the anomalous traces, in such cases, is more intuitive as they are morel likely to contain information about system failure. Representative sampling should only be emphasized when the budget permits i.e. the budget is higher than the fraction of anomalies.

Reviewing state-of-the-art shows the research gaps concerning a distributed system trace sampler. This motivated us to develop a novel solution for this problem, which we present in Chapter 5.

2.5 Layered Queueing Network

This section works as a primer to Layered Queueing Networks (LQN), which we use as a modeling abstraction in Chapter 3 and 4. We initially provide the preliminaries on queueing networks and how it has been gradually extended to LQNs. Subsequently, we discuss the cases

where an LQN is used to estimate system performance. Finally, we discuss LQN model solvers and which of them to use under different circumstances.

A software system can be considered as a service station where the users of the system visit to receive a service. If there is only one instance of the station and there is more than a single user, users may have to wait in a queue to receive a service. If the service rate is high, there will be fewer users waiting in the queue to be served. This nature of a software can be modeled as a queueing system to evaluate its performance. A typical queueing system has a queue, where the requests from the clients wait to be served, with a scheduling discipline *e.g.*, First Come First Served (FCFS), based on which the requests are served, and one or multiple servers which serve the requests. In queueing terms, these requests are referred as jobs. It is common to assume that the service time of the servers will follow an exponential distribution and the job arrival rate will follow a Poisson process. Queues with such property are known as Markovian Queues. A system of multiple queues, connected by a routing network, is referred as a queueing network.

The jobs of a queueing network can be of a single type or different types. This is known as the class of the job based on which a queueing network can be either of single or multiple class. Depending on the nature of job arrivals and departures, queueing networks can be divided into two types: open and closed. In a open queueing network, a job can arrive from outside the network and also leave it. On the other hand, in a closed network, a job cannot arrive from outside or depart from the network. Modern client-server systems can be modeled using either open, closed or mixed queueing networks depending on the situation. The scenario where the web server receives requests from the clients can be modeled as a open network. However, considering the scenario where there are a certain number of simultaneous users utilizing the system, a closed network model can be used. There can also be mixed networks, where some job classes are open whereas the other job classes are closed [BGdMT06].

Although the underlying Continuous Time Markov Chain (CTMC) of a queueing network can be used to obtain different performance estimates, it is often computationally expensive. An efficient approach to obtain those estimates for a closed queueing network is using Mean Value

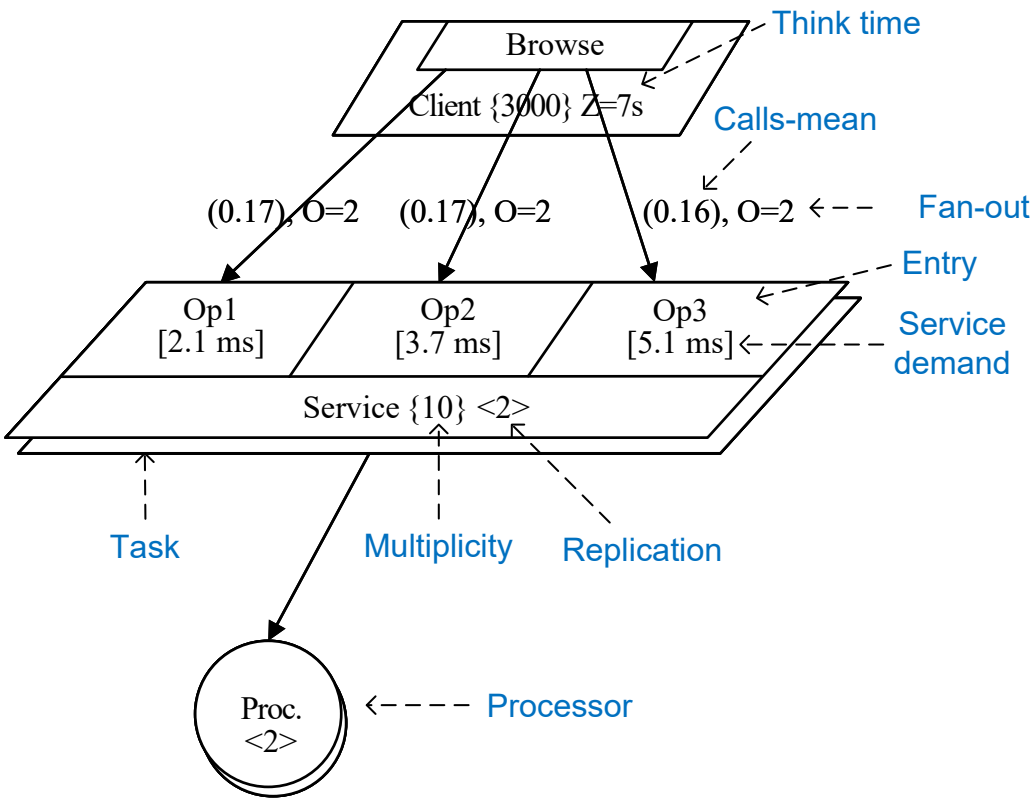


Figure 2.6: Different components of a Layered Queueing Network (LQN), illustrated with a simple model

Analysis (MVA) [RL80]. This technique provides the mean value of different performance metrics like throughput, queue-length, waiting time, etc. Queueing network based abstractions have been extended over the years to adapt with the complexity of distributed systems. One of the earliest works is presented by Sauer *et al.* [SM78], where the authors coined the term Extended Queueing Network (EQN) to propose a class of queueing networks where a job can posses multiple resources at the same time. Layered Queueing Networks (LQN) [WNPM95, RS95, FAOW+08] are another such extension that considers a process can execute in two phases. The first phase is for serving the request and the second phase is for tasks internal to the server such as deleting temporary files, clearing database cache, closing network connections, etc. It can also be used to model parallel service executions [FW98]. Another benefit of LQN is that it can model modern multilayer applications where a server can sometimes also work as an client. This is an important feature for modeling microservices as individual microservices often work as both a client and a server. The multilayer model can be solved iteratively, using techniques like MVA, where each layer is considered as closed queueing network.

A LQN is defined upon a set of basic components:

- **Processor**: A processor can be the actual processor of the system or a placeholder for one or multiple tasks. A processor component is explicitly indicated in Fig. 2.6.
- **Task**: A task is usually used to represent a system process. A task has a queue where jobs are scheduled according to the scheduling discipline of the processor. A task component is explicitly indicated in Fig. 2.6.
- **Entry**: A task is associated with one or multiple entries. Entries can be considered as multiple classes of jobs. All the entries share the same queue of the task. An entry component is explicitly indicated in Fig. 2.6.
- **Activity**: Each of the entries are associated with one or more activities. Activities are the lowest level execution units that consume processor time. An activity component is explicitly indicated in Fig. 2.6.

These basic components can have additional properties such as:

- **Service Demand**: Each of the activities has a host or service demand [SCBK15]. Service demand for an activity can be defined as the total service time of a job across multiple visits to that activity.
- **Think Time**: Think times represent the average time that a user delays between submitting requests. In Fig. 2.6, we see that the Client task has a think time. In an LQN, only a reference task can have a think time. A reference task represents users in an LQN, which in this case is the Client task.
- **Calls-Mean**: An entry or activity can call other entries to complete a request. This call can be either synchronous or asynchronous. These calls are associated with a calls-mean value which indicates on average how many calls are made for a request. These calls-mean values are used as routing probabilities among the components of the system.

- **Multiplicity and Replication**: The processor and task can have a multiplicity and replication value. Simply, multiplicity refers to the number of worker thread and replication specifies the number of replicas of the element serving in similar capacity. If the replication parameter is used, the calls are associated with fan-in and fan-out values. Fan-in indicates the number of replicas of the component from which the call is made. Fan-out indicates the number of replicas of the component to which the call is made. If not explicitly defined, the value for both of them is assumed to be one.

LQN evolved from the necessity of a more expressive abstraction required to model modern client-server systems. Since its introduction, along with its widely used solution tool [FAOW+08], LQN has been utilized to model different distributed systems and performance evaluation scenarios. These include web server [DFJR98], web applications [UM06, SD11], web services [TMW10] and virtual machine provisioning in cloud [SD12]. These works provides important guidelines on modeling and estimating the performance of modern distributed systems. Recently, LQN has been used to design a adaptive runtime controller that provides scaling decisions for VM-based cloud applications [BLF+18]. However, provided that the microservices are deployed in containers, this environment will be distinct from the cloud environment and so for such cases, the controller is not applicable.

LINE [PC17] and LQNS [FAOW+08] are two tools to solve LQN models. Both these tools solve an LQN model iteratively, considering each layer of the LQN as a closed queueing network. Compared to LQNS, LINE can be parameterized with different solvers, not based only on MVA. For example, when parameterized with the fluid ODE solver, jobs can be represented as continuous values, allowing easier estimation of percentiles of response times in high load. Such a metric is important in order to analyze the reliability of the system. Thus, LINE can be used to analyze both the performance and reliability of the system using the same LQN model. LINE also provides option to develop LQN models programmatically, which makes it simpler to model large systems, like a FaaS platform, with known communication pattern. On the other hand, a major advantage of LQNS is due to its efficient implementation with C++, it can solve models significantly faster compared to LINE. This makes LQNS suitable to use in

runtime scenarios like in an autoscaler.

In our case, for Chapter 3 we used only the LQNS solver since we need to quickly solve an LQN model to scale the system at runtime. For Chapter 4, we have leveraged both the LINE and LQNS solver.

2.6 Summary

In this chapter, we have presented the background concepts required for Chapters 3-5. Initially, in Sections 2.2-2.4, we present the subject-matters related to our research objectives. These includes microservices and containers, serverless and FaaS platform, and distributed tracing. We also present recent studies related to our specific research problems. We also highlight their limitations that motivated us to propose our novel solutions. Finally, in Section 2.5, we present a primer on LQN, which is an integral part of our contributions provided in Chapter 3 and 4. The section essential to understand those contributions.

Chapter 3

Capacity Scaling at Runtime

3.1 Introduction

Microservices define a cloud-native architecture for software development [KQ17] that is increasingly accepted in the software industry thanks to their synergy with DevOps [ZBCS16]. Microservices have evolved from service oriented architecture with the aim of delivering a set of scalable services by decentralizing business logic among fine-grained services [PZA+17]. This property results, among other benefits, in greater control of performance since scaling needs for a system can be addressed by adding capacity only to the sections of an application that actually need the extra capacity. In addition, using containers for deploying microservices inherently improves the underpinning performance management thanks to fast start-up times [APC+15] and ease of replication and reconfiguration. When these microservice-based systems are deployed in a private cloud, such deployments, being in-house, provide further flexibility because it allows leveraging different system design artefacts, *e.g.,* UML designs, during scaling.

In this chapter, we have considered an autoscaling scenario from the context of microservices, which in particular are deployed in a private cloud environment. For autoscaling cloud applications, rule based approaches are common in the industry [QCB18]. Such approaches usually scale stateless services horizontally and stateful services vertically. However, recent studies [GDK+18, ITT18] have shown that where both horizontal and vertical scaling are applicable,

they can provide different performance gains based on the current workload. Thus, depending on the current workload, a scaling decision should be based on the potential performance improvement after applying both vertical and horizontal scaling, either separately or in a combination. This can be achieved using the concepts of queueing models. However, using such concepts for microservices involve multiple research challenges. For accurate performance estimations of microservices, a queueing model should both abstract its explicit properties, like fractional CPU share, and the implicit properties inherited from service oriented architecture, like operating simultaneously as a server and client.

Researchers have proposed multiple methods for autoscaling cloud applications based on metaheuristics [CB17], application profiling [QCB16], and analytical methods [BLF+18]. However, these methods are not particularly suitable for microservices as they do not consider such platform specific metrics like container start-up time [HvHK+17] and message queue status [GLHR18]. Moreover, it remains the problem on how to tune the capacity estimation process for microservices. Recently, a number of rule based autoscalers have been proposed for microservices [FDN16, TBB+17, KRP+17, GLHR18]. Some of these scalers utilize message queue metrics [GLHR18], which can provide better insights of the system state than traditional container level metrics used in other studies [FDN16, TBB+17, KRP+17]. However, prior work focuses only on horizontal scaling, whereas we consider both vertical and horizontal scaling.

To aid in this autoscaling context, we present ATOM, a model-driven autoscaler for microservices. For any particular workload, ATOM assesses the effect of the workload on each individual microservice and potential performance improvement after applying various horizontal and vertical scaling configurations. The scaling configurations include the number of replicas for each of the microservices and the allocated CPU capacity (we also used the term "CPU share" interchangeably) for those replicas. To abstract a microservices application and reason on the potential performance gain, ATOM leverages a Layered Queueing Network (LQN) [WNPM95, RS95] model. Such models are instantiated by host demands determined by monitoring the traffic between microservices. By applying various configurations on a model, ATOM aims to provide a scaling strategy that maximizes the revenue of the system with minimal CPU shares. We have formulated this scaling scenario as a non-linear mixed-integer program using

the weighted sum method [MA10]. To search for an optimal solution for a given workload, considering the non-linear nature of the problem and the presence of integer variables, we have used a genetic algorithm (GA).

Overall, our contributions can be summarized as follows:

- an autoscaling controller that maximizes the revenue, while minimizing the CPU shares, of a microservices application without violating the service level agreement (SLA)
- LQN models to estimate the performance of a microservices application with different number of replicas and CPU shares
- a GA based method that searches a sample space of LQN models to provide an optimal scaling strategy
- an experimentation, using real measurements from a microservices application Sock Shop[1], demonstrating that ATOM outperforms the rule based approaches for heavy workloads

Rest of the chapter is organized as follows: Section 3.2 presents a motivating example for our work. Section 3.3 presents the performance modeling aspect of microservices-based systems with LQN. Section 3.4 discusses about our proposed autoscaler ATOM. Section 3.5 presents the experiments and results. The chapter is finally concluded with a summary of the contributions in section 3.6.

3.2 Motivating Example

In this section, we highlight the importance of reasoning between horizontal vs vertical scaling, depending on the current system workload.

[1]Microservice Demo: Sock Shop [`microservices-demo.github.io`]

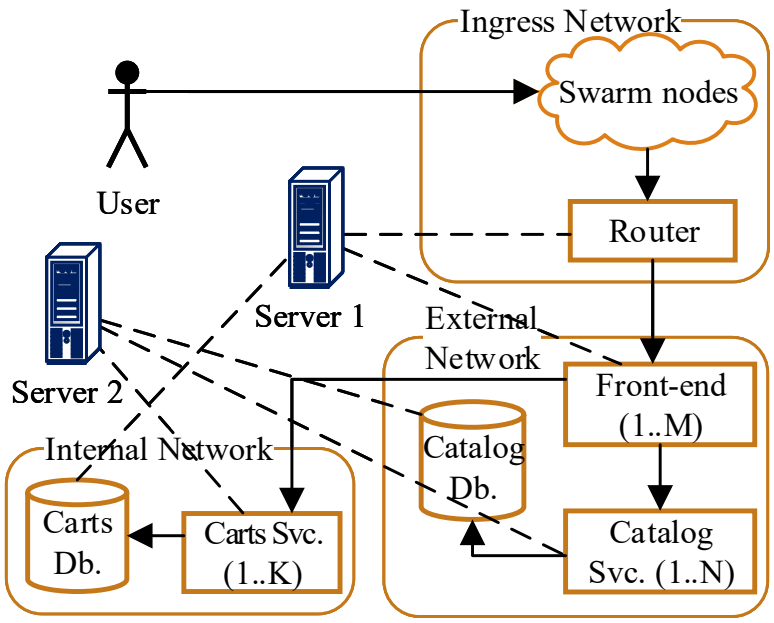


Figure 3.1: The architecture of the sock shop application

3.2.1 Running Case

Before illustrating our example, we first need to define horizontal and vertical scaling from the context of this study. On one hand, we define horizontal scaling (also referred as scale out) as increasing the number of replicas of the microservices. For example, if a microservice previously had one replica and now has two, each with a similar CPU capacity, it means that the service is scaled horizontally. On the other hand, we define vertical scaling (also referred as scale up) as increasing the CPU capacity of a replica of a microservice. So if a replica previously had a CPU share of 0.5 (*i.e.*, it is allocated half a CPU core) and now has 1.0, it means that the service is scaled vertically.

In this example we want to illustrate that scaling strategies for microservices should not be solely based on static rules, such as 'always scale a stateless microservices horizontally" or "always apply either vertical or horizontal scaling to a specific microservice". Instead, if a service can be both vertically and horizontally scaled the scaling decision should be driven by workload characteristics. To illustrate this point, we have used a microservices application named Sock Shop. It is an e-commerce website that allows one to view and buy different types of socks. The architecture of the Sock Shop running case, deployed on two Docker[2] swarm

[2]Docker: Enterprise Container Platform [`www.docker.com`]

Table 3.1: Two cases where the front-end microservice is the bottleneck

Case	System Workload					Front-end Config.	
	Request Distribution			Conc. Users	Think Time	CPU Share	Replica
	Home	Catalog	Carts				
A	57%	29%	14%	1000	7 sec	0.2	1
B				4000		1.0	

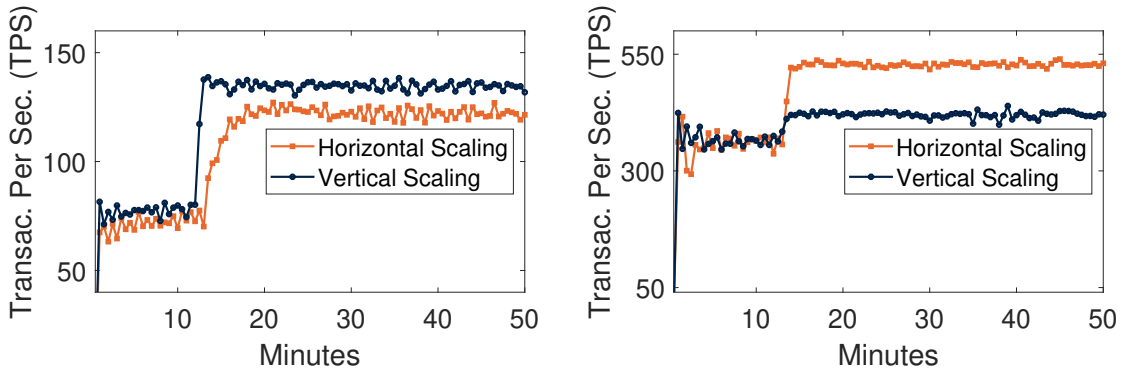
nodes, is illustrated in Figure 3.1. The front-end, catalog and carts services can have up to M, N and K replicas, respectively, which are load balanced by a router service. In addition, the catalog and carts services need to persist data. For that, they have their private database services. These database services and the router service are stateful, whereas the other services are stateless.

We have considered a browsing workload [Men02] in this application, which creates a scenario where a user can visit the website and add or remove items in the cart. In this scenario, we have investigated two cases where the front-end microservice is the bottleneck and requires additional capacity. The workload and system setup used in the two cases are summarized in Table 3.1. The workload is specified by the mix of requests, i.e., the number and types of concurrent users issuing synchronous requests, and their think times, the times in-between a request completion and issue of the next request. The system configuration includes the CPU share and the replica number of the front-end microservice. A CPU share of 0.2 means that the microservice can at most utilize 20% of a CPU core, even if the rest of the CPU remains idle. The cases, A and B, represent a light and heavy workload respectively.

3.2.2 Comparing scaling rules

For the evaluation of the two cases, vertical scaling was applied by doubling the CPU share of the front-end microservice (the bottleneck); horizontal scaling was applied by doubling the number of replicas of the front-end. The results in Figure 3.2, for Cases A and B, show clearly that it is better to tailor the choice of vertical and horizontal scaling strategy to the workload rather than always depending on one strategy or the other.

In the light-workload case (A), Figure 3.2a shows that both strategies gave increased throughput, but vertical scaling responded more quickly and reached a higher steady-state throughput



(a) Case A (b) Case B

Figure 3.2: Effect of vertical and horizontal scaling over the front-end microservice in two different cases

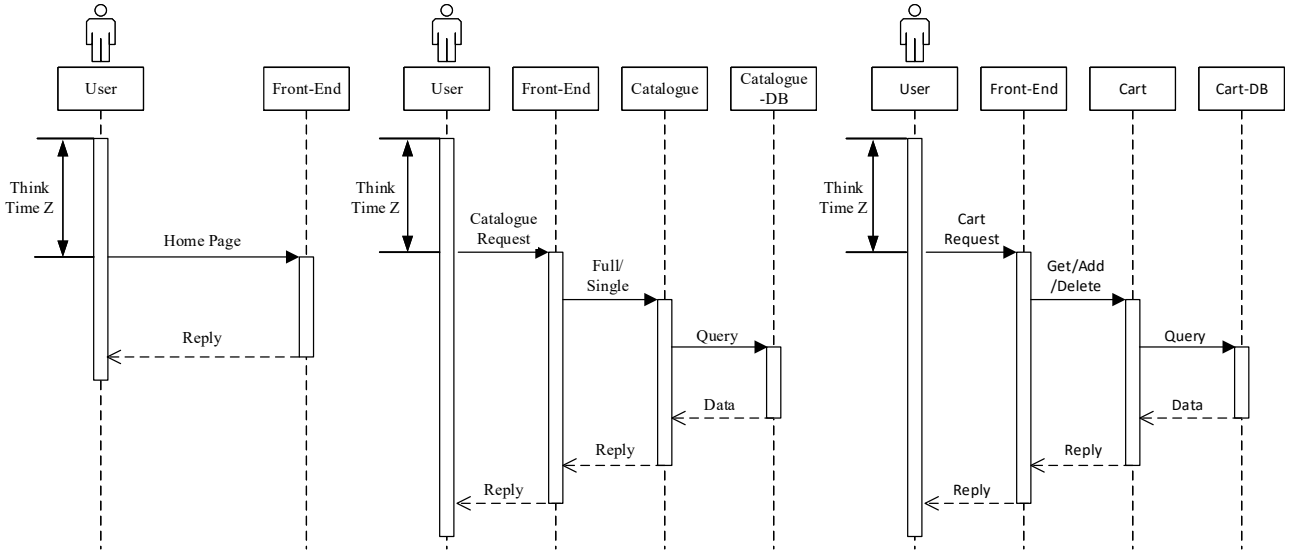
(about 10% higher). The more rapid response is due to a faster ramp-up of capacity, and the higher steady-state capacity is due to the well-known inefficiency of multiple servers sharing a light or moderate load, compared to a more powerful single server [HB13]. This inefficiency is captured in the queueing model used by ATOM for decision-making.

In the heavy-workload case (B), Figure 3.2b shows that vertical scaling is much less effective at increasing steady-state capacity, about 20% less. The two strategies respond about equally quickly in heavy load, so horizontal scaling is overall much more effective. Vertical scaling gave lower throughput in heavy traffic because the front-end microservice code is not internally parallel, thus it cannot leverage the availability of the extra CPU cores. This is not uncommon in software, as demonstrated by the Sock Shop implementation itself.

Summarizing, our results suggest that, when a service can both be vertically or horizontally scaled, an optimal runtime autoscaler would reason on which of the two scaling actions to take based on workload, individual microservice features, and overall software system architecture. In the next sections, we adopt a queueing-based approach to address this problem.

3.3 Performance Model

This section illustrates the use of Layered Queueing Network (LQNs) for performance modeling of microservices. It also presents different service demand estimation methods, which is required



(a) Request for home page (b) View catalogue or items (c) Add or remove items or view cart

Figure 3.3: Different user interaction scenarios considered in the LQN model

to instantiate the LQN models in concrete systems.

3.3.1 LQN Modeling

For LQN modeling, we have identified three user behavioral patterns concerning the sock shop running case presented in section 3.2. These are: visiting the home page, viewing the catalogue or items and viewing or adding or removing items from the cart. The users are associated with a think time. All the user requests are received by the Front-end service. To the front-end, the request is either for the home page or for catalogue or cart service. If the request is related to catalogue or cart service, the front-end forwards it to the corresponding service. The cart or catalogue service decides the specific nature of user request. For the catalogue service, it can be either for viewing the whole catalogue or a single item. Whatever, the request is, the catalogue service has to send a query, corresponding to the request, to its database before it can send the reply to the front-end. The same pattern also applies for the cart service. The only difference is that, it can receive three types of request instead of two types. When the front-end receives a reply from either catalogue or cart service, it forwards the reply to the user. These scenarios are depicted in the sequence diagram presented in Figure 3.3.

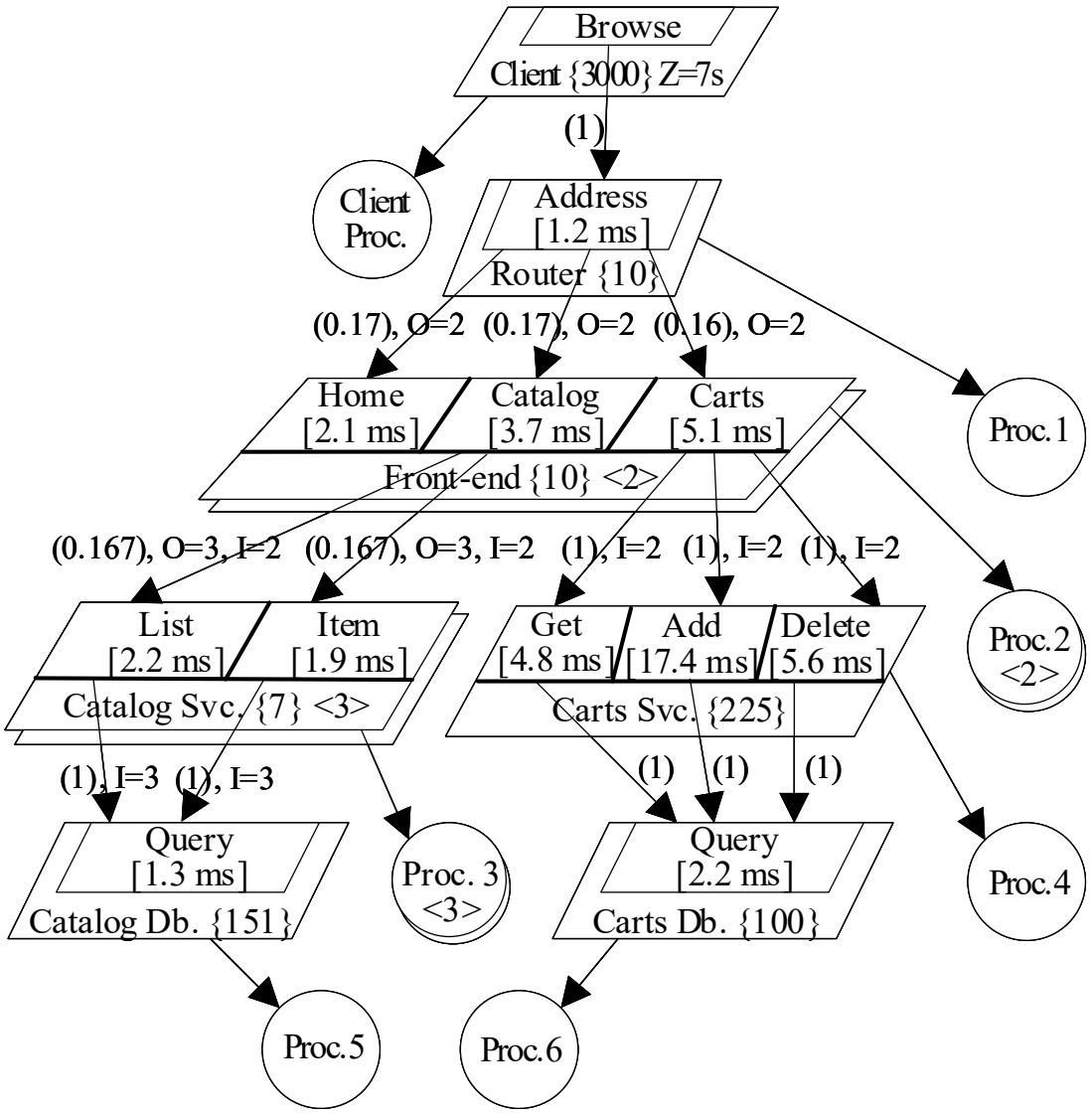


Figure 3.4: An LQN model of the Sock Shop application. The boxes, nested boxes and circles represent tasks, entries and processors respectively. Calls are shown by arrows, which are associated with a call mean, fan-in and fan-out. The values between square, curly and angle brackets specify service demands, multiplicity and number of replicas respectively.

LQN models can naturally abstract these scenarios where a microservice can sometimes also be a client of another microservice. This property often leads to the use of LQN in modeling distributed systems [UM06, SD11, TMW10] and motivates the adoption of these models in microservice autoscalers. For ease of illustration, the LQN model[3] we have derived for the running case is shown in Figure 3.4. In the LQN model, microservices are abstracted by tasks. The reference task (Client) represents the system workload. The entries of the task, which are equivalent to service classes in a queueing network, hosts the features or APIs exposed by the

[3]The model is also available for download in `https://github.com/alimulgias/ATOM`

microservices to their clients (entries). The tasks are placed in a processor, which in this case represents the CPUs in the host servers.

In a LQN model, the performance impact of vertical or horizontal scaling can then be assessed either by scaling the service demands of the activities in a processor according to the CPU share (vertical scaling) or, for horizontal scaling, by increasing the level of replication of a task. This inherently increases the number of service queues for that task. An autoscaler can then reason on expected performance by analyzing the LQN with standard solvers [FAOW+08, PC17] and obtaining corresponding metrics such as throughput, utilization and response times expected for each microservice after performing the scaling action.

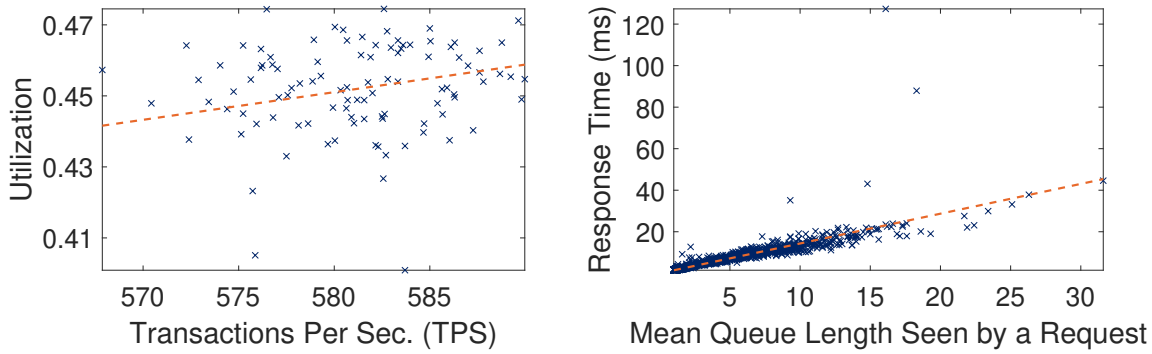
3.3.2 Service Demand Estimation

The tasks of an LQN model can include one or more activities. An activity consumes a CPU time which is represented by its service demand, which needs to be estimated to parameterize an LQN. A common method is to use a linear regression model based on the utilization law [SCBK15] presented in (3.1).

$$U_i = \sum_k X_k D_{i,k} \tag{3.1}$$

Here, U_i means the utilization of resource i, X_k means the throughput of class k and $D_{i,k}$ means the service demand of a job of class k in resource i . To get the service demand, initially some utilization and throughput samples are collected from system measurements. The utilization function is then fitted by ordinary Least Square (LSQ) regression, possibly with non-negativity constraints, to obtain the service demands.

The regression method assumes some variability in the observed throughputs, which is not always present in observations on microservices. To illustrate this, in Figure 3.5a, we have plotted the estimation data, for a single experiment, of the *View item* feature of the Sock Shop application. With such single experiments, it is difficult to obtain a wide range of throughput, *i.e.*, Transactions per Second (TPS), and utilization values, which is also seen in the figure.



(a) Utilization technique

(b) Queue Length technique

Figure 3.5: Service demand estimation using two different techniques

Within this narrow range, the data do not show strong correlation between the variables, which in turn means that accurate estimates of demand cannot be found this way. We may need to run several experiments to obtain a wide range of values. This, in particular, is a issue for microservices as they are finely grained and contains a simple business logic. This makes microservices less resource intensive than traditional applications. Thus, estimating their service demands based on utilization becomes more difficult.

More accurate estimates can be found using more fine-grained observations on the system, of the response time of individual operations or groups of operations versus as a function of the queue length at their arrival [KPSCD09]. This technique is based on the formula for estimating response time using the mean-value analysis arrival theorem [KPSCD09]. The formula is presented in Equation (3.2). In the equation, R_i is the mean response time of a job of class i, $T^{(i)}$ is the residual time of the job in execution seen by the job of class i upon arrival, D_k is the mean host demand of class k and $A_k^{(i)}$ represents the mean number of jobs of class k seen by the job of class i upon arrival.

$$R_i = T^{(i)} + \sum_{k=1}^{K} D_k(c + A_k^{(i)}) \tag{3.2}$$

where,

$$c = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

Table 3.2: Workload patterns for model validation

#	Request Distribution			Concurrent Users	Think Time
	Home	Catalog	Carts		
1	57%	29%	14%	1000, 2000, 3000	7 sec
2	34%	33%	33%	1000, 2000, 3000	
3	57%	29%	14%	1500, 2500, 4000	10 sec
4	34%	33%	33%	1000, 2000, 3000	

When applied to the previous case, the results are shown in Figure 3.5b. From the figure, it is seen that the variability in data is significantly higher than the data for utilization samples, simplifying the estimate of demands. It is also evident that the estimate will be insensitive to anomalies in the data. This suggests that the technique is applicable for microservices.

3.3.3 Validation

To validate the LQN model output against real system measurements, we have performed multiple experiments. We have considered a subset of the Sock Shop application from Figure 3.1 that contains business logic. The workloads for the experiments are presented in Table 3.2. The request mix and the number of concurrent users have been selected such that they create light, normal and heavy load in the system. The think times are set as suggested in literature for such studies [Men02, RCK+09]. For workload 2 and 4, we have deployed the microservices in Docker compose mode, creating a scenario with a single host server. For workload 1 and 3 we have deployed the microservices in two Docker swarm nodes, creating a scenario with multiple host servers, where server 1 hosts the front-end and cart services and server 2 hosts the other three services. Within these servers, to avoid the approximation error of multiserver queueing nodes, we have kept a single CPU online. For workload generation, we have used the Locust[4] tool with a distributed load testing configuration. All the models are solved using the LQN simulator (LQSIM).

The percent errors, considering the TPS and utilization, between the model and system measurement are presented in Table 3.3. We presented the minimum, maximum and average errors for both TPS and utilization. From the table, it is seen that all the average errors are less

[4]Locust - A modern load testing framework [www.locust.io]

Table 3.3: Percent error between model and system measurement

Service Name	TPS % Error			Util. % Error		
	min	max	avg	min	max	avg
Front-End	0.05	7.68	2.26	1.06	9.98	5.05
Carts Svc.	0.04	6.08	2.17	0.37	4.08	1.89
Catalog Svc.	0.09	7.83	2.79	0.66	4.01	2.04
Catalog Db.	0.43	7.75	2.81	0.05	1.7	0.95
Carts Db.	0.06	4.29	1.71	0.58	7.45	3.5

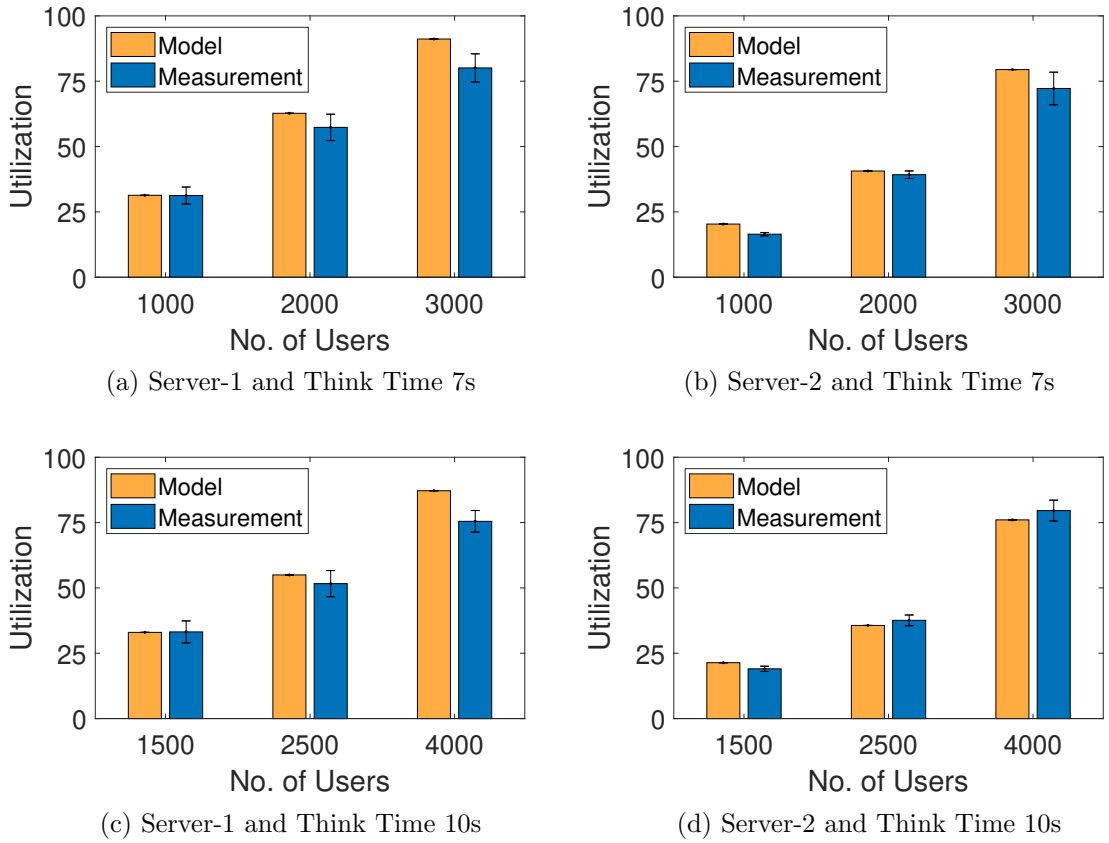
than 5.05%, and even the maximum error, observed in the utilization of front-end microservice, is only 9.98%. These errors are well accepted for performance modeling [LZGS84, SD11]. To further investigate the model accuracy, we have considered the total CPU utilization of the two servers participating in Docker swarm. From the results, as presented in Figure 3.6, it is seen that, for all the workloads, the model estimations are close to the system measurement. To further demonstrate accuracy of individual service, we have used a heavy workload (workload 1 in Table 3.2) with 3000 users. From the results, as presented in Table 3.4, it can be seen that for all of the components, the throughput and utilization values are within the acceptable range of 5% to 10%[LZGS84, SD11]. These results indicate that the model can be used for further analysis like evaluating the system performance for different scaling configurations.

Table 3.4: Feature and microservice specific TPS and utilization from model and measurement for workload 1 with 3000 users

Service Name	Feature Name	Model TPS	Measured TPS	% Error	Model Util.	Measured Util.	% Error
Front-end	Home	236.3	221.3	6.3	75.2	65.9	9.3
	Catalog	120.2	110.9	7.7			
	Cart	58	55.6	4.1			
Carts Svc.	Get	19.1	18.5	3.1	16	14.2	1.8
	Add	19.1	18.5	3.1			
	Delete	19.7	18.5	6.1			
Catalog Svc.	List	60.2	55.5	7.8	19.2	15.4	3.8
	Item	60.1	55.5	7.7			
Catalog Db.	Query	120.2	110.9	7.7	12	12.6	0.6
Cart Db.	Query	58.1	55.6	4.3	48.2	44.3	3.9

3.4 ATOM: Autoscaling Microservices

In this section we discuss the details of our autoscaler ATOM. Initially, we present how a user can use ATOM, the system considered by ATOM and its components. Subsequently,



(a) Server-1 and Think Time 7s

(b) Server-2 and Think Time 7s

(c) Server-1 and Think Time 10s

(d) Server-2 and Think Time 10s

Figure 3.6: Utilization of different servers according to model estimation and system measurement for workload pattern 1 and 3

we formulate the problem that is addressed in ATOM. Finally, we present the optimization method incorporated in ATOM to find solutions for the problem and suggest optimal scaling configurations.

3.4.1 Context and Architecture

To use ATOM, the user needs a deployed microservice application and its LQN model. The LQN model can be obtained in two ways. The first method is likely for an application deployed in a private cloud, which is the primary focus of this work. In such a case, the UML design of the application should be available since the application is deployed within the organization. These UML artefacts can be can be used to automatically generate a model following a transformation technique similar to Gu *et al.* [GP05]. The second method is for applications deployed in a public cloud. In such cases, the service provider may not have access to the design artefacts

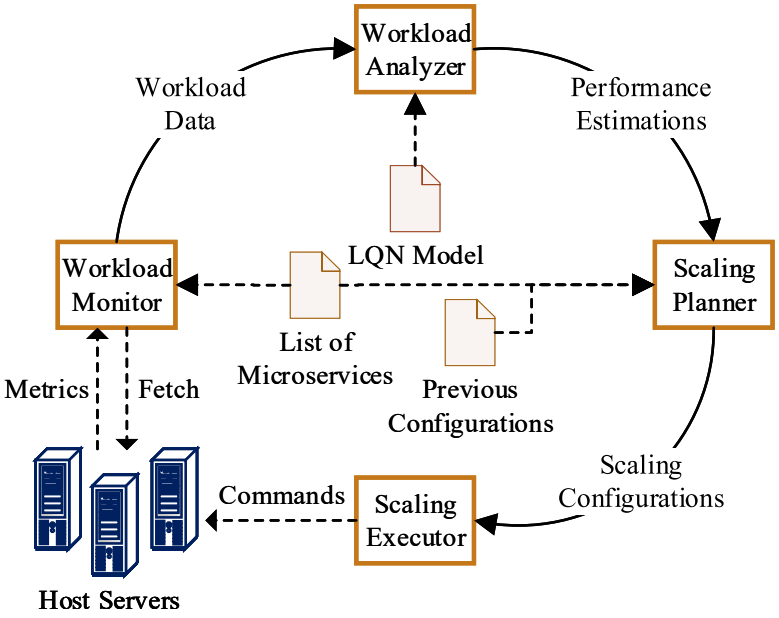


Figure 3.7: Overview of ATOM's components and their interactions

or model-based specifications. However, due to the fine-grained nature of microservices, a suitable model may be developed in principle by only monitoring the communication among the microservices. As mentioned earlier, service demands are important parameters for LQN models. In this work, we have assumed that service demands are estimated offline, using either the same private cloud infrastructure that will be used for deployment or an infrastructure similar to it. We plan in future work to add an online demand estimation method.

ATOM considers a system which executes different transactions using microservices. Each transaction type combines a set of features of the microservices and yields different revenue. Thus, the transactions can be prioritized based on their revenue. ATOM acts periodically. At a moment when it is invoked, suppose the system has previously been optimized but has experienced an increase or other change in the workload. ATOM aims to estimate a scaling strategy for each of the microservices so that the revenue of the system is still optimal.

The components of ATOM, as shown in Figure 3.7, are aligned with the MAPE-K (monitor, analyze, plan and execute with a shared knowledge base) [KC03] control model. The workload monitor counts the user requests for each feature of the system, using a set of time intervals within a monitoring window [ML13] to obtain a set of samples for each feature. These samples are forwarded to the workload analyzer after a monitoring window is completed. Based on

the monitoring data, the workload analyzer needs to update two things in the LQN model: the concurrent number of users in the system (N) and the request mix. Since we are using a workload generator, the system that we consider is a closed loop system. Thus, we assume that we know the value of N. To calculate the request mix, the workload analyzer initially determines the average request count for each type of feature in a monitoring window. From these average request counts, the request mix is determined as the fraction of requests for different system features. The analyzer then updates the LQN model with the value of N and new request mix. To update the model, the value of N is set as the multiplicity of client task and the request mix is used to update the call parameters among the entries of the model.

The analyzer uses a search strategy, described in Section 3.4.2 and 3.4.3, to vary the number of replicas (representing horizontal scaling) and the host demands (representing vertical scaling) and find the best combination. This result is forwarded to the scaling planner. Initially, the planner translates the received result from modeling to a system perspective using a map between the LQN model and the microservices. After that, the planner decides the scaling strategies (number of replicas and CPU share of each replica) for each microservices. The planner also includes an option to use the previous scaling configurations as a reference to prevent a drastic change in the current monitoring window. This can be achieved either from the perspective of TPS or total allocated CPU capacity. If the option is chosen, the planner does not change a configuration in the current monitoring window unless the TPS improves or total CPU share does not change more than a threshold value. After that, the planner creates a scaling script for the scaling command executor. Finally, the executor component runs the scaling commands specific to each microservice. All these components share a knowledge base that includes the LQN model, list of microservices and their mapping to LQN and scaling configurations from the previous iteration.

3.4.2 Problem Statement

ATOM provides the scaling decisions by solving multiple LQN models. To generate these models, ATOM needs to decide the number of replicas for each microservice and the CPU

shares for all the replicas of each microservice. Considering there are N microservices, these decisions are represented by the following variables.

- $r^{(t)} = [r_i^{(t)}], r_i^{(t)} = 1, \ldots, Q_i, \forall i = 1, \ldots, N$. Replica vector: denotes the number of replicas for each of the microservices i for the time interval t, where Q_i is the upper bound of replicas that a microservice can have.
- $s^{(t)} = [s_i^{(t)}], s_i^{lb} \leq s_i^{(t)} \leq s_i^{ub}, \forall i = 1, \ldots, N$. CPU share vector: denotes the CPU share for each of the replicas of a microservice i for the time interval t. The share is bounded by upper and lower bound s_i^{ub} and s_i^{lb}.

Our objective is to determine the values of $r^{(t)}$ and $s^{(t)}$ that maximize the revenue $B^{(t)}$ and minimize total allocated CPU capacity $C^{(t)}$ for the next time interval t. Before determining these values, their upper and lower bounds need to be set. For a private cloud, this can be done without any assumptions since different service information such as, which services can utilize multiple CPUs or which services can have multiple replicas, is more likely to be available. This information can be utilized to reduce the size of the vectors and also the number of scaling strategies that need to be evaluated. From a public cloud perspective, since such information is unlikely to be available, we have to set the bounds with rules *e.g.*, each of the service replicas will at most be allocated a single CPU.

We define the revenue $B^{(t)}$ as the number of transactions completed per unit time. The transactions are weighted by a numerical coefficient expressing their business value. These weights vary according to the priority of a feature of the application. The features are associated with the service classes of different microservices. We consider that a microservice i has E_i number of service classes. The number of transactions per unit time for a given period t and a particular service class j of microservice i is denoted by $X_{ij}^{(t)}$. The weight of the transactions is denoted by ψ_{ij}. In calculating the revenue, it is common to ignore the transactions of a particular set of services. Thus we define a set of revenue calculation services $I = \{i \in \mathbb{N} | i \in \{1...N\} \wedge \phi(i) \neq 0\}$, where $\phi(i) = 0$ if a microservice i is ignored. We also define the set of replication values and CPU share vector as $R = \{r \in \mathbb{N}^N | \forall i = 1 \ldots N, r_i = 1 \ldots Q_i\}$

and $S = \{s \in \mathbb{R}^N | \forall i = 1, \ldots, N, s_i^{lb} \leq s_i \leq s_i^{ub}\}$. Considering these variables, we can calculate $B^{(t)}$ according to (3.3).

$$B^{(t)} = \sum_{i \in I} \sum_{j=1}^{E_i} \psi_{ij} X_{ij}^{(t)} \tag{3.3}$$

The total CPU share $C^{(t)}$ can be defined simply as the sum of the CPU shares of all the replicas of the microservices. This can be calculated as $C^{(t)} = \sum_i r_i s_i$. With these definitions of $B^{(t)}$ and $C^{(t)}$, we express the objective function $\Theta^{(t)}$ for our optimization problem as a weighted sum in (3.4).

$$\Theta^{(t)} = \max_{(r^{(t)}, s^{(t)})} \tau_1 \hat{B}^{(t)} - \tau_2 \hat{C}^{(t)} \tag{3.4}$$

$$\text{s.t:} \quad W_{ij}^{(t)}(r^{(t)}, s^{(t)}) \leq W_{ij}^{\max}, \forall i \in I, j = 1, \ldots, E_i \tag{3.5}$$

$$C_k^{(t)}(r^{(t)}, s^{(t)}) \leq C_k^{\max}, \forall k \tag{3.6}$$

$$U_i^{(t)}(r^{(t)}, s^{(t)}) \leq U_i^{\max}, \forall i \tag{3.7}$$

$$r^{(t)} \in R, s^{(t)} \in S$$

In (3.4), $\hat{B}^{(t)}$ and $\hat{C}^{(t)}$ represents the normalized value of $B^{(t)}$ and $C^{(t)}$. These are normalized since it is suggested for a weighted sum method if the magnitudes of the objectives differ significantly [MA10], which is our case. The relative importance of revenue and total CPU share are defined by the weights τ_1 and τ_2. The constraints for the optimization problem are presented in (3.5)-(3.7). The function $W_{ij}^{(t)}$, in (3.5), estimates the response time for a service class j of microservice i for a time period t. The estimated response time cannot be higher than the maximum limit $W_{ij}^{\max}$ mentioned in the SLA. The function $C_k^{(t)}$, in (3.6), calculates the total allocated CPU share of a server k for a time period t. This value must not exceed the server's capacity $C_k^{\max}$. In (3.7), $U_i^{(t)}$, represents the utilization value of a microservice i for a time period t, which must not exceed its limit $U_i^{\max}$.

A summary of the decision variables and performance measures is provided in Table 3.5. These variables and measures are utilized by ATOM to maximize the objective function.

Table 3.5: Relevant notation for the optimization problem

$r^{(t)}$	Replica numbers for all microservices for time period t
$s^{(t)}$	CPU shares for all microservices for time period t
$X_{ij}^{(t)}$	TPS of service class j of microservice i for time period t
$W_{ij}^{(t)}$	Response time of service class j of microservice i for time period t
W_{ij}^{max}	Maximum allowed response time of service class j of microservice i
$C_k^{(t)}$	Total CPU share allocated in server k for time period t
C_k^{max}	Maximum CPU capacity of a server k
$U_i^{(t)}$	The CPU utilization value for a microservice i for time period t
U_i^{max}	The maximum CPU utilization value allowed for a microservice i

Algorithm 3.1 Solution Candidate Generation

Input: LQN Model Γ, *timeLimit* and *tolerance*
Output: Set of solution candidates G

Init: generate initial $config$ set and $G \leftarrow \varnothing$
while $time \leq timeLimit$ **do**
  $currentCandidates \leftarrow \varnothing$
  for each $(r, s) \in config$ **do**
    $\Gamma' \leftarrow updateReplication(r, \Gamma)$
    $\Gamma'' \leftarrow updateCalls(r, \Gamma')$
    $\Gamma''' \leftarrow updateHostDemand(s, \Gamma'')$
    $(fval, c) \leftarrow solveModel(\Gamma''')$
    if $c \leq tolerance$ **then**
      $currentCandidates \leftarrow currentCandidates \cup \{(r, s), fval\}$
    end if
  end for
  $config \leftarrow generateConfig(currentCandidates)$
  $G \leftarrow G \cup currentCandidates$
end while

3.4.3 Optimization Method

To find the optimal scaling strategy, ATOM needs to generate a set of solution candidates and evaluate their quality. The steps for solution candidate generation is presented in Algorithm 3.1. The auxiliary functions that has been used in the algorithm are summarized in Table 3.6. The algorithm requires three parameters: the LQN model, time limit for algorithm execution and the constraint tolerance value.

Initially, the algorithm generates a random set of configurations ($r \in R$,$s \in S$). This initial set should satisfy the constraints in (3.6). For a given r and s, the total CPU share C_k allocated in a server k can be calculated as $\sum_i r_i s_i z_{ik}, \forall i$ where $z_{ik} \in \{0, 1\}$ represents whether the replicas of a microservice i is placed in server k. After that, a time bounded searching starts, based on

Table 3.6: Auxiliary functions for Algorithm 3.1

updateReplication	Update the replication parameter of the tasks and processors.
updateCalls	Update the fan-in, fan-out and call means for all tasks and activities.
updateHostDemand	Update the host demand of each activity of a task.
solveModel	Solve the model and provide the value of the objective function.
generateConfig	Generate a new set of population using the current set of candidate solutions.

the *timeLimit* parameter, to find the optimal solution. Ideally, this searching process should not be time bounded as we may not find the optimal solution within that time limit. However, as this is a runtime searching process, it is impractical to run the process indefinitely. Thus, we must use this time bound, following which the current best solution will be returned, to ensure that a scaling strategy is always suggested after each monitoring window. The value of this bound should be less than the monitoring window because otherwise, a scaling strategy can not be executed before receiving the data of the next monitoring window. So, for example, in case of a 5 minute monitoring window, we can use a 2 minute time bound. In addition, to ensure that the searching process does not start with a zero-knowledge, if the workload is unchanged, we can utilize the scaling configurations from the previous monitoring window to generate the seed in the current window.

The search process is as follows. For each r and s pair, the LQN model is updated. This update process has three steps. At first, the *updateReplication* function updates the replica number of a task-processor pair according to the replicas of the corresponding microservice. After that the fan-in and fan-out values of the tasks and the calls parameter among the entries is updated by the $updateCalls$ function. Considering a chain of tasks $A \rightarrow B \rightarrow C$, the fan-in of task B is $F^B_{in} = r_A$ and the fan-out is $F^B_{out} = r_C$, where r_B and r_C is the number of replicas of task B and C. According to the current request mix, if the value of a call parameter from B to C is $D_{B,C}$, it is updated as $\frac{D_{B,C}}{r_C}$. Finally, the $updateHostDemand$ function scales the host demands of the activities of a task by dividing the host demands by the CPU share of the corresponding microservice for that task.

After completing the update process, the model is solved. Since a large number of models

need to be solved at runtime, the solution for each model must be generated quickly. Thus, an analytic model solver LQNS [FAOW+08] is used for solution generation. The solver is invoked with the option Bard-Schweitzer single step mean value analysis (MVA) [BGdMT06], which allows faster model solving. We have checked the solution provided by LQNS to verify that the estimated r and s satisfy the response time and utilization constraint in (3.5) and (3.7) considering the given *tolerance*. The response time W_{ij} is obtained from the residence time for the corresponding call to the entry (representing a service class j) of a task (representing a microservice i). The residence time is defined as the sum of queueing and service time for a call to an entry. In LQN context, the service time is an output that represents the time a server remains busy to serve a request, including the nested service calls. The utilization value U_i^t for a microservice i is estimated from the utilization value of its corresponding processor in the model. If the constraints ($W_{ij}^{(t)}$, $C_k^{(t)}$ and $U_i^{(t)}$) are satisfied, the throughput ($X_{ij}^{(t)}$) values of the tasks, corresponding to the features of the microservices, and the configuration (r and s) is added to the set of current candidate solutions. This set is used to generate a set of new configuration for the next iteration.

Since it is infeasible to evaluate all the scaling configurations at runtime, only a subset of solution candidates will be generated. In such cases, a meta-heuristic approach is commonly used to generate a sufficiently good set of solution candidates. Here, we have used a genetic algorithm (GA) for this purpose. GA can provide optimal solutions for highly non-linear problems [Mit97] and is a powerful approach for problems with integer variables [YGL96]. This is important in our case as we have a non-linear mixed-integer program. The integer variables in our case is the number of replicas. The non-linearity originates from the involvement of the LQN model, as optimizations based on such queueing models are generally non-linear [CSB02]. It is important that how these numbers are increased depending on the characteristics of the current workload. In ATOM, we used the MATLAB implementation of the genetic algorithm[5]. At each iteration, we have used a GA to generate a set of scaling configurations, to be evaluated in the next iteration, based on the current set of solution candidates. After this generation, the current candidate solutions are merged with the set of all solution candidates. The algorithm

[5]Genetic Algorithm - MATLAB & Simulink - [`www.mathworks.com/discovery/genetic-algorithm.html`]

continues until it reaches a pre-defined time limit. Finally, this set of the solution candidates are forwarded to the scaling planner.

Since we are running a GA at runtime with time constraints, it is possible that the scaling strategy provided by the algorithm can be further improved. This can be both from the perspective of revenue maximization and CPU share minimization. To address this issue, the planner uses two quick fixes. Firstly, to minimize CPU shares, it checks whether a microservice was allocated less CPU share in the previous monitoring window. If this is true, it replaces the current configuration for that microservice with the previous one. Now, if the change does not affect the TPS significantly, the previous configuration is chosen for that microservice, otherwise the current configuration is kept. Secondly, to increase the TPS, ATOM reduces the number of replicas while increasing the CPU share of each replica, keeping the total CPU share same. It then checks again whether the TPS is affected significantly and if not, it keeps the modified configuration. This improves the TPS since reducing the number of replicas also reduces the parallelization overhead. Before sending the scaling strategy, the planner also checks for a drastic change in the total CPU share, as mentioned in section 3.4.1, if that option is chosen. After that, the planner creates the scaling configurations from the best solution candidate. These configurations are passed to the scaling executor. It then triggers the necessary system commands to scale the system.

3.5 Evaluation

In this section we present the evaluation of ATOM. We initially provide the details of our experimental setup and the baseline autoscalers that we have developed to compare with ATOM. After that, the performance comparison is presented according to different elasticity and performance metrics.

3.5.1 Experimental Setup

For ATOM's performance evaluation, we have deployed the Sock Shop application in Docker swarm mode in two servers. The details of the servers are provided in Table 3.7. We reduced

Table 3.7: Docker swarm server config. for performance evaluation

#	Deployed Microservices	Online CPU	CPU Freq.	Memory
1	Router, Front-end, Carts Db.	4	0.8 GHz.	64 GB
2	Catalog Svc., Carts Svc., Catalog Db.		1.2 GHz.	16 GB

the CPU frequency in those servers to create high resource contention. The CPU frequency that has been presented in Table 3.7 is the reduced CPU frequency. We have developed two rule based autoscalers to compare their performance with ATOM. Such autoscalers are common in industry [QCB18] for example Amazon Web Services (AWS)[6] and Kubernetes[7].

Both autoscalers monitor the CPU utilization of the microservices. If the CPU utilization level reaches the current limit, the autoscalers double the amount of CPU share currently allocated to the microservices. The first scaler (UH) allocates the resource horizontally and the second one (UV) vertically. For example, if the CPU utilization reaches 35%, a value near to the limit of 40%, UH creates 2 replicas each having 0.4 CPU share and UV increases the CPU share of the replica to 0.8. UH operates only on the stateless microservices whereas UV operates on both stateful and stateless microservices. Thus, for UH scenarios we have allocated a full CPU core for each of the stateful microservices.

Here we have configured the Sock Shop application to handle 500 concurrent users in a browsing workload. After this initial setting, we have evaluated UH, UV and ATOM considering an increase in workload in multiple combinations of request mix and concurrent number of users (N). The request mix is similar to the suggested mix in TPC-W benchmark [Men02]. The values of N are chosen to create scenarios ranging from moderate to heavy increase in the current workload. The details of the workload is provided in Table 3.8.

Before comparing ATOM with the baseline scalers, we have compared different variations of ATOM. The variations are based on ATOM's likelihood to change the CPU shares at each monitoring window. As mentioned in section 3.4.1, this has been controlled by the planner by

[6]Amazon EC2 Auto Scaling - [aws.amazon.com/ec2/autoscaling]

[7]Horizontal Pod Autoscaler - [kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale]

Table 3.8: Workloads considered in evaluating the autoscalers

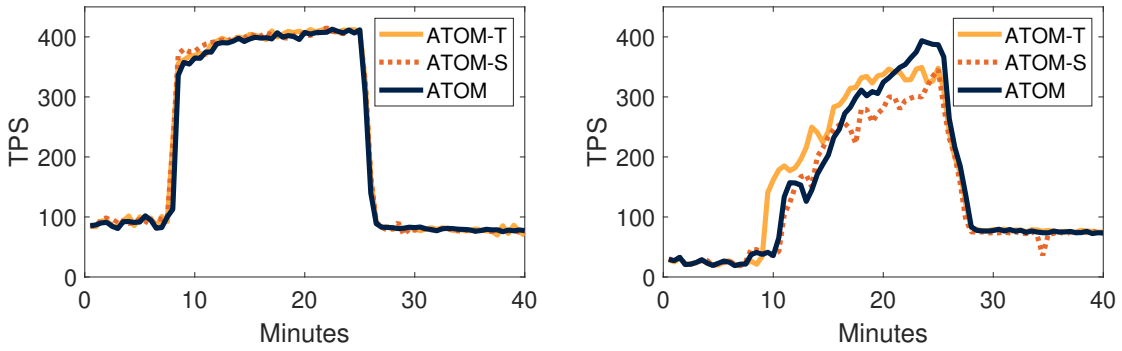
Name	Request Distribution			Concurrent Users	Think Time
	Home	Catalog	Carts		
Browsing Mix	63%	32%	5%	1000, 2000, 3000	7 sec
Shopping Mix	54%	26%	20%	1000, 2000, 3000	
Ordering Mix	33%	17%	50%	1000, 2000, 3000	

being conservative about changing the system configuration on each monitoring window. It provides two options for achieving this: not allowing a large change in CPU share or discarding the current configuration if the potential improvement in TPS is not significant. We call these versions a conservative version of ATOM. The conservative version that changes CPU shares by looking potential improvement in TPS is named ATOM-T and the one that allows only a small change to the total allocated CPU capacity is named ATOM-S.

For performance comparisons, we have chosen two workloads from the light browsing mix and heavy ordering mix with $N = 3000$. It is seen from Figure 3.8 that all the versions of ATOM produce nearly identical improvement in TPS for the browsing workload. In case of ordering mix, ATOM-T yields slightly higher average TPS whereas ATOM-S yields slightly less TPS than ATOM. Although the TPS for ATOM increases slowly at the beginning, later it increases sharply and achieves a higher value than ATOM-T. For ATOM, such sharp increases can occur, possibly with a sharp and short decrease period before, whereas for ATOM-T and ATOM-S the improvement is more steady. An issue for the conservative versions is that it can be difficult to determine the threshold value that gradually improves the scaling strategy rather than completely stopping the improvement. This is particularly detrimental in the case where the system experiences an increase in workload for a long period. Thus, for the latter experiments we have used ATOM.

3.5.2 Results

For all the workloads in Table 3.8, we have run the experiments for 40 minutes where the first 25 minutes represent the increase in workload. The effect on the TPS by ATOM and the baseline autoscalers is presented in Figure 3.9. From the figure, it is seen that there is a

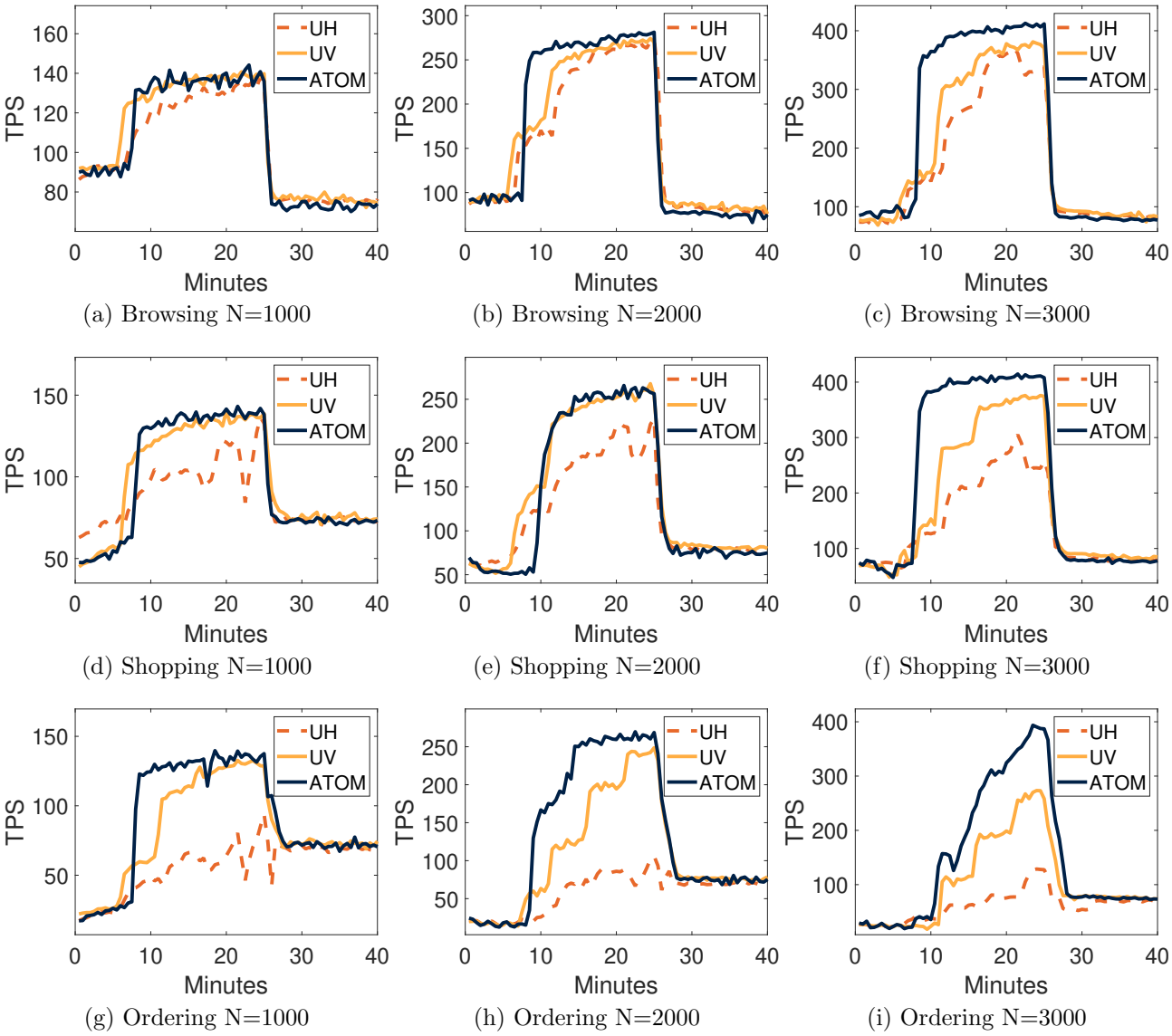


(a) Browsing N=3000 (b) Ordering N=3000

Figure 3.8: Performance comparison of ATOM with its conservative versions: ATOM-S and ATOM-T

delay in the scaling action for ATOM. In these experiments, the monitoring window has been set to 5 minutes for all the scalers. In addition to this delay, ATOM has on average a 2.5 minute delay for its optimization and planning, whereas the other autoscalers initiate scaling actions immediately after the monitoring window. However, from the figure it is seen that the delay does not effect the performance of ATOM significantly compare to others. This is because, although there is a step by step improvement in TPS for the baseline autoscalers, the TPS improves significantly in case of ATOM in the first step. This is expected as ATOM monitors the workload and then provides an optimal scaling configuration for that workload. On the other hand, the baseline scalers monitor the current utilization and cannot estimate the maximum amount of required CPU share from this information only. Thus, they improve the TPS in a step by step basis.

For a quantitative evaluation, we have used the following metrics: total under-provisioned time (T_u) [HKR13, DBC18], total under-provisioned area (A_u) [HKR13, DBC18] and TPS. The metrics indicate how an autoscaler performs when there is a increase in the workload. In the context of microservices, T_u can be defined as $\sum_i T_u^{(i)}$, where $T_u^{(i)}$ is the total time for a microservice i in an under-provisioned state. Similarly, A_u can be defined as $\sum_i A_u^{(i)}$ where $A_u^{(i)}$ is the total under-provisioned area for an microservice i. The under-provisioned area represents the extent of under provisioning for a time period t. It is defined as the product of an under-provisioned period and the difference of required and allocated CPU capacity for a microservice in that period. TPS is defined as the total number of transactions completed per second. We



(a) Browsing N=1000 (b) Browsing N=2000 (c) Browsing N=3000

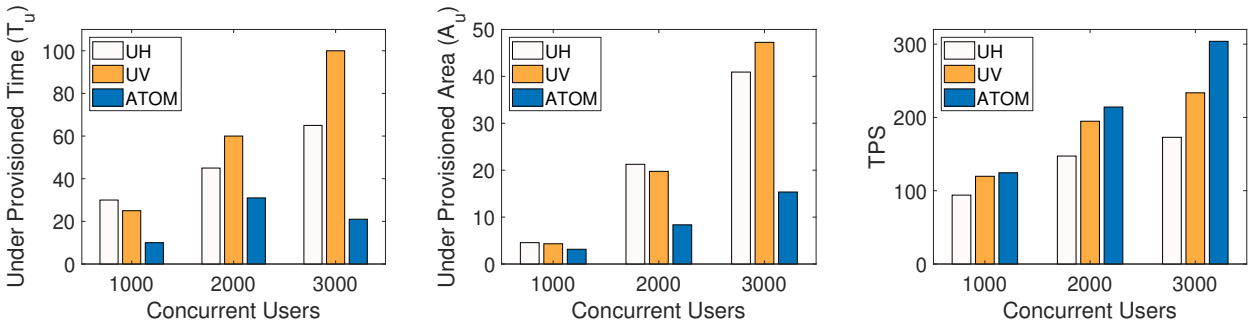
(d) Shopping N=1000 (e) Shopping N=2000 (f) Shopping N=3000

(g) Ordering N=1000 (h) Ordering N=2000 (i) Ordering N=3000

Figure 3.9: The effect over time on the TPS by ATOM and baseline autoscalers under different workload

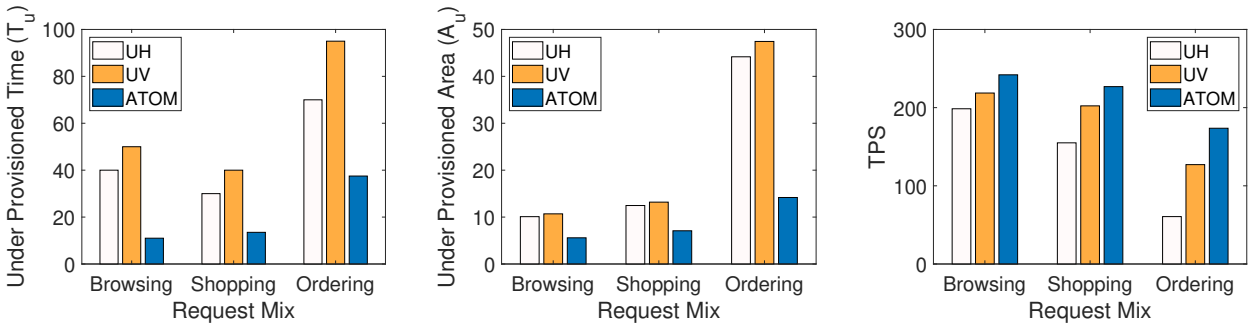
have considered the TPS during the increased load period since the TPS is affected in that period and the autoscalers aim to resolve this issue. For T_u and A_u a lower value is expected, which is the opposite case for TPS.

In Figure 3.10 we compared the performance of the scalers, according to the considered metrics, with increase in concurrent number of users (N). The results are considering 3 microservices since UH does not scale the router and 2 database services. The results show that ATOM outperforms UH and UV considering all the metrics, particularly for higher values of N. It is seen that with the increase of N, T_u and A_u increases for all the scalers though lower is



(a) T_u - lower is better (b) A_u - lower is better (c) TPS - higher is better

Figure 3.10: Comparison of the autoscalers for different elasticity and performance metrics with the increase of concurrent users

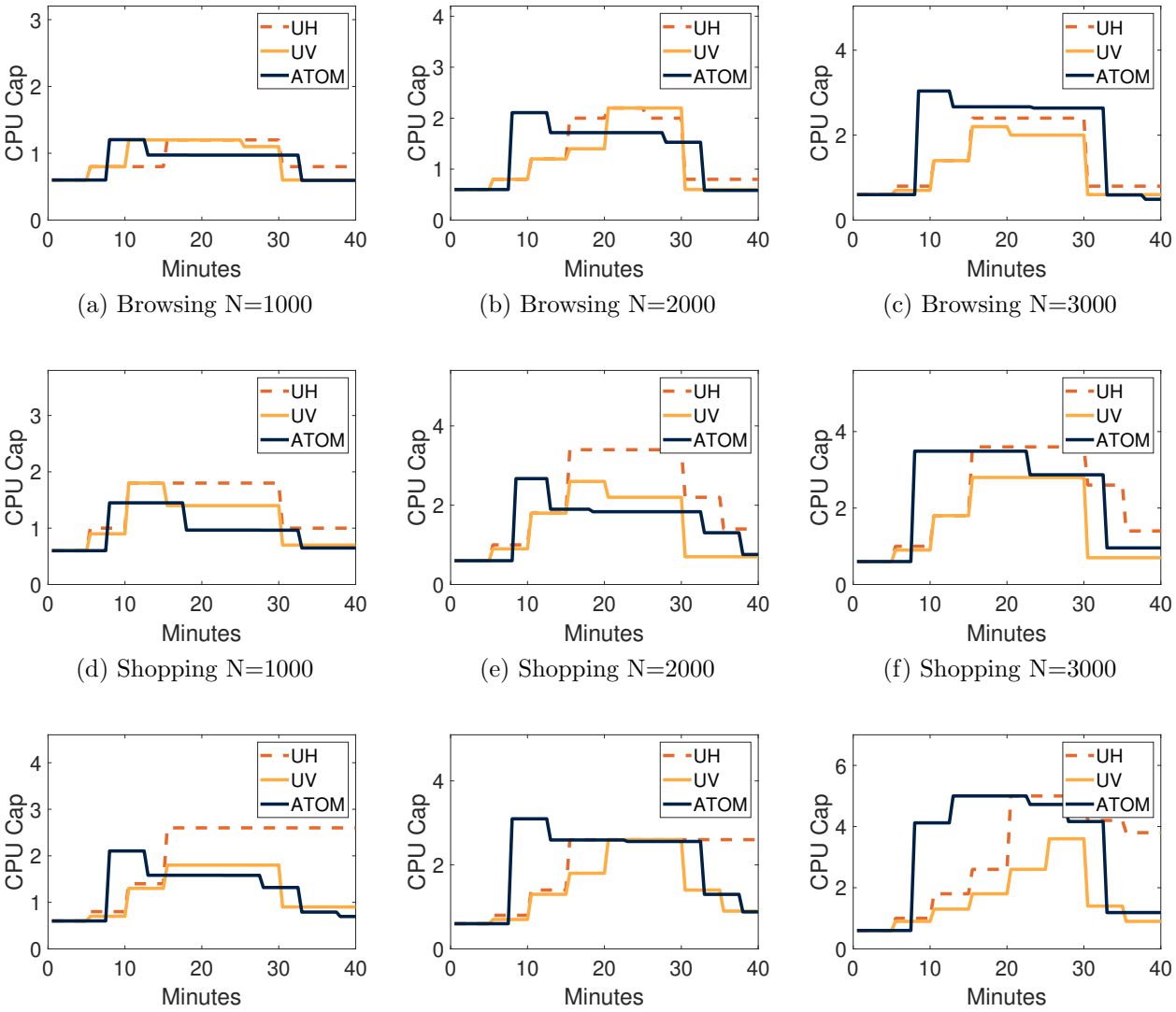


(a) T_u - lower is better (b) A_u - lower is better (c) TPS - higher is better

Figure 3.11: Comparison of the autoscalers for different elasticity and performance metrics with change in request mix

better in this case. However, this increase is expected as with the increase in N, the initial gap between required and allocated CPU capacity also increases. Among these values, the values for ATOM is the lowest because it can reduce the CPU capacity gap significantly after the first monitoring window. This helps ATOM in improving the TPS than the other scalers. Considering $N = 3000$, the improvement in TPS is 30% higher for ATOM than the next best approach UV.

We also present the results from the perspective of change in request mix in Figure 3.11. In the figure, we can see that, in terms of TPS, ATOM by far outperforms both UH and UV in ordering mix, while performing similar in browsing and shopping mix. For the ordering mix, the TPS is 37% higher than UV, which is the second best approach. From Figure 3.11a, we see that ATOM has a very low T_u comparing to UH and UV in all the mixes. However, as seen in Figure 3.11b, since the difference in A_u is not as high as it is in T_u for browsing and shopping



(a) Browsing N=1000 (b) Browsing N=2000 (c) Browsing N=3000

(d) Shopping N=1000 (e) Shopping N=2000 (f) Shopping N=3000

(g) Ordering N=1000 (h) Ordering N=2000 (i) Ordering N=3000

Figure 3.12: The over time CPU capacity allocated by ATOM and baseline autoscalers under different workload

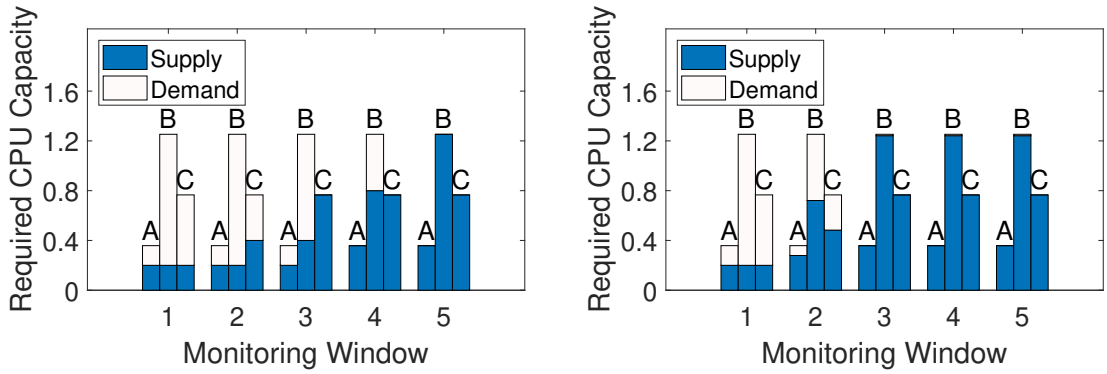
mix, the TPS is nearly similar for all the scalers in browsing and shopping mix.

We now focus on the CPU capacity allocated by the scalers to achieve the results presented from Fig. 3.9 to 3.11. The capacity allocated by the scalers over time is presented in Fig. 3.12. As before, these figures are considering the three microservices that all of the scalers consider. From the figure, we see that the all the scalers increases the CPU capacity after determining the increase in workload by receiving the data from first monitoring window *i.e*, after 5 minutes. They also reduces the CPU capacity when determining the decrease in workload after 30 minutes. We also see that the maximum capacity allocated by the scalers are similar

but ATOM allocates that capacity sooner than UH and UV. This confirms the reason behind ATOM achieving higher TPS than UH and UV. We also see that, not only ATOM deallocates the CPU capacity when the workload decreases, it also deallocates when the workload remains in an increased level. This is because since ATOM runs the optimization program for a short period, it may not find the optimal solution in one run. However, as it periodically receives the data and runs the optimization program, it finds scaling strategies that yield similar TPS with less capacity *i.e.,* a better strategy. Thus, it may also deallocate CPU capacity while the workload is still in an increased level, which is evident from the figure.

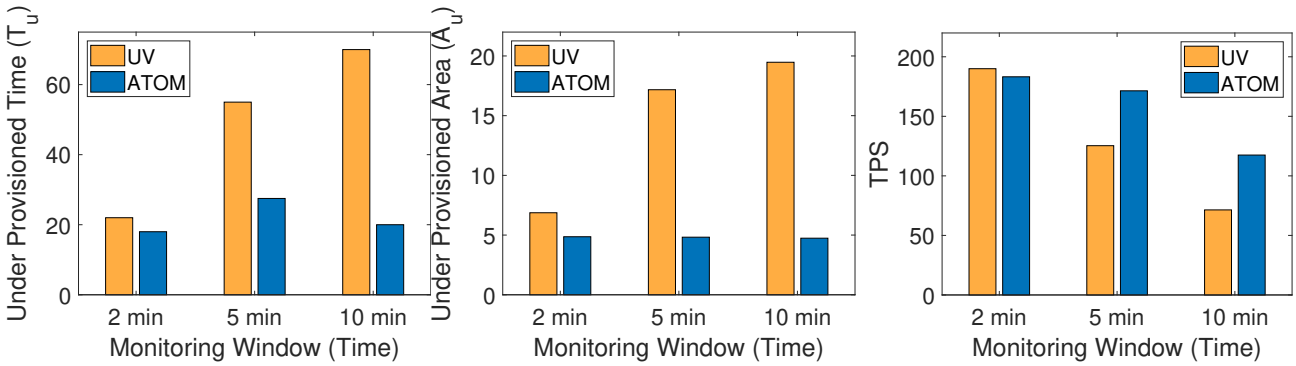
Besides being able to reason on horizontal and vertical scaling choices, another reason for ATOM's better performance is that it addresses bottleneck shifts better than UH and UV. A bottleneck shifting can cause a delay in reducing the difference between required and allocated CPU capacity for a microservice. We present one such scenario in Figure 3.13 for the ordering mix with $N = 2000$, where a bottleneck shifting scenario occurs among 3 microservices - router, front-end and cart service. For UV, from Figure 3.13a, it is seen that the router remains in starvation up to window 3 as the front-end does not receive an additional capacity until window 3. The front-end remains in starvation up to window 4 as the cart service's requirement is fulfilled in window 3 and from there the front-end's large capacity gap cannot be reduced solely in window 4. On the other hand, ATOM starts to remove all the bottlenecks at once in window 2 since it has obtained the whole picture of the system's capacity requirement after window 1. The bottlenecks are partially resolved in window 2 due to ATOM's delay in optimization, affecting the average CPU allocation in that window. Continuing from window 3, the bottlenecks are fully resolved by ATOM.

To observe the effect of a monitoring window over the scalers' performance, we have considered 3 different windows and evaluated them using an ordering mix with $N = 2000$. Since UH is outperformed by UV, in these experiments, we have only considered UV. From the results, as presented in Figure 3.14, it is seen that T_u and A_u remains almost same for ATOM in all cases. This is because those values for ATOM are dominated by its optimization time, which has been bounded by a fixed value of 2 minute in these experiments. The results show that ATOM outperforms UV for the 5 and 10 minute window. For the 2 minute window, they



(a) UV (b) ATOM

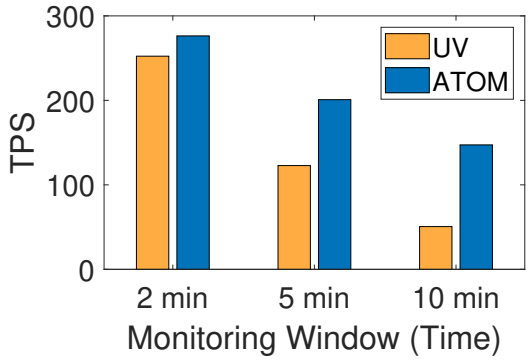
Figure 3.13: The demand vs. supply in CPU capacity for 3 microservices, involved in bottleneck shifting, after each scaling action by UV and ATOM. The microservices are: A - Router, B - Front-end and C - Cart Svc.



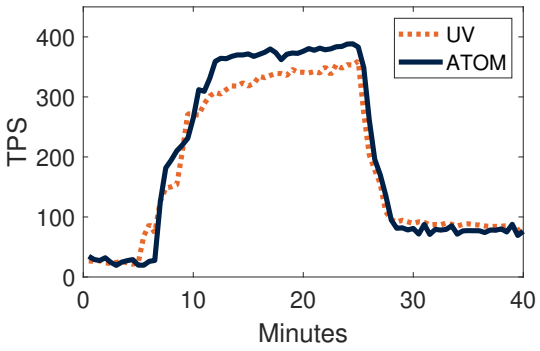
(a) T_u - lower is better (b) A_u - lower is better (c) TPS - higher is better

Figure 3.14: Performance comparison of UV and ATOM with change in the monitoring window size

perform similarly but the numbers from UV can be considered slightly better. However, such a performance for a short monitoring window is not guaranteed for two reasons. Firstly, for a large number of bottleneck shifts, which is likely for microservices, even with a short window that allows quick changes in system configuration, the time for resolving those bottlenecks can be significantly high. Secondly, such quick shifts in system configuration can affect the system steady state if new replicas are created or a microservice needs to be restarted. This case is not applicable to UV as it only scales vertically. However, vertical scaling is not an effective option in all scaling scenarios [GDK+18], which is also evident from our experiments. When we consider a similar experiment with $N = 3000$, from Fig. 3.15a we see that ATOM produces better results than UV for all monitoring windows. The reason it outperformed UV for the 2 minute window is the code of the microservices are not internally parallel and thus it can

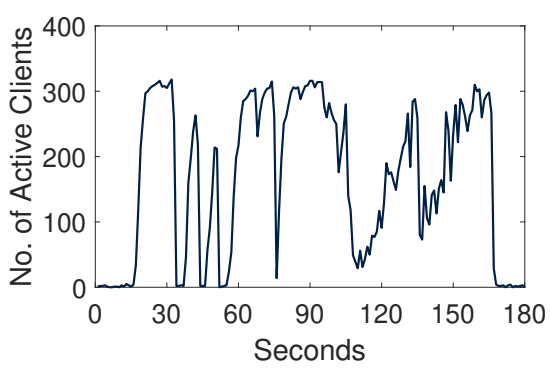


(a) TPS - higher is better

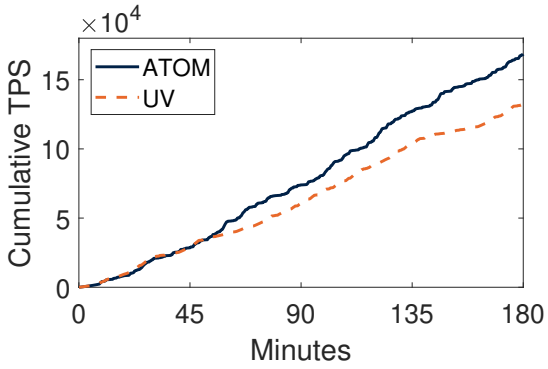


(b) TPS (2 min monitoring window)

Figure 3.15: TPS yielded by UV and ATOM with $N = 3000$



(a) Arriving Clients ($I = 4000$)



(b) Cumulative TPS

Figure 3.16: TPS yielded by the scalers with high burstiness in workload

not utilize more than 1 CPU core. We can see from Fig. 3.15b that the initial increase in TPS is similar for both ATOM and UV. However, once UV allocated a full CPU core, its further actions have no impact on the TPS. ATOM on the other hand, applied both vertical and horizontal scaling and yielded better results.

It is important for an autoscaler to work with workloads having burstiness. Burstiness can occur in the workload for various reasons such as, start of office hours, reducing product prices for sale, new product release, etc. We have created such workloads by injecting burstiness in the ordering mix with $N = 500$ using the index of dispersion I [MCCS09]. We have used two values, $I = 400$ and $I = 4000$, representing moderate and high burstiness. We have evaluated the performance of UV and ATOM with these two workloads. From the results, we have not observed any significant difference for moderate burstiness. However, for high burstiness, ATOM has outperformed UV. We have presented the results in Figure 3.16. It is seen that

when using ATOM as an autoscaler, the traffic surges are also reflected in the system TPS, which is not exactly the case of UV. This also results in a better cumulative TPS of the system yielded by ATOM, which is 28% higher than UV. The reason for UV being outperformed is that since the CPU capacity is bounded, the utilization values do not reflect the traffic surges. In addition, the low load periods affect the average utilization. Thus, UV does not get an appropriate picture of the workload intensity and cannot scale accordingly.

3.6 Summary

In this paper we have presented ATOM, a model-driven autoscaler tailored to microservices application. Leveraging the ease of scaling and replicating microservices in Docker-based deployments, we have first demonstrated that if a services can be both vertically and horizontally scaled the workload characteristics need to be taken into account to decide which one of the two scaling actions is better. We have then proposed an autoscaler that leverages layered queueing network models to scale application capacity in the presence of time-varying workloads, including those affected by burstiness. Results indicate that ATOM can significantly improve over scalers based on horizontal or vertical scaling actions only.

Chapter 4

Cold Start Aware Capacity Planning

4.1 Introduction

Function-as-a-Service (FaaS) platforms, based on the serverless execution model [WLZ+18], allow to deploy the codes as individual functions without having to manage the underlying infrastructure. This facilitates DevOps practices [JbAPT18] by providing more flexibility to each development team and increasing the pace of delivery of code updates. The availability of open source platforms, like OpenFaaS and OpenLambda, has made it possible to install an on-premises FaaS platform, which calls for dedicated sizing and resource allocation methods in order to meet service-level agreements (SLAs).

FaaS platforms are designed to implement event-driven applications, which react to a change of state as a result of events generated by the environment and execute associated business logic. In a FaaS platform, this logic is termed as a function, which is usually packaged as a container. To reduce resource wastage, FaaS containers are offloaded from the memory given that they remain idle for specific time period. When a new request for an offloaded function arrives, the request is blocked until the function is loaded again. This issue is known as the *cold start* issue [WLZ+18, LRC+18].

During the capacity planning process, the cold start issue can pose a significant trade-off be-

tween latency and memory allocation optimization. A cold start occurs when a function is invoked while the corresponding container is not yet loaded in memory, which adds a delay in sending the response needed to spin-up the container and the function runtime dependencies. Despite hurting performance, this mechanism aims at reducing memory consumption by offloading functions that are idle for a sufficiently long time. This poses a trade-off between response time SLAs and available memory to support concurrent execution of more functions, which needs to be considered upon sizing an on-premises installation.

To address this issue, we can draw parallels between a FaaS platform and a Time to Live (TTL) cache. Similar to FaaS, a TTL caching system periodically offloads its cached objects. Due to this, the cold start issue resembles the object hit rate improvement problem in a TTL cache, in which we determine the optimal time to keep objects in cache [BSG+18]. Thus, analysis methods from TTL cache research, such as characteristic time approximation [CTW02], may be in principle also applicable to FaaS sizing for estimating the required memory capacity. However, from our study we have identified two limitations of such an approach. First, contrary to TTL cache misses, the latency incurred by cold starts of different functions can vary widely [Shi19]. Next, while a large fraction of TTL research considers fixed-sized objects, a function consumes different amount of memory depending on whether it is idle or in execution.

In this paper, we present COCOA (COld start aware Capacity planner fOr function-As-a-service), a sizing method that leverages a stochastic modeling approach based on layered queueing networks (LQN) [FAOW+08] and M/G/1-type queueing systems for capacity prediction. To consider the effect of cold starts, we have incorporated the probability of experiencing cold starts, by each function, with the LQN model. These probabilities are estimated from an M/M/1/setup/delayedoff model - a variant of M/M/k setup class of models [GDHBSW13] - which we solve using matrix-analytic methods as a special case of M/G/1-type system. Setup models can approximate the cold start probability for a function, taking into account the cold starts. To predict the required capacity, COCOA follows an iterative process. It repeatedly solves the LQN model to find a set of function idle times and a CPU configuration such that the function response times are just below the SLA. To accelerate the searching process, we have designed a parallel algorithm, where each parallel branch utilizes binary search. Once the

idle times and CPU configuration are obtained, COCOA estimates the CPU utilization value for each function. These estimations are integrated with a capacity estimation method for TTL cache [FRR12] to predict the required memory capacity.

Overall, we summarize our contributions as follows:

- We investigate in Section 4.2 the similarity between a FaaS platform and TTL cache from the cold start perspective and illustrate that TTL cache analysis, despite promising, is alone insufficient for FaaS capacity estimation.

- We present in Section 4.3 an LQN-based performance modeling technique for FaaS platforms that captures the effect of cold starts over function response times, correcting the limitations of TTL cache analysis when applied to this setting.

- In Section 4.4 we propose COCOA, a sizing method for on-premises FaaS platforms leveraging our LQN model and demonstrate its effectiveness in reducing resource over-provisioning while ensuring response time SLAs.

Lastly, in Section 4.5 we validate our framework against data from simulation. We present our concluding remarks in Section 4.6.

4.2 Similarity between FaaS and TTL caches

In this section, we present the analogy between a FaaS platform and a TTL cache. We also demonstrate that, to ensure the SLA on response time, an "off-the-shelf" availability-aware resource provisioning approach, used in a TTL cache, can overestimate the required capacity for a FaaS platform. Subsequently, we present the observations that indicate how this can be resolved with a performance model.

4.2.1 Analogy

TTL caches in Content Delivery Networks (CDNs) facilitate faster page loading and reduce the load at the origin server. In such caching systems, each cache object is associated with a TTL value. Once an object residing in the cache reaches its TTL value, it is evicted [BSG+18]. If the cache can serve the request for an object, it is termed as a cache hit. The fraction of requests served, for a particular period, is called the hit rate. Longer TTL values can improve the hit rates but are more costly as they require a larger cache size.

Similar to TTL caches, to reduce the number of cold starts, a possible solution, from the point of view of the end user, is to keep the functions in memory for longer periods. Software developers, often enforce this by using a ping hack[1]. However, such a solution is not practical from the server-side perspective. This is because such a technique will keep the functions loaded in memory for an indefinite time, which significantly increases the required memory capacity. A way around to this problem is to determine an optimal idle time that ensures a certain degree of availability and reduce the number of cold starts to an acceptable limit. We notice the analogy of this problem with the configuration of TTL caches [BSG+18]. In such systems, the goal is to determine a characteristic time [CTW02], which is set as the TTL value, that maximizes the cache hit rates with a space constraint. The hit rate h_i for an object i with a request arrival rate λ_i is defined by (4.1), considering its TTL (T_i) value is reset upon a new request [DMT+19].

$$h_i = 1 - e^{-\lambda_i T_i} \tag{4.1}$$

This TTL value is similar to the idle time of the functions. It represents a period for which a object is kept in the cache, even though no new request is received, which ensures a certain degree of availability. Similarly for FaaS, an idle time represent a period where the functions are kept loaded in memory even though they are idle. Therefore, we can use this concept of TTL to estimate the idle times for the functions. To realize this, we can simply solve (4.1)

[1] A tool that periodically sends a request to the functions to keep it in the memory, like Lambda Warmer - `github.com/jeremydaly/lambda-warmer`

to get a TTL value for a particular hit rate. This value can be set as the idle time of the function. It will ensure the needed degree of availability and reduce the number of cold starts. Consequently, this will help to satisfy the response time constraints.

4.2.2 Example

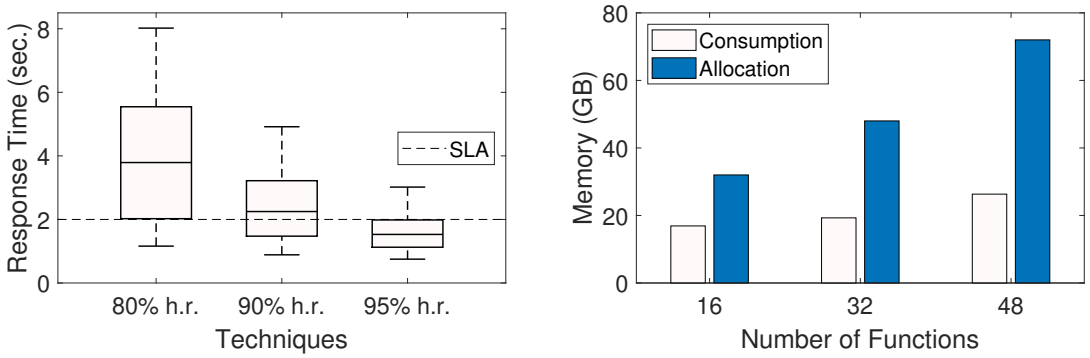
To illustrate the concept, we have developed a discrete-event simulator for a FaaS platform, referred to in the rest of the chapter as *FaasSim*[2]. Developing *FaasSim* was necessary since popular performance modeling tools, e.g., JMT [BCS09], cannot model the cold start scenario we need to consider for FaaS. In FaasSim, we followed the microservices pattern [Hef16] for the functions, where each function does exactly one job, to ensure complete separation of concerns. The functions are considered running on a multi-core CPU which follows a processor sharing scheduling discipline.

In the simulation, we have considered an open workload model where the requests arrive following a Poisson process. To introduce popularity among the functions, meaning their invocation probabilities will be different, we have used the Zipf distribution [Gla94]. The function service times are set such that they are at-most half of the SLA value of 2 seconds, when there is no resource contention. The cold start times are chosen from a recent study on popular FaaS platforms [Shi19] that, apart from the platform, also considered factors like programming languages and deployment sizes, which can significantly effect the magnitude of cold starts. Although there are similar studies [WLZ+18, MEHW18], the work by Mikhail [Shi19] is more comprehensive covering at least three cloud vendors, programming languages and deployment sizes. We have considered the cold starts corresponding to realistic deployment sizes[3].

We have run the simulation in three settings with 16, 32 and 48 functions and observed the effect of different hit rates over the function response times. The function idle times are set by solving (4.1) for the specific hit rate. This hit rate have also been used to estimate the memory capacity. The hit rate is related to the average runtime memory consumption (m) as

[2]The simulator is available at - `https://github.com/alimulgias/FaasSim`

[3]The cold start times with a very small deployment size like 1 KB is ignored since it is relatively small for a function in production.



(a) Hit rates and response time

(b) Maximum memory consumption

Figure 4.1: Evaluation of an availability-aware approach in estimating the capacity of a FaaS platform and ensuring the SLA for response time

$m = \sum_i h_i \theta_i$, where θ_i is the memory requirement of each function [FRR12]. We have used this value of m as an estimation of the required memory capacity, which is allocated for all the functions, and compared it with the actual memory consumption value obtained from *FaasSim*. The findings from the simulation are presented in Fig. 4.1.

4.2.3 Observations

In Fig. 4.1a, we plot the response times of each of the 48 functions for different hit rates. We see that even with 95% hit rate, there are response times that violate the SLA. However, for 95% hit rate, more than half of the function response times are much lower than the SLA. This indicates that all the functions do not require the same hit rate to ensure the SLA. In Fig. 4.1b, we present a comparison between the estimated memory capacity and maximum consumption for 95% hit rate. Although the capacity notably increases with the number of functions, the consumption is less sensitive to it. This is because memory consumption is primarily dependent on the workload parameters. In addition, the consumption is not very high since most of the functions remain idle while resident in memory, which is not considered during the estimation.

From these observations, it is clear that an availability-aware approach is not adequate for optimal capacity estimation that ensures the SLA for response time. Such an approach only considers the volume of cold starts, whereas we also need to consider its effect on the response time. For a particular workload, firstly, we should know the cold start probabilities of the func-

tions for different idle times. Subsequently, depending on these probabilities and the severity of cold starts, we need to approximate the function response times. Thus, we need a performance model incorporating all these factors. The model will also help in fine-grained capacity estimation by providing the resource utilization estimates. In the following section we present our performance model.

4.3 Modeling Cold Starts in FaaS

COCOA requires a model to analyze the effect of hit rates over response times considering the severity of colds starts. This section initially demonstrates how to model these cold starts and estimate their probabilities for different workloads. After that, the response time prediction process, using an LQN, and its validation are also presented.

4.3.1 Estimating Cold Start Probabilities

Unlike commercial FaaS platforms, open source platforms, like OpenFaaS, allow concurrent function execution in same container [ACR+18]. We focus on this function concurrency approach. We propose to consider, from a modeling standpoint, the function as a server of a queueing model, representing the admission control buffer to the function, and the cold start delay as the initial setup time of the server before beginning service. The functions also have an idle time which is equivalent to the idle server waiting time before it is shut down. Considering these similarities, a cold start may be modeled as a M/M/1/setup/delayedoff model, which is a variant of M/M/k/setup class of models [GDHBSW13]. The M/M/k/setup models consider a setup cost, usually in the form of a time delay, when turning the server on. Its "delayedoff" variant considers an idle time before turning the server off.

Although Gandhi *et al.* [GDHBSW13] provided the exact solution for an M/M/k/setup/delayedoff model, this applies when the number of servers is $k \geq 2$. However, in our case we need to model each function separately. Thus, we have a function representing a single server, which can be

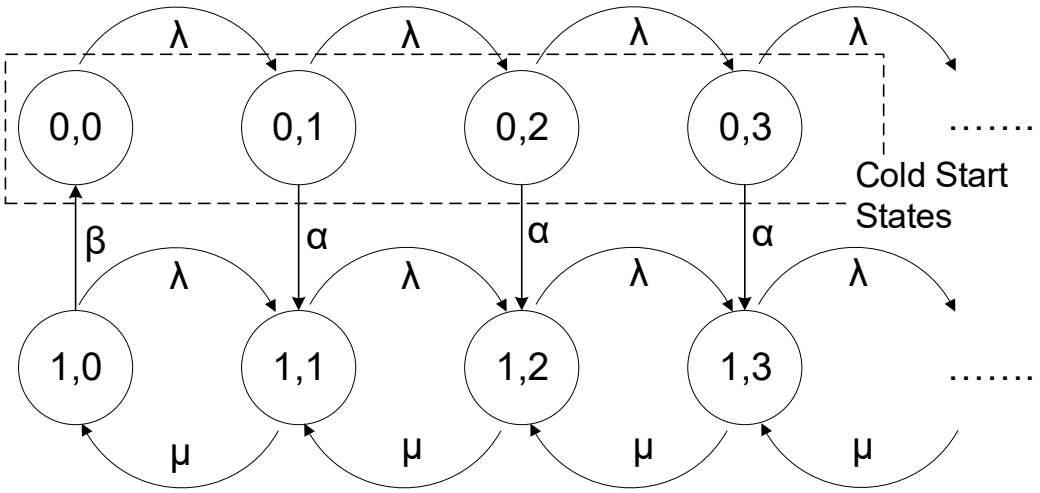


Figure 4.2: The M/M/1/setup/delayedoff model for a function

either on or off. To get different performance indices for such a model, we may directly solve its underlying Continuous Time Markov Chain (CTMC).

The CTMC transitions are presented in Fig. 4.2. Each CTMC state (i, j) has two parameters: i tracks whether the function resides in memory or not ($i \in \{0, 1\}$), while j tracks the number of jobs ($j \in \mathbb{Z}^*$) in the admission queue to enter service in the function. A transition from (i, n) to $(i, n+1)$ occurs with rate λ, transition from $(i, n+1)$ to (i, n) occurs with rate μ, and transition from $(0, j)$ to $(1, j)$ occurs with rate α. These rates describe the mean inter-arrival time ($\frac{1}{\lambda}$), mean service time ($\frac{1}{\mu}$) and mean cold start time ($\frac{1}{\alpha}$) respectively. There is a special transition from $(1, 0)$ to the initial state $(0, 0)$ with rate β, which describes the function idle time ($\frac{1}{\beta}$). In a CTMC, all holding times are considered to be exponentially distributed. However, in a real system the idle time of a function is set to a deterministic value. To address this issue, we can use the method of phases and make this transition Erlang-k distributed with rate $k\beta$. To realize this, we introduce $k - 1$ extra states between $(1, 0)$ and $(0, 0)$. The transitions between all these states occur with a rate $k\beta$. This keeps the mean identical to the original exponential, $\frac{1}{\beta}$, but reduces the variance by k times. Thus, for large enough k, the transition will display a behavior close to deterministic.

The effects of cold starts vary depending on the sequence of request arrivals. If a request arrives when the function is being loaded into memory due to a recent request, its response time will be affected to some extent. The severity of the queueing overhead will depend on the residual

cold start time of the previous request. However, this does not need to be modeled explicitly thanks to the memoryless property of the exponential distribution. Considering this, as shown in Fig. 4.2, it is clear that the cold start states are $(0, j), \forall j$. We can calculate the cold start probability of the functions from the stationary distribution (π) of their CTMC. We indicate with $\pi_{i,j}$ the probability of state (i, j), then the cold start probability is defined as $\sum_j \pi_{0,j}$. We can get the stationary distribution by solving the CTMC. This can be done efficiently using the matrix-analytic method, since the CTMC sparsity structure makes it equivalent to an M/G/1-type process [LR99]. The latter is analyzed using the MAMSolver [RS02, RS07]. The MAMSolver considers an M/G/1-type process represented by an infinitesimal generator Q, which has the following structure.

$$Q = \begin{bmatrix} \hat{L} & \hat{F}^{(1)} & \hat{F}^{(2)} & \hat{F}^{(3)} & \hat{F}^{(4)} & \dots \\ \hat{B} & L & F^{(1)} & F^{(2)} & F^{(3)} & \dots \\ 0 & B & L & F^{(1)} & F^{(2)} & \dots \\ 0 & 0 & B & L & F^{(1)} & \dots \\ 0 & 0 & 0 & B & L & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Here, the sub-matrices B, F and L represents the backward, forward and local transitions respectively. A backward transition represents going a level down i.e., the number of job reduces by one. A forward transition, on the other hand, represents going a level up i.e., the number of job increases by one. Finally, a local transition represents that the level does not change but the environmental state (e.g. moving from cold states to warm states) changes. The symbol "ˆ" is used to represent the boundary part of the process. This can be different from the other part depending on system dynamics. For example, in Fig. 4.2, considering the transition between state $(0, 0)$ and $(1, 0)$ is Erlang-k, there will be $(k - 1)$ additional states in this level compared to others. Thus all the transitions from this level will be different to the others. For all the other levels, the B, F and L matrices will be similar, which means they have similar backward, forward and local transitions.

As mentioned earlier, in the CTMC presented in Fig. 4.2, we need to consider the transition between state $(0,0)$ and $(1,0)$ as Erlang-k to model the function idle time as approximately deterministic. This makes the dimensions of the boundary sub-matrices $\hat{B}$, $\hat{F}^{(1)}$ and $\hat{L}$ depended on the value of k. Their respective dimensions are $2 \times (k+1)$, $(k+1) \times 2$ and $(k+1) \times (k+1)$. The sub-matrices are provided below.

$$\hat{B} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & \mu \end{bmatrix} \quad \hat{F}^{(1)} = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \\ 0 & \lambda \\ \vdots & \vdots \\ 0 & \lambda \end{bmatrix} \quad \hat{L} = \begin{bmatrix} -\lambda & 0 & 0 & \dots & 0 \\ k\beta & -\lambda - k\beta & 0 & \dots & 0 \\ 0 & k\beta & -\lambda - k\beta & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & k\beta & -\lambda - k\beta \end{bmatrix}$$

The dimension for all the regular transition matrices i.e., B, F and L is 2×2. They are as follows.

$$B = \begin{bmatrix} 0 & 0 \\ 0 & \mu \end{bmatrix} \quad F^{(1)} = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \quad L = \begin{bmatrix} -\lambda - \alpha & \alpha \\ 0 & -\lambda - \mu \end{bmatrix}$$

Using the matrices above to define the infinitesimal generator Q, the cold start probabilities can be obtained using the state probabilities returned from the MAMSolver.

4.3.2 Predicting Response Time

To model the performance of FaaS platforms in private clouds, we need to understand how they differ from public clouds in managing its memory and CPU capacity. In public clouds, a function is packaged within a container or sandbox and deployed in a VM. The CPU capacity allocated to the functions is usually proportional to the memory requirement defined by the client [WLZ+18], as is the case at present in commercial offerings such as AWS Lambda and Google Cloud Functions. In private clouds, a function is packaged in a container that runs on

top of a container platform like Docker or Kubernetes. This FaaS ecosystem is orchestrated using an open source framework like Apache OpenWhisk or OpenFaaS. The resource allocation can either be on a per-container basis or it can also be allocated for the entire platform.

The latter strategy for resource allocation in private cloud does not require to handle the CPU requirement of each container. In addition, considering that all the functions do not have simultaneous service requests, this strategy allows a container to take up more CPU resources, if available, rather than restricting it. This makes it more reasonable in a private cloud setup. Thus, for our performance model, we consider that the CPU capacity is allocated following this strategy. This means that the function will contend for the allocated CPU capacity to execute their jobs. For each function, there will be two types of jobs - regular tasks when the function is warm and resuming-the-service tasks when the function is cold. Both the cold and warm jobs will run in parallel on the CPU using a processor sharing mechanism. The percentage of cold jobs can be estimated from the CTMC as discussed in section 4.3.1.

A FaaS platform provides the benefit of fine-grained resource management but it also requires similar monitoring and performance estimation techniques. Thus, we need the performance estimates of each individual function under different workloads. This means that, we need a performance model that includes all the functions. This increases the modeling complexity if the the FaaS platform has a large number of functions. To mitigate this problem, we need a modeling scheme which is scalable with respect to the number of functions. A Layered Queueing Network (LQN) [FAOW+08] model is often considered as such a scalable modeling scheme. This has been already demonstrated in literature by using LQN to model large scale Service Oriented Architectures (SOA) [TMW10].

The proposed LQN model is presented in Fig. 4.3. The model has two main building blocks - the tasks and the processors. In LQN models, tasks translate into different system resources, usually the software resources. They carry out different operations which are defined by their entries. The tasks are executed on the processors, which represent the physical entity, like the CPU, that carries out the physical executions. Although each of the functions is a software resource, we defined them by the entries rather than the tasks. The reason behind this choice is

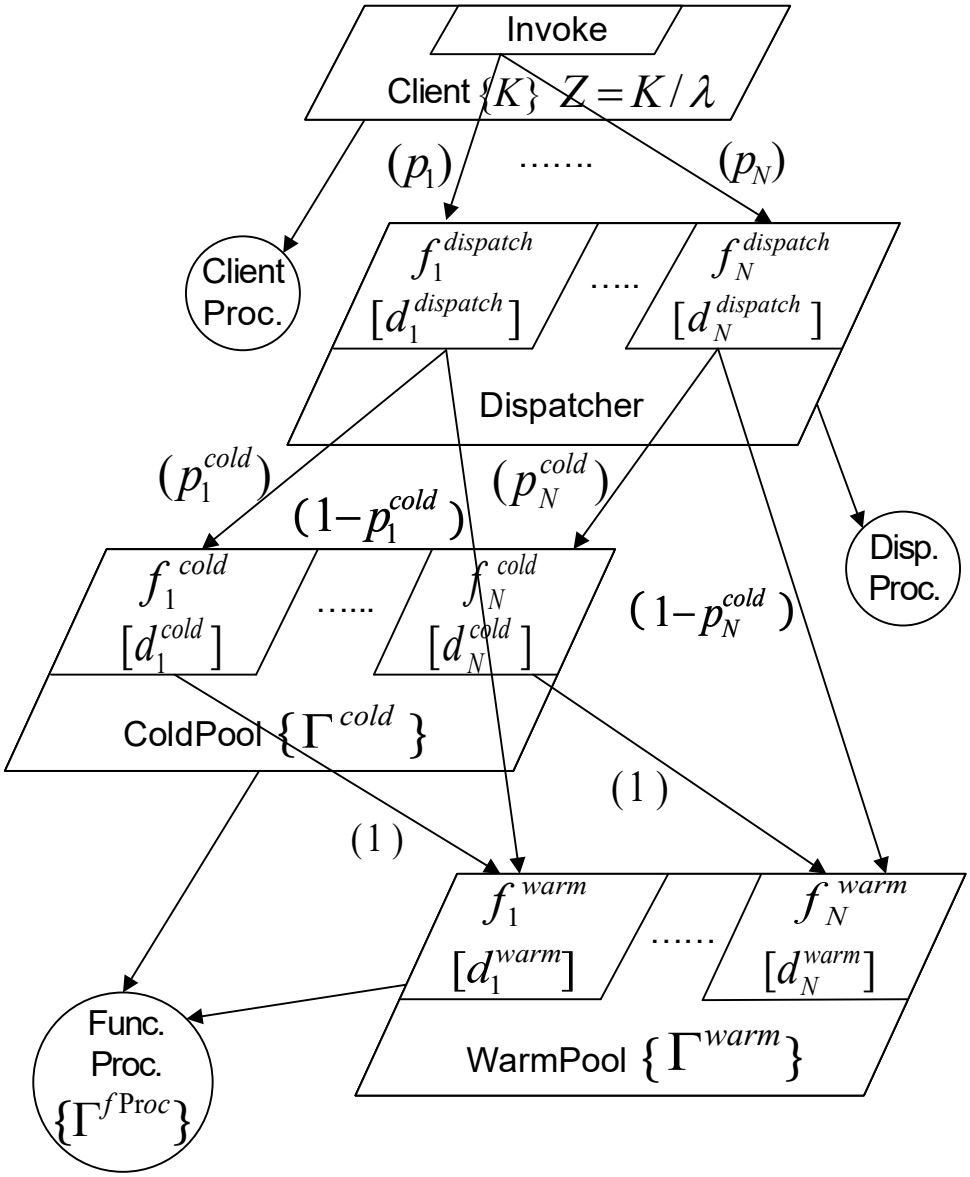


Figure 4.3: An LQN model for a FaaS platform

twofold. Firstly, it makes the LQN model more compact and manageable. Secondly, it reduces the model solving delay as the number of function increases.

The second reason, behind preferring entries over tasks, is particularly more important because the number of tasks in the LQN model has a significant impact over the solution generation time. This is because the LQN models are solved by repeatedly solving its sub-models until the convergence criterion is met [Fra00]. The number of sub-models increases with the increase of number of tasks. A FaaS platform, even though private, is likely to have a large number of functions (for example several tens or hundreds) and thus if they are modeled by tasks, it will significantly affect the solution generation time. In worst cases, the model solvers can even

fail to generate solutions. These scenarios make it impractical to use the model for further analysis under different system configurations. It should be noted that, in principle, an LQN model does not have any limits on the number of entries but a model solver can apply such limits. For example, the LQNS solver is a widely used model solver but it has a limit of 1000 entries [FMW+13]. This means that, we cannot model a FaaS system with functions in the scale of thousands if we use the LQNS solver. We assume that for a private FaaS platform that number is sufficient but public FaaS providers cannot model the entire platform with a single LQN model. In such cases, the service providers need to model the systems by considering subsystems with functions in the scale of tens or hundreds.

Since each function has two types of jobs, in the LQN model, we use two tasks, *ColdPool* and *WarmPool*. The entries in the *ColdPool* define the cold jobs for all the functions. Similarly, the entries in the *WarmPool* define the warm jobs. Since every cold job is followed by a warm job, there is a call from the cold entries to the warm entries. The proportion of cold and warm jobs is controlled by the *Dispatcher* task based on the cold start probabilities. This is done by setting the cold start probability of each function to the call mean value from its *Dispatcher* entry to the *ColdPool* entry. The percentage of calls to each function, based on their popularity, is modeled using the reference task *Client* by setting the percentage value as the call mean from the *Client* entry to the *Dispatcher* entry.

The LQN model requires two parameters, namely the service demands of the activities and the multiplicities of the modeling constructs. The service demand for a job is the total service time across all visits when there is no resource contention. Each function has different service times for its cold and warm jobs. The service times of the cold and warm jobs are defined by the cold start rate (α) and the service rate (μ) respectively. The service demands should be set in the activities of the corresponding entries. The multiplicities translate into different system entities depending on the modeling constructs. The multiplicity of the reference task indicates the number of clients present in the system, considering the system as a closed network [HB13]. However, we can also consider the system as open like our *FaasSim* simulator. To do so, we have adapted the think time Z as K/λ, where K and λ represents the total number of clients and the open arrival rate [SPJN98].

The multiplicities of the processors indicate the number of available CPU cores. Since we do not consider the *Dispatcher* a bottleneck, we assume it executes separately from the functions, on a single CPU core. The multiplicities of the *ColdPool* and *WarmPool* indicate the number of process threads available for the function containers. Container platforms like Docker allow this on a container basis, which means that we can put a limit on how many threads a container can create[4]. However, in LQNs entries do not have a multiplicity property, which we are using to model the functions. Thus, in the model, we consider that the functions share two thread pools for cold and warm jobs. This assumption does not significantly affect the performance estimates if the number of threads, in both pools, are sufficiently large to start processing a job immediately.

To generate the LQN model, we can use the LINE tool [Cas19]. It provides an API for programmatically generating the LQN models. Thus, using LINE, we can generate large and complex models without any manual intervention. We can also change different parameters and observe their effects on the system performance. This would allow us to analyze the system under different configurations for a particular workload. LINE also provides different solvers, like LQNS [FAOW+08], which can provide performance approximations faster for large and complex models. This would benefit in analyzing a model with different parameters within a practical time bound.

4.3.3 Model Validation

We have used the LINE modeling language [Cas19] to build our models, which are solved by the LQNS solver [FAOW+08]. The results are validated with the *FaasSim* simulator. The simulation parameters are presented in Table 4.1. Here, we consider more large-scale settings, compared to Section 4.2, that include up to 128 functions. The function popularity is controlled by different values of the Zipf parameter that are common in cache based studies [Cas18]. We consider all combinations of number of functions (N), Zipf parameter (η) and arrival rate (λ). For each of those combinations, we have generated 30 models. In each of the models, we have

[4]Such limits are put to prevent unnecessary thread creation causing memory leaks. However, the limits are never too small to affect the concurrency.

Table 4.1: Simulation parameters for model validation

N	Number of functions	16, 32, 64, 96, 128
η	Zipf parameter	0.6, 1.0, 1.4
λ	Arrival rate	0.2, 0.5, 0.8
μ	Service rate	[1, 2]
α	Cold start rate	[0.037, 0.5]
β	Idle lifetime rate	[0.00083, 0.00556]

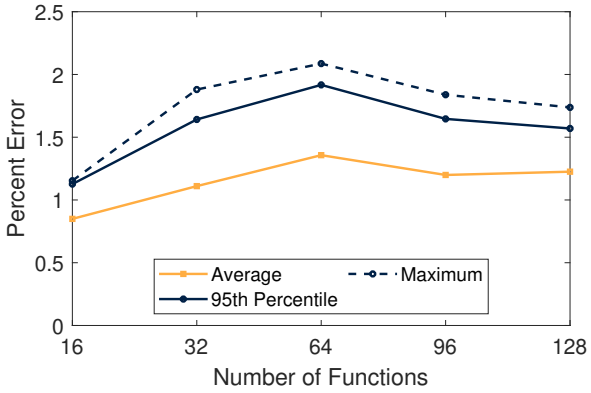


Figure 4.4: The mean, 95^{th} percentile and the maximum modeling error in estimating the response time of each function. Here we have considered 16 to 128 functions, arrival rate $\lambda = 0.8$ and Zipf popularity parameter $\eta = 1.0$

chosen the service (μ), cold start (α) and idle lifetime (β) rates for the functions randomly from the given range. The range for service and cold start rates are the same as Section 4.2. The idle lifetime rates are chosen from [Shi19] such that it can trigger cold starts. The service demand of the *Dispatcher* is set to a negligible value since we do not consider it as a bottleneck.

Since cold start affects the system response time, we are concerned that how accurately our model captures that affect and estimate the response time of each function. To illustrate the modeling error, considering a heavy workload with arrival rate 0.8 and Zipf parameter 1.0, in Figure 4.4, we have plotted the mean, 95^{th} percentile and maximum percent error with the increase in number of functions. We can see that even the maximum error is less than 2.5%. The average and the 95^{th} percentile of the error remains around 1-2%. The increase of the number of functions has a negligible effect over the error. The exact numeric values, considering multiple Zipf parameters, are presented in Table 4.2. The maximum error across all the parameters is 2.38%. The maximum average error and 95^{th} percentile of the error is 1.29% and 2.08% respectively. Such errors are not significant for response times in a multi-class modeling problem. Thus we conclude that the LQN model can accurately estimate the

Table 4.2: Percent error in estimating the response time of each function–Across all the Zipf parameters in Table 4.1

N	λ **= 0.2**			λ **= 0.5**			λ **= 0.8**		
	avg	**95p**	**max**	**avg**	**95p**	**max**	**avg**	**95p**	**max**
16	1.24	**2.08**	**2.38**	0.87	1.57	2.2	0.91	1.48	1.65
32	1.25	2.01	2.3	1.17	1.87	2.21	1.15	1.75	2.16
64	0.85	1.7	2.06	1.21	1.87	2.15	1.27	1.72	2.09
96	0.82	1.36	1.98	1.11	1.57	1.83	1.25	1.7	2.05
128	0.92	1.31	1.71	1.11	1.51	1.95	**1.29**	1.7	2.1

response time for all the functions considering cold starts. We leverage this model in our method, COCOA, which we present in the following section.

4.4 Capacity Planning

In this section we present our method COCOA. Initially we provide an overview that discusses the its components, inputs and outputs. Subsequently, we present the problem formulation, focusing on the resource management issues amid cold starts. After that, we discussed the optimization approach used in COCOA that provides a capacity plan and function idle times to satisfy the SLAs.

4.4.1 Overview

COCOA provides resource allocation decisions for a FaaS platform in terms of its memory and CPU configurations. It also provides the function idle times that can ensure the SLA given the suggested configuration is applied. These idle times allows to control the magnitude of cold starts, which in turn aids in governing the response times. The configurations can be applied both on the hardware level or on the software level. This means that the memory and CPU constraints can be applied on the physical server or the container platform like Docker. COCOA has multiple components each focusing on a particular tasks and expecting different inputs, which are illustrated in Fig. 4.5.

Since COCOA is a model-based approach, we need a set of parameters to instantiate its model.

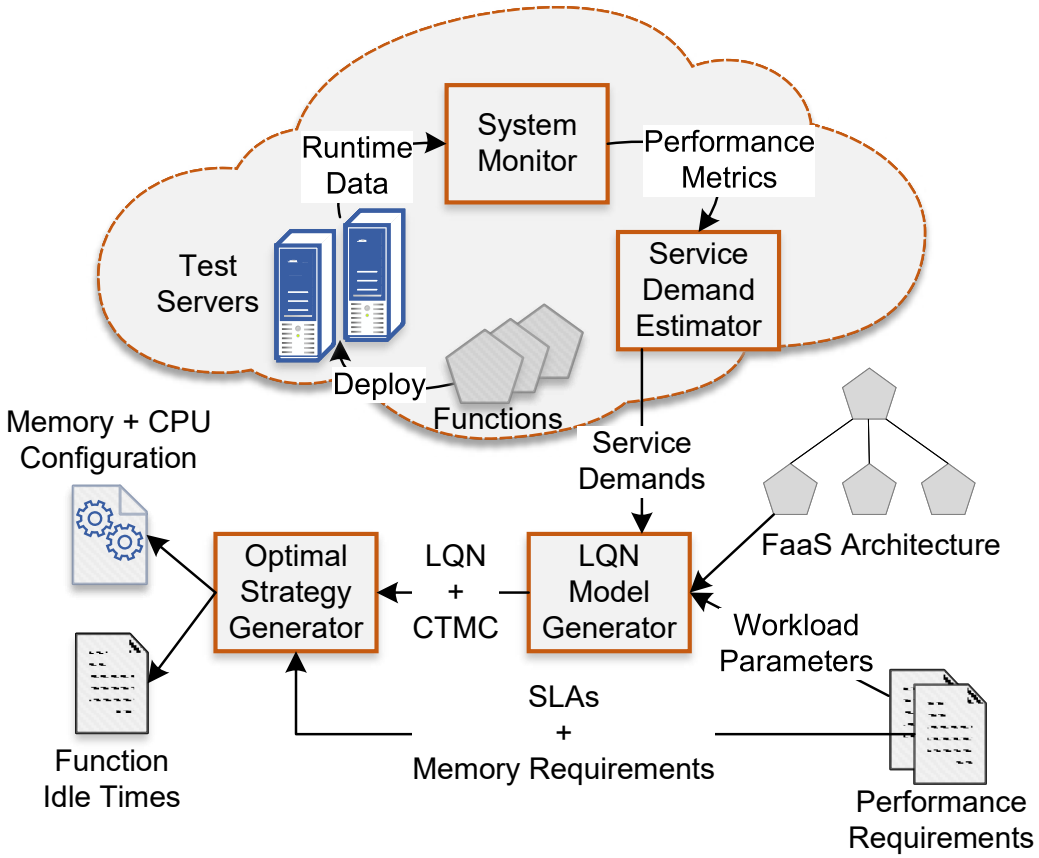


Figure 4.5: An overview of the COCOA approach

Firstly, we need the service demands of the functions. These can be estimated in a test environment, when the functions are being developed, using state-of-the-art methods [GCW19]. The workload parameters like the arrival rate and function popularity can be estimated from the performance requirements or the historical data. Once the parameters are estimated, they are passed to the component LQN Model Generator. The Model Generator also requires the architecture of the FaaS platform, particularly containing the information about how the functions communicate.

Based on these inputs, the CTMC and LQN model is generated and forwarded to the next component, the Optimal Strategy Generator. It utilizes the models to provide memory and CPU configurations. It needs the SLAs for function response times and both the memory requirements when they are idle and in execution. The generator then searches for the idle times, under different CPU configurations, that do not violate the SLA with minimal memory consumption. These idle times are used to estimate the maximum memory consumption, based on which the memory capacity is suggested.

4.4.2 Problem Statement

We consider a system of N functions. These functions are executed on a multi-core CPU with C cores. Every function f_i has two different memory usage, one is while in execution (θ_i^{on}) and the other is while being idle (θ_i^{off}) Considering a request for function f_i, if the function is not in the memory, a cold start occurs. Due to this cold start, a request experiences an extra delay. This extra delay is incurred to load the function in the memory. When a function is loaded in the memory, it is associated with a timeout value T_i. While the function is still in the memory, for each new request, the timeout is reset to the original value. A function is removed from the memory if it reaches the timeout limit.

We focus on two specific costs, the cost of CPU and cost of the memory. We define the per unit CPU and memory cost as τ_c and τ_m respectively. Thus, the cost for the CPU will be $B = \tau_c C$. The memory cost is calculated based on maximum memory consumption. To estimate this, we incorporate the idea of a different memory usage, when the function is idle, with the estimator for TTL cache [FRR12]. Based on this, given the function CPU utilization is ρ_i, the average memory consumption (m) can be estimated using (4.2).

$$m = \sum_i h_i(\rho_i \theta_i^{\text{on}} + (1 - \rho_i)\theta_i^{\text{off}}) \tag{4.2}$$

The system's memory capacity should be adequate when there is a spike in memory consumption. This occurs when there is a surge in requests within a short period. This increases the memory consumption because more functions starts execution, for which the memory requirement is much higher than being idle. It is sufficient to consider this increase in memory consumption by the functions in execution. Considering the memory consumption by the functions in execution is U, we define the expectation in (4.3).

$$E[U] = \sum_i h_i \rho_i \theta_i^{\text{on}} \tag{4.3}$$

We can approximate the maximum consumption as $v = \kappa E[U]$. The value of κ is calculated,

using Markov's inequality, [Nel13] such that the upper-bound of $P(U \geq v)$ is a negligible value ϵ. Based on this, we define the approximation for maximum memory consumption ($m^{\max}$) in (4.4). Now, we can define the memory cost as $A = \tau_m m^{\max}$.

$$m^{max} = \sum_i h_i(\kappa\rho_i\theta_i^{\text{on}} + (1-\rho_i)\theta_i^{\text{off}}) \tag{4.4}$$

Considering these cost functions A and B, our objective function, z, is defined in (4.5). Here, our goal is to find T, a vector including the idle times T_i of all the functions, and C, the number of CPU cores, that minimizes a weighted sum of normalized memory and CPU cost.

$$z = \min_{(T,C)} \omega_A \hat{A} + \omega_B \hat{B} \tag{4.5}$$

$$\text{subject to:}$$

$$C \leq C^{\max} \tag{4.6}$$

$$W_i(T,C) \leq W^*, \forall i \tag{4.7}$$

$$T \in \mathbb{R}_+^N, C \in \mathbb{Z}_+$$

The constraints for the objective function are provided in (4.6) and (4.7). The first constraint in (4.6) is regarding the maximum number of allowed CPU cores. This applies when the CPU constraint is imposed on the software level and the total physical CPU capacity is not accessible. The second constraint in (4.7) addresses the SLA for response time. Here, W_i is a function of T and C which returns the response time for a platform function f_i. This response time should be less than the limit W^* mentioned in the SLA.

4.4.3 Optimal Strategy Generation

Using the objective function in (4.5), COCOA provides an optimal strategy that includes the memory and CPU capacity, and the function idle times (T). The steps for searching for this optimal strategy is provided in Algorithm 4.1. The notation used in the algorithm is described in

Table 4.3: Notations used in Algorithm 4.1

N	Number of functions in the FaaS platform
λ	Arrival rate of the function requests
P	Cold start probabilities of the functions
S^{cold}	Service rates of the cold start tasks
S^{warm}	Service rates of the regular tasks
X	Solution candidates for the current workload
T	Maximum instance idle times of the functions
T^{part}	Partition set of maximum instance idle times
W	Estimated response times of the functions
W^*	Response time of the functions in SLA
C^{conf}	Set of available CPU configurations
L	Iteration limit for searching the optimal T

Algorithm 4.1 Searching Optimal Strategy

Input: N, λ, P, S^{cold}, S^{warm}, C^{conf}, W^*, L
Output: T, W
1: **Init:** $T \leftarrow ApproxCT(N)$, $T^{part} \leftarrow \emptyset$, $X \leftarrow \emptyset$
2: **for each** $C \in C^{conf}$ **do**
3: **for** $i = 1 : L$ **do**
4: $W \leftarrow SolveModel(\lambda, P, S^{cold}, S^{warm}, T, C)$
5: $(T, T^{part}) \leftarrow UpdateT(T, T^{part}, W, W^*)$
6: **end for**
7: **if** $W_i \leq W^*, \forall_i$ **then**
8: $X \leftarrow X \cup \{\lambda, P, W, T, C\}$
9: **end if**
10: **end for**
11: $(T, W) \leftarrow RecommendT(X)$

Table 4.3. A summary of the auxiliary functions used in the algorithm is provided in Table 4.4. Here, we discuss these steps in detail.

The searching process starts searching for an optimal strategy with an initial instance of T. This is obtained by a characteristic time approximation technique for a CDN cache [CTW02]. It requires to solve $m = \sum_i h_i$, where h_i is defined in (4.1), for a particular value of m. However, as we have a large pool of functions, it is sufficient to estimate a single value T^* for all the functions instead of approximating $T_i \in T, \forall_i$ [MGL14]. Thus, here we have used a second definition of h_i, replacing T_i with T^* in (4.1).

After the initialization, COCOA fine-tunes the idle times, such that the function response times are just under the SLA limit, to ensure minimal memory consumption. For this, it solves the LQN model in iteration, upon adjusting the idle times, to observe its effect on the response

Table 4.4: Auxiliary functions for Algorithm 4.1

ApproxCT	Provides the initial values of T using a characteristic time approximation technique for a CDN cache.
SolveModel	Solves the CTMC and LQN model to provide the response time of each function
UpdateT	Fine-tunes the function idle times T using binary search such that the SLAs are not violated.
RecommendT	Suggests the optimal solution by comparing the memory and CPU cost of each candidate solution.

time. The idle times are adjusted using the concept of binary search. It starts with an initial searching interval, $(0\ T^*]$, for each T_i and reduces the length of the interval by half on each iteration. The endpoints of the intervals are adjusted depending on whether the response time constraint is satisfied or not. The value of T_i is updated with the midpoint of the searching interval. For this process to work, the initial value, T^*, should be sufficiently large so that there is no cold starts and thus the response time is not affected. For this purpose, we have solved $m = \sum_i h_i$ by setting m to a value close to N.

COCOA runs this fine-tuning process for different CPU configurations (CPU cores). Although the number of CPU cores can be any integer, practically we only need to consider some common options, like multiples of 2 with 32 as the limit. This accelerates the analysis process. In addition, for each configuration, this process is run in parallel, making it even faster. For each run, if a T is found, that does not violate the SLA, it is considered as a candidate solution. After completing the process, the optimal solution is selected by comparing the memory and CPU cost. Its corresponding CPU configuration and idle times are suggested just the same. However, the memory capacity is suggested by considering the value $\min(m^{\max}, \sum_i \theta_i^{\text{on}})$ as an upper-bound and calculating the aggregated size of required number of RAM modules[5].

4.5 Evaluation

In this section, we evaluate the performance of COCOA with respect to availability-aware approaches based on the principle of TTL cache. Initially, we discuss the experimental setup

[5]We have considered that each of the memory module is 8GB but this is configurable depending on the availability of RAM modules.

and the research questions we aim to answer from these experiments. After that, we present our analysis of the obtained results.

4.5.1 Experimental Setup

We have evaluated COCOA using the *FaasSim* simulator. Here, we consider 64, 96 and 128 functions varying in terms of their invocation probabilities, controlled by the Zipf popularity parameter (η), service time ($\frac{1}{\mu}$) and cold start time ($\frac{1}{\alpha}$) according to the parameters in Table 4.1. However, the idle times ($\frac{1}{\beta}$) from Table 4.1 are not used because we want to estimate these with COCOA such that the response times constraints are satisfied with minimal memory and CPU capacity. We have set the functions memory requirement following the limits in AWS Lambda [Ser]. The percentage (0-1) of idle function memory consumption is considered to be log-normally distributed with a desired mean of 0.2. From the experiments, we aim to answer the following research questions:

- **RQ1:** Can COCOA reduce memory over-provisioning compared to the availability-aware approaches?
- **RQ2:** Can COCOA predict the required memory capacity that meets the maximum memory demand?
- **RQ3:** Can COCOA predict the memory and CPU capacity to satisfy the SLAs for function response time?

4.5.2 Results

To answer **RQ1**, we have compared COCOA and the availability-aware approach with two hit rates, 0.8 and 0.95. We present the result for a single experiment in Fig. 4.6. We can see that the required memory capacity estimated by COCOA is much lower than the other two approaches. The results for all the parameters are presented in Table 4.5. In all the cases, the estimates from COCOA is much less compared to the other two approaches. Considering

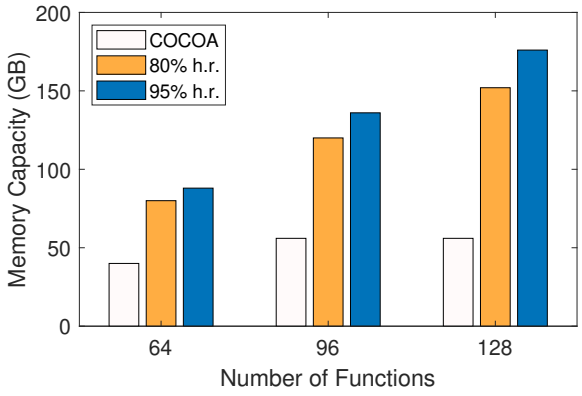


Figure 4.6: Comparing the memory capacity predicted by COCOA and the availability-ware approaches for $\lambda = 0.5$ and $\eta = 1.0$

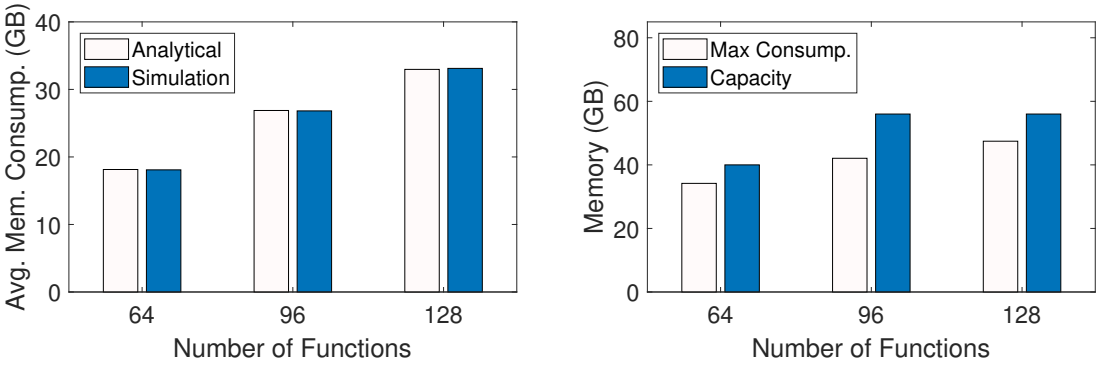
Table 4.5: Comparing the predicted memory capacity of different approaches - averaged across the Zipf parameters from Table 4.1

N	λ **= 0.2**			λ **= 0.5**			λ **= 0.8**		
	COCOA	**80% h.r.**	**95% h.r.**	**COCOA**	**80% h.r.**	**95% h.r.**	**COCOA**	**80% h.r.**	**95% h.r.**
64	32	88	104	45.3	80	93.3	42.7	85.3	101.3
96	40	128	152	58.7	125.3	146.7	48	120	141.3
128	48	157.3	184	58.7	157.3	184	64	168	197.3

the 95% hit rate, the capacity estimated by COCOA is 51-74% less. The reason is easy to understand - COCOA can take "well-informed" decisions by leveraging its performance model, which is not possible for the availability-aware approaches.

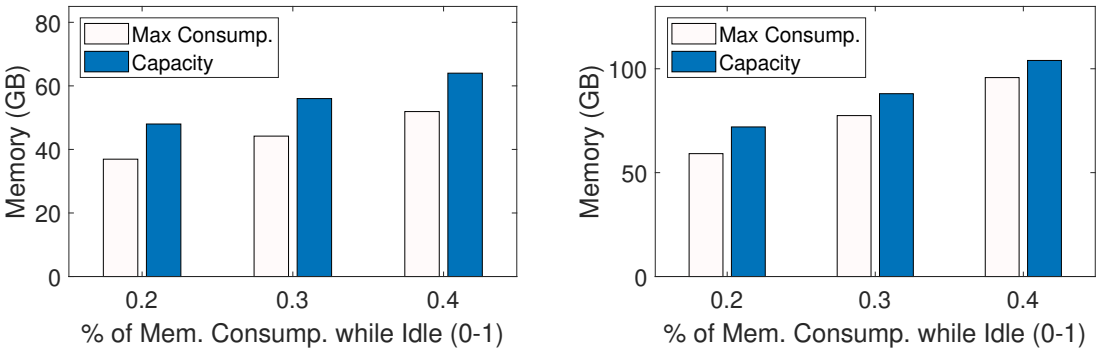
From Table 4.5, we see that, in two cases, COCOA predicts a higher capacity for $\lambda = 0.5$ than $\lambda = 0.8$, which is counter-intuitive. This is because we have used a different upper-bound of $P(U > v)$ to estimate κ for $\lambda = 0.8$. For $\lambda = 0.8$ it is 0.1 but for $\lambda = 0.2$ and 0.5 it is 0.05. The reason is, for high arrival rates, the spike in memory consumption from the expectation is less than low arrival rates. Here, κ is the coefficient to represent this extent. A larger upper-bound of $P(U \geq v)$ will result in a smaller value of κ. So for higher arrival rates, to reduce over-provisioning, κ should be approximated with a larger upper-bound of $P(U \geq v)$.

To answer **RQ2**, we have compared the memory consumption values from the simulation with the values from COCOA. From Fig. 4.7a, we can see that the average memory consumption values from the simulation agrees with analytical approximation from COCOA. From Fig. 4.7b, we see that the memory capacity also meets the maximum demand. From all the experiments,



(a) Avg. Memory Consumption (b) Capacity vs. Consumption

Figure 4.7: Comparing the runtime memory consumption obtained from the analytical approximation and simulation for $\lambda = 0.5$ and $\eta = 1.0$

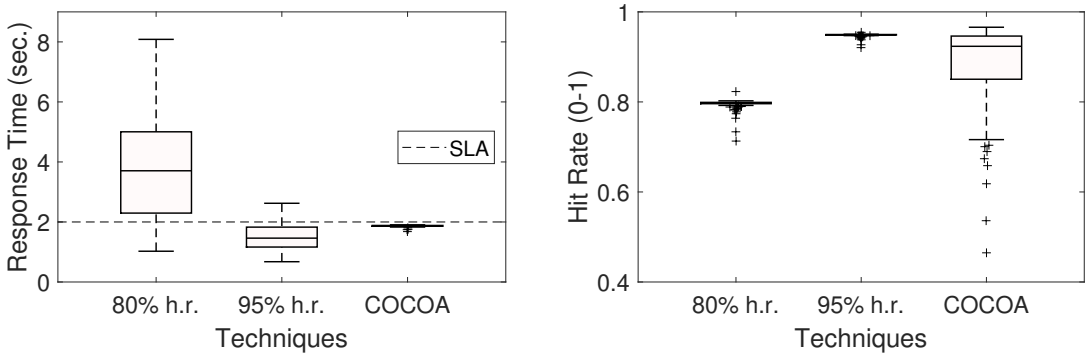


(a) 64 Functions (b) 128 Functions

Figure 4.8: Illustrating that COCOA can meet the maximum memory demand for different percentage of memory consumption while the functions are idle

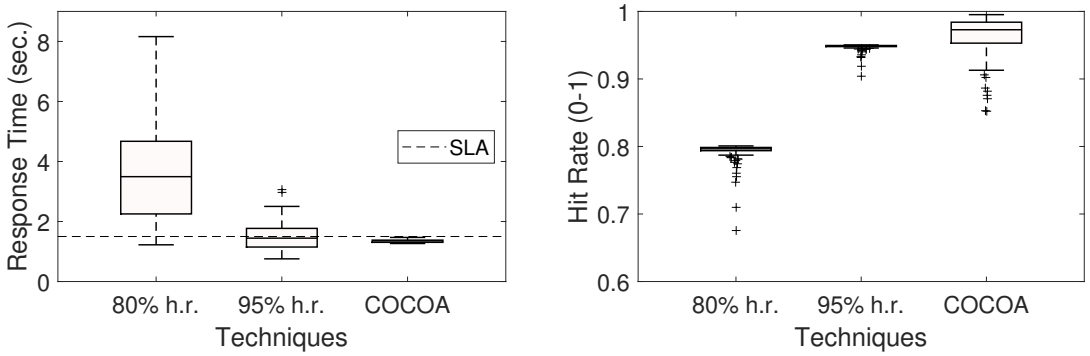
we observe only 5 cases where there is a memory deficit greater than 0.5 GB with a maximum value of 3.2 GB. We have also done a sensitivity analysis changing the desired mean of percentage of memory consumption by idle functions. We used two settings with 64 and 128 functions with $\lambda = 0.8$ and $\eta = 1.0$. As seen from Fig. 4.8, in both cases, COCOA can satisfy the maximum demand.

To answer **RQ3**, we have investigated the response time of each of the functions. We have seen that across all the parameters, COCOA can ensure the SLA for response time. We present the response time of each function, for a single experiment, in Fig. 4.9. The SLA in this case is 2 seconds. From the figure, we see that COCOA satisfies the SLA for all the functions with hardly any variance. On the other hand, even the policy based on 95% hit rate has violations. The violations are even more when the SLA is 1.5 seconds. We have seen 45% violations i.e.,



(a) Response time (b) Hit rate

Figure 4.9: Response time and hit rate distribution of 128 functions with a target SLA of 2 seconds for $\lambda = 0.5$ and $\eta = 1.0$



(a) Response time (b) Hit rate

Figure 4.10: Response time and hit rate distribution of 128 functions with a target SLA of 1.5 seconds for $\lambda = 0.8$ and $\eta = 1.0$

there are violations for 57 out of 128 functions

To illustrate how COCOA ensures the SLA without over-provisioning, we have investigated the hit rates of each function obtained from the simulation. In Fig. 4.9b and 4.10b, we have plotted the hit rates, which correspond the experiments from Fig. 4.9a and 4.10a. As expected, we see that the hit rates are fixed for 80% and 95% hit rates. However, COCOA adjusts the idle times of the functions such that the hit rates are just sufficient to satisfy the SLA. These adjustments reduce the memory consumption when the functions are idle. COCOA uses this insight on hit rates to suggest a much lower memory capacity. We see that, for a 2 seconds SLA, the lowest hit rate a function has is only 46%. However, COCOA can also increase the hit rates, if required, as seen in Fig. 4.10b. Here the hit rates for some functions are even higher than 95% to satisfy a stricter SLA of 1.5 seconds.

4.6 Summary

In this Chapter, we have presented COCOA, a cold start aware sizing method for on-premises FaaS platforms. We have first demonstrated that though an availability-aware approach for TTL cache can partly ensure the SLA for response time, it leads to resource over-provisioning. To resolve this issue, COCOA leverages an LQN model and M/M/k setup models to obtain different performance estimates and consequently, predict the required system capacity. We have illustrated the improvements yielded by COCOA with multiple experiments. The results shows that COCOA can help in sizing FaaS platforms that satisfy the SLAs, while reducing resource over-provisioning.

Chapter 5

Storage Management for Distributed Tracing

5.1 Introduction

The distributed tracing approach is tailored primarily to monitor and profile applications built with the microservices architecture [Ric18]. In a microservice ecosystem, with the increase of microservices, the volume of the trace data increases significantly [GPW+20]. This is a problem for a cloud data center. A large data center can consist of hundreds of nodes and a single node hosts hundreds of microservices. By the end of each day, these microservices can generate tens of gigabytes of trace data per node. Considering all the nodes, the total generated data are in the scale of tens of terabytes per day. It is obvious that most of these trace data are not interesting or anomalous and thus there is no value in storing all of those. Rather, only a fraction of those traces are stored to reduce the storage capacity. This fraction can be interpreted as the storage budget [LCMGF18].

To accommodate the storage budget, a sampling strategy needs to be deployed. A common choice is to use head-based sampling [PSM+20]. Here, the sampling decision is taken when the request is first received, rather than when the response is served. This strategy is easy to implement since it only requires considering the budget as the sampling rate. However,

since this strategy does not consider the information of the trace, the hit rate of interesting or anomalous capture is low. To counter this, a tail-based sampling [PSM+20] strategy is preferred. It takes the sampling decision after the response is served when the entire trace is available. As a result, the information of the traces can be utilized and more interesting traces can be captured.

A tail-based sampling strategy should be online and ideally without any batch processing. This means, we must decide either to save or discard a trace on the fly rather than storing it temporarily for batch processing. Recently, researchers have proposed different sampling strategies based on unsupervised learning [LCMGF18, LCPAM19, HCY+21]. However, these works have limitations like inaccurate clustering due to high number of data dimensions [LCMGF18], requiring batch processing [LCMGF18], low amplitude signals for anomalous traces [LCPAM19], not considering the size of the budget relative to the fraction of anomalies (e.g. budget is smaller than the fraction of anomalies) [LCMGF18, LCPAM19, HCY+21].

To address these shortcomings, we propose a novel method, namely *SampleHST*, for distributed tracing. On the one hand, SampleHST focuses on sampling only anomalous traces when the storage budget is comparatively lower than the fraction of expected anomalies. On the other hand, when the budget is higher, SampleHST samples both the normal and anomalous traces, with a bias towards anomalous ones. Regardless of such a bias, SampleHST is a fair sampling method. Based on the definition of max-min fair share [Kes97], we consider a sampling approach fair when it samples more from the group of traces that constitutes a small fraction of the traces generated. In other words, it increases the representation of the rare or anomalous traces among the sampled ones. Thus, the bias in SampleHST basically helps to achieve fair sampling. This fair sampling is similar to the concepts of representative sampling, which has been highlighted in recent studies [LCMGF18, LCPAM19].

To identify different types of anomalies, we propose to use two count based trace representations. These representations are based on the Bag of Words (BoW) model [ZJZ10] and Family of Spectral Graph Distances (FGSD) graph model [VZ17]. A novel idea that we incorporated with SampleHST is creating a pipeline of a tree based classifier, namely Half Space Tree (HST)

[TTL11], with an online clustering method. By taking a trace as an input, we generate a distribution of the so-called mass values from a forest of HST. We use this distribution to cluster the traces using an online clustering method based on the mean-shift algorithm [BA12]. Once the clustering is complete, we decide to sample the trace based on its association with the clusters, where certain associations are prioritized over the others.

To compare the performance of SampleHST with other sampling approaches, we used two different datasets. The first dataset is a production data, provided by a cloud data center company, that contains 77,577 traces. The second dataset is generated from a local deployment of the Death Star Bench (DSB) microservice suite [GZC+19], which is already used in literature to demonstrate the effectiveness of such a sampler [LCPAM19]. We compared the results with two recent approaches [LCMGF18, LCPAM19]. For the production data, we see SampleHST yields 2.3× to 9.5× better sampling performance. When we consider representative sampling in a high budget case, SampleHST is 1.6× more fair in terms of Jain index [JCH84]. For the DSB traces, we have considered two different cases, each representing a different part of the system. In all the cases, SampleHST produces a high sampling performance. In comparison, the sampling performance of SampleHST is 1.2× to 19× better across all the budgets.

In a nutshell, the key contribution of this study is as follows.

- Integration of two different trace models for anomaly detection - Bag of Words model for point anomalies and FGSD model for collective anomalies.
- A novel idea to sample the traces by forming clusters using the mass distribution of the traces obtained from Half Space Trees.
- An online clustering method, based on the mean shift algorithm, that considers hyper-rectangles as cluster shape instead of the commonly used hyper-spheres.
- A novel method SampleHST to sample the traces based on their cluster association, where the associations are prioritized based on the clusters proximity towards the origin of the clustering hyper-plane.

- Setting evaluation criteria, considering the proportion of anomalies and budget, and comparing the sampling performance of SampleHST with two recent approaches.

Rest of the chapter is organized as follows. In section 5.2 we demonstrate the motivation for developing SampleHST. In section 5.3, we present how we can model the distributed traces considering different types of anomalies and how those anomalies can be detected with a classifier. In section 5.4, we present the SampleHST sampling method. We evaluate the sampling performance of SampleHST with production data in section 5.5. In section 5.6, we discuss how SampleHST can be extended for collective anomalies and show its effectiveness in such cases. Finally, a summary of the chapter is provided in section 5.7.

5.2 Issues with Tail-based Sampling

Unlike head-based sampling, a tail-based sampling strategy takes the sampling decision after the response is served i.e. when the entire trace is available [PSM+20]. As a result, the information of the traces can be utilized and more interesting traces can be captured. Before evaluating these samplers, we first need to define the interesting traces and the evaluation criteria. In this section, we first provide this definition and criteria, which is followed by the experiments that show the limitations of the current samplers.

5.2.1 Trace Anomaly Definition

A key point before designing a tail-based sampling strategy is to define what makes a trace more interesting or anomalous. For the rest of the chapter, we will use the term anomalous traces to indicate interesting traces. This definition of anomalous trace is important because the trace representation depends on it. In the context of this work, we use the broad definition of anomalies provided by Chandola *et al.* [CBK09]. Precisely, we consider two types of anomalies – point anomalies and collective anomalies. Other type of anomalies, *e.g.*, contextual anomalies, are not considered for this study.

Point anomalies are those where a trace has abnormally low or high value for one or more trace properties. For example, suppose we define a trace property that counts the occurrence of HTTP code 203 among its spans. If there is a large deviation than the mean value, it is an anomaly. On the other hand, collective anomalies are those where there is an abnormal workflow pattern within traces. For example, if a group of spans always creates the same call-chain (like span A calls span B and span B calls span C) and in a particular trace we see a deviation, this trace is an anomaly.

5.2.2 Evaluation Criteria

It is important to set appropriate criteria for evaluating the performance of a tail-based sampler. In some recent works, researchers have used the notion of representative sampling to evaluate their proposed method [LCPAM19]. Although, this adoption of representative sampling provides a visualization of sampling fairness, it is not a numeric metric that can be used for comparison. Further, it does not provide the accuracy of the method, with respect to the storage budget and the fraction of anomalous traces in the data. For example, if the budget (bounded between 0 to 1, indicating what percentage of the traces can be saved) is smaller than the expected fraction (bounded between 0 to 1, indicating a percentage of the traces) of the anomalous or interesting traces, the focus should be sampling the anomalous traces. This is because, with such a small budget the goal is to identify the faults in the system, which will not appear in the normal traces. Considering this we define the following evaluation criteria:

- For infrequent anomalous traces, where the prevalence of anomalies is less than the storage budget, the primary evaluation metric should be the Recall.
- For low storage budgets, where the prevalence of anomalies is greater than the storage budget, the primary evaluation metric should be the Precision.
- When sampling N traces from a collection of traces containing N anomalies, the primary evaluation metric should be the F1-Score.

- Fairness measure like the Jain index [JCH84] can be a secondary metric and used to quantify the degree of representative sampling.

Here, the definition of precision and recall is same as the definition in the machine learning domain. The F1-Score is the harmonic mean of precision and recall. We discuss in detail about Jain index in Section 5.5.

5.2.3 Demonstration

We now demonstrate our motivation for a new tail-based sampler. For this demonstration, we have chosen two recently proposed sampler - one for the point anomalies and the other for the collective anomalies scenario.

For the point anomalies scenario, we have chosen a method [LCMGF18] based on a hierarchical clustering algorithm, namely Purity Enhancing Rotations for Cluster Hierarchies (PERCH) [KMKM17]. In short, this PERCH-based method maintains a fixed size balanced binary tree of the traces and samples from the tree periodically. The tree is maintained in such a way that the frequent traces are more likely to be removed when there is a space shortage. For the collective anomalies scenario, we have chosen the Sifter approach [LCPAM19]. It considers a trace as a directed acyclic graph (DAG) and predicts the anomalies by inspecting all the k-length paths from the DAG. Here, each path is represented as a n-gram i.e. a sequence of n words. The association among these words are learnt using a variant of *word2vec* algorithm [LM14]. The traces are sampled based on the familiarity of these n-grams. We have implemented both these approaches using Python libraries[1][2].

We now consider a case with the production data the production data from a cloud data center. The data contains $\sim 5\%$ point anomalies. We have provided a storage budget of 5% (we can store 5% of the traces) to PERCH-based method and executed it. As mentioned above, since the budget and the percentage of anomalies are roughly the same, the evaluation criteria should be F1-Score. After the execution we have got an F1-Score of 0.1, which is just slightly above

[1]Gensim - `https://radimrehurek.com/gensim/models/word2vec.html`
[2]PERCH - `https://github.com/iesl/xcluster`

the value of random sampling 0.05. Although, for a low budget of 0.5%, where the evaluation criteria is precision, we see the precision value of the method is 0.41, where for random it is only 0.04. This shows that the PERCH-based method has potential in low budget scenarios but not for a moderate budget.

The primary drawback of such clustering method is it is hard to do clustering in high number of dimensions [Fuk13], which can be often the case for point anomalies scenario. Thus, for clustering, we need to find a representation that uses less number of dimensions. In addition, this PERCH-based method is a batch process as it needs to maintain a binary tree. This is a major problem when we have no mechanism for temporary storage and need to immediately take the sampling decision.

The production data we have do not contain any collective anomalies. Thus, we have used the Death Star Bench (DSB) traces introduced by Las-Casas *et al.* [LCPAM19]. We have considered the *Compose Post* API for this experiment. The trace repository[3] contains both the traces with normal and broken executions, where the broken executions are the ones with runtime faults. Like [LCPAM19], we have considered the broken traces as anomalies and created dataset where 5% of the traces are broken. We than executed the Sifter method with 5% budget and obtained an F1-Score of 0.35, which is better than the point anomalies case but still low.

The issue with the Sifter approach is that it samples completely in a probabilistic manner. This is problem because if the sampling probabilities for the anomalies are not large enough, the trace may not be sampled. To be certain that our implementation is not producing small probabilities, we checked the probability reported for the Sifter method [LCPAM19]. We have seen the highest probability obtained for Compose Post API is ~ 0.3. This means 2 out of 3 anomalous traces will not be sampled, which will definitely affect the F1-Score. This is confirmed from our experiments. Another issue with Sifter is its trace representations. If we consider data with runtime faults, the normal and erroneous executions will be different and so will be their traces. Thus, rather than considering a slow process requiring to check every path, we can consider the complete DAG structure of the trace.

[3]Death Star Bench Traces `https://gitlab.mpi-sws.org/cld/trace-datasets/deathstarbench_traces`

In the next sections, we present our tail-based sampler that addresses the above issues. We first present how we can solve the issues with point anomalies. After that we present the extension to our method, which makes it work also for the collective anomalies.

5.3 Anomaly Detection from Trace Stream

The first step of building of a tail-based sampler is to be able to differentiate between normal and anomalous traces. After that, we need to decide how to use that information to sample the traces. In this section, we discuss the first step that is differentiating between the normal and anomalous traces. Here we have used the production trace data containing point anomalies. We first discuss how we can represent these traces. Subsequently, we evaluate the performance of different state of the art anomaly detection methods using the representation. Finally, we discuss the adjustments we made to Half Space Trees [TTL11], the method that performed best, to better suit our case of distributed tracing.

5.3.1 Bag of Words Model for Point Anomalies

The production data that we consider consists of spans in the scale of hundreds of thousands. This data is generated in a cloud data center. They come from a heterogeneous collection of microservices in the form of spans that can grouped into traces. A span is an immutable data structure that supplies the value of a collection of categorical and continuous variables at a particular point in time. The spans contain a *traceId*, based on which they can be grouped to form traces.

We consider each trace as a document where the span properties are considered as terms or words. The properties that are uninformative (meaning they are either same or unique for all the traces e.g. *productName* or *id*) are ignored. The *traceId* is used only for trace reconstruction, not for trace representation. Continuous fields related to latency have not been used as they require non-standard distance measures for integration in an approach since the other variables

are discrete. In addition, identifying latency anomalies falls into a separate class of problem, which is not in the scope of this research.

Now, we represent each trace as a count vector $x = (x_1, \ldots, x_d, \ldots, x_D)$, where D is the number of different terms that have been seen across all the traces in the dataset. For example, the HTTP code 200 is one term and a specific URL could be another one. Each dimension $x_d \geq 0$ is an integer value counting how many time a particular term appears in a trace. The resulting count data is agnostic of assumptions on the dataset, except knowledge of the total dimension D and what each dimension means. We can acquire such knowledge from an initial monitoring period. This knowledge-base can also be updated in frequent intervals depending on the changes in microservices and their deployment architecture.

5.3.2 Evaluation of Anomaly Detection Methods

We now present, how the BoW model helps in identifying anomalies. We initially present the considered methods, followed by the anomaly detection results.

Methods

We have evaluated a wide range of methods for this experiment, each based on different concepts for identifying the anomalies. These concepts includes, local density estimate: *K-Nearest Neighbor* (KNN) and *Local Outlier Factor* (LOF), tree-based classification: *Isolation Forest* and *Half Space Trees* (HST) [TTL11], boosting: *Lightweight Online Detection of Anomalies* (LODA) [Pev16], and neural network: *Deep Belief Net and One Class Support Vector Machine* (DBN+OCSVM) [ERKL16]. It should be noted that apart from HST, all the other methods require a batch of data to decide what are the anomalies. HST on the other hand, can decide based on one trace at a time, making it particularly suitable for streaming data.

Numerical Results

To evaluate the performance of the methods, we need the ground truth. Due to the large scale of the production dataset, a manual data labelling is not feasible. Instead, we used the

Table 5.1: Results of different anomaly detection methods on the production dataset

		Isolation Forest	**KNN**	**LOF**	**LODA**	**DBN + OCSVM**	**HST**
Six Days (4892 anomalies)	Precision	0.73	0.77	0.73	0.62	0.47	0.94
	Recall	0.72	0.72	0.72	0.60	0.97	0.70
	F1-Score	**0.73**	**0.74**	**0.73**	**0.61**	**0.64**	**0.80**

popular offline DBSCAN clustering algorithm to cluster the traces. We considered the smallest clusters, that composed $\sim 5\%$ of the total traces as anomalies. The results of the experiment is presented in Table 5.1. The dataset contains traces from six consecutive days with 77577 traces. For all the batch methods, we kept a similar batch size of 2000. Even though, HST processed one trace at a time, from the results we see that it is still the best method. Neither of the method reached an F1-score value of 0.8, which HST yielded.

In addition, to its superior F1-Score, HST has other benefits from the perspective of a streaming platform. Due to the way HSTs are designed, for a particular trace, we only need to update a single mass value per tree. It only need to query its already stored mass values to determine whether a trace is normal or anomalous. We can also built the HST forest, without the data, if we use a $[0, 1]$ workspace [TTL11]. This will reduce the time taken during the training, where we can only use the computing resource to update the mass values of the node. Due to all these benefits, we have decides HST as our classifier that will indicate the nature of a trace. In the next section, we present the further modifications we applied to HST.

5.3.3 Modifications to Half Space Trees

Before applying the modifications to HST, we first need to decide which version of it to use. There is a variant of HST known as HS*-Trees (HS*T) [TZLT13], which aims to deal with the sparsity in the tree structure. When we experimented with the production data set, we indeed observed a significant degree of sparsity, quantified by the number of terminal nodes that are non-zero. This means that majority of the terminal nodes have a zero mass value leading to a memory wastage. Recognizing this can help to largely reduce memory consumption. Sparsity arises in the traces by virtue of the number unique feature vectors that appear in the stream and

the rarity of several features. In other words, there are many traces that, aside from timestamp and latency, are identical across all other features. Once the categorical properties of the spans are vectorized as count data, there are relatively few types of traces that occur repeatedly, thus the HST mass will accumulate within a small set of terminal nodes. To address this issue, we opted for the HS*T. In the rest of the study, we used the term HST and HS*T interchangeably.

We now present the optimizations that have been applied to HS*T.

Data Format

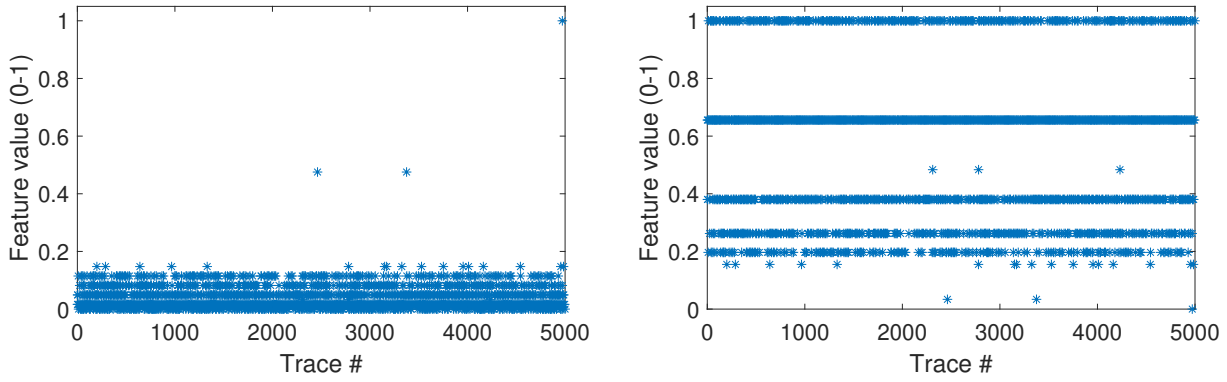
The ultimate goal of this trace representation is to see whether it is suited to a classification method for anomaly detection. It is common that some classification methods require the data to be scaled between [0,1]. Since our trace data is not scaled by default, we need to apply the min-max scaling technique to scale the data. This can be done simply using (5.1) to scale a data point x_d to $\hat{x}_d$ from dimension d.

$$\hat{x}_d = \frac{x_d - \min x_d}{\max x_d - \min x_d} \tag{5.1}$$

An issue with such count data scaling is that outliers can often cluster the normal values at one end of the range. This pattern in the data makes the prediction difficult particularly for a tree based method as it relies on randomized partitioning of the input space, i.e., random split points will be chosen in the segment [0,1] to branch the tree along a dimension. Therefore if the points are all clustered in a small portion of the range [0,1] the tree will struggle to separate the samples along that dimension.

For this reason, we define a novel transformation function to be used before applying the min-max scaling. This transformation is particularly useful for the count data corresponding to the production dataset. However, it can be applied to other datasets as well if it follows a similar pattern discussed above. We propose to apply the transformation function f as defined in (5.2)

$$f(x) = \frac{1}{1 + g(x)} \tag{5.2}$$



(a) Min-Max Scaling without f

(b) Min-Max Scaling with f

Figure 5.1: Comparing the scaled value of HTTP 200 code counts with and without applying the transformation function f

Here, in (5.2), we have found it sufficient to use $g(x) = x$ but we could also define, for example, $g(x) = \log(x)$ considering large values of x. This transformation f allows us to control the stratification of the count data by giving some flexibility in modulating how much a randomized classifier should resolve large outlier counts vs. normal data counts. This is because since $g(x)$ appears at the denominator, the large outliers will be squeezed near 0, therefore not suppressing the ability to resolve the normal values. We have seen in the production dataset that it mostly consist of low counts, making this transformation useful.

The impact of this transformation is clearly visible when we see the values of a particular feature. We consider the feature corresponding to the frequency of HTTP code 200 in the trace. We plotted the values of this feature (Fig. 5.1) with and without applying the transformation function for 10000 randomly chosen traces. From Fig. 5.1a, it is clearly visible that without the transformation the values are clustered at the lower end of the range. This is because one extreme outlier whose original value is 62 i.e. the HTTP code 200 appears 62 times in a trace. The 99^{th} percentile of the feature is only 8. Thus, upon applying min-max scaling it packed other values near the lower end of the range. However, if we apply f, the values are more evenly spread as seen in Fig. 5.1b. Although, one can argue that filtering the outlier solves the problem, it is not always that straightforward. For an instance, if the outlier occurs periodically at a slow rate, we may want to sample at in the beginning but not later. Thus filtering out large traces statically is not ideal.

Note that, since the transformation bounds the input to $(0, 1]$, we can choose not to apply min-max scaling. We also opted for this strategy for rest of the study.

Mass Accumulation

In a streaming environment, after the training phase, the HS*Ts should periodically update the mass values of the terminal nodes. This enables the trees to reflect the current observations. Tan *et al.* have suggested to use a window-based method for this periodic updates [TTL11]. In this approach, we first need to define a fixed length window (the length is defined by the number of observed traces). After that, for each terminal node, we need two variables, one to store the mass values of previous window and the other to update the mass of the current window. When a window finishes, we set the value of previous window to the value of the current one and reset the current values to 0. When we query the mass value for newly observed trace, the value of the previous window is returned.

The approach discussed above has two issues. Firstly, it requires to define a window length in terms of observations, whereas it is more natural to define it in terms of actual time periods e.g., hour, day, week. Secondly, since we are discarding the mass values of the previous window periodically, we might miss a long-term pattern in them. Thus, choosing a window length that can reflect such patterns can be a critical issue. To address these issues, we introduce the concept of time-based window and mass accumulation. Here, we define a window in terms of a time period. After that, as oppose to discarding the mass of the previous window, we *accumulate* mass in the HS*T terminal nodes. That is, when we update the HS*Ts, we do not erase the previously recorded values, but increase them according to the number of new traces falling in each branch of the tree.

Depending on how mass accumulation is implemented, we can further save the storage required by HS*T. If the mass values are updated only at the end of a window, we require an extra variable per terminal node. However, if we update this value after observing a trace, we will not require that extra variable. This may not sound much but consider a case where the HS*T implementation can not reduce the number of terminal nodes due to uniformly distributed

input data. As a result, for a tree of depth 15, we will have 2^{15} extra variables. Overall for a forest with q trees, we will require $q2^{15}$ variables. Thus, with this alternate implementation, we save the storage space for those $q2^{15}$ variables. It should be noted that, with this approach the mass values will continuously change within a window, which can impact the anomaly detection performance. Thus, rather than continuous update, we have opted for updating the mass values only at the end of each window.

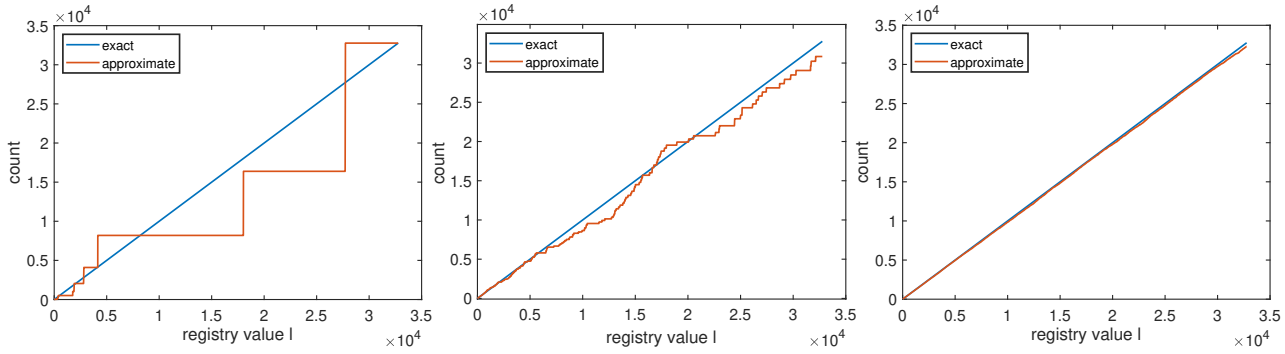
Approximate counting

A novel idea that we have explored to further reduce HS*T storage consumption is using *approximate counting* methods. The idea is that the mass values stored in the terminal nodes of HS*T can be replaced by approximate counters that take less space. To the best of our knowledge, this combination of approximate counters with HST has not been implemented in the literature.

Approximate counting methods are commonly used, for example, in high-speed network packet counting, to identify surges in traffic flows. The idea is to use less bits to represent a number. Since we are skipping some values of the counter, an approximate counting method needs to decide when the counter should be increased. This is usually done by introducing a probability that the counter should be increased. For example, if the register value l stands for an approximate counter value $A(l) = 2^l$, then we go from value $l = 2$ to value $l' = 3$ with probability $1/(8-4) = 1/4$. This is because it takes on 4 increments to go from $2^l = 4$ to $2^{l'} = 8$. Similarly, to increase $l' = 3$ to $l'' = 4$ we will use probability $1/(16-8) = 8$, and so forth.

The method we have discussed is the traditional approach for approximate counting, named Morris' algorithm [Mor78]. Here the increment probability from l to $l+1$ is $1/D(l)$, with $D(l) = A(l+1) - A(l)$ and $A(l) = 2^l$. The function $A(l)$ is called the *estimation function*, which represents the value of the counter associated to register value l. If the register ranges in $0, \ldots, L-1$, the maximum number that can be represented will be $A(L-1) = 2^{L-1}$.

While Morris' algorithm allows for significant memory savings, it is fairly coarse-grained in



(a) Morris' algo. 4-bit register (b) Estimtn. $\varepsilon = 0.10$ 9-bit register (c) Estimtn. $\varepsilon = 0.01$ 14-bit register

Figure 5.2: Approximate counting methods: counting from 0 to 2^{15}

the count values. Figure 5.2a shows an example run of Morris algorithm. By definition, the method takes as little as 4 bits to approximately represent numbers from 0 to 2^{15}. However, the approximation is increasingly worse the larger the numbers grow.

In recent years, new expressions for the $A(l)$ function have been proposed that trade accuracy for storage. In particular, the estimation function in (5.3) is provably optimal [EFK15].

$$A_\varepsilon(l) = \frac{(1+2\varepsilon^2)^l - 1}{2\varepsilon^2}(1+\varepsilon^2) \tag{5.3}$$

where ε is a parameter of the algorithm. The constant ε needs to ensure that $A_\varepsilon(L-1)$ exceeds the desired maximum counter value M. The smallest the value of ε, the better the approximation is, but the storage saving will be less. Examples for $\varepsilon = 0.10$ and $\varepsilon = 0.01$ are shown in figures 5.2b and 5.2c, which result in better approximation accuracy but smaller savings to 9 and 14 bits, out of the 15 necessary to represent the increments exactly.

The production dataset we consider contains 77577 traces. This means that the maximum mass value, in the worst case, a terminal node can have is also 77577. Thus, it will require 17 bits to represent this value. However, from the illustrations above, it is clear that with approximate counting we can represent similar mass values with high accuracy using 14 bits. If we want more compression, we can use 9 bits, which will also yield a close approximation but reduce the memory consumption by almost a half.

Performance after Modifications

We have applied the above discussed changes to an openly available code-base of HS*T [TZLT13]. The code-base is based on the Matlab programming language. We used this HS*T implementation as it is apart from two modifications. Firstly, as suggested by Tan *et al.* [TTL11], we opted for a $[0, 1]$ workspace. This means that, the maximum and minimum values of the features are assumed by the HS*T to be 1.0 and 0.0, rather than in the min-max range observed in the training dataset. We can set this range because the feature transformations we have used ($f(x) = 1/(1+x)$) lie in $(0, 1]$. This adjustment improves over the min-max estimate by already accounting for future values that are smaller (resp. larger) than the minimum (resp. maximum) observed in the training dataset. Secondly, we opted for depth-dependent split dimension. This means when splitting a node, instead of using the normal procedure of picking a dimension at random, we require all nodes at the same depth level to use the same split dimension. This change is motivated by the factor that in HS*T we can use a large depth value due to its low memory consumption. Since with a increase in depth the number of nodes grows exponentially, using the same split dimension at the same level ensures a balance in randomness.

As before, we used the production data to test these modifications. With the experiment, we particularly focused on this particular aspect: how does HST performs with and without approximate counting, when the other modifications are applied? This is because approximate counting will make the mass values less precise, it tends in general to reduce the F1 score of the anomaly detection method. The production trace consists of about 77577 observations, so that a terminal node in a tree in the worst case will require 17 bits to represent the mass value. Indeed, we have seen trees where terminal nodes had mass values over 77,000, thus requiring 17 bits. The results of the experiments are shown in Table 5.2.

From the table we see that, without approximate counting the F1-Score is around 0.97. This is an significant improvement from the number we have seen in Table 5.1, which is 0.8. As expected, with approximate counting, we see for larger values of ε, the F1-Score reduces. However, even there the lowest value is 0.90. This means we can get an F1-Score of 0.90 with approximate counting with 4 bits. However, this should not be expected in all the cases. Here

Table 5.2: Approximate counting results. All experiments use a HS*T with $\mathbf{T} = 500$ trees, depth $\mathbf{D} = 15$. We considered each day as a window and used the first day to built the trees.

ε	**bits**	**Precision**	**Recall**	**F1-Score**
exact	17	0.9522	0.9853	0.9669
0.01	14	0.9522	0.9853	0.9669
0.10	9	0.9525	0.9848	0.9667
0.25	7	0.9458	0.985	0.9632
0.5	5	0.9164	0.9895	0.9482
0.75	4	0.8336	0.9952	0.9003

we still achieve such high F1-Score because, the mass values are not uniform and can be easily segmented into a small number of groups. As long as the high mass groups can be separated from the low mass groups, we will have a high F1-Score. However, if the mass scores are more uniform, such coarse level of approximate counting will not yield good results. Overall, these results indicate that there is strong potential of compressing the space taken by HST with approximate counting, while not affecting the F1-Score.

For the rest of the study, we used the HST structure we discussed in this section. We now focus on how to use HST for sampling the traces.

5.4 Sampling with Storage Budget Constraints

As already mentioned earlier, when we perform the sampling, we have a storage budget constraint i.e. the percentage of traces we can save. In this section, we present how we can accommodate this budget in a sampling method based on HST.

5.4.1 Cluster based Sampling

Although the HS*Ts can help in classifying the anomalous traces, in reality we need to utilize this classification output in a sampling process. This process is not a straightforward one like always sample the anomalous traces because we often want to sample the normal traces as well. A logical approach could be grouping the traces and deciding whether to sample a trace or not based on its group association. This grouping can be done using unsupervised clustering

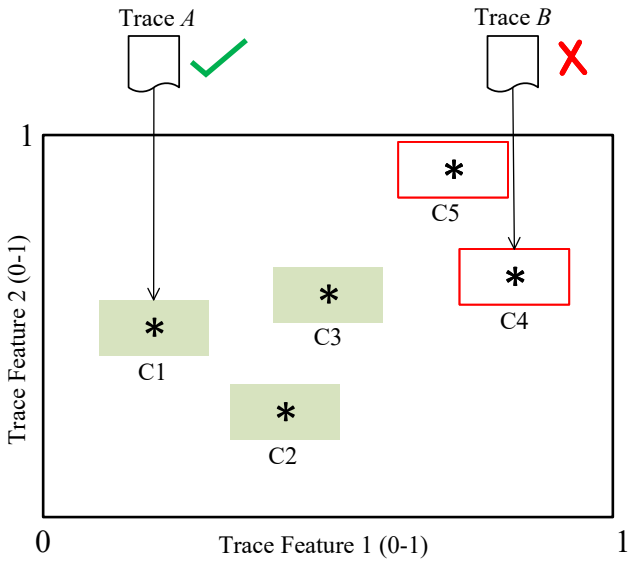


Figure 5.3: Illustration of trace sampling based on trace-cluster association. If the observed trace is associated with one of the prioritized clusters (the solid-green colored clusters), it is likely to be sampled.

algorithms. This concept is illustrated in Fig. 5.3. Here, the clusters represent different groups of the traces. The clusters that represent more rarely observed traces are prioritized for sampling *i.e.*, if a newly observed trace is associated with such clusters, it is likely to be sampled. This sampling mechanism needs to achieve two objectives: clustering the traces and making the sampling decision by prioritizing the clusters. Before achieving these objectives, we need to decide on the data format that will be used for clustering.

In principle, one may use the feature vector passed to HS*Ts to cluster the traces directly. However, when clustering in a high-dimensional space, in addition to the higher computational cost, it is harder to achieve accurate density estimation [Fuk13], which is an essential requirement for clustering accuracy. Therefore, we propose a new approach we have not found in the literature. We consider the distribution of mass across the trees in the HS*T forest and select in addition to the mean a low percentile of the mass score. Low percentiles are expected to significantly differ from the mean when there is at least a subset of trees in the forest that identifies the trace as an anomaly. Because we work on high dimensional spaces with hundreds of features, but the trees having a depth at most around 15 to 17, not all trees will be able to correctly identify an anomaly, therefore by looking at low percentiles we can try to recognize that a trace was flagged as an anomaly by some group of trees.

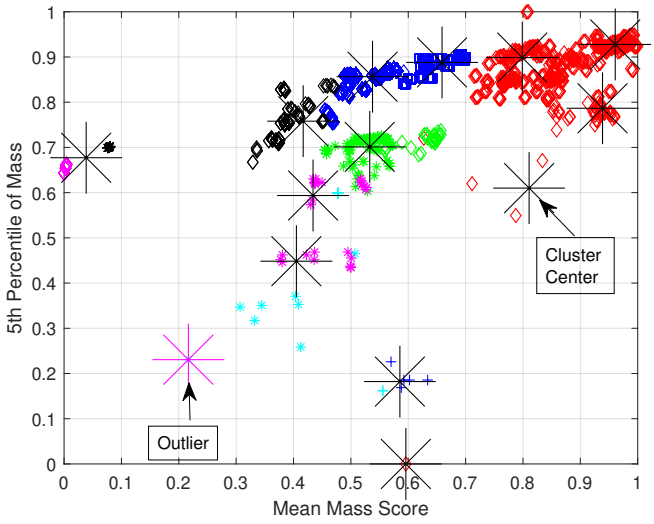


Figure 5.4: The production trace plotted using the mass-based properties. The colors and marker shapes indicate the DBSCAN original clusters. The cluster centers are estimated with a baseline online clustering method.

Based on the above discussion, we propose to use two mass-based properties for clustering: the mean mass score m and a percentile of the mass score p. Since we want to use a low percentile value, here we have considered the 5th percentile. So, we now represent a trace with a unique pair (m, p) that will be used for clustering. The projection of the productions trace in this 2-dimensional space is shown in Figure 5.4. The figure shows in different colors the clusters obtained by DBSCAN. It is clearly seen that the mass-based properties cluster the traces in distinct groups and the cluster centers are also appropriately detected using a baseline streaming clustering method [HAM17].

This mass-based clustering is at the core of our sampling approach. We call it *SampleHST* since it aims to sample from different trace groups that are generated by leveraging the mass-based properties of HSTs. We provide an overview of the approach in Fig. 5.5. The approach assumes that for each request, it will receive the spans related to the corresponding microservices. The trace aggregator will then compose them to form the traces. Subsequently, it will convert the trace to the BoW data format. As mentioned earlier, our assumption is that we know, in advance, the number of terms or words, *i.e.*, the number of dimensions in the data that will be clustered, since such knowledge can be obtained from the historical data. However, in cases where such information is unavailable, we can define a learning period for the trace aggregator where it will collect such information. This learning process can also be run periodically to

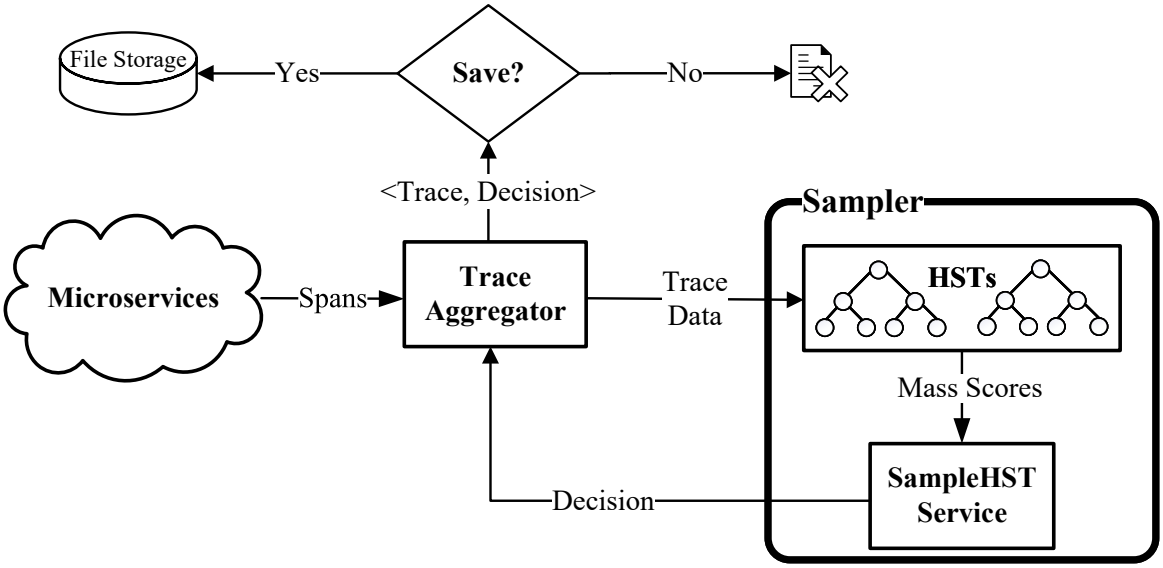


Figure 5.5: An overview of the SampleHST architecture

update the knowledge on the traces. Once the conversion is complete, the data is passed to the sampler component, which will decide whether to save the trace or not. As seen in the figure, the sampler component contains the HSTs and a clustering method namely *SampleHST Service* We already discussed about the HSTs. The SampleHST service works with the mass scores it receives from HSTs. It includes two components: SampleHST Clustering, which clusters the mass scores and SampleHST Controller, which controls the overall sampling mechanism and makes the sampling decision by determining which clusters should be prioritized. We now discuss these components in detail in the following sections.

5.4.2 SampleHST Clustering

SampleHST Clustering is based on the concepts of online clustering algorithm for streaming data [BA12, HAM17]. We particularly focus on utilizing the mean shift algorithm [FH75]. The benefit of this algorithm is we can consider the cluster centers as the mean of its data points. As a result, we do not require to store the previous data to perform online clustering. This is the foundation of SampleHST clustering but we further adapted it to fit with our trace data. We discuss these adaptions in the following sections.

Cluster Shape

A common assumption for online clustering algorithms for data streams is that the cluster shape is a hyper-sphere [BA12, HAM17]. However, if we apply a min-max scaling to the mass-scores, the data dimensions are bounded between 0 and 1. Thus using a hyper-sphere is counter intuitive since the clustering space is basically a hyper-cube. As the clustering method will split this space into further sub-spaces, it is intuitive to consider clusters that are either hyper-cubes or hyper-rectangles. This allows, in principle, a denser partitioning of the clustering space. However, we first want to investigate whether this change in cluster shape aligns with the concept of mean shift algorithm [BA12].

Mean shift algorithm falls into the class of gradient-based clustering methods [Fuk13]. The objective of such methods is to form clusters by identifying the valleys of the mixture density function. This is achieved by calculating the density gradient for each sample and moving them towards the direction of the gradient that is away from the valley. Over multiple iterations, this process should converge the samples in the cluster center. Since the actual density function is typically unknown, we consider its kernel density estimate [Ros56, Par62] using which the gradient of the samples are calculated. This kernel density estimation is based on a kernel function that provides the degree of similarity. The support of these kernel functions resemble to different shapes [LW19]. For a spherical function, the shift of the samples towards the gradient is the local mean [FH75]. This means that moving the samples towards the mean of their local region will eventually converge them to the cluster center, hence the name mean shift algorithm.

This mean shift property can be utilized in an online clustering method [BA12]. In an online setting, we receive one sample at a time and based on its cluster association, we need to update the cluster center. As we know that, for each sample, the shift towards the cluster center is the current mean of the cluster samples, we can consider the current mean value as an estimate of the cluster center. However, we have seen this property being used with spherical kernel functions. Since, we want to utilize the shape of our clustering space by using cubical clusters, we investigate whether the mean-shift analysis can be done with a kernel function

whose support is a hyper-cube. Here we consider an arithmetic average kernel whose support is a hyper-rectangle [LW19]. The kernel is multivariate, which is obtained by averaging univariate kernels. We present the kernel, considering we have d data dimensions, in (5.4). We show that the mean-shift analysis applies for this kernel if the clustering bandwidth [WJ94] is equal, thus forming a hyper-cube, in all dimension.

Theorem 5.1. *For the additive kernel defined as*

$$K_d(u_1, \dots u_d) = \begin{cases} \frac{3}{d2^{d+1}} \sum_{k=1}^{d} (1 - u_k^2) & \text{if } |u_k| < 1, \forall k \\ 0 & \text{otherwise,} \end{cases} \tag{5.4}$$

the mean-shift algorithm at each iteration shifts each sample with a value equal to the local mean if the support is given by a hyper-cube.

Proof. Assume that we have N observations $\{x_i\}_{i=1}^{N}$, with each data point having d dimensions, that is $x_i = (x_{i,1}, ..., x_{i,d})$, then we can define the density gradient estimate $\hat{\nabla} p(x) \equiv \nabla \hat{p}(x)$ at $x = (x_1, ..., x_d)$ as

$$\begin{aligned} \nabla \hat{p}(x) &= \frac{1}{Nh^d} \sum_{i=1}^{N} \nabla K \left(\frac{x_1 - x_{i,1}}{h}, \dots, \frac{x_d - x_{i,d}}{h} \right) \\ &= \frac{1}{Nh^d} \sum_{i=1}^{N} \sum_{k=1}^{d} \frac{\partial}{\partial x_k} K \left(\frac{x_1 - x_{i,1}}{h}, \dots, \frac{x_d - x_{i,d}}{h} \right) \cdot e_k \end{aligned} \tag{5.5}$$

where $\hat{p}(x)$ is the kernel density estimator [BA12, LW19] for an unknown density p, h is the bandwidth in all d dimension, $K(\cdot)$ is the kernel function and e_k is the k-th standard unit vector. Now using (5.4) as our kernel function, we get

$$\begin{aligned} &\sum_{k=1}^{d} \frac{\partial}{\partial x_k} K \left(\frac{x_1 - x_{i,1}}{h}, \dots, \frac{x_d - x_{i,d}}{h} \right) \cdot e_k \\ &= \sum_{k=1}^{d} \frac{\partial}{\partial x_k} \left[\frac{3}{d2^{d+1}} \sum_{l=1}^{d} \left(1 - \left(\frac{x_l - x_{i,l}}{h} \right)^2 \right) \right] \cdot e_k \\ &= \frac{3}{d2^{d+1}} \sum_{k=1}^{d} \frac{\partial}{\partial x_k} \left(1 - \left(\frac{x_k - x_{i,k}}{h} \right)^2 \right) \cdot e_k \\ &= \frac{3}{d2^d h^2} \sum_{k=1}^{d} (x_{i,k} - x_k) \cdot e_k \end{aligned} \tag{5.6}$$

Substituting $\nabla K(\cdot)$ in (5.5) with (5.6), the density gradient estimate is

$$\begin{aligned}\nabla\hat{p}(x) &= \frac{1}{Nh^d}\frac{3}{d2^dh^2}\sum_{i=1}^{N}\sum_{k=1}^{d}(x_{i,k}-x_k)\cdot e_k \\ &= \frac{N_x}{Nh^d}\frac{3}{d2^dh^2}\sum_{k=1}^{d}\frac{1}{N_x}\sum_{x_{i,k}\in S_r(x)}(x_{i,k}-x_k)\cdot e_k \\ &= \frac{N_x}{Nh^d}\frac{3}{d2^dh^2}\sum_{k=1}^{d}(\mu_k-x_k)\cdot e_k\end{aligned} \tag{5.7}$$

Here, in (5.7), N_x is the number of data points for a region $A_h(x)$, μ_k is the mean of all the data points in that region along the k^{th} dimension. Note that, the volume of the region $A_h(x)$ is h^d and the probability density estimate $\hat{p}(x)$ over the region using a uniform kernel is $\frac{N_x}{Nh^d}$.

The objective of the mean shift algorithm is to move away from the valley and towards the function mode. This can be achieved through gradient ascent. That is, for two consecutive iteration t and $t+1$ the shift in the variable x_j in k^{th} dimension can be expressed as

$$x_{j,k}^{t+1} = x_{j,k}^{t} + c\frac{\nabla\hat{p}(x)}{\hat{p}(x)} \tag{5.8}$$

Now substituting the value of $\hat{p}(x)$ and $\nabla\hat{p}(x)$ and considering $c = \frac{d2^dh^2}{3}$ in (5.8), we show in (5.9) that the shift is equal to the mean of x in dimension k. This mean that, from the online clustering perspective, with the arrival of each sample, the cluster center shifts u_k in dimension k.

$$x_{j,k}^{t+1} = x_{j,k}^{t} + \left(\mu_k - x_{j,k}^{t}\right) = \mu_k \tag{5.9}$$

Based on (5.9), we conclude that the mean shift algorithm is applicable to the kernel in (5.4) when the cluster bandwidth h is equal in all dimensions.

□

Cluster Creation

In *SampleHST* clustering we create a cluster when a trace cannot be assigned to any of the existing clusters, resulting in a new cluster consisting of only that trace. For spherical clusters, it is sufficient to only consider the radius of the cluster to assign a data point. However, since we are considering a cluster as a hyper-rectangle, we need to consider all of the d data dimensions separately. Thus, instead of defining a single radius value, we must define a bandwidth vector, $H = \{h \in \mathbb{R}^d | \forall i = 1, \ldots, d,\ 0 < h_i \leq 1\}$, that has the same dimension as a data point. Each value in the bandwidth vector $h_i \in H$ defines the Manhattan distance from the center to the boundary of the cluster in i^{th} dimension. If we define a vector of Manhattan distance (separately in each dimension) between a cluster centroid and a new data point as $M = \{m \in \mathbb{R}^d | \forall i = 1, \ldots, d, 0 \leq m_i \leq 1\}$, if $\forall i\ m_i \leq h_i$, we assign the data point to that cluster. In other words, if a data point falls within the bandwidth of a cluster centroid, it is assigned to that cluster. Otherwise, a new cluster is created.

Centroid Update

Updating the cluster center or centroid appropriately is critical to achieve high sampling accuracy since the centroid distance is used to decide the mapping of traces to clusters. A trivial strategy would be updating a cluster centroid whenever a new trace was assigned to that particular cluster. However, this policy can bring sharp changes to the cluster centroid due to some traces that fall just within the cluster bandwidth.

It is in general preferable to update the centroid giving more importance to traces that are unequivocally within that cluster, and so that they are in a smaller region around the centroid. This is the concept of *cluster kernel region* [HAM17]. Let the bandwidth vector of a cluster be specified by $\mathbf{H}$, which is a vector of the width of the rectangle around the centroid. We define the kernel region as the sub-rectangle with bandwidth $r\mathbf{H}$, where the scalar r quantifies the proportion of the cluster area considered as the kernel region. This is illustrated in Fig. 5.6. A common value of r seen in the literature is 0.5 [HAM17], which we also adopted for SampleHST.

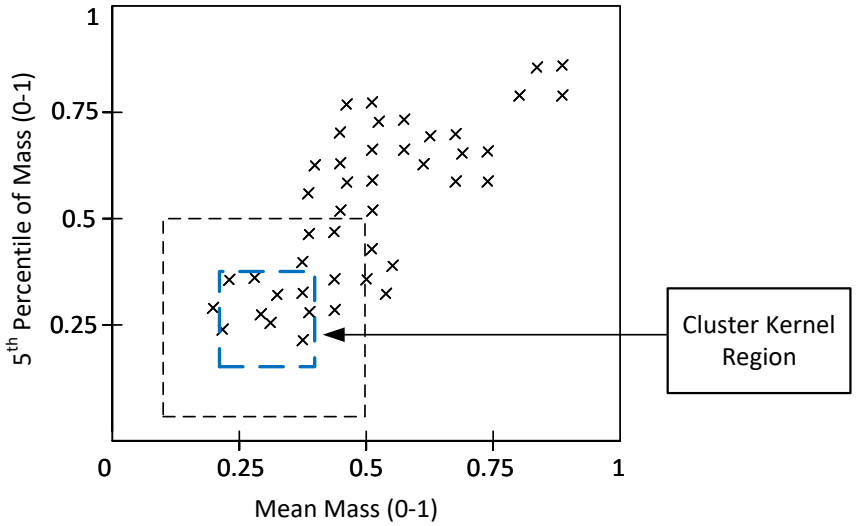


Figure 5.6: Illustrating the kernel region of the cluster

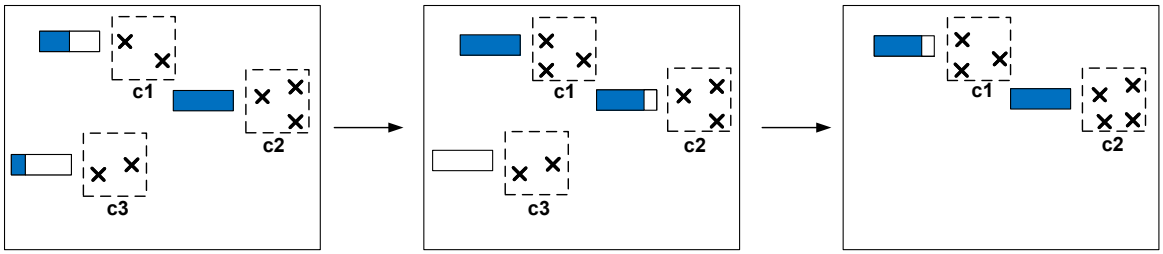


Figure 5.7: Demonstrating the cluster removal process. There are 3 clusters initially and each of them has a life property. Each time there is a new observation, the life of the corresponding cluster is reset to full value, while the life of other clusters are reduced. A cluster is removed once its life ends. We can see that cluster `c3` is thus removed.

Cluster Freshness

Since we are dealing with streaming data, we must consider cluster freshness. We define cluster freshness as a cluster property that indicates whether a cluster population (number of observed traces in that cluster) remained static for a particular amount of time. This is important because the clusters that are static for long periods are unlikely to be relevant and affect the sampling policy. For example, let us consider a sampling policy which aim to sample a trace if it falls in one of the *bottom-k* clusters (in terms of size). Now if a cluster with a small population remained static for a while, it can become one of the *bottom-k* clusters. This will prioritize that cluster for sampling. Nevertheless, we cannot sample from that cluster unless a new trace associated with that cluster is observed. In addition, the cluster will continue to ignore the $(k+1)^{\text{th}}$ cluster from the bottom for sampling. This will ultimately affect the sampling

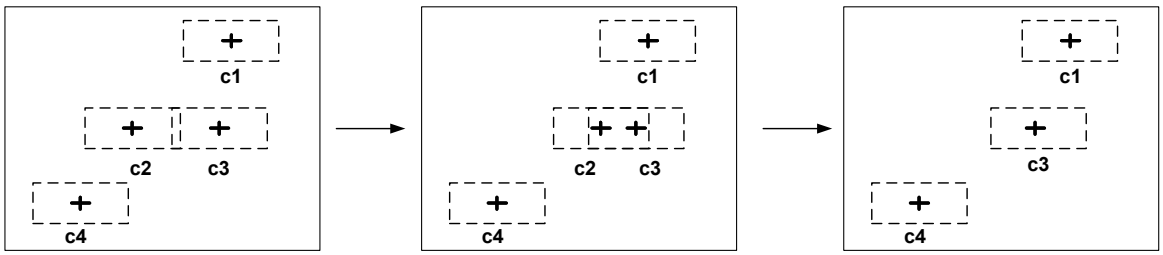


Figure 5.8: Demonstrating cluster merging process. Initially, though there is an overlap between the boundary of cluster `c2` and `c3` they are not merged. Once there centroid overlaps, they are merged into single cluster `c3`

performance. To avoid this, we need to regularly remove the clusters whose population have remained static for a while.

Typically this cluster removal is done using a *decay* and *life* value [HAM17]. We start with a cluster with its *life* set to 1 and gradually reduce it. Once *life* is ≤ 0, we remove the cluster. Reducing the life is an event-driven process. Upon receiving a new sample, we just need to set the *life* of its associated cluster to 1 and reduce the life of the other clusters by $\frac{1}{\text{decay}}$. The *decay* value can be set as the average number of traces in the work cycles within the sampling window. We can define a work cycle as a sequence of consecutive periods where we received at least 1 trace. The intuition behind this work cycle is, we wait at least for a period, while receiving traces, to see whether a cluster receives a new trace. If a trace is not received, the cluster is considered to be irrelevant and thus removed. The overall cluster removal process is illustrated in Fig. 5.7.

Cluster Merging

During online clustering, it is likely to observe overlaps among the clusters. Since we forget the previous data points, because each cluster is summarized by its center and bandwidth vector, it is not possible to adjust the cluster boundaries based on the density of the data. However, we need to address these overlaps as they are indications of possibly inaccurate clustering. Inaccurate clustering can affect the sampling performance.

In such scenario, we can merge the overlapping clusters. However, the merging decision can taken based on different policies with different definitions of overlapping:

Algorithm 5.1 SampleHST Clustering

Input: *trace*, *clusters*, *bandwidth*, *decay*
Output: *clusters*, *traceLocality*

```
1: if clusters = ∅ then
2:    {clusters, traceLocality} ← SpawnCluster(clusters, trace)
3: else
4:    clusters ← PruneClusters(clusters)
5:    initAssignIndex ← FindAssociation(trace, clusters, bandwidth)
6:    if initAssignIndex = -1 then
7:       {clusters, traceLocality} ← SpawnCluster(clusters, trace)
8:    else
9:       clusters ← AssignToCluster(trace, clusters, initAssignIndex)
10:      mergeIndex ← FindMergeCandidate(clusters, bandwidth, initAssignIndex)
11:      if mergeIndex > −1 then
12:         clusters ← MergeClusters(clusters, mergeIndex, initAssignIndex)
13:         if mergeIndex > initAssignIndex then
14:            traceLocality ← mergeIndex − 1
15:         else
16:            traceLocality ← mergeIndex
17:         end if
18:      else
19:         traceLocality ← initAssignIndex
20:      end if
21:   end if
22:   clusters ← DecayClusters(clusters, decay)
23: end if
```

- We can merge two clusters when simply their boundary overlaps [BA12]. Although simple, such an approach has an stability issue. Using such definition of overlapping can drastically change the clustering pattern as it allows to merge two clusters only due to the arrival of a new distant data point in one of the clusters.

- Another policy could be merging two clusters only when the centroid of one overlaps with the boundary of the other. It is easy to see why such a policy is less drastic because one distant point cannot shift the cluster center unless the cluster has a very few samples.

In this study, we have opted for the latter policy, which is demonstrated in Fig. 5.8.

Algorithm

We now discuss the overall algorithm for *SampleHST* clustering. It is presented in Algorithm

Table 5.3: Auxiliary functions used in Algorithm 5.1

SpawnCluster	Creates a new cluster with the current trace data. The details are provided in section **Cluster Creation**.
PruneClusters	Removes the clusters whose life have expired. The details are provided in section **Cluster Freshness**.
FindAssociation	Finds the cluster where the current trace can be assigned. Returns -1 if no cluster is found. The details are provided in section **Cluster Creation**.
AssignToCluster	Assigns the current trace to a cluster and update its center if necessary. The details are provided in section **Cluster Creation** and **Centroid Update**.
FindMergeCandidate	Finds whether the cluster associated with the current trace can be merged with any other cluster. Returns -1 if no cluster is found. The details are provided in section **Cluster Merging**.
MergeClusters	Merges two clusters. The cluster centroid and its population is duly updated. Total number of clusters is reduced by 1. The details are provided in section **Cluster Merging**.
DecayClusters	Reduces the life of each cluster by $\frac{1}{decay}$. The details are provided in section **Cluster Freshness**.

5.1. The functions that have been used in the algorithm is summarized in Table 5.3. The method initially creates a new cluster with first trace. After that, for every newly received trace, the following steps are taken. Firstly, the method aims to remove the clusters that are not fresh. Secondly, the method aims to find the locality of the newly received trace. If no such cluster is found, the method returns the value -1 to indicate that this trace is an outlier. If SampleHST clustering receives -1, it creates a new cluster and also returns -1 as the cluster index to indicate a new cluster creation. Otherwise, the trace is assigned to the corresponding cluster. Thirdly, SampleHST clustering checks whether the current cluster, where the trace is assigned, overlaps with the other clusters. If so, it merges the clusters. It first checks whether the centroid of the current cluster overlaps with the boundaries of the other clusters. If such clusters are found, the current cluster is merged with the one with minimum distance. Finally, the reduces the life of each cluster using the *decay* parameter.

5.4.3 SampleHST Controller

The SampleHST controller ultimately decides whether to sample a trace or not by utilizing the clustering algorithm. It starts the process by receiving the mass scores from HST. After that it performs a series of steps. We discuss these steps in this section.

Sampling Limit

The SampleHST controller initially calculates the sampling limit for the next sequence of j traces. We refer to this sequence as a window. For a given budget τ, the sampling limit is defined as τw_s where w_s is the window size. Since, the budget is pre-defined, we only require the window size to calculate the sampling limit. Defining this window size is critical for the sampling accuracy. Consider these two cases with 5% budget. In the first case the window size is 100. So we can at most sample 5 traces per 100. In the second case, the window size is 1000 meaning we can sample 50 per 1000. From percentage perspective this two cases are the same, but we have to consider that anomalies may not be well spread across time and appear in clusters. Due to this, if 10 anomalies appear in a window of size 100, with a 5% budget, we can at most sample 5 out of those 10 anomalies. This will affect the sampling accuracy. Considering this, we define this window in terms of actual time period, which we refer to as *primary window*. This allows to use a natural period as a window rather than fine-tuning it in terms of trace observations.

While setting the sampling limit, we consider the average *primary window* size, i.e. the average number of traces observed in the window, as an estimate of the sampling window size. The limit is then calculated using the budget. Once a window is completed (end of a day or an hour), the sampling window size is updated again and the limit is re-calculated. For the first window, we use a parameter that provides an expected value of the primary window size. This value can be easily obtained from the historical data since the primary window is defined with actual time periods.

A issue with the above approach is we can observe more traces than expected in a sampling window. This means that the sampling limit needs to adjusted. We can do this by considering a *secondary window* period, which is smaller than the primary one (e.g. if the primary window is a day, the secondary window could be an hour). We can use the average number of traces seen in the secondary window and adjust the sampling limit with that value. Note that, in a day there are 24 hours but we may not receive traces in all hours. Thus in calculating the average, we only consider the hours where we received at least one trace.

Online Min-Max Scaling

The SampleHST clustering method uses the mean and the 5^{th} percentile of the mass to cluster the traces. Since we are using HS*T, we use the mass value $m[l]2^l$ where $m[l]$ is the mass of the terminal node where the trace falls into and l is the depth of that tree node. To standardize the HS*T mass scores, we scale down the augmented mass using the maximum mass value possible. This maximum mass value is defined as $w2^d$ where d is the tree depth and w is the number of observed traces.

The clustering algorithm works on the logarithm of these standardized mass scores. This is to cluster together, more naturally, traces that are in the same order of magnitude for their masses. The clustering algorithm also requires the data to be normalized between $[0, 1]$, which can be achieved using min-max scaling. This is performed by the controller. It starts with a zero knowledge about these min-max values and update them on the fly. Once the min-max values are updated, we need to re-scale the cluster centroids. Otherwise, the cluster centroids will not be consistent with future shifts in the mass scores. To re-scale the centroids we need to store the previous min-max values. When the min-max values are updated, the previous scaling is first inverted using the previous min-max values and then the centroids are scaled again using the current ones.

Distance-based Cluster Ranking

While sampling the traces, we need to find out where its associated cluster is positioned based on a ranking criteria. This ranking process is critical for accurately sampling interesting traces. One option could be to use the cluster size as the ranking measure. Ideally, a small sized cluster should include the less frequent traces. However, due to the nature of SampleHST clustering, that creates and deletes clusters online, the small sized clusters may not always include the less frequent traces.

Consider the `Cluster 1` in Fig. 5.9. It has only 4 traces associated with it but all of them have high mass values. This means that the trace has hit HST nodes with a high mass count.

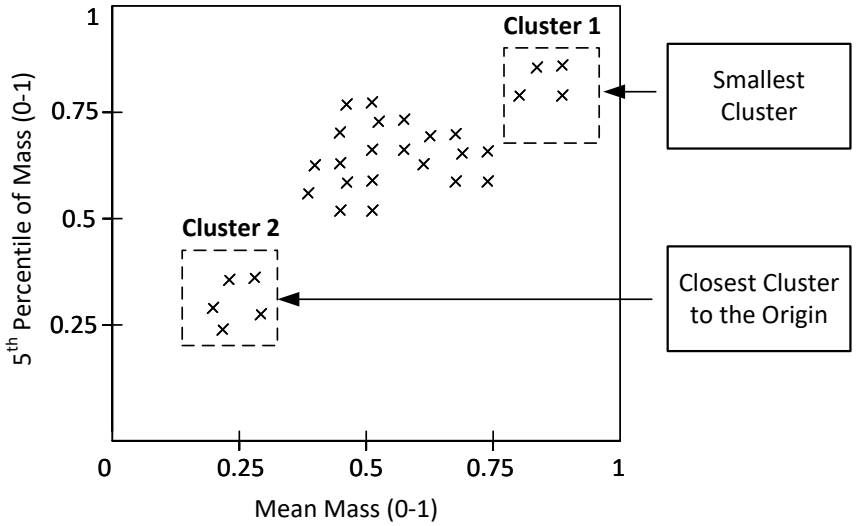


Figure 5.9: Illustrating the distance based cluster ranking concept

This indicate that these traces are quite frequent. However, if we prioritize this cluster for sampling based on size, we would contradict with the anomaly detection principle of HST - high mass value indicates frequent observations. Such scenarios occur when a "decayed" cluster is removed and after an interval, we again receive traces that are close to the removed cluster centroid. Eventually, that cluster will grow in size and not be sampled from but it will affect the initial sampling performance.

To counter this issue, we have decided to use a new metric to rank the clusters. We use the distance of the cluster centroid from the origin $(0, 0)$ to rank the clusters. Thus, if a cluster is closest to the origin, as shown in Fig. 5.9, traces associated with it will be sampled first even if that cluster is not smallest. The distance metric resolves the potential disagreement between the HST and SampleHST clustering. Although this metric can ignore the smallest cluster at some point, that smallest cluster is more likely to become one of the larger ones. Ultimately, the most interesting and smallest clusters are likely to form near the origin. The distance metric will facilitate to sample traces associated with those closest clusters.

Sampling Decision

The sampling decision procedure needs to decide whether to sample a trace or not instantaneously. Here, the challenge is to convert the clustering outcome to a sampling decision. For

that, we have designed a rule-based system. The first rule is a simple one. At first, we check whether a new cluster is created (indicated by -1 from the SampleHST clustering algorithm). In such cases, the method always sends a positive response for sampling. For cases where the trace is associated with an existing cluster, we apply the second rule. This rule relies on generating a pool of clusters, which we refer as *selection pool*. The process of generating this *selection pool* has two phases, which are as follows.

The first phase of generating the selection pool begins by ranking the clusters based on their distance from the origin. Once the clusters are ranked, we need to decide how many of those we consider in our selection pool. We initially select the cluster that is closest to the origin. After that, we gradually add the clusters until the population of the selected clusters reaches a threshold. The threshold is set using parameter that defines the percentage of the total population (θ) allowed in the selection pool. While adding the clusters in the selection pool, we impose two restrictions. The first one deals with the case where we still have not reached the threshold and can add another cluster. Here we only add a cluster if by adding it the selection pool, its population reaches at most two times the threshold. This ensures that the selection pool population does not get significantly larger than the expectation. The second case involves checking whether the cluster center falls in a region of high mass, because such region indicates that the cluster corresponds to more common traces. We can define a parameter to define a segment in the clustering hyper-plane that corresponds to the high mass region. It is easy to understand that the high mass region is the top-right corner of plane as that part corresponds to high mass scores. For example, as illustrated in Fig. 5.10, we can simply divide the 2D clustering plane to 3×3 segments and consider the top-right segment as a region of high mass. It should be noted that, we can also define regions that represent rare traces and decide to sample a trace a if it is associated with a cluster in those regions.

After creating the initial selection pool, we start the second phase by checking the actual value of the percentage total population in the selection pool. We denote this with $\hat{\theta}$. Ideally it will be θ, but it is not guaranteed because of multiple reasons e.g. the distribution of cluster size or the imposed restrictions discussed above. If the actual percentage is at least not half of the budget, we add more clusters in the selection pool. This also covers the scenario where

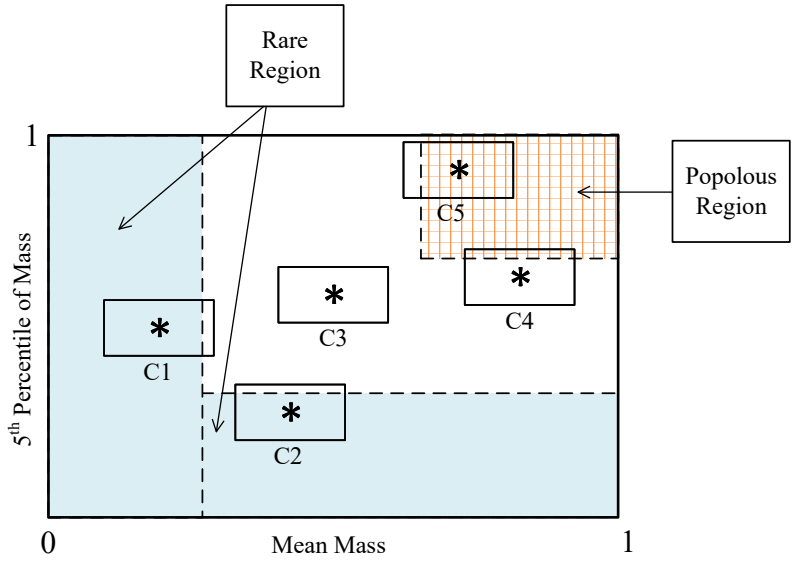


Figure 5.10: Illustrating the rare and populous region of the clustering plane for trace sampling

the budget is much higher than θ. This addition process is as follows. The clusters are added depending of the magnitude of the budget (τ) in comparison to ($\hat{\theta}$). We define the magnitude M as $\left\lfloor (\tau - \hat{\theta})/\hat{\theta} + \frac{1}{2} \right\rfloor$. We then make M attempts to add the clusters in a probabilistic manner, where in the k^{th} attempt, the k^{th} nearest cluster from the origin (which is not already included in the selection pool) is chosen with a probability P^k. The intuition behind this selection probability P^k is that we consider each attempt to add a cluster as an independent event, where each event has the same probability P. We define $P = \max(\tau, S)$ where τ is the budget and S is the sampling eagerness. Sampling eagerness is defined in (5.10).

$$S = R(1 - U) \tag{5.10}$$

The value sampling eagerness is bounded between $[0, 1]$ and a high value indicates we need to sample more. It is defined in terms of budget utilization (U) and the relative trace position in the current window (R). Budget utilization is simply the ratio of number of traces sampled so far to the sampling limit. Relative trace position is the ratio of the trace index in current window to the sampling window size. The intuition behind eagerness is as follows. The more closer we are to the end of sampling window, if the budget is less utilized, we should sample more. We defined the selection probability as the maximum of budget and eagerness. Here, the budget is also considered because, for a very high budget, the eagerness can be smaller than it.

In those cases, the selection probability should be guided by the budget.

Upon deciding the selection pool, we see whether the new trace is associated with any of the clusters in the pool. Now if it is not present, we decide not to sample it. We can also add special rules like if the associated cluster of the trace falls in a specific region, we will sample it regardless. Currently, we define a region around the bottom left corner of the clustering plane and if the associated cluster center falls in that region, the trace is sampled. On the other hand, if the trace is associated with the *selection pool*, we choose either of the following two paths. We follow the first path when the budget is greater than or equal to the actual percentage of population in the selection pool ($\tau \geq \hat{\theta}$). In that case, we sample the associated trace straightaway. Conversely, if the budget is less than the actual percentage, we follow the second path that takes a probabilistic sampling decision. This is to sample cautiously as we may have larger clusters in the selection pool containing common traces. In this path, we set the probability of sampling (P_s) using (5.11).

$$P_s = \begin{cases} \frac{\tau}{\hat{\theta}} & \text{if } \Gamma_c > \Gamma_\mu + k\Gamma_\sigma \\ 1 & \text{otherwise} \end{cases} \tag{5.11}$$

Here, in (5.11), we set the probability based on the cluster size. Firstly, if the size of the cluster (Γ_c), which is associated with the current trace, is greater than the sum of mean (Γ_μ) and k standard deviation (Γ_{sigma}) of the cluster size in selection pool, we set the sampling probability to $\tau/\hat{\theta}$. This means that, if there are N traces, the size of the selection pool will be $N\hat{\theta}$ and we would like to sample $N\tau$ traces from those in the selection pool. Secondly, if $\Gamma_c \leq \Gamma_\mu + k\Gamma_\sigma$, we set the sampling probability to 1. This means that if the cluster is sufficiently small, we decide to sample the corresponding trace. The value of k is set using Chebyshev's inequality [Fel08]. It allows to determine the minimum percentage of values within k standard deviation of the mean. For a given percentage (V), we can solve the inequality to determine the value of k. We notice that, this percentage V is related to the ratio of $\tau/\hat{\theta}$. Because, if τ is much smaller than $\hat{\theta}$, we want to sample only if the associated cluster is smaller than the majority of the clusters. As the value of τ increases compared to $\hat{\theta}$, we can consider the larger clusters i.e. larger value

Algorithm 5.2 Sampling Decision

Input: $massScores$, $budget$, $idxPriWindow$, $windowSize$, $remainingTarget$, $windowTarget$, $clusters$, $bandwidth$, $decay$
Output: $decision$

if $idxPriWindow > windowSize$ **then**
  AdjustParameters()
end if
$logMass = \log(massScores)$
$scaledScore = \text{ScaleScores}(logMass)$
if HasMaxMinChanged() **then**
  ReScaleClusterCenters()
end if
$(clusters, traceLocality) \leftarrow \text{GetTraceLocality}(scaledScore, clusters, bandwidth, decay)$
$relativePos = \frac{idxPriWindow}{windowSize}$
$budgetUtil = \frac{windowTarget - remainingTarget}{windowTarget}$
if $remainingTarget > 0$ **then**
  $decision = \text{IsTraceInSelectionPool}(clusters, traceLocality, budget, relativePos, budgetUtil)$
end if
if $decision$ **then**
  $remainingTarget = remainingTarget - 1$
end if

of V. Thus, we consider $\hat{V} = \tau/\hat{\theta}$, where $\hat{V}$ is and estimate of the minimum percentage V, we can calculate the value of k using (5.12).

$$k = \sqrt{\frac{1}{1 - \frac{\tau}{\hat{\theta}}}} \equiv \sqrt{\frac{\hat{\theta}}{\hat{\theta} - \tau}} \tag{5.12}$$

We now present the sampling decision taking process in Algorithm 5.2. The functions that we have used in the algorithm is summarized in Table 5.4. The process begins by receiving the mass scores. Initially, necessary adjustments are made to the sampling window size estimate and the sampling limit. After that, a series of data pre-processing is performed. At first, the logarithm of the mass scores are scaled using the min-max value. If the current scores change the minimum or maximum value, the cluster centers are also re-scaled and the min-max values are updated accordingly. Once this pre-processing part is completed, the received data is clustered by the SampleHST clustering method. Finally, if the sampling limit is not reached, a decision is taken based on the inclusion of the trace in the *selection pool*. This step is skipped if we have already reached the sampling limit.

Table 5.4: Auxiliary functions used in Algorithm 5.2

AdjustParameters	Adjusts the sampling limit if the current window size is larger than the estimation. The details are provided in section **Sampling Limit**.
ScaleScores	Scales the mass scores using the min-max values. If the min-max value changes raises a flag. The details are provided in section **Online Min-Max Scaling**.
HasMaxMinChanged	Check whether a flag is raised from the ScaleScores function. The details are provided in section **Online Min-Max Scaling**.
ReScaleClusterCenters	Re-scale the cluster centers using the previous and current min-max values. The details are provided in section **Online Min-Max Scaling**.
GetTraceLocality	Use Algorithm 5.1 to return the cluster index of current mass score.
IsTraceInSelectionPool	Determines whether the trace falls in any of the clusters in the *selection pool*. The details are provided in section **Sampling Decision** and **Distance-based Cluster Ranking**.

5.5 Evaluation considering Point Anomalies

We have implemented the *SampleHST* tool chain using the Matlab programming language. In this section, we present how it performs on the production data. We initially present the experimental configuration. After that we discuss the results of these experiments. Note that, the production data contains the point anomalies. Extension for the collective anomalies are presented in the next section.

5.5.1 Experimental Setup

For the experiments, we have considered five cases, each with a different storage budget. Since roughly 5% of the traces are anomalous in the production data, we include a case where the budget is also 5%. The evaluation criteria for this case is the F1-Score. We have also chosen three smaller budget cases of 0.5%, 1% and 2%. For all these cases, the evaluation criteria is the precision. Finally, we also consider a high budget case of 10% where the evaluation criteria is the recall. We have compared the results with two other trace samplers: uniform random and PERCH-based method. The uniform random sampler is in-principle same as a head-based sampler. This means that it samples the traces with a probability that is the same as the budget. We use this method as a baseline to demonstrate how the other sampling methods

improves the sampling performance.

Since recent sampling methods [LCMGF18, LCPAM19] focused on representative sampling, we also want to compare the selected methods on this aspect using a numeric metric. Furthermore, such a metric provides an additional dimension for comparing the sampling performance, in particular for high budget cases as in such cases, we can actually consider both normal and anomalous trace groups for sampling. In this study, we have used the Jain index [JCH84] to measure the degree of representative sampling. The index, bounded between $[0, 1]$, is a measure of fairness that indicates the "equality" while resource allocation *i.e.*, if all the groups receives an equal resource, the index value is 1. In our case, the groups are the clusters that we obtain from DBSCAN clustering and resource is the quota we allocate from the budget to each group. Thus, if we sample from these groups in a balanced way, regardless of the group size, we will achieve a high fairness index. The index can be calculated using (5.13) where $X_i = \frac{T_i}{O_i}$. Here, for each cluster i, T_i is the number of traces sampled by an method and O_i is the optimal number of traces that should be sampled. Note that, since we know the overall distribution of the traces among the groups and sampling budget, we can easily calculate O_i using the max-min fair allocation approach [Jaf81]. This is not the case for the sampling methods, since they do not know the final trace distribution.

$$\mathcal{J}(X_1, X_2, \ldots, X_n) = \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n \sum_{i=1}^{n} X_i^2} \quad X_i \geq 0 \tag{5.13}$$

We want to explore the experimental results on the basis of three research questions. These are as follows.

- **RQ1** What is the sampling performance of the SampleHST method, with and without approximate counting?
- **RQ2** How does SampleHST perform when we consider the cluster shapes as hyper-rectangles instead of hyper-cubes?
- **RQ3** How does SampleHST performs compared to other sampling approaches?

5.5.2 Results

Before discussing the research questions, to provide an insight of how the SampleHST clustering method works, we illustrate its output in Fig. 5.11. Since, this is an online clustering method, we divided the total time frame in 20 periods and show the clustering status for those periods. We immediately see that in the first window, the data points are less segregated. This is because of the online min-max scaling. In the initial period, the min-max values are not steady, which affects the data points as well. As we progress towards the end, we can see that the clusters are nicely segregated.

We also see that the number of clusters continue to change throughout these periods. The clusters around the top right corner remains stable, but the ones around the bottom left corner, changes the positions frequently. This is because, the top right clusters are of frequent traces whereas the bottom left ones are of the infrequent ones. The infrequent trace clusters are not stable because they decay quickly due to not receiving traces in some work cycles. The sampling performance of SampleHST depends on how these frequent and infrequent trace clusters are prioritized during the sampling.

We now focus on the results for **RQ1** in Table 5.5. Here we initially consider a clustering bandwidth of $h = 0.1$ and then also consider two other bandwidths which are 10% smaller and larger than 0.1 respectively. This is to observe the impact of bandwidth on sampling performance. We can see, for a bandwidth of $h = 0.11$, the sampling performance for all budgets are considerably higher with values ≥ 0.86. With low budget cases, this bandwidth provides higher precision compared to other as the traces can be grouped more precisely. However, when we need to consider both precision and recall, like the case with 5% budget, we see the bandwidth $h = 0.1$ yields better F1-Score. This is because, here with this bandwidth we can sample more anomalous traces by slightly compensating the precision, resulting in a better F1-Score. However, we see in all the cases the deviation in score is ≤ 0.05, indicating a robust performance yielded by SampleHST. When we change the mass update mechanism to approximate from exact, we see no significant difference as shown in Table 5.6. This is because the scores produced by HST are qualitatively similar in both cases, which can be seen from

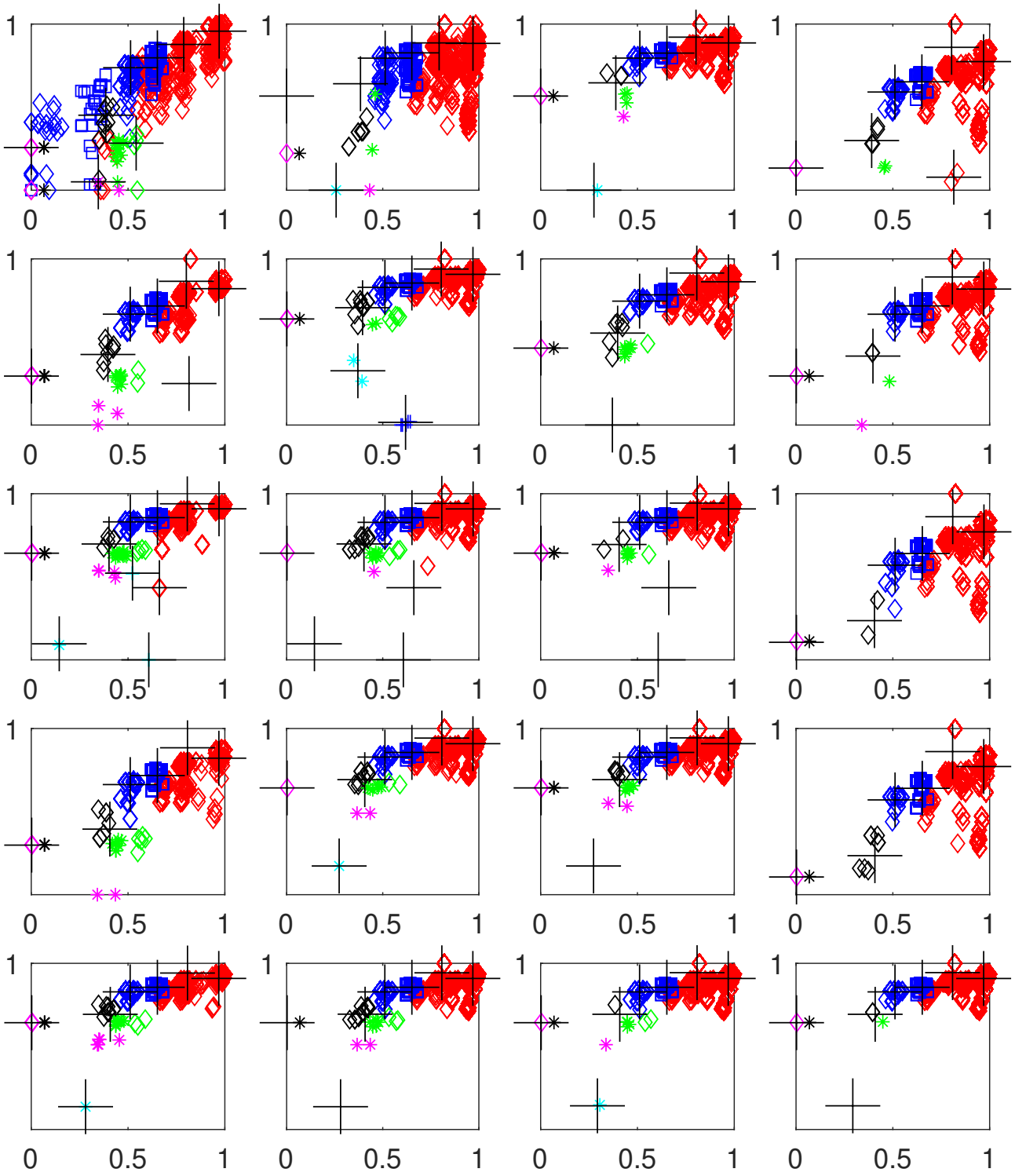


Figure 5.11: Illustrating the output of the SampleHST clustering algorithm. The X-axis and Y-axis represent mean and 5^{th} percentile of mass respectively. The colored symbols represent different DBSCAN labels. The + signs are the cluster centers estimated by the SampleHST clustering algorithm. The output is presented in 20 windows. As we move from left to right, we move towards the next window.

Fig. 5.12. As we discussed in section 5.3.3, although approximate counting introduces error in mass values, the normal and anomalous traces can still be separated. This is because of the

Table 5.5: Performance of SampleHST on the Production data with different clustering bandwidths. The score for $0.5\% - 2\%$ budget is the value of precision, for 5% budget it is the F1-Score and for 10% budget it is the recall.

Budget	**Score (0-1)**		
	$h = 0.1$	$h = 0.09$	$h = 0.11$
0.5%	0.84	0.83	0.87
1%	0.83	0.81	0.86
2%	0.86	0.82	0.88
5%	0.92	0.89	0.90
10%	0.94	0.95	0.92

Table 5.6: Sampling results for different budget on the Production data with exact and approximate counting

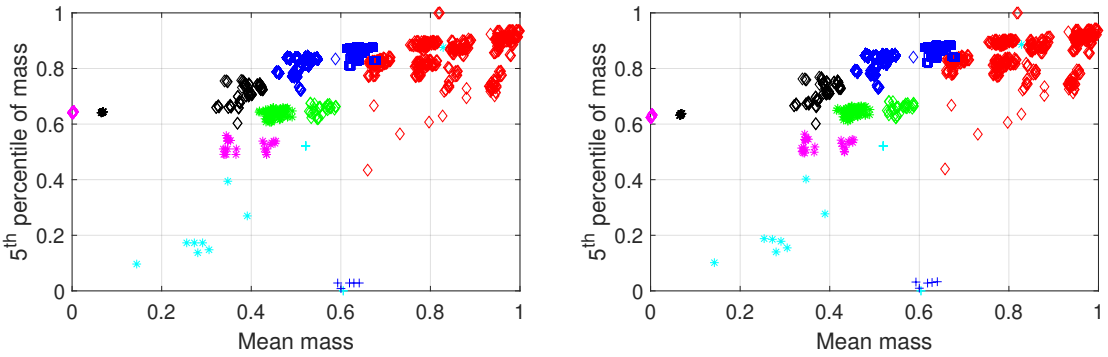
Budget	**Exact Counting**				**Approximate Counting**			
	J	**P**	**R**	**F1**	**J**	**P**	**R**	**F1**
0.5%	0.404	0.840	0.093	0.167	0.424	0.829	0.091	0.164
1%	0.590	0.832	0.182	0.298	0.594	0.827	0.180	0.296
2%	0.723	0.860	0.372	0.520	0.726	0.860	0.372	0.520
5%	0.750	0.924	0.907	0.915	0.750	0.924	0.905	0.914
10%	0.88	0.807	0.943	0.869	0.879	0.808	0.941	0.869

Legend: J - Jain Index, P - Precision, R - Recall, F1 - F1-Score

definition of mass score $m[l]2^l$, where $m[l]$ is the mass of the corresponding node, which is in level l. Due to approximate counting, the error propagates through the term $m[l]$ but the 2^l part balances out the error, particularly in the case where the value of l is high.

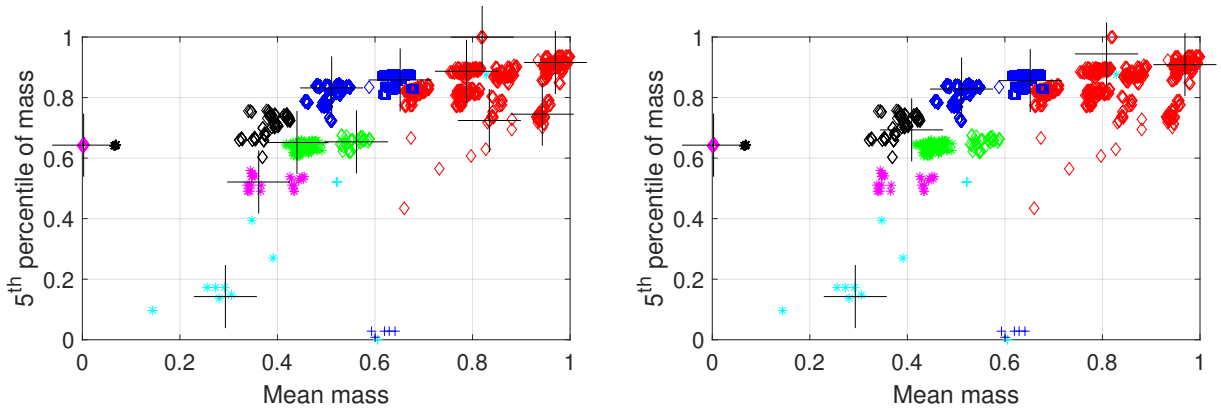
The mass scores work as anomaly signals to the SampleHST, which are not always likely to be equally strong in all clustering dimensions. In such cases, the traces may not be segregated ideally in that dimension. Thus rather than using a small clustering bandwidth in that dimension, as illustrated in Fig. 5.13, we can chose a large one to remove clusters containing similar traces, allowing a more precise clustering. In other words, we can opt for hyper-rectangles, with unequal clustering bandwidths in each dimension, instead of hyper-cubes. The aim of **RQ2** is to evaluate the performance in such cases. We present the results, for 5% budget case, in Table 5.7. Indeed, we observe that the F1-Score for bandwidth $[0.1, 0.3]$ reaches 0.95, which is higher than the one we achieved for hyper-cubes presented in Table 5.5. For the latter experiments with the production data, we have used this clustering bandwidth of $[0.1, 0.3]$.

The results corresponding to **RQ3** is presented in Table 5.8. We see that SampleHST is by far the best method across all budgets. The random sampler performs the worst as expected. We also see that the PERCH-based method does not perform much better from the perspective



(a) Exact Counting (b) Approximate Counting

Figure 5.12: Comparing the mass scores, with exact and approximate counting, for the production data. From a clustering perspective, both the scores are qualitatively similar.



(a) $h_1 = h_2$ (b) $h_1 \neq h_2$

Figure 5.13: Comparing the clusters generated by SampleHST with equal and unequal clustering bandwidth. With the unequal bandwidth, *i.e.*, hyper-rectangles, more precise clusters are generated.

of precision, recall and F1-Score. From the fairness perspective, the PERCH-based method scores much higher than the random sampler but still it can not outperform SampleHST. In the lowest budget case of 0.5%, the Jain score produced by the PERCH-based method is close to SampleHST. However, with the increase in budget, the gap across all the metrics becomes significant. The results show that even though the PERCH-based method can achieve better Jain score in low budgets, it is not precise in sampling the anomalous traces which is evident from the precision score. For example, the highest precision it yielded is 0.41 whereas in that case the precision of SampleHST is 0.94, which is 2.3× higher. As we mentioned earlier, identifying such anomalous traces is difficult for such clustering methods due to the high number of dimensions of the input data. Indeed, our input data have 105 dimensions,

Table 5.7: Performance of SampleHST on the Production data considering hyper-rectangles with 5% storage budget

Bandwidth	Jain	Precision	Recall	F1-Score
0.05, 0.1	0.76	0.9	0.91	0.91
0.05, 0.2	0.75	0.91	0.91	0.91
0.05, 0.3	0.74	0.93	0.91	0.92
0.1, 0.2	0.74	0.94	0.91	0.92
0.1, 0.3	0.73	0.97	0.92	0.95

Table 5.8: Comparing the performance of different methods on the Production data. The evaluation criterion for $0.5\% - 2\%$ budget is the precision, for 5% budget it is the F1-Score and for 10% budget it is the recall. The numbers correspond to the evaluation criteria are indicated in bold.

Budget	Random				PERCH-based				SampleHST			
	J	P	R	F1	J	P	R	F1	J	P	R	F1
0.5%	0.1	**0.05**	0.01	0.01	0.32	**0.41**	0.03	0.05	0.41	**0.94**	0.1	0.18
1%	0.1	**0.04**	0.01	0.01	0.24	**0.18**	0.03	0.04	0.5	**0.95**	0.21	0.34
2%	0.11	**0.06**	0.03	0.04	0.32	**0.13**	0.04	0.07	0.47	**0.96**	0.41	0.58
5%	0.13	0.05	0.05	**0.05**	0.47	0.11	0.09	**0.1**	0.73	0.97	0.92	**0.95**
10%	0.18	0.05	**0.1**	0.06	0.56	0.09	**0.15**	0.11	0.88	0.79	**0.94**	0.86

Legend: J - Jain Index, P - Precision, R - Recall, F1 - F1-Score

making the segregation of anomalous and normal traces difficult for PERCH. SampleHST, on the other hand, eliminates this problem by using the the mass scores, which have only 2 dimensions and provide indication on the trace status (normal or anomalous), to perform the clustering.

We now focus on the case with high budget (10%). Firstly, we consider the primary evaluation criteria recall. We see that SampleHST easily outperforms the PERCH-based method. Secondly, when we consider representative sampling, we see that the Jain score produced by SampleHST is 1.6× better than the PERCH-based method. The reason for SampleHST performing better is as follows. The primary objective of SampleHST is to sample as much as anomalous traces possible. In high budget cases, it only shifts focus towards normal traces when the primary objective is fulfilled. Now, anomalous traces can create many groups, each with a small size, whereas normal traces create a small number of large groups. This is indeed the case with the production data. As a result, when SampleHST samples most of the traces from anomalous groups, it satisfies the demands of majority of the groups, making it more fair. This is reflected in the Jain score. We conclude that, SampleHST is better in representative sampling, where it is a reasonable objective.

5.6 Extension for Collective Anomalies

In the previous sections, we presented the SampleHST approach for point anomalies. We now present how SampleHST can be applied to the collective anomalies scenario. We show that, with only changing the trace representation, SampleHST produces a robust sampling performance.

5.6.1 Graph Model for Collective Anomalies

A single trace commonly spans across multiple microservices. The trace data can preserve this information about the call-chain among the microservices. As a result, we can extract a workflow from the trace. How these call-chains are preserved depends on the trace format. For example, the Death Star Bench (DSB) traces follow the X-Trace format [FPKS07]. An X-Trace is composed of multiple events, which is referred as reports, from different services. Using the *Source* and *EventID* property of each report in a trace we can construct a directed acyclic graph (DAG) that represents the workflow among the services [LCPAM19].

The Sifter method discussed before considers this DAG. It predicts the anomalies by inspecting all the k-length paths from the DAG. However, it ignores the overall graph, which in many cases is not ideal for anomaly detection. For example, consider the traces related to the errors within a particular software component. These will mostly be structurally different from the typical traces. Thus, using an overall graph representation will possibly lead to better anomaly detection. Considering this, we need to use an unsupervised way to learn the features that represent the overall graph.

We have used the family of graph spectral distances (FGSD) [VZ17] method for the graph representation. The main reason for this was beside providing quality graph features, it also works in an unsupervised way and does not require to store previous traces to provide the current representation. FGSD is based on the spectral analysis of graphs. In such an analysis, the focus is to study the graph properties related to the set of graph eigenvalues. For FGSD the Laplacian matrix (the difference between the degree and adjacency matrix) is used to calculate the pairwise distances between the graph vertices. The final distance matrix is subsequently

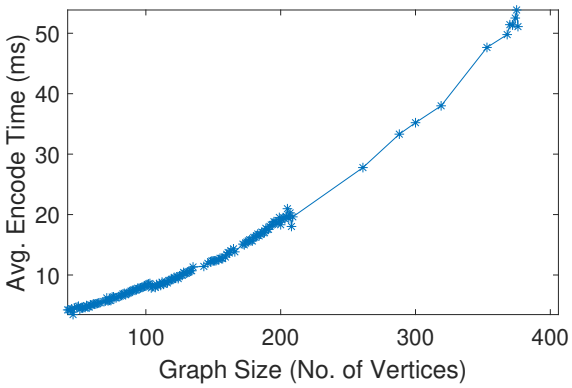


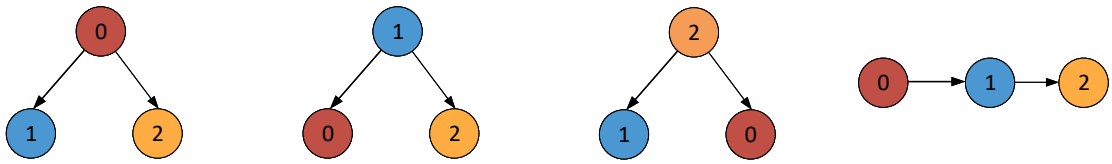
Figure 5.14: Time complexity of FGSD algorithm with respect to graph sizes.

converted to a histogram, which is considered as the set of features. Since FGSD works on a simple connected graph, we need to convert the DAG to its undirected form.

To implement FGSD in a real-time scenario we need to consider two issues. The first issue is scalability. Here, our scalability concern is the about the graph size and the time it requires to transform it to a feature vector. As mentioned already, the feature vector is a histogram of pairwise spectral distances. The number of histogram bins and range have no significant impact over the transformation tine. However, it depends significantly on the number of vertices in the input graph. In the context of DSB dataset it is the number of events. As reported by Verma *et al.* [VZ17], the time complexity of FGSD is $\mathcal{O}(N^2)$ where N is the number of vertices in the graph, as such the method will not be applicable to all traces.

Although a complexity of $\mathcal{O}(N^2)$ seems too onerous, we wanted to measure the maximum graph sizes that may still be processed in practical scenarios. We have tested this time complexity of FGSD using the DSB dataset. Here, we consider the traces whose corresponding graph size falls within 5^{th} to 95^{th} percentile. In other words, we ignored the graphs that are either too big or too small. We have tested each of those selected graphs 1000 times and considered the average time. We plotted the average time against the graph sizes in Fig. 5.14. We see that the initial growth in conversion time is roughly linear. The highest encoding time is around 50ms and till size 100, it is less than 10ms. Thus, adding this pre-processing layer in an anomaly detection process does not seem impractical if the graph sizes are such.

The second issue that we need to consider is graph isomorphism. The goal of FGSD is to de-



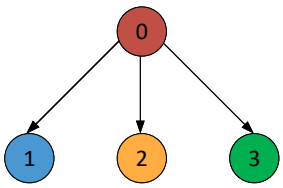
$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

(a) Case 1 (b) Case 2 (c) Case 3 (d) Case 4

Figure 5.15: Examples of directed graphs of traces and the adjacency matrix of their undirected version. Only the first 3 directed graphs are isomorphic, whereas the undirected versions of all the graphs are isomorphic.

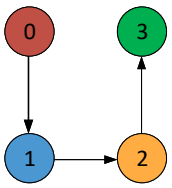
termine whether two graphs have a similar structure. As a result, if two graphs are isomorphic, FGSD will provide a similar feature vector for them. Two graphs G and H are isomorphic when for every pair of vertices (u, v) that are adjacent in G, we can find a pair $(f(u),f(v))$ in H that are also adjacent where f is a bijection (one-to-one correspondence) between the vertices of G and H. In other words, *if we can relabel the nodes so that two graphs are identical, they are isomorphic*. This means that, despite having different parent-child relationship among the spans or traces in two separate traces, they will have similar FGSD feature vector when the corresponding graphs are isomorphic. However, the method will be particularly helpful if the traces differ in length and forking pattern.

For illustration purpose, let us consider 3 traces of length 3 (containing 3 spans). We present the corresponding directed graphs in Fig. 5.15a-5.15c. The figures also present the adjacency matrix of the corresponding undirected graphs, which is considered by FGSD. As seen in the figure, though the index of the root span is different, the structure of the graphs are similar and both the directed and undirected version of the graphs are isomorphic. Thus, FGSD will provide a similar feature vector for all these traces. In Fig. 5.15d we present a different orientation of the same spans. We can see that this directed graph is non-isomorphic to the directed graphs in Fig. 5.15a-5.15c. However, the undirected version of the graph in Fig. 5.15d (as shown by the adjacency matrix), is isomorphic to the undirected version of the graphs in Fig. 5.15a-5.15c. As a result, FGSD will yield the same feature vector for this trace as well.



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

(a) Case 1



$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

(b) Case 2

Figure 5.16: Examples of non-isomorphic graphs of traces of length four

Now we consider two more variants of the above mentioned traces where we add one extra span (shown in Fig. 5.16). As before, we also present the adjacency matrix of the corresponding undirected graphs in the figure. Here, it is clear that both the directed and undirected graphs of these traces are non-isomorphic. As a result their feature vector will be dissimilar and thus they can be differentiated by a anomaly detection algorithm. Note that, unlike FGSD, the count-based feature vector will yield the same feature vector for both traces.

5.6.2 Anomaly Detection

We now test how this FGSD based trace representation aids in anomaly detection. Here, as used before for collective anomalies, we use the DSB trace data. We particularly considered two APIs: *User Timeline* and *Compose Post*. These are also used for evaluation of the Sifter method [LCPAM19]. For these APIs, we mixed up the normal and broken traces, similar to the work of Las-Casas *et al.* [LCPAM19] and marked the broken traces as anomalies. Both the dataset contained 1000 traces with 5% anomalies. Since there are more than 1000 traces available for both those APIs, we randomly generated 30 combinations of them and report the average performance. It should be noted that the graphs generated from these traces are more complex than the ones we used for illustration in Fig. 5.15 and 5.16. For the *User Timeline* API, the median node size is 56. The *Compose Post* API, on the other hand, contains more complex traces with a median node size of 392.

To understand the impact of FGSD on HST for anomaly detection, we consider the HST

Table 5.9: Comparing the anomaly detection results of three methods on the DSB dataset

	User Timeline			Compose Post		
	Avg. Precision	**Avg. Recall**	**Avg. F1-Score**	**Avg. Precision**	**Avg. Recall**	**Avg. F1-Score**
Sifter	0.667	0.922	0.773	0.907	0.881	0.893
FGSD+DBN +OCSVM	0.919	0.943	0.931	0.845	0.889	0.857
FGSD+HST	0.961	0.999	0.980	0.949	0.944	0.946

version for section 5.3 i.e. we do not apply the HST modifications (presented in 5.3.3) for this experiment only. As we want to compare the performance of HST with a method intended for collective anomalies scenario, we use a variant of Sifter as an anomaly detection method. The concept behind this variant is as follows. Since Sifter provides a sampling probability for each trace, we can simply apply a threshold over this probability and if we observe a trace above the threshold, we mark it as an anomaly. In addition, we incorporated a recent anomaly detection method DBN+OCSVM with FGSD and also compared its result with others. The particular reason for choosing this method is that it allows to compare the performance of another classifier other than HST, when combined with FGSD.

We compare the results on the DSB dataset in Table 5.9. We see that for the Compose Post API, Sifter produces slightly better F1-Score than FGSD+DBN+OCSVM. However, FGSD+DBN+OCSVM significantly outperforms Sifter for the User Timeline API. This shows that FGSD can provide proper representation of a graph that can be used for anomaly detection. When we integrate FGSD with HST, we even achieve a higher F1-Score. FGSD+HST improves between 6% and 27% the F1-score of Sifter. HSTs are also 5% to 17% more accurate in F1-score than DBN+OCSVM.

Based on these results, it is clear that the FGSD based trace representation and the HST classifier can detect the collective anomalies from trace streams. In the next section, we test whether this combination can produce accurate sampling performance.

5.6.3 Sampling Performance

We now apply the SampleHST method, with FGSD based trace representation, based on datasets generated from the DSB traces. To assess the long term sampling performance, the

Table 5.10: Performance of SampleHST on the Compose Post API of the DSB dataset with different clustering bandwidth. The Score for $0.5\% - 2\%$ budget is the precision, for 5% budget it is the F1-Score and for 10% budget it is the recall.

Budget	**Score (0-1)**		
	$h = 0.1$	$h = 0.09$	$h = 0.11$
0.5%	0.94	0.94	0.94
1%	0.95	0.96	0.95
2%	0.96	0.96	0.96
5%	0.92	0.90	0.92
10%	0.89	0.86	0.89

Table 5.11: Sampling results for different budget on the Compose Post API with exact and approximate counting

Budget	**Exact Counting**			**Approximate Counting**		
	P	**R**	**F1**	**P**	**R**	**F1**
0.5%	0.938	0.095	0.172	0.937	0.095	0.172
1%	0.952	0.192	0.319	0.957	0.192	0.32
2%	0.962	0.387	0.552	0.962	0.387	0.552
5%	0.964	0.876	0.917	0.963	0.866	0.911
10%	0.888	0.89	0.885	0.878	0.884	0.877

Legend: J - Jain Index, P - Precision, R - Recall, F1 - F1-Score

datasets are generated such that they contain the same number of traces as the production data. We have injected 5% broken traces in the datasets, which are marked as anomalies. We aim to answer the same research questions introduced in Section 5.5.1. For **RQ1**, similar to Section 5.5.2, we initially consider the sensitivity to change in bandwidth. We present the results for the the dataset involving the Compose Post API in Table 5.10. We observe that for both $h = 0.1$ and $h = 0.11$ the sampling performance is similar. Thus, we have decided to use the bandwidth $h = 0.1$ which we also used for the production data. We present the sampling performance with and without approximate counting in Table 5.11. Note that, we do not calculate the Jain score here as there are only two groups: normal and anomalies, which reduces the significance of fairness. From the results, we see that, in all the cases, SampleHST produces high values of corresponding evaluation criteria. The values produced by approximate counting is roughly similar. Like the case for point anomalies, this similarity is due to the reason that the mass scores remain same for both the exact and approximate counting.

For **RQ2**, we have also used the Compose Post API. The bandwidths we use for this experiment is the same set of bandwidths used in Table 5.7. Like the production data experiments, here we consider the 5% budget case as the evaluation criteria for this case is the F1-Score *i.e.*,

Table 5.12: Sampling performance on the Compose Post API considering hyper-rectangles with 5% storage budget

Bandwidth	Precision	Recall	F1-Score
0.05,0.1	0.96	0.88	0.92
0.05,0.2	0.97	0.92	0.94
0.05,0.3	0.97	0.91	0.94
0.1,0.2	0.97	0.92	0.94
0.1,0.3	0.97	0.91	0.94

Table 5.13: Comparing the performance of different methods on the Compose Post API of the DSB dataset. The evaluation criterion for $0.5\% - 2\%$ budget is the precision, for 5% budget it is the F1-Score and for 10% budget it is the recall. The numbers correspond to the evaluation criteria are indicated in bold.

Budget	Random			Sifter			SampleHST		
	P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
0.5%	**0.04**	0	0.01	**0.27**	0.04	0.07	**0.92**	0.09	0.17
1%	**0.05**	0.01	0.02	**0.29**	0.09	0.13	**0.95**	0.19	0.32
2%	**0.04**	0.02	0.02	**0.29**	0.17	0.21	**0.96**	0.39	0.55
5%	0.05	0.05	**0.05**	0.29	0.42	**0.35**	0.97	0.91	**0.94**
10%	0.04	**0.09**	0.06	0.27	**0.76**	0.40	0.63	**0.92**	0.75

Legend: J - Jain Index, P - Precision, R - Recall, F1 - F1-Score

both the precision and recall is important. The results are presented in Table 5.12. As we can see, the average F1-Score is 0.94. The minimum F1-Score that we achieve in this case is 0.92 for bandwidth $[0.05, 0.1]$, which is still higher than the F1-Score of 0.91 we achieved with a hyper-cube. This indicate that using hyper-rectangles as cluster shapes has the possibility to achieve high sampling performance even with the FGSD based trace representation. Thus, for the latter experiments we have used hyper-rectangles.

The results concerning **RQ3** is provided in Table 5.13 and 5.14. Here we have considered both the Compose Post and User Timeline API. We compare the results with the Sifter method and use the Random method as a baseline. We see that for all budgets, SampleHST outperforms the Sifter method. It may seem counter intuitive by seeing the sampling precision, recall and F1-Score for Sifter is much smaller than its anomaly detection values shown in Table 5.9 but it is not. The sampling values are smaller here because, as suggested for Sifter [LCPAM19], we do not apply any threshold here and sample a trace using its sampling probability. However, since the sampling probabilities generated by Sifter are low, many of the anomalous traces are not sampled. For example, let us consider the case of the User Timeline traces. Here the mean probability generated by Sifter is ~ 0.03 and the 95^{th} percentile is ~ 0.12. Although, the 95^{th}

Table 5.14: Comparing the performance of different methods on the User Timeline API of the DSB dataset. The evaluation criterion for $0.5\% - 2\%$ budget is the precision, for 5% budget it is the F1-Score and for 10% budget it is the recall. The numbers correspond to the evaluation criteria are indicated in bold.

Budget	**Random**			**Sifter**			**SampleHST**		
	P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
0.5%	**0.05**	0	0.01	**0.06**	0.01	0.02	**0.89**	0.1	0.17
1%	**0.04**	0.01	0.01	**0.06**	0.02	0.03	**0.92**	0.19	0.31
2%	**0.05**	0.02	0.03	**0.05**	0.03	0.04	**0.95**	0.39	0.55
5%	0.05	0.05	**0.05**	0.05	0.08	**0.06**	0.98	0.98	**0.98**
10%	0.05	**0.1**	0.07	0.05	**0.15**	0.08	0.98	**1.0**	0.99

Legend: J - Jain Index, P - Precision, R - Recall, F1 - F1-Score

percentile is 4 times higher, its value is still very low for a selection probability. As a result, we have less anomalies sampled, eventually leading to low precision, recall and F1-Score values.

From these experiments it is evident that, like point anomalies, SampleHST can produce a robust sampling performance for collective anomalies. This is certainly encouraging as for SampleHST, the transition from point to collective anomalies can be done with minimal effort - we only need to change the trace model. Since both the trace models are based on count data, the rest of the SampleHST pipeline can remain the same.

5.7 Summary

In this chapter we propose a novel sampling method for distributed tracing named *SampleHST*. The primary objective of SampleHST is to take its sampling decision based on the proportion of sampling budget and the fraction of interesting or anomalous traces. If the budget is lower, the priority is to sample the anomalous traces. On the other hand, when the budget higher, the normal traces are sampled as well. This sampling process is based on an online clustering mechanism. The traces are first clustered using their mass scores, which is an indication of the observed frequency of trace groups. After that, if the budget permits, the sampling decisions are taken based on the trace locality. Our experiments, that considers both data from the literature and production data from a cloud data center, show that SampleHST by far outperforms some of the recent approaches.

Chapter 6

Conclusion

Here, in this chapter, we present our concluding remarks. Initially, we summarize the achievements from each of the contributing chapters. This is followed by future research directions.

6.1 Summary of Thesis Achievements

In this thesis we have identified three scenarios, involving systems built with fine-grained services e.g., the microservices architecture and FaaS, that require a novel resource management approach. Each of these scenarios impacts different system resources. In the first scenario, which involves a system built with the microservices architecture, we consider the compute capacity in terms of CPU. In the second scenario, which is concerned with installing a private FaaS platform, the resource under consideration is the primary memory. In the final scenario, where we consider a data center, hosting microservices, that generates traces with distributed tracing, the resource under consideration is the storage. To address these issues, we identify three research objectives, **RO1-RO3**, that have been introduced in Chapter 1. Each of these objectives has been the focus in Chapters 3-5 respectively. The summary of the achievements of these three chapters is as follows.

The first research objective, **RO1**, is to autoscale the CPU capacity of microservices. This is our focus in Chapter 3 where we present an autoscaler, namely ATOM, that scales the CPU

capacity of microservices at runtime. We have divided RO1 into two further sub-objectives: **RO1.1**, modeling the performance of microservices and **RO1.2**, optimal capacity scaling. By completing these objectives, ATOM can assess different scaling configurations on the fly and scales the system with the optimal configuration for the current workload. To complete RO1.1, it utilizes a layered queueing network (LQN) model of the system. The model considers the dynamics of such systems such as the workload, shared resources, and multiple application layers. By solving the model, ATOM gains an insight of critical performance metrics such as the system throughput and utilization. For ATOM, these performance metrics are the key to completing RO1.2, i.e., applying optimal scaling configurations. ATOM compares the performance of different combination of vertical and horizontal scaling configurations using the LQN model, based on which an optimal configuration is suggested. It should be noted that this comparison is a time-bounded process since the decision is taken at runtime. We have used a popular microservices application named Sock Shop to evaluate the performance of ATOM. The experiments show that ATOM easily outperforms the baseline autoscalers under a wide range of workloads, including the ones with burstiness.

The second research objective, **RO2**, is to design a cold start aware capacity planner for FaaS platforms, which has been addressed in Chapter 4. Here, we present a capacity planner, namely COCOA, for an on-premises FaaS platform. While suggesting a required capacity, COCOA considers the trade-off between latency delay and consumption of primary memory due to cold starts. It eventually identifies a limit of the cold starts, such that the SLAs for response time are not violated, while limiting the memory usage of the functions. To achieve this, we divide RO2 into two more sub-objectives: **RO2.1**, modeling the function performance considering cold starts and **RO2.2**, capacity planning leveraging the cold start model. To complete RO2.1, COCOA leverages an M/M/k setup and LQN model to estimate the function response times. Using such models, it can consider a varying degree of cold starts and assess their impact on function response times and their memory consumption. Based on this assessment, we address RO2.2. That is, COCOA provides a deployment configuration, which includes the CPU and memory capacity that ensure the SLAs for function response times. We have developed, FaasSim, a simulator for an event-driven FaaS platform to evaluate COCOA. We demonstrate

that COCOA outperforms availability-ware approaches, used in TTL cache, in suggesting a deployment configuration while satisfying the SLAs.

We address our final research objective, **RO3**, in Chapter 5. In this chapter, we present SampleHST, a trace sampler for distributed tracing, which is targeted at microservices hosted in a data center. SampleHST works under a budget constraint meaning it only selects a fraction of the traces, which is referred as the budget, to be saved for future troubleshooting. Like before, we divide RO3 into two sub-objectives: **RO3.1**, identifying the traces that are anomalous in an unsupervised manner and **RO3.2**, incorporate this identification process in a trace sampler. To address RO3.1, SampleHST initially converts a trace to either a bag of words or a graph-based representation. Subsequently, it uses a forest of half space trees to characterize a trace based on its mass scores. This characterization allows to separate the normal traces from the anomalous ones. Using this characterization, we address our second sub-objective RO3.2. The sampler initially incorporates the mass scores of the traces in an online clustering mechanism to create multiple groups of anomalous and normal traces. Subsequently, the traces are sampled based on the given budget, where the priority is to sample from the anomalous trace groups. We have compared SampleHST with recently proposed trace sampling methods using both production data and data reported in literature. In both cases, SampleHST by far outperformed the recently suggested methods.

6.2 Future Work

There are multiple opportunities to extend the research contributions that we presented in this thesis. Here, we discuss the most interesting ones for each of the contributions.

For ATOM, a possible line of future research direction is using online profiling to estimate the service demands of the microservices. Service demands are an important component of the LQN model, which we assume is profiled statically via performance testing. Estimating it from the online monitoring data will remove this extra step while performance testing. One possible way to achieve this could be using the traces generated from distributed tracing. Such traces

already include the request response times. It is also possible to extract the system throughput, provided the server clocks are synchronized. However, we still need to incorporate other metrics, e.g., queue length and utilization, to calculate the service demand. The challenge is to infer those values with minimal code instrumentation. Another extension on this line of work is to infer the LQN model itself. This will be particularly helpful for a cloud service provider who might not have access to such models. The challenge here is to infer the system architecture, which is relevant to the performance of the system, without violating user privacy. On a separate note, another line of future work could be investigating the impact of microservices migration, in conjunction with vertical and horizontal scaling, to improve the system performance. This is possible because moving a microservice replica to a faster server can sufficiently serve the increase in workload. However, the challenge is to consider the trade-off between availability and the improvement in performance.

For COCOA, an interesting future work could be considering a scenario where one replica is not sufficient to serve the current workload, particularly the ones affected by burstiness. This work requires changing the structure of the LQN model, we consider, where a function is represented by an entry. This can be achieved by replacing an entry with a task to represent a function. However, this solution can raise a scalability issue for the existing model solvers. The model solvers significantly slow down with the increase of number of tasks once it reaches in the range of hundreds and in some cases, they might even fail to solve the model. Thus, further work is required to improve the efficiency of the model solvers. Another way to solve this problem is considering each of the replicas as entries but this will require changing the structure of the LQN, rather than the replication property, to test every new deployment configuration. It will be interesting to see the trade-off between model accuracy and solver speed for both these approaches. A different line of work could be considering SLAs in terms of percentile of response times rather than the mean. This requires the support of model solvers to estimate the variance in response time. We need to assess the accuracy of such performance estimations before utilizing them in capacity planning.

For SampleHST, there are also multiple future research directions. An interesting work would be investigating the impact of updating the structure of the HSTs as time progresses. This

updating mechanism has the potential to identify a change in the trace pattern and thus further preventing erroneous sampling decision. Another line of work could be incorporating more properties in the clustering dimension and assess the sampling performance. This is possible since the SampleHST clustering method does not have a restriction in the number of clustering dimension. It would be also interesting to see how adding these extra dimensions effect the speed of the sampling process and what is the trade-off between speed and sampling accuracy. Finally, another promising idea is to integrate the continuous trace properties, e.g., the response time, with the trace representation. This opens the possibility to identify and sample the traces with latency anomalies in an integrated approach. This research issue has already gained the attention of the researchers, but still in an early stage.

Bibliography

[ACR+18] 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 *Proceedings of the USENIX ATC*, pages 923–935. USENIX, 2018.

[AGW+21] Ahmad Alnafessah, Alim Ul Gias, Runan Wang, Lulai Zhu, Giuliano Casale, and Antonio Filieri. Quality-Aware DevOps Research: Where Do We Stand? *IEEE Access*, 9:44476–44489, 2021.

[And15] Charles Anderson. Docker. *IEEE Software*, 32(3):102–105, 2015.

[APC+15] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance Evaluation of Microservices Architectures Using Containers. In *Proceedings of the International Symposium on Network Computing and Applications (NCA)*, pages 27–34. IEEE, 2015.

[BA12] Rashmi Dutta Baruah and Plamen Angelov. Evolving local means method for clustering of streaming data. In *Proceedings of the International Conference on Fuzzy Systems*, pages 1–8. IEEE, 2012.

[BCC+17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[BCS09] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: Performance Engineering Tools for System Modeling. *SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.

[Ber14] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[BG20] Luciano Baresi and Martin Garriga. Microservices: The Evolution and Extinction of Web Services? In *Microservices*, pages 3–28. Springer, 2020.

[BGdMT06] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, 2006.

[BGO+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[BKB20] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. Using Application Knowledge to Reduce Cold Starts in FaaS Services. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 134–143. ACM, 2020.

[BKFL17] Cornel Barna, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu. Delivering Elastic Containerized Cloud Applications to Enable DevOps. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 65–75. IEEE, 2017.

[BLF+18] Cornel Barna, Marin Litoiu, Marios Fokaefs, Mark Shtern, and Joe Wigglesworth. Runtime Performance Management for Cloud Applications with Adaptive Controllers. In *Proceedings of the International Conference on Performance Engineering (ICPE)*, pages 176–183. ACM/SPEC, 2018.

[BLV+19] André Bauer, Veronika Lesch, Laurens Versluis, Alexey Ilyushkin, Nikolas Herbst, and Samuel Kounev. Chamulteon: Coordinated Auto-Scaling of Micro-Services. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 2015–2025. IEEE, 2019.

[BNCK20] Jasmin Bogatinovski, Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Self-Supervised Anomaly Detection from Distributed Traces. In *Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, pages 342–347. IEEE, 2020.

[BPSAP+19] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the International Middleware Conference*, pages 41–54, 2019.

[BSG+18] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-Based Caching for Content Delivery. *IEEE/ACM Transactions on Networking*, 26(3):1063–1077, 2018.

[Cas18] Giuliano Casale. Analyzing replacement policies in list-based caches with non-uniform access costs. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, pages 432–440. IEEE, 2018.

[Cas19] Giuliano Casale. Automated Multi-paradigm Analysis of Extended and Layered Queueing Models with LINE. In *Companion Proceedings of the International International Conference on Performance Engineering (ICPE)*, pages 37–38. ACM/SPEC, 2019.

[CB17] Tao Chen and Rami Bahsoon. Self-Adaptive Trade-off Decision Making for Autoscaling Cloud-Based Services. *IEEE Transactions on Services Computing*, 10(4):618–632, 2017.

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):15:1–15:58, 2009.

[CLL19] MinSu Chae, HwaMin Lee, and Kiyeol Lee. A performance comparison of linux containers and virtual machines using Docker and KVM. *Cluster Computing*, 22(1):1765–1775, 2019.

[CP21] Anurag Choudhry and Anshu Premchand. Microservices and DevOps for Optimal Benefits from IoT in Manufacturing. In *Proceedings of the International Conference on Recent Trends in Machine Learning, IoT, Smart Cities and Applications*, pages 375–384. Springer, 2021.

[CSB02] Mung Chiang, Arak Sutivong, and Stephen Boyd. Efficient nonlinear optimizations of queuing systems. In *Proceedings of the Global Telecommunications Conference (GLOBECOM)*, volume 3, pages 2425–2429. IEEE, 2002.

[CTW02] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical Web Caching Systems: Modeling, Design and Experimental Results. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(7):1305–1314, 2002.

[DBC18] Salvatore Dipietro, Rajkumar Buyya, and Giuliano Casale. PAX: Partition-aware autoscaling for the Cassandra NoSQL database. In *Proceedings of the Network Operations and Management Symposium (NOMS)*, pages 1–9. IEEE/IFIP, 2018.

[DFJR98] John Dilley, Rich Friedrich, Tai Jin, and Jerome Rolia. Web server performance measurement and modeling techniques. *Performance evaluation*, 33(1):5–26, 1998.

[DMT+19] Mostafa Dehghan, Laurent Massoulie, Don Towsley, Daniel Sadoc Menasche, and Yong Chiang Tay. A Utility Optimization Approach to Network Cache Design. *IEEE/ACM Transactions on Networking*, 27(3):1013–1027, 2019.

[Edu19] IBM Cloud Education. What is FaaS (Function-as-a-Service)? `https://www.ibm.com/cloud/learn/faas`, 2019. Accessed: 2022-01-15.

[EFK15] Gil Einziger, Benny Fellman, and Yaron Kassner. Independent counter estimation buckets. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, pages 2560–2568. IEEE, 2015.

[EGHS16] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. DevOps. *IEEE Software*, 33(3):94–100, 2016.

[EGVE+20] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the Costs of Serverless Workflows. In *Proceedings of the International Conference on Performance Engineering (ICPE)*, pages 265–276. ACM/SPEC, 2020.

[Elg18] T. Elgamal. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *Proceedings of the Symposium on Edge Computing (SEC)*, pages 300–312. IEEE/ACM, 2018.

[ERKL16] Sarah M Erfani, Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. *Pattern Recognition*, 58:121–134, 2016.

[FAOW+08] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2008.

[FDN16] Luca Florio and Elisabetta Di Nitto. Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 357–362. IEEE, 2016.

[Fel08] Willliam Feller. *An introduction to probability theory and its applications, vol 2.* John Wiley & Sons, 2008.

[FFH13] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-Based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud. In

Proceedings of the International Conference on Software Engineering (ICSE), pages 512–521. IEEE, 2013.

[FFRR15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *Proceedings of the International Symposium On Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE, 2015.

[FH75] Keinosuke Fukunaga and Larry Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

[FMW+13] Greg Franks, Peter Maly, Murray Woodside, Dorina C. Petriu, Alex Hubbard, and Martin Mroz. Layered Queueing Network Solver and Simulator User Manual. `http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf`, 2013. Accessed: 2022-01-15.

[FPKS07] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *Proceedings of the Symposium on Networked Systems Design & Implementation (NSDI)*. USENIX, 2007.

[Fra00] Roy Gregory Franks. *Performance analysis of distributed server systems.* PhD thesis, Carleton University, 2000.

[FRR12] Christine Fricker, Philippe Robert, and James Roberts. A Versatile and Accurate Approximation for LRU Cache Performance. In *Proceedings of the International Teletraffic Congress (ITC)*, pages 1–8. IEEE, 2012.

[Fuk13] Keinosuke Fukunaga. *Introduction to statistical pattern recognition.* Elsevier, 2013.

[FW98] Greg Franks and Murray Woodside. Performance of Multi-Level Client-Server Systems with Parallel Service Operations. In *Proceedings of the International Workshop on Software and Performance*, pages 120–130. ACM, 1998.

[GAMKF20] Deeksha Gorige, Eyhab Al-Masri, Sergey Kanzhelev, and Hossam Fattah. Privacy-Risk Detection in Microservices Composition Using Distributed Tracing. In *Proceedings of the Eurasia Conference on IOT, Communication and Engineering (ECICE)*, pages 250–253. IEEE, 2020.

[GC20] Alim Ul Gias and Giuliano Casale. COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020. ©2020 IEEE.

[GCW19] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven autoscaling for microservices. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004. IEEE, 2019. ©2019 IEEE.

[GDHBSW13] Anshul Gandhi, Sherwin Doroudi, Mor Harchol-Balter, and Alan Scheller-Wolf. Exact Analysis of the M/M/k/setup Class of Markov Chains via Recursive Renewal Reward. In *Proceedings of the SIGMETRICS*, pages 153–166. ACM, 2013.

[GDK+14] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, Model-driven Autoscaling for Cloud Applications. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 57–64, 2014.

[GDK+18] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Model-driven optimal resource scaling in cloud. *Software & Systems Modeling*, 17(2):509–526, 2018.

[GHBRK12] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Transactions on Computer Systems*, 30(4):14, 2012.

[Gla94] Steven Glassman. A caching relay for the world wide web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.

[GLHR18] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments. In *Proceedings of the International Conference on Performance Engineering (ICPE)*, pages 157–167. ACM, 2018.

[GP05] Gordon P Gu and Dorina C Petriu. From UML to LQN by XML algebra-based model transformations. In *Proceedings of the Internatioanl Workshop on Software and Performance*, pages 99–110. ACM, 2005.

[GPW+20] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1387–1397. ACM, 2020.

[GZC+19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2019.

[GZH+19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 19–33, 2019.

[HAM17] Richard Hyde, Plamen Angelov, and Angus Robert MacKenzie. Fully online clustering of evolving data streams into arbitrarily shaped clusters. *Information Sciences*, 382:96–114, 2017.

[HB13] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge Univ. Press, 2013.

[HCY+21] Zicheng Huang, Pengfei Chen, Guangba Yu, Hongyang Chen, and Zibin Zheng. Sieve: Attention-based Sampling of End-to-End Trace Data in Distributed Microservice Systems. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 436–446. IEEE, 2021.

[Hef16] Eslam Hefnawy. Serverless Code Patterns. `https://serverless.com/blog/serverless-architecture-code-patterns`, 2016. Accessed: 2022-01-15.

[HIB20] Mubin Ul Haque, Leonardo Horn Iwaya, and M Ali Babar. Challenges in docker development: A large-scale study using stack overflow. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. ACM/IEEE, 2020.

[HKR13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 23–27. USENIX, 2013.

[HLZT17] MohammadReza HoseinyFarahabady, Young Choon Lee, Albert Y Zomaya, and Zahir Tari. A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, pages 241–255. Springer, 2017.

[HS17] Wilhelm Hasselbring and Guido Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-commerce. In *Proceedings of the International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246. IEEE, 2017.

[HvHK+17] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance Engineering for Microservices: Research Challenges and Directions. In *Compan-*

ion Proceedings of the International Conference on Performance Engineering (ICPE), pages 223–226. ACM/SPEC, 2017.

[ITT18] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. Combined Vertical and Horizontal Autoscaling Through Model Predictive Control. In *Proceedings of the European Conference on Parallel Processing*, pages 147–159. Springer, 2018.

[Jaf81] Jeffrey Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[Jai20] Shashank Mohan Jain. *Linux Containers and Virtualization*. Springer, 2020.

[JbAPT18] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. Towards a benefits dependency network for DevOps based on a systematic literature review. *Journal of Software: Evolution and Process*, 30(11):e1957, 2018.

[JCH84] Rajendra K Jain, Dah-Ming W Chiu, and William R Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[Kay20] Paridhika Kayal. Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope. In *Proceedings of the World Forum on Internet of Things (WF-IoT)*, pages 1–6. IEEE, 2020.

[KBBML16] Hamzeh Khazaei, Cornel Barna, Nasim Beigi-Mohammadi, and Marin Litoiu. Efficiency Analysis of Provisioning Microservices. In *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 261–268. IEEE, 2016.

[KC03] Jeffrey O Kephart and David M Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[Kes97] S Keshave. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley, 1997.

[KHL+18] Young Ki Kim, M Reza HoseinyFarahabady, Young Choon Lee, Albert Y Zomaya, and Raja Jurdak. Dynamic Control of CPU Usage in a Lambda Platform. In *Proceedings of the International Conference on Cluster Computing (CLUSTER)*, pages 234–244. IEEE, 2018.

[KMKM17] Ari Kobren, Nicholas Monath, Akshay Krishnamurthy, and Andrew McCallum. A hierarchical algorithm for extreme clustering. In *Proceedings of the SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 255–264. ACM, 2017.

[KPSCD09] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. Estimating Service Resource Consumption From Response Time Measurements. In *Proceedings of the International Conference on Performance Evaluation Methodologies and Tools*, pages 1–10. ICST, 2009.

[KQ17] Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software*, 126:1–16, 2017.

[KRP+17] Hamzeh Khazaei, Rajsimman Ravichandiran, Byungchul Park, Hadi Bannazadeh, Ali Tizghadam, and Alberto Leon-Garcia. Elascale: Autoscaling and Monitoring as a Service. In *Proceedings of the Annual International Conference on Computer Science and Software Engineering*, pages 234–240. IBM Corp., 2017.

[KWJM19] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 80–90. IEEE, 2019.

[KWS+18] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless an-

alytics. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 427–444. USENIX, 2018.

[LCMGF18] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. Weighted sampling of execution traces: Capturing more needles and less hay. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 326–332, 2018.

[LCPAM19] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the Symposium on Cloud Computing (SoCC)*, pages 312–324. ACM, 2019.

[LCW+09] Jim Li, John Chinneck, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. Performance Model Driven QoS Guarantees and Optimization in Clouds. In *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 15–22. IEEE Computer Society, 2009.

[LM14] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[LR99] Guy Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. SIAM, 1999.

[LRC+18] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *Proceedings of the International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[LVZ+18] Wes Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George Leavesley. Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads. In *Companion Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, pages 195–200. IEEE/ACM, 2018.

[LW19] Nicolas Langrené and Xavier Warin. Fast and stable multivariate kernel density estimation by fast sum updating. *Journal of Computational and Graphical Statistics*, 28(3):596–608, 2019.

[LZGS84] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[MA10] R Timothy Marler and Jasbir S Arora. The weighted sum method for multi-objective optimization: new insights. *Structural and multidisciplinary optimization*, 41(6):853–862, 2010.

[MCCS09] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Injecting Realistic Burstiness to a Traditional Client-Server Benchmark. In *Proceedings of the International. Conference on Autonomic Computing (ICAC)*, pages 149–158. ACM, 2009.

[MEHW18] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *Companion Proceedings of the International Conference on Utility and Cloud Computing Companion (UCC)*, pages 181–188. IEEE, 2018.

[Men02] Daniel A Menascé. TPC-W: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3):83–87, 2002.

[MFMR19] Kunal Mahajan, Daniel Figueiredo, Vishal Misra, and Dan Rubenstein. Optimal Pricing for Serverless Computing. In *Proceedings of the Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[MGL14] Valentina Martina, Michele Garetto, and Emilio Leonardi. A unified approach to the performance analysis of caching systems. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, pages 2040–2048. IEEE, 2014.

[Mit97] Tom M Mitchell. *Machine Learning*. McGraw Hill, 1997.

[ML13] Shicong Meng and Ling Liu. Enhanced Monitoring-as-a-Service for Effective Cloud Management. *IEEE Transactions on Computers*, 62(9):1705–1720, 2013.

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[MRBA16] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling Performance & Resource Management in Kubernetes. In *Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, pages 257–262. IEEE/ACM, 2016.

[MSD+19] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 1–6. USENIX, 2019.

[NCK19a] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly detection and classification using distributed tracing and deep learning. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 241–250. IEEE, 2019.

[NCK19b] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly detection from system tracing data using multimodal deep learning. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*, pages 179–186. IEEE, 2019.

[Nel13] Randolph Nelson. *Probability, Stochastic Processes, and Queueing Theory: The Mathematics of Computer Performance Modeling*. Springer, 2013.

[Par62] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[PC17] Juan F Pérez and Giuliano Casale. Line: Evaluating Software Applications in Unreliable Environments. *IEEE Transactions on Reliability*, 66(3):837–853, 2017.

[Pev16] Tomáš Pevnỳ. Loda: Lightweight on-line detector of anomalies. *Machine Learning*, 102(2):275–304, 2016.

[PRN+19] Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, and Germán Moltó. On-premises serverless computing for event-driven data processing applications. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*, pages 414–421. IEEE, 2019.

[PSM+20] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[PVM+19] Carlo Puliafito, Carlo Vallati, Enzo Mingozzi, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Container Migration in the Fog: A Performance Evaluation. *Sensors*, 19(7):1488, 2019.

[PZA+17] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*, 34(1):91–98, 2017.

[QBJ+20] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 805–825. USENIX, 2020.

[QCB16] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications*, 65:167–180, 2016.

[QCB18] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. *ACM Computing Surveys*, 51(4):73, 2018.

[RCK+09] Jerry Rolia, Giuliano Casale, Diwakar Krishnamurthy, Stephen Dawson, and Stephan Kraft. Predictive modelling of SAP ERP Applications: Challenges and Solutions. In *Proceedings of the International. Conference on Performance Evaluation Methodologies and Tools*, pages 9:1–9:9. ICST, 2009.

[RCP20] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Hierarchical Scaling of Microservices in Kubernetes. In *Proccedings of the International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37. IEEE, 2020.

[Ric18] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[RL80] Martin Reiser and Stephen S Lavenberg. Mean-value Analysis of Closed Multichain Queuing Networks. *Journal of the ACM*, 27(2):313–322, 1980.

[RNC19] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*, pages 329–338. IEEE, 2019.

[Ros56] Murray Rosenblatt. Remarks on some nonparametric estimates of a density function. *Annals of Mathematical Statistics*, 27(3):832–837, 1956.

[RS95] Jerome A. Rolia and Kenneth C. Sevcik. The Method of Layers. *IEEE transactions on software engineering*, 21(8):689–700, 1995.

[RS02] Alma Riska and Evgenia Smirni. MAMSolver: A Matrix Analytic Methods Tool. In *Proceedings of the Modelling Techniques and Tools for Computer Performance Evaluation*, pages 205–211. Springer, 2002.

[RS07] Alma Riska and Evgenia Smirni. ETAQA Solutions for Infinite Markov Processes with Repetitive Structure. *INFORMS Journal on Computing*, 19(2):215–228, 2007.

[SCBK15] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92:51–71, 2015.

[SD11] Yasir Shoaib and Olivia Das. Web application performance modeling using layered queueing networks. *Electronic Notes in Theoretical Computer Science*, 275:123–142, 2011.

[SD12] Yasir Shoaib and Olivia Das. Using layered bottlenecks for virtual machine provisioning in the clouds. In *Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, pages 109–116. IEEE Computer Society, 2012.

[Seb02] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[Ser] Amazon Web Services. AWS Lambda limits. `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`. Accessed: 2022-01-15.

[SFG+20] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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 *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 205–218. USENIX, 2020.

[Shi19] Mikhail Shilkov. Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP. `https://mikhail.io/serverless/coldstarts/big3`, 2019. Accessed: 2022-01-15.

[Shk19] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019.

[SL20] Joel Scheuner and Philipp Leitner. Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software*, 170:110708, 2020.

[SM78] CH Sauer and EA MacNair. Simultaneous Resource Possession in Queueing Models of Computers. *SIGMETRICS Performance Evaluation Review*, 7(1-2):41–52, 1978.

[SPJN98] Christiane Shousha, Dorina Petriu, Anant Jalnapurkar, and Kennedy Ngo. Applying performance modelling to a telecommunication system. In *Proceedings of the International Workshop on Software and Performance*, pages 1–6. ACM, 1998.

[TBB+17] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, 72:165–179, 2017.

[Thö15] Johannes Thönes. Microservices. *IEEE Software*, 32(1):113–116, 2015.

[TMW10] Mirco Tribastone, Philip Mayer, and Martin Wirsing. Performance Prediction of Service-Oriented Systems with Layered Queueing Networks. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 51–65. Springer, 2010.

[TTL11] Swee Chuan Tan, Kai Ming Ting, and Tony Fei Liu. Fast anomaly detection for streaming data. In *Proceedings of the International Joint Conference on Artificial Intelligence(IJCAI)*, pages 1511–1516. AAAI, 2011.

[TZLT13] Kai Ming Ting, Guang-Tong Zhou, Fei Tony Liu, and Swee Chuan Tan. Mass estimation. *Machine Learning*, 90(1):127–160, 2013.

[UM06] Alexander Ufimtsev and Liam Murphy. Performance Modeling of a JavaEE Component Application using Layered Queuing Networks: Revised Approach and a Case Study. In *Proceedings of the Conference on Specification and verification of component-based systems*, pages 11–18. ACM, 2006.

[VEIST17] Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The spec cloud group's research vision on faas and serverless architectures. In *Proceedings of the International Workshop on Serverless Computing*, pages 1–4, 2017.

[VETT+18] Erwin Van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță, and Alexandru Iosup. Serverless Is More: From PaaS to Present Cloud Computing. *IEEE Internet Computing*, 22(5):8–17, 2018.

[VZ17] Saurabh Verma and Zhi-Li Zhang. Hunt for the unique, stable, sparse and fast feature learning on graphs. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, pages 87–97, 2017.

[WJ94] Matt P Wand and M Chris Jones. *Kernel smoothing*. CRC press, 1994.

[WLZ+18] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 133–146. USENIX, 2018.

[WNPM95] M Woodside, John E Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. *IEEE Transactions on Computers*, 44(1):20–34, 1995.

[WZXG20] Tao Wang, Wenbo Zhang, Jiwei Xu, and Zeyu Gu. Workflow-aware automatic fault diagnosis for microservice-based applications with statistics. *IEEE Transactions on Network and Service Management*, 17(4):2350–2363, 2020.

[YGL96] Takao Yokota, Mitsuo Gen, and Yin-Xiu Li. Genetic algorithm for non-linear mixed integer programming problems and its applications. *Computers & Industrial Engineering*, 30(4):905–917, 1996.

[YHC21] Guangba Yu, Zicheng Huang, and Pengfei Chen. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *Journal of Software: Evolution and Process*, page e2413, 2021.

[ZBCS16] Liming Zhu, Len Bass, and George Champlin-Scharff. DevOps and Its Practices. *IEEE Software*, 33(3):32–34, 2016.

[ZBYC11] Zhiliang Zhu, Jing Bi, Haitao Yuan, and Ying Chen. SLA Based Dynamic Virtualized Resources Provisioning for Shared Cloud Data Centers. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*, pages 630–637. IEEE, 2011.

[Zim17] Olaf Zimmermann. Microservices tenets. *Computer Science-Research and Development*, 32(3):301–310, 2017.

[ZJZ10] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010.

[ZSK12] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A Survey on Unsupervised Outlier Detection in High-Dimensional Numerical Data. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 5(5):363–387, 2012.

[ZWM+20] Yuan Zuo, Yulei Wu, Geyong Min, Chengqiang Huang, and Ke Pei. An intelligent anomaly detection scheme for micro-services architectures with temporal and spatial data analysis. *IEEE Transactions on Cognitive Communications and Networking*, 6(2):548–561, 2020.

Appendix A

FaasSim: Function-as-a-Service Simulator

Here we present the description of the *FaasSim* tool we used for evaluating COCOA in Chapter 4. The architecture of the platform considered in FaasSim is presented in Figure A.1. In *FaasSim*, the requests arrive following a Poisson process with mean arrival rate λ. There is a dispatcher component that receives all the request and forwards them to corresponding functions. Each of the functions f_i has a probability p_i of being invoked for service. If a function is more popular, its p_i will be higher. Each of the functions has its own service times. The dependencies of these functions are abstracted and any additional time required for those dependencies are considered to be included with the its service times. We assume that the functions are built with microservices pattern. Thus, each function performs exactly one job and does not invoke other functions. The functions are considered running on a multi-core CPU which is approximated as a processor sharing queueing system, i.e., a form of time-sharing with infinitesimally small service quantum.

We have developed FaasSim using the Java programming language. FaasSim runs the simulation using two objects: *Function* and *Job*. A Function object has properties like the mean cold start, service and idle time. It also has two memory consumption values, one for while in execution and the other when it is idle. If we run a simulation with N number of functions,

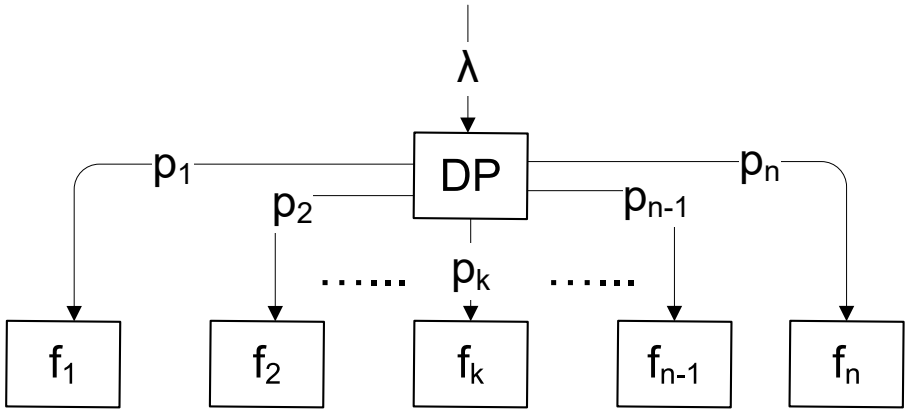


Figure A.1: The architecture of the FaaS platform considered by FaasSim

we initially need to instantiate all the functions with values of these properties. A Job object, which we simply refer as a job, on the other hand, is continuously created by the dispatcher as the simulation progresses. Each job is associated with a function. The probability of a function being associated with a job, p_i, is determined using the Zipf distribution. The value of the distribution shape is passed as a parameter.

A job can be of two types: warm or cold. Depending on whether the corresponding function is loaded in the memory, a type is assigned to the job. The job service time is considered to follow an exponential distribution, with its rate parameter is defined based on the job type. If it is cold, the mean cold start time is used. Otherwise, the mean service time is used. A job also has a residual time property indicating the amount of CPU time it still requires. If a job is cold, once it finishes, a warm job is created automatically. It is easy to understand that this cold job represents the cold start period. When calculating the function response time corresponding to a job, we consider both the warm and cold (if created) job.

Besides the job creation, there are two important aspects of FaasSim: maintaining a delay list and polling. Firstly, the delay list is used like a delay station where the jobs wait because a cold job, which represents loading a function, is still not completed. Once the function is loaded, all the jobs are flushed, means they now can consume CPU time. The amount of time a job spends in the delay list is considered when we calculate the response time. Secondly, we do polling to ensure that a function does not remain loaded in the memory past the time it is allowed. This is controlled by the idle time of each function, which is passed as an input. We perform polling

in intervals, which can be configured in the code.

FaasSim generates M number of samples, which is a configurable option in the code. The outputs it generates are on the basis of these samples. We can also mention a warm-up period, where the first K samples are ignored. We can run the simulation with the same parameters multiple times to calculate the average. For each run of a simulation, the output includes, the CPU utilization, mean memory consumption, maximum memory consumption, mean function response times and logs. The logs can be used to generate information like the mean cold start probability of a function, which can be used for cross-checking. The average and maximum memory consumption values are reported in two ways. In one case, we consider both the function's idle and in-execution memory consumption values. In the other case, only the function's in-execution memory values are considered. This represents the case where there is a guaranteed memory reservation policy for a function.

FaasSim is available for download at `https://github.com/alimulgias/FaasSim`. The URL also contains the instructions on how to execute it.