

Imperial College of Science, Technology and Medicine
Department of Computing

Automated Learning of Performance Models for Cloud-Native Applications

Runan Wang

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy of Imperial College London
April 2024

Declaration of Originality

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

Copyright Statement

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.

Abstract

DevOps is widely used in software development, allowing engineers to release and update software products frequently. It is important to consider system performance to ensure the quality of cloud-native applications. However, working on system performance prediction for microservices and serverless-based applications can be costly and error-prone when models are parameterized on real data. In this thesis, we employ automated learning of performance models from coarse-grained to fine-grained levels. We first focus on accurate model parameterization for microservices. The challenge we tackle is to fit the monitoring data with higher-order moments of the distribution to precisely predict the performance. Next, we focus on automatically generating performance models for serverless applications. The main issue is the information gap between the structural modeling and monitoring granularity, as well as high-level automation. Lastly, we focus on performance diagnosis with code-level tracing strategies to provide inference for internal failures. Even though distributed tracing provides observations at multiple granularities, excessive code instrumentation increases the overheads in terms of computation and storage.

To address the first challenge, we propose a service demand distribution estimation method based on coarse-grained inter-departure times, for both single and multiple service classes. Our experiments yield better performance on fitting real traces of microservices compared to the baseline with Exponential distributions. For the gaps in the automatic performance model generation, we propose to build a compact queueing model based on static analysis and fine-grained profiling data. The experiments on serverless workflows deployed on a public cloud achieve accurate prediction, with a mean error under 7.3%. To mitigate the issues in performance diagnosis, we present a methodology combining code-level analysis to devise tracing data collection strategies and relate probe placements aided by novelty detection methods. Our experimental results show a higher recall score of 90% locating single and multiple faults for 15% latency anomalies.

Acknowledgements

I would like to express my heartfelt gratitude to the individuals who have provided mentorship, collaboration, and unwavering support during my PhD journey.

First and foremost, I extend my deepest thanks to my co-supervisors, Giuliano Casale and Antonio Filieri for their guidance, encouragement, and support. I am deeply appreciative of Giuliano for generously dedicating his time, sharing his wealth of research experience over the past four years. He is supportive not only in enriching my academic abilities, but also in providing me with a lot of help in life. I am also grateful to Antonio, whose insights shaped the direction of the research by combining topics. Also while being abroad, Antonio provides valuable perspectives and feedbacks on academic papers.

I would express my gratitude for the collaboration with the research group in Sun Yat-sen University. I am lucky to work with Pengfei Chen and Guangba Yu in this collaboration project, which is an integral part in my PhD thesis. Guangba helped me on the experimental sections to collect valuable data to verify my methodology. I also want to thank Pengfei for his time on reviewing our collaborative work and timely feedbacks.

I would like to thank all the members of my research group, QORE Lab, for their collaboration and support. They consistently provide me with valuable feedback on my research and greatly assist me in polishing my academic papers and conference presentations. There are so many unforgettable memories in the research group to have meetings and parties that make my PhD life colorful.

Last but not least, I want to express my heartfelt thanks to my parents, Chengli Wang and Yacong Liu, whose support has been my strength to overcome difficulties overseas, especially during the challenging times of the COVID-19 pandemic. I am deeply grateful for the financial support they provided, enabling me to pursue my PhD and life in UK. Their support allows me to study abroad without any worries, by giving me unwavering encouragement and understanding.

Dedication

To my husband, Joe (Zheyu), whose love and care provided steadfast support throughout the difficulties and joys of the PhD journey.

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Motivation	1
1.2 Research Objectives	5
1.3 Contributions	6
1.4 Publications	7
1.5 Thesis Organization	9
2 Background	10
2.1 Introduction	10
2.2 Cloud-native Applications	11
2.2.1 Microservice and Serverless	11
2.2.2 Related Work	15
2.3 Performance Modeling	18

2.3.1	Overview	18
2.3.2	Related Works	22
2.4	Static Source Code Analysis	26
2.5	Distributed Tracing	28
3	Service Demand Distribution Estimation	30
3.1	Introduction	30
3.2	Preliminaries	33
3.3	Problem formulation	35
3.4	Service demand distribution estimation for single service class	38
3.4.1	State space reduction	38
3.4.2	Global optimization based estimation	40
3.4.3	Heuristics-based estimation	41
3.5	Service demand distribution estimation for multiple service classes	45
3.5.1	Class aggregation	45
3.5.2	Multi-class heuristics-based estimation	49
3.6	Single-class Experiments	52
3.6.1	Experimental setup	52
3.6.2	Data preprocessing and clustering	55
3.6.3	Numerical experiment results	56
3.6.4	Analysis of results on measured traces	58
3.7	Multi-class Experiments	61

3.7.1	Numerical experiment results	61
3.7.2	Analysis of results on measured traces	63
3.8	Threats to Validity	66
3.9	Summary	67
4	Code-driven Performance Model Generation	68
4.1	Introduction	68
4.2	Preliminaries and Background	71
4.2.1	Intra-procedure and inter-procedure analysis	71
4.2.2	Taint analysis	71
4.2.3	Parametric dependencies - a performance-related review	72
4.3	Methodology	74
4.3.1	Static analysis for structure extraction	76
4.3.2	Code profiling for model parameterization	81
4.3.3	LQN model generation	84
4.4	Evaluation	85
4.4.1	Experimental setup	85
4.4.2	Experimental results	89
4.5	Threats to Validity	93
4.6	Summary	94

5	Fine-grained Performance Diagnosis with Distributed Tracing	96
5.1	Introduction	96
5.2	Related work	98
5.3	Methodology	100
5.3.1	Overview	100
5.3.2	Code-driven tracing data learning	103
5.3.3	Anomaly identification	109
5.3.4	Enhanced search method for probe placement	111
5.4	Evaluation	114
5.4.1	Experimental setup	114
5.4.2	Experimental results	116
5.5	Threats to Validity	121
5.6	Summary	122
6	Conclusion	124
6.1	Summary of Thesis Achievements	124
6.2	Future Work	126
	Bibliography	128

List of Tables

3.1	Workload mixes	53
3.2	Service demand distribution parameter estimation results for FCFS	59
3.3	Service demand distribution parameter estimation results for FCFS with FCFS- single method	59
3.4	Service demand distribution parameter estimation results for PS	61
3.5	Parameter estimation results for numerical experiments with 3 classes	62
3.6	Parameter estimation results for numerical experiments with 4 classes	63
3.7	Parameter estimation results with FCFS	64
3.8	Parameter estimation results with PS	65
4.1	Notations in Algorithm 5	80
4.2	Description of functions used in Algorithm 6	83
4.3	FaaS-based workflow patterns	87
4.4	The comparison results based on LQN node reduction	89
4.5	MRE of compared results in Figure 4.7	90
4.6	Comparison results on different number of processors on 4 workflow patterns with limited resource $P = 1$	92

4.7	Comparison results on different workloads on 4 workflow patterns with limited resource $N = 10$	92
5.1	Dataset under evaluation	115
5.2	Compared results of enhanced BB algorithm with limited budget	120
5.3	Compared results of enhanced BB algorithm without budget constraint	120

List of Figures

2.1	Context and techniques mapping to thesis chapters	12
2.2	Monolith architecture V.S. Microservice architecture	13
2.3	Illustrative example of a serverless workflow	15
2.4	Example of a closed queueing network	20
2.5	An example LQN model	21
2.6	Flow chart of AST generation from source code	26
2.7	Illustrative example of the generation of call graph and control flow graph	27
2.8	Example of distributed tracing concepts of the given function workflow	28
3.1	The reference closed queueing model for a microservice	36
3.2	Modified model for PS with permanent customers perpetually looping at the queueing station	38
3.3	c^2 conditional on the queue length seen upon arrivals for simulation (a) and the real trace (b)	43
3.4	Example: submodel generation based on class aggregation with 3 classes	47
3.5	Synchronize submodels with iterations	49
3.6	Conditional skewness estimation	52

3.7	The process of experimental comparison	54
3.8	The CCDF of response times (a) and the execution times (b) of maximum likelihood estimation under different number of clusters	56
3.9	CCDF of response time for different settings of c^2	57
3.10	CCDF of response time for different numbers of users N	58
3.11	CCDF of response time with different utilization for the fitted models with FCFS.	59
3.12	CCDF of response time with different utilization for the fitted models with PS.	61
3.13	CCDF of response times of multi-class for numerical experiments	62
3.14	Comparison results for iterative calibration	63
3.15	CCDF of response times for multi-class with FCFS	64
3.16	CCDF of response times for multi-class with PS	65
4.1	The overview of the proposed methodology	75
4.2	An exempld source code in Python from Azure Functions	77
4.3	The corresponding control flow graph of the source code example in Figure 4.2.	78
4.4	An example excerpt of code with taint analysis results highlighted (a), and the control flow graph reduction process (b).	79
4.5	Loop simplification in LQN modeling	82
4.6	Serverless workflows patterns in the experiments	87
4.7	Mean response times of different workflow patterns (a) and of $wf1$ under different workloads (b), comparing model prediction and real trace measurements.	91
5.1	Overview of the proposed methodology	101

5.2	Example of encoding of trace probe location	104
5.3	Example of CFG and corresponding dominator tree	106
5.4	Bisection sampling on feasible paths	107
5.5	Comparison results on heuristics-based starter strategies	117
5.6	Detection results for different OCC models. AE-based models achieve the best recall score among all OCC models.	119

Chapter 1

Introduction

1.1 Motivation

DevOps [BWZ15] has been widely adopted in the industry, becoming an important part of software development methodologies. Compared with traditional software development, DevOps strengthens the cooperation between developers and operators, enabling integration of software development, technical operations and quality assurance (QA) with continuous integration and continuous delivery (CI/CD) [FF06, Vir15, SBZ17]. CI/CD pipelines enable the automated testing, integration, and deployment of code changes. Thus, DevOps aligns with a high degree of automation throughout the whole pipeline to shorten the software development life cycle (SDLC) and deliver high-quality applications.

Cloud-native is a recent paradigm for web-based and service-oriented applications. Rapid development and deployment in DevOps cycles is highly aided by the transition from monolith to microservices and serverless execution. Microservices are self-contained components providing different functional services in the entire application or system. Each microservice can be designed, developed and deployed separately with different programming languages or frameworks. Thus, microservice-based applications benefit from more modular and scalable solutions for software development and deployment, especially in rapid CI/CD cycles. Serverless computing is a novel cloud computing that aims at making operations concerns invisible to developers

and cloud users [BCC⁺17, ESVE⁺20], where developers need to focus solely on designing, developing and deploying code instead of maintaining servers. DevOps practices can enable automatic testing and monitoring of serverless functions to achieve faster SDLC and reduce manual intervention. Thus, DevOps practices are integral for managing cloud-native applications, as they enable the rapid and reliable development, testing, deployment, and scaling of individual microservices and serverless functions.

Although DevOps provides software engineers with advantages such as frequent releases of new features and fast resolution of technical issues, how to keep the speedy pace of deliveries and ensure the quality of cloud-native applications at the same time is an open challenge in the context of DevOps [BEF⁺19]. We aim to learn performance models that can help to describe the system with a simplified abstraction, further enabling simulation and forecasting for use by both developers and operators. Thus, performance prediction is a topic of great interest in the QA area with the adoption of DevOps for cloud-native applications.

Stochastic performance models, such as queueing networks [GCMM01], layered queueing networks [FAOW⁺08], and Petri nets [MBC⁺98], are widely used to represent web applications, providing explicit analysis of system performance predictions. However, there are still research gaps in accurate performance modeling for cloud-native applications. First, from the perspective of performance model specification, even if we assume that a microservice or serverless-based application has already been abstracted with an appropriate model, working on performance prediction on such a model is not trivial when models are parameterized based on real data. Based on the structure of the model abstraction, stochastic performance models often require specifying parameters for performance prediction. Model parameterization is a key factor in the accuracy of model predictions. Among the parameters of a performance model, service demand is a critical aspect that should be specified [SCBK15]. Service demand generally refers to the cumulative time a request spends receiving service from system resources, such as CPU or disk, accumulated over all visits. The accuracy of service demand specification is decisive for the predictive effectiveness of performance models. In practice, it is typically not feasible to directly collect accurate service demand data from the system at runtime. Therefore, service demand can be estimated using measurements of CPU utilization and response time obtained

through system monitoring of coarse-grained data. Several classical approaches, such as utilization law, and response time approximations based on linear regression, have been applied to capture the service demand for performance models. However, most of the existing approaches for service demand estimation mainly focus on estimating the mean value of service demand. Restricting attention to the mean can limit the accuracy of service demand estimation. Thus, estimating service demand distribution with higher-order moments can improve accuracy of performance predictions, such as higher percentiles of the response time.

To enhance the applicability to represent broader range of applications, such as serverless workflows, our focus shifts towards developing a model generation methodology that can meet the high degree of automation required in DevOps context. Cloud-native applications are service-oriented and include complex service communication and interactions. How to select appropriate performance models to abstract the system is challenging when trading off model complexity and representation power. From the perspective of system observability, while coarse-grained monitoring can be used to extract individual services as a single component with a closed queueing system, complex interactions between services and functions may not be comprehensively captured by a single queueing system. In practice, simply abstracting microservice or serverless applications as a closed queueing system is not sufficient to represent the parallel executions of functions. Complicated dependencies of interactive invocations may be structured hierarchically; ignoring interactions at different calling levels may lead to oversimplification and inaccurate performance prediction. Thus, to model complex microservice systems and serverless workflows, fine-grained performance modeling can be involved to characterize the dependencies between services. At the same time, due to serverless functions often implementing small functionalities of an application, more information can be learned to build performance model directly from their source code to reason about the internals of the applications.

However, learning fine-grained performance model structures with sophisticated calling relationships inside cloud-native applications is not trivial. Only relying on expert knowledge to depict the model structure as for traditional systems is not always feasible for large-scale application in DevOps, since high automation is required by the CI/CD pipeline of the cloud-native

applications. Manual intervention to extract the topological structure may reduce the accuracy of the performance modeling. Additionally, obtaining the complete reconstruction of the model skeleton from learning tracing or monitoring data potentially risks missing essential components or dependencies because of the limited capability of covering all executions. In contrast to monolithic or service-based applications, where larger functionalities are often aggregated behind each endpoint, the source code of individual serverless functions tends to be more focused and succinct. This characteristic makes it conducive to static code analysis for inferring additional information about the internals of Function-as-a-Service (FaaS) applications. To address the challenges associated with performance model structure extraction, static code analysis can be leveraged to infer internal information about cloud-native applications. Utilizing code-level control and data flow allows for enhancing the granularity of performance modeling and enables automatic generation of model structures. Additionally, it facilitates the integration of performance modeling into the Software Development Life Cycle (SDLC), thereby supporting accurate performance prediction and simulation. Furthermore, with fine-grained performance model extraction, end-to-end monitoring may not suffice to adequately explore data for model parameterization. Coarse-grained monitoring lacks the granularity required by the internal modeling structure. Thus, fine-grained profiling at the code level is necessary to trace the executions for parameterization during model generation.

Besides accurate performance modeling, performance diagnosis is crucial to ensure the QA of cloud-native products in DevOps contexts. It allows developers to identify bottlenecks and inefficiencies in resource usage to further improve the applications. Performance diagnosis plays an important role in detecting performance issues, enabling developers and operators to address potential faults before releasing them in production by integrating diagnostic techniques like logging, monitoring, and tracing into the CI/CD pipeline. Automated performance diagnosis also helps the developers locate internal and external failures of the applications at different observed levels. Coarse-grained-level performance diagnosis can help to detect anomalies at a single service or function granularity by tracing requests from end-to-end paths. While fine-grained performance diagnosis provides valuable insights into observing the behaviors of procedures at the code level, it is especially beneficial for developers seeking to understand the

internals of their code for performance issues detection and optimization. Distributed tracing serves as a valuable tool for extracting dependencies among various serverless functions and enhancing the detection of anomalies that may arise in their composition. This technique mainly relies on code instrumentation to capture execution data at various levels of granularity, from monitoring end-to-end request paths to observing individual code segments within a serverless function. Customized instrumentation for distributed tracing offers developers opportunities to monitor their applications at different granularities by injecting code snippets to start the tracing. However, as monitoring and tracing become more challenging due to complex calling relationships and context propagation, higher levels of automation are necessary for instrumenting serverless functions, especially with limited observability of the underlying systems. Thus, issues arise when considering where to automatically instrument the code to achieve code-level performance diagnosis and provide internal inference for developers to locate anomalies.

1.2 Research Objectives

Based on the above gaps in the current research, we define the following research objectives for this thesis.

- **RO1. Accurate performance model parameterization:** Assuming a microservice can be abstracted with a single queueing network, the first objective is to estimate critical model parameters, specifically the service demand distribution, which significantly impacts the accuracy of performance predictions. The service demand distribution can be accurately estimated using coarse-grained monitoring data directly collected from the microservice systems, thereby providing not only the mean values but also the entire distribution of service demand.
- **RO2. Automated performance model generation:** The second objective is to generate performance models for serverless workflows or applications automatically via static code analysis to extract the performance skeleton, and enhance the estimation of model parameters with fine-grained performance profiling at code level.

- **RO3. Fine-grained Performance Diagnosis:** The last objective of this thesis is on fine-grained performance diagnosis of internal anomalies through smart distributed tracing inference at the code level. This approach aims to aid developers in comprehending the intricacies of their code to detect performance issues more effectively.

1.3 Contributions

The contributions of this thesis are summarized as follows.

- To address the first research objective of performance model parameterization, **RO1**, we propose to capture higher-order moments of the service demand distribution for a microservice from monitoring traces. We first generate a closed queueing model to abstract software performance and use it to model the departure process of requests completed by the software service as a Markovian Arrival Process (MAP). This allows the formulation of an estimation problem for the service demand into an optimization problem, which aims to find the first multiple moments of the service demand distribution with the inter-departure MAP. We then estimate the service demand distribution for different classes of service with a maximum likelihood algorithm and novel heuristics to mitigate the computational cost of the optimization process for scalability. We apply our method to real traces from a microservice-based application and demonstrate that its estimations lead to greater prediction accuracy than exponential distributions assumed in traditional service demand estimation approaches for software services.
- To achieve our second research objective, **RO2**, we propose to build performance models for FaaS workflows, combining static analysis and fine-grained code profiling. To learn the topology of the model, we apply static analysis on the source code to obtain the inter-procedural call graph of the orchestrator for defining the workflow and the intra-procedural control flow for individual serverless function. To more accurately characterize the model parameterization, we propose to inject code hooks to trace system function calls during the profiling stage and capture the distribution of the service demand based on

profiling data instead of estimating based on coarse-grained system measurement. Then, we derive the performance models for serverless workflows by mapping the static graph and code profiling data. The experimental results show that it can accurately predict the performance of the application under different resource provision strategies and workloads.

- To address the final objective, **RO3**, we propose a methodology to instrument an application with code-level tracing to infer the location of faults, taking into account constraints in terms of the maximum cost of the instrumentation and testing. First, we explore the probe allocation space to assess how instrumenting different possible locations impacts our ability to localize injected performance faults. Because the space of possible probe allocations is typically too large to be exhaustively explored, we introduce a novel heuristic that exploits the control flow graph of the application to iteratively decide where to explore next to gather the most information without exceeding a maximum budget of computational resources allowed for the exploration. Second, we formulate an optimization problem to decide which probes to retain in production in order to maximize the probability of locating the internal fault by considering instrumentation latency overhead and costs. We use the traces collected during the previous phase to infer the ability of a probe placement choice to locate a fault. Experimental results show high performance in locating single and multiple faults for performance anomalies with increased latency within the provided instrumentation constraints.

1.4 Publications

The work in this PhD thesis has led to the following publications:

[Wan20] R Wang. **Learning Performance Models Automatically**. International Conference on Service-Oriented Computing (ICSOC), 2020, PhD Symposium.

We published our blueprint of the PhD plan in this paper. My main contribution in this paper is to propose to employ model discovery learning for performance models automatically

for cloud applications and provide potential solutions for performance model generation and resource demand calibration.

[WCF21] R Wang, G Casale, A Filieri. **Service demand distribution estimation for microservices using Markovian arrival processes**. International Conference on Quantitative Evaluation of Systems (QEST) 2021.

This work proposed a service demand distribution estimation method for microservices by using MAP. The estimation problem has been formulated as an optimization program to find the optimal parameters of the service demand distribution based on the measured inter-departure times. The maximum likelihood algorithm for service demand distribution estimation and novel heuristics were applied to mitigate the computational cost of the proposed optimization methods. This work covers the single-class service demand distribution estimation in Chapter 3.

[WCF22b] R Wang, G Casale, A Filieri. **Estimating multi-class service demand distributions using Markovian arrival processes**. ACM Transactions on Modeling and Computer Simulation (TOMACS), 2022.

This journal version generalized the service demand distribution estimation to multi-class services based on [WCF21]. This work applied class aggregation to map the multi-class estimation problem to several equivalent two-service class submodels. Several experiments have been performed to evaluate the multi-class service demand distribution estimation method with numerical experiments and real system traces. The discussion of the detailed contributions is in Chapter 3.

[WCF22a] R Wang, G Casale, A Filieri. **Enhancing performance modeling of serverless functions via static analysis**. International Conference on Service-Oriented Computing (ICSOC), 2022. *Best paper award*.

This paper presented a methodology for automatic performance modeling of serverless functions aided by static code analysis. Besides, solutions of fine-grained profiling were proposed to enhance the accuracy of the performance model parameterization. This work conducted exper-

iments with serverless workflows on the public cloud to verify the effectiveness of the proposed method. The detailed contribution of this work is presented in Chapter 4.

[AGW⁺21] A Alnafessah, AU Gias, R Wang, L Zhu, G Casale, A Filieri **Quality-aware DevOps research: Where do we stand?**. IEEE access, 2021.

This research paper is a joint work by our research group (QORE Lab¹). This work surveyed recent studies on quality-aware software in DevOps, in which the group members organized and analyzed the existing work based on the DevOps lifecycle stages. I mainly contributed to the DevOps testing section, which is tightly related to the background and context of my PhD thesis.

1.5 Thesis Organization

The structure of this thesis is organized as follows.

- In Chapter 1, we have presented the motivation, research gaps, and main contributions of this thesis.
- In Chapter 2, we introduce the background and preliminaries of techniques and concepts to understand this thesis.
- In Chapter 3, we present the contributions to address the RO1, where we estimate the service demand distribution with coarse-grained monitoring in microservices.
- In Chapter 4, we provide the solutions for automated performance modeling of serverless functions in a fine-grained manner, targeting the RO2.
- In Chapter 5, we mainly focus on the RO3 for fine-grained performance diagnosis with code-level tracing inference for serverless functions.
- In Chapter 6, we draw conclusions on the contributions of this thesis and provide future directions.

¹<https://qore.doc.ic.ac.uk>

Chapter 2

Background

2.1 Introduction

Adopting DevOps leads to shortened development cycles and the involvement of automation. However, with the increasing speed of the iterations of developers and operators and the high automation of cloud-native applications, the risk and the cost of quality assurance increase at the same time. Learning how to achieve accurate and automatic performance modeling for cloud-native applications is of great importance to ensure the quality of software in the practices of DevOps, which is the main objective of this thesis. Thus, to comprehensively understand the cohesive contributions of the thesis, we present in this chapter the preliminaries and background knowledge. The entire mapping details of the necessary concepts and thesis chapters are shown in Figure 2.1.

The main context of the cloud-native applications discussed in Chapters 3 to 5 includes microservices, serverless architectures, Function-as-a-Service (FaaS) and DevOps cycles. A microservice architecture is able to isolate the components in a tightly coupled monolithic application that makes it into smaller independent services that can be updated and integrated separately instead of considering the entire system. This makes it possible to achieve faster continuous integration and continuous deployment. Microservices provide the insight to design efficient applications in the context of DevOps, while serverless is an execution model to run

the different functionalities of functions on the billing mode of a single function. We focus on microservice-based performance modeling in Chapter 3 and move to discuss serverless functions in Chapters 4 and 5.

To representatively learn cloud-native systems, several techniques are involved in this thesis. First, performance models are generated and parameterized at different levels. We introduce stochastic performance models in Chapters 3 and 4 to abstract services based on microservices and serverless workflows. Analytical performance models are introduced to predict the performance of cloud-native applications by simulation and analytical solvers. From coarse-grained monitoring to fine-grained profiling and tracing, code analysis bridges the gap between the performance model representation level and the necessary monitoring information in Chapter 4 and Chapter 5. Code analysis enables lower-level model component learning and performance diagnosis. Distributed tracing is also an essential technique for studying the underlying system and aids to learn the dependencies between serverless functions.

The structure of Chapter 2 is as follows. In Section 2.2, we introduce the main background of the context of this thesis. We first present the overview of the necessary concepts of the cloud-native architecture, and then we study the current state-of-the-art and the research gaps for our research problems in Chapters 3, 4 and 5, especially from the perspective of performance modeling. Next, we move to the discussion on the performance-related techniques in Sections 2.3 to 2.5. In Section 2.3, we introduce the preliminaries of performance models, which is basic to understanding the contributions in Chapters 3 and 4. In Sections 2.4 and 2.5, we present the preliminaries of static code analysis and distributed tracing, respectively.

2.2 Cloud-native Applications

2.2.1 Microservice and Serverless

Microservice

Microservices and microservice-based architectures have been aided by DevOps in recent years

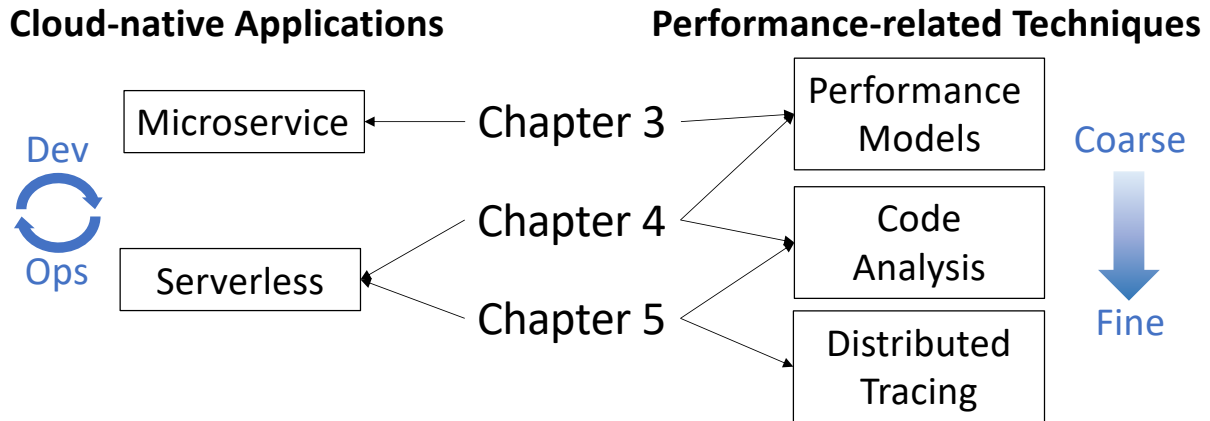


Figure 2.1: Context and techniques mapping to thesis chapters

for software development and system design. Microservice-based architectures are an architectural design consisting of a set of loosely-coupled microservices. Shifted away from monolithic architectures, in microservice-based architectures, the application is decomposed into smaller and independent units for pieces of business logic. The communication between different microservices is typically implemented with defined APIs through networks, for example, using lightweight protocols like REST API.

Additional orchestration mechanisms are required for coordinating the interactions between different microservices. Technologies like *Docker*¹ and *Kubernetes*² are often used to package and manage microservices in *containers* for easier deployment and management. *Docker* is the orchestration platform for developing and running applications in containers. *Kubernetes* is a powerful open-source container management platform developed by Google, which uses a cluster of master and worker nodes. *Containers* are executable packages containing all the necessary to run the application code on different computing environments. All the source code, runtime, tools, libraries, dependencies, and metadata configurations are packaged in containers to support executions across different servers or platforms.

We provide an illustrative comparison between monolithic and microservice architectures in Figure 2.2. The microservice application in the example is implemented on Kubernetes within Docker and containers. Transferring from monolith to microservice, each service is isolated

¹<https://www.docker.com/>

²<https://kubernetes.io>

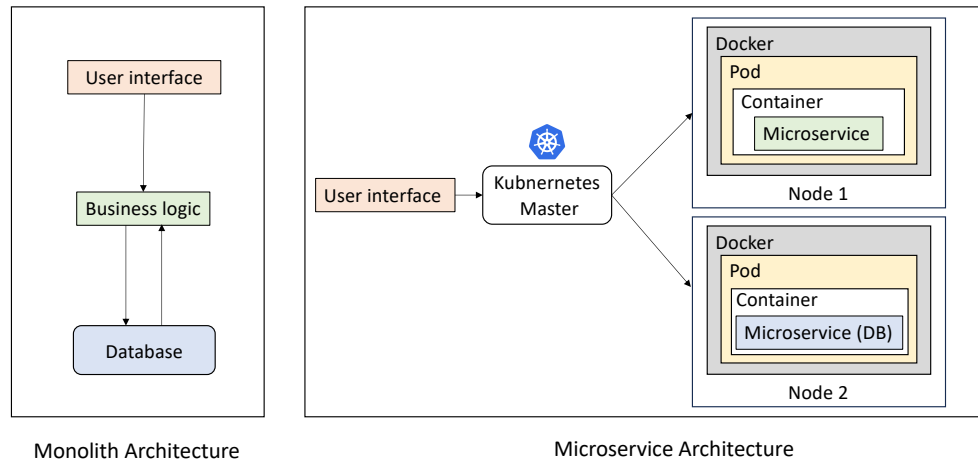


Figure 2.2: Monolith architecture V.S. Microservice architecture

with a microservice (including the database) packaged in a container running with pods. With Kubernetes, containers can run with the Docker engine on each worker node. The combination of microservices, Docker, and containers aligns well with DevOps practices and CI/CD pipelines, enabling rapid and reliable application development and deployment.

There are several advantages to adopting microservices in a DevOps context. On the one hand, from the perspective of developers, software failures are isolated in each microservice that would not impact the whole system. Besides, to accelerate a CI/CD pipeline, it is easier to adapt and change individual microservices without altering the entire design. Smaller teams can work on individual microservices simultaneously, allowing for the use of the best-suited technology for each microservice. On the other hand, from the users' and operators' points of view, microservice-based applications can be scaled and managed individually, which is cost-effective and flexible.

In summary, microservices and microservice-based architecture provide a way to build complex, scalable, and flexible systems by breaking them down into manageable and independent components. Although microservice architectures provide software engineers with advantages like frequent releases of new features and fast resolution of technical issues, keeping a speedy pace of delivery to production and ensuring the quality of the software at the same time remains an open challenge in practice. The resource usage of microservices varies depending on the implemented functionality and the changing workloads. Testing and debugging distributed

systems can be more challenging with decomposed services. Robust monitoring and tracing are crucial for understanding the health and performance of microservices. Suitable abstraction of microservices is also important to the performance prediction of applications and systems.

Serverless and FaaS

In serverless computing, cloud providers offer automatic infrastructure management and provision to execute the function code. It has recently gained increasing attention in the industry due to the potential for significant cost savings and on-demand billing modes.

Function-as-a-Service (FaaS) is a cloud computing execution model introduced within serverless computing that allows developers to deploy single functions or pieces of code as basic building blocks [BCC⁺17]. Compared to monolithic applications and microservice-based architectures, FaaS-based applications can be triggered and served by events, such as HTTP requests, time triggers or the status of the object storage. Serverless platforms are able to automatically scale the number of function instances in response to workload changes, where the billing is at the granularity of single functions.

There are several cloud vendors that offer serverless platforms and FaaS solutions. AWS Lambda³ is an Amazon Web Services serverless platform, supporting a wide range of event sources and integrations. Azure Functions⁴ is Microsoft Azure FaaS solution that provides similar capabilities to AWS Lambda. Google Cloud Functions⁵ is designed for building and deploying event-driven functions. Besides, there are also open-source alternatives such as OpenFaaS⁶ or KNative⁷ for developers and users to design and execute serverless functions.

Serverless workflows enable orchestrating multiple serverless functions and services into complex, event-driven workflows. Classic scientific workflows can be transferred to serverless execution by decomposing different functional units into single and individual serverless functions. We briefly introduce an illustrated example of serverless workflow for AI-driven image classifi-

³<https://aws.amazon.com/lambda>

⁴<https://learn.microsoft.com/en-us/azure/azure-functions/>

⁵<https://cloud.google.com/functions>

⁶<https://www.openfaas.com>

⁷<https://knative.dev>

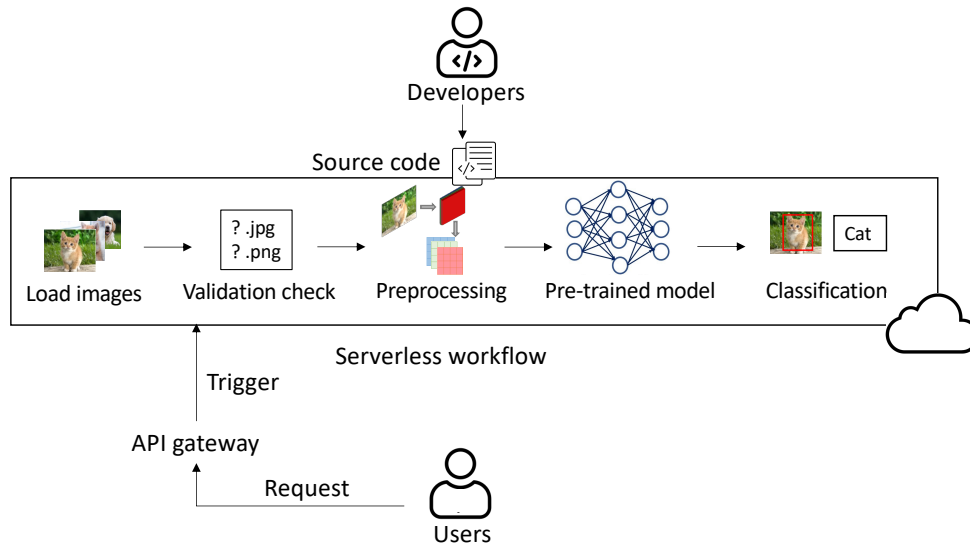


Figure 2.3: Illustrative example of a serverless workflow

cation in Figure 2.3. The development of individual serverless functions requires defining the order of function execution and the flow of data inside the serverless workflows. There exist solutions for serverless workflow orchestration and definition on the cloud platforms. For example, AWS Step Functions, Azure Logic Apps and Azure Durable Functions are designed to run workflows that connect to various serverless services by Amazon and Microsoft. Serverless workflows are particularly useful for automating data pipelines and complex tasks that involve multiple steps and services.

FaaS platforms enable automatic management, scaling, and billing of the execution of FaaS-based workflows to take over most operational efforts from developers and users. However, maintaining the QoS requirements and meeting service-level agreements (SLAs) of FaaS applications remains an outstanding concern [TPN⁺20].

2.2.2 Related Work

As microservices have become a prevalent architectural style for modern applications, understanding their performance implications and optimizing their execution has become a key research area in both industry and academia. This section investigates the existing literature in recent years related to performance engineering of microservices.

Traditional performance engineering techniques may not directly apply to microservices due to their unique characteristics compared to monolithic architectures. Tapia *et al.* [TMF⁺20] present a comparative study of the performance between monolithic systems and microservice-based architectures, pointing out that the necessary to address the challenges in performance measurements and performance testing in the migration from monolith to microservices. In [HVHK⁺17], the authors discuss the need for new solutions in performance engineering for microservices, as well as the open issues and possible research directions in automated performance testing, monitoring, performance measurements and modeling of microservices. In [EBS⁺20], the authors discuss the importance of measuring the performance of microservices and monitoring their behaviors. Metrics such as response time, throughput, and error rate are used to evaluate the performance of individual microservices and the system as a whole. Monitoring solutions for microservices are integrated without the knowledge of implementation. In [GLHR18], Gotin *et al.* investigate the performance metrics for microservices in CloudIoT environments. The authors present a case study to evaluate the performance metrics including microservice CPU utilization, message queue length and queue delays. They explore the vulnerability of each metric to variations in the microservice characteristics. In [BS20], the authors propose a black-box monitoring approach to efficiently track the runtime performance of microservices in cloud computing environments. This approach focuses on microservice performance metrics at different levels, such as power consumption, application performance (CPU usage, execution time), and network performance to generate accurate metrics with less overhead.

Microservice prediction and modeling are essential for optimizing the performance and resource allocation in microservices architectures. By analyzing historical data and patterns, it can enable predicting future workloads and resource demands, allowing for proactive scaling and efficient resource allocation. In [CSOQ21], the authors introduce a microservice performance model and a methodology to automatically derive model instantiating values from a single execution trace by simulation, which can predict the performance of large-scale applications. In the proposed performance model, they mainly apply queueing time, I/O time, CPU usage and service outputs to capture the microservice behaviors. Besides, [JPG19] focuses on performance

modeling for cloud-based microservice applications, in which the authors explore the performance modeling to estimate the capacity of microservices for optimizing the performance of the dynamic software architectures. In [RL19], the authors introduce a predictive framework designed to estimate the end-to-end tail latency of containerized microservices, providing valuable insights for performance optimization and ensuring consistent, responsive user experiences. In [GSC⁺21], the authors present a framework called SuanMing, which is designed for predicting and explaining performance degradation issues in microservice applications. SuanMing employs machine learning techniques to analyze historical performance data and generate predictions about potential future performance problems. The system's key strength lies in its ability to provide explainable insights, helping operators and developers understand the root causes of performance issues. In [ZHZ⁺21], Sinan, a resource management system for cloud-based microservices is introduced. Sinan utilizes machine learning techniques to optimize resource allocation in microservices environments, with a focus on ensuring the QoS requirements. The system aims to improve the efficiency and reliability of microservice applications by dynamically allocating resources based on real-time performance data and QoS metrics, thereby enhancing the overall user experience. In [AIB⁺20], the authors address the critical challenge of efficiently managing containerized microservices in cloud environments. This paper introduces a novel approach to autoscaling, leveraging predictive algorithms that take into account bursts in traffic or demand to adjust resource allocation. [KL20, PCLH21] explore different strategies including machine learning-based, neural network-based methodologies to predict the performance of microservice-based applications for resource optimization.

The question of how to predict the performance of serverless functions is closely followed by researchers. However, fine-grained analytical performance modeling for serverless functions still lacks investigations to our knowledge. Eismann *et al.* [EGVE⁺20] propose to use mixture density networks to predict the response time distribution of a single serverless function and then estimate the cost of serverless workflow execution by Monte-Carlo simulation. In [ARIM20], the authors develop a framework called COSE for serverless function configuration with a trace-based performance model. Based on the performance model, they apply Bayesian Optimization to obtain the optimal serverless function configuration. These works can be identified

as data-driven performance predictions for serverless functions, which may not give an explicit, interpretable abstraction of a serverless application.

On the other hand, model-driven performance prediction can help developers and providers to better understand different performance predictions and reason about performance issues or design alternatives. Boza *et al.* [BAV⁺17] propose to use $M(t)/M/\infty$ queues to model serverless functions, enabling the calculation of performance and cost. Mahmoudi *et al.* [MK20a, MK20b] propose an analytical performance model by using a continuous-time semi-Markov process to accurately predict the performance metrics. However, this work mainly focuses on modeling aspects of the computing platform to support tuning its configuration, and does not directly relate to the internals of serverless functions. Lin *et al.* [LK20] use probabilistic directed acyclic graph abstractions to predict the end-to-end response times of serverless applications. The smallest representable unit in this work is a whole serverless function, which may limit the performance prediction accuracy due to the coarse modeling granularity.

2.3 Performance Modeling

This section is a preliminary to performance models, which are used for representative abstraction of cloud-native applications in Chapters 3 and 4. We first provide an overview to introduce the basic motivation to use the discussed performance models in the following chapters and provide illustrations from queueing systems to layered queueing networks. Next, we study the state-of-the-art focusing on performance models generation and parameterization.

2.3.1 Overview

Learning performance modeling is the discovery process of building performance models based on studying information about system obtained via testing and monitoring. Building performance models is quite useful in DevOps practices since it can provide the possibility to answer a series of what-if questions about system performance, e.g., related to changing user behaviors,

mixtures of workloads or even about the impact of updates to functions and methods inside the applications. Besides, performance modeling can also help to speculate about a system's structure without looking deep into each component development, especially in the DevOps lifecycles. For instance, from the point of view of the developers, they can analyze and predict the system performance based on simulation results in each DevOps iteration while calibrating the accuracy of performance models as needed. Even if the developers have little clues about the architectural design or implementation choices behind a system, they can still learn useful information based on the components in performance models. Moreover, it also benefits the cloud providers who own the microservices or serverless platforms or infrastructures. In general, the providers are not aware of the internals of the cloud applications (e.g., source codes). Thus, building performance models through monitoring of the deployed applications can help generate basic performance models with limited information about the applications. The infrastructure providers can infer the customer usage of microservices and achieve resource management with such performance prediction models.

There are mainly two open research questions on building performance models, including the model abstraction level and the model parameterization. The abstraction level and model skeleton capture the structure of the underlying application, while accurate model parameterization is decisive in obtaining accurate quantitative performance prediction results.

Queueing Networks

Assuming a software system (e.g., client-server system) working with service hosting on a service provider's facility (e.g., CPU), the customers visiting the provided service are the user source of the system. New requests from the user source may wait in a queue to be served by the server. Such a simple system can be abstracted with a basic queueing network (QN) as follows [LZGS84].

If the jobs of the user source arrive externally and leave the system after receiving the service, we can model the system as an *open queueing network*. While a *closed queueing network* can represent systems with only internal arrivals and departures, where the total number of jobs in

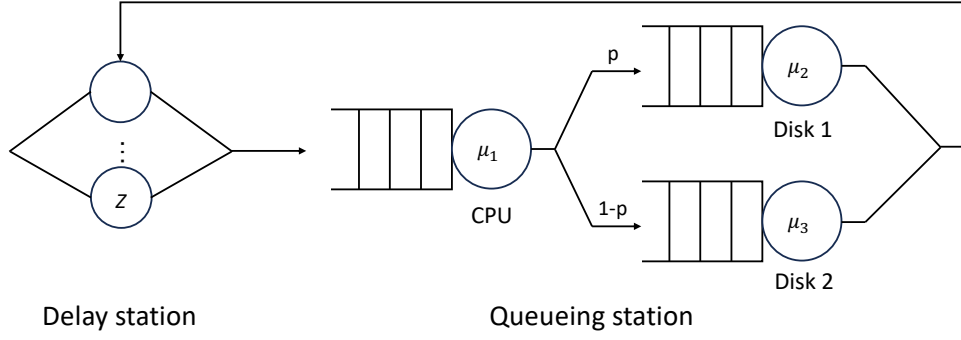


Figure 2.4: Example of a closed queueing network

the system is constant. A *mixed queueing network* is open for some workloads and closed for others. Many computer systems exhibit this interactive behavior, in which users interact with a system via threads contending for access to a CPU and processes blocking for I/O, and then the servers wait for a response. In the following chapters, we mainly utilize closed queueing networks to abstract components in cloud-native applications. Thus, we provide an illustrative example of closed queueing networks in Figure 2.4 for a better understanding.

In this closed queueing network, there are two stations for the user source and the server, which are called *delay* and *queueing stations*, respectively. The delay station initially captures the total number of jobs in the systems, and jobs will depart from the delay station after specified think times to simulate the time between requested tasks. The jobs can belong to different service classes to represent different requirements on the resources. Typically, queues are abstractions of service facilities, representing the serving capacity of different resources. The service queues have scheduling policies indicating the order of the waiting jobs to be served, for example, First-Come-First-Serve (FCFS) and Processor-Sharing (PS). For example, in 2.4, if there is a job arrival at the queueing station, it will be first served by a CPU queue and then moved to one of the other two queues of the disks. After being served by all required servers, jobs depart from the queueing station and arrive again at the delay station.

Besides its components and topology, a complete queueing system is described by several parameters. Example parameters are shown in Figure 2.4. First, for the delay station, the think times between different jobs arrival at the queueing station can be specified with the parameter z , which is often defined as the mean value for an exponential distribution, thus simulating

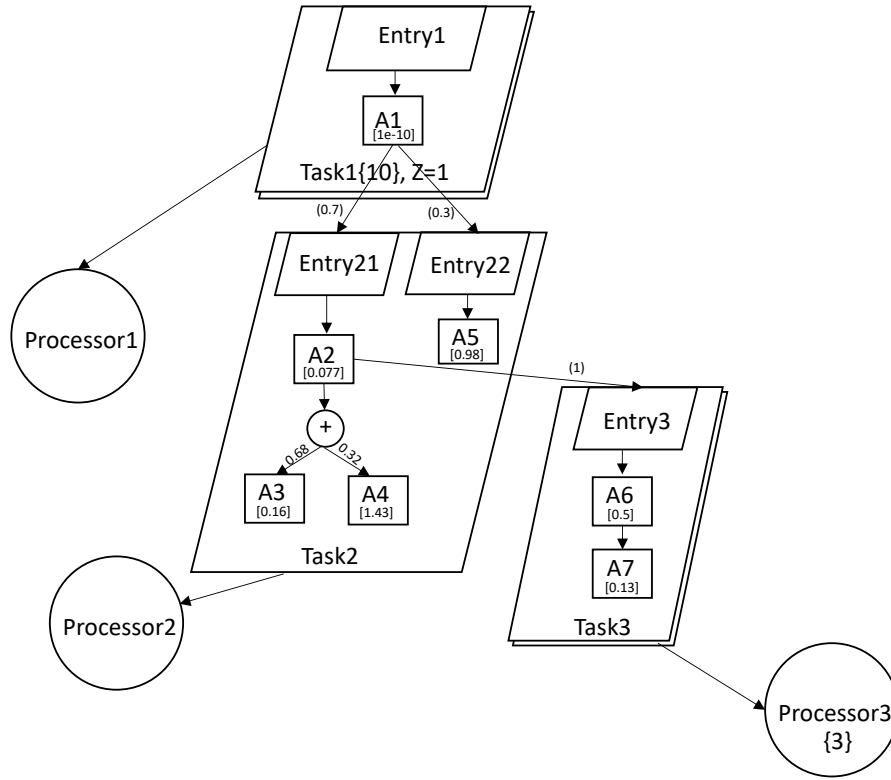


Figure 2.5: An example LQN model

the arrivals of jobs according to a Poisson process. Among the parameters of a performance model, service demand is a critical factor that impacts prediction accuracy [SCBK15]. Service demand can be defined as the total service time of a job across multiple visits to be served at a specific resource. For example, in Figure 2.4, $1/\mu_1$ is the service demand for the CPU with service rate μ_1 if we assume the service demand is also exponentially distributed. Transitions can leave from a server to others to complete the job with a specified probability, where the out probabilities should sum to one for each node.

Layered Queueing Networks

LQNs are an extended queueing network formalism that has been widely used to abstract web applications [ILFW05]. The main components of LQN models covered in this thesis are shown with an example in Figure 2.5. The large parallelograms, denoted as *Task*, represent software and hardware entities. There are mainly two types of tasks: a task representing the clients, which is known as a reference task, and tasks representing the servers processing incoming requests. Tasks are hosted on resources that are denoted as processors in the circle, and

multiprocessor hosts can be specified with a multiplicity figure, e.g., in Figure 2.5 with $\{3\}$ for Processor 3. Smaller parallelograms inside a task are called *Entry* and represent different service classes provided by a task (e.g., different endpoints). The detailed operations inside each entry can be described with a set of *Activity* specified with their execution order, where rectangular nodes for activities and circular “+” nodes represent probabilistic choices. Each activity is parameterized with service demand, for example, specifying the mean value of the exponential demand distribution represented by the required resources. Activities can make requests to different entries by sending synchronous or asynchronous calls by defining the call mean value that is on average how many calls are made. For instance, in the top task, 10 concurrent clients are sending synchronous requests to *Entry21* and *Entry22* and the arcs are labelled with the value of the mean number of requests.

2.3.2 Related Works

We mainly study the related works on service demand via the estimation of model parameters and automatic model generation, which is important to RO1 and RO2 in this section.

Service Demand Estimation

To enable accurate performance modeling, model parameterization brings out an important problem of service demand estimation. Statistical methods can estimate service demand based on regression and optimization. From [SCBK15], we can learn that regression is one of the most commonly used statistical methods to solve parametric estimation problems. Generally, linear regression has been employed to solve service demand estimation problems, mostly based on monitoring measures like utilization and response times. Rolia *et al.* [RV98] introduce the resource demand estimation problem with linear regression techniques. In general, linear regression has been employed to estimate service demand, mostly based on utilization [CCT07, CCT08a] and response time [PPSC13]. There are different approaches adopted by researchers to solve the linear regression to estimate utilization, such as least trimmed squares regression in [CCT07, CCT08b], least squares, least absolute deviations regression and support vector regression in [KKRR11]. However, most of the mentioned approaches based on regression only

allow obtaining the mean value of the resource demand instead of full distributions, lacking higher-order properties of the demand.

With the development of machine learning, many researchers adopted machine learning-based algorithms (MLAs). In [CDS10, SBC⁺08, AGEK18, GEE⁺19], MLAs including decision trees, support vector machines, k-nearest-neighbors, linear regression, artificial neural networks and random forest were used to represent the dependencies and learn to estimate service demand automatically. Neural networks are applied to estimate service demands for the ability to predict time series data. For example, in [DMD⁺17a], an RNN-based CPU utilization prediction has been proposed. Deep learning methods have also been adopted to improve the accuracy of service demand estimation. For example, in [IBC⁺20], the authors provide a sliding window selection method based on deep learning to select the training data and then train neural network models for utilization estimation. Moreover, machine learning can also help select the optimal approaches for estimation on account of a variety of existing resource demand approaches. In [Gro16], the authors propose using several MLAs to predict the suitable approach for service time estimation at runtime. The optimal estimation approach selection was defined as a classification problem. In [CZDF15], the authors present a fuzzy neural network-based ensemble method to choose the optimal model for resource prediction on the cloud. In [IBE⁺19], the authors consider the random decision forest (RDF) to be the classifier to select the best prediction model for resource demand estimation.

Neural networks like recurrent neural networks (RNNs) may also be applied to estimate resource demands for their ability to predict time series data [DMD⁺17b]. Machine learning can also help to select the optimal approaches for estimation via varieties of existing resource demand approaches [LZSL17]. However, also, in the case of MLAs, the existing methods often benefit the predictive capabilities for the service demand with historical data that lacks explainable depictions of the full distributions to be specified in the representative performance models.

There are many examples of works modeling systems with a Poisson arrival process and an exponentially distributed service time [BLP10, KML15]. However, the information captured by the service demand distribution is not leveraged. This information is especially important under

FCFS scheduling, where mean demands are no longer sufficient to characterize performance exactly as in PS.

Performance Modeling

There are several existing works on performance modeling, which can be typically classified into model-driven and data-driven methodologies.

In [MADD04], Menasce *et al.* provide border insights into how performance modeling can help to ensure the QoS of computer systems, including analytical models and capacity planning methodologies. For example, using QN models to abstract the database service and the web server for performance measurement analysis. In [HB13], the authors provide an overview of queueing theory and its application in modeling and analyzing the performance of computer systems.

Directly building performance models is not trivial since the software systems may not be designed with performance-related patterns. However, developers usually adopt software architecture models at design time to depict the components of the system [CFD⁺11]. Thus, several works focus on model transformation from design-oriented models to performance-related models. In [CM00], the authors propose to jointly use different Unified Modeling Language (UML) diagrams to extract model components and user profiles to generate an extended QN model for performance analysis. In [PS02], they study a model transformation for XML-based UML models to LQN models, which generates the model skeleton from the high-level software components and specifies the LQN parameters.

The Palladio Component Models (PCM) [BKR09] is another open-source modeling framework primarily used for modeling and analyzing performance from system architectures. In [KR08], Philipp *et al.* present the automated model-to-model transformation from PCM to Queueing Petri Nets for component-based architecture. Besides, in [PC13], Juan *et al.* generate LQN models automatically from PCM instances by mapping the PCM components to the LQN formulation and apply fluid analysis to obtain the performance measurements. Recently, TOSCA specifications have been extended to specify several concerns of serverless applica-

tions [ZGTC21] and can be used to generate LQN performance models.

However, existing methods for generating architecture-level models like UML and PCM rely on manual analysis and domain knowledge, which hardly support high-degree automation expected in DevOps. In addition, the description languages of architectural models in previous works are independent of deployment, which brings complexity to frequent deployment and automatic calibration of performance models responding to new alternative deployments.

Data-driven performance modeling is able to predict system performance-relevant metrics relying on historical performance measurements from the monitored software systems. In [SRSG⁺15], the authors develop a novel method for utilizing rich process logs to analyze the performance of scheduled processes. The proposed method combines simulation, queueing analysis, and statistical methods, which is the discovery of an individual-case model from data based on an extension of the Petri Nets formalism. Yadwadkar *et al.* [YHG⁺17] introduce a data-driven system that uses a novel hybrid offline and online data collection and modeling framework to provide accurate performance estimates with minimal data collection to predict workload performance for different user-specified metrics of a VM placement. Zhang *et al.* [ZGJ⁺17] propose to predict performance characteristics based on quantitative assessment of expert judgments, and the ranking of design alternatives is achieved based on historical product design data. Based on the historical data from the systems, machine learning algorithms are adopted for performance modeling to predict the system measurements. For example, in [MBV⁺18], they study several machine learning methods for modeling the performance of representative scientific applications on high-performance computing platforms with the collected data.

Most of the existing data-based performance modeling methods regard the entire system as whole to predict the performance under given workloads, regarding the system internals as black boxes. However, this may increase the estimation costs when the workload changes since it requires retraining the model with new data.

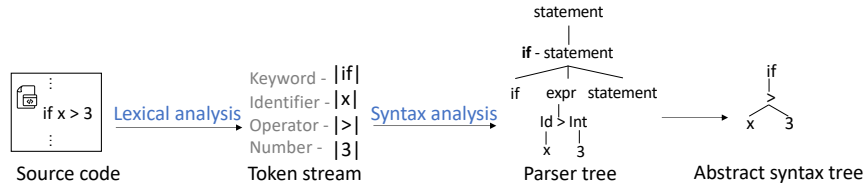


Figure 2.6: Flow chart of AST generation from source code

2.4 Static Source Code Analysis

Static analysis is widely used to infer information about a program by reasoning on the structure and features of its source code, or convenient intermediate representations, without actually executing the program [NNH15]. It is opposed to dynamic analyses that require executing the program. For example, it can infer which statements in a program affect the value of a variable at a specific line. Typically, with access to the source code, lexical analysis can first be applied to extract the tokens (keywords in the programming language) from the code text. Each token represents a unit of an instruction. Tokens are then rearranged into an abstract syntax tree (AST) by syntax analysis, resulting in a tree structure where nodes are the instructions and edges are the relationships based on the programming language grammar. For instance, the AST make explicit the presence of a function call, the related input arguments, and the instructions composing the function. Thus, an AST is a data structure used to reason about the grammar of a programming language in the context of the instructions provided in the source code.

Two widely used representations that can be statically extracted are control and data flow graphs. Static code analysis works by traversing all the possible paths that could be executed at runtime, constructing in a graph notation, a control flow graph (CFG). CFGs capture (a superset of) all the possible paths that can be executed at runtime. A CFG represents how the evaluation of conditional statements (e.g., branches and loops) determines the next code block to be executed. At the intra-procedural level, CFGs represent the dependencies between code blocks and all the possible execution orderings, subject to the decisions at conditional nodes. Intra-procedural CFGs can be related to one another via the program call graph. A call graph (CG) captures all the callee functions it can invoke for each caller function, providing an

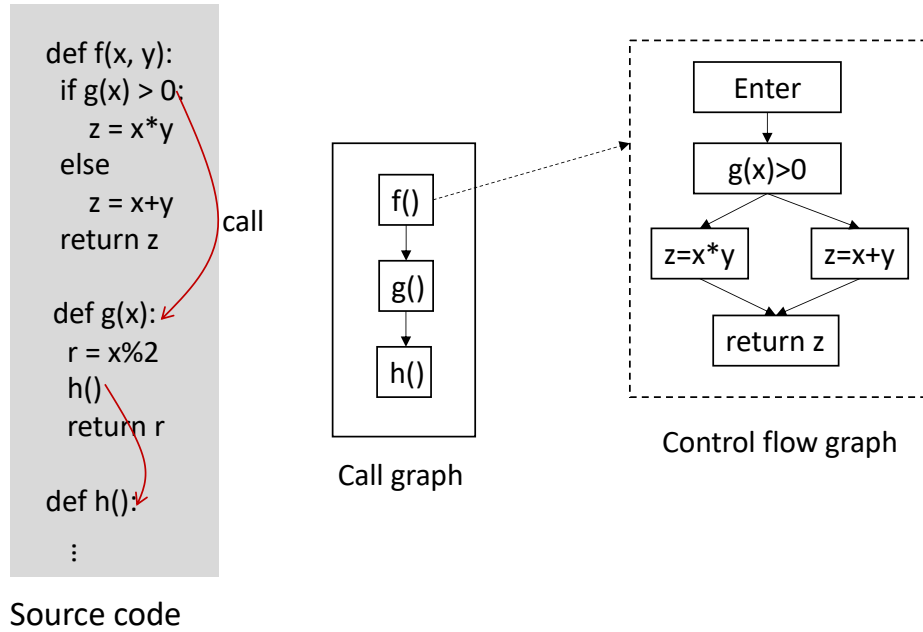


Figure 2.7: Illustrative example of the generation of call graph and control flow graph

inter-procedural representation of the dependencies and interactions among functions. There is an example of the generation of CFG and CG from a piece of simple source code in Figure 2.7. We can learn in the source code that function `f(x, y)` contains a function call to function `g(x)`, and inside `g(x)` there is a call to function `h()`. Then, we can extract the relationship between different function calls and obtain the inter-procedural information to construct a call graph. For example, in function `f(x, y)`, there are branches conditional on the return value from `g(x)`. Different procedures of `f(x, y)` are depicted with the execution order of individual statements in the control flow graph.

Instead, a data flow graph represents the propagation of information throughout program statements and variables [KSK17]. Static data flow analysis, for example, can compute execution paths that propagate values of interest from their sources (e.g., where the data is received from the user) to sinks (e.g., where data is stored in a database). Data can also flow through dependent nodes of a CFG, e.g., through the arguments of a function invocation or the decision at a conditional node.

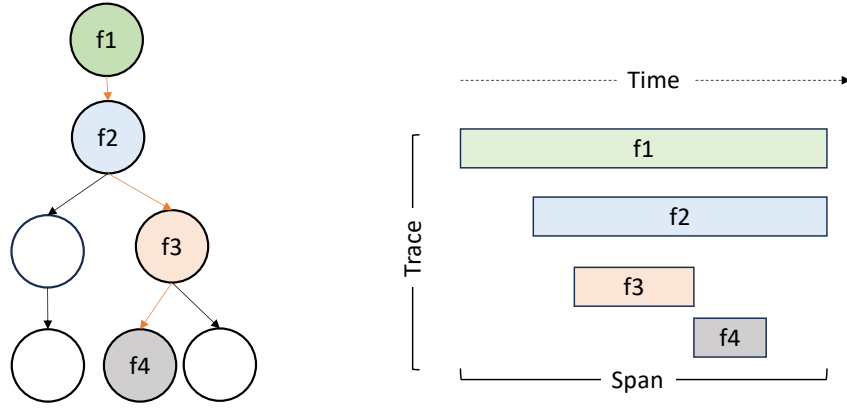


Figure 2.8: Example of distributed tracing concepts of the given function workflow

2.5 Distributed Tracing

Distributed tracing provides observations of requests for services that propagate through distributed computing environments. It is critical to complex cloud-native applications, for example, in the context of microservice [LPX⁺22, YCC⁺21] and serverless architectures [SKM22] for health checks, performance issues diagnosis and debugging. Distributed tracing is enabled by metadata context propagation to collect informative data about the monitored system.

We first show an example of distributed tracing concepts with a given workflow in Figure 2.8. *Trace* and *span* are core components in distributed tracing to understand the behaviors of applications. A trace provides a high-level description of how the request passes through various services that can communicate with each other in an application. A full execution path of each transaction is constructed by a trace in which all operations are needed to be executed. Each trace can be defined by a unique identifier and consists of one or a group of spans. A span is a representation of a logical operation involved with processing a request or transaction. Spans can expose the execution metadata of current working units, such as the starting and ending timestamps, the execution status, operation names, etc. Spans can also be nested to represent constituent sub-operations that enable an arbitrarily fine-grained capture of the orders of processed procedures. The relationship between nested spans can be depicted in a hierarchical parent-child structure, where a parent span is broken down into (possibly overlapping) segments by its children spans. Correlating the composed spans with

context propagation enables the trace data to track the whole execution. In distributed tracing, context propagation refers to the ability to pass contextual information across different services or components involved in a distributed system. This context allows for tracking a specific transaction or request as it traverses multiple services or microservices.

Metrics are numerical values that represent the state of a system at a sampling point. It can include response times, error rates, and system resource utilization logged in a trace. Metrics play a vital role in distributed tracing, offering a quantifiable way to monitor and analyze the performance of various services within a distributed system.

Tracing data can be generated with either black-box approaches or annotation-based systems [SBB⁺10]. Black-box distributed tracing tools take the instrumentation control from the users and offer tracing agents to be set up on their infrastructure. In contrast, annotation-based tracing schemas rely on the instrumentation of application code with common libraries to sniff and preserve customized tracing information.

Distributed tracing can be customized by developers and users with code instrumentation, i.e., by adding code to an application or service to capture trace data. The instrumentation code is responsible for creating spans, adding trace context to outgoing requests, and recording timing information. OpenTelemetry⁸ is an open-source platform for instrumentation and data collection. It offers vendor-neutral auto-instrumentation libraries and APIs that allow you to trace end-to-end requests. However, OpenTelemetry does not have any built-in analysis or visualization tools. Zipkin⁹ and Jaeger¹⁰ are examples of other open-source tools with user interfaces that can visualize distributed traces and store tracing data in the database.

⁸<https://opentelemetry.io>

⁹<https://zipkin.io>

¹⁰<https://www.jaegertracing.io>

Chapter 3

Service Demand Distribution Estimation

3.1 Introduction

Stochastic modeling of microservices is often used in simulation-based and analytical evaluations of system performance. A classic approach to software performance prediction relies on Markovian models, which can help to describe the system analytically and efficiently simulate and forecast performance as needed by both developers and operators. Stochastic models such as queueing networks [GCMM01], layered queueing networks [FAOW⁺08] and Petri nets [MBC⁺98] have been widely used to model web applications. Similarly, software architecture models are appropriate to describe changes in software components and resources [MK18a], which can be used in conjunction with stochastic models to holistically describe the quality of service in a complex distributed system.

However, choosing the correct model parameters is challenging for the predictive accuracy of performance models. In this paper, we focus on *service demands* which are model parameters describing resource consumption for a *single* request to a processing unit such as a CPU, a GPU, a disk or other resource. This slightly differs from defining service demands as the cumulative time a request spends receiving service from a resource, accumulated over all visits prior to

completion. Demands are usually estimated by applying regression methods to measurements of resource utilization and response time collected via system monitoring. Several different approaches for service demand estimation have been proposed over the years, such as utilization law [CCT08a], response time approximations based on linear regression [PPSC13], non-linear optimization [WHQ⁺12], as well as machine learning methods [DMD⁺17b]. Most of the existing approaches for service demand estimation mainly focus on estimating the *mean* service demand, as opposed to considering the *Service demand distribution*, which is the probability distribution of the demand of time consumption on a specific server without any queueing time. This is different from the response time distribution, which captures the amount of elapsed time from when a request is sent to the time it is completed by the server, including both the service time and the time for queueing. However, as we show in this paper, restricting attention to the mean can limit the accuracy of performance prediction. For example, higher-order moments can affect the predictive accuracy for critical metrics, such as higher percentiles of the response time.

As applications are nowadays updated directly while running in production, to build useful performance models for software services, besides accuracy, it is also important to derive an analytical model directly from runtime data. In this work, we focus on microservice-based systems, which are widely used in industry. To learn the service demand distribution, we first model each microservice as a closed queueing system, with the finite closed population representing the maximum parallelism level for served requests due to the microservice admission control. Within this queueing system, the service demand for the queueing station is characterized as an acyclic phase-type (APH) distribution [NM81, HT02]. After generating the continuous-time Markov chain (CTMC) for this model, we filter the departure transitions into a Markovian arrival process (MAP) to characterize the inter-departure times of requests served by the microservice [Neu92, Cas11]. The problem of service demand estimation can then be formulated as an optimization problem to infer the service demand distribution (i.e., the APH) that maximizes the likelihood of the departure MAP having produced the collected runtime data. In particular, the optimal parameters of the service demand distribution are obtained from matching moments of the APH distribution by a global search with the maximum

likelihood estimation procedure.

Since the likelihood function may be non-convex, to overcome the high cost of global optimization, we propose a heuristic estimation method. In this approach, the estimation is divided into sub-problems aimed at fitting different moments. The required measurements for fitting consist of the inter-departure times, response times and the time of departure, which can be directly collected via network traffic sniffing from pairs of arrival and departure events for requests processed at the microservice.

Based on the work on estimating the service demand distribution for a single service, we then extend our heuristic estimation method to multi-class service processes that can characterize software systems exposing multiple types of services, e.g., the different microservice endpoints. To estimate the service demand for multi-class service queues, we apply class aggregation to simplify the multi-class estimation problem into a sequence of two-class sub-problems that can be used to approximate the initial model. Class aggregation for multi-class queues can effectively help to reduce the computational difficulty of solving CTMCs in the presence of state space explosion. Then, we propose to estimate the service demand distribution for multi-class services with the heuristic method, which is extended to matching the first three moments for different service classes with class aggregation.

To evaluate our method, we validate it against real traces with a single service class, which are obtained from a microservice-based application. The results show that our method can fit the distribution of real traces with a high degree of accuracy. We also evaluate our extended work on real traces collected from the microservice application with different numbers of service classes. The experimental results indicate an accurate fit performance of the proposed model prediction to the real traces.

The rest of the chapter is organized as follows. In Section 3.2, we recall the necessary background and definitions. In Section 3.3, the inter-departure time model and problem formulation are introduced. In Section 3.4, we discuss our proposed service demand distribution estimation method with global optimization and the heuristic method for a single service class. The multi-class service demand distribution estimation method is introduced in Section 3.5. We present

our experimental results in Section 3.6 and Section 3.7, respectively. Then, we discuss the threats to validity in Section 3.8. Finally, we draw conclusions in Section 3.9.

3.2 Preliminaries

Acyclic Phase-type Distribution. A phase-type (PH) distribution [O0] is defined as the distribution of absorbing time in a continuous-time Markov chain (CTMC) with finite states $\{1, 2, \dots, w, w + 1\}$, where the first w states are transient, and the last one is absorbing. The infinitesimal generator matrix of this CTMC is

$$\mathbf{G} = \begin{bmatrix} \mathbf{T} & \mathbf{t} \\ \mathbf{0}^T & 0 \end{bmatrix}$$

The sub-generator $\mathbf{T}$ has dimension $w \times w$ and specifies the transition rate from state i to state $j \neq i$ when this does not lead to an absorption. We also define $\mathbf{t} = -\mathbf{T}\mathbf{e}$, where $\mathbf{e}$ denotes a column vector of 1 with appropriate dimension. We can further describe the stationary distribution of the transient states with $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_w)$, subject to $\boldsymbol{\alpha}\mathbf{e} = 1, \alpha_i \geq 0$. A PH distribution may thus be compactly specified as $PH(\boldsymbol{\alpha}, \mathbf{T})$.

An acyclic PH (APH) distribution [BHT05] is a subset of PH distributions with an acyclic underlying Markov chain. This implies that any state in the underlying Markov chain cannot be visited more than once before absorption. If a random variable Y has APH distribution with parameter $\boldsymbol{\alpha}'$ and $\mathbf{T}'$, we write $Y \sim APH(\boldsymbol{\alpha}', \mathbf{T}')$.

Service demand distribution modeling. In this paper, the service demand distribution is modeled as an APH distribution. This is because Markovian distribution models such as the APH are flexible for moment fitting and composition with CTMCs. Given a set of moments, the parameters of an $APH(\boldsymbol{\alpha}, \mathbf{T})$ may be obtained with various PH distribution fitting methods [HT02]. In this work, we will use the method of moment matching, which can fit the parameters to match an arbitrary number of moments of a reference on empirical distribution, so that the service demand distribution is tackled by finding moments to fit an APH. In par-

ticular, we consider using the first three moments to study the APH distribution of the service demand. The third moment (skewness, ν) is considered for its characterization of the fitting performance of the end of the tail

$$\nu = \frac{E[(X - m)^3]}{(E[(X - m)^2])^{3/2}} \quad (3.1)$$

Here, the random variable X denotes the service demand placed by a request at the resource and its first three moments may be given in terms of the mean (m), the squared coefficient of variance (SCV, c^2) and the skewness (ν), which are related as follows:

$$E[X^2] = (1 + c^2)m^2 \quad (3.2)$$

$$E[X^3] = \nu(c^2)^{3/2}m^3 + 3m^3c^2 + m^3 \quad (3.3)$$

Thus, we assume that the service demand distribution may be approximately fitted from knowledge of the first three moments $E[X]$, $E[X^2]$ and $E[X^3]$ which in turn are described by parameters m , c^2 , and ν . Our analysis does not rely strongly on the number of moments chosen and may be generalized to situations where more than three moments are used to fit an APH distribution.

Markovian Arrival Processes. PH distribution provides a more general way to capture the service demand distribution, however, it is not always feasible to obtain the measurements from the system that can be directly used for fitting PH-distributed service demand. Here, we consider using MAPs that can reveal the underlying PH process with an explicit counting process.

MAPs [Neu92] are able to incorporate correlations between successive inter-arrival times. An n -state MAP consists of two stochastic processes, referring to a counting process and a phase process modeled by a finite state (n states) CTMC with infinitesimal generator $\mathbf{Q}$. Let $\mathbf{D}_0$ be a matrix associated with transitions without arrivals with non-negative off-diagonal elements; $\mathbf{D}_0$ and $\mathbf{D}_1$ satisfy $\mathbf{Q} = \mathbf{D}_0 + \mathbf{D}_1$ and $(\mathbf{D}_0 + \mathbf{D}_1)\mathbf{e} = \mathbf{0}$.

3.3 Problem formulation

As mentioned, we propose to observe the departure process of a microservice and determine parameters of service demand distribution, modeled as an APH, using maximum likelihood estimation.

To ensure the quality and efficiency of frequent interactions between different services, in microservice-based systems, connection pooling is applied to maintain the number of open connections for the REST-based interactions over HTTP. Hence, each microservice has a finite upper bound on the maximum number of concurrent users. Thus, a closed queueing model can capture the finite parallelism level of microservices.

Moreover, increasing populations of jobs under closed model may be used to approximate open models [ADW13]. Therefore, the proposed method is still applicable in principle to capture open workload-based systems. In this paper, we then abstract the microservice-based applications as closed queueing models as follows. Figure 3.1 illustrates the analytical model for this service, consisting of a closed queueing network describing both the microservice buffer and serving process, as well as the think time of clients issuing requests. The model features N concurrent users, each modeled as a job. Scheduling could be either First-come-first-served (FCFS) or Processor-sharing (PS) order, depending on the implementation details of the admission control system within the microservice. We assume exponentially distributed user think times at the delay station. The problem we study is to determine the APH service demand distribution in the queueing station. Note that since we focus on a single class of jobs, the queueing model admits a product-form solution for the steady-state distribution, while no specific product-form simplifications are available to stochastically analyze the departure process at the queue. As we show next, the service distribution identification problem requires instead a detailed Markovian analysis.

Departure process modeling with MAP. Referring to [ANN00], the inter-event times in queueing models can be captured with a quasi-birth-and-death (QBD) process as follows. The state space of a QBD process is the set of $\{(k, l) | 1 \leq k \leq n_l, l \geq 0\}$ can be divided into levels,

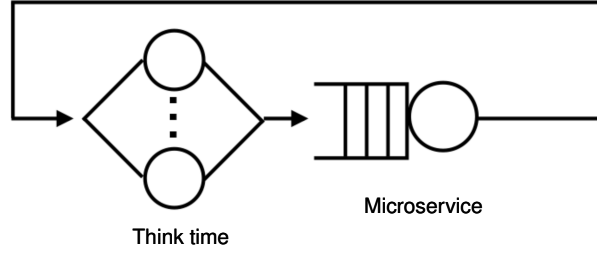


Figure 3.1: The reference closed queueing model for a microservice

where k is referred to as the phase and l is the level of the QBD. Transactions are only allowed between adjacent levels in QBD processes. Therefore, the generator matrix $\mathbf{Q}$ of such a QBD process is given by the quasi-birth-death process (QBD)

$$\mathbf{Q} = \begin{bmatrix} \mathbf{L}^0 & \mathbf{F}^0 & & \\ \mathbf{B}^1 & \mathbf{L}^1 & \mathbf{F}^1 & \\ & \mathbf{B}^2 & \mathbf{L}^2 & \mathbf{F}^2 \\ & & \ddots & \ddots \end{bmatrix}$$

where $\mathbf{L}^l$ represents the transition rates within level l , $\mathbf{F}^l$ is for the transitions from level l to level $l + 1$ and $\mathbf{B}^l$ describes the backward transactions from level l to level $l - 1$ when $l \geq 1$.

We can generate the infinitesimal generator $\mathbf{Q}$ of the underlying CTMC and then filter the events associated with job departures in matrix $\mathbf{D}_1$. That is, all and only the transition rates for departures from the queue are included in the non-negative matrix $\mathbf{D}_1$. Then, a MAP can be used to model the departure process with representation $(\mathbf{D}_0, \mathbf{D}_1)$, where $\mathbf{D}_0 = \mathbf{Q} - \mathbf{D}_1$.

We consider a MAP = $\{\mathbf{D}_0, \mathbf{D}_1\}$ that represents the departure process of the queueing station, our objective is to estimate the parameters for service demand distribution as $APH(\boldsymbol{\alpha}, \mathbf{T})$ from the observed inter-departure times (IDTs). We denote the time between two successive departure events i and $i - 1$ as $X_i = t_i - t_{i-1}$. Thus, the IDTs of jobs are the set $\mathbf{X} = [X_1, X_2, \dots, X_n]$. Since the departure process is modeled as a MAP, the IDTs follow a PH distribution $PH(\boldsymbol{\pi}, \mathbf{D}_0)$, where $\boldsymbol{\pi} = \boldsymbol{\pi}(-\mathbf{D}_0)^{-1}\mathbf{D}_1$ indicating the stationary distribution of the embedded chain. If $\mathbf{D}_0$ is acyclic, then the PH distribution specializes into an APH one. This

distribution produces an interval stationary initialization for the MAP.

For the MAP described above, the joint probability density function (PDF) of IDTs $\mathbf{X}$ is

$$f(\mathbf{X}) = \boldsymbol{\pi} e^{\mathbf{D}_0 \mathbf{X}_1} \mathbf{D}_1 e^{\mathbf{D}_0 \mathbf{X}_2} \mathbf{D}_1 \dots e^{\mathbf{D}_0 \mathbf{X}_n} \mathbf{D}_1 \mathbf{e} \quad (3.4)$$

For computational convenience, we assume that the given departure events are independent.

Thus, we have the approximation

$$\log f(\mathbf{X}) = \sum_{i=1}^n \log(\boldsymbol{\pi} e^{\mathbf{D}_0 \mathbf{X}_i} \mathbf{D}_1 \mathbf{e}) \quad (3.5)$$

In general, let $\mathbf{S}$ be the parameter set of the service demand distribution to be estimated. The log-likelihood for the IDTs is

$$\log f(\mathbf{S}|\mathbf{X}) = \sum_{i=1}^n \log(\boldsymbol{\pi} e^{\mathbf{D}_0(\mathbf{S}) \mathbf{X}_i} \mathbf{D}_1(\mathbf{S}) \mathbf{e}) \quad (3.6)$$

where $\mathbf{D}_0(\mathbf{S})$ and $\mathbf{D}_1(\mathbf{S})$ explicit the functional dependencies between $\mathbf{D}_0$, $\mathbf{D}_1$ and the service demand distribution parameters $\mathbf{S}$ we seek for, i.e., its moments m , c^2 , and ν . Then the service demand distribution estimation problem can be formulated as finding the parameters that maximize the log-likelihood of the IDTs measured from the monitoring traces.

Data preprocessing. For a real system, there could be several requests from the users arriving within a very short period of time. If we directly take all the samples in the trace, it could be quite time-consuming to calculate the likelihood function in Equation (3.6), especially due to the cost of evaluating the matrix exponential.

To address the issue, we observe that the inter-departure times of jobs can be grouped into different patterns. In order to accelerate the execution times, we apply clustering based on k-means [KM99] to partition the IDTs to obtain K groups of data with cluster centroids $\mathbf{Y} = [Y_1, Y_2, \dots, Y_K]$. Then, the log joint density in Equation (5.2) may be approximated based

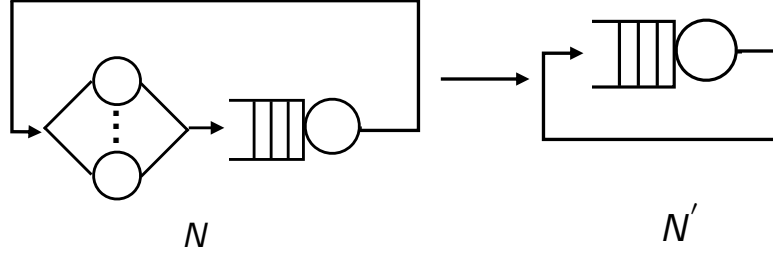


Figure 3.2: Modified model for PS with permanent customers perpetually looping at the queueing station

on the IDT clusters as

$$\log f(\mathbf{Y}) = \sum_{i=1}^K l_i \log(\boldsymbol{\pi} e^{\mathbf{D}_0 \mathbf{Y}_i} \mathbf{D}_1 \mathbf{e}) \quad (3.7)$$

where l_i denotes the number of points in cluster i .

We only briefly discuss the clustering of IDTs in this section and give an experimental example of the trade-off between computational complexity and effectiveness is shown in Section 3.6.2.

3.4 Service demand distribution estimation for single service class

The moment estimation of service demand distribution is based on measurements at runtime. Our measured observation consists of timestamps of arrival and departure for a sample set of successive requests, from which it is easy to derive quantities such as inter-departure times and response time at every resource of interest, we now discuss how we use this data to find a maximum likelihood estimation of the moments. Since an APH can then be recovered from the moments via moment matching, this provides a solution to the service demand distribution fitting problem.

3.4.1 State space reduction

As we increase the number of concurrent users, the state space of the CTMC can easily suffer state-space explosion. Assume a single-server queue where the service demand SD of the queue

node is APH distributed and a delay node as shown in the example in Figure 3.1. First, we consider the jobs in the queue are processed in a first-come, first-served (FCFS) order. Only one job can be served by the server at one time. Let P denote the number of phases in the service process, i.e., the number of columns in α . The number of states in the state space is

$$s = N \cdot P + 1 \quad (3.8)$$

where N is the number of users in the system.

Instead, if the server follows a processor sharing (PS) scheduling strategy, i.e., multiple jobs can be served simultaneously, and the total number of states is

$$s = \sum_{i=0}^N (N + 1 - i) \binom{i + P - 2}{i} \quad (3.9)$$

To mitigate the complexity of dealing with PS scheduling, we approximate the model as follows. We focus on the mean queue length that can be efficiently estimated using approximate mean value analysis (AMVA) [CSS02]. Based on solving the CTMC, the computational complexity is $O(N^3)$ even for FCFS scheduling. However, AMVA complexity is independent of N . The computational complexity is $O(I)$ for a single class case, where I is the number of iterations required for AMVA to converge, which is drastically fast to obtain the estimated number of permanent users for the queue in milliseconds.

The structure of the modified model is shown in Figure 3.2. Instead of considering all the individual jobs circulating in the delay and queue nodes as usual, we propose a modified model where the mean number of jobs residing at the queueing station in the original model is forced in the modified model to *permanently* reside at that station with N' *permanent users*, to mimic a similar average congestion level. We approximate N' with

$$N' = \frac{N - 1}{N} E[Q(N)] \quad (3.10)$$

where $E[Q(N)]$ is the expected number of jobs at the queueing station in the original model.

Note that the number of jobs N' looping in the new model may be seen as an approximation for the expected number of users at an arrival instant in the queue based on Schweitzer's Approximation [CSS02], which gives the $\frac{N-1}{N}$ term.

For a real system with a large number of users, this approximation can lead to a significant reduction of the computational cost, while providing accurate results.

3.4.2 Global optimization based estimation

Given the observed trace data, a common approach for parameter estimation is maximum likelihood estimation (MLE) [PF02], which casts the estimation as a global optimization problem. Thus, we propose an estimation method that combines MLE with simulations of queueing models to approximate the APH distribution of the service demand.

Algorithm 1 describes the implementation of this method in detail. The algorithm requires a set of clustered IDTs and searching boundaries as inputs. We first define a function to calculate the likelihood based on departure MAPs, as shown in Lines 1 to 16. We use moment matching [BHT05] to fit an APH distribution for the service demand, pointing to *APHFit* in Line 2. Note that we need to satisfy that the APH distribution is feasible with given parameters, i.e., both α and T are neither empty nor zero. After obtaining the service demand distribution SD , a queueing model is generated with a queueing station of SD . The current queueing model can be solved by analyzing the underlying CTMC, obtaining the infinitesimal generator Q . By analyzing the transitions in Q , the transition rates of departure events on the queue node can be filtered for D_1 in Line 5-7.

However, the computation of the infinitesimal generator involves the evaluation of a matrix exponential, which is computationally expensive and can incur numerical instability [MVL03]. To mitigate these issues, we use CTMC uniformization [RT88] which is an efficient numerical method for computing transient measures involving matrix exponential. For transient analysis, uniformization techniques can be applied with sub-generator D_0 and the initial distribution π of the MAP. Since the transient rate in D_0 of a real system could be large, to guarantee

Algorithm 1 Global optimization based estimation method

Input: $\mathbf{Y} \leftarrow$ Set of clustered inter-departure times $[Y_1, Y_2, \dots, Y_n, l_1, l_2, \dots, l_n]$, where Y_i is the centroid value and l_i is the number of points in cluster i
 $LB \leftarrow$ moment lower bound
 $UB \leftarrow$ moment upper bound

Output: $SD \leftarrow$ Estimated service demand distribution

- 1: **function** $Likelihood(\mu_1, \mu_2, \mu_3, \mathbf{Y})$
- 2: Fit service demand distribution $APH(\boldsymbol{\alpha}, \mathbf{T}) \leftarrow APHFit[\mu_1, \mu_2, \mu_3]$
- 3: **if** $APH(\boldsymbol{\alpha}, \mathbf{T})$ is feasible
- 4: Generate a queueing network model with service demand $APH(\boldsymbol{\alpha}, \mathbf{T})$
- 5: Solve the generated model and obtain the infinitesimal generator $\mathbf{Q}$
- 6: Filter $\mathbf{D}_1$ from $\mathbf{Q}$
- 7: $\text{MAP} \leftarrow \{\mathbf{Q} - \mathbf{D}_1, \mathbf{D}_1\}$, generate $\boldsymbol{\pi}$
- 8: **for** $i = 1$ to n
- 9: $\beta \leftarrow CTMCUniform(\boldsymbol{\pi}, \mathbf{Q} - \mathbf{D}_1, Y_i)$
- 10: $L \leftarrow L + \log(\beta \mathbf{D}_1 \mathbf{e}) l_i$
- 11: **end for**
- 12: **else**
- 13: $L \leftarrow -\infty$
- 14: **end if**
- 15: **return** L
- 16: **end function**
- 17: $[\mu_1^*, \mu_2^*, \mu_3^*] = \underset{\mu_1, \mu_2, \mu_3}{\operatorname{argmax}} Likelihood(\mu_1, \mu_2, \mu_3, \mathbf{Y})$
- 18: **return** $SD = APH(\boldsymbol{\alpha}, \mathbf{T}) \leftarrow APHFit[\mu_1^*, \mu_2^*, \mu_3^*]$

stable calculations, we adopt the scaling method from [SBM07], involving a scaling factor q to avoid floating-point errors. In Line 9, the scaling CTMC uniformization method is defined as $CTMCUniform$, which takes $\boldsymbol{\pi}$, $\mathbf{Q} - \mathbf{D}_1$ and the centroids of the cluster as the input. The approximated transient probability β can be obtained with $CTMCUniform$. Then the log-likelihood value can be computed using Equation (3.7) at Line 10. Therefore, we can obtain the optimal moments to fit the service demand distribution by maximizing the likelihood on the monitoring traces at Line 17.

3.4.3 Heuristics-based estimation

As illustrated in the last section, the global optimization for service demand distribution estimation needs to consider a large search space. It could be time-consuming to obtain the optimal parameter set maximizing the likelihood value. The method presented in this section

estimates the parameters sequentially, rather than jointly, offering a heuristic estimation that trades accuracy for speed.

Mean service demand estimation. The mean value of service demand can be efficiently estimated based on performance measurements from monitoring traces. We refer to the work in [PPSC13], which allows estimating the expected value of service demand with queue length and response times. Both queue length and response times are easily measured with system monitoring. As we target modeling the departure process in the queueing station, the input dataset of the estimation method contains the following data by calculating from system monitoring at departing occurrence.

- The response times from the monitoring traces (R)
- The queue length seen upon arrivals (A)

Considering a single class of jobs in the system, let N be the size of the population in the closed queueing network. It is known that the mean service demand $E[D]$ for the single-class case can be estimated as [PPSC13]:

$$E[D] = \frac{E[R]}{1 + E[A]} \quad (3.11)$$

where $E[R]$ and $E[A]$ are the expected values of response time and queue length seen upon arrival, respectively.

SCV estimation. To estimate the second moment of service demand, we investigate the state-dependent behavior of the system. The estimation formulation is derived from the SCV on the mean queue length seen upon arrival. In a queueing system, the response time of a job is related to the number of jobs in the queueing station either waiting or receiving service. We define R_i as the response time of the i^{th} request and A_i is the queue length seen upon the arrival of the request i . The response times can then be grouped into $\max(A)$ sets of data according to the value of A . Let R' be the grouped set of response times (e.g., $\mathbf{R}'_k = [R_i, \dots, R_j]$ where $R_i, \dots, R_j$ have the same queue length A_k with $0 \leq k \leq \max(A)$ and R'_{ki} is the i^{th} response time in R'_k), resulting in a representation with (R'_k, A_k) .

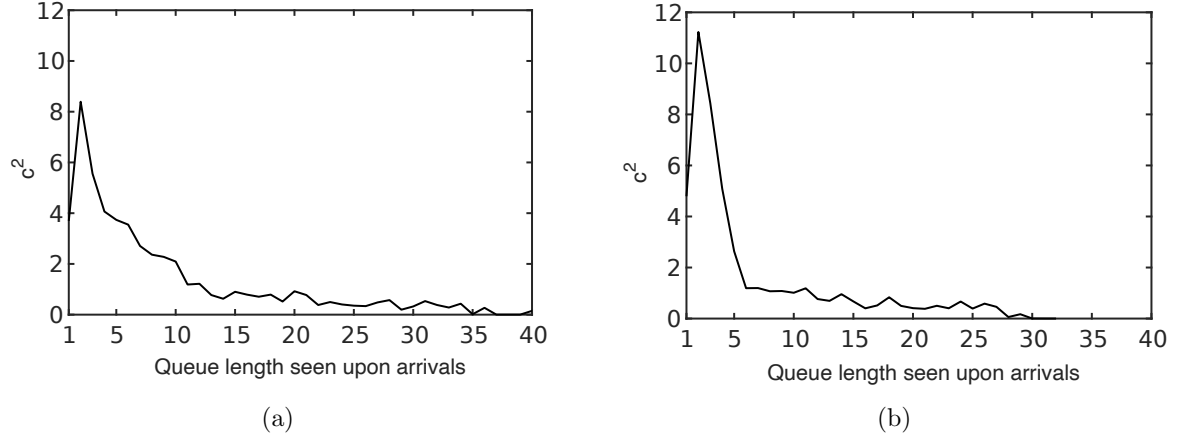


Figure 3.3: c^2 conditional on the queue length seen upon arrivals for simulation (a) and the real trace (b)

We propose to estimate c^2 using the following heuristic expression

$$c^2 = \max \frac{E[(R'_{ki} - E[R'_k])^2]}{E[R'_k]^2} \quad (3.12)$$

To demonstrate the accuracy of the approximation for SCV, we conduct an experiment with $N = 100$. We analyze real trace data from system monitoring and compare the c^2 of real traces to the simulated queueing model with estimated parameters based on MLE. We can see from Figure 3.3(a) that the c^2 conditional on the queue length first increases to the maximum and then decreases with the growth of queue length. The same pattern can also be observed for the real trace as shown in Figure 3.3(b). The simulated c^2 value is 8.3. It can be seen that the $\max(c^2)$ of the simulation is close to the one for the real trace with $c^2 = 11.2$.

Heuristic service demand distribution estimation. Our heuristic method to accelerate the MLE estimation is shown in Algorithm 2. Compared to the global optimization in Algorithm 1, the heuristic-based method is used to estimate the service demand distribution with a series of methods to fit the first three moments, including mean service demand (m), SCV (c^2) and the skewness (ν) estimation.

The algorithm requires a set of IDTs, the departure times, the response time of each job, and the queue lengths seen upon arrivals. In Line 1, we first compute the arrival times with the

Algorithm 2 Heuristics based estimation method

Input: $Y \leftarrow$ Set of clustered inter-departure times $R \leftarrow$ Set of response times $[R_1, R_2, \dots, R_n]$ $DT \leftarrow$ Set of departure times $[dt_1, dt_2, \dots, dt_n]$ $\nu = [\nu_1, \nu_2, \dots, \nu_j] \leftarrow$ searching set of skewness ν **Output:** $SD \leftarrow$ Estimated service demand with APH distribution1: Compute A from DT and R 2: $m \leftarrow \text{meanEstimate}(A, R)$ 3: $c^2 \leftarrow \text{scvEstimate}(A, R)$ 4: $\mu_1, \mu_2 \leftarrow m, c^2$ /*Compute the first two moments according to Equation (3.2)*/5: **for** $i = 1$ to j **do**6: $\mu_3 \leftarrow \nu_i$ /*Compute the third moment based on Equation (3.3)*/7: $L \leftarrow \text{Likelihood}(\mu_1, \mu_2, \mu_3, \mathbf{Y})$ /*Algorithm 1 at Line 2 to 16*/8: **end for**9: $\mu_3^* = \underset{\mu_3}{\text{argmax}} \text{Likelihood}(\mu_1, \mu_2, \mu_3, \mathbf{Y})$ 10: **return** $SD = APH(\boldsymbol{\alpha}, \mathbf{T}) \leftarrow APHFit[\mu_1, \mu_2, \mu_3^*]$

departure times DT and response times R and then calculate the queue length seen upon arrivals for each job. The mean service demand is estimated based on response times and the queue length seen upon arrivals. Then the algorithm estimates c^2 through Equation (3.12). The only search parameter for our method now is the skewness ν . Here we perform MLE on the departure process with Equation (3.6). As in the search-based global optimization, we first generate a set of candidate third moments parameters μ_3 with skewness values $\nu = [\nu_1, \nu_2, \dots]$. For each ν_i , we generate a queueing model with corresponding service demand and then calculate the likelihood value with function *Likelihood* in Algorithm 1 from Line 1 to 16. The algorithm will search on all candidates in the set, and the process is repeated with multiple candidate points for robustness. The estimated result of the skewness is finally decided by the one that maximizes the likelihood value. Therefore, the final result is obtained by three-parameter estimation methods.

3.5 Service demand distribution estimation for multiple service classes

For a microservice, different services can be identified into the classes based on different endpoints to model heterogeneous service processing. In this section, we generalize our service demand distribution estimation method to the multi-class service queues. Single-class service demands are often insufficient to capture the dynamic behaviors in the server, resulting in inaccurate predictions. For example, ignoring the presence of multiple classes can systematically underestimate or overestimate the service demand distribution for a queueing station.

Multi-class service demand distribution estimation can, however, be intractably expensive to solve with either the global search or the heuristics-based method mentioned in sections 3.4. First, the state space explosion problem could be further worsened by the use of different APH distributions for each class, with the state space roughly growing in size as the product of the corresponding state spaces for models with a single class. Moreover, the optimization problem can be challenging due to the increased number of search parameters, which are now to be differentiated across service classes.

To mitigate the execution complexity of searching for the optimal parameters, class aggregation [GCS86, BB19] can be applied to service demand distribution estimation. In particular, multi-class estimation problem can be systematically reduced to an equivalent two-class estimation problem. We also introduce an extension of our heuristics-based method for two classes where the model includes only a *tagged* class and an *aggregated* class, in a sense described later in this section. Finally, we evaluate the accuracy and scalability of the multi-class estimation.

3.5.1 Class aggregation

First, we illustrate the class aggregation process through the example shown in Figure 3.4. In this procedure, we tackle the estimation of service demand distribution of each class by solving R two-class submodels, with R being the total number of classes in the original model. For the

problem in Figure 3.4, we need to solve three submodels, so that the service demand distribution for a given job class (e.g., class A) is estimated based on a submodel with a tagged class (e.g., A) and an aggregated class that includes all other classes (e.g., B and C). By recursively applying this approach to all classes, considering each in turn as the tagged one, the problem is mapped to a collection of submodels, which can be solved independently of each other. In essence, this approach approximates the service demand distribution in a more scalable fashion than the original model by reducing the level of fidelity by which we represent the other classes. When solving each submodel, we only extract from that submodel the estimated service demand distribution of the tagged class.

The detailed description of submodel generation based on class aggregation is as follows. Referring to Figure 3.4, assume our objective is to estimate the service demand distribution for the tagged class A . In the aggregated class, the number of users is directly calculated with $N_{agg} = N_B + N_C$. Let denote p as the probability to be as class B ; $1 - p$ is the probability to be class C . The probability for the aggregated class is decided by the throughputs of each class, for example, $p = \frac{X_B}{X_B + X_C}$, where X_i represents the throughput of class i . The aggregated mean value m_{agg}^A for the submodel from tagged class A is calculated with the throughputs and the probabilities of the aggregated classes. Similarly, the aggregated SCV and the skewness can be obtained as a mixture distribution in the same way, where the original parameters of the service demand distribution for each class are the component of the mixture. With this definition, the moments have the following expression

$$E[M_{agg}^i] = \sum_{p \in \mathbf{P}, c \in \mathbf{C}} p E[M_c^i] \quad (3.13)$$

where M_c^i represents the i^{th} moment of the service demand distribution of class c and $\mathbf{C}$ is the set of service classes. $\mathbf{P}$ represents the probabilities of each class except the tagged class.

Multi-class MAP. Multi-class service demand distribution is an extension of the single-class case. Multi-class departure process MAP can also be represented with $MAP = \{\mathbf{D}_0, \mathbf{D}_1\}$, where $\mathbf{D}_1 = \sum_{i=1}^N D_1^c$. Let $\boldsymbol{\pi}_c$ be the stationary distribution that follows a departure of class c that satisfies $\boldsymbol{\pi}_c = \boldsymbol{\pi}(-\mathbf{D}_0)^{-1} \mathbf{D}_1^c$.

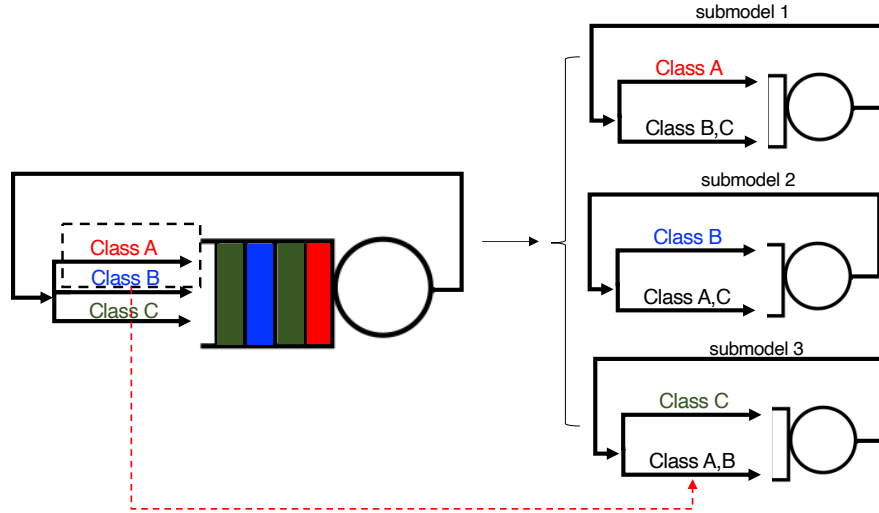


Figure 3.4: Example: submodel generation based on class aggregation with 3 classes

For each inter-departure time X_i extracted from the system monitoring traces, we denote c_i as the class label of the current departure job. The value of c_i can be repeated within the class set $\mathbf{C}$ of the queueing model, e.g., $\mathbf{C} = [c_1, c_2, c_2, c_3, \dots]$.

For multi-class MAPs, we assume that the IDTs and the class labels of the departure jobs are independent. Thus, the PDF of IDTs is

$$f(\mathbf{X}, \mathbf{C}) = \pi e^{D_0 X_1} D_1^{c_1} e^{D_0 X_2} D_1^{c_2} \dots e^{D_0 X_n} D_1^{c_n} e \quad (3.14)$$

Based on (3.14), the log joint PDF for multi-class can be approximated with

$$\log f(\mathbf{C}, \mathbf{X}) = \sum_{i=1}^K \log(\pi e^{D_0 X_i} D_1^{c_i} e) \quad (3.15)$$

Multi-class MEAN and SCV estimation. For multi-class service demand distribution estimation, we also aggregate classes for the third-moment estimation. The mean value of service demand is calculated referring to the response-time-based estimation extended to the multi-class scenario. Let

- response times of class c from the monitoring traces be represented as $\mathbf{R}_c$.

- queue length of class j seen upon arrival of class c be A_c^j .

For multi-class queue, the two scheduling policies we consider have different mean estimation expressions due to different behaviors of the server. For FCFS, we need to consider the different classes of jobs queueing in the server to wait for the service

$$E[D_c] = \frac{E[R^c] - \sum_{i \in C, i \neq c} E[D_i] A_c^i}{1 + A_c^c} \quad (3.16)$$

While for PS, the mean service demand can be estimated with Equation (3.17) which is directly extended from the single class.

$$E[D_c] = \frac{E[R^c]}{1 + \sum_{i \in C} A_c^i} \quad (3.17)$$

SCV estimation c^2 for class c can be generalized with multi-class jobs using the following equation

$$c_c^2 = \max \frac{E[(R_{ki}'^c - E[R_k'^c])^2]}{E[R_k'^c]^2} \quad (3.18)$$

For each class c , we group the response times using the same method as for the single class case, where $R_k'^c$ is the grouped set of response time with queue length A_k .

Iterative synchronization of the submodels. To synchronize the submodel for the tagged class with the estimated moments by the heuristic estimation method, we then iterate on the results of solving submodels. As mentioned before, the service demand distribution for R different classes is estimated considering a different tagged class each time. However, also the service demand distribution results for the aggregation classes can be used to calibrate the accuracy of the estimation iteratively, as shown in Figure 3.5.

We define an iterative method to solve all $|\mathbf{C}|$ two-class submodels with the $|\mathbf{C}|$ groups of estimated parameters with m, c^2 and ν for each class. Here, we denote ν_{agg}^c as the skewness value for the aggregated class in a submodel with a tagged class c . The parameters of each aggregated class (e.g., ν_{agg}^A) can be updated after obtaining the skewness values (e.g., calculate a new $\nu_{agg}^{A'}$ with ν_B and ν_C using Equation (3.13) from the current estimation iteration. Thus, the updated parameters of the aggregated classes in each iteration can be used to synchronize the

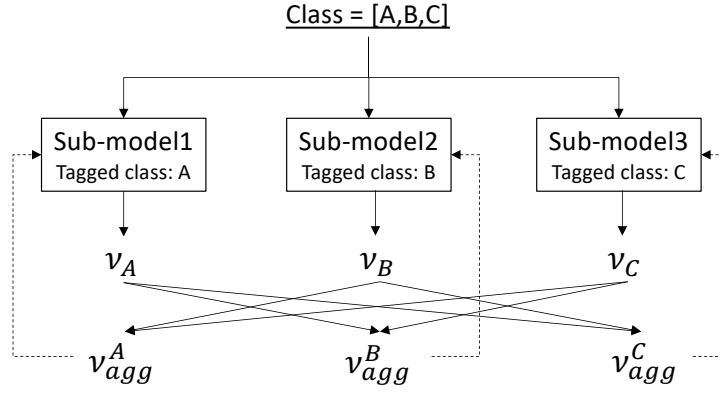


Figure 3.5: Synchronize submodels with iterations

submodel, which provides a calibration on the moments for aggregated classes. With iterative calibration of fitting the APH distribution of service demand in the queue node for aggregation class, the estimated skewness would be converged.

3.5.2 Multi-class heuristics-based estimation

Service demand distribution estimation for multi-class can be split into two sub-problems. The first one is to estimate skewness for each class with the tagged class and aggregated class. The second problem is to iteratively calibrate the accuracy of the estimated skewness.

Assuming that there are $|\mathbf{C}|$ job classes in total, we need to generate $|\mathbf{C}|$ submodels to solve this problem. In each submodel, there are two classes of jobs with N_{tag} and N_{agg} users. The heuristics-based method for multi-class is implemented as shown in Algorithm 3. This algorithm requires the following inputs. The throughput $\mathbf{X}$ for different classes (calculated from monitoring traces), the number of users N_c and think time Z_c in each class, and the clustered set of IDTs and class labels.

We define the function *Likelihood_multi* to calculate the likelihood on multi-class departure MAPs in Algorithm 3 referring to Line 3-25. In this procedure, the service demand distribution is first fitted conditionally on the synchronized iteration with the prior information of the skewness as shown from Line 7 to 11. Then a two-class submodel for estimation of current ν_c is generated. The submodel is then solved by analyzing the CTMC to obtain the

Algorithm 3 Heuristics based estimation method for multi-class

Input: $\mathbf{C} \leftarrow$ service class set, $\mathbf{m}, \mathbf{c}^2 \leftarrow$ mean and SCV set, $\mathbf{Z} \leftarrow$ user think times, $\mathbf{X} \leftarrow$ throughputs for different service classes, $\mathbf{Y} = (\mathbf{Y}_1 = [Y_{1_1}, \dots, Y_{1_n}, l_{1_1}, \dots, l_{1_n}], \mathbf{Y}_2 = [Y_{2_1}, \dots, Y_{2_n}, l_{2_1}, \dots, l_{2_n}]) \leftarrow$ Set of clustered inter-departure times for each service class, $\mathbf{N} \leftarrow$ Set of number of users in each class

- 1: $\mathbf{C}, \mu_3 \leftarrow CHECKScv(\mathbf{C}, \mathbf{m}, \mathbf{c}^2, \mathbf{Z})$ /*Defined in Algorithm 4*/
- 2: Initialize $iter = 1$
- 3: **function** Likelihood_multi($\mu_1, \mu_2, \mu_3, iter, \mathbf{Y}$)
- 4: $APH_1(\alpha, \mathbf{T}) \leftarrow APHFit[\mu_1, \mu_2, \mu_3]$
- 5: $m_{agg}, c_{agg}^2, \nu_{agg} \leftarrow m, c^2, \nu, \mathbf{X}$ /*From Equation (3.13)*/
- 6: Compute the first three moments for aggregated class $[\mu'_1, \mu'_2, \mu'_3]$ based on Equations (3.2) nad (3.3)
- 7: **if** $iter == 1$
- 8: $APH_2(\alpha', \mathbf{T}') \leftarrow APHFit[\mu'_1, \mu'_2]$
- 9: **else**
- 10: $APH_2(\alpha', \mathbf{T}') \leftarrow APHFit[\mu'_1, \mu'_2, \mu'_3]$
- 11: **end if**
- 12: **if** APH_1 and APH_2 are feasible
- 13: $submodel \leftarrow GENERATEModel(APH_1, APH_2, N_{tag}, N_{agg})$
- 14: $\mathbf{Q} \leftarrow solve(submodel)$
- 15: $MAP \leftarrow \{D_0, D_1^{tag} + D_1^{agg}\}$, generate π
- 16: **for** $k = 1$ to n
- 17: $\beta_1 \leftarrow CTMCUniform(\pi, D_0, Y_{1_k})$
- 18: $\beta_2 \leftarrow CTMCUniform(\pi, D_0, Y_{2_k})$
- 19: $L \leftarrow L + l_{1_k} \log(\beta_1 D_1^{tag} e) + l_{2_k} \log(\beta_2 D_1^{agg} e)$
- 20: **end for**
- 21: **else**
- 22: $L \leftarrow -\infty$
- 23: **end if**
- 24: **return** L
- 25: **end function**
- 26: **while** $tol \neq 0$ **do**
- 27: **for** i in $\mathbf{C}$ **do**
- 28: Relabel samples with tagged and aggregated classes
- 29: $\mathbf{Q}len \leftarrow reCalculate(\mathbf{Q}len)$
- 30: Update users with two classes $\mathbf{N}_{tag}, \mathbf{N}_{agg}$
- 31: $\mu_3 \leftarrow m, c^2, \nu$ with Equation (3.3)
- 32: $\mu_3^* = \underset{\mu_3}{\operatorname{argmax}} Likelihood_multi(\mu_1, \mu_2, \mu_3, iter, \mathbf{Y})$
- 33: **end for**
- 34: $[\mu'_1, \mu'_2, \mu'_3] \leftarrow momentAgg(\mathbf{X}, [\mu_1, \mu_2, \mu_3^*])$
- 35: $\nu'_{agg} = \nu_{agg}$
- 36: $\nu_{agg} \leftarrow \mu'_3$
- 37: $tol = diff(\nu_{agg}, \nu'_{agg})$
- 38: $iter = iter + 1$
- 39: **end while**

Algorithm 4 *CHECKScv*

Input: $\mathbf{C} \leftarrow$ service class set, $\mathbf{m}, \mathbf{c}^2 \leftarrow$ mean and SCV set, $\mathbf{Z} \leftarrow$ user think times

- 1: $\mathbf{m}_{agg} \leftarrow \text{meanAgg}(\mathbf{Z}, \mathbf{m})$ /*According to Equation (3.13)*/
- 2: $\mathbf{c}_{agg}^2 \leftarrow \text{scvAgg}(\mathbf{Z}, \mathbf{c}^2)$ /*Using Equation (3.13) and Equation (3.2)*/
- 3: **for** $i = 1$ in $|\mathbf{C}|$ **do**
- 4: **if** $c_i^2 \leq 1$ **then**
- 5: **if** $c_i^2 == 1$ **then**
- 6: $\nu_i = 2$ /*the skewness of Exponential distribution*/
- 7: **end if**
- 8: Calculate ν_i of $\text{Erlang}(m_i, c_i^2)$
- 9: $\mathbf{C} \leftarrow$ remove i from $\mathbf{C}$
- 10: Compute μ_{3i} with (m_i, c_i^2, ν_i) using Equation (3.3)
- 11: **end if**
- 12: **end for**
- 13: **return** $\mathbf{C}, \mu_3$

infinitesimal generator Q . The two-class departure MAP now can be constructed as the process $(D_0, D_1^{tag}, D_1^{agg})$, which are generated by filtering different classes on Q . In Line 12 to 15, the result of log-likelihood for the tagged class with relabeled inter-departure times is obtained.

Algorithm 3 starts with calculating the number of classes and the set of third moments according to Algorithm 4. Then, it iterates on each service class, updating the number of uses in the tagged and aggregated class and relabeling the samples with two classes, referring to Line 28-30. The algorithm calculates the third moment of the current tagged service class and obtains the likelihood value via *Likelihood_multi*. Then the first three moments of the current tagged class can be then obtained with the third moment candidate that maximized the likelihood. The parameters of the skewness of the remaining classes are solved with the same procedure by solving with submodels as illustrated.

After solving all submodels based on the MLE method, the skewness of each class can be obtained. The submodels can be synchronized with updated aggregated skewness values. The aggregated skewness for each submodel is calibrated with moment aggregation based on the throughputs. At the end of each iteration, the algorithm compares the current skewness for the aggregated classes in each submodel with the updated values.

To accelerate the convergence of the algorithm, we add additional conditions to estimate the skewness as shown in Figure 3.6 and Algorithm 4. For each service class, there are three cases

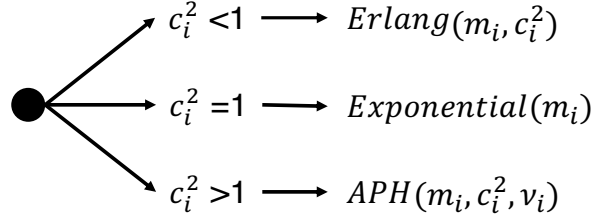


Figure 3.6: Conditional skewness estimation

to fit for the APH distributed service demand in the queue node. First, if $c_i^2 < 1$, it can be then fitted from an Erlang distribution for the service demand with m_i, c_i^2 . Thus, there is no estimation of the skewness for class i since it can be directly calculated based on the Erlang distribution. In this case, Algorithm 3 only involves the rest of class in $\mathbf{C}$ except class i according to Algorithm 4. Next, if $c_i^2 = 1$, the service demand distribution of this class should form an Exponential distribution with the skewness equal to 2.

3.6 Single-class Experiments

This section introduces the experimental setup and evaluation metrics, together with a comparison of our method against the baseline algorithm.

3.6.1 Experimental setup

To evaluate the proposed service demand distribution estimation, we conduct several experiments using an open-source microservice-based application called Sock Shop¹. Sock Shop consists of 13 different services, and all services communicate using REST APIs over HTTP. We use Docker Compose for the multi-container orchestration to deploy the microservice-based system. This benchmark has been well studied in recent papers [RL19, AMPJ17] focusing on microservice performance, since it emulates a real-world ecommerce website by using microservices and cloud-native technologies. Additionally, the Sock Shop application provides the flexibility to set up experiments for deploying partial or entire microservices to monitor n-tier

¹<https://microservices-demo.github.io/>

Table 3.1: Workload mixes

<i>CPU level</i>	<i>Low</i>	<i>Medium</i>	<i>High</i>
CPU utilization (U)	33%	43%	95%
Number of users (N)	50	100	300

services as desired [KSK19]. To meet our performance modeling abstraction, in the experiment, we target a service that does not interact with other databases, which avoids indirect drifts in the response time due to the state of the database.

To evaluate our proposed service demand distribution estimation methodology with coarse-grained monitoring data, we then conduct load tests to generate closed workloads with different numbers of users by using Locust². Locust is an open-source load testing tool that allows the simulation of microservice users by defining tasks with Python and injecting an artificial workload into the system.

The details of the experimental environment are described as follows. For the deployment of the application, we use a server running Ubuntu 16.04.7, and our target service is pinned to a separate CPU core. Locust is running on 6 different client machines to simulate the concurrent users for the microservice. We experiment with different populations of users to assess the corresponding CPU utilization level as shown in Table 3.1. In the following experiments, we consider 3-level of CPU utilization to simulate the system under low (33%), medium (43%), and high utilization (95%) to assess the performance of our proposed method under varying intensities of system load.

Other performance model parameters are set as follows. In all the experiments, the users' think time is set with an exponential distribution with a mean value of 0.1 seconds. The number of servers for the queueing station is fixed as 1 to represent a single server in a microservice.

First, we run numerical experiments to compare the estimation results to known service demand distribution parameters. In numerical experiments, the trace data is collected by generating of closed queueing models and solve the models with simulation. Numerical experiments aim to establish a ground truth by using known parameters of the service demand distribution to

²<https://locust.io/>

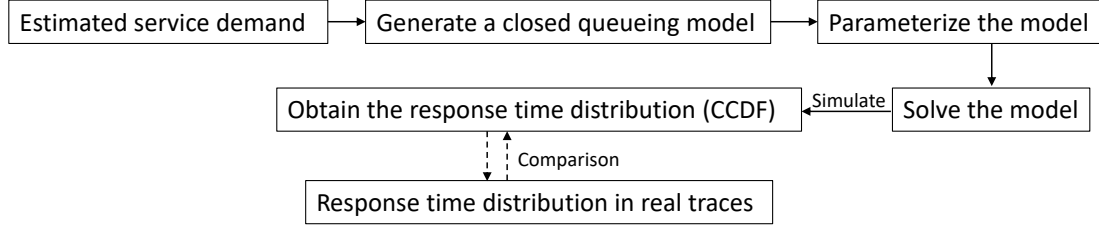


Figure 3.7: The process of experimental comparison

evaluate the accuracy of the proposed estimation method. In the single-class experiments, we fix the mean value of the service demand at a value of 0.7. Then, we vary the value of the squared coefficient of variance (SCV) to assess the performance of the estimated distribution: 1) $c^2 = 0.5$ for a concentrated distribution around the mean, 2) $c^2 = 1$ for a balanced spread of values in the distribution around the mean value, 3) $c^2 = 4$ for a wider spread of values of the distribution with respect to the mean, 4) $c^2 = 10$ for a higher level of variability exhibiting greater deviation from the mean.

Next, to evaluate on real microservice traces, we conduct experiments with the workload mixes in Table 3.1 with Sock Shop and capture the network traffic with a dockerized `tcpdump` that is triggered over HTTP. During the experiments, we monitor HTTP traffic on the source and destination nodes of our target service. To avoid the impact on the cold start of the microservice, after generating the concurrent users from Locust, we monitor the setup microservice for 5 minutes to warm up without collecting any data. Then after the cold-start phase, the collected traffic data is parsed to extract the request and response information of each request. For each workload pattern, we collect 10,000 successive HTTP requests and response pairs to build up the trace dataset. Each sample in the trace dataset consists of the timestamps of the request arrival and departure.

Baseline algorithm. The service demand distribution is usually modeled as exponential fitting the estimated mean value [SCBK15]. In our experiments, we also fit as a baseline with an exponential distribution for service demand.

Metrics. Since the real service times of systems are usually difficult to measure, to evaluate the performance of the proposed estimation method on real trace data, we opt to compare the

measurable response time from the end-to-end monitoring of the microservice to the predicted values from performance models. We use the complementary cumulative distribution function (CCDF) to compare the response time distribution modeling accuracy of the model prediction to the real trace following the methodology as shown in Figure 3.7. After estimating this service demand distribution, we construct the closed queueing models. Then we use the simulation-based solver JMT in LINE [PC17] to solve the generated models, and obtain the corresponding response times CCDF from the parameterized model and baseline for comparison with the measure CCDF. The only difference between the baseline and our experiments is the service demand distribution, since the service demand distribution of the baseline is estimated by an exponential distribution with the mean value m .

3.6.2 Data preprocessing and clustering

To demonstrate the trade-off between computational cost and approximation accuracy depending on chosen number of clusters K , we conduct the experiment with the microservice trace when $N = 50$. As mentioned in Section 3.6.1 each trace dataset consists of 10000 samples, in this experiment, we refer the original data without clustering to $K = 10000$. Then we estimate the service demand distribution with both the original data ($K = 1000$) and clustered data for different values of $K = [10, 50, 100, 500, 1000]$. Figure 3.8 presents the accuracy and execution time comparison for different K . It can be observed from Figure 3.8(b) that with clustered IDTs the execution time drops by almost 33%, comparing the results between the original data ($K = 10000$) and $K = 1000$. The accuracy of estimation with the clustered data is evaluated in Figure 3.8(a) by CCDF diagrams of the response time distributions. While small values of K lead to a coarse approximation, by increasing K , the CCDF for the clustered data rapidly converges to the curve representing the results with the original data (non-clustered). As can be noted from Figure 3.8(a), the curves become indistinguishable for $K \geq 100$. We can thus conclude that our clustering heuristics does not significantly reduce the accuracy of the estimation when $K \geq 100$, while significantly reducing the computational cost of the estimation.

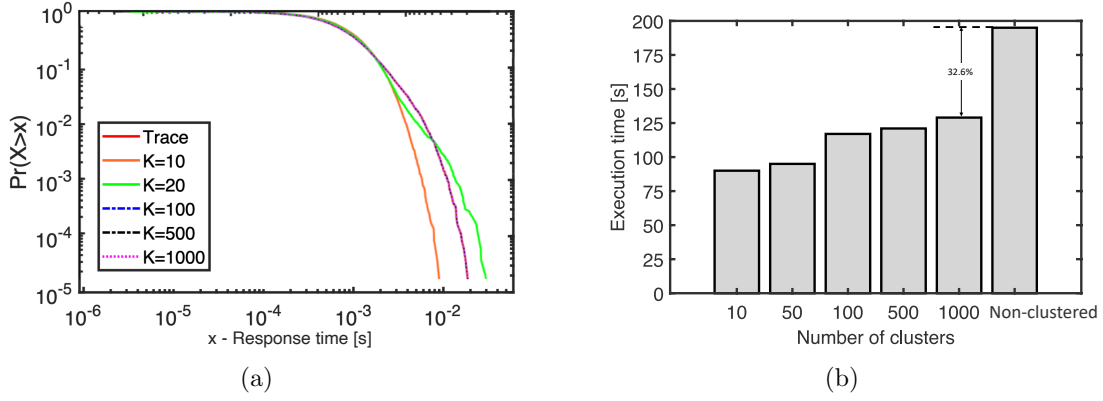


Figure 3.8: The CCDF of response times (a) and the execution times (b) of maximum likelihood estimation under different number of clusters

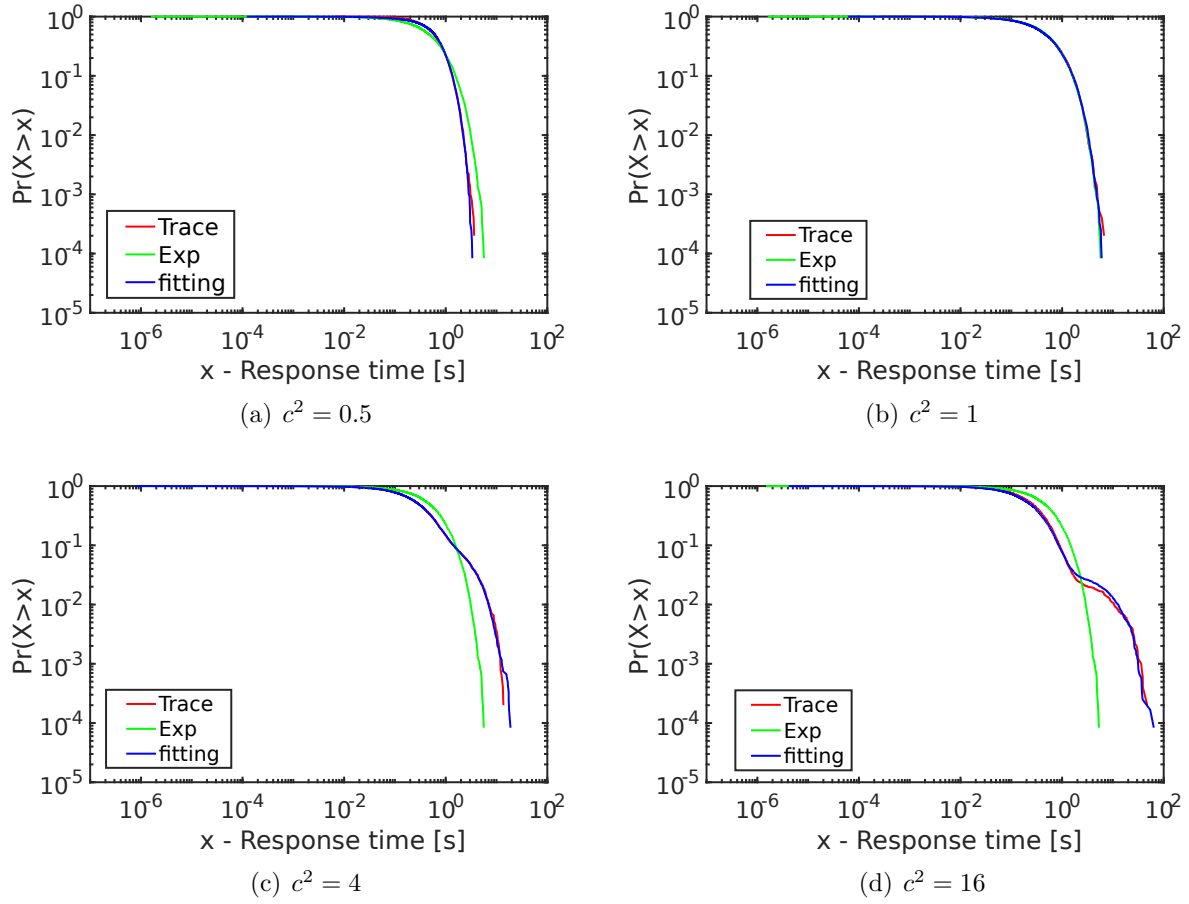
3.6.3 Numerical experiment results

We conduct the following numerical experiments to assess our global-search-based method for the service demand distribution estimation, aiming to obtain a ground truth with pre-setting parameters.

We generate closed queueing networks with a single server queue that can simulate the behaviors of a simplistic microservice, which is denoted as *generated model* in the following description. In the generated model, we give the moments of service demand distribution at the queueing node with different values, so that we can have a ground truth of the moments of service demand distributions. Then we collect samples of timestamps of arrival and departure as the trace data by solving the generated models. In this way, we can compare the estimated results from global optimization to these values setting in the generated model.

The details of the numerical experimental design are as follows. For the setting of pre-known service demand distribution, we fix the mean service demand m with a random value 0.7 and then set different SCV with $c^2 \in \{0.5, 1, 4, 16\}$. Therefore, there are 4 generated queueing models with different service demand distributions. Then we execute Algorithm 1 with these 4 traces data and compare the estimated SD to the known values and the baseline algorithm. The results of the comparison are plotted as CCDF diagrams in Figure 3.9.

It can be seen from the response times distributions that our estimation method achieves good

Figure 3.9: CCDF of response time for different settings of c^2

fits for all settings of c^2 and outperforms the exponential distribution, especially for the tail of the distributions. For larger c^2 , such as $c^2 = 16$, we can observe that estimating the mean m of the exponential distribution alone cannot capture the full distribution of service demand, with the baseline algorithm decreasing much faster than our method (e.g., the green curve in Figure 3.9), resulting in an inaccurate prediction of the response times.

Next, we turn to evaluate the estimation accuracy under different numbers of users. In this part of the experiment, we still set the service demand distribution with known values of the generated model, and vary the number of users in $N \in \{1, 5, 10\}$. The assigned parameters of service demand in the generated models are $m = 0.7$ and $c^2 = 16$, which represent a higher variability than in the first numerical experiment. During solving generated models, we notice that with $N = 10$ the CPU utilization can reach over 95% which is close to saturation. As shown in Figure 3.10, all the green curves with baseline algorithm fail to model the response

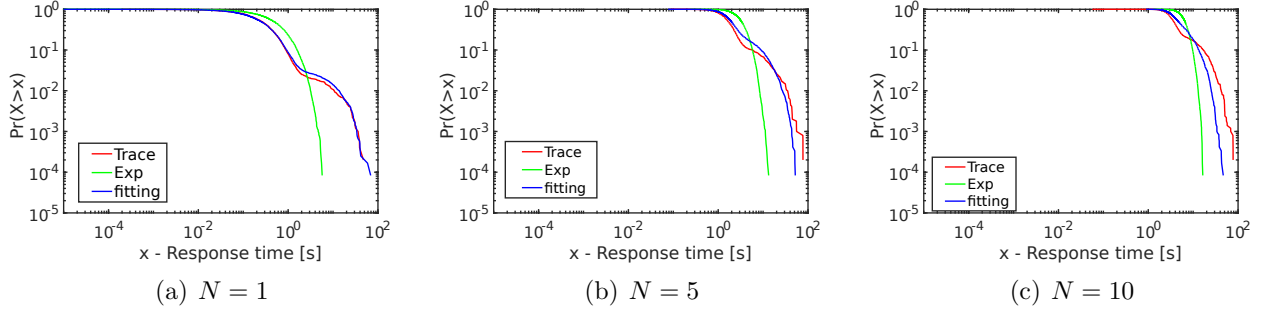


Figure 3.10: CCDF of response time for different numbers of users N

time distribution accurately under different workloads. While, the fitting results of our method achieve very close fit to the trace data under light load N . For $N = 5$ and $N = 10$, the results of the proposed method still yield good performance for both body and tail distribution compared to the baseline.

3.6.4 Analysis of results on measured traces

Analysis with FCFS. We now turn our attention to experiments of the real trace on the microservice application. First, we present our experimental results with FCFS scheduling policy under different workload patterns.

We notice that for a single server queueing system under FCFS scheduling, the service times are measurable by sampling IDTs when the server is not idle, which provides an explicit way to evaluate our proposed estimation method. Therefore, for FCFS scheduling, we add another comparison by directly calculating the first three moments from these sampled service times, which is denoted the *FCFS-single* method. We evaluate the model prediction results with FCFS-single method, baseline algorithm, and our method to the real trace data. The CCDF of response times for low, medium, and high CPU utilization are plotted in Figure 3.11, respectively. The estimated parameters for service demand distribution with FCFS are summarized in Table 3.2, and the parameters based on FCFS-single method are in Table 3.3.

First, compared to the baseline algorithm with exponential distribution, it can be seen that for all 3 different N , our method produces a closer fit. As one can see for the body fitting, both

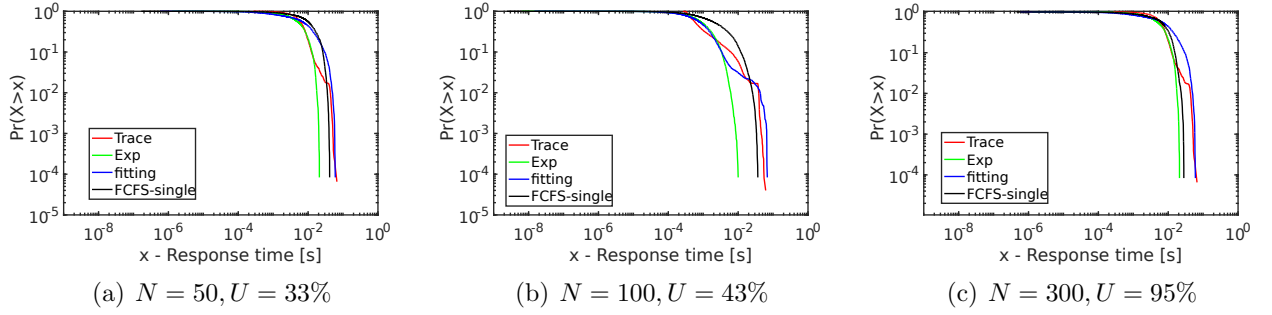


Figure 3.11: CCDF of response time with different utilization for the fitted models with FCFS.

Table 3.2: Service demand distribution parameter estimation results for FCFS

N	$m \times (10^{-4})$	c^2	ν	$Likelihood \times (10^4)$
50	8.33	8.24	11	6.62
100	6.21	11.22	57	14.31
300	3.50	8.03	12	10.64

Table 3.3: Service demand distribution parameter estimation results for FCFS with FCFS-single method

N	$m \times (10^{-4})$	c^2	ν
50	13.61	2.31	4
100	7.96	2.85	5
300	3.72	2.87	7

baseline and our proposed model yield good performance. For $N = 50$ and $N = 100$, our model fits the body with better accuracy, whereas the baseline method is outperformed for $N = 300$. However, for the fitting of the tail, the baseline method is worse in terms of accuracy for all values of N .

Then we compare the fitting results of our heuristic method to the direct measurement - FCFS-single method. We observe in Figure 3.11 that our estimation results yield a similar accuracy of fitting the tail of the distribution under low and medium utilization levels as the FCFS-single method. Both the blue and black curves are much closer to the one of the real trace data. At the same time, our method achieves more accurate fitting of the body when $N = 100$. For higher CPU utilization, the results from our method show slightly weaker fitting in the middle of the body, but achieve a better fit for the tail of the distribution. We interpret that the real microservice system is not precisely served by FCFS scheduling, while it exhibits some state-dependent degree of CPU sharing. The proposed heuristic method captures instances of large

service variance that occur in certain system states, which can be inferred from c^2 in Table 3.3.

Overall, under FCFS scheduling policy, our estimated distribution achieves a better fit throughout the distribution curve, producing an approximation of the real trace distribution accurately. Compared to the baseline and FCFS-single method, our proposed method can model the tail behaviors, which is significant to model different intensive workloads.

Analysis with PS. To analyze the impact of scheduling policies, we conduct experiments with $N \in \{50, 100, 300\}$ and PS scheduling policy. The parameter estimation results with PS scheduling are shown in Table 3.4 and the fitting comparison is plotted in Figure 3.12.

First, compared to the results with FCFS scheduling, switching to PS scheduling impacts the parameter estimation results of the baseline in a significant manner. Comparing the estimated skewness to the results for FCFS in Table 3.2, we can observe smaller values of skewness in Table 3.4. In comparison, the likelihood values for different workloads with PS are still close to the one with FCFS.

For the fitting comparison on CCDF diagrams, under light and medium load, it can be seen in Figure 3.12(a) and 3.12(b), the baseline method first fits well at the beginning, and it decays faster from the middle body, ultimately not able to capture the tail for light and medium load. Our method also fits the body with low inaccuracy and outperforms the baseline for the tail fitting, showing a more accurate fit for the distribution. However, by increasing the number of users to $N = 300$, which is close to 100% CPU utilization, the baseline method achieves a closer fit to the data with PS scheduling in Figure 3.12(c).

The above experimental results with FCFS and PS scheduling indicate that in reality, the actual system scheduling will factor in several elements such as caching, memory bandwidth, and operating system scheduling, which are neither perfectly PS nor FCFS. Therefore, our results indicate that either model provides a reasonable approximation to the observed system behavior, but FCFS appears more suitable to model heavy loads.

To summarize, the previous results indicate that in the majority of instances, the proposed method is able to estimate the service demand distribution with higher fidelity than simple

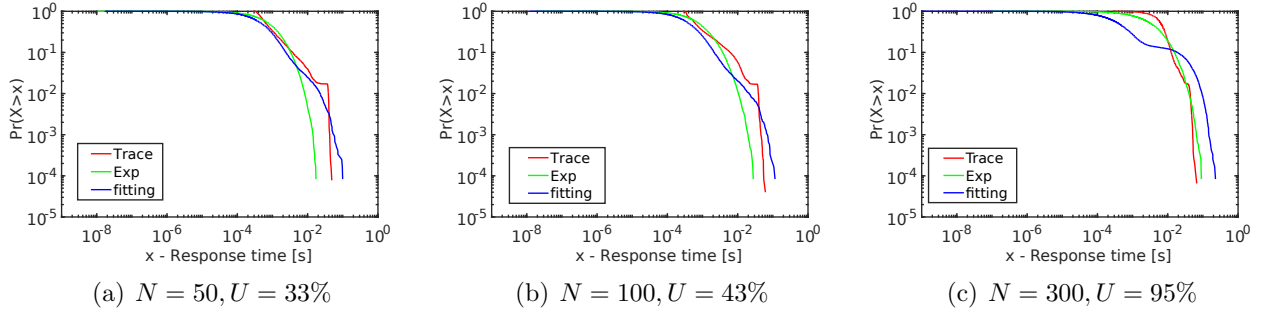


Figure 3.12: CCDF of response time with different utilization for the fitted models with PS.

Table 3.4: Service demand distribution parameter estimation results for PS

N	$m \times (10^{-4})$	c^2	ν	$Likelihood \times (10^4)$
50	8.33	8.24	9	6.11
100	6.21	11.22	10	14.53
300	3.50	8.03	4	10.68

exponential service demand. By comparing the CCDF of the response time distribution, the proposed heuristic method can also capture the tail behaviors with higher accuracy.

3.7 Multi-class Experiments

3.7.1 Numerical experiment results

To evaluate the effectiveness of the proposed heuristic-based estimation for multi-class jobs, numerical experiments are conducted based on traces collected via simulation.

In the following experiments, we first generate closed queueing models with 3 service classes of service and samples from the simulation results to construct the datasets for evaluation. To avoid the difference in service distributions being driven by their mean value, we set the same mean value for the service demand distribution for different classes in the queue node to $m = 5$ and c^2 is selected as 1, 4, 8 for each class. In this way, we can have the ground truth for comparison to the result of our proposed method. The details of the parameter settings in the sampling models are shown in Table 3.5 and Table 3.6. In the experiment of Table 3.5, we set the mean and SCV values for the generated model. Thus, ν of 3 classes can be

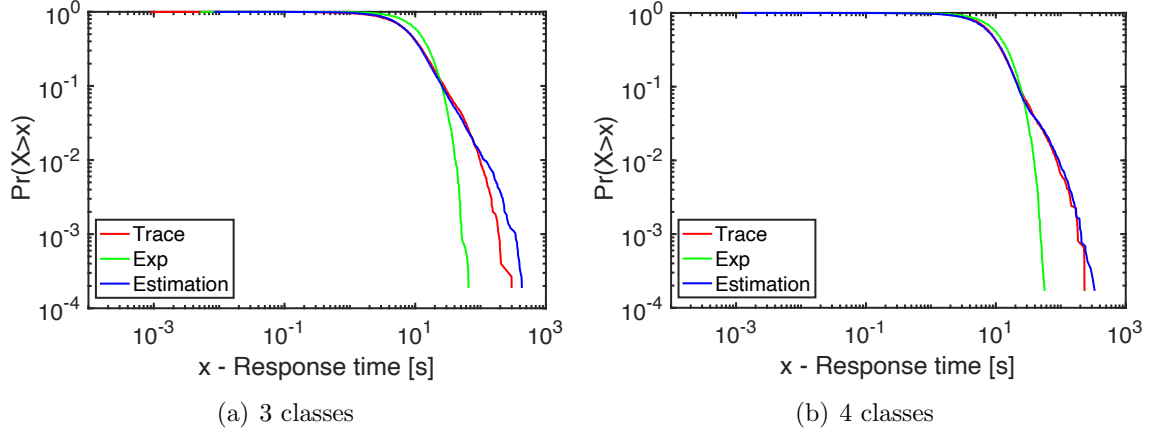


Figure 3.13: CCDF of response times of multi-class for numerical experiments

Table 3.5: Parameter estimation results for numerical experiments with 3 classes

	m	<i>Estimated</i> m	c^2	<i>Estimated</i> c^2	ν	<i>Estimated</i> ν
Class1	5	5.15	1	1.62	2.00	50.0
Class2	5	5.13	4	4.36	5.31	5.4
Class3	5	5.00	8	18.96	7.98	12.8

compared to the baseline (Column ν) by calculating the skewness of the fitted APH with m and c^2 . For Table 3.6, we also set the skewness values in $\nu = [4, 7, 10]$ of the generated model. Therefore, in the second experiment, the accuracy of the skewness estimation can be directly measured by comparing the setting skewness and the experimental results. The CCDF of the above experiments are shown in Figure 3.15(a) and 3.15(b).

First, compared to the baseline algorithm with exponential distribution, our estimation results yield better performance from the body to the tail. The green curve of the exponential distribution decays faster from the middle, resulting in an inaccurate prediction of the tail of the response times. We can then observe from the parameter estimation results in Table 3.5 that the estimated values of m and c^2 are close to the given parameters of the generated model. If the optimal result reaches a bound value, the algorithm will then fit from the first two moments instead of three moments to reduce the inaccuracy. The results in Table 3.6 show that the skewness estimation for all 3 classes is accurate.

Next, we evaluate the effectiveness of iterative calibration with different submodels. We compare the CCDFs with and without iterative calibration (e.g., with a single execution) in Figure

Table 3.6: Parameter estimation results for numerical experiments with 4 classes

	m	<i>Estimated m</i>	c^2	<i>Estimated c^2</i>	ν	<i>Estimated ν</i>
Class1	5	5.06	1	1.55	4	8.1
Class2	5	4.40	4	4.14	7	8.4
Class3	5	4.89	8	10.77	10	12.3

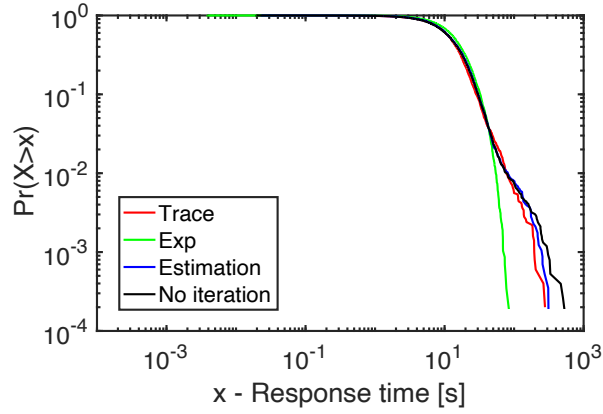


Figure 3.14: Comparison results for iterative calibration

3.14. The black curve represents the CCDF without iteration. It can be seen that for the body of the distribution, there is no obvious difference between our estimation and the one without calibration. However, our estimation method captures the tail of the distribution quite well compared to the simulated trace data. Instead, the method without iteration deviates from the tail. Therefore, we conclude that, for the multi-class case, the calibration with iteration on the skewness can help to improve the accuracy of the service demand distribution estimation for multi-class cases.

3.7.2 Analysis of results on measured traces

In the following experiments, we evaluate our estimation method with real traces from the same microservice application (Sock Shop) as before. We concluded that the scheduling policy under high load is intended to be FCFS, while more likely to be PS with lower load. In this section we have run experiments with two parameterizations: $N = \{10, 10, 10\}$ and $N = \{10, 10, 10, 10\}$ for 3 and 4 service classes, which are under medium load, configuring the worst case for estimation, since neither FCFS nor PS will constitute a perfect fit.

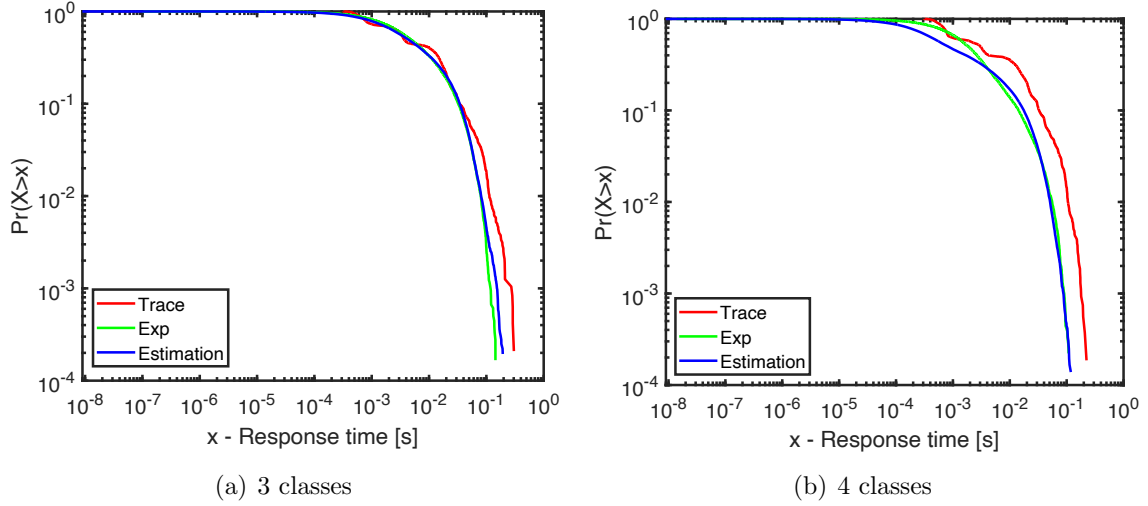


Figure 3.15: CCDF of response times for multi-class with FCFS

Table 3.7: Parameter estimation results with FCFS

3 classes				4 classes			
	$m \times 10^{-3}$	c^2	ν		$m \times 10^{-3}$	c^2	ν
Class 1	1.8	11.1	16.1	Class1	1.4	10.2	14.3
Class 2	4.4	2.4	14.9	Class2	3.8	2.5	12.9
Class 3	17.5	0.7	1.6	Class3	16.1	0.7	1.6
				Class4	1.4	10.0	15.1

Analysis with FCFS. Comparative results for $N = 30$ for 3 classes and $N = 40$ with 4 classes are shown in Figure 3.15(a) and Figure 3.15(b) respectively. Table 3.7 shows the corresponding parameter estimation results for the service demand distribution. Note that c^2 of class 3 is less than one, and the service demand distribution for class 3 is estimated with an Erlang distribution as illustrated in Section 3.5. Thus, in these two tables, ν of class 3 is obtained by calculating the skewness of the Erlang distribution.

For three classes, both the baseline algorithm and our estimation are able to fit the response time distribution of the real trace, while our estimation is better on the tail. It can be seen from Figure 3.15(b) that our estimation achieves almost the same fitting accuracy as the exponential one. Compared to the baseline algorithm with FCFS for both three and four classes, the heuristics-based method performs similarly to the exponential distribution, however, the fitting from our estimation is still a fair approximation of the real trace behavior.

Analysis with PS. Table 3.8 and Figure 3.16 show the estimation results with PS. For fitting

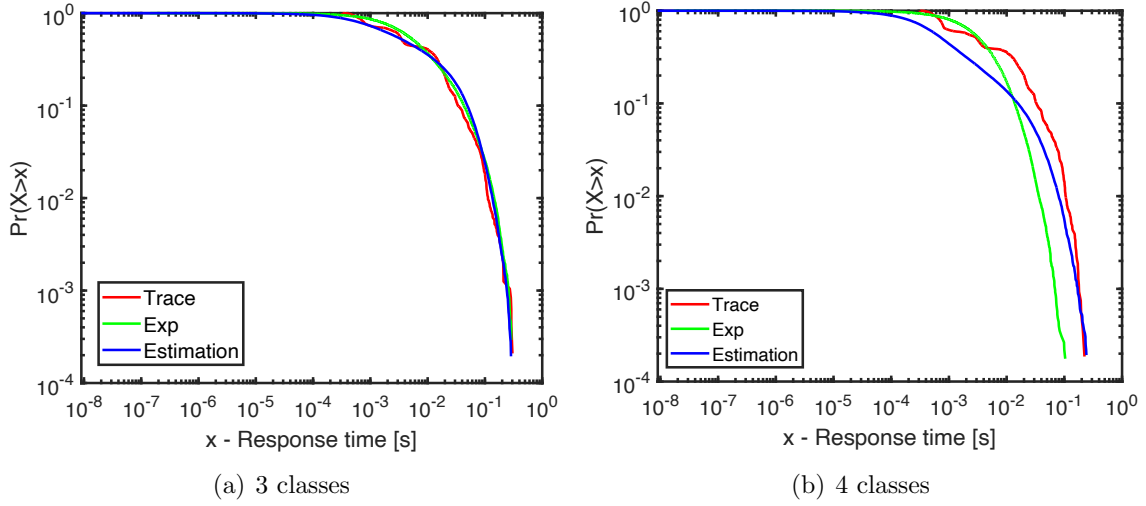


Figure 3.16: CCDF of response times for multi-class with PS

Table 3.8: Parameter estimation results with PS

3 classes				4 classes			
	$m \times 10^{-3}$	c^2	ν		$m \times 10^{-3}$	c^2	ν
Class 1	2.8	11.1	5.1	Class1	3.2	10.2	5.5
Class 2	5.7	2.4	3.9	Class2	5.7	2.5	3.5
Class 3	26.1	0.7	1.6	Class3	22.8	0.7	1.6
				Class4	3.0	10.0	5.4

the service demand distribution of three service classes in Figure 3.16(a), both our estimation result (the blue curve) and the exponential one (green) are close to the real trace, indicating a good characterization of the service demand of the real system. Figure 3.16(b) shows how our estimation method outperforms the baseline algorithm for the four classes case. Moreover, comparing the fitting on the tail of the distribution, our result is closer to the real trace. Instead, with the exponentially distributed service demand, the predictive curve of the response times decays faster, especially for the four classes trace. Thus, the heuristics-based method has a better fit throughout the response time distribution.

In summary, for three and four service classes, the estimation results with both FCFS and PS achieve reasonable accuracy in capturing the real system. We can draw the same conclusion as the single-class experiments that the tail behaviors based on our estimation achieve good accuracy to capture the response times distribution, especially for high percentiles, while also being more efficient to compute.

3.8 Threats to Validity

In this section, we outline potential threats to the validity of our findings, aiming to identify and address factors that may introduce bias or limitations in our study.

- **Model abstraction and experimental simplification:** The assumption of independence among microservices oversimplifies real-world scenarios, potentially leading to inaccuracies in the model. In our experiments, we use the independent microservice in Sock Shop as the benchmark for real trace results comparison. However, we can consider to utilize other monitoring tools to extract end-to-end data for each microservice first and then capture inter-service relationships, improving the accuracy of the model. Additionally, consider incorporating stochastic queueing models to better represent the complexity of microservice-based applications.
- **Selection bias in experiment parameters:** There may be a selection bias in setting experiment parameters, potentially limiting the generalizability of the results. We conduct experiments with fixed different values of mean and scv to verify the accuracy of our proposed method in the numerical experiments. A possible way to mitigate the threats is to test a wider range of random values for the mean and scv to ensure comprehensive results. Exploring different configurations of the queueing model, such as the think times of users and the number of concurrent users, is also helpful to assess their impact on the accuracy of the estimation method and mitigate potential biases. However, these queueing model parameters should not impact the accuracy of the estimation results.
- **System workload mixes:** In our experiments, we consider the system with low, medium, and high CPU utilization levels that may not fully capture the behavior of the microservices, especially under extreme workloads. It is interesting to include experiments with near 100% CPU utilization to explore potential behaviors of the microservices in responding to requests under extreme workload conditions.
- **Baseline distribution comparison:** Our experiments mainly focus on the comparison with the exponential distribution that is the most common method for service demand esti-

mation in recent papers [BLP10, KML15]. However, this may overlook other potential distributions that could better fit the service demand in comparison. In the future, it may be valuable to explore and test other distributions by providing mean and SCV values based on the estimation algorithm to ensure a comprehensive comparison and selection of the most suitable distribution for the model.

3.9 Summary

In this chapter, we introduced a service demand distribution estimation method by using Markovian arrival processes to improve the prediction accuracy. The following points highlight the distinctive contributions:

- We presented a closed queueing model for a microservice and characterized the service demand of the queue node with an APH distribution based on departure MAP.
- We applied global optimization-based and heuristics-based estimation to fit the service demand distribution with coarse-grained monitoring data, where heuristics-based estimation method can efficiently reduce the computational cost compared to the global optimization approach.
- For multi-class service demand distribution estimation, we applied class aggregation to reduce the number of classes, mapping to a collection of two-class service demand estimations, and introducing an iterative calibration to increase the accuracy of the two-class estimation by combining their individual results.
- We further showed that the proposed estimation method with both optimization and heuristic solution yields a better performance on fitting real traces of microservices compared to state-of-the-art alternatives.

Chapter 4

Code-driven Performance Model Generation

4.1 Introduction

Serverless computing is a novel cloud computing paradigm that aims at making operations concerns invisible to developers and cloud users [BCC⁺17, ESVE⁺20]. It has recently gained increasing attention in industry due to the potential for significant cost savings and on-demand billing modes. Function-as-a-Service (FaaS) is a cloud computing execution model introduced within serverless computing that allows developers to deploy single functions as basic building blocks [BCC⁺17]. Compared to monolithic applications and microservice-based architectures, FaaS-based applications can be triggered and served by events (e.g., HTTP requests) and executed on demand. There are several cloud vendors providing FaaS capabilities like AWS Lambda, Google Cloud Functions and Microsoft Azure Functions, as well as open-source alternatives such as OpenFaaS or KNative.

As FaaS is easy to use by developers with mainly focusing on writing the code logic, developers can write individual serverless functions and compose them in complex workflows deployed on the FaaS platforms. FaaS platforms enable automatic management, scaling, and billing of the execution of FaaS-based workflows to take over most operational efforts from developers and

users. However, maintaining Quality-of-Service (QoS) requirements and meeting service-level agreements (SLAs) of FaaS applications remains an outstanding concern [TPN⁺20].

Performance models provide analytical prediction and simulation results to help to reason about and improve the quality of FaaS-based applications. Accurate and efficient performance modeling benefits not only the developers and operators, but also FaaS providers. On the one hand, with performance models, the developers have a better understanding and prediction capabilities of the quality of the application under different workloads and deployment configurations, which may also help direct development decisions. On the other hand, FaaS providers can take advantage of accurate cost prediction and resource management, inferring related metrics from performance models. There are well-established stochastic models such as queueing networks [GCMM01], layered queueing networks (LQNs) [FAOW⁺08], Petri nets [MBC⁺98] that can describe the system with a simplified abstraction. Among them, LQNs are particularly suitable for capturing the dependencies and interactions between different FaaS functions.

Building performance models for FaaS applications accurately and efficiently is a non-trivial problem. Our insight is to exploit established control and data flow analysis methods [NNH15] to improve the granularity of performance models for FaaS-based applications, ultimately improving the accuracy of models and performance predictions. However, building LQN models for FaaS functions and workflows is still challenging due to the information gap between modeling and monitoring granularity compared to the classical performance modeling for web applications and microservice-based applications. Meanwhile, high-level automation for performance modeling is required to trade manual instrumentation off modeling cost and precision.

The first challenge in building LQN models for FaaS applications is learning the topological graph representing the application behavior on both inter- and intra-function levels. Attempting to accurately and completely reconstruct this structure only from traces or monitoring data may be difficult because it relies on the test inputs capable of covering all the relevant execution traces. However, when functions are observed as black boxes, i.e., without knowing which parts of their code have been exercised, there is no reliable way to ascertain whether any behavior has remained uncovered. In turn, the LQN model inferred from such partial traces may itself

be incomplete. If any feasible paths of the function have not been exercised enough during data collection phase, the LQN model could be incomplete.

Additionally, appropriate model parameterization is critical to define effective and efficient parameter estimation methods. Estimating service demand for individual endpoints from system monitoring measurements, like utilization or response time, is particularly challenging, with most methods typically resorting to regression algorithms to combine different measurements [SCBK15]. However, these methods estimate service demand based on queueing theory and may lead to inaccurate results due to the uncertainty introduced by the approximation based on the queueing theory. At the same time, the end-to-end monitoring cannot explicitly explore the data for model parameterization, since it is coarse-grained that cannot match the granularity of the internal modeling. Thus, a fine-grained and less-overhead tracing methodology is needed for the parameterization during LQN generation.

This paper proposes to build performance models for FaaS workflows combining static analysis and code profiling. We assume that the source code and configuration metadata of FaaS functions and workflows are accessible. To learn the topology of the model, we apply static analysis on the source code to obtain the inter-procedural call graph of the orchestrator defining the workflow composing the individual functions, and the intra-procedural control flows for each function. To more accurately characterize the model parameterization, we propose to inject code to hook system function calls during the profiling stage and capture the distribution of the service demand based on profiling data instead of estimating based on system measurement. The profiling data, being measured within the process executing the function, depends only on the function inputs, while it is largely workload-independent since queueing time does not affect the measures. Then, we derive the LQN models for serverless workflows by mapping the static graphs and code profiling data. In the experiments, we implement FaaS-based workflows representing different function compositing patterns to evaluate our proposed method. We compare the results by solving LQN to the data collected from the workflow execution. The experimental results yield model predictions with a mean error under 7.3% in all the evaluated scenarios.

The remainder of the paper is structured as follows. In Section 4.2, we give background on static analysis, taint analysis and performance parametric dependencies. In Section 4.3, we discuss the methodology of building LQN model based on static analysis and code profiling. In Section 4.4, we conduct experiments with different FaaS workflows to evaluate the effectiveness and efficiency of our proposed modeling method. We discuss the threats to validity in Section 4.5. In Section 4.6, we draw conclusions of our contribution in this chapter.

4.2 Preliminaries and Background

4.2.1 Intra-procedure and inter-procedure analysis

In the static analysis of source code, each code statement is evaluated to extract the control flow. Inside a single function, each node represented in the CFG can be a compilation unit. The edges between nodes in the CFG for a single function can depict the dependencies between different code blocks. Intra-procedural analysis can be applied to capture the available information on execution procedures in a single function. Thus, at the intra-procedural level, CFGs can capture all the possible execution orderings, subject to the decisions at conditional nodes.

While for serverless workflows, we are able to learn the dependencies and relationships between function interactions. Intra-procedural CFGs can be related to one another via the program call graph. For each caller function, a call graph captures all the callee functions it can invoke, providing an inter-procedural representation of the dependencies and interactions across functions. Function execution information can be propagated along inter-procedural control flow edges.

4.2.2 Taint analysis

Taint analysis is a data flow analysis that can track the impact of the program variables on different locations of the code statements. While typically used for security purposes [TPF⁺09],

taint analysis can capture the path in which the input parameters flow into function invocations.

Taint checking is the process of examining data for tainted marking and enforcing defined policies through the presence of tainted data. Static taint analysis usually consists of rules to define the source of the tainted data (e.g., user inputs, file I/O) and track the tainted data throughout the given code. The result of the static taint analysis can specify the sanitized locations (e.g., function invocations) of the tainted data, in order to prevent any vulnerabilities like injection attacks. In contrast, dynamic taint analysis is basically set up during execution and monitoring to detect if the tainted data is reaching the defined sensitive operations or resources to raise alarms for security.

There are some key definitions used in the following section of taint checking:

- **Source:** Taint sources can be data inputs or sources of data that may be manipulated to be insecure. For example, a taint source can be an operation to get the value of an HTTP request parameter or an input variable of a given function.
- **Sink:** Taint sinks can be locations in the functions, programs, or systems where the tainted data can be used or operated. In the context of code analysis, taint sinks can be referred to function calls or function invocations that involve the tracking inputs or parameters.
- **Propagation:** Taint propagation is a method in taint checking to determine the flow routes of tainted data from the sources to sinks. Typically, tainted data can be transmitted from a tainted source to the name variable. Subsequently, the data can also be transmitted into other variables and operations through the program.

4.2.3 Parametric dependencies - a performance-related review

Parametric dependency between input data and function operations refers to how the behavior of a function or computation is influenced by the values of one or more parameters (input variables). In other words, it describes how changes in input parameters affect the outcome or behavior of a function or operation. This concept is fundamental in various fields, including

mathematics, engineering, computer science, and machine learning. This concept is fundamental in computer programming, where functions are defined to take inputs (parameters) and produce outputs based on those inputs.

In terms of the development of systems, with current adoption of DevOps, software development life cycle is shortened and high-quality systems and applications are to be delivered continuously. Parameterizing the performance models in the context of DevOps is more challenging for frequent releases and new features. This is because parameters like resource demands and conditional logic usually depend on the input parameters and functional updates. Therefore, learning parametric dependencies can improve the accuracy of performance prediction.

To identify and specify the parametric dependencies of performance models, reverse engineering is also adopted in some works including static analysis and dynamic analysis of source codes. In [KKKR08], a Palladio component model-based performance prediction model is implemented by analyzing codes to extract the parametric dependencies. In particular, this work employs both static analysis of source code and dynamic analysis of binary code to extract the data flow and control flow to reverse the dependencies of input parameters.

Another work involving reverse engineering is in [KKR10]. In this work, the authors address the issues of one key factor that can be used to predict response times is depended on both input parameters and running environments. With genetic search method, the authors use code analysis and runtime monitoring to identify the parametric for the abstraction of parameterized behavior models.

In [MK18b], the authors analyze the source code with static analysis to extract the structure and control flow and also employ dynamic analysis into monitoring data and then update the performance models with parametric demands of resource and branches. Moreover, an automated updating and transformation of performance models have been implemented for agile development.

Learning parametric dependencies for performance models contains identifying the parametric dependencies and representing the relationships between dependent and independent vari-

ables [GEE⁺19]. The authors in [BKR07] provide specifications of parametric dependencies for branch, loop and resource demand within PCM to improve the prediction accuracy. In [CCT08b], the authors propose to construct the call path and detect the parametric dependencies from runtime monitoring data. This approach first analyzes event records and extracts call graphs and control flows. Parametric dependencies are determined with values of input parameter and features of control flows by approximating the probability mass function. In [EWvK18], both dependencies and independent relations are leveraged to specify the parametric dependencies for component-based applications, which has been implemented with Descartes Modeling Language, as well as in [WEG⁺18]. [GEE⁺19] provides a machine learning-based approach to identify parametric dependencies from monitoring data automatically. They formulate the extracting dependencies between parameters as feature selection problems and apply feature selection methods including filter, wrapper and embedded to determine the dependencies from feature space. To describe the relations between indecent and dependent variables, the authors in [AGEK18] evaluate the performance of ten machine learning methods on representing the parametric dependencies. Moreover, they propose a meta-selection model for choosing the optimal algorithm to capture the relationship based on a decision tree.

4.3 Methodology

Our methodology for modeling serverless applications combining static analysis and code profiling includes three main phases: a static analysis to learn the topology for LQN structure modeling, a dynamic analysis with code profiling to collect data for model parameterization, and LQN model generation.

Overview. The overview of the proposed methodology is shown in Figure 4.1. Assume that after the development of individual serverless functions or FaaS-based workflows, we are able to access and instrument the source code and configuration metadata either from developers or cloud providers. Black-box functions whose performance models have been constructed with alternative methods can in principle be included in the LQN model as well, with possible

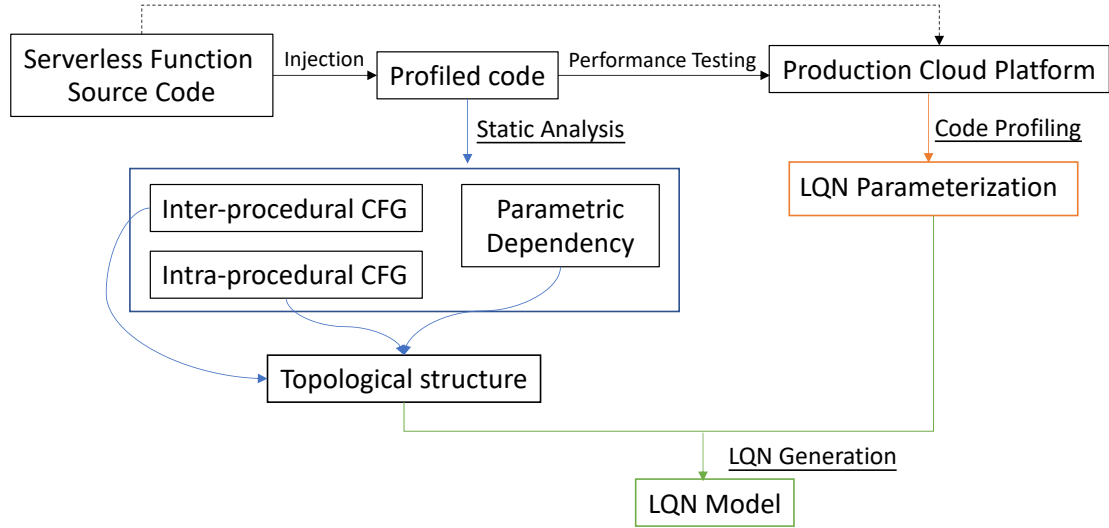


Figure 4.1: The overview of the proposed methodology

increase of the overall uncertainty of the model. However, in this paper, we focus on modeling functions whose source code is accessible. First, to define the topological structure of the workflow, we apply static analysis to extract the intra-procedural and inter-procedural control flow graphs from the source code of FaaS-based workflows, including individual functions and orchestration code composing them. Besides, we try to infer the dependencies between the input parameters of each individual serverless function and which function calls they affect; this can help to reduce the number of nodes in the topological graph. We then inject profiling instructions into the source code to enable code-level profiling during performance testing (we will refer to the instrumented code as *profiled code* in the following parts). Next, the profiled code can be deployed on the production platform as required for performance testing and data collection. After exercising the test inputs, the service demand distribution is captured with the profiling data. The availability of the static graph also allows inspecting if any static execution path has not been covered, enabling the developer to decide whether

1. the static path is effectively not executable (e.g., the FaaS application does not require all the features of a library function, thus using only some of its possible behaviors)
2. the static path implements features not relevant to ensure the SLAs thus it was deliberately not exercised during performance testing

3. the performance test suite needs improvement to cover more missing relevant application behaviors

Finally, we can generate LQN models using the topological graph for components specification, and accurately characterize the model parameters with code-profiling data. In the remainder of this section, we will detail each of the three phases.

4.3.1 Static analysis for structure extraction

We assume that there are two major components in a given FaaS-based application: an orchestration function defining workflows to compose individual serverless functions with suitable patterns (e.g., sequential or parallel execution), and a set of individual serverless functions implementing different functionalities. We first construct the topology of the LQN from static source code analysis. The static analysis provides a fast way to capture the internal control flows by identifying (a superset of) all feasible paths of the programs. This can help build a complete topological graph, whereas certain parts could be missed in monitoring data if the inputs used to test the system do not cover all of its features extensively.

Static Graph Generation. In this paper, we use both inter- and intra-procedural static graphs to derive the topological structure of the LQN models. A call graph (CG) is mainly responsible for extracting the calling relationships on the workflow, which is extracted from the orchestration function. While intra-procedural information can be obtained by generating a control flow graph (CFG) for each serverless function. Both CG and CFG are constructed by traversing nodes in the abstract syntax tree (AST) on the profiled code, resulting in a collection of code blocks and control nodes representing conditional execution [NNH15]. Combining both CG and CFG of the serverless workflow, we obtain an inter-procedural static description of the system that we call the static graph (SG) as $SG = \{CG, CFG\}$.

Given the control graph of an individual serverless function as $CFG = (\mathbf{N}, \mathbf{E})$, the control flow is formalized by conditions, loops, function calls, and sequential code blocks. In

```

1 def main(req: func.HttpRequest) -> func.HttpResponse:
2     logging.info('Python HTTP trigger function processed a request.')
3     name = req.params.get('name')
4     if not name:
5         try:
6             req_body = req.get_json()
7         except ValueError:
8             pass
9         else:
10            name = req_body.get('name')
11     if name:
12         return func.HttpResponse(f"Hello, {name}. This HTTP triggered function executed successfully.")
13     else:
14         return func.HttpResponse(
15             "This HTTP triggered function executed successfully. Pass a name in the query string or in "+
16             "the request body for a personalized response.",
17             status_code=200
18 )

```

Figure 4.2: An example source code in Python from Azure Functions

CFG , $\mathbf{N}$ and $\mathbf{E}$ denote the nodes and edges, respectively. In order to enable further analysis of the CFG, we then define each node N_i as a tuple (i, l_s, l_e) , where l_s and l_e are the starting and end lines of the i^{th} code block. A directed edge in $\mathbf{E} = \{(N_i, N_j), \dots\}$ describes the relationship between nodes N_i and N_j . The example source code in Figure 4.2 can be retrieved from the tutorial¹, and Figure 4.3 shows the control flow for the example serverless function. The resulting CFG is represented with $\mathbf{N} = \{N_1, N_2, \dots, N_9\}$, $\mathbf{E} = \{(1, 2), (2, 3), (3, 4), (4, 5), (4, 6), (4, 7), (3, 7), (7, 8), (7, 9)\}$ and, for example, $N_9 = (9, 13, 18)$ (where 9 is the 9th node in the CFG and 13 and 18 are referring to Line 13 and 18).

Data Flow Analysis for Parameter Dependency Inference. The static graph of serverless applications could be large due to redundant nodes representing the statements whose execution times are not influenced by a function's input parameters (e.g., constant time initialization). Thus, from a performance perspective, these statements could be aggregated into single blocks to reduce the size and fragmentation of the static graph, for those statements that the execution consumption is deterministic.

We apply a static data flow analysis to the profiled code to infer the potential parametric dependencies between input parameters and function calls using taint analysis. This can help

¹<https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-cli-python>

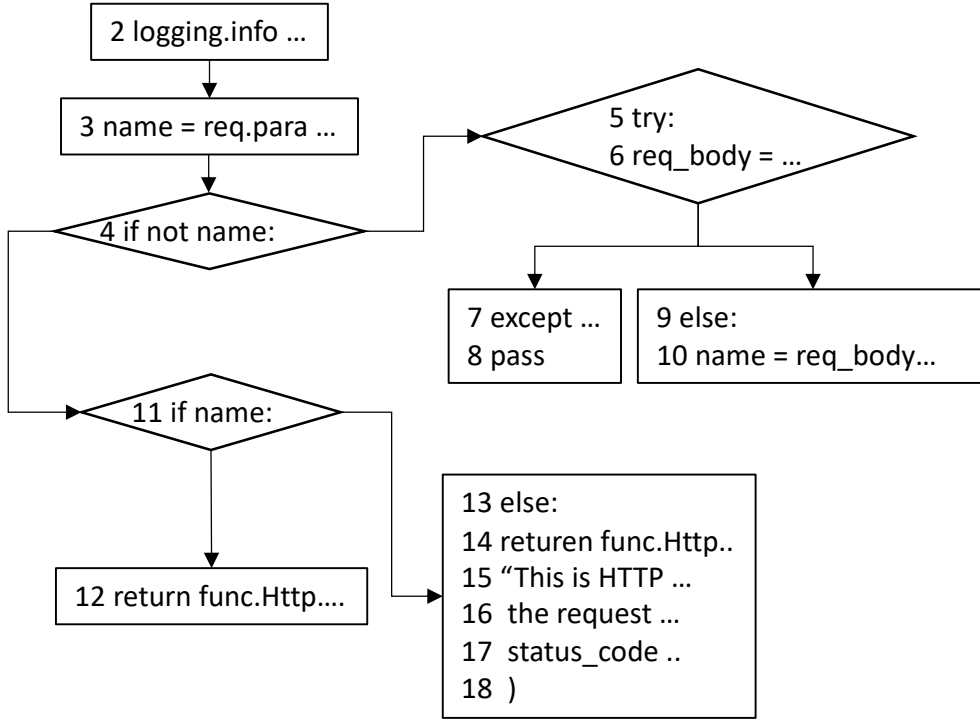


Figure 4.3: The corresponding control flow graph of the source code example in Figure 4.2.

reduce the number of nodes in a static graph by aggregating input-independent nodes, in turn lowering the computation complexity of solving the inferred LQN models. Taint analysis in this phase works by marking a statement in the source code as tainted if its execution or assigned value is affected by function input values. The parametric dependency inference can be formulated as detecting any existing tainted statement in the nodes of the CFG. If there are input parameters used by the statement of any node, we then deduce that the execution times of the detected node N_i are potentially dependent on such input parameters. Formally, potential parametric dependencies can be described as a set of $PD_i = (source, sink, lineno)$, where *source* and *sink* are the identifiers of a function input parameter and a function call whose arguments are affected by the input parameter, while *lineno* is the line number identifying the call site of the sink function to distinguish possible multiple calls.

If there is no potential parametric dependency detected in N_j , we can infer that the demand for executing N_j is not impacted by function inputs, resulting in a reduction of N_j by aggregating it with its predecessor. Figure 4.4(a) shows an example source code with the taint analysis

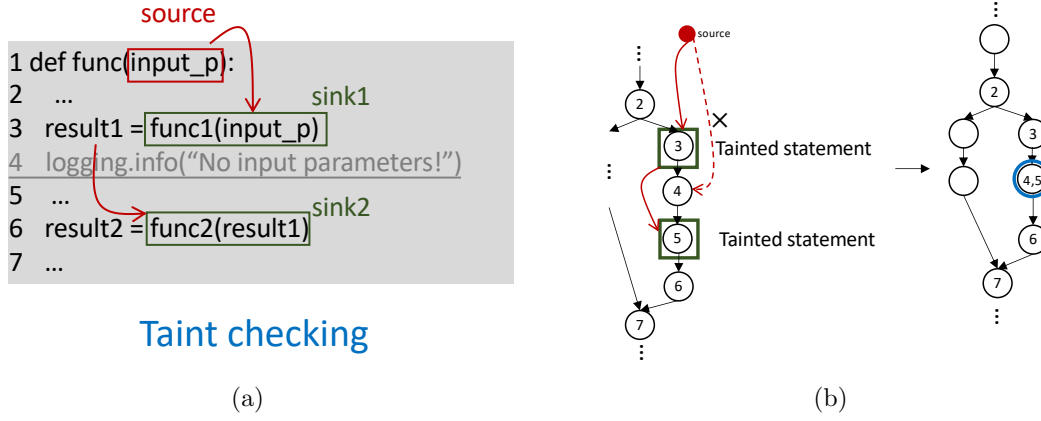


Figure 4.4: An example excerpt of code with taint analysis results highlighted (a), and the control flow graph reduction process (b).

Algorithm 5 Control flow graph reduction with potential parametric dependencies

Input: $CFG \leftarrow$ Control flow graph of the source code $CFG = (N, E)$

$PD \leftarrow$ Set of potential parametric dependencies $[PD_1, PD_2, \dots, PD_n]$

Output: $CFG_r \leftarrow$ Optimized CFG with reduction on nodes

- 1: Initialize $r_N = \emptyset$, $E_r = \emptyset$
 - 2: **for** N_i in CFG **do**
 - 3: **for** PD_i in PD **do**
 - 4: **if** $lineno$ is not in the range of $[l_s, l_e]$ **then**
 - 5: $r_N \leftarrow r_N \cup N_i$
 - 6: $succ \leftarrow$ all successors of N_i , $pred \leftarrow$ all predecessors of N_i
 - 7: update l_e of $pred$ to include N_i
 - 8: **end if**
 - 9: **end for**
 - 10: $N_r \leftarrow N \setminus r_N$
 - 11: **for** $succ_i$ in $succ$ **do**
 - 12: **for** $pred_i$ in $pred$ **do**
 - 13: $E_r \leftarrow E \setminus (pred_i, N_i)$, $E_r \leftarrow E \cup (pred_i, succ_i)$
 - 14: **end for**
 - 15: $E_r \leftarrow E \setminus (N_i, succ_i)$
 - 16: **end for**
 - 17: **end for**
 - 18: **return** $CFG_r \leftarrow (N_r, E_r)$
-

results highlighted. Let the left graph in Figure 4.4 be the original CFG of the example code, with $N_4 = (4, 3, 3)$ and $N_5 = (5, 4, 4)$. Figure 4.4(b) shows the reduction of node N_5 into node $N_{4,5}$ due to no detected parametric dependencies in node N_5 .

Algorithm 5 formulates the control flow graph reduction process based on taint analysis. The algorithm takes the intra-procedural control flow graph and the detected parametric depen-

Table 4.1: Notations in Algorithm 5

N_i	Calculate the mean value of the given data
PD_i	Try to find the feasible solution by matching the first three moments
$\mathbf{r}_N$	Fit APH distribution from the first three moments by moment matching
$\mathbf{E}_r$	Fit Erlang distribution with the mean value and scv
suc_i	sucessors
$pred_i$	predecessors

dencies as inputs. The algorithm traverses N_i in the original CFG and checks if any potential parametric dependencies occurred at N_i . If no dependency exists, the current node N_i is added to the untainted set $\mathbf{r}_N$. Additionally, it requires operating on the edges that would be affected by removing the untainted node. At Line 6 and 7, the algorithm first finds all predecessors and successors of the untainted node and then revises the end line number of all predecessors with l_e of N_i . Then at Line 10, the untainted node is removed from the original graph. From Line 11 to Line 16, the algorithm iterates on the nodes and removes the edges containing the affected nodes. By fully connecting the nodes in **succ** and **pred**, the new edges are generated to form the untainted edge set $\mathbf{E}_r$. After iterating on all the nodes in the original graph, the reduced graph $CFG_r = (\mathbf{N}_r, \mathbf{E}_r)$ is generated by combining nodes in which there are no function calls or parametric dependencies. It can be noticed that the parametric dependency inference is only capable of detecting potential relationship between function calls and input parameters from the code syntax. For example, consider $y = 0*x; f(y)$; most taint analyses would conclude that the invocation of f may depend on the input parameter x . This may lead to a conservative over-approximation, with possibly only a subset of the statically detected dependencies satisfied during runtime. In this case, the static graph could have been further reduced, realizing that $0*x$ is identically 0. Nonetheless, even when non-optimal, taint analysis may still help to reduce the size of the static graph.

4.3.2 Code profiling for model parameterization

Application-level monitoring data may be too coarse-grained to accurately infer service demand parameters of LQN activities, representing the operations inside individual serverless functions. However, it is not sufficient to match and characterize the accurate service demand of the activities corresponding to individual functions or parts of functions described in LQN models. Thus, it requires more fine-grained measurement data inside functions to instead propose instrumenting the code of a function to obtain fine-grained measurements that can bridge the information gap between the granularity of the topology extracted via static analysis and the data used for model parameter inference.

Code-level profiling. Code-level profiling provides the possibility to obtain the workload-independent execution profile of the individual serverless function, enabling a more accurate LQN model parameterization. To avoid changing any functionality of the source code and try to instrument the code as less as possible, we only wrap the *MAIN* function block into a wrapper function and inject a decorator to record the execution times with a standard line-level profiler [lin] (while we refer mainly to Python code in this work, similar profiler utilities exist for all mainstream programming languages), without additional manual instrumentation on the rest of the code. Considering the granularity of the information provided by the static graph, we implement the code-level profiling with a standard line-level profiler [lin] that can record the execution information line by line at the profiling phase. The only instrumentation is injecting a wrapper function above *MAIN* function and a decorator for the line profiler. Here, we take the assumption that the performance test inputs are representative of all relevant production behaviors. If executable paths in the static graph are not covered by the current test inputs, while they may affect the application’s SLAs, the developer has the opportunity to identify the gap and produce additional performance tests. As workload-independent execution times can be profiled, we can perform offline profiling on the production platform as required and then deploy the non-instrumented functions to the target cloud service without the profiling instruction.

We denote a sample from the collected profiling data as $s = (lineno, dt, iter)$, where dt is

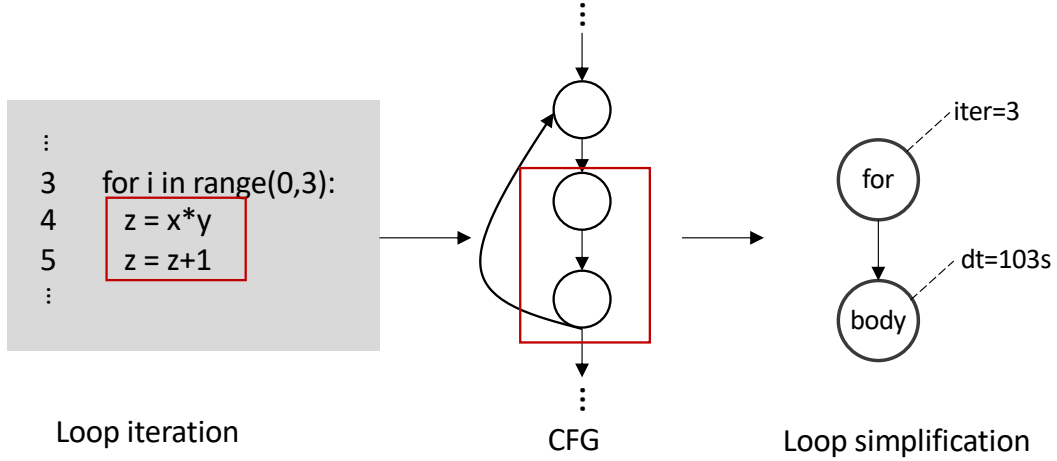


Figure 4.5: Loop simplification in LQN modeling

the execution duration of the statement at line $lineno$, and $iter$ is the iteration counter to distinguish different iterations in a loop. We can then map the profiling data into the static graph according to line numbers $lineno$ in s and (l_s, l_e) of nodes in CFG_r to extend the static graph with profiling data; we will refer to this extended structure as profiled static graph.

Besides obtaining the execution times of nodes in the graph, we also need to learn the probabilities of branches and the number of iterations for loops to infer the remaining parameters of an LQN model. For the branch probabilities, we define the executed path of each test input request $\mathbf{eP}$. Each $eP_i \in \mathbf{eP}$ represents one of the feasible paths in the static graph that has been executed according to the profiling data $\mathbf{s}$. Therefore, the probability of a given selection path ^{i} on each conditional statement can be derived as the fraction of eP_i taking each branch over the number of eP_i evaluating the corresponding condition.

For loop iterations, we represent the body of *for* or *while* loop as an entire activity and infer the expected number of iterations from the profiling data, which is consistent with the typical specification of LQNs. There is a simple illustration in Figure 4.5. This can be further optimized by considering the branch probabilities inside loops to indicate a probabilistic loop, however, this is out of the scope of our current modeling method. In our proposed method, the number of iterations of loops in each execution can be directly extracted from the profiled data with $iter$.

Algorithm 6 Activity service demand estimation**Input:** $dt = (dt_1, dt_2, \dots, dt_n) \leftarrow$ Profiled execution time for each node in the CFG**Output:** $sd \leftarrow$ Service demand distribution for activities in the CFG

```

1: Initialize  $sd = \emptyset$ 
2: for  $i = 1$  to  $n$  do
3:    $m1 = \text{mean}(dt_i)$  /*mean: calculate the mean value*/
4:    $m2 = \text{mean}(dt_i.^2)$ 
5:    $m3 = \text{mean}(dt_i.^3)$ 
6:    $isfeasible = \text{CheckAPH}(m1, m2, m3)$ 
7:   if  $isfeasible$  is True then
8:      $APH(\alpha, T) \rightarrow \text{APHFit}(m1, m2, m3)$ 
9:      $sd_i = APH(\alpha, T)$ 
10:  else
11:     $scv = m2/m1^2 - 1$ 
12:     $sd_i = \text{ErlangFit}(m1, scv)$ 
13:  end if
14: end for

```

Table 4.2: Description of functions used in Algorithm 6

mean	Calculate the mean value of the given data
CheckAPH	Try to find the feasible solution by matching the first three moments
APHFit	Fit APH distribution from the first three moments by moment matching
ErlangFit	Fit Erlang distribution with the mean value and scv

LQN Activity Service Demand Distribution. Service demands are critical parameters for the specification of activities, as they represent the cumulative computation time the activity requires to run. To capture the demand of the activities in LQN, we model the service demand distribution with acyclic phase-type (APH) distributions and Erlang distributions by moment matching. The details of the service demand distribution estimation is implemented according to Algorithm 6. Based on the execution duration dt in the profiling data, we first try to fit an APH distribution by matching the first three moments of dt as shown in Line 3 to Line 5. If there is no solution for APH distribution with the current data, we then fit an Erlang distribution with mean value and squared coefficient of variation (scv). In this way, the service demand of each activity can be directly characterized by the profiling data.

4.3.3 LQN model generation

To construct an LQN model from the profiled static graph, we first define a reference task to represent the incoming workload and an orchestration task to abstract the workflow logic composing individual serverless functions. Then, each individual serverless function can be modeled with a single task hosted on a separate processor, since it can be deployed with different configurations of resources and even to different platforms.

The entry node of the LQN is then specified according to the entry point of each function. We further assume that the sequential or parallel (fork-join) composition of the functions is specified in the orchestration function, e.g., using Azure Functions code constructs. The degree of concurrency allowed to each function is specified in the configuration metadata. The scheduling policy of the processor can be specified as either First-come-first-serve (FCFS), if the source code is with single-thread implementation, or Processor-sharing (PS) if function invocations can be interleaved on the same processor. Both scheduling policies are supported in LQN modeling [FMW⁺05].

LQN Activity Graph Characterization. The static graphs and profiling information collected so far allows for a systematic construction of the LQN model. First, we consider that the orchestration function is allowed to specify the workflow patterns with HTTP calls to invoke the individual serverless functions. Each activity inside the orchestration entry can be defined according to the nodes in the call graph and takes the role of sending synchronous and asynchronous calls for parallel execution to the entries of individual functions in the lower layers of the LQN. Whereas, the skeleton of the activity graph for an individual function can be directly derived from the reduced CFG. The activity graph representing the set of activities ***act*** is defined as $AG = \{\mathbf{act}, \mathbf{sd}, \mathbf{prec}\}$, where ***sd*** presents the service demand and the precedence relation among activities is denoted as ***prec***.

Now we discuss the procedure of activity graph specification for the serverless function f following the approach in [FAOW⁺08] in Algorithm 7. For each activity representing N'_i , all the successors and predecessors of N'_i are computed at Line 5. There are mainly 4 types of activity

precedence included in our method from Line 6 to Line 16:

1. If the current node is included in its predecessors, it indicates that loops are occurring at N'_i which can be extracted with the number of iterations $iter_i$ derived from profiled data.
2. If the current node only has one successor and one or fewer predecessors, it means that N'_i is sequentially connected to its successor.
3. When there is more than one successor of N'_i , there are branches with *IF* or *SWITCH* statements for jumping to different nodes, whose branching probabilities have previously been computed from profiling data.
4. If there is more than one predecessor, different conditional blocks can be merged at N'_i .
From the orchestration function, we also capture parallelism and synchronization among the execution of different serverless functions.

Combining all the listed cases, our method can describe the operator precedence in the activity graph including sequential interactions, conditioning and merging on branch nodes, as well as fork-join synchronization.

4.4 Evaluation

In this section, we first introduce the experimental setup and metrics to evaluate the accuracy of performance models constructed with our method. The comparison of LQN model predictions against execution monitoring traces for serverless workflows with different composition patterns is presented afterwards.

4.4.1 Experimental setup

There are existing serverless workflow applications like those discussed in [EGVE⁺20, aws]. However, considering the representation of different workflow execution patterns in practice,

Algorithm 7 Algorithm for Activity Graph Precedence Extraction

Input: $N \leftarrow$ Nodes in CFG of serverless function f
 $prob \leftarrow$ A set of branch probabilities of each node in CFG
 $iter \leftarrow$ A set of number of iterations of each node in CFG
Output: $prec \leftarrow$ Defined precedence in the activity graph for serverless function f

- 1: Initialize $prec = \emptyset$
- 2: $N' \leftarrow DFS(N)$
- 3: **for** N_i in N' **do**
- 4: Initialize AcP
- 5: $suc \leftarrow$ all successors of N_i , $pred \leftarrow$ all predecessors of N_i
- 6: **if** N_i in $pred$ **then**
- 7: $AcP \leftarrow Loop(N_i, suc \setminus N_i, iter_i)$
- 8: **else**
- 9: **if** $|suc| = 1 \& |pred| \leq 1$ **then**
- 10: $AcP \leftarrow Seq(pred, N_i, suc)$
- 11: **else if** $|suc| > 1$ **then**
- 12: $|suc| > 1, AcP \leftarrow OrFork(N_i, suc, prob_i)$
- 13: **else if** $|pred| > 1$ **then**
- 14: $AcP \leftarrow OrJoin(pred, N_i)$
- 15: **end if**
- 16: **end if**
- 17: $prec \cup AcP$
- 18: **end for**
- 19: **return** $prec$

these public benchmarks are either too simple to reveal real-world workflow orchestration or too complex for migration to our desired public cloud platform. Thus, for simplification, we aim to design and implement serverless workflows under evaluation with common workflow compositions that satisfy our current supported programming language.

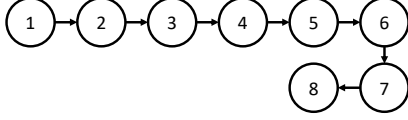
Inspired by the transfer of classical machine learning-based prediction and classification to serverless execution on public cloud platforms [tut], we first implement four serverless workflows in Python as examples in the following experiments to assess the performance of our proposed methodology. Our implemented serverless workflows include sequential, branching, parallel and complex execution scenarios, which is shown in Figure 4.6. The source code of the serverless workflow implementation is available at this repo².

We create 13 serverless functions and 4 orchestration functions to define a collection of common workflow patterns on Microsoft Azure, where it provides Azure Functions Service for serverless

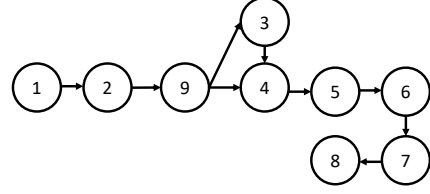
²<https://github.com/runanwang07/Serverless-workflows>

Table 4.3: FaaS-based workflow patterns

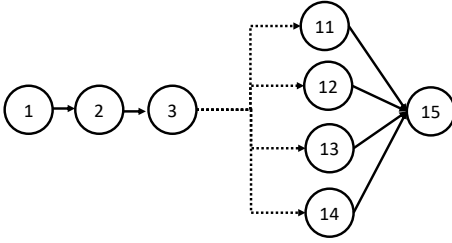
	<i>wf1</i>	<i>wf2</i>	<i>wf3</i>	<i>wf4</i>
Number of functions	8	9	8	14
c^2 of execution times	0.26	5.71	0.84	7.44



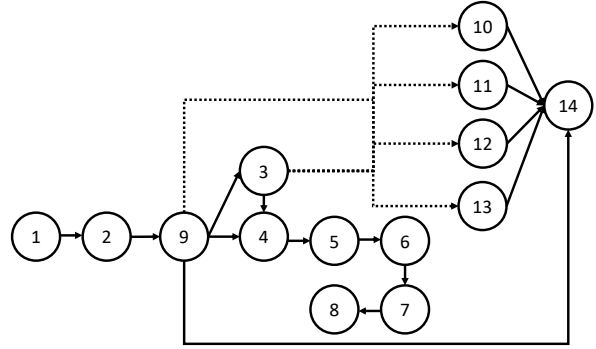
(a) wf1 - Sequential



(b) wf2 - Branching



(c) wf3 - Parallel



(d) wf4 - Complex

Figure 4.6: Serverless workflows patterns in the experiments

execution and management. The individual serverless functions are adapted from public examples that use TensorFlow with Azure functions [aze] and models from Onnx Model Zoo [onn]. The functionality of different workflows aims to simulate scientific workflows for image classification and prediction, including preprocessing of input images and classification based on machine learning algorithms or pre-trained models. Different parameters of the composition workflows implemented by the four orchestration functions are shown in Table 4.3, where c^2 is the squared coefficient of variation of the execution times. It can be observed from Table 4.3 that the values of SCV vary across different workflow patterns, providing varying spreads of the distributions of the execution times for later comparisons.

To evaluate the accuracy of our modeling method, we conduct several experiments with dif-

ferent workloads and compare the performance predictions from the LQN models against the application-level monitoring data of the serverless workflows. This is because in practice, it is not always feasible to measure the execution times directly from the serverless workflow execution. Instead, coarse-grained response time data is often easily captured by the monitoring tools without any instrumentation of the workflows.

The experimental environment is as follows: All individual serverless functions are developed using Python 3.7 and deployed with Azure Functions 3.0. To generate experimental data, we conduct performance testing for both profiled and non-profiled code to generate closed workloads with different intensities using Locust [loc]. Subsequently, we utilize the response times of requests from real traces as ground truth to evaluate our model-based predictions. For real trace collection and storage, we employ Azure Application Insights as the monitoring tool and expose the code profiling data on the same service, which is integrated with Azure Function to easily adopt. Before collecting trace data, we typically spend 3 minutes spawning concurrent users and warming up individual functions to mitigate the impact of the cold-start phase.

Given that workload-independent execution times can be profiled in isolation, we operate under the assumption that we can conduct offline profiling on the production platform as needed to collect profiling data, utilizing the profiled source code during performance testing. Additionally, during the offline profiling, we assume that the performance testing can accurately represent the expected user behaviors in production and that the data quality from the performance testing is sufficient to model the serverless workflow. Then we deploy the non-instrumented functions to the target cloud service (without the profiling instructions) to collect application-level runtime monitoring data.

The static analysis is built on top of the *ast module* in Python 3.7. For the taint analysis on the static data flow, we use the open-source tool Pyre shipped with Pysa [Pys] to infer the potential parametric dependencies. To obtain the analytical results, we use LQNS via LINE to solve the generated LQN models [Cas20].

We compare the model prediction accuracy of mean response times to the collected traces,

Table 4.4: The comparison results based on LQN node reduction

	Number of Activities		Execution times (s)		MRE	
	Original	Reduced	Original	Reduced	Original	Reduced
<i>wf1</i>	91	67	2.793	2.092	0.044	0.030
<i>wf2</i>	96	69	2.804	2.133	0.029	0.038
<i>wf3</i>	27	21	1.545	1.542	0.222	0.236
<i>wf4</i>	122	89	3.682	3.257	0.103	0.117

using mean relative error (MRE) as our comparison metric

$$MRE = \frac{|m - m'|}{m} \quad (4.1)$$

where m is computed with the mean response times of the monitored execution traces and m' is from LQN predicted response times.

4.4.2 Experimental results

We first evaluate the static graph reduction based on inferring static parametric dependencies. Next, to evaluate the accuracy of parameterization for LQN models, we conduct extensive experiments with different settings of the number of processors and the dynamic auto-scaling to simulate two resource provision scenarios. Here, we regard these two experimental settings as limited resources and sufficient resources to simulate the realistic scaling scenarios in the following discussion.

LQN Model Node Reduction. In Section 4.3.2, we introduced Algorithm 5 to reduce the size of the static graph by aggregating code blocks independent of input parameters, with the ultimate goal of further reducing the size and complexity of the generated LQN models. Here, we compare the accuracy and efficiency of the original LQN models to the reduced LQN models. From Table 4.4, we can observe that after node reduction, for example, the number of activities of *wf2* is reduced to 69, which indicates that nearly 30% of nodes have been merged according to the static parametric dependencies. Besides, it can be noticed from the table that the execution times of solving model are decreased by up to 25% for *wf2*, while the MRE increases only to a small degree for all the subjects, which is likely an acceptable trade-off

Table 4.5: MRE of compared results in Figure 4.7

	workflow pattern				workload		
	<i>wf1</i>	<i>wf2</i>	<i>wf3</i>	<i>wf4</i>	1	5	10
Model	9.081	0.927	14.728	1.923	9.081	9.081	9.081
Measurement	9.501	0.901	13.912	1.744	9.501	9.613	8.390
MRE	0.044	0.029	0.058	0.103	0.044	0.055	0.082

between prediction accuracy and model complexity in most situations. We can conclude that the reduction of nodes in static graphs can directly help to reduce the number of activities in the LQN model, thus saving analysis costs. The savings come with a marginal increase in the MRE for three out of four subjects, while the MRE marginally decreased for *wf1*. Overall, the impact of reduction on the MRE appears marginal.

Sufficient Resource. In the following experiments with sufficient resources, we assume that dynamic auto-scaling is enabled for each FaaS function and there is no need to operate on the configuration of the resources. In LQN models, we set the multiplicities of each processor to 100 to simulate sufficient resource provision not to limit the scaling out of the individual functions.

We evaluate the above 4 different workflow patterns and take *wf1* as an example under different workloads. The comparison of LQN model predictions against the real traces is shown in Figure 4.7, and the details of model accuracy evaluation are in Table 4.5. It can be seen from Figure 4.7(a) that the mean response times among different workflow patterns vary in a range of 0 to 15 seconds, while all prediction results are close to the measurements. Figure 4.7(b), which zooms on *wf1*, shows that there is no obvious increase in response times as the number of clients grows. This is because under sufficient provision, all required resources can be allocated and there are no significant queueing times for each request. Therefore, the LQN modeling results capture the correct trend of response times changing with workloads. Besides, it can be observed from Table 4.5 that the prediction of the LQN model yields good accuracy with an average MRE over the four workflows of 5.5% (min=2.9% for *wf2*, max=10.3% for *wf4*), indicating a fairly accurate characterization of the performance of the FaaS workflows.

Limited Resource. For the limited resources experiments, we tune the configuration of each serverless function to vary the number of cores for the processor. Practically, we first identify

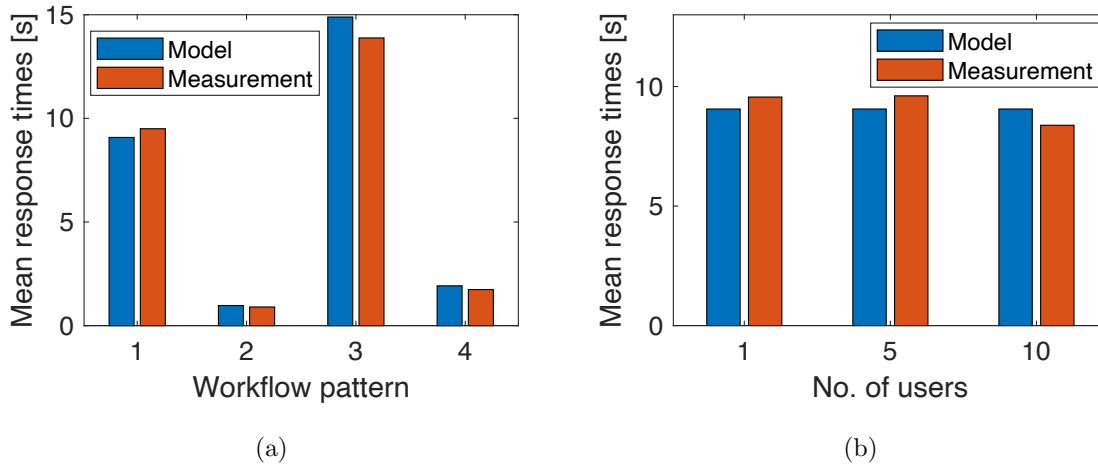


Figure 4.7: Mean response times of different workflow patterns (a) and of *wf1* under different workloads (b), comparing model prediction and real trace measurements.

the most resource-demanding serverless function as the bottleneck function and then study the accuracy of our model for different values of the maximum number of instances on Azure and, coherently, of the multiplicity parameter of the corresponding LQN activity.

However, when a single processor is allowed, Azure Function Consumption plan limits the allocated memory to 1.5Gb, which in the case of *wf3* and *wf4* is not sufficient to serve 10 users without scaling strategies for the most resource-demanding serverless function. Therefore, we selected the second most resource-demanding serverless function as the bottleneck function for *wf3* and *wf4*. The comparison of results on LQN model prediction and monitoring traces measurements is shown in Tables 4.6 and 4.7.

First, we investigate the model performance with increasing concurrent users between $N = 1$ to 10 with only one processor ($P = 1$) available for the bottleneck function. It can be seen from Table 4.6 that, with increased workload intensity, the mean response time grows with different trends. For example, in *wf1*, the response time with 10 users is nearly 5 times higher than with 1 user due to the contention on the bottleneck function that forces the users to wait. Nevertheless, regardless of the variation trends in the response time, the model predicts accurately the performance measurements from monitoring traces in all four workflows, with average MRE across all the experiments of about 6.2% (min=0.8% for *wf1* with $N = 10$, max=12.2% for *wf4* with $N = 10$).

Table 4.6: Comparison results on different number of processors on 4 workflow patterns with limited resource $P = 1$

	<i>wf1</i>			<i>wf2</i>		
	$N = 1$	$N = 5$	$N = 10$	$N = 1$	$N = 5$	$N = 10$
Model	9.081	22.718	49.016	0.927	1.454	3.085
Measurement	9.501	24.397	48.606	0.901	1.394	2.952
MRE	0.044	0.069	0.008	0.029	0.043	0.045
	<i>wf3</i>			<i>wf4</i>		
	$N = 1$	$N = 5$	$N = 10$	$N = 1$	$N = 5$	$N = 10$
Model	10.144	10.664	12.545	1.923	2.229	3.215
Measurement	9.737	12.125	13.626	1.744	2.3	3.655
MRE	0.042	0.120	0.079	0.103	0.031	0.122

Table 4.7: Comparison results on different workloads on 4 workflow patterns with limited resource $N = 10$

	<i>wf1</i>			<i>wf2</i>		
	$P = 1$	$P = 5$	$P = 10$	$P = 1$	$P = 5$	$P = 10$
Model	49.016	10.325	9.083	3.085	0.928	0.927
Measurement	48.606	13.27	8.018	2.952	1.149	0.879
MRE	0.008	0.222	0.133	0.045	0.192	0.054
	<i>wf3</i>			<i>wf4</i>		
	$P = 1$	$P = 5$	$P = 10$	$P = 1$	$P = 5$	$P = 10$
Model	12.545	10.144	10.144	3.215	1.923	1.923
Measurement	13.626	8.600	10.489	3.655	2.031	1.895
MRE	0.089	0.180	0.033	0.120	0.053	0.015

Next, we evaluate model prediction accuracy using an intense workload ($N=10$) and varying the number of processors available for the bottleneck function. Table 4.7 shows the comparative data for the number of processors P between 1 and 10. As expected, increasing the number of processors reduces the response time for all the workflows, albeit with different trends. The average MRE across all the experiments is in this case about 9.5% (min=0.8% for *wf1* with $P = 1$, max=22.2% for *wf1* with $P = 5$). While average MRE remained under 10%, we observed a performance deterioration for $P = 5$. By observing the execution traces, we conjecture this deterioration may be caused of some implicit optimization or automation happening on the serverless platform around $P = 5$ which is not accurately captured by our models and may require additional investigation.

Summary. The evaluation of our LQN modeling strategy for serverless functions based on static analysis and code profiling may be summarized with the following two observations. First,

node reduction on the static graph leads to smaller LQN models, saving computation time for both LQN model generation and model-based performance prediction, with negligible impact on prediction accuracy. Second, model-based performance prediction achieved a close fit to the measurements from monitoring traces (average MRE=7.3%), under different workload intensity and in both sufficient and limited resources. Finally, we remark that the availability of the static graph also allows for assessing the coverage of the performance test inputs, highlighting possible execution paths relevant to the satisfaction of the application's SLAs that are not exercised (enough), thus driving the refinement of the performance test suite.

4.5 Threats to Validity

To enhance the robustness and applicability of our methodology, it is imperative to thoroughly examine potential threats to validity that could impact our results.

- **Programming language selection:** For the experimental and evaluation simplification, we choose Python-implemented serverless functions that may limit the generalizability of the methodology, as other languages like JavaScript and Java are commonly used in practice. Potentially, the entire methodology may be applied to other programming language-based workflows by adapting the algorithms to analyze the AST for constructing CFGs accordingly. However, we do assume that CFGs of adequate precision to allow the desired granularity of instrumentation can be computed.
- **Accessibility of source code:** Assuming that all source code is accessible may not always hold true in practice. While we can model the invocation of functions whose code is not available similar to the invocation of an external service, more experiments may be needed to assess the impact of this approximation on the overall modeling accuracy. Developing methods to capture coarse-level information for black-box functions instead of depicting the entire activity graph could ensure that the methodology can still be applied in scenarios where complete access to source code is not feasible.

- Comparison of modeling granularities: While comparing the performance of fine-grained models to real execution monitoring, there may be a lack of comparison between coarse-grained and fine-grained modeling. We have compared the predicted results from fine-grained models to the collected monitoring data, showing good modeling ability with low error. It could be helpful to include comparisons between coarse-grained and fine-grained modeling to demonstrate the accuracy improvement of prediction as the modeling granularity increases.
- Representativeness of profiling data: We take the assumption that profiling data is representative of workflow behaviors. However, this may lead to incomplete fine-grained models due to coverage limitations. In our experiments, we perform controlled experiments to exercise each possible path with different inputs to ensure the coverage of all feasible execution paths. Potential solutions for realistic applications may be integrating techniques to improve experimentation coverage into the methodology to enhance the characterization of real behaviors and mitigate the risk of generating incomplete models.

4.6 Summary

In this chapter, we presented a new method to build LQN performance models for serverless applications using information from static analysis to enhance model-based prediction accuracy. In summary, our contributions for performance modeling of serverless functions are as follows:

- We applied static source code analysis into serverless functions to extract CFGs and CGs to learn the code-level model skeletons for performance model generation.
- We introduced static dataflow analysis for parametric dependencies to optimize the model structure with taint checking to reduce the static graph.
- We utilized code-level profiling to learn the fine-grained execution information, bridging the information gaps between structure and observability.

- We proposed an automated LQN model generation methodology and improved the model characterization with enhanced parameterization.
- We evaluated our proposed method with serverless workflows on Microsoft Azure Functions to show the modeling accuracy with different resource provisions.

Chapter 5

Fine-grained Performance Diagnosis with Distributed Tracing

5.1 Introduction

Distributed tracing has emerged in recent years as a technique to achieve observability in end-to-end applications with less effort and lower costs than traditional logging and monitoring tools [PSM⁺20]. In the context of serverless computing, distributed tracing can be used to extract the dependencies between different serverless functions and improve the detection of anomalies arising in their composition [SKM22]. The technique leverages code instrumentation to record runtime data at different levels of granularity, ranging from monitoring end-to-end request paths to observing individual code portions within a serverless function. Some instrumentation frameworks and libraries, such as OpenTelemetry [tel], offer developers the ability to easily inject tracing commands in their codebase across a large range of programming languages.

Tracing probes are code snippets that specify which parts on execution should be monitored, thus defining the logical structure and content of the trace data records – called *spans*. A developer can design a distributed tracing that delivers observability into the performance of an application at the desired level of granularity for each of its parts. However, deciding an appropriate placement of instrumentation probes may not be trivial. On the one hand, as

code-level instrumentation of distributed tracing is tightly related to the internal logic and implementation of functions, there is still a lack of methods for developers to automatically instrument their code with tracing probes without manual operations and expert knowledge. On the other hand, distributed tracing comes with its own cost, due to coding and testing time, as well as latency overheads in production. Additional computation cost is also induced by collection, processing, and storage of the new tracing data [EPP⁺12]. Thus, a *decision problem* arises concerning where to instrument the code and how many probes should be injected into the application.

To automatically compute a probe allocation that maximizes the chances of finding the location of faults that manifest as increased latency, while minimizing the instrumentation overhead, we propose a methodology in two phases.

First, we explore the probe allocation space to assess how instrumenting different possible locations impacts our ability to localize injected performance faults. Because the space of possible probe allocations is typically too large to be exhaustively explored, we introduce a novel heuristic that exploits the control flow graph of the application to iteratively decide where to explore next in order to gather the most information without exceeding a maximum budget of computational resources allowed for the exploration. For each explored probe placement, we run systematic performance tests to collect execution traces and use novelty-based anomaly detection methods to attempt to locate the injected fault from the collected traces.

Second, we formulate an optimization problem to decide which probes to retain in production in order to maximize the probability of locating a fault with a (near-)minimal set of probes, and therefore minimal instrumentation latency overhead and costs. We use the traces collected during the previous phase to infer the ability of a probe placement choice to locate a fault. Especially, we devise a tailored branch and bound search strategy that exploits the structure of the control flow graph and specialized meta-heuristics to efficiently sift through the otherwise intractably large search space.

In preliminary experiments, we abstract the anomaly behaviors inside functions with single and multiple delays to simulate the internal faults that impact the performance of serverless

functions. The experimental results show good accuracy on the different sensitivity levels of internal faults with over 90% recall score for up to 15% latency anomalies, and the enhanced search method achieves better efficiency in locating the anomalies with minimal instrumentation overhead.

In summary, the main contributions are:

- A CFG-based encoding of tracing probe placement at code level for fine-grained observations.
- Heuristics-based strategies to initialize the probe locations for tracing data collection.
- A global search with the branch-and-bound algorithm to elicit the best tracing probe placement for anomaly diagnosis, enhanced with search and bounding strategies to achieve fast convergence.

The rest of this paper is structured as follows. Section 5.2 presents the state-of-the-art of this paper. Section 5.3.1 provides an overview of the proposed method and Section 5.3.2 presents the code-driven probe placement encoding and heuristics-based trace data collection. Section 5.3.3 demonstrates the novelty detection-based models for anomaly identification. In Section 5.3.4, we illustrate an enhanced branch-and-bound search method. Section 5.4 evaluates the proposed methodology with serverless tracing data. Section 5.5 discusses the threats to the validity of this work. Section 5.6 draws the conclusion of this chapter.

5.2 Related work

From the perspective of serverless developers, how to automatically achieve fine-grained troubleshooting for internal faults of serverless functions still lacks investigation in the state of the arts.

Customized tracing and profiling enable developers and users to monitor the system at their preferred level and focus on areas of interest. However, instrumentation overhead consistently

presents a problem, impacting both the end-user experience and the efficiency of the programs. In [AST⁺19], the authors state that it is infeasible to enable all possible instrumentation all the time for a function due to the unacceptable overhead. Several works aiming at reducing the instrumentation overheads, for example, In [BL92], Ball *et al.* propose a set of optimal algorithms enhancing efficiency by reducing counter numbers and trace file sizes for program profiling and instruction tracing, to reduce the overhead of profilers. The proposed approaches improve efficiency in profiling and tracing within individual programs by optimizing the placement of counters and tracing instrumentation, which is particularly beneficial for analyzing performance and identifying bottlenecks within single applications or specific components of a system. However, serverless applications may require more powerful techniques that can track and monitor the performance of transactions or requests as they traverse through various services and components, providing a holistic view of the system's performance.

Troubleshooting and faults localization are more challenging for serverless functions due to the limitation of accessibility and control of the infrastructures for developers [SKM22]. Manner *et al.* propose to use log data for fault detection and resolution of serverless functions via semi-automated monitoring and debugging [MKW19]. However, logs and alert messages are difficult to customize and analyze at the code level to track the procedure executions within and across functions. In [LKW⁺18], GammaRay is introduced to capture the causal dependencies for debugging and optimization of serverless applications on AWS Lambda. The advantage of this proposed methodology is that it requires no intervention from the developers to record the performance data. Emre *et al.* present an automated instrumentation framework based on calling context trees for workflow-centric tracing, aiming at the data of the requests processed within and among workflows with low overhead in [AST⁺19]. In [BWK21], the authors evaluate several distributed tracing approaches and present a developer-driven tracing method. They focus on the integration of distributed tracing in serverless applications. This work improves observability for both developers and platform providers. While most of the existing works on serverless troubleshooting cover function-level observations, there is still a research gap in lacking fine-grained observations inside serverless functions for developers to pinpoint the locations of the faults.

To obtain the required visibility of executions, customized tracing and monitoring can be applied to collect and correlate performance metrics and trace data with instrumentation [PSM⁺20]. However, the quality and frequency of instrumentation are critical to the resulting observation. There are studies on the automatic instrumentation of tracing at different levels. In [EPP⁺12], the proposed method provides dynamic instrumentation of tracing within machines and across clusters. In [PO19], Popa and Oprescu introduce data flow analysis into the codebase to obtain the instrumentation points for data-centric distributed tracing. Although these studies address the issues of instrumentation with only expert knowledge, there are limited works to our knowledge that provide code-driven instructions for developers for fine-grained instrumentation of spans for serverless functions, especially on code-level troubleshooting.

5.3 Methodology

5.3.1 Overview

The ultimate goal of our method is to decide the locations where instrumentation probes should be injected in the production code to minimize the overhead and cost of monitoring while achieving high fault localization capabilities.

To achieve this goal automatically and efficiently, without relying on human expertise, we propose a systematic experimentation and optimization process involving three parts: 1) construction of an extensive model of the performance profile of the application in normal operating conditions, i.e., assuming no faults, 2) inject faults and efficiently collect monitoring traces that can help ascertain the fault localization capability of different instrumentation locations, and 3) optimize the set of instrumentation probes to embed in the production code to minimize the overhead for the users while retaining as much fault localization ability as possible. Figure 5.1 gives an overview of our proposed methodology.

Code abstraction and normal operation. A key insight underlying our methodology is the abstraction of the functions composing the application into their CFG [NNH15]. We assume

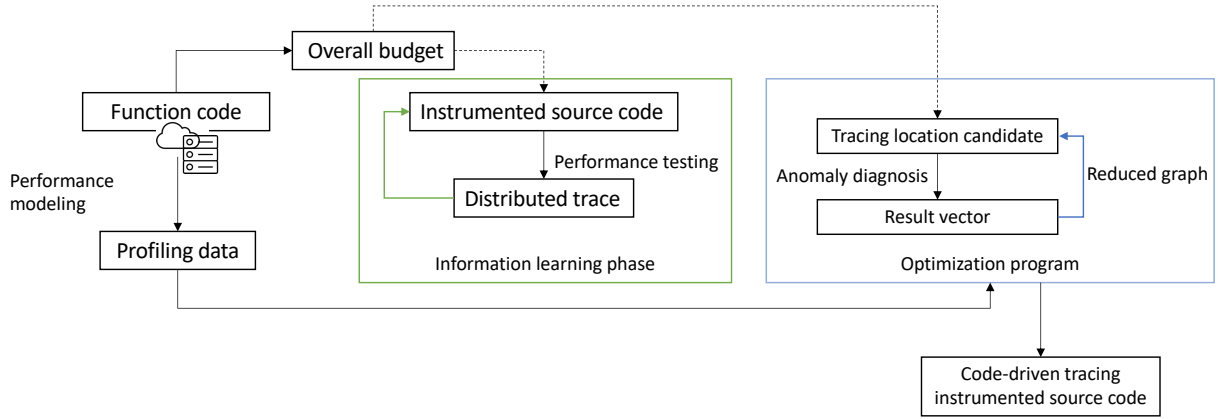


Figure 5.1: Overview of the proposed methodology

that the code of (most) serverless functions is accessible either from the cloud or from a private server and preliminarily compute its aggregated CFG that captures all the possible executions of the application. While a CFG is usually an overapproximation of the possible behaviors (i.e., it may include also execution paths that may never actually executed with the inputs received by the application), it is efficient to compute statically from the source code and provides a compact structure to reason about the possible execution paths within the application. In particular, as we will elaborate in the next section, the branches in the CFG partition all the possible executions in segments, such that each execution path is uniquely identified by the sequence of segments it exercises.

To complete a reference model of the normal operation of the application, to be later used for anomaly detection, we perform an initial systematic performance testing. We profile the current version of the application to collect trace data without faults to estimate the execution time of code blocks on different paths exercised by a reference workload. Typically, intense performance testing for profiling purposes should be conducted on a shadow production deployment – i.e., resembling the execution resources of the production environment to collect realistic performance metrics while not affecting the quality of service experienced by external customers.

Code-driven tracing data learning. This phase (green box in Figure 5.1) consists in injecting faults in the application and learning the fault localization power of different instrumentation choices based on the collected execution traces.

To collect such traces, we need to decide how to instrument the code. This instrumentation has different goals than the final instrumentation of the production code our approach will produce as output: we aim at collecting the most informative tracing information that will later help us decide the minimal set of probes to retain in production. Therefore, we will not try to minimize the number of probes, but instead aim at collecting traces useful for discriminating at different levels of granularity the location of the injected faults.

Because each instrumentation and injected fault requires repeating the performance tests, it is usually intractable to exhaustively try all the possible instrumentation options. Additionally, as we shall demonstrate throughout the paper, a relatively small number of locations usually provide enough data to detect an anomaly and localize the corresponding fault within reasonably small portions of code.

We instead propose in Section 5.3.2 an iterative exploration of possible probe locations to evaluate based on a heuristic search guided by the CFG structure. To prevent an excessive use of computational resources and time for the testing, we require the user to set an overall budget constraint M that represents the cost (e.g., in dollars) of the total resources available for this phase. These resources normally include the additional CPU time directly used by the instrumentation probes and tracing operations, and the cost for generating the test workload.

In abstract terms, the problem we aim to solve can be summarized in the following optimization program

$$\begin{aligned}
 \max_{n,k} \quad & \log n + \log k \\
 \text{s.t.} \quad & k u_{app} + n u_{probe} \leq M \\
 & n, k \geq 1
 \end{aligned} \tag{5.1}$$

where n is the number of probes injected into the code and k is the number of executions of the baseline application. u_{app} is the cost of a run of the un-instrumented application, including the cost of generating the test requests, and u_{probe} is the average cost of a probe. The objective function is concave and equivalent to maximizing the product nk .

The optimization program (5.1) is just a conceptual formalization of the goal of this phase. In practice, the overall budget is constrained across multiple iterations as will be discussed in the next section. As mentioned above, in this phase, we do not aim to minimize the number of probes. On the contrary, we want to use the entire budget – without exceeding it – to gather as many traces as possible, while deciding the instrumentation to test at each iteration based on the information we collected before and a search heuristic guided by the CFG.

Optimal probe placement. The last phase of our method (blue square in Figure 5.1) uses the set of traces $\mathbf{d}$ collected in the learning phase and an indication of the maximum number of probes n used to collect them to produce the final probe placement to be retained in production.

Contrary to the learning phase, whose main goal was exploration and information gathering, this phase aim at minimizing the set of probes in the final instrumentation while preserving the most anomaly detection and fault localization capabilities possible. This optimization problem relies only on the set of traces $\mathbf{d}$ and does not require additional performance tests. We formulate it in Equation (5.2):

$$f(n, \mathbf{S}, \mathbf{d}) = \text{score}(\mathbf{S}, \mathbf{d}) - \text{cost}(n, \mathbf{S}) \quad (5.2)$$

where $\mathbf{S} = (S_1, \dots, S_m)$ denotes the space of possible location combinations to inject tracing probes, where each S_i represents a set of code locations to instrument and $i \neq j \Rightarrow S_i \neq S_j$. The detailed encoding of S_i is discussed in Section 5.3.2. $\mathbf{d}$ is the dataset of traces collected from the previous phase and n the maximum number of probes used during learning, which is used to limit the maximum number of probes enabled in each S_i . The identification *score* and *cost* functions relate to the expected fault localization capability and overhead of the final instrumentation and will be discussed in Section 5.3.3.

5.3.2 Code-driven tracing data learning

In this section we present our CFG-based heuristic for the systematic exploration of the probe placement space to collect traces from which we will estimate the fault localization power of

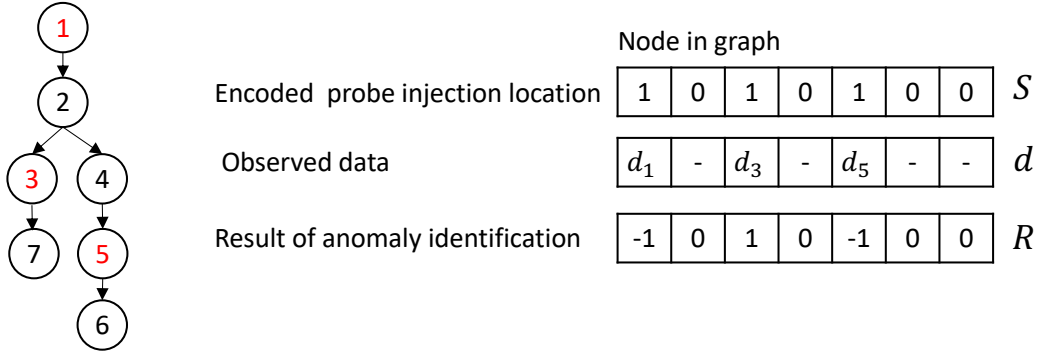


Figure 5.2: Example of encoding of trace probe location

different probe locations. To inform the decision problem of where to instrument the code, we first extract the structure of functions as a control flow graph (CFG) via static code analysis. A CFG can be represented with a graph notation, $CFG = (\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ and $\mathbf{E}$ denote the nodes and edges, respectively. Nodes in the CFG represent possible locations in the code where probes can be placed, while edges represent partial orderings of the locations executed during a run corresponding to branch instructions in the code.

To align distributed tracing data with the CFG structure, we assume that when a probe is placed at a location l within a function, the corresponding span starts when the location is executed and ends when the function returns. Such spans are nested under the root span that monitors that function.

Code-level tracing probe encoding. Figure 5.2 shows an example encoding of probes locations from a CFG. The left-hand side represents an example control flow, where node 1 indicates the entrypoint of the function, and nodes 6 and 7 are possible return points. After node 2, the execution may branch towards either 3 or 4, depending on the inputs. We highlight in red the nodes in the example CFG where probes are placed. The top vector on the right-hand side shows a binary encoding of the instrumented location, where $S_i = (1, 0, 1, 0, 1, 0, 0)$ indicates that probes are placed at locations 1, 3, and 5. The observed data d_1 , d_3 , and d_5 represent the execution time for the spans starting with the probes at the corresponding locations and ending with the return of the function.

Anomaly identification in this work is based on a novelty detection method that tests the

observed data against the expected values under normal operating conditions and produces a vector R that assigns -1 to the data classified as anomaly and 1 as data matching the expectation. 0 represents infeasible classification for execution segments not instrumented for the current set of observations. We will detail our novelty detection method in Section 5.3.3.

Heuristics-based data collection. The set of traces gathered during each iteration of performance testing depends on the probes used to instrument the code for that iteration. Because the number of possible locations may be intractably large and the process may be repeated for multiple fault injection, it is usually intractable to instrument every location of the code within a reasonable overall computational budget M .

We instead exploit the structure of the CFG to devise a heuristic probe placement to narrow down as much as possible the possible locations of an injected fault. We achieve this by injecting probes at the successors of each branch node. When a probe is injected at the location corresponding to a node of the CFG, we assume its monitoring span to extend until the function returns, i.e., it measures the execution time from the node to the first return node encountered during the execution.

We define $\sigma = \{\sigma_1, \sigma_2, \dots\}$ as a set of all CFG nodes that are successors of a branch. For example, in Figure 5.3, the blue nodes of the CFG on the left are included into σ since they are the successors of branch nodes – namely, of 2 and 4.

Intuitively, each branch splits the execution paths that reach it in two disjoint subsets. We can thus use them as natural classification predicates to identify the subgraph of a CFG where a fault is located. However, not all splits carry the same localization power. Consider the CFG on the left-hand side of Figure 5.3. If a fault has been injected in the code under node 3 (successor of 2), placing a probe on node 3 will very specifically localize the fault. If a fault is instead placed, e.g., under node 5, instrumenting node 4 will provide less specific information about the location of the fault (which could be in any node ≥ 4). In other words, the fewer further execution splits of the execution are possible after a branch successor, the more specific the localization information obtainable instrumenting that successor are.



Figure 5.3: Example of CFG and corresponding dominator tree

We can formalize this intuition by constructing the dominator tree of the CFG. A dominator tree is defined as a directed graph structure where each node immediately dominates its successors [BDP15]; a node n_i dominates a node n_j if every path from the entrypoint of the CFG to n_j must go through n_i . Given a CFG, the corresponding dominator tree $dTree$ can be efficiently constructed using the Lengauer-Tarjan algorithm [LT79]. We can then sort the nodes in σ in ascending order based on the maximum distance from a leaf in $dTree$. For example, node 3 and node 5 have distance zero, while node 4 has distance two, as expected from the discussion above.

Putting it all together, let us assume a fraction $0 < p < 1$ of the overall learning budget M is allocated for the current iteration, which allows placing a maximum of n probes, then:

1. if $|\sigma| = n$, we instrument all the locations of nodes σ
2. if $|\sigma| > n$, we instrument the locations of the first n nodes in σ , after sorting them according to their maximum distance from a leaf in the dominator tree of the CFG, as discussed above
3. $|\sigma| < n$, we instrument all the locations of nodes in σ and then select $n - |\sigma|$ additional

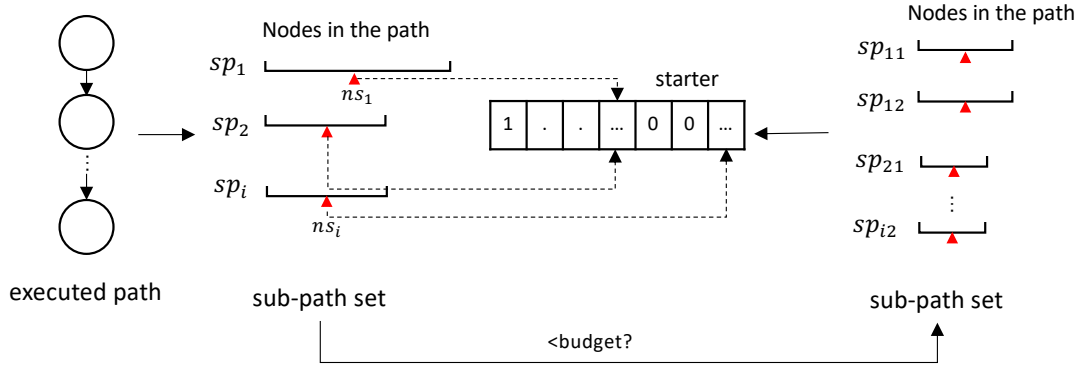


Figure 5.4: Bisection sampling on feasible paths

location to instrument based on two secondary heuristics described below.

Residual budget allocation. If $|\sigma| < n$, the budget constraints allows to instrument an additional set of $r = n - |\sigma|$. Recall that our goal in this phase is to gather as much information as possible, provided we do not exceed the budget. We thus consider selecting r additional nodes either 1) randomly from the CFG or 2) using a bisection strategy on a sample of collected paths that revealed an anomaly.

The bisection strategy works as follows. Assume in a previous iteration we detected an anomaly along an execution path ω through the nodes of the CFG. We can partition ω into sub-paths $sp = (sp_1, sp_2, \dots)$ based on the branch successors that were instrumented, i.e., each sp_i is a sequence of non-instrumented nodes between elements of $\sigma \cup \top$, where $\top$ represent return statements of the function. Then, we can iteratively find the middle of each segment sp_i and add an instrumentation probe at its location. This process is akin to a binary search for the location of a fault within each segment and is exemplified in Figure 5.4. We compare these two strategies experimentally in Figure 5.5, and discuss the results in the evaluation section.

Budget distribution across iterations. The conceptual model in the optimization program (5.1) does not prescribe a specific distribution of the overall learning budget M across iterations. In fact, the heuristics described in this section would apply also in case a single iteration is conducted. Nevertheless, it may be practically difficult to estimate the costs of instrumentation probes u_{probe} and of performance testing of the baseline application u_{app} . For

Algorithm 8 Heuristic trace data collection

Input: $CFG = (\mathbf{N}, \mathbf{E})$,
 M overall budget limit,
 $0 < p \leq 1$ fraction of budget per iteration,
 u_{app} and u_{probe} cost estimates

Output: $\mathbf{d} \leftarrow$ Collected traces

- 1: Initialize $S \leftarrow (1, 0, \dots, 0)$
- 2: Initialize $R \leftarrow (0, 0, \dots, 0)$
- 3: Initialize $r \leftarrow M$
- 4: Initialize $\mathbf{d} \leftarrow \emptyset$
- 5: **while** $r > 0$ **do**
- 6: $r_i \leftarrow r \cdot p$
- 7: $n \leftarrow estimatedMaxProbes(r_i, u_{app}, u_{probe})$
- 8: $S \leftarrow heuristic(\mathbf{N}, R, n)$
- 9: $\mathbf{d}_i \leftarrow$ Collect observed data with S
- 10: $\mathbf{d} \leftarrow \mathbf{d} \cup \mathbf{d}_i$
- 11: $R \leftarrow detectAnomaly(S, \mathbf{d})$
- 12: $r \leftarrow r -$ actual cost of the iteration
- 13: **end while**
- 14: **return** $\mathbf{d}$

example, the cumulative CPU time consumed by a probe statically placed at a certain code location depends on how often such location is exercised by the test workload (e.g., the number of iterations of a loop). A more conservative strategy to avoid exceeding the overall budget limit M would be to allocate a fraction $0 < p < 1$ of the available budget to the next iteration. After the iteration both the available budget can be reduced by the actual amount used and the estimates of u_{app} and u_{probe} can be refined with the new observations.

Algorithm 8 summarizes the heuristic instrumentation and trace collection process described in this section. Notice that we always include a probe at the entrypoint of a function (node 1; line 1), which will account for the total execution time, i.e., until return. The function `heuristic` at line 8 also includes the chosen secondary node selection heuristic for the situations where n_i is larger than the number of branch nodes σ in $\mathbf{N}$. The algorithm implemented by the function `detectAnomaly` at line 11 will instead be described in the next section.

5.3.3 Anomaly identification

We use a novelty detection strategy to localize the likely location of a fault leading to a performance anomaly. This strategy compares the performance metrics associated with a node in the CFG under normal operation conditions – i.e., no fault injected – with its metrics after a fault has been injected. The performance metric we focus on is the execution time, where instrumenting a node i corresponds to estimating the time between the execution of the node and the function’s return. After a fault has been injected, or it later occurs in production, its localization can be formulated as a novelty detection problem to recognize new patterns or outliers that deviate significantly from the profiling data.

Feature extraction. Let $\mathbf{P}$ be the set of traces through the CFG exercised during performance testing. Similarly to the procedure described in Section 5.3.2 for the bisection heuristics, we can see a path $\omega \in \mathbf{P}$ as the concatenation of segments $(sp_1, sp_2, \dots)$ that share at most one instrumented node. The execution time of a segment of an execution trace can be computed as difference of the execution times computed at its extremes: recall that the instrumentation of a node spans from the node to the function’s return and that the entrypoint of a function is always instrumented. Different paths may share segments.

Let i and j be the start and the end nodes of a segment sp . The dataset of traces $\mathbf{d}$ computed during the tracing data learning phase can be processed into a set of tuples $(i, j, time(i) - time(j))$. We assume that for each segment exercised during the trace learning phase the corresponding time duration under normal conditions (no fault injected) has also been estimated.

Novelty detection models. Using the performance data taken under normal condition and baseline, novelty detection can be framed as a one-class classification (OCC) [KM14] classification problem. We evaluate four different methods for this task: distance-based k-means clustering (*kmeans*) [Ste06], kernel-based one-class support vector machines (*ocsvm*) [HS13], isolation forest (*iforest*) [LTZ08] and deep learning-based autoencoders (*AE*) [N⁺11]. Each of these models is trained on the same dataset of segment performance under normal conditions and we use the following thresholds η to decide whether an estimated duration for a segment

should be considered an anomaly:

1. for the *kmeans* models, we calculate the distance between each data point to the centroid of its cluster and set the threshold as the η^{th} percentile of the distance.
2. for *ocsvm* and *iforest*, we compute the distance of each training point to the decision boundary and the average path length across isolation trees, respectively. The threshold value is defined with the η interquartile ranges of the maximum and minimum quartile of the distances and path lengths.
3. for trained *AE*-based models, training loss $\mathcal{L}$ and reconstruction error (ϵ) measured with the mean squared error are used to decide the anomaly label for test data by comparing $\epsilon > \mathcal{L} + \eta$.

Anomaly identification scores. The result of anomaly detection can be formalized a vector R whose entries assign to the corresponding nodes of the CFG the values -1, 1, or 0 if the execution time from the node to the return of the function is classified as anomaly (-1), normal (1), or the node cannot be classified due to lack of observations (0); the latter case corresponds to a non-instrumented node (or a node never exercised during performance testing, which could signal a limitation in the workload generation).

If a node is classified as anomalous the fault is likely to be located along a path originating from the node. Looking at the partial order among instrumented nodes (and the corresponding path segments) it is often possible to narrow down the location of a fault. For example, consider again the CFG and anomaly detection result R in Figure 5.2. The instrumented nodes are 1, 3, and 5. Because node 1 is anomalous but node 3 is normal, the fault is likely to be located along the path between node 1 and node 6. Because node 5 is also anomalous, the most likely location of the fault is between nodes 5 and 6.

The maximum localization resolution can be obtained only when a node is classified as anomalous and all its immediate successors are also instrumented and classified as normal. In this case, the fault is likely related to the operation performed at the code location corresponding to

the anomalous node. However, excessively rewarding such high resolution in the search process may lead to overfitting on instrumenting around the locations where faults have been injected during the trace collection phase. This may be desirable in domains where the developers can confidently restrict the possible causes of a performance anomaly (e.g., invocations of known expensive remote functions).

However, in general, one may prefer a wider coverage of the CFG, albeit with possibly lower resolution, as allowed by the limit on the instrumentation overhead. For this purpose, we classify a node as *suspicious* if it *may be* the location of a fault, as opposed to (strictly) *anomalous*, i.e., flagged as anomalous and whose successors have also been instrumented and flagged as normal. In the example of Figure 5.2, nodes 1 and 5 would be suspicious. We can then refine the `score` function in Equation (5.2) as the weighted sum:

$$\text{score}(R) = w_a \sum_i AS_a(R) + (1 - w_a) \sum_i AS_s(R) \quad (5.3)$$

where $AS_a(R)$ counts how many nodes can be identified as strictly anomalous with a candidate probe placement S and $AS_s(R)$ counts how many nodes can be identified as suspicious, given the same probe placement. The weight $0 \leq w_a \leq 1$ allow to trade-off the resolution of the detection versus coverage of the CFG. The goal of the optimization phase will thus be to infer from the dataset of traces collected in the previous phase which probe placement will maximize `score` while minimizing the instrumentation cost. We present our strategy to efficiently solve such optimization problem in the next section.

5.3.4 Enhanced search method for probe placement

Given the dataset of traces collected during the iterative trace learning phase, the optimization problem of deciding a probe placement that minimizes overhead and maximizes anomaly identification can be solved by search-based methods.

Using the anomaly detection score defined in Equation (5.3), the original optimization problem

in (5.2) can be rewritten as:

$$\operatorname{argmax}_{S \in \mathcal{S}} f(S, R) \quad (5.4)$$

$$f(S, R) = \text{score}(R) - u_{\text{probe}} \sum_i [S \neq 0] \quad (5.5)$$

i.e., we aim at finding the minimum-size set of probes that maximizes the detection score, based on the desired resolution vs coverage trade-off.

For a CFG with n nodes, the search space of possible probe placements has cardinality $|\mathcal{S}| \approx O(2^n)$. Optionally, such a space may be reduced by bounding the cost component in Equation (5.5), but it would in general remain intractable for exhaustive search. Instead, we propose in the remaining of this section a branch and bound strategy enhanced by exploiting the structural information in the CFG.

Branching exploration. The search strategy starts from the initial probe allocation which instruments only the entrypoint of the CFG. At each branching step, the unexplored children of currently instrumented nodes are considered for additional probe allocations. Without any bounding strategy, this basic exploration process would search through all the possible probe allocations in a breadth-first order through the CFG nodes, which is usually an intractably large search space.

Tabu meta-heuristic. The information in the CFG can be used to build a graph-based Tabu list [ADT16] to prevent the repetitive exploration of nodes through the search process. The core idea is to combine the information in the anomaly detection error with the dominance of nodes in the CFG. Suppose the j -th element of the anomaly detection vector R for an explored probe placement is flagged as normal ($R[j] = 1$).

All nodes dominated by node j should themselves be normal and therefore are added to the Tabu list (TL) to avoid exploring nodes that cannot contribute to increasing anomaly scores – recall that a probe in node j accounts for the execution time of an execution path from j to a return of the function, thus a delay at any point along any such paths would be observed also by the probe in j . Nodes added to TL are excluded from future probe allocation exploration

thus incrementally reducing the search space and accelerating the convergence of the process.

Bounding strategy. Besides incrementally pruning the search space with the Tabu list, we bound the search exploration by keeping track of the current local optimal scores at any explored node. At each iteration i of the search process, the possible child branches to explore are computed starting from the current probe allocation candidate S_i . The function $f(\cdot, \cdot)$ (Equation (5.5)) of each child is then compared with the score of its parent and, if lower, the child is excluded from further branching. This strategy helps to bound the exploration, focusing on children that can improve the value of the current partial allocation.

Enhanced branch-and-bound search algorithm. Algorithm 9 summarizes our branch-and-bound strategy. The initial candidate S_0 corresponds to the allocation of a probe to the entrypoint of the CFG: $S_0 = (1, 0, \dots)$ (line 5). We push this candidate to the queue ν of candidates to explore (line 6) and register it as the initial best candidate θ . The iterative search process is within the loop at line 7. The first candidate S_i is popped from the queue ν , the novelty detection algorithms are used to compute R_i and the Tabu list is updated. The value of the candidate S_i , $f(S_i, R_i)$, is computed and stored as the lower-bound LB to be used to bound the exploration of possible child branches from S_i ; if LB exceeds the current *best* known value, we update the best-known value and the corresponding candidate (line 13). Children of S_i to be explored are generated at line 15, provided the budget n is not exceeded and excluding nodes in the Tabu list TL . Each children is evaluated according to the objective function $f(\cdot, \cdot)$ and only those with value greater or equal than their parent's LB are pushed into the working queue (loop at line 16). When no more candidates are left to explore, the loop exits and the algorithm returns the best explored probes allocation θ .

Algorithm 9 Enhanced branch-and-bound search

Input: $CFG \leftarrow$ control flow graph
 $n \leftarrow$ maximum number of probes to place
 $\mathbf{d} \leftarrow$ collected observation traces dataset

Output: $\theta \leftarrow$ Tracing probe placement vector

- 1: Initialize $\nu \leftarrow \text{queue}()$
- 2: Initialize $\theta \leftarrow S_0$
- 3: Initialize $best \leftarrow -\infty$
- 4: Initialize $TL \leftarrow \emptyset$
- 5: Initialize $S_0 \leftarrow (1, 0, \dots, 0)$
- 6: $\text{push}(\nu, S_0)$
- 7: **while** ν is not $\emptyset$ **do**
- 8: $S_i \leftarrow \text{pop}(\nu)$
- 9: $R_i \leftarrow \text{detectAnomaly}(S_i, \mathbf{d})$ // novelty detection
- 10: $TL \leftarrow \text{tabu}(CFG, R_i, TL)$ // update Tabu list
- 11: $LB \leftarrow f(S_i, R_i)$ // current node value
- 12: **if** $LB \geq best$ **then**
- 13: $best \leftarrow LB, \theta \leftarrow S_i$
- 14: **end if**
- 15: $\mathbf{C} \leftarrow \text{child}(CFG, S_i, n, TL)$ // generate candidates
- 16: **for** $C_j \in \mathbf{C}$ **do**
- 17: $R_j \leftarrow \text{detectAnomaly}(C_j, \mathbf{d})$
- 18: **if** $f(C_j, R_j) \geq LB$ **then**
- 19: $\text{push}(\nu, C_j)$
- 20: **end if**
- 21: **end for**
- 22: **end while**
- 23: **return** θ

5.4 Evaluation

5.4.1 Experimental setup

Aiming to validate the idea and demonstrate its effectiveness in achieving high-fault diagnosis capabilities while minimizing the instrumentation cost, we conduct controlled experiments as follows. We deploy an OpenFaaS [ope] serverless platform on our local Kubernetes cluster, with two master nodes and ten worker nodes. Each node has an 8-core 2.10GHz CPU, 16GB memory, and runs with Ubuntu 18.04 OS. For each serverless function, we instrument the trace logic based on the OpenTelemetry framework [tel], which is a vendor-neutral open-source observability framework for instrumenting, generating, collecting, and exporting trace data. We use the OpenTelemetry Collector [ote] to collect traces of functions and persist them in the

Table 5.1: Dataset under evaluation

	Dataset	Fault injection level	Number of traces
Single fault	<i>SF30</i>	30%	2399
	<i>SF15</i>	15%	2417
	<i>SF3</i>	3%	2399
Multiple faults	<i>MF30</i>	30% for each fault	2419
	<i>MF15</i>	15% for each fault	2373
	<i>MF3</i>	3% for each fault	2463

Grafana Tempo [Tem]. We generate function workloads based on the *locust* framework [loc] from one separate node in the same cluster. We generate 5 concurrent users to send requests with images from the Oxford-IIIT Pet Dataset [PVZJ12] to the serverless function during the experiments.

To maintain control over the internal code logic, we opt to create our own serverless function for image resizing on OpenFaaS to evaluate the proposed prototype. This approach allows for easier observation and subsequent fault injection, eliminating the need to invest time in learning the functionality of the serverless function itself. Then, we inject different level of synthetic faults into the function simulating internal performance anomalies. Based on our observations, we have found that an increase in latency of over 50% can be easily detected by both statistics-based and machine-learning-based novelty detection models. Therefore, in our experiments, we consider injecting faults equivalent to [30%, 15%, 3%] of the original function mean execution times for both single and multiple fault scenarios. These injected faults are simulated by adding delays using the Python time sleep function. The summary of the dataset under evaluation is presented in Table 5.1. All traces are collected after a 5-minute warm-up phase for the serverless function.

The reconstruction of the CFG can be obtained automatically by applying static analysis of the code, as a graph representation. For example, in this paper, we focus on the functions in Python and implement the CFG extraction by traversing the abstract syntax tree based on the official Python AST library. The probes are injected according to the nodes, in particular their corresponding code line number, with a simple script.

Evaluation metrics. We focus on evaluating the effectiveness and accuracy of the probe

placement result rather than the instrumental overhead since it is negligible in the experimental analysis of the proposed methodology on the serverless function.

To evaluate the performance of the proposed method, all the training data (without internal faults) are labeled with 1, and the testing data are labeled as -1 for anomalies and 1 for normal data. The ground truth of the best probe placement locations for each dataset in Table 5.1 is calculated based on the identification index in Equation (5.5). To ensure that the identification index for comparison as positive values, we scale the score in Equation (5.5) by multiplying it with 3500.

We use the following metrics in the experimental comparison. For the performance of the tracing probe placement, we compare the number of *iterations* to obtain the solution for the efficiency and the *identification index* to show effectiveness. We also specify a *relative score* to evaluate the performance of the probe placement inference to the ground truth among 4 datasets, as

$$rs = \frac{|f' - f|}{f} \quad (5.6)$$

where f' is the identification index of the solution from search algorithms and f is computed from the best locations in the ground truth.

In the analysis of novelty detection with different sensitive levels, *recall*, which is known as sensitivity to identify positive samples (abnormal observations) is applied.

5.4.2 Experimental results

We first compare the performance of different heuristic probe location generation strategies. Next, we conduct experiments with different novelty detection models on different levels of internal faults to assess the sensitivity of anomaly identification. Finally, we analyze the experimental results for code-level probe inference with the enhanced BB algorithm under two demanding conditions: limited budget and sufficient budget.

Strategies for heuristic generation of probe locations. In Section 5.3.2, we propose

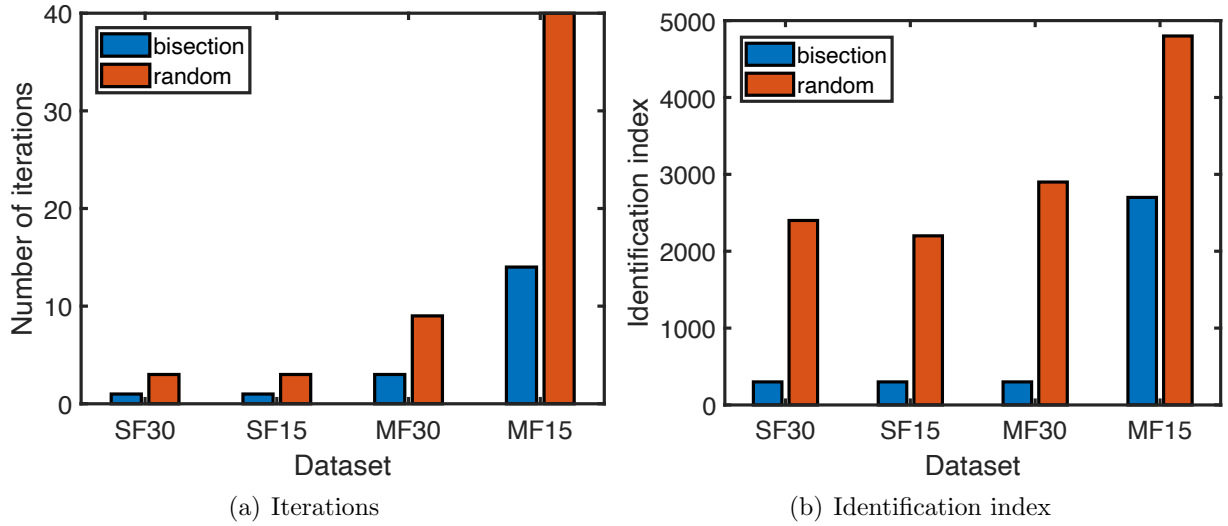


Figure 5.5: Comparison results on heuristics-based starter strategies

random selection-based and static bisection-driven strategies to generate heuristic starting locations for tracing data collection at the code level. We here compare the number of iterations of the optimization program to converge and the anomaly identification index value with different heuristics in Figure 5.5. It can be seen in Figure 5.5(a) that with static generation strategy (blue bars), the optimization program terminates with fewer iterations compared to the random selection on all the datasets under evaluation. It only takes 1 iteration to complete the program for *SF15* and *SF30* with the bisection-based generation of probe locations. This is because no more search candidates can be generated based on the current tracing locations and observations, by following the strategies in the enhanced search algorithm.

However, within the budget, static generation of the heuristic starting locations cannot reach the same identification index value as the random heuristics, as shown in Figure 5.5(b). While across different datasets, the random-selection-based method achieves higher values than bisection-based generations, indicating better tracing probe placement to locate the faults. This is because the solution from the optimization program based on different heuristic strategies not only depends on the graph structure, but also relates to the actual fault location in the functions. Static generation limits the exploration of the underlying CFG. However, random selection is more flexible by increasing the opportunities to hit the possible locations of anomalies. Intuitively, for simple CFG structures like sequential executions, it may be efficient to generate

the tracing location points for data collection with static heuristics to iteratively evaluate sub-paths. While random selection-based generation could be a better choice for functions with complex execution orders (e.g., multiple branches and loops) to involve more randomness when exploring the whole CFG.

Sensitivity analysis of novelty detection models. Synthetic faults injected into the functions can be set at different sensitivity levels to verify the effectiveness of different novelty detection models in Section 5.3.3. We conducted numerical experiments to simulate single and multiple faults on the tracing data without internal faults to control the sensitivity level. The sensitive evaluation is based on testing data where there is no limitation of the budget and all observed data is available, to show the comprehensive detection ability.

In Figure 5.6, we take both single and multiple faults with different sensitivity levels into consideration. First, it can be seen from 5.6(a) that with 30% increase of delay in the function, except *kmeans*, the other trained models achieve a high recall score close to 1, providing promising outcomes of the detection for single fault injection. While the performance of *kmeans* is also adequate with a recall value of 0.75 on *SF30*. For multiple faults in Figure 5.6(b), with 30% level, all the trained models obtain good performance on the recall score no less than 80%. Among all trained models, *AE*-based algorithms outperform with a recall value 0.99 on 30% level for both single and multiple-fault injection.

When the synthetic fault is at 15% and 3% level, the difference in the recall value among all models is obvious. In Figure 5.6(a), *AE*-based method outperforms others with a recall value 0.97 on 15% injected faults. While all four presented methods cannot ensure correct identification of the anomalies at 3% level, this appears reasonable since the injected faults are too subtle compared to the original execution times. By computing the statistics of the dataset, the standard deviation of tracing dataset without faults is greater than 3% level of faults injection, indicating that 3% increase of the function execution time may come from the system noise. Thus, in practice, abnormal behaviors cannot be captured at this level. In Figure 5.6(b), there are multiple faults existing in the functions with 3% increase of the latency, *AE*-based model slightly improves recall value compared to other methods. Consequently, the

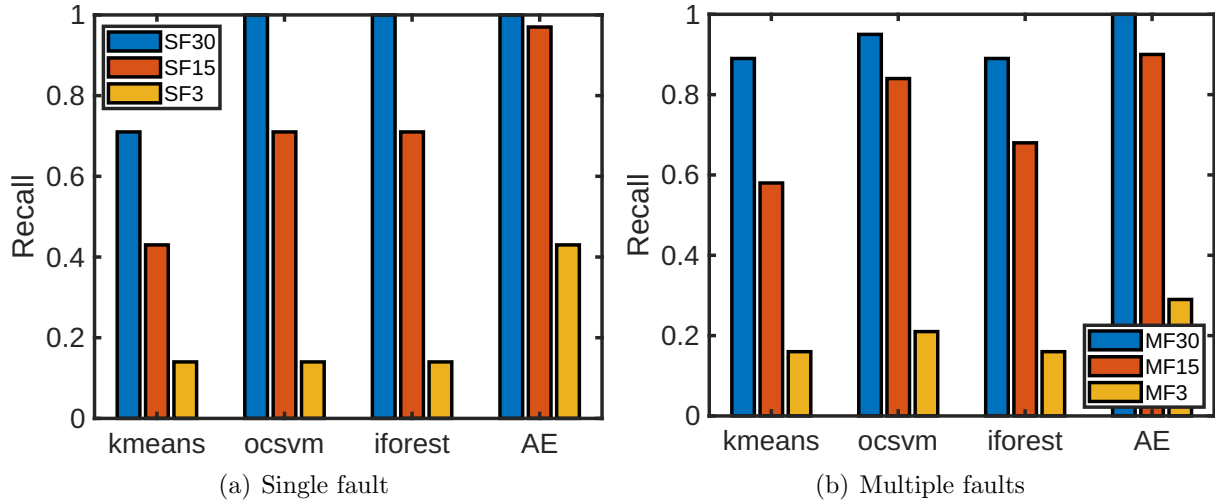


Figure 5.6: Detection results for different OCC models. AE-based models achieve the best recall score among all OCC models.

neural network-based model is more sensitive to detecting different levels of anomalies in our problem.

Budget-constrained BB result analysis. Two budget scenarios are considered to assess the enhanced BB algorithm for tracing probe location inference. To reduce the impact of randomness in the heuristic starter generation, we run 20 times the optimization programs to take the average results. In the following experiments, we apply *AE*-based novelty detection models and random generation of heuristic starting locations on data collection (in section 5.3.2) to evaluate the performance of the enhanced BB. We assume that there is an extreme constraint on the budget specified for the total cost to obtain the instrumentation location to identify the faults. Within the budget, it is possible to not locate each fault, but the outputs of the BB methods are the best locations within the budget to place the probes. We also conduct another experiment without budget limitation to allow the BB-based methods to fully explore the underlying CFG of the functions.

First, we compare the number of iterations to obtain the probe placement inference from the optimization program. We can learn from Table 5.2 that under a limited investigation budget, compared to the original BB without any searching and bounding strategies, the enhanced BB can achieve fast convergence by saving 65% of the number of iterations on average. For *SF30* and *SF15* with a single fault, the enhanced search can obtain the probe placement solution

Table 5.2: Compared results of enhanced BB algorithm with limited budget

	Iterations		Identification index (10^3)		Relative score		Probes
	Original	Enhanced	Original	Enhanced	Original	Enhanced	Both
<i>SF30</i>	10	1	3.3	3.3	1	1	2
<i>SF15</i>	14	3	3.3	3.3	1	1	2
<i>MF30</i>	20	10	2.9	2.9	0.63	0.63	3
<i>MF15</i>	30	17	2.0	2.0	0.71	0.71	4

Table 5.3: Compared results of enhanced BB algorithm without budget constraint

	Iterations		Identification index (10^3)		Relative score		Probes
	Original	Enhanced	Original	Enhanced	Original	Enhanced	Both
<i>SF30</i>	10	1	3.3	3.3	1	1	2
<i>SF15</i>	14	3	3.3	3.3	1	1	2
<i>MF30</i>	77	9	4.8	4.8	1	1	4
<i>MF15</i>	317	145	4.4	4.4	0.96	0.96	6

within 3 iterations. It is important to emphasize that the size of the underlying BB trees for different datasets is also related to the specific fault locations, except for the number of faults.

When there is no limitation of budgets, all the nodes in the CFG can be observed, although this is usually infeasible in practice. Compared to the results in Table 5.2, the iteration values remain the same for *SF30* and *SF15*, indicating that the limit budget is enough to obtain the probe placing locations based on our method. The iterations of the multiple-fault dataset grow compared to the values under the budget limit, since more observations are available when expanding the BB trees during searching. The enhanced BB can still efficiently save over 50% iterations (a minimum of 54% for *MF15* and a maximum of 88% for *MF30*) compared to the original algorithm, as shown in Table 5.3. Thus, the enhanced features with graph dependencies can accelerate the optimization process to locate the function faults.

Next, we turn to a brief discussion on the identification index and the relative score with different budgets. Among Table 5.2 and 5.3, two BB search algorithms obtain the same identification index for single-fault datasets *SF15* and *SF30* with a relative score 1. The results suggest that the inferred tracing probe locations correctly capture the internal anomalies as the best placement in the ground truth. For *MF30*, the relative score of the sufficient budget improves to 1 compared with the value in Table 5.2. Besides, the identification index of *MF15* also increases over 2 times and the relative score is upgraded to 96% by capturing all internal anomalies. It

can be inferred that without budget limitation, more observations are provided to identify the anomalies and suspicious nodes that lead to a higher identification index. Note that the final placement solutions are dependent on the performance of novelty detection models on different datasets. This can be explained by the fact that when all anomalies are detected in *MF15*, the relative score is not equal to 1 due to the inclusion of false positive labels produced by the detection models. These false positives increase the number of probes in the placement inference, which affects the comparison with the best solution in the ground truth.

By comparing the number of probes in the inferred solution to a fully distributed tracing (via instrumentation of all 12 nodes in the CFG under evaluation), the solutions from either search algorithm in Table 5.3 reduce the instrumented nodes to provide the inference for fault localization.

In summary, the overall results of BB search for anomaly diagnosis are tightly related to the locations and number of faults in the serverless functions. Multiple faults and complex execution orders in the CFG complicate the problem of tracing probe inference, thus it requires more budgets to produce the optimal solutions of probe locations for anomaly diagnosis. In comparison, the enhanced algorithm takes less time to provide the same identification results under both demanding constraints.

5.5 Threats to Validity

In this section, we identify and discuss potential threats to the validity of our methodology and results.

- Assumption of offline information gathering: Our methodology proposed is based on the assumption of the ability to conduct offline information gathering with the same configurations as the production environment. However, this may not always be feasible or applicable, potentially limiting the generalizability of the methodology to other serverless platforms.

- Synthetic fault injection: Injecting sleep functions into different locations of the source code to simulate synthetic increases in latency may introduce bias and oversimplify real latency increase by internal anomalies due to the blocking nature of sleep functions. The increased latency through the sleep function can intentionally create synthetic internal faults to assess the performance of diagnosis. It could be possible to use wait functions that allow for performing other tasks while waiting for specific operations to complete, which may provide a more realistic simulation of performance anomalies and reducing bias in the experimental results.
- Limited experimentation scope: Conducting controlled experiments with fully accessible source code may limit the scope of validation and effectiveness demonstration, potentially overlooking the performance faults originating from IO or network bottlenecks.
- Required inputs of instrumentation cost and overall budget: Assuming that instrumentation cost can be measured and specified by developers may oversimplify the realistic development and management of serverless applications, potentially leading to an inaccurate representation of the optimization process. In practice, the optimization program can be reformulated under realistic constraints that can be measured, such as resource consumption to adapt the current solution to the necessary scenarios.

5.6 Summary

In this chapter, we have presented a code-level tracing methodology for internal anomaly diagnosis of serverless functions via distributed tracing. We highlighted our contributions as follows:

- We formulated the code-level performance diagnosis with optimal probe placement for developers as an optimization problem to consider both instrumentation overheads and the anomaly detection performance.

- We exploited the code-driven encoding of tracing probe locations to detect the faults with heuristics-based data collection at a fine-grained level driven by static code analysis.
- We devised a heuristics-based trace data collection method to instrument the source code for tracing data learning.
- We trained path-based novelty detection models with the profiling data to detect anomalies inside serverless functions at code level.
- We proposed an enhanced search-based algorithm to obtain the instrumentation locations for tracing probes at the code level for anomaly diagnosis.
- The experimental results show that the instrumentation overheads can be reduced with our enhanced search methods and keep the same good accuracy to locate the anomalies.

Chapter 6

Conclusion

In this chapter, we draw our conclusion for the achievements in each chapter of this thesis, followed by possible directions for future work based on current contributions.

6.1 Summary of Thesis Achievements

This thesis focuses on the current research challenges in the DevOps context for cloud-native applications, especially from the perspective of performance engineering. We explore the methodologies of automated learning of performance models for cloud-native applications, focusing on the performance model abstraction and parameterization, automatic model generation and performance diagnosis from coarse to fine granularities. We outline below the contributions from Chapter 3 to Chapter 5 towards addressing the research objectives.

In Chapter 3, we address the performance model parameterization in **RO1** by resolving two sub-objectives. First, we propose to characterize the behaviors of an individual microservice with a closed-workload-based system as a closed queueing model with single and multiple service classes. Then, to estimate one of the most important parameters in the queueing model - service demand distribution, we focus on the departure process of the queue node in the queueing model and model the departure process with MAP based on monitoring data. To further improve the

accuracy of the service demand distribution estimation, we propose to capture the higher-moments of the distribution instead of only targeting the mean value in previous methods. We capture the service demand of the queue node with an APH distribution by fitting the parameters with global search and heuristics-based optimization. To resolve the complexity of handling multi-class service demand, we propose to utilize class aggregation to reduce the number of service classes. Our experiments in Chapter 3 show that the proposed methodology for service demand distribution estimation can achieve better performance on fitting real traces of microservices compared to only relying on first-order moments for both single and multiple service classes.

In Chapter 4, we solve the higher-degree automation required for performance model generation in **RO2**. In this work, we propose to apply an LQN model to abstract serverless workflows as follows. To bridge the research gap between fine-grained performance modeling and the information gained from monitoring serverless functions, we divide the research objectives into two sub-problems. First, for fine-grained performance modeling, we propose to apply static analysis to the accessible source code of the serverless function to extract the internal operations and relationships of a single function. This enables the model skeleton to be obtained to build the detailed LQN components. To match the accurate parameters in the LQN models, we derive the fine-grained data for service demand estimation by instrumenting the source code to obtain the code-level profiling data, which is consistent with the level of the model characterization. This helps to further enhance the model accuracy. We then present a detailed methodology for automatic LQN model generation. We illustrate the process of building LQN models using static analysis for topological structure inference, including the activity graphs and parameter feeding with the code-level profiling data. The experimental results indicate that the proposed methodology can accurately capture the inter-procedure and intra-procedure relationships of serverless workflows and produce accurate performance prediction results with a low mean error.

In Chapter 5, we focus on fine-grained performance diagnosis with code-level tracing of serverless functions (**RO3**). First, to enable different levels of observability, we propose to instrument the source code of serverless functions automatically based on static analysis. Driven by the

CFG obtained via static analysis, we can extract the possible solution space for the tracing probe localization. With fine-grained distributed tracing data, novelty detection-based models are trained with several machine learning-based algorithms. Next, to enhance the probe localization problem, we formulate an optimization problem with constrained instrumentation overheads of distributed tracing. We solve this by using search-based algorithms to decide the code-level tracing locations for performance diagnosis. Our experimental results show that the proposed methodology can help developers locate the internal faults of the serverless function with good accuracy and keep minimal instrumentation overheads.

6.2 Future Work

Based on our current contributions, there are several possible directions to extend this research. We present potential opportunities related to performance modeling and management for cloud-native applications as follows.

First, for performance model abstraction and parameterization in Chapter 3, as part of future work, it may be possible to integrate our work in an orchestration toolchain and extend it to microservices architectures and serverless function chains with LQNs. In such cases, a single component of a microservice or an individual serverless function can still be abstracted as a single queueing model, suggesting that the currently proposed estimation method can possibly capture the service demand distribution based on data from distributed tracing and monitoring. The most challenging part in this direction is to characterize the relationships between interactions with the current monitoring granularity. Possible solutions could be using model-to-model transformation to extract the dependencies between different components from software models to analytical performance models. The current estimation method can still provide a way for the model parameterization with monitoring data. This will help developers and operators in the DevOps cycle to ensure the quality of the applications in production.

For the contributions in Chapter 4, among possible future research directions, we can improve the automatic LQN model generations in the following ways. First, as we only consider static

parametric dependencies in the data flow analysis, only syntax-based dependencies of function calls are extracted, which is not sufficient for optimal CFG reduction. Thus, we can explore dynamic data flow analysis based on code-level profiling data to formulate performance-related parametric dependencies. This may be solved by data-driven techniques based on machine learning, like feature selection, to obtain the dynamic parametric dependencies for model complexity reduction. Second, we can also apply clustering techniques to workload classification to improve the accuracy of the performance modeling based on the serverless function inputs and structures. This enables refined definitions of service classes of entries in the LQN models, which can further enhance the model prediction accuracy. Third, conducting additional comparative experiments to demonstrate the improvements from coarse-grained to fine-grained modeling would be beneficial. We could continue to employ the methodology to extract the model skeleton but focus solely on reconstructing the CFG of the orchestration functions. This approach would allow us to understand the calling relationships between serverless functions and estimate the service demand for each function using aggregated profiling data. Moreover, intra- and inter-function LQN models can help relate performance bottlenecks to code artefacts, potentially helping locate the root causes of SLO violations.

Based on the works completed in Chapter 5, possible future directions may target improving the robustness of anomaly diagnosis over changes in the underlying production environments of the serverless functions. This scenario is important to the lifecycles in DevOps contexts with frequent releases. First, the offline information-gathering phase could be enhanced by several optimal profiling strategies if the source code is available for injecting hooks. Then, we consider that the tracing data can suffer data shifts due to different resource provisions, platform configurations and even new alternatives of the function development. Potentially, data-driven adaptive learning methods may contribute to the generalization and robustness of code-level anomaly diagnosis. Finally, we conducted conduct controlled experiments aiming at validating the idea and demonstrating its effectiveness in achieving high fault diagnosis capabilities while minimizing the instrumentation cost in Chapter 5. Thus, a potential plan is to experiment further on larger-scale problems with different sources of performance faults, e.g., I/O or network bottlenecks, which may require additional abstractions to complement

interaction information in the CFG graphs.

Overall, another future direction for the research discussed in this thesis involves integrating the contributions from Chapters 3, 4, and 5 to develop a comprehensive methodology for performance modeling and diagnosis. This approach may be applied to systems and applications for which only partial source code is accessible. Initially, to generate the model structure, we could utilize model transformation for black-box components and static analysis for white-box functions. Subsequently, a semi-fine-grained model could be created based on the acquired model topology and parameterized using the methodologies described in Chapters 3 and 4 for each respective case. For performance diagnosis, it would be possible to identify function-level performance anomalies and code-level faults using different available observations by applying the novelty detection models from Chapter 5. This would establish the optimization program for diagnosis.

Bibliography

- [ADT16] A. Amuthan and K. Deepa Thilak. Survey on tabu search meta-heuristic optimization. In *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPEs)*, pages 1539–1543, 2016.
- [ADW13] Jonatha Anselmi, Bernardo D’Auria, and Neil Walton. Closed queueing networks under congestion: Nonbottleneck independence and bottleneck convergence. *Mathematics of operations research*, 38(3):469–491, 2013.
- [AGEK18] Vanessa Ackermann, Johannes Grohmann, Simon Eismann, and Samuel Kounev. Black-box learning of parametric dependencies for performance models. *Sigma*, 1:10–0, 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.
- [AIB⁺20] Muhammad Abdullah, Waheed Iqbal, Josep Lluís Berral, Jorda Polo, and David Carrera. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, 15(3):1448–1460, 2020.
- [AMPJ17] Carlos M Aderaldo, Nabor C Mendonça, Claus Pahl, and Pooyan Jamshidi. Benchmark requirements for microservices architecture research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13. IEEE, 2017.

- [ANN00] Allan T Andersen, Marcel F Neuts, and Bo Friis Nielsen. Lower order moments of inter-transition times in the stationary QBD process. *Methodology and Computing in Applied Probability*, 2(4):339–357, 2000.
- [ARIM20] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 129–138. IEEE, 2020.
- [AST⁺19] Emre Ates, Lily Sturmman, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 165–170, 2019.
- [aws] <https://github.com/aws-samples/live-to-vod-workflow-using-aws-serverless-platform>.
- [aze] <https://github.com/Azure-Samples/functions-python-tensorflow-tutorial>.
- [BAV⁺17] Edwin F. Boza, Cristina L. Abad, Mónica Villavicencio, Stephany Quimba, and Juan Antonio Plaza. Reserved, on demand or serverless: Model-based simulations for cloud budget planning. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017.
- [BB19] Alexandre Brandwajn and Thomas Begin. First-come-first-served queues with multiple servers and customer classes. *Performance Evaluation*, 130:51–63, 2019.
- [BCC⁺17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.
- [BDP15] Sandrine Blazy, Delphine Demange, and David Pichardie. Validating dominator trees for a fast, verified dominance test. In *Interactive Theorem Proving: 6th Inter-*

- national Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings* 6, pages 84–99. Springer, 2015.
- [BEF⁺19] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Monica Villavicencio, Jürgen Walter, et al. How is performance addressed in devops? In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 45–50, 2019.
- [BHT05] Andrea Bobbio, András Horváth, and Miklós Telek. Matching three moments with minimal acyclic phase type distributions. *Stochastic models*, 21(2-3):303–326, 2005.
- [BKR07] Steffen Becker, Heiko Kozirolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, pages 54–65, 2007.
- [BKR09] Steffen Becker, Heiko Kozirolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.
- [BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 59–70. ACM Press, 1992.
- [BLP10] Maury Bramson, Yi Lu, and Balaji Prabhakar. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review*, 38(1):275–286, 2010.
- [BS20] Rolando Brondolin and Marco D. Santambrogio. A black-box monitoring approach to measure microservices runtime performance. *ACM Trans. Archit. Code Optim.*, 17(4), nov 2020.

- [BWK21] Maria C Borges, Sebastian Werner, and Ahmet Kilic. Faaster troubleshooting-evaluating distributed tracing approaches for serverless applications. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 83–90. IEEE, 2021.
- [BWZ15] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect’s perspective*. Addison-Wesley Professional, 2015.
- [Cas11] Giuliano Casale. Building accurate workload models using markovian arrival processes. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):357–358, 2011.
- [Cas20] Giuliano Casale. Integrated performance evaluation of extended queueing network models with line. In *Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, 2020.
- [CCT07] Giuliano Casale, Paolo Cremonesi, and Roberto Turrin. How to select significant workloads in performance models. In *CMG Conference Proceedings*, pages 58–108. Citeseer, 2007.
- [CCT08a] Giuliano Casale, Paolo Cremonesi, and Roberto Turrin. Robust workload estimation in queueing network performance models. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 183–187. IEEE, 2008.
- [CCT08b] Giuliano Casale, Paolo Cremonesi, and Roberto Turrin. Robust workload estimation in queueing network performance models. pages 183 – 187, 03 2008.
- [CDS10] Paolo Cremonesi, Kanika Dhyani, and Andrea Sansottera. Service time estimation with a refinement enhanced hybrid clustering algorithm. In *International Conference on Analytical and Stochastic Modeling Techniques and Applications*, pages 291–305. Springer, 2010.
- [CFD⁺11] Andrea Ciancone, Antonio Filieri, Mauro Luigi Drago, Raffaella Mirandola, and Vincenzo Grassi. Klapersuite: An integrated model-driven environment for reliability and performance analysis of component-based systems. In *Objects, Mod-*

- els, Components, Patterns: 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings 49*, pages 99–114. Springer, 2011.
- [CM00] Vittorio Cortellessa and Raffaella Mirandola. Deriving a queueing network based performance model from uml diagrams. In *Proceedings of the 2nd international workshop on Software and performance*, pages 58–70, 2000.
- [CSOQ21] Clément Courageux-Sudan, Anne-Cécile Orgerie, and Martin Quinson. Automated performance prediction of microservice applications using simulation. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2021.
- [CSS02] Paolo Cremonesi, Paul J Schweitzer, and Giuseppe Serazzi. A unifying framework for the approximate solution of closed multiclass queueing networks. *IEEE Transactions on Computers*, 51(12):1423–1434, 2002.
- [CZDF15] Zhijia Chen, Yuanchang Zhu, Yanqiang Di, and Shaochong Feng. Self-adaptive prediction of cloud resource demands using ensemble model and subtractive-fuzzy clustering based fuzzy neural network. *Computational intelligence and neuroscience*, 2015, 2015.
- [DMD⁺17a] M. Duggan, K. Mason, J. Duggan, E. Howley, and E. Barrett. Predicting host cpu utilization in cloud computing using recurrent neural networks. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 67–72, 2017.
- [DMD⁺17b] Martin Duggan, Karl Mason, Jim Duggan, Enda Howley, and Enda Barrett. Predicting host cpu utilization in cloud computing using recurrent neural networks. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 67–72. IEEE, 2017.
- [EBS⁺20] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. Microservices: A performance tester’s dream or nightmare? In *Proceedings*

- of the *ACM/SPEC International Conference on Performance Engineering*, pages 138–149, 2020.
- [EGVE⁺20] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 265–276, 2020.
- [EPP⁺12] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.
- [ESVE⁺20] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2020.
- [EWvK18] S. Eismann, J. Walter, J. von Kistowski, and S. Kounev. Modeling of parametric dependencies for performance prediction of component-based software systems at run-time. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 135–13509, April 2018.
- [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.
- [FF06] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.
- [FMW⁺05] Greg Franks, Peter Maly, Murray Woodside, Dorina C Petriu, Alex Hubbard, and Martin Mroz. Layered queueing network solver and simulator user manual. *Dept. of Systems and Computer Engineering, Carleton University (December 2005)*, pages 15–69, 2005.
- [GCMM01] Michele Garetto, R Lo Cigno, Michela Meo, and M Ajmone Marsan. A detailed and accurate closed queueing network model of many interacting TCP flows. In

- Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1706–1715. IEEE, 2001.
- [GCS86] Balbo Gianfranco, Bruell Steven C, and Ghanta Subbarao. The solution of homogeneous queueing networks with many job classes. *Journal of Systems and Software*, 6(1):41–53, 1986.
- [GEE⁺19] J. Grohmann, S. Eismann, S. Elflein, J. V. Kistowski, S. Kounev, and M. Mazkatli. Detecting parametric dependencies for performance models using feature selection techniques. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 309–322, Oct 2019.
- [GLHR18] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. Investigating performance metrics for scaling microservices in cloudiot-environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 157–167, 2018.
- [Gro16] Johannes Grohmann. *Reliable Resource Demand Estimation*. PhD thesis, 10 2016.
- [GSC⁺21] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Ar-ian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. Suanming: Explainable prediction of performance degradations in microservice applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 165–176, 2021.
- [HB13] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.
- [HS13] Maryamsadat Hejazi and Yashwant Prasad Singh. One-class support vector machines approach to anomaly detection. *Applied Artificial Intelligence*, 27(5):351–366, 2013.

- [HT02] András Horváth and Miklós Telek. Phfit: A general phase-type fitting tool. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 82–91. Springer, 2002.
- [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 *Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion*, pages 223–226, 2017.
- [IBC⁺20] Waheed Iqbal, Josep Lluís Berral, David Carrera, et al. Adaptive sliding windows for improved estimation of data center resource utilization. *Future Generation Computer Systems*, 104:212–224, 2020.
- [IBE⁺19] Waheed Iqbal, Josep Lluís Berral, Abdelkarim Erradi, David Carrera, et al. Adaptive prediction models for data center resources utilization estimation. *IEEE Transactions on Network and Service Management*, 16(4):1681–1693, 2019.
- [ILFW05] Tauseef A Israr, Danny H Lau, Greg Franks, and Murray Woodside. Automatic generation of layered queuing software performance models from commonly available traces. In *Proceedings of the 5th international Workshop on Software and Performance*, pages 147–158, 2005.
- [JPG19] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32, 2019.
- [KKKR08] Thomas Kappler, Heiko Koziolk, Klaus Krogmann, and Ralf Reussner. Towards automatic construction of reusable prediction models for component-based performance engineering. *Software Engineering 2008*, 2008.
- [KKR10] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Trans. Software Eng.*, 36:865–877, 11 2010.

- [KKRR11] A. Kalbasi, D. Krishnamurthy, J. Rolia, and M. Richter. Mode: Mix driven on-line resource demand estimation. In *2011 7th International Conference on Network and Service Management*, pages 1–9, Oct 2011.
- [KL20] Peng Kang and Palden Lama. Robust resource scaling of containerized microservices with probabilistic machine learning. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 122–131. IEEE, 2020.
- [KM99] K Krishna and M Narasimha Murty. Genetic k-means algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 29(3):433–439, 1999.
- [KM14] Shehroz S Khan and Michael G Madden. One-class classification: taxonomy of study and review of techniques. *The Knowledge Engineering Review*, 29(3):345–374, 2014.
- [KML15] Achyutha Krishnamoorthy, R Manikandan, and B Lakshmy. A revisit to queueing-inventory system with positive service time. *Annals of Operations Research*, 233(1):221–236, 2015.
- [KR08] Heiko Koziolk and Ralf Reussner. A model transformation from the palladio component model to layered queueing networks. In *SPEC International Performance Evaluation Workshop*, pages 58–78. Springer, 2008.
- [KSK17] Uday P Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data flow analysis: theory and practice*. CRC Press, 2017.
- [KSK19] Moonjoong Kang, Jun-Sik Shin, and JongWon Kim. Protected coordination of service mesh for container-based 3-tier service traffic. In *2019 International Conference on Information Networking (ICOIN)*, pages 427–429. IEEE, 2019.
- [lin] https://github.com/pyutils/line_profiler.

- [LK20] Changyuan Lin and Hamzeh Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):615–632, 2020.
- [LKW⁺18] Wei-Tsung Lin, Chandra Krintz, Rich Wolski, Michael Zhang, Xiaogang Cai, Tongjun Li, and Weijin Xu. Tracking causal order in aws lambda applications. In *2018 IEEE international conference on cloud engineering (IC2E)*, pages 50–60. IEEE, 2018.
- [loc] <https://locust.io/>.
- [LPX⁺22] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27:1–28, 2022.
- [LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.
- [LTZ08] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 eighth IEEE international conference on data mining*, pages 413–422. IEEE, 2008.
- [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.
- [LZSL17] Shasha Liao, Hongjie Zhang, Guansheng Shu, and Jing Li. Adaptive resource prediction in the cloud using linear stacking model. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 33–38. IEEE, 2017.
- [MADD04] Daniel A Menasce, Virgilio AF Almeida, Lawrence W Dowdy, and Larry Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

- [MBC⁺98] M Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. Modelling with generalized stochastic petri nets. *ACM SIGMETRICS performance evaluation review*, 26(2):2, 1998.
- [MBV⁺18] Preeti Malakar, Prasanna Balaprakash, Venkatram Vishwanath, Vitali Morozov, and Kalyan Kumaran. Benchmarking machine learning methods for performance modeling of scientific applications. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 33–44. IEEE, 2018.
- [MK18a] Manar Mazkatli and Anne Kozirolek. Continuous integration of performance model. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 153–158, 2018.
- [MK18b] Manar Mazkatli and Anne Kozirolek. Continuous integration of performance model. pages 153–158, 04 2018.
- [MK20a] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, 2020.
- [MK20b] Nima Mahmoudi and Hamzeh Khazaei. Temporal performance modelling of serverless computing platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 1–6, 2020.
- [MKW19] Johannes Manner, Stefan Kolb, and Guido Wirtz. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems*, 34:99–104, 2019.
- [MVL03] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM review*, 45(1):3–49, 2003.
- [N⁺11] Andrew Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

- [Neu92] Marcel F Neuts. Models based on the Markovian arrival process. *IEICE Transactions on Communications*, 75(12):1255–1265, 1992.
- [NM81] Marcel F Neuts and Kathleen S Meier. On the use of phase type distributions in reliability modelling of systems with two components. *Operations-Research-Spektrum*, 2(4):227–234, 1981.
- [NNH15] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2015.
- [onn] <https://github.com/onnx/models>.
- [ope] <https://docs.openfaas.com/>.
- [ote] <https://opentelemetry.io/docs/collector/>.
- [O0] Colm Art O’Cinneide. Characterization of phase-type distributions. *Stochastic Models*, 6(1):1–57, 1990.
- [PC13] Juan F Pérez and Giuliano Casale. Assessing sla compliance from palladio component models. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 409–416. IEEE, 2013.
- [PC17] Juan F Pérez and Giuliano Casale. Line: Evaluating software applications in unreliable environments. *IEEE Transactions on Reliability*, 66(3):837–853, 2017.
- [PCLH21] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 154–167, 2021.
- [PF02] Jian-Xin Pan and Kai-Tai Fang. Maximum likelihood estimation. In *Growth curve models and statistical diagnostics*, pages 77–158. Springer, 2002.
- [PO19] Nicolae Marian Popa and Ana Oprescu. A data-centric approach to distributed tracing. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 209–216. IEEE, 2019.

- [PPSC13] JuaHon F Pérez, Sergio Pacheco-Sanchez, and Giuliano Casale. An offline demand estimation method for multi-threaded applications. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 21–30. IEEE, 2013.
- [PS02] Dorina C Petriu and Hui Shen. Applying the uml performance profile: Graph grammar-based derivation of lqn models from uml specifications. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 159–177. Springer, 2002.
- [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.
- [PVZJ12] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3498–3505. IEEE, 2012.
- [Pys] Pyre. <https://pyre-check.org/>.
- [RL19] Joy Rahman and Palden Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210. IEEE, 2019.
- [RT88] Andrew Reibman and Kishor Trivedi. Numerical transient analysis of markov models. *Computers & Operations Research*, 15(1):19–36, 1988.
- [RV98] Jerome Rolia and Vidar Vetland. Correlating resource demand information with arm data for application services. In *Proceedings of the 1st international workshop on Software and performance*, pages 219–230, 1998.
- [SBB⁺10] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

- [SBC⁺08] Abhishek B. Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M. Voelker. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev.*, 36(2):16–25, August 2008.
- [SBM07] Roger B Sidje, Kevin Burrage, and Shev MacNamara. Inexact rouniformization method for computing transient distributions of markov chains. *SIAM Journal on Scientific Computing*, 29(6):2562–2580, 2007.
- [SBZ17] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.
- [SCBK15] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92:51–71, 2015.
- [SKM22] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, 54(11s):1–32, 2022.
- [SRSG⁺15] Arik Senderovich, Andreas Rogge-Solti, Avigdor Gal, Jan Mendling, Avishai Mandelbaum, Sarah Kadish, and Craig A Bunnell. Data-driven performance analysis of scheduled processes. In *Business Process Management: 13th International Conference, BPM 2015, Innsbruck, Austria, August 31–September 3, 2015, Proceedings 13*, pages 35–52. Springer, 2015.
- [Ste06] Douglas Steinley. K-means clustering: a half-century synthesis. *British Journal of Mathematical and Statistical Psychology*, 59(1):1–34, 2006.
- [tel] <https://opentelemetry.io/>.
- [Tem] <https://grafana.com/docs/tempo/>.

- [TMF⁺20] Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores, and Theofilos Toulkeridis. From monolithic systems to microservices: A comparative study of performance. *Applied sciences*, 10(17):5797, 2020.
- [TPF⁺09] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.
- [TPN⁺20] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 311–327. Association for Computing Machinery, 2020.
- [tut] <https://github.com/Azure-Samples/functions-python-tensorflow-tutorial>.
- [Vir15] Manish Virmani. Understanding devops & bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pages 78–82. IEEE, 2015.
- [Wan20] Runan Wang. Learning performance models automatically. In *International Conference on Service-Oriented Computing*, pages 40–46. Springer, 2020.
- [WCF21] Runan Wang, Giuliano Casale, and Antonio Filieri. Service demand distribution estimation for microservices using markovian arrival processes. In *International Conference on Quantitative Evaluation of Systems*, pages 310–328. Springer, 2021.
- [WCF22a] Runan Wang, Giuliano Casale, and Antonio Filieri. Enhancing performance modeling of serverless functions via static analysis. In *Service-Oriented Computing: 20th International Conference, ICSOC 2022, Seville, Spain, November 29–December 2, 2022, Proceedings*, pages 71–88. Springer, 2022.
- [WCF22b] Runan Wang, Giuliano Casale, and Antonio Filieri. Estimating multiclass service demand distributions using markovian arrival processes. In *ACM Transactions on Modeling and Computer Simulation*. Submitted, 2022.

- [WEG⁺18] Jürgen Walter, Simon Eismann, Johannes Grohmann, Dušan Okanovic, and Samuel Kounev. Tools for declarative performance engineering. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 53–56, 2018.
- [WHQ⁺12] Wei Wang, Xiang Huang, Xiulei Qin, Wenbo Zhang, Jun Wei, and Hua Zhong. Application-level cpu consumption estimation: Towards performance isolation of multi-tenancy web applications. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 439–446. IEEE, 2012.
- [YCC⁺21] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *WWW 2021*, page 3087–3098. ACM, 2021.
- [YHG⁺17] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465, 2017.
- [ZGJ⁺17] Zi-jian Zhang, Lin Gong, Yan Jin, Jian Xie, and Jia Hao. A quantitative approach to design alternative evaluation based on data-driven performance prediction. *Advanced Engineering Informatics*, 32:52–65, 2017.
- [ZGTC21] Lulai Zhu, Giorgos Giotis, Vasilis Tountopoulos, and Giuliano Casale. RdoF: Deployment optimization for function as a service. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 508–514. IEEE, 2021.
- [ZHZ⁺21] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: ML-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, pages 167–181, 2021.