Imperial College London
Department of Computing

Application Decomposition and Optimization for Function as a Service

Lulai Zhu

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

Statement of Originality

I declare that this thesis was composed by myself, and that the work it presents is my own, except the TOSCA models of the Thumbnail Generation and NLP Pipeline applications (see Figures 4.1 and 4.10), which were created in collaboration with our partners in the RADON project. Experiment data referenced in the thesis has been published at `https://zenodo.org/` with DOIs 10.5281/zenodo.7014842, 10.5281/zenodo.5043700, 10.5281/zenodo.3894501, 10.5281/zenodo.3894566 and 10.5281/zenodo.4159520.

Copyright Declaration

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

Under this licence, you may copy and redistribute the material in any medium or format for both commercial and non-commercial purposes. You may also create and distribute modified versions of the work. This on the condition that you credit the author.

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

Function as a service (FaaS) is a new cloud computing paradigm that realizes the idea of serverless. In the FaaS paradigm, an application is built of stateless functions whose resources are provisioned and managed by the cloud provider in place of their owner. FaaS saves operating costs, reduces management effort, enables seamless scalability and augments development productivity, compared to traditional cloud computing paradigms such as infrastructure as a service and platform as a service. Despite these benefits, three challenges often arise as obstacles to software vendors taking advantage of the FaaS paradigm and relevant technologies: migration of a monolith to the serverless architecture, cost optimization of serverless applications and performance modeling of FaaS-oriented frameworks.

To address the first challenge, we propose a semi-automatic approach, RADF, for decomposing the architecture of a monolithic application. RADF divides the entire decomposition process into two stages, namely coarse- and fine-grained decompositions. The former identifies candidate microservices through interface analysis, while the latter composes the identified microservices with serverless functions. RADF has been compared with other approaches in the literature, leading to a decomposition solution better than the available baselines.

To tackle the second challenge, a model-driven approach called RDOF is proposed for optimizing the deployment of a serverless application. RDOF predicts the performance of an application by instantiating a layered queueing network, and searches for the optimal configuration by running a genetic algorithm. We have demonstrated the applicability of RDOF on Amazon Web Services, which is now the most popular public cloud platform as to serverless computing. RDOF may also be extended to Google Cloud.

The third part of the thesis focuses on closed queueing networks (QNs) with discriminatory processor sharing (DPS), which are found useful for modeling the performance of the OpenFaaS framework. Specifically, we propose a fluid approach to transient and steady-state analysis of closed QNs consisting of delay and DPS stations. This approach is as accurate as the state of the art for steady-state analysis. A refined method for approximating response time distributions through transient analysis is further introduced. It outperforms prior work in handling non-exponentially distributed service times.

Acknowledgements

First of all, I am deeply grateful to my PhD supervisor, Dr. Giuliano Casale, for his invaluable advice, continuous encouragement and exceptional patience over the last four years. He perfectly demonstrates an example of a hard-working, critically-thinking and self-motivated academic. In particular, I would sincerely like to thank Giuliano for offering me the opportunities to participate in the DICE and RADON projects.

I would like to also express special gratitude to our partners in the DICE and RADON projects, other members of the Quality of Service research lab and anyone else who has helped me along the way. Thank you all for making my PhD studies a fascinating intellectual adventure with so much fun. I feel a great pleasure to meet you in my life and will always remember the time we spent together.

It is impossible for me to pursue a PhD degree at my favorite university, Imperial College London, without constant and unconditional support from my families. I cannot thank them more for providing me with comfortable circumstances to achieve this goal, especially my wife Zhen Liu, who devoted herself to taking care of me and our child so that I could put most of my energy into research.

This work has been partially funded by the European Union's Horizon 2020 research and innovation program under grant agreement Nos. 644869 (DICE) and 825040 (RADON).

Dedication

I dedicate this thesis to my dear son, Hongyi Zhu. You are the one who makes every day so brilliant and enjoyable to me. May your life be full of happiness, and may all your dreams come true. Growing up is hard to do, but I will always be there for you.

"In the original substance of the mind there is no distinction of good and evil. When the will becomes active, however, such distinction exists. The faculty of innate knowledge is to know good and evil. The investigation of things is to do good and remove evil."

Yangming Wang (1527)

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Motivation and Objectives 1

1.2 Contributions and Thesis Outline 4

1.3 Publications 7

2 Background **10**

2.1 Serverless Computing 10

2.1.1 General Description 10

2.1.2 Important Concepts 13

2.1.3 Function-as-a-Service Offerings 14

2.2 Machine Learning 17

2.2.1 Natural Language Processing 17

2.2.2 Cluster Discovery 23

2.3 Queueing Theory 30

2.3.1 Phase-Type Distributions 30
2.3.2 Continuous-Time Markov Chains 34
2.3.3 Queueing Networks 39
2.3.4 Layered Queueing Networks 49
2.4 RADON Modeling Profile 53
2.4.1 Basics of TOSCA Language 53
2.4.2 Extension to RADON Particles 55

3 RADF: Architecture Decomposition for Function as a Service 70

3.1 Introduction 70
3.2 Related Work 72
3.3 Running Example 74
3.4 Overview 75
3.4.1 Refactoring Strategy 76
3.4.2 Reference Workflow 77
3.5 Interface Analysis 78
3.5.1 Semantic Similarity 81
3.5.2 Structural Similarity 83
3.5.3 Aggregation 85
3.6 Microservice Identification 86
3.7 Code Refactoring 88
3.8 Tool Support 92
3.9 Evaluation 93

3.9.1 Capability for Microservice Identification 93

3.9.2 Feasibility for Code Refactoring 98

3.10 Conclusion 101

4 RDOF: Deployment Optimization for Function as a Service 102

4.1 Introduction 102

4.2 Motivation 104

4.2.1 FaaS Compute Services 104

4.2.2 Running Example 105

4.2.3 Related Work 107

4.3 Performance Prediction 108

4.3.1 Layered Queueing Network 108

4.3.2 Service Time Estimation 112

4.4 Deployment Optimization 113

4.4.1 Problem Formulation 113

4.4.2 Complexity Reduction 114

4.5 Tool Support 116

4.6 Evaluation 117

4.6.1 Accuracy of Performance Model 117

4.6.2 Utility of Optimization Approach 119

4.7 Case Study 121

4.8 Conclusion 123

5 Fluid Approximation of Closed Queueing Networks with Discriminatory Processor Sharing 125

5.1 Introduction . . . 125

5.2 Related Work . . . 129

5.3 Model Definition . . . 131

5.4 Fluid Approximation . . . 132

5.5 Application . . . 140

5.5.1 Concept of Chains . . . 141

5.5.2 Response Time Distributions . . . 141

5.5.3 System Response Time Distributions . . . 144

5.6 Validation . . . 146

5.6.1 Model Parameter Settings . . . 147

5.6.2 Tool Parameter Settings . . . 148

5.6.3 Transient Analysis . . . 149

5.6.4 Steady-State Analysis . . . 152

5.6.5 Response Time Distribution Analysis . . . 154

5.6.6 Summary . . . 157

5.7 Case Study . . . 158

5.7.1 OpenFaaS Framework . . . 158

5.7.2 Performance Model . . . 159

5.7.3 Evaluation Experiments . . . 161

5.8 Conclusion . . . 163

6 Conclusion **165**

6.1 Summary of Thesis Achievements 165

6.2 Future Work 167

Bibliography **168**

List of Tables

2.1 Comparison of AWS Lambda, Google Cloud Functions and Microsoft Azure Functions from a user perspective. 15

2.2 Comparison of OpenFaaS and OpenWhisk from a user perspective. 16

2.3 Part-of-speech tags used in the universal dependencies treebanks [NDMG+16]. . 19

2.4 The inputs of the MVA algorithms. 42

2.5 The outputs of the MVA algorithms. 42

2.6 The graphical notation of LQNs. 50

2.7 Precedences supported by LQNs. 52

2.8 Valid settings for the properties of a `Precedence`. 59

2.9 Valid settings for the clustering properties of a `WebApplication`. 62

3.1 Definitions of API operations exposed by the Cargo Shipping system (Part 1). . 79

3.2 Definitions of API operations exposed by the Cargo Shipping system (Part 2). . 80

3.3 The TF-IDF weighting scheme for computing the semantic vector of API operation i. 82

3.4 Comparison of the Jaccard index and the proposed measure on the API operations 10 and 11 of the Cargo Shipping system. 85

3.5 Pairwise distances between the API operations of the Cargo Shipping system. . 86

3.6 Notation used in Algorithms 3.1, 3.2, 3.3 and 3.4. 88

3.7 Parameter settings of clustering algorithms for coarse-grained decomposition. . . 95

3.8 F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices. 97

3.9 Preferable combinations of clustering algorithms and validity indices for coarse-grained decomposition. 98

3.10 Quality metrics for assessing the decomposition solution to the Cargo Shipping system. 99

3.11 Quality metrics of the decomposition solution obtained by applying RADF to the Cargo Shipping system. 100

3.12 Quality metrics of decomposition solutions obtained by applying the baseline approaches to the Cargo Shipping system. 100

4.1 Maximum memory and concurrency imposed by AWS Lambda, Google Cloud Functions and Microsoft Azure Functions. 104

4.2 Accuracy evaluation results for different memories but the same concurrency under the reference workload. 118

4.3 Accuracy evaluation results for the same memory but different concurrencies under the reference workload. 118

4.4 Accuracy evaluation results for different interarrival times under the optimal configuration (256 MB and 4). 119

4.5 Parameter settings of the derived TOSCA models. 120

4.6 Utility evaluation results for the derived TOSCA models with different numbers of functions. 121

4.7 The obtained optimal and feasible solutions to the derived TOSCA model with two functions and random seed 1. 121

5.1 Contextual notation used throughout Chapter 5. 128

5.2 Model parameter settings for the validations. 148

5.3 Tool parameter settings for the validations. 149

5.4 The time spent by the fluid approach in computing transient mean queue lengths for each example. 150

5.5 The average times spent by the fluid approach and the AMVA algorithm in computing steady-state mean queue lengths for the considered set of model instances. 152

5.6 The times spent by the original and the refined method in computing station response time distributions for each example. 155

5.7 The times spent by the original and the refined method in computing system response time distributions for each example. 155

5.8 Runtime options for controlling the CPU resources of a Docker container. 160

5.9 Benchmarking results of requests to the two OpenFaaS functions. 162

5.10 Evaluation results of the performance model for OpenFaaS. 163

List of Figures

1.1 An end-to-end decomposition and optimization methodology toward serverless computing. 7

2.1 The architecture of a serverless application. 11

2.2 A hyperexponential time span with k phases. 32

2.3 An Erlang or hypoexponential time span with k phases. 33

2.4 An example of an open QN. 39

2.5 An example of a closed QN. 39

2.6 A concrete example of an LQN. 50

2.7 The structure of a service template. 54

3.1 The TOSCA model of the Cargo Shipping system. 74

3.2 The refactoring strategy for decomposing a monolith into serverless functions. . 76

3.3 The reference workflow for decomposing a monolith into serverless functions. . . 78

3.4 The NLP pipeline for extracting keywords from the description of an API operation. 81

3.5 A scenario that cannot be properly dealt with by either static or dynamic strategy. 84

3.6 API operations of the Cargo Shipping system in a two-dimensional representation. 86

3.7 Caliński-Harabasz, Dunn, gamma, Davies-Bouldin and silhouette indices of partitions resulting from hierarchical clustering on the API operations of the Cargo Shipping system. 87

3.8 The architecture decomposition feature of the RADON decomposition tool. . . . 93

3.9 Average times spent in searching for the coarse-grained decomposition solutions under different combinations of clustering algorithms and validity indices. 96

3.10 Average F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices. 97

4.1 The TOSCA model of the Thumbnail Generation application. 106

4.2 The mean response time of the `create_thumbnail` function versus the total operating cost of the Thumbnail Generation application. 107

4.3 The LQN structure for modeling an S3 bucket. 110

4.4 The LQN structure for modeling a Lambda function. 110

4.5 The LQN structure for modeling a group of remote interactions. 111

4.6 The entry for modeling a synchronous interaction. 111

4.7 The entry for modeling an asynchronous interaction. 111

4.8 The implementation of the deployment optimization feature in the RADON decomposition tool. 117

4.9 The topology of the derived TOSCA models. 119

4.10 The TOSCA model of the NLP Pipeline application. 122

5.1 The serial topology. 147

5.2 The parallel topology. 147

5.3 The error in using the fluid approach to approximate transient mean queue lengths for each example. 151

5.4 The approximated and simulated transient mean queue lengths at the first DPS station for each example. 151

5.5 The average errors in using the fluid approach and the AMVA algorithm to approximate steady-state mean queue lengths for the considered set of model instances. 153

5.6 The errors in using the original and the refined method to approximate station response time distributions for each example. 156

5.7 The errors in using the original and the refined method to approximate system response time distributions for each example. 156

5.8 The approximated and simulated response time CDFs of class-1 jobs at the first DPS station for serial examples. 157

5.9 The approximated and simulated response time CDFs of chain-1 jobs with respect to the delay station for parallel examples. 157

5.10 Interactions within the API gateway and an OpenFaaS function during a synchronous invocation. 159

5.11 The closed QN for modeling the performance of OpenFaaS. 160

Chapter 1

Introduction

1.1 Motivation and Objectives

The last decade has witnessed the spread of cloud computing in companies of all sizes and at a great scale [LAK17]. Public cloud platforms offer their services based on different cloud computing paradigms. *Infrastructure as a service* (IaaS) and *platform as a service* (PaaS) are two standard paradigms that software designers may consider when looking for a hosting solution to their cloud applications [BBG10].

IaaS services (e.g. Amazon EC2[1]) allow subscribers to purchase base cloud infrastructure, virtual machines, block storage and virtual private cloud among others, to run their applications [DWC10]. An IaaS user takes almost full control of the entire technology stack, from the kernel of the operating system upward. This gives high flexibility but meanwhile leaves the burden of complex operations, for example fault tolerance, which could exceed the development effort for the product [JSSS+19].

PaaS services (e.g. Google App Engine[2]) instead present an integrated environment for subscribers to develop and deploy applications with built-in runtime and middleware on provider-managed cloud infrastructure [DWC10]. Although a PaaS user no longer has to worry about

[1] `https://aws.amazon.com/ec2/`
[2] `https://cloud.google.com/appengine/`

operational tasks, other concerns are raised in company with the PaaS paradigm. These involve limited flexibility, technology lock-in and vendor dependence, making PaaS a less preferable choice in the market [JSSS+19].

To overcome the drawbacks of the IaaS and PaaS paradigms, a new option called *function as a service* (FaaS) has been introduced by Amazon, alongside the release of the first FaaS compute service, AWS Lambda[3], in 2015. The sudden popularization of AWS Lambda then prompts the emergence of similar services (e.g. Google Cloud Functions[4] and Microsoft Azure Functions[5]) from other cloud providers and FaaS-oriented frameworks (e.g. OpenFaaS[6] and OpenWhisk[7]) from open-source contributors. The latter are particularly useful for the adoption of FaaS on private clouds.

In the FaaS paradigm, an application is composed of stateless functions registered on the cloud and executed in response to predefined triggers, and the cloud provider provisions and manages resources required to run the functions on behalf of its customer [BCC+17]. FaaS and PaaS are both considered as a realization of *serverless computing* [FIMS17]. However, FaaS compute services improves upon early PaaS offerings in the senses of the pay-as-you-go cost model, seamless scalability and higher development productivity [CIMS19, CAvdH+20]. Since FaaS is a more successful paradigm than PaaS, the term "serverless computing" primarily refers to FaaS alone or the combination of FaaS with *backend as a service* (BaaS) [RC17, JSSS+19]. In contrast to IaaS services, FaaS offerings substantially relieve the burden of complex operations by avoiding user involvement in resource provisioning and management.

The serverless community survey 2020[8] reveals that 75% participants had used serverless technologies thus far and that 45% of the rest were planning to do so in the next twelve months. A swift paradigm evolution in cloud computing toward FaaS is now ongoing. Nevertheless, many open challenges still need to be resolved in the context of DevOps for serverless computing, including deployment automation, testing and monitoring, benchmarking, cost optimization,

[3] https://aws.amazon.com/lambda/
[4] https://cloud.google.com/functions/
[5] https://azure.microsoft.com/en-us/services/functions/
[6] https://www.openfaas.com/
[7] https://openwhisk.apache.org/
[8] https://github.com/jeremydaly/serverless-community-survey-2020

migration as well as continuous integration/continuous delivery [YBLW19]. The goal of this thesis is to come up with effective approaches to three of these challenges that currently prevent software vendors from fully exploiting the FaaS paradigm and relevant technologies:

Migration of a Monolith to the Serverless Architecture

The monolithic architecture used to be a prevailing architectural pattern for software design, resulting in a large number of existent applications being implemented as monoliths whose code is difficult to understand and maintain, may only be scaled at the application level and is not suitable for development teams to work independently [New15, Ric18]. These shortcomings become increasingly significant as the size of the application grows. The serverless architecture naturally solves them by defining an application as a combination of stateless functions complemented with BaaS components such as object storages, databases and message queues. Unfortunately, because of the intrinsic incompatibility between monoliths and the FaaS paradigm, software vendors cannot apply serverless technologies to legacy products that were designed in the monolithic architecture. This calls for an approach to migration of a monolith to the serverless architecture.

Cost Optimization of Serverless Applications

The FaaS paradigm enforces complete decoupling of computation from storage and strong isolation at the function level, which enables the cloud provider to scale each function separately according to its own workload. This fine-grained autoscaling mechanism leads to a pay-as-you-go cost model, where payment is made in proportion to resources consumed instead of resources allocated. Particularly, no operating costs are incurred in the absence of function invocations. The pay-as-you-go cost model makes FaaS an economical paradigm even for bursty workloads. As pointed in [ESVE+21], saving operating costs (47%) is the main driver for practitioners to choose serverless computing, followed by reducing management effort (34%) and improving scalability (34%). A FaaS user however can hardly minimize the total operating cost of an application under the specified performance requirements with an arbitrary function configuration, which calls for an approach to cost optimization of serverless applications.

Performance Modeling of FaaS-Oriented Frameworks

It is also possible to incorporate serverless computing into private clouds by installing a FaaS-oriented framework with a container orchestrator, for example Kubernetes[9] or Docker.[10] As opposed to public clouds, which have presumably infinite resources, private clouds are built on physical infrastructure like a data center owned or rented by users themselves. Capacity planning is therefore a necessary task that can largely affect the costs for private cloud users to deploy and operate a new application. A common method for ensuring the capacity of a software system is to devise a performance model for the system under study, create a benchmark of the system on a standard processing unit and perform a what-if analysis to determine how many processing units are needed under the reference workload [MA98]. In the case of serverless adoption on private clouds, this calls for an approach to performance modeling of FaaS-oriented frameworks.

1.2 Contributions and Thesis Outline

This thesis aims to tackle three open challenges, i.e. migration of a monolith to the serverless architecture, cost optimization of serverless applications and performance modeling of FaaS-oriented frameworks, helping software vendors to unlock the benefits of the FaaS paradigm and relevant technologies. The primary contributions of the thesis are threefold:

- An architecture decomposition approach for refactoring a monolithic application into a FaaS-based one through *similarity analysis* and *cluster discovery* [JD88] of operations exposed by the interface of the application.
- A deployment optimization approach for finding the optimal deployment scheme of a serverless application using a *layered queueing network* [WNPM95, RS08] for performance prediction and a *genetic algorithm* [Gol89] for cost optimization.
- A fluid approximation approach to transient and steady-state analysis of *closed queueing networks* [BCMP75] with *discriminatory processor sharing* [Kle67], which can be applied to modeling the performance of the OpenFaaS framework.

[9] `https://kubernetes.io/`
[10] `https://www.docker.com/`

The first two approaches have been implemented as the features of the RADON decomposition tool,[11] part of the DevOps toolchain offered by the RADON project.[12] The third approach has been supported by the fluid solver of the LINE engine,[13] a MATLAB toolbox developed by our research group[14] for performance analysis.

The remainder of the thesis is structured as follows. Chapter 2 describes the overall background, including serverless computing, machine learning, queueing theory and the RADON modeling profile. The details of the contributions are discussed across Chapters 3, 4 and 5. At the end, Chapter 6 sums up the achievements and looks ahead to future work. We present here a preview of the main chapters and connections between them:

Chapter 3. RADF: Architecture Decomposition for Function as a Service

In this chapter, a semi-automatic approach called RADF[15] is proposed for decomposing the architecture of a monolithic application. RADF adopts a two-stage refactoring strategy, where the entire process is divided into the coarse- and fine-grained decomposition stages. As such, a monolithic application is refactored first into a microservices-based one and then into a FaaS-based one. We identify candidate microservices by clustering the API operations according to their semantic and structural similarities. A number of serverless functions are created for implementing API operations assigned to each identified microservice. We have evaluated RADF in terms of its capability for microservice identification and feasibility for code refactoring. In the evaluation experiments, RADF obtains a decomposition solution with lower coupling and relatively balanced cohesion compared to baselines arising from previous approaches.

Chapter 4. RDOF: Deployment Optimization for Function as a Service

We propose a model-driven approach, RDOF,[16] for optimizing the deployment of a serverless application in this chapter. RDOF predicts the performance of an application consisting of serverless functions by instantiating a layered queueing network (LQN), and searches for the optimal configuration of each function by running a generic algorithm (GA). The resultant

[11] `https://decomposition-tool.readthedocs.io/en/latest/`
[12] `https://radon-h2020.eu/`
[13] `http://line-solver.sourceforge.net/`
[14] `http://qore.doc.ic.ac.uk/`
[15] RADF stands for RADON architecture decomposition for FaaS.
[16] RDOF stands for RADON deployment optimization for FaaS.

deployment scheme can minimize the total operating cost of the application while satisfying the specified performance requirements under the reference workload. We have evaluated RDOF with respect to the accuracy of the performance model and the utility of the optimization algorithm on Amazon Web Services (AWS). The evaluation results indicate that the relative error of the LQN for performance prediction is less than 10% and the saving rate of the GA for cost optimization is greater than 10%. Notably, RDOF is extensible to Google Cloud.

Chapter 5. Fluid Approximation of Closed Queueing Networks with Discriminatory Processor Sharing

As a multi-class variant of the classic egalitarian processor-sharing (EPS) discipline, discriminatory processor sharing (DPS) provides a suitable way to model systems where share exists to control the service access of heterogeneous jobs. Although DPS is a more fine-grained scheduling discipline than EPS, the behavior of closed queueing networks (QNs) with DPS remains poorly understood. We propose in this chapter a fluid approach to transient and steady-state analysis of closed QNs comprising delay and DPS stations. Our reference model features an arbitrary routing topology, phase-type service time distributions and class switching. The proposed approach has been validated against simulation for both transient and steady-state analysis. A refined method for approximating response time distributions at either station or system level through transient analysis is also introduced, and has been demonstrated to improve upon prior art in the case of non-exponentially distributed service times. Additionally, we devise a performance model for the OpenFaaS framework on a Docker swarm. This model exhibits a relative error of no more than 10% or even lower if solved by the fluid approach.

End-to-End Decomposition and Optimization Methodology toward Serverless Computing

The three approaches proposed in Chapters 3, 4 and 5, i.e. RADF, RDOF and fluid approximation, jointly form an end-to-end decomposition and optimization methodology toward serverless computing, as illustrated in Figure 1.1. Suppose that the application under consideration is originally designed as a monolith. At first, the architecture of the application is decomposed with RADF, which yields a decomposition solution compatible with the FaaS paradigm. A prototype of the application is then created by instantiating the decomposition solution on

either a public or a private cloud according to the hosting decision. If the target cloud is public, RDOF is used to optimize the deployment of the application so that the total operating cost is minimized under the specified performance requirements. If the target cloud is private, capacity planning is carried out to ensure adequate resources for the application, and fluid approximation serves as the technique to analyze the performance model. A serverless version of the application ready for production is thus accomplished.

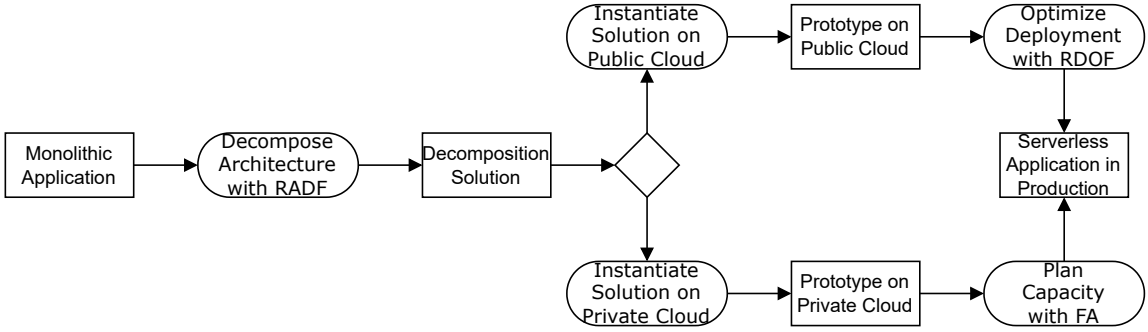


Figure 1.1: An end-to-end decomposition and optimization methodology toward serverless computing.

1.3 Publications

I have co-authored the following papers during the PhD program, and my individual contributions to each of them are summarized below:

[ZTC22] L. Zhu, D. A. Tamburri and G. Casale. **RADF: Architecture Decomposition for Function as a Service**. Manuscript in preparation for publication, 2022.
This paper proposes a semi-automatic approach, RADF, for refactoring a monolithic application into FaaS-based one. I am the primary contributor of the paper, who reviewed the related work, devised the refactoring strategy, adapted similarity analysis and clustering discovery techniques, implemented the proposed approach and performed evaluation experiments. The technical details are presented in Chapter 3.

[ZGTC21] L. Zhu, G. Giotis, V. Tountopoulos and G. Casale. **RDOF: Deployment Optimization for Function as a Service**. In *Proceedings of the 14th IEEE International Conference on Cloud Computing*, 2021.

We propose in this paper a model-driven approach, RDOF, for finding the optimal deployment scheme of a serverless application. I contribute the majority of the paper, exploring FaaS compute services, defining the performance model, formulating the optimization problem, implementing the proposed approach and conducting evaluation experiments. The main results are reported in Chapter 4

[AGW+21] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale and A. Filieri. **Quality-Aware DevOps Research: Where Do We Stand?** *IEEE Access*, 2021.
This paper provides a holistic survey of quality-aware DevOps research on architecture design, modeling and infrastructure as code, continuous integration/continuous delivery, testing and verification as well as runtime management. Possible directions for future work, particularly AI for DevOps and DevOps for AI software, are also tracked. I am the contributor of the architecture design and model-driven DevOps part.

[ZCP20] L. Zhu, G. Casale and I. Perez. **Fluid Approximation of Closed Queueing Networks with Discriminatory Processor Sharing**. *Performance Evaluation*, 2020.
A fluid approach to transient and steady-state analysis of closed QNs consisting of delay and DPS stations is proposed in this paper. Moreover, we introduce a refined method for approximating station and system response time distributions through transient analysis. I contribute most of the paper, with the idea of adding self-looping jobs being an outcome of joint efforts. The proposed approach is elaborated and applied to performance prediction of OpenFaaS functions in Chapter 5

[CAvdH+20] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, S. N. Srirama, D. A. Tamburri, M. Wurster, L. Zhu. **RADON: Rational Decomposition and Orchestration for Serverless Computing**. *Software-Intensive Cyber-Physical Systems*, 2020.
This paper clarifies the research agenda of the RADON project, which aims to develop an open-source DevOps framework centered around serverless computing. Specifically, we investigate open challenges that the RADON framework needs to address, envision the toolchain and methodology that it will deliver, and describe industrial use cases that can benefit from the

framework. I am the contributor of the sections pertaining to quality assurance, and the editor of the paper as well.

Chapter 2

Background

2.1 Serverless Computing

This thesis centers around serverless computing, a compelling form of cloud computing that could radically reshape today's landscape of software applications. The next three sections give an introduction to serverless computing and important concepts therein as well as popular function-as-a-service (FaaS) offerings.

2.1.1 General Description

The recent preference for a fine-grained cloud-native architecture in industry leads to the emergence of the FaaS paradigm and draws an extensive attention to serverless computing. What does the oxymoron "serverless computing" mean? Serverless computing advocates offloading the responsibility of operational tasks, including fault tolerance, placement, load balancing, autoscaling, logging, monitoring and upgrading, to the cloud provider so that developers can completely concentrate on implementing the business logic of their applications. This causes an illusion that there existed no servers to care about, although serverless applications are still hosted on anonymous servers managed by the cloud provider.

In a nutshell, serverless computing is FaaS plus backend as a service (BaaS). FaaS offerings enable the deployment of the core computing components, i.e. functions, for a serverless application. A function is defined as a modular piece of code running in a stateless container. The stateless nature of the FaaS paradigm greatly facilitates the cloud provider to operate each function individually in the interest of its customers. Serverless functions by themselves are most likely to serve as generic webhooks, event handlers or scheduled actions. These alone may not seem to be very attractive due to the limited use cases. As a result, FaaS is often combined with BaaS to implement sophisticated business logic. Various BaaS services are available on cloud platforms for fulfilling specific application requirements, such as file manipulation, database management and remote notification, allowing the outsourcing of behind-the-scenes functionalities that are possibly stateful. FaaS and BaaS together constitute a rich ecosystem for building serverless applications.

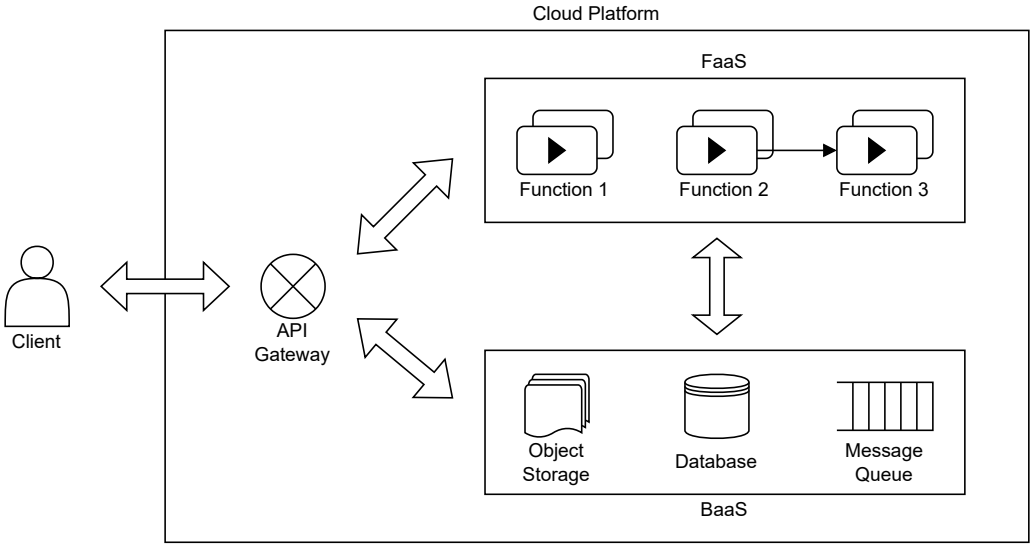


Figure 2.1: The architecture of a serverless application.

Figure 2.1 illustrates the architecture of a serverless application, which consists of stateless functions complemented with BaaS components like object storages, databases and message queues. The features of a serverless application are typically enacted by invoking the corresponding functions through the API gateway of the cloud platform. During the execution, a function may access BaaS components or invoke other functions. The latter in particular empowers developers to describe a complex workflow as a directed acyclic graph of small tasks, each carried out by a single function. It is also possible to trigger the execution of a serverless

function by accessing a BaaS component through the API gateway. For example, a client of a shopping application may be designed to insert an invoice into a message queue, generating an event to invoke a function that fetches the invoice from the message queue and sends it to the customer by email. Serverless functions are additionally configurable to react in response to scheduled events. This is useful when an application needs to perform periodic jobs.

Serverless computing synthesizes the short-lived computations of FaaS with the reliable networking, persistent storage, detailed monitoring and robust security of BaaS. The combination of FaaS and BaaS brings cloud users several notable benefits. These are the main motivations for the rapid popularization of serverless computing:

- *Pay-as-you-go cost model.* Since resource provision and management are delegated to the cloud provider, users pay for resources on a consumption basis rather than on an allocation basis. The creation and destruction overhead as well as the idle time are not charged. This pay-as-you-go cost model largely saves the operating costs of a cloud application especially under a bursty workload.
- *Little management effort.* Modern FaaS and BaaS offerings significantly relieve the burden of complex operations. For example, developers only need to specify a few essential parameters, namely runtime, memory, concurrency and timeout, for a serverless function, and do not have to worry about any low-level operation of an object storage, which is fully taken care by the cloud provider.
- *Seamless scalability.* A serverless application is built of various virtual components, such as stateless functions and object storages. This results in a fine-grained architecture, enabling the cloud provider to achieve efficient multi-tenant multiplexing on heterogeneous infrastructure and scale each application component up and down quickly to meet the variable demand.
- *Higher development productivity.* The cloud provider is responsible to operate components of the same type in a portable and reproducible way. Users can encapsulate the core business logic of their applications into reusable components. Furthermore, blueprints of common components may be shared in public marketplaces or repositories. The delivery time of new products to the market is then shortened.

2.1.2 Important Concepts

Performance modeling for serverless computing is a recurrent research question involved in Chapters 4 and 5. Understanding the technical details of serverless computing, particularly FaaS, is a key requisite to find the answer. In the following, we overview important concepts that appear frequently in the context of serverless computing.

Function Instances. The computation of a serverless application is conducted by the *instances of functions* that compose it. These instances are fully managed by the cloud provider and act as tiny servers for predefined triggers. We identify two basic invocation scheduling disciplines in serverless computing: *first-come-first-served* (FCFS) and *processor-sharing* (PS). In the former case, a function instance simply process one invocation at a time, and extra invocations will be queued, dropped or retried after a delay when no instances are available. Examples of FaaS offerings that adopt the FCFS discipline are AWS Lambda[3], Google Cloud Functions[4] and OpenWhisk[7]. By contrast, Microsoft Azure[5] and OpenFaaS[6] allows an instance to process multiple invocations simultaneously, and there is no limit on how many invocations that the instance can accept. This mimics the well-known PS discipline. It is worth noting that functions on Microsoft Azure are bundled up into function apps and dynamically scaled in groups. We distinguish instances of such function apps from the others by calling them *app instances*.

Cold/Warm Start. Function instances may be running in three different states: *initialization*, *busy* and *idle*. When the workload of a function grows, additional instances of that function are launched to handle more invocations. A new instance first enters the initialization state, where a runtime environment is established for hosting it. This process is referred to as *cold start*. After that, the instance begins to serve invocations and is considered to be in the busy state. Both cold start and execution time of the instance are billed by the cloud provider. Once all the invocations are done, the instance switches to the idle state. Idle instances will be preserved for some time to deal with further invocations. The reuse of an idle instance is termed *warm start*. Since the length of warm start is often negligible, the response time of an invocation to a function can be calculated as the sum of network delay and execution time plus cold start if a new instance is created.

Memory, Concurrency and Timeout. Although it is the cloud provider's responsibility to scale a serverless function on demand, resources for function instances can be controlled by specifying the *memory* of the function. This parameter normally determines not only memory but also CPU resources allocated to each function instance. With 1792 MB memory, an instance of a Lambda function can use the equivalent of a full vCPU, i.e. a thread of a CPU core. Developers may also need to set the *concurrency* of a function, for example when the throughput of the downstream services are constrained. The interpretations of concurrency are different depending on the basic discipline adopted to schedule invocations. In the case of FCFS scheduling, the concurrency parameter restricts the number of instances that a function can have. As for PS scheduling, it corresponds to the maximum number of invocations that an instance can process concurrently. If the duration of an invocation is too long, the function will abort the execution and returns an error. This aims to prevent the incurrence of excessive costs in anomalous situations like deadlocks and is generally governed via a parameter called *timeout*.

2.1.3 Function-as-a-Service Offerings

As mentioned earlier, the core computing components of a serverless application are stateless functions. We now report the common features of popular FaaS offerings, covering both commercial FaaS compute services on public cloud platforms and free FaaS-oriented frameworks in the open-source community.

Most public cloud platforms incorporate FaaS compute services, such as AWS Lambda, Google Cloud Functions and Microsoft Azure Functions. AWS Lambda has enabled a rich set of languages for coding functions, including C#, Go, Java, Node.js, PowerShell, Python and Ruby. Users of Google Cloud Functions may merely choose among Go, Node.js and Python, and there are not many options for Microsoft Azure Functions either: C#, F#, Java and Node.js. On public cloud platforms, the access of functions to other services is under strict control of an identity and access management (IAM) system. A function with adequate permission can employ a storage service to keep data persistent. Function executions may be triggered by events generated from other internal services or by HTTP requests sent to the API gateway.

Logging and monitoring services are also present to help in debugging and testing functions. Google Cloud restricts the number of functions per project to 1000. However, developers can deploy as many functions as they want on AWS and Microsoft Azure. Public cloud platforms measure resources allocated to each instance in terms of memory. The maximum memory of a function is 3000 MB, 2048 MB and 1536 MB for AWS Lambda, Google Cloud Functions and Microsoft Azure Functions, respectively. AWS assigns each account a quota of 1000 concurrency, which is upgradable if requested, whereas Google Cloud allows a function to have a concurrency of up to 1000. In particular, no concurrency limit is imposed by Microsoft Azure Functions. The maximum timeout that can be specified for a function varies across different platforms as well: 900 s on AWS, 540 s on Google Cloud and 600 s on Microsoft Azure. Table 2.1 compares the three FaaS compute services from a user perspective.

Table 2.1: Comparison of AWS Lambda, Google Cloud Functions and Microsoft Azure Functions from a user perspective.

FaaS Service	AWS Lambda	Google Cloud Functions	Microsoft Azure Functions
Release Time	April 2015	July 2018	November 2016
Programming Languages	C#, Go, Java, Node.js, PowerShell, Python, Ruby	Go, Node.js, Python	C#, F#, Java, Node.js
Access Control	AWS IAM	Cloud IAM	Azure RBAC
Persistent Storage	Amazon S3, Amazon DynamoDB, etc.	Cloud Storage, Cloud Datastore, Cloud SQL, etc.	Azure Blob Storage, Azure Table Storage, etc.
Event Sourcing	Amazon S3, Amazon SQS, Amazon API Gateway, etc.	Cloud Storage, Cloud Pub/Sub, HTTP, etc.	Azure Blob Storage, Azure Queue Storage, HTTP, etc.
Logging & Monitoring	Amazon CloudWatch, AWS X-Ray	Google Stackdriver	Built-in Logger, Azure Monitor
Maximum Number of Functions	Unbounded	1000 per project	Unbounded
Maximum Memory	3008 MB per function instance	2048 MB per function instance	1536 MB per app instance
Maximum Concurrency	1000 per account (upgradable)	1000 per function	Unbounded
Maximum Timeout	900 s	540 s	600 s

Two FaaS-oriented frameworks, OpenFaaS and OpenWhisk, are getting prominent in the open-source community. Both of them claim to be Kubernetes[9]native. Notably, the former is installable on a Docker[10] swarm, while the latter may simply run on a single Docker[10] node. OpenFaaS provides templates for writing functions in seven languages: C#, Go, Java, JavaScript, PHP, Python and Ruby. These templates are Gradle[17] projects that are ready to be compiled. As for OpenWhisk, developers can implement functions in one more language, Swift, but have to create projects by themselves for managing the source code. OpenWhisk facilitates the application of three event feeding patterns, namely hook, polling and connector, whereas only the first and third ones are feasible with OpenFaaS. The integration of a third-party proxy, for example Kong[18] or Traefik[19], is essential for OpenFaaS to secure function invocations. By contrast, OpenWhisk ensures the security of functions based on built-in utilities and may be configured to employ a third-party proxy as an alternative solution. Table 2.2 compares the two FaaS-oriented frameworks from a user perspective.

Table 2.2: Comparison of OpenFaaS and OpenWhisk from a user perspective.

FaaS Framework	OpenFaaS	OpenWhisk
Start Time	December 2016	February 2016
Open-Source License	MIT	Apache v2
Corporate Backer	VMware	IBM
Development Language	Go	Scala
Container Platform	Kubernetes, Docker (swarm)	Kubernetes, Docker (single node), etc.
Function Coding Style	Template-Based	Traditional
Supported Languages	C#, Go, Java, Node.js, PHP, Python, Ruby	C#, Go, Java, Node.js, PHP, Python, Ruby, Swift
Event Feeding Patterns	Hook, Connector	Hook, Polling, Connector
Security Solutions	Proxy	Built-in, Proxy

[17] `https://gradle.org/`
[18] `https://konghq.com/`
[19] `https://traefik.io/`

2.2 Machine Learning

Machine learning is a field of inquiry devoted to building systems that can learn from experience or more precisely improve performance based on historical data. This section reviews machine learning techniques for natural language processing and cluster discovery, which are leveraged by the architecture decomposition approach proposed in Chapter 3.

2.2.1 Natural Language Processing

Natural language processing (NLP) employs computational models for machine understanding and generation of human languages, scaling the capabilities of computers to carry out various linguistic tasks such as spelling & grammar correction, information retrieval, text classification, speech recognition, sentiment analysis, automatic translation and question answering. Text pre-processing and/or semantic representation is an essential part of modern NLP procedures for accomplishing most of these tasks. In the following, we briefly introduce text pre-processing and semantic representation techniques adopted in Section 3.5 to compute semantic similarities between API operations.

Text Pre-Processing

The goal of text pre-processing is to convert a raw text into a sequence of linguistic items suitable for a specific NLP task. Typical text pre-processing steps include tokenization, part-of-speech tagging, dependency parsing, named entity recognition and stemming/lemmatization among others, which can be concatenated into an NLP pipeline.

Tokenization. In the scope of NLP, tokenization means separating words, numbers and punctuation marks, collectively termed *tokens*, out of a flowing text. It is the first step of almost every NLP pipeline. Words are neatly delimited by white spaces in English and many continental European languages like French, German and Spanish. This apparently helps with word separation, despite not being always sufficient. Tokenizing a text written in Asian languages, such

as Chinese, Japanese and Thai, that do not mark word boundaries explicitly is more difficult. For simplicity, our discussion is restricted to English only.

Numbers and punctuation marks are generally kept as individual tokens, because they convey semantic and syntactic information in the text under processing. For example, strings like "$19.99" would be of major interest in the task of crawling an e-commerce website for the prices of products on sale. Additional complications arise from the fact that the roles of certain punctuation marks are subject to the surrounding contexts. For example, a period has three possible roles: the end of a sentence (as in "It's rainy outside."), the abbreviation of a word (as in "A.M.") and a decimal point (as in "3.14"), which need to be dealt with differently.

The standard tokenization approach is to sequentially search for strings that match a series of *regular expressions* [Kle51, Kle56]. Such a search can be done efficiently by compiling the regular expressions into *deterministic finite automata* and running the resultant automata on the input text to identify all the matching strings. It is also viable to resolve ambiguities caused by punctuation marks with carefully designed regular expressions. A common reference in this respect is the collection of regular expressions for creating tokens in the Penn treebank,[20] a syntactically parsed corpus released by the linguistic data consortium.[21]

Part-of-Speech Tagging. A part of speech is a class of words with similar syntactic functions. In English grammar, words are mainly classified into nine parts of speech: nouns, verbs, adjectives, adverbs, pronouns, prepositions, conjunctions, interjections and determiners. The first four plus interjections are open classes, where new members are continually coined. By contrast, the rest are closed classes, whose membership is relatively stable. NLP tasks often requires a more fine-grained classification of words. As an example, Table 2.3 lists part-of-speech tags used in the universal dependencies treebanks [NDMG+16].

Parts of speech provide valuable clues about syntactic relations between neighboring words and the structure of a phrase, a sentence or even a paragraph. Knowing what part of speech each word belongs to can significantly increase the accuracy of NLP models for a wide range

[20] `https://catalog.ldc.upenn.edu/LDC99T42`
[21] `https://www.ldc.upenn.edu/`

Table 2.3: Part-of-speech tags used in the universal dependencies treebanks [NDMG+16].

Open Class		**Closed Class**	
Tag	**Part of Speech**	**Tag**	**Part of Speech**
NOUN	Noun	PRON	Pronoun
PROPN	Proper Noun	AUX	Auxiliary
VERB	Verb	ADP	Preposition/Postposition
ADJ	Adjective	PART	Particle
ADV	Adverb	CONJ	Coordinating Conjunction
INTJ	Interjection	SCONJ	Subordinating Conjunction
Others: PUNCT - Punctuation, SYM - Symbol, X - Unspecified		DET	Determiner
		NUM	Numeral

of tasks such as spelling & grammar correction, and is a particularly important prerequisite for word lemmatization. The process of assigning part-of-speech tags, like NOUN and VERB, to a sequence of words, usually full sentences, according to their definitions and contexts is known as part-of-speech tagging.

By its nature, part-of-speech tagging is a sequence labeling problem whose input and output are a sequence $\boldsymbol{w} = (w_i)$ of words and a sequence $\boldsymbol{p} = (p_i)$ of part-of-speech tags respectively, each output tag p_i corresponding exactly to an input word w_i. *Hidden Markov models* [BP66] and *conditional random fields* [LMP01] are two types of stochastic models for solving sequence labeling problems. Being *generative*, the former give the joint probability $P(\boldsymbol{w}, \boldsymbol{p})$, from which the conditional probability $P(\boldsymbol{p} \mid \boldsymbol{w})$ is then computed. The latter are *discriminative* in the sense that the conditional probability $P(\boldsymbol{p} \mid \boldsymbol{w})$ is instead estimated straightforwardly. Part-of-speech taggers can be built on *recurrent neural networks* [Elm90] or *transformers* [VSP+17] as well, but these neural models perform similarly to the stochastic ones for English.

Stemming/Lemmatization. For grammatical reasons, English words are present in either *primitive* or *inflectional* forms. Take the verb "ask" as an example. "ask" is the primitive form that serves in the present simple tense. The inflectional forms of "ask" include "asking" (present participle), "asks" (present third-person-singular) and "asked" (past simple/participle). In some NLP tasks such as information retrieval, it is preferable to reduce inflected words like "asking" to a canonical form like "ask". Such a process is called word normalization. Stemming and lemmatization are two optional techniques for normalizing words.

From a morphological point of view, a word is composed of one or more *morphemes*, i.e. the smallest meaningful lexical units in a language. English morphemes are divided into two categories: *stems* and *affixes*. Every word has a stem, which acts as the central morpheme and provides the principal meaning. An affix is a morpheme attached before a stem (prefix) or after a stem (suffix), adding a supplementary meaning. For example, the plural noun "cats" is made of a stem "cat" and a suffix "s". Stemming is to reduce inflected words to their stems based on the spelling of each word. One of the most popular stemmers is the Porter stemmer [Por80], which strips the suffix of a word by executing a cascade of rewrite rules like deleting the only "s" at the end of the word.

Despite being simple, stemming is a crude normalization technique. It may reduce words of different roots to the same stem (over-stemming), e.g. "universe" and "university" to "univers", or words of the same root to different stems (under-stemming), e.g. "data" and "datum" to "dat" and "datu". As also exemplified, the resultant stems are not necessarily valid words. If these issues really matter, lemmatization should be chosen as the normalization technique. A *lemma* is the citation form used in dictionaries for a set of words with the same stem, the same part of speech and the same meaning. Lemmatization is to reduce inflected words to their lemmas based on the spelling, part of speech and context of each word. This may be via direct dictionary lookup or complete morphological parsing.

Semantic Representation

A word originally appears as a string of characters, which is not a satisfactory representation for semantic computation. The *distributional hypothesis* [Har54] states that linguistic items occurring in similar contexts tend to have similar meanings. Accordingly, the semantics of words can be represented as vectors learned from their distributions.

TF-IDF Weighting. Term frequency-inverse document frequency (TF-IDF) [Luh57, Jon72] is a statistical measure of how relevant a term in a vocabulary is to a document in a corpus. Let $\mathcal{T}$ and $\mathcal{D}$ denote a vocabulary and a corpus, t a term in the vocabulary $\mathcal{T}$, i.e. $t \in \mathcal{T}$, and d a document in the corpus $\mathcal{D}$, i.e. $d \in \mathcal{D}$. The TF-IDF weight of t in $\mathcal{T}$ with respect to d in $\mathcal{D}$ is

defined as the product of two factors:

$$\text{tf-idf}(t, d, \mathcal{D}) = \text{tf}(t, d)\text{idf}(t, \mathcal{D}), \tag{2.1}$$

where $\text{tf}(t, d)$ is the term frequency of t in d, and $\text{idf}(t, \mathcal{D})$ is the inverse document frequency of t in $\mathcal{D}$. Specifically, the former scores the regularity of t in d by the number of times that t occurs in d:

$$\text{tf}(t, d) = |\{w \in d \mid w = t\}|, \tag{2.2}$$

whilst the latter scores the specificity of t in $\mathcal{D}$ by the logarithm of the inverse fraction of documents containing t in $\mathcal{D}$:

$$\text{idf}(t, \mathcal{D}) = \log\left(\frac{|\mathcal{D}|}{|\{d \in \mathcal{D} \mid t \in d\}|}\right). \tag{2.3}$$

The curly brackets in (2.2) and (2.3) symbolize bags and sets, respectively. Alternative definitions of the TF and IDF factors are available and possibly more practical depending on the scenario.

A *term-document matrix* can be created through TF-IDF weighting, with each entry being the TF-IDF weight of a term in a vocabulary with respect to a document in a corpus. The rows of the term-document matrix in particular represent terms in the vocabulary as per their relevance to documents in the corpus, thereby constituting a *vector space model* of the vocabulary based on the corpus. Vector space models are crucial to NLP tasks that rely on semantic analysis, for example text classification. Thanks to the *additive compositionality* of vector representations, the semantic vector of multiple words is simply the summation of those across each individual word. This largely facilitates representing the overall semantics of a text. Moreover, vector representations enable the quantification of to what extent a pair of texts are semantically similar through the *cosine similarity*:

$$s^{\text{sem}} = \cos\theta = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|_2 \|\boldsymbol{v}\|_2}, \tag{2.4}$$

where θ is the included angle between the semantic vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ of the two texts. It is worth pointing out that the additive compositionality and cosine similarity are natural properties of any vector space model.

Word Embedding. Semantic vectors arising from TF-IDF weighting have as many entries as the number of documents in the underlying corpus, and most of these entries are zero or negligible. Such a sparse vector model does not work very well due to its high dimensionality and lack of informative features. A series of techniques including word2vec [MCCD13], GloVe [PSM14], fastText [BGJM17], BERT [DCLT19] and GPT [RNSS18] has recently emerged to help in learning a dense vector space model, termed a word embedding, from a corpus. There are two types of word embeddings, namely *static* and *dynamic*. Models generated using word2vec, GloVe or fastText are said to be static, since they associate every word with a fixed semantic vector. By contrast, models produced via BERT or GPT fit the semantic vector of a word to the surrounding context and are thus deemed to be dynamic.

The first word embedding technique is word2vec, which trains a two-layer neural network for predicting the target word for some context words (continuous bag-of-words) or the context words for a target word (continuous skip-gram) and extracts the final weights at the hidden layer as the entries of semantic vectors. Standing for global vectors, GloVe builds a vector space model from the ratios of word-word co-occurrence probabilities in a corpus. Vector representations resulting from GloVe exhibit meaningful linear substructures. To deal with words out of vocabulary and languages with rich morphology, fastText improves upon word2vec by feeding character-level n-grams instead of whole words as input. The semantic vector of an unknown word can thus be synthesized from those of its constituent n-grams by applying the additive compositionality. Notably, fastText has been demonstrated on a range of 157 languages [GBG$^+$18].

Dynamic word embedding techniques such as BERT and GPT take advantage of transformers, a novel form of multi-layer neural network that adopts an encoder-decoder architecture with self-attention to differentially process each part of the input data in parallel. Bidirectional encoder representations from transformers (BERT) constructs a vector space model by training a bidirectional transformer encoder on two simultaneous tasks: masked word prediction and next sentence judgment. Unlike BERT, generative pre-training transformer (GPT) leverages a unidirectional transformer decoder and trains it on the task of next word prediction. Vector space models obtained by GPT can capture long-range contexts.

2.2.2 Cluster Discovery

Clustering is an unsupervised machine learning technique that aims to divide a collection of M data points $x_1, x_2, \ldots, x_M$ into K disjoint groups $C_1, C_2, \ldots, C_K$, termed *clusters*, according to their similarities or distances. As such, data points in the same cluster are more similar or closer in some sense to each other than to those in a different cluster. Clustering algorithms normally requires the specification of a few input parameters. Some of the parameters are difficult to determine from a priori knowledge. A common solution is to collect partitions under multiple parameter settings and keep the one that optimizes a validity index computed based on the data set. We investigate next the clustering algorithms and validity indices adopted in Section 3.6 to find groups of correlated API operations.

Clustering Algorithms

Four types of clustering algorithms are extensively used in practice: linkage-, centroid-, density- and graph-based. The hierarchical [Joh67], K-medoids [KR90], DBSCAN [EKSX96] and spectral [SM00] clustering algorithms are famous instances of these types that do not have restrictions on the similarity or distance measure.

Hierarchical Clustering. A hierarchy of clusters can be built in either a bottom-up or a top-down manner, called the *agglomerative* and the *divisive* strategy respectively. The former starts with each data point assigned to a separate cluster and iteratively merges the closest two clusters until the stop criterion is satisfied. Conversely, the latter performs clustering by initially assigning all the data points to a single cluster and splitting the one with the greatest diameter at each iteration. The hierarchical clustering algorithm under consideration is agglomerative because it is generally more efficient than one that adopts the divisive strategy.

Algorithm 2.1 briefly describes the hierarchical clustering algorithm [Joh67]. The behavior of this algorithm is significantly subject to the definition of the inter-cluster distance, known as the *linkage*. Below are three typical definitions:

Algorithm 2.1: The hierarchical clustering algorithm [Joh67].

Input:

- M data points: $x_1, x_2, \ldots, x_M$
- Number of clusters: K

Output:

- K clusters: $C_{l_1}, C_{l_2}, \ldots, C_{l_K}$

1: For every pair of data points x_i and x_j, compute their distance $d(x_i, x_j)$
2: Create M clusters $C_1, C_2, \ldots, C_M$, each containing one data point x_i
3: For every pair of clusters C_k and C_l, compute their distance $D(C_k, C_l)$
4: While there are more than K clusters left,
5: Merge the closest clusters C_k and C_l into a new cluster $C_{k'}$
6: Compute the distance $D(C_{k'}, C_{l'})$ between the new cluster $C_{k'}$ and every other one $C_{l'}$

- the *single linkage* (i.e. the shortest distance), where

$$D(C_k, C_l) = \min_{x_i \in C_k, x_j \in C_l} d(x_i, x_j); \tag{2.5}$$

- the *complete linkage* (i.e. the longest distance), where

$$D(C_k, C_l) = \max_{x_i \in C_k, x_j \in C_l} d(x_i, x_j); \tag{2.6}$$

- the *average linkage* (i.e. the average distance), where

$$D(C_k, C_l) = \frac{1}{|C_k||C_l|} \sum_{x_i \in C_k} \sum_{x_j \in C_l} d(x_i, x_j). \tag{2.7}$$

Hierarchical clustering is applicable to a wide range of scenarios, provided that an appropriate definition of the inter-cluster distance is chosen. The single linkage is helpful in obtaining elongated clusters as remote data points may be chained together in the same cluster. If the complete linkage is in use, two clusters will be merged late in the hierarchy when they contain at least one pair of data points that are far from each other. Therefore, clusters arising from the complete linkage are often compact in the sense of a small diameter. The average linkage is more preferable for spherical clusters but also adequate to deal with other cases.

K-Medoids Clustering. The most popular algorithm for partitioning a data set is perhaps K-means [Mac67], which clusters data points by minimizing the *sum of squared error*. At the beginning, K data points are randomly drawn to represent the clusters. Every data point is then assigned to the cluster with the closest representative, and the *centroids* of the clusters are used as the new representatives. The two steps are iterated until the resultant partition no

longer changes. As a major limitation, the K-means clustering algorithm requires the distance measure to be Euclidean.

Algorithm 2.2: The K-medoids clustering algorithm [KR90].

Input:

- M data points: $x_1, x_2, \ldots, x_M$
- Number of clusters: K

Output:

- K clusters: $C_1, C_2, \ldots, C_K$
- K medoids: $o_1, o_2, \ldots, o_K$

1: For every pair of data points x_i and x_j, compute their distance $d(x_i, x_j)$
2: Randomly select K data points as the medoids $o_1, o_2, \ldots, o_K$
3: Compute the total deviation TD
4: While the total deviation TD is not minimal,
5: Swap a medoid o_k with a non-medoid x_i such that the total deviation TD is reduced most
6: Construct K clusters $C_1, C_2, \ldots, C_K$ by associating each data point x_i to the closest medoid o_k

An alternative algorithm that can overcome this limitation is K-medoids. As shown in Algorithm 2.2, K-medoids differs from K-means in two aspects. It always selects existent data points—*medoids*—as the representatives of the clusters, and aims to minimize the *total deviation*, i.e. the sum of the distance from each data point x_i to the closest medoid o_k:

$$TD = \sum_{i=1}^{M} \min_{k \in [1..K]} d(x_i, o_k). \tag{2.8}$$

These make the K-medoids clustering algorithm not only relatively robust in the case of outliers but also able to handle arbitrary distance measures. Notably, the K-medoids problem is NP-hard to solve exactly, and the algorithm that we present is a widely applied heuristic named partitioning around medoids (PAM) [KR90].

DBSCAN Clustering. The basic idea behind density-based clustering is to identify density patterns inherent in the data set and associate each data point to the density pattern that attracts it. Density-based spatial clustering of applications with noise (DBSCAN) is a non-parametric density-based clustering algorithm. In the setup of DBSCAN, two data points within a distance ϵ are deemed to be *neighbors*, and one with at least m neighbors is viewed as a *core point*. The above notions yield three types of connectivities between a pair of data points x_i and x_j:

- if x_i is a core point and x_j is a neighbor of x_i, then x_j is said to be *directly density-reachable* from x_i;
- if there exists a sequence $x_i, \ldots, x_j$ where each data point is directly density-reachable from the previous one, then x_j is said to be *density-reachable* from x_i;
- if x_i and x_j are density-reachable from the same core point, then x_i and x_j is said to be *density-connected*.

DBSCAN essentially finds clusters as groups of density-connected data points. In addition, it classifies a non-core point as *noise* if that point is not directly density-reachable from any core points. Algorithm 2.2 summarizes the DBSCAN clustering algorithm [EKSX96]. This is an abstract version of DBSCAN for illustrative purposes only. The original algorithm constructs one cluster at a time and does not need to store the neighbors of every data point, thereby being less spatially intensive.

Algorithm 2.3: The DBSCAN clustering algorithm [EKSX96].

Input:
- M data points: $x_1, x_2, \ldots, x_M$
- Neighborhood of a data point: ϵ
- Minimum neighbors of a core point: m

Output:
- A number of clusters: $C_1, C_2, \ldots$
- A number of noise points: $y_1, y_2, \ldots$

1: For every pair of data points x_i and x_j, compute their distance $d(x_i, x_j)$
2: For every data point x_i,
3: Find all the data points within the neighborhood ϵ of x_i
4: Label x_i as a core point if it has at least m neighbors
5: Find groups of core points that are density-reachable from each other as the clusters C_1, C_2, ...
6: For each non-core point x_i,
7: If x_i is directly density-reachable from a core point x_j of a cluster C_k, label it as a border point belonging to C_k
8: Otherwise, label x_i as a noise point

Spectral Clustering. Given the pairwise similarities between data points, a data set can be depicted as a weighted undirected graph termed the *similarity graph*, where each vertex represents a data point and each edge characterizes the neighborhood relationship between two data points. Solving the clustering problem is then equivalent to seeking the best cuts of the similarity graph, which is feasible via spectral decomposition of the *Laplacian matrix* derived

from the edge weights. Specifically, the first K eigenvectors of the Laplacian matrix form a K-dimensional representation of the data points and can thus be fed into a simple clustering algorithm such as K-means for cluster discovery. This is known as spectral clustering.

Algorithm 2.4: The spectral clustering algorithm [SM00].

Input:

- M data points: $x_1, x_2, \ldots, x_M$
- The number of clusters: K
- The scaling factor of the Gaussian kernel: σ

Output:

- K clusters: $C_1, C_2, \ldots, C_K$

1: For every pair of data points x_i and x_j, compute their distance $d(x_i, x_j)$
2: Construct a similarity graph by transforming each pairwise distance $d(x_i, x_j)$ into an edge weight $w(x_i, x_j)$ through the Gaussian kernel $G(d; \sigma)$
3: Compute the normalized random-walk Laplacian matrix $\boldsymbol{L}^{\mathrm{RW}}$ of the similarity graph
4: Solve the eigenvalue problem $\boldsymbol{L}^{\mathrm{RW}}\boldsymbol{u} = \lambda\boldsymbol{u}$ for the first K eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_K$
5: Build a matrix $\boldsymbol{U}$ with the K eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_K$ as columns
6: Obtain K clusters $C_1, C_2, \ldots, C_K$ by performing the K-means clustering algorithm on the rows of the matrix $\boldsymbol{U}$

Multiple spectral clustering algorithms have been proposed. They mainly differs in the choice of the similarity graph and/or the Laplacian matrix. Algorithm 2.4 illustrates the spectral clustering algorithm adapted from [SM00]. A fully connected similarity graph is constructed to model the data set, and the weight of the edge between two data points x_i and x_j is computed as

$$w(x_i, x_j) = G(d(x_i, x_j); \sigma) = e^{-\left(\frac{d(x_i, x_j)}{\sigma}\right)^2}, \tag{2.9}$$

where $G(d; \sigma)$ denotes the *Gaussian kernel* that takes an input distance d and is tuned by a scaling factor σ. The Laplacian matrix in use is given by

$$\boldsymbol{L}^{\mathrm{RW}} = \boldsymbol{D}^{-1}(\boldsymbol{D} - \boldsymbol{A}) = \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}, \tag{2.10}$$

where $\boldsymbol{D} = \mathrm{diag}(\sum_{j=1}^{M} w(x_i, x_j)) \in \mathbb{R}^{M\times M}$ and $\boldsymbol{A} = [w(x_i, x_j)] \in \mathbb{R}^{M\times M}$ are the outdegree and adjacency matrices of the similarity graph respectively, and $\boldsymbol{I} \in \mathbb{R}^{M\times M}$ is an identity matrix. $\boldsymbol{L}^{\mathrm{RW}}$ is referred to as the *normalized random-walk Laplacian matrix*. A justification of the algorithm is beyond the scope of this thesis. We point the reader to [VL07] for more information.

Validity Indices

Validity indices computed based on the data set are called *internal validity indices*. They are often applied to automatically parameterizing clustering algorithms. The gamma [BH75] and silhouette [Rou87] indices are two normalized internal validity indices that do not require data points to be explicitly defined on a vector space.

Gamma Index. The gamma index [BH75] is an adaption of the Goodman and Kruskal's gamma statistic for clustering validity analysis. Consider two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of the same length. A pair of indices r and s are said to be *concordant* if $(a_r - a_s)(b_r - b_s) > 0$. Otherwise, they are said to be *discordant* if $(a_r - a_s)(b_r - b_s) < 0$. Note that pairs associated with equality are ignored. Let N_+ and N_- be the numbers of concordant and discordant pairs in $\boldsymbol{a}$ and $\boldsymbol{b}$. The gamma statistic can then be written as

$$\Gamma = \frac{N_+ - N_-}{N_+ + N_-}, \tag{2.11}$$

which varies from -1 to 1 and quantifies the rank correlation between $\boldsymbol{a}$ and $\boldsymbol{b}$.

In the context of clustering, $\boldsymbol{a}$ and $\boldsymbol{b}$ are interpreted as the pairwise distance and indicator vectors respectively. Both of them contain a total of $N(N-1)/2$ entries, each corresponding to a pair of data points x_i and x_j $(i < j)$. Specifically, the (i,j)th entry of $\boldsymbol{a}$ is the distance between x_i and x_j, while that of $\boldsymbol{b}$ is either 0 or 1 depending on whether x_i and x_j lie in different clusters or the same cluster. N_+ thus counts the number of times that the distance between a pair of data points x_i and x_j belonging to the same cluster is smaller than the distance between another pair of data points $x_{i'}$ and $x_{j'}$ belonging to different clusters:

$$N_+ = \sum_{(x_i,x_j)\in P_\mathrm{W}} \sum_{(x_{i'},x_{j'})\in P_\mathrm{B}} \mathbb{1}_{\{d(x_i,x_j)<d(x_{i'},x_{j'})\}}, \tag{2.12}$$

where $P_\mathrm{W} = \{(x_i, x_j) \mid i < j \wedge (\exists k, x_i \in C_k, x_j \in C_k)\}$ and $P_\mathrm{B} = \{(x_i, x_j) \mid i < j\} \setminus P_\mathrm{W}$ are the sets of pairs of data points within the same cluster and between different clusters respectively, and $\mathbb{1}_A$ denotes the *indicator function* whose value is either 0 or 1 depending on the occurrence

of event A. By contrast, N_- tallies the number of times that the opposite situation occurs:

$$N_- = \sum_{(x_i,x_j)\in P_\mathrm{W}} \sum_{(x_{i'},x_{j'})\in P_\mathrm{B}} \mathbb{1}_{\{d(x_i,x_j)>d(x_{i'},x_{j'})\}}. \tag{2.13}$$

When the gamma index peaks at 1, the data set is perfectly partitioned in the sense that no pair of data points within the same cluster is farther than any pair of data points between different clusters.

Silhouette Index. The silhouette index [Rou87] combines within-cluster compactness and between-cluster separateness. Consider a data point x_i assigned to a cluster C_k. The average distance between x_i and other data points in C_k is

$$a(x_i) = \frac{1}{|C_k|-1} \sum_{\substack{x_j\in C_k \\ j\neq i}} d(x_i,x_j), \tag{2.14}$$

whilst the average distance between x_i and those in the closest cluster other than C_k is

$$b(x_i) = \min_{\substack{l\in[1..K] \\ l\neq k}} \frac{1}{|C_l|} \sum_{x_j\in C_l} d(x_i,x_j). \tag{2.15}$$

$a(x_i)$ and $b(x_i)$ respectively measure how compact x_i is to the current cluster, i.e. C_k, and how separate x_i is from the best alternative, i.e. the closest cluster other than C_k. The silhouette value of x_i is given by

$$s(x_i) = \begin{cases} 1 - \dfrac{a(x_i)}{b(x_i)} & \text{if } a(x_i) < b(x_i), \\ \dfrac{b(x_i)}{a(x_i)} - 1 & \text{if } a(x_i) > b(x_i), \\ 0 & \text{otherwise,} \end{cases} \tag{2.16}$$

which ranges from -1 to 1 and quantifies to what extent x_i matches the current cluster with respect to the best alternative. In the special case that x_i is the sole data point in C_k, $a(x_i)$ is undefined according to (2.14) and $s(x_i)$ is neutrally set to 0.

The silhouette index can be computed from the silhouette values of individual data points in two ways. One is to simply take the average of all the values over the entire data set:

$$\bar{s}_\mathrm{P} = \frac{1}{M} \sum_{i=1}^{M} s(x_i). \tag{2.17}$$

The other is to average first the values of data points in each cluster and then the results across all the clusters:

$$\bar{s}_{\mathrm{C}} = \frac{1}{K}\sum_{i=1}^{K}\frac{1}{|C_k|}\sum_{x_i \in C_k} s(x_i). \tag{2.18}$$

2.3 Queueing Theory

We review in this section the fundamentals of queueing theory, which underlies the deployment optimization and fluid approximation approaches proposed in Chapters 4 and 5. Queueing theory is a mathematical study of various models for capturing the queueing behavior of a system. Our focus is particularly on queueing networks and their layered extension.

2.3.1 Phase-Type Distributions

The *exponential* distribution is the most important distribution for performance modeling purposes. The temporal specification of a performance model, including the service time in a queue and the firing time of a transition, is often represented exactly or approximately using the exponential distribution.

A continuous random variable X is said to be exponentially distributed with rate λ ($\lambda > 0$), if its *probability density function* (PDF) is given by

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.19}$$

or equivalently if its *cumulative distribution function* (CDF) is given by

$$F(x) = \begin{cases} 1 - e^{-\lambda x} & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.20}$$

Accordingly, the *expectation*, *variance* and *coefficient of variation* of the exponential distribu-

tion are

$$\mathbb{E}(X) = \frac{1}{\lambda}, \tag{2.21}$$

$$\mathrm{Var}(X) = \frac{1}{\lambda^2}, \tag{2.22}$$

$$\mathrm{CV}(X) = 1. \tag{2.23}$$

The importance of the exponential distribution is based on the fact that it possesses several desirable properties:

- If X is an exponential random variable, then

$$\mathbb{P}(X \leq s + t \mid X > t) = \mathbb{P}(X \leq s) \quad \text{for } s \geq 0, t \geq 0. \tag{2.24}$$

  Suppose that X represents the waiting time until an event. (2.24) implies that if the event has yet occurred in the first t time units, the probability that it will occur in the next s time units is the same as if t had never elapsed. This is called the *memoryless property*. The exponential distribution is the only memoryless continuous distribution.
- If $X_1, X_2, \ldots, X_n$ are n independent exponential random variables with respective rates $\lambda_1, \lambda_2, \ldots, \lambda_n$, then the random variable $Y = \min_{i=1}^{n} X_i$ is exponentially distributed with rate $\sum_{i=1}^{n} \lambda_i$ and the probability that $X_j = \min_{i=1}^{n} X_i$ is $\lambda_j / \sum_{i=1}^{n} \lambda_i$.
- If the sequence of interarrival times between consecutive events are independent and identically distributed exponential random variables with rate λ, then the total number $N(t)$ of events that occur in time interval $[0, t]$ $(t \geq 0)$ forms a Poisson Process with rate λ, and vice versa.
- If n independent Poisson processes with respective rates $\lambda_1, \lambda_2, \ldots, \lambda_n$ are merged into a single process, then the resulting process is a Poisson process with rate $\sum_{i=1}^{n} \lambda_i$.
- If a Poisson process with rate λ is split into n subprocesses so that each event is assigned to the ith subprocess with probability p_i $(\sum_{i=1}^{n} p_i = 1)$, then the resulting subprocesses are independent Poisson processes with respective rates $\lambda p_1, \lambda p_2, \ldots, \lambda p_n$.

Although these properties make the exponential distribution mathematically tractable, it is not a suitable approximation to an actual distribution with a coefficient of variation greater or

less than one. In this case, a phase-type distribution, such as the hyperexponential, Erlang or hypoexponential distribution, should be considered.

The hyperexponential distribution can be used to approximate an empirical distribution with a coefficient of variation greater than one. A hyperexponential time span is shown in Figure 2.2. It consists of k parallel exponential phases with respective rates $\lambda_1, \lambda_2, \ldots, \lambda_k$, and the probability of the ith phase being chosen as the current one is p_i ($\sum_{i=1}^{k} p_i = 1$).

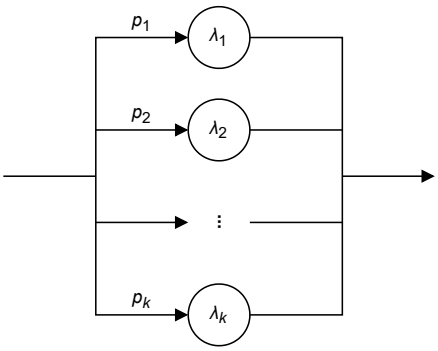


Figure 2.2: A hyperexponential time span with k phases.

The hyperexponential distribution is defined by the following PDF and CDF:

$$f(x) = \sum_{i=1}^{k} p_i \lambda_i e^{-\lambda_i x} \quad \text{for } x \geq 0, \tag{2.25}$$

$$F(x) = 1 - \sum_{i=1}^{k} p_i e^{-\lambda_i x} \quad \text{for } x \geq 0. \tag{2.26}$$

In addition, its expectation, variance and coefficient of variation are given by

$$\mathbb{E}(X) = \sum_{i=1}^{k} \frac{p_i}{\lambda_i}, \tag{2.27}$$

$$\mathrm{Var}(X) = 2\sum_{i=1}^{k} \frac{p_i}{\lambda_i^2} - \left(\sum_{i=1}^{k} \frac{p_i}{\lambda_i}\right)^2, \tag{2.28}$$

$$\mathrm{CV}(X) = \sqrt{\frac{2\sum_{i=1}^{k} \frac{p_i}{\lambda_i^2}}{\left(\sum_{i=1}^{k} \frac{p_i}{\lambda_i}\right)^2} - 1} \geq 1. \tag{2.29}$$

An empirical distribution with a coefficient of variation less than one can be approximated using the Erlang or hypoexponential distribution. Figure 2.3 shows an Erlang or hypoexponential

time span, which is composed of k exponential phases connected in series. If the rates of all the phases are identical, i.e. $\lambda_i = \lambda_j$ for every $i, j \in [1\,..\,k]$, an Erlang time span is obtained; otherwise if the rates are pairwise different, i.e. $\lambda_i \neq \lambda_j$ for every $i, j \in [1\,..\,k]$ and $i \neq j$, the time span is hypoexponential.



Figure 2.3: An Erlang or hypoexponential time span with k phases.

The PDF and CDF of the Erlang distribution are

$$f(x) = \frac{\lambda e^{-\lambda x}(\lambda x)^{k-1}}{(k-1)!} \quad \text{for } x \geq 0, \tag{2.30}$$

$$F(x) = 1 - e^{-\lambda x} \sum_{i=0}^{k-1} \frac{(\lambda x)^i}{i!} \quad \text{for } x \geq 0. \tag{2.31}$$

Further, the numerical characteristics of the Erlang distribution are as follows:

$$\mathbb{E}(X) = \frac{k}{\lambda}, \tag{2.32}$$

$$\mathrm{Var}(X) = \frac{k}{\lambda^2}, \tag{2.33}$$

$$\mathrm{CV}(X) = \sqrt{\frac{1}{k}} \leq 1. \tag{2.34}$$

The PDF and CDF of the hypoexponential distribution can be written as

$$f(x) = \sum_{i=1}^{k} \left(\prod_{j=1, j\neq i}^{k} \frac{\lambda_j}{\lambda_j - \lambda_i} \right) \lambda_i e^{-\lambda_i x} \quad \text{for } x \geq 0, \tag{2.35}$$

$$F(x) = 1 - \sum_{i=1}^{k} \left(\prod_{j=1, j\neq i}^{k} \frac{\lambda_j}{\lambda_j - \lambda_i} \right) e^{-\lambda_i x} \quad \text{for } x \geq 0. \tag{2.36}$$

Furthermore, the hypoexponential distribution has the following numerical characteristics:

$$\mathbb{E}(X) = \sum_{i=1}^{k} \frac{1}{\lambda_i}, \tag{2.37}$$

$$\mathrm{Var}(X) = \sum_{i=1}^{k} \frac{1}{\lambda_i^2}, \tag{2.38}$$

$$\mathrm{CV}(X) = \sqrt{\frac{\sum_{i=1}^{k} \frac{1}{\lambda_i^2}}{\left(\sum_{i=1}^{k} \frac{1}{\lambda_i}\right)^2}} \leq 1. \tag{2.39}$$

2.3.2 Continuous-Time Markov Chains

Markov processes are a powerful and general tool for analyzing performance models. In principle, any performance model can be mapped onto an equivalent Markov process and then mathematically solved in terms of that process. Apart from measures of interest, the underlying Markov process may also be used to determine the fundamental properties of the performance model.

A *stochastic process* $\{X(t) \mid t \in T\}$ is a collection of random variables $X(t)$ indexed by parameter $t \in T$, which is often interpreted as time if the *parameter space* $T \in \mathbb{R}_{\geq 0}$. The set S of all possible values that $X(t)$ can assume is referred to as the *state space* of the stochastic process.

A stochastic process is said to be *Markovian*, or possess the *Markov property*, if the conditional CDF of $X(t_{n+1})$ given the observed values $X(t_n)$, $X(t_{n-1})$, ..., $X(t_0)$ depends only on the last value $X(t_n)$ and not on the earlier values $X(t_{n-1})$, $X(t_{n-2})$, ..., $X(t_0)$:

$$\begin{aligned} &\mathbb{P}(X(t_{n+1}) \leq s_{n+1} \mid X(t_n) = s_n, X(t_{n-1}) = s_{n-1}, \ldots, X(t_0) = s_0) \\ =&\mathbb{P}(X(t_{n+1}) \leq s_{n+1} \mid X(t_n) = s_n). \end{aligned} \tag{2.40}$$

The above gives the general definition of a Markov process. Our interest is particularly in *continuous-time Markov chains* (CTMCs), of which the parameter space T is a continuous subset of $\mathbb{R}_{\geq 0}$ and the state space S is a countable set. Moreover, since the target systems often exhibit a time-independent pattern of dynamic behavior, the investigation here is restricted to *time-homogeneous* CTMCs.

A Markov process is said to be *time-homogeneous*, or possess the *time homogeneity*, if the conditional CDF of $X(t_{n+1})$ given the observed value $X(t_n)$ depends only on the time difference

$t_{n+1} - t_n$ and not on the observation time t_n:

$$\mathbb{P}(X(t_{n+1}) \leq s_{n+1} \mid X(t_n) = s_n) = \mathbb{P}(X(t_{n+1} - t_n) \leq s_{n+1} \mid X(0) = s_n). \tag{2.41}$$

Let $\pi_i(t)$ denote the *transient state probability* that the CTMC is in state i at time t, and $p_{i,j}(t)$ the *transition probability* that the CTMC travels from state i to state j during the period of time $[0, t]$:

$$\pi_i(t) = \mathbb{P}(X(t) = i), \tag{2.42}$$

$$p_{i,j}(t) = \mathbb{P}(X(t) = j \mid X(0) = i). \tag{2.43}$$

By conditioning on the initial state i, we get

$$\begin{aligned} \pi_j(t) &= \mathbb{P}(X(t) = j) \\ &= \sum_{i \in S} \mathbb{P}(X(0) = i)\mathbb{P}(X(t) = j \mid X(0) = i) \\ &= \sum_{i \in S} \pi_i(0) p_{i,j}(t). \end{aligned} \tag{2.44}$$

Upon conditioning on an intermediate state k, it follows from the Markov property (2.40) and the time homogeneity (2.41) that

$$\begin{aligned} p_{i,j}(u+v) &= \mathbb{P}(X(u+v) = j \mid X(0) = i) \\ &= \sum_{k \in S} \mathbb{P}(X(u) = k \mid X(0) = i)\mathbb{P}(X(u+v) = j \mid X(u) = k, X(0) = i) \\ &= \sum_{k \in S} \mathbb{P}(X(u) = k \mid X(0) = i)\mathbb{P}(X(u+v) = j \mid X(u) = k) \\ &= \sum_{k \in S} \mathbb{P}(X(u) = k \mid X(0) = i)\mathbb{P}(X(v) = j \mid X(0) = k) \\ &= \sum_{k \in S} p_{i,k}(u) p_{k,j}(v). \end{aligned} \tag{2.45}$$

(2.45) is known as the *Chapman-Kolmogorov equation.*

For each pair of states $i, j \in S$, define the *instantaneous transition rate*:

$$q_{i,j} = \lim_{\Delta t \to 0} \frac{p_{i,j}(\Delta t) - p_{i,j}(0)}{\Delta t}. \tag{2.46}$$

It can be shown that $q_{i,j}$ exists under rather general conditions. Since $\sum_{j \in S} p_{i,j}(\Delta t) = 1$ and

$\sum_{j\in S} p_{i,j}(0) = 1$,

$$\begin{aligned}\sum_{j\in S} q_{i,j} &= \sum_{j\in S} \lim_{\Delta t\to 0} \frac{p_{i,j}(\Delta t) - p_{i,j}(0)}{\Delta t} \\ &= \lim_{\Delta t\to 0} \frac{\sum\limits_{j\in S} p_{i,j}(\Delta t) - \sum\limits_{j\in S} p_{i,j}(0)}{\Delta t} \\ &= \lim_{\Delta t\to 0} \frac{1-1}{\Delta t} \\ &= 0.\end{aligned} \tag{2.47}$$

Substituting (u, v) with $(t, \Delta t)$ and $(t, 0)$ in the Chapman-Kolmogorov equation (2.45) gives

$$p_{i,j}(t+\Delta t) = \sum_{k\in S} p_{i,k}(t) p_{k,j}(\Delta t), \tag{2.48}$$

$$p_{i,j}(t) = \sum_{k\in S} p_{i,k}(t) p_{k,j}(0). \tag{2.49}$$

Subtracting (2.49) from (2.48), dividing both sides by Δt, and taking the limit as $\Delta t \to 0$ on both sides, we derive the *Kolmogorov forward equation*:

$$\begin{aligned}\frac{dp_{i,j}(t)}{dt} &= \lim_{\Delta t\to 0} \frac{p_{i,j}(t+\Delta t) - p_{i,j}(t)}{\Delta t} \\ &= \lim_{\Delta t\to 0} \frac{\sum\limits_{k\in S} p_{i,k}(t) p_{k,j}(\Delta t) - \sum\limits_{k\in S} p_{i,k}(t) p_{k,j}(0)}{\Delta t} \\ &= \sum_{k\in S} p_{i,k}(t) \lim_{\Delta t\to 0} \frac{p_{k,j}(\Delta t) - p_{k,j}(0)}{\Delta t} \\ &= \sum_{k\in S} p_{i,k}(t) q_{k,j}.\end{aligned} \tag{2.50}$$

Differentiating both sides of (2.44), and applying the Kolmogorov forward equation (2.50) and again (2.44), yields

$$\begin{aligned}\frac{d\pi_j(t)}{dt} &= \frac{d\sum\limits_{i\in S} \pi_i(0) p_{i,j}(t)}{dt} \\ &= \sum_{i\in S} \pi_i(0) \sum_{k\in S} p_{i,k}(t) q_{k,j} \\ &= \sum_{k\in S} \sum_{i\in S} \pi_i(0) p_{i,k}(t) q_{k,j} \\ &= \sum_{k\in S} \pi_k(t) q_{k,j}.\end{aligned} \tag{2.51}$$

(2.51) is preferably written in the vector-matrix form:

$$\frac{d\boldsymbol{\pi}(t)}{dt} = \boldsymbol{\pi}(t)\boldsymbol{Q}, \tag{2.52}$$

where $\boldsymbol{\pi}(t) = (\pi_j) \in \mathbb{R}^{1\times|S|}$ and $\boldsymbol{Q} = [q_{i,j}] \in \mathbb{R}^{|S|\times|S|}$. $\boldsymbol{Q}$ is referred to as the infinitesimal generator matrix. Given the initial value $\boldsymbol{\pi}(0)$, the transient state probability vector $\boldsymbol{\pi}(t)$ can be solved from (2.52) using the matrix exponential:

$$\boldsymbol{\pi}(t) = \boldsymbol{\pi}(0)e^{\boldsymbol{Q}t}. \tag{2.53}$$

The long-term measures and properties are often of the main concern in the analysis of a performance model, because the undesirable bias introduced by the initial state probabilities will vanish over an extended period of time. For determining these measures and properties, the *steady-state probability vector* is essential.

A vector $\boldsymbol{\pi} = (\pi_j) \in \mathbb{R}^{1\times|S|}$ is said to be the (unique) steady-state probability vector of a CTMC if

- as $t \to \infty$, the transient state probability vector $\boldsymbol{\pi}(t)$ converges independently of the initial state probability vector $\boldsymbol{\pi}(0)$ to $\boldsymbol{\pi}$:

$$\lim_{t\to\infty} \boldsymbol{\pi}(t) = \boldsymbol{\pi}; \tag{2.54}$$

- π_j is strictly positive for every state $j \in S$, and $\boldsymbol{\pi}$ satisfies the normalization condition:

$$\boldsymbol{\pi}\mathbf{1}^T = 1, \tag{2.55}$$

  where $\mathbf{1} \in \mathbb{N}^{1\times|S|}$ is a row vector of all ones.

Taking the limit as $t \to \infty$ on both sides of (2.52), we obtain from (2.54) that

$$\boldsymbol{\pi}\boldsymbol{Q} = \lim_{t\to\infty} \boldsymbol{\pi}(t)\boldsymbol{Q} = \lim_{t\to\infty} \frac{d\boldsymbol{\pi}(t)}{dt} = \lim_{t\to\infty} \frac{\boldsymbol{\pi}(t)}{t} = \lim_{t\to\infty} \boldsymbol{\pi}(t) \cdot \lim_{t\to\infty} \frac{1}{t} = \boldsymbol{\pi} \cdot 0 = \mathbf{0}, \tag{2.56}$$

where $\mathbf{0} \in \mathbb{N}^{1\times|S|}$ is a row vector of all zeros. The steady-state probability vector $\boldsymbol{\pi}$ can be solved from (2.56) under the normalization condition (2.55).

Let $S_i(\tau)$ denote the sojourn time that the CTMC spends in state i after being observed there at time τ. Applying the time homogeneity (2.41) gives

$$\begin{aligned} \mathbb{P}(S_i(\tau) > s) &= \mathbb{P}(X(t) = i \text{ for } \tau < t \leq \tau + s \mid X(\tau) = i) \\ &= \mathbb{P}(X(t) = i \text{ for } 0 < t \leq s \mid X(0) = i). \end{aligned} \tag{2.57}$$

(2.57) implies that the distribution of $S_i(\tau)$ is independent of τ. Without loss of generality, replace $S_i(\tau)$ by S_i, and rewrite (2.57):

$$\mathbb{P}(S_i > s) = \mathbb{P}(X(t) = i \text{ for } 0 < t \leq s \mid X(0) = i). \tag{2.58}$$

From (2.58), the Markov property (2.40) and the time homogeneity (2.41), it follows that

$$\begin{aligned}
\mathbb{P}(S_i > u + v \mid S_i > u) &= \mathbb{P}(X(t) = i \text{ for } 0 < t \leq u + v \mid X(t) = i \text{ for } 0 \leq t \leq u) \\
&= \mathbb{P}(X(t) = i \text{ for } 0 < t \leq u + v \mid X(u) = i) \\
&= \mathbb{P}(X(t) = i \text{ for } 0 < t \leq v \mid X(0) = i) \\
&= \mathbb{P}(S_i > v).
\end{aligned} \tag{2.59}$$

Comparing (2.59) with (2.24), we see that the distribution of S_i is memoryless. As the exponential distribution is the only continuous distribution with the memoryless property, S_i is necessarily exponentially distributed. When a transition out of state i occurs, the CTMC enters new state j with some probability. As a result, for each pair of states $i, j \in S$ and $i \neq j$, the *transition time* $T_{i,j}$ that the CTMC travels from state i to new state j is an independent exponential random variable.

Let $\lambda_{i,j}$ denote the *transition rate* that the CTMC travels from state i to new state j. The distribution of the sojourn time $S_i = \min_{j \in S, j \neq i} T_{i,j}$ is exponential with rate $\sum_{j \in S, j \neq i} \lambda_{i,j}$, and the probability that $T_{i,k} = \min_{j \in S, j \neq i} T_{i,j}$ is $\lambda_{i,k} / \sum_{j \in S, j \neq i} \lambda_{i,j}$. Therefore, conditioning on whether $S_i \leq \Delta t$ yields

$$\begin{aligned}
p_{i,k}(\Delta t) &= \mathbb{P}(X(\Delta t) = k \mid X(0) = i) \\
&= \mathbb{P}(S_i \leq \Delta t)\mathbb{P}\left(T_{i,k} = \min_{j \in S, j \neq i} T_{i,j}\right) + o(\Delta t) \\
&= \left(1 - e^{-\sum_{j \in S, j \neq i} \lambda_{i,j} \Delta t}\right) \frac{\lambda_{i,k}}{\sum_{j \in S, j \neq i} \lambda_{i,j}} + o(\Delta t) \\
&= \left(1 - \left(1 - \sum_{j \in S, j \neq i} \lambda_{i,j} \Delta t + o(\Delta t)\right)\right) \frac{\lambda_{i,k}}{\sum_{j \in S, j \neq i} \lambda_{i,j}} + o(\Delta t) \\
&= \lambda_{i,k} \Delta t + o(\Delta t) \quad \text{for } i \neq k,
\end{aligned} \tag{2.60}$$

where $o(\Delta t)$ is an infinitesimal of higher order than Δt. As $p_{i,k}(0) = 0$ for $i \neq k$, from (2.46)

and (2.60), we have

$$q_{i,k} = \lim_{\Delta t \to 0} \frac{p_{i,k}(\Delta t) - p_{i,k}(0)}{\Delta t} = \lim_{\Delta t \to 0} \frac{\lambda_{i,k}\Delta t + o(\Delta t) - 0}{\Delta t} = \lambda_{i,k} \quad \text{for } i \neq k. \tag{2.61}$$

Further, applying (2.47) and (2.61) gives

$$q_{i,i} = -\sum_{j \in S, j \neq i} q_{i,j} = -\sum_{j \in S, j \neq i} \lambda_{i,j}. \tag{2.62}$$

2.3.3 Queueing Networks

Queueing networks (QNs) are a popular modeling formalism for performance analysis of computer and communication systems. As opposed to a single-station queueing model, a QN consists of multiple interconnected *stations* that represent individual components critical to the performance of the system under study. Two types of stations, *delay* and *queueing*, are often used as building blocks of QNs. A delay station, also known as an *infinite-server* (IS) station, is equipped with an unlimited number of servers. By contrast, a queueing station possesses finite servers and manages them according to a predefined scheduling discipline. The servers at a station typically characterize resources provided by the corresponding component. In QNs, a workload is depicted as a *class of jobs* that move from one station to another and demand a certain amount of service time at each. A class with an endless stream of arriving jobs is called an *open class*, whilst one with a fixed population of resident jobs is referred to as a *closed class*. QNs can be divided into three categories, namely *open*, *closed* and *mixed*, depending on whether they contain only open, only closed or both types of job classes. Figures 2.4 and 2.5 give examples of an open and a closed QN, respectively. The open QN is composed of two parallel queueing stations that serve continuously arriving jobs, which then

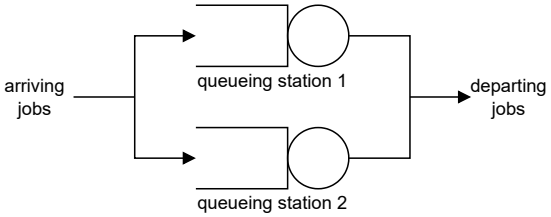


Figure 2.4: An example of an open QN.

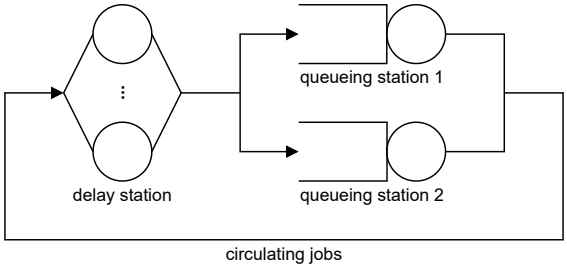


Figure 2.5: An example of a closed QN.

depart from the model upon service completion. Besides two parallel queueing stations, the closed QN comprises one more delay station, and jobs resident in the model circulate around the former and the latter.

We are particularly interested in a subset of general QNs investigated by Baskett, Chandy, Muntz and Palacios [BCMP75]. *BCMP networks* make relatively weak assumptions and possess several desirable properties, thus being not only sufficient for use but also mathematically tractable. Stations in a BCMP network are of the following four types:

- *–/M/*1*-FCFS.* This is a single-server queueing station that adopts the *first-come-first-served* (FCFS) discipline. Jobs are served in the order that they arrive. The service time of every job is exponentially distributed with the same rate regardless of its class.
- *–/G/*1*-PS.* There is also a single server at this station. Jobs are served simultaneously under the *processor-sharing* (PS) discipline. If n jobs are currently present at the station, each one occupies a fraction $1/n$ of the service capacity.
- *–/G/*∞*.* This type corresponds to the aforementioned delay (or IS) station, which holds an infinite number of servers and assigns a dedicated server to each incoming job.
- *–/G/*1*-LCFS PR.* There is a single server at the station. An arriving job instantly begins to receive service, preempting the one being served at the moment. Upon service completion of a job, the most recently preempted one resumes service at the point where it was interrupted. The implemented scheduling discipline is known as *last-come-first-served preemptive-resume* (LCFS PR).

For the last three types of stations, the service time distributions must have a rational Laplace transform, e.g. phase-type distributions. The service rates of jobs may be load-dependent at any station. Both open and closed classes are allowed in a BCMP network. The arrival processes of the open classes are Poisson with possibly load-dependent rates. When traveling among stations, a job can switch from one class to another. This is termed *class switching*. In the rest of the section, we restrict our attention to load-independent QNs without class switching. Load-dependent QNs are beyond the scope of this thesis. However, contents presented below are extensible to QNs with class switching by applying the *concept of chains* (see Section 5.5.1).

The state of a BCMP network with M stations and K classes can be defined as a vector $\boldsymbol{n} = (n_{i,k}) \in \mathbb{N}^{MK}$ whose (i,k)th entry is the number of class-k jobs at station i. The steady-state probability of the model being in state $\boldsymbol{n}$ has the following form:

$$\pi(\boldsymbol{n}) = \frac{\prod_{i=1}^{M} F_i(\boldsymbol{n}_i)}{G}, \tag{2.63}$$

where $F_i(\boldsymbol{n}_i)$ is a factor that only depends on the state $\boldsymbol{n}_i = (n_{i,k}) \in \mathbb{N}^K$ of station i, and G is a normalizing constant chosen such that the steady-state probabilities sum up to 1. Let $\mathcal{D}$ and $\mathcal{Q}$ be the sets of delay and queueing stations, respectively. For an open BCMP network,

$$F_i(\boldsymbol{n}_i) = \begin{cases} \prod_{k=1}^{K} \frac{U_{i,k}^{n_{i,k}}}{n_{i,k}!} & \text{if } i \in \mathcal{D}, \\ \left(\sum_{k=1}^{K} n_{i,k}\right)! \prod_{k=1}^{K} \frac{U_{i,k}^{n_{i,k}}}{n_{i,k}!} & \text{otherwise,} \end{cases} \tag{2.64}$$

where $U_{i,k}$ is the mean utilization of class-k jobs at station i. As for a closed BCMP network,

$$F_i(\boldsymbol{n}_i) = \begin{cases} \prod_{k=1}^{K} \frac{D_{i,k}^{n_{i,k}}}{n_{i,k}!} & \text{if } i \in \mathcal{D}, \\ \left(\sum_{k=1}^{K} n_{i,k}\right)! \prod_{k=1}^{K} \frac{D_{i,k}^{n_{i,k}}}{n_{i,k}!} & \text{otherwise,} \end{cases} \tag{2.65}$$

where $D_{i,k}$ is the mean service demand of class-k jobs at station i. As shown in (2.63), (2.64) and (2.65), the steady-state probabilities of the model are resolvable into multiplicative factors of which each involves only the state of a single station. This implies that stations in a BCMP network behave as if they were single-station queueing models and can thus be analyzed in isolation from one another. QNs with such a property are called *separable* or *product-form* QNs.

The direct use of (2.63) requires taking into account all the possible states so as to obtain the normalizing constant, which is an extremely cumbersome procedure for a closed QN because the total number of the states grows exponentially with the scale of the model. More efficient techniques have been developed to address this problem. Next, we describe the *mean value analysis* (MVA)[22] algorithms popularized by the classic system performance book of Lazowska

[22]The term "MVA" normally appears in the context of closed QNs. For conciseness, we refer to the technique that solves a model for the mean values of performance measures without explicitly computing the normalizing constant as MVA, no matter whether the workload type is open, closed or mixed.

et al. [LZGS84]. These algorithms evaluate the mean performance measures of open, closed and mixed QNs straightforwardly without the normalizing constant. Tables 2.4 and 2.5 summarize the inputs and outputs of the MVA algorithms, respectively. In addition, we use a vector with an entry for each class to represent the workload intensity of a given QN: $\boldsymbol{\lambda} = (\lambda_k) \in \mathbb{R}^K$ if the model is open; $\boldsymbol{N} = (N_k) \in \mathbb{N}^K$ if the model is closed; $\boldsymbol{I} = (I_k) \in \mathbb{R}^K$, where I_k is either λ_k or N_k depending on whether k indexes an open or a closed class, otherwise.

Table 2.4: The inputs of the MVA algorithms.

Input	**Description**
Number of stations: M	Number of stations
Number of classes: K	Number of classes
Set of delay stations: $\mathcal{D}$	Collection of indices to all the delay stations
Set of queueing stations: $\mathcal{Q}$	Collection of indices to all the queueing stations
Set of open classes: $\mathcal{O}$	Collection of indices to all the open classes
Set of closed classes: $\mathcal{C}$	Collection of indices to all the closed classes
Arrival rate: ${\lambda_k}^*$	Arrival rate of class-k jobs in the model
Population: ${N_k}^\dagger$	Total number of class-k jobs in the model
Mean think time: ${Z_k}^\dagger$	Average time between the completion and next submission of a class-k job
Mean service demand: $D_{i,k}$	Average overall service time required by a class-k job at station i

*† These values are exclusive to the parameterizations for open and closed classes, respectively.

Table 2.5: The outputs of the MVA algorithms.

Output	**Description**
Mean utilization: $U_{i,k}$	Average server usage by class-k jobs at station i
Mean queue length: $Q_{i,k}$	Average number of class-k jobs at station i
Mean residence time: $Rd_{i,k}$	Average overall sojourn time spent by a class-k job at station i
Mean system queue length: Q_k	Average number of class-k jobs in the system
Mean system throughput: X_k	Average completion rate of class-k jobs in the system
Mean system response time: Rp_k	Average time between the submission and completion of a class-k job

The MVA algorithms for open, closed and mixed QNs differ in detail. To help with the understanding, we append the parenthesized workload intensity to an output quantity if it is associated with a particular workload type or value, and introduce in advance two fundamental principles that the MVA algorithms are based on:

- *Little's law.* For a stable system, the mean queue length is equal to the mean throughput times the mean response time. Applying this relation to a QN at the server, station and system levels respectively results in

$$U_{i,k} = X_k D_{i,k}, \tag{2.66}$$

$$Q_{i,k} = X_k R d_{i,k}, \tag{2.67}$$

$$Q_k = X_k R p_k. \tag{2.68}$$

  (2.66) is also known as the *utilization law.*

- *Arrival theorem.* For a product-form QN, the joint queue length distribution seen by a job upon arrival at a station is the same as the joint queue length distribution at any point in time with the job removed. If the model is open, we have

$$A_{i,k}(\boldsymbol{\lambda}) = Q_i(\boldsymbol{\lambda}), \tag{2.69}$$

  where $A_{i,k}(\boldsymbol{\lambda})$ and $Q_i(\boldsymbol{\lambda})$ are the mean queue lengths of jobs at station i upon arrival of a class-k job and at an arbitrary time respectively. If the model is closed, it follows that

$$A_{i,k}(\boldsymbol{N}) = Q_i(\boldsymbol{N} - \boldsymbol{e}_k), \tag{2.70}$$

  where $A_{i,k}(\boldsymbol{N})$ and $Q_i(\boldsymbol{N} - \boldsymbol{e}_k)$ are the mean queue lengths of jobs at station i upon arrival of a class-k job and at an arbitrary time with one less class-k job respectively.

Besides, the following simple relations between the aggregate and specific performance measures are also useful in the development of the MVA algorithms:

$$Q_i = \sum_{k=1}^{K} Q_{i,k}, \tag{2.71}$$

$$R p_k = \sum_{i=1}^{M} R d_{i,k}. \tag{2.72}$$

A closed-form solution exists in the case of open QNs due to the fact that the mean system throughput is already available as the arrival rate for each class k:

$$X_k(\boldsymbol{\lambda}) = \lambda_k. \tag{2.73}$$

This enables us to immediately unlock the mean utilization of class-k jobs at station i with (2.66). The mean residence time of a class-k job at station i is given by

$$Rd_{i,k}(\boldsymbol{\lambda}) = \begin{cases} D_{i,k} & \text{if } i \in \mathcal{D}, \\ D_{i,k}(1 + A_{i,k}(\boldsymbol{\lambda})) & \text{otherwise.} \end{cases} \tag{2.74}$$

It is easy to show (2.74) for delay and FCFS stations by splitting the mean residence time of a job into the waiting and the service part. The latter piece of (2.74) is also true for PS and LCFS PR stations, because the steady-state probabilities of an open QN are independent of scheduling disciplines adopted by the queueing stations as in (2.63) and (2.64). By applying (2.74), (2.69), (2.71) and (2.67), noticing that $D_{i,k}Rd_{i,l}(\boldsymbol{\lambda}) = D_{i,l}Rd_{i,k}(\boldsymbol{\lambda})$, and further applying (2.66), we derive

$$\begin{aligned} Rd_{i,k}(\boldsymbol{\lambda}) &= D_{i,k}(1 + A_{i,k}(\boldsymbol{\lambda})) = D_{i,k}(1 + Q_i(\boldsymbol{\lambda})) = D_{i,k}\left(1 + \sum_{l=1}^{K} Q_{i,l}(\boldsymbol{\lambda})\right) \\ &= D_{i,k}\left(1 + \sum_{l=1}^{K} Q_{i,l}(\boldsymbol{\lambda})\right) = D_{i,k}\left(1 + \sum_{l=1}^{K} X_l(\boldsymbol{\lambda}) Rd_{i,l}(\boldsymbol{\lambda})\right) \\ &= D_{i,k} + \sum_{l=1}^{K} X_l(\boldsymbol{\lambda}) D_{i,k} Rd_{i,l}(\boldsymbol{\lambda}) = D_{i,k} + \sum_{l=1}^{K} X_l(\boldsymbol{\lambda}) D_{i,l} Rd_{i,k}(\boldsymbol{\lambda}) \\ &= D_{i,k} + \sum_{l=1}^{K} U_{i,l}(\boldsymbol{\lambda}) Rd_{i,k}(\boldsymbol{\lambda}) = \frac{D_{i,k}}{1 - \sum_{l=1}^{K} U_{i,l}(\boldsymbol{\lambda})} \quad \text{for } i \in \mathcal{Q}. \end{aligned} \tag{2.75}$$

Combining (2.74) and (2.75) leads to

$$Rd_{i,k}(\boldsymbol{\lambda}) = \begin{cases} D_{i,k} & \text{if } i \in \mathcal{D}, \\ \dfrac{D_{i,k}}{1 - \sum_{l=1}^{K} U_{i,l}(\boldsymbol{\lambda})} & \text{otherwise.} \end{cases} \tag{2.76}$$

Once the mean residence time is known, (2.67) can be used to obtain the mean queue length of class-k jobs at station i. Finally, we determine the mean system response time and the mean system queue length for each class k according to (2.72) and (2.68) respectively. Algorithm 2.5 summarizes the MVA algorithm for open QNs [DB78].

Algorithm 2.5: The MVA algorithm for open QNs [DB78].

Input:

- Number of stations: M
- Number of classes: K
- Set of delay stations: $\mathcal{D}$
- Set of queueing stations: $\mathcal{Q}$
- Arrival rate: λ_k
- Mean service demand: $D_{i,k}$

Output:

- Mean utilization: $U_{i,k}(\boldsymbol{\lambda})$
- Mean queue length: $Q_{i,k}(\boldsymbol{\lambda})$
- Mean residence time: $Rd_{i,k}(\boldsymbol{\lambda})$
- Mean system queue length: $Q_k(\boldsymbol{\lambda})$
- Mean system throughput: $X_k(\boldsymbol{\lambda})$
- Mean system response time: $Rp_k(\boldsymbol{\lambda})$

1: For every class k, compute the mean system throughput $X_k(\boldsymbol{\lambda})$ from (2.73)
2: For every station i and every class k, compute the mean utilization $U_{i,k}(\boldsymbol{\lambda})$ from (2.66)
3: For every station i and every class k, compute the mean residence time $Rd_{i,k}(\boldsymbol{\lambda})$ and the mean queue length $Q_{i,k}(\boldsymbol{\lambda})$ from (2.76) and (2.67) respectively
4: For every class k, compute the mean system response time $Rp_k(\boldsymbol{\lambda})$ and the mean system queue length $Q_k(\boldsymbol{\lambda})$ from (2.72) and (2.68) respectively

In the case of closed QNs, the mean residence time of a class-k job at station i can be written as

$$Rd_{i,k}(\boldsymbol{N}) = \begin{cases} D_{i,k} & \text{if } i \in \mathcal{D}, \\ D_{i,k}(1 + A_{i,k}(\boldsymbol{N})) & \text{otherwise.} \end{cases} \tag{2.77}$$

The justification of (2.77) is similar to that of (2.74). From (2.77), (2.70) and (2.71), we get

$$\begin{aligned} Rd_{i,k}(\boldsymbol{N}) &= D_{i,k}(1 + A_{i,k}(\boldsymbol{N})) = D_{i,k}(1 + Q_i(\boldsymbol{N} - \boldsymbol{e}_k)) \\ &= D_{i,k}\left(1 + \sum_{l=1}^{K} Q_{i,l}(\boldsymbol{N} - \boldsymbol{e}_k)\right) \quad \text{for } i \in \mathcal{Q}. \end{aligned} \tag{2.78}$$

Combining (2.77) and (2.78) results in

$$Rd_{i,k}(\boldsymbol{N}) = \begin{cases} D_{i,k} & \text{if } i \in \mathcal{D}, \\ D_{i,k}\left(1 + \sum_{l=1}^{K} Q_{i,l}(\boldsymbol{N} - \boldsymbol{e}_k)\right) & \text{otherwise.} \end{cases} \tag{2.79}$$

The Little's law is additionally applicable to a closed QN at the model level:

$$N_k = X_k(\boldsymbol{N})(Z_k + Rp_k(\boldsymbol{N})). \tag{2.80}$$

Solving (2.80) for the mean system throughput of class-k jobs and then applying (2.72) yield

$$X_k(\boldsymbol{N}) = \frac{N_k}{Z_k + Rp_k(\boldsymbol{N})} = \frac{N_k}{Z_k + \sum_{i=1}^{M} Rd_{i,k}(\boldsymbol{N})}. \tag{2.81}$$

(2.79), (2.81) and (2.67) allow us to determine the mean residence times, the mean system throughputs and the mean queue lengths of a closed QN with a workload intensity $\boldsymbol{N}$, given its mean queue lengths under the workload intensity $\boldsymbol{N} - \boldsymbol{e}_k$ for each class k. As shown in Algorithm 2.6, the MVA algorithm for closed QNs [RL80] is based on the iterative application of the three equations, starting from the trivial case where there are no jobs and incrementing the workload intensity until it reaches the specified value. After the algorithm terminates, other performance measures can be evaluated using (2.66), (2.72) and (2.68).

Algorithm 2.6: The MVA algorithm for closed QNs [RL80].

Input:

- Number of stations: M
- Number of classes: K
- Set of delay stations: $\mathcal{D}$
- Set of queueing stations: $\mathcal{Q}$
- Population: N_k
- Mean think time: Z_k
- Mean service demand: $D_{i,k}$

Output:

- Mean queue length: $Q_{i,k}(\boldsymbol{N})$
- Mean residence time: $Rd_{i,k}(\boldsymbol{N})$
- Mean system throughput: $X_k(\boldsymbol{N})$

1: For every station i and every class k, assign the mean queue length $Q_{i,k}(\boldsymbol{0})$ to be 0
2: For every workload intensity $\boldsymbol{N'}$ with N'_1 from 0 to N_1, N'_2 from 0 to N_2, ..., N'_K from 0 to N_K except for all 0s,
3: For every station i and every class k, compute the mean residence time $Rd_{i,k}(\boldsymbol{N'})$ from (2.79)
4: For every class k, compute the mean system throughput $X_k(\boldsymbol{N'})$ from (2.81)
5: For every station i and every class k, compute the mean queue length $Q_{i,k}(\boldsymbol{N'})$ from (2.67)

To solve a closed QN with N_k jobs for class k, the MVA algorithm has to obtain the mean queue lengths for a total of $\prod_{k=1}^{K}(N_k + 1)$ different workload intensities, thus requiring a large amount of time and space especially when the model contains many classes. This motivates the development of the so-called *approximate mean value analysis* (AMVA) algorithm. The basic idea behind the AMVA algorithm is to estimate the mean queue length of jobs at station i with

one less class-k job as

$$Q_i(\boldsymbol{N}-\boldsymbol{e}_k) \approx h_k(Q_{i,1}(\boldsymbol{N}), Q_{i,2}(\boldsymbol{N}), \ldots, Q_{i,K}(\boldsymbol{N})), \tag{2.82}$$

where $h_k : \mathbb{R}^K \to \mathbb{R}$ is an interpolation function that only depends on the workload intensity $\boldsymbol{N}$ instead of $\boldsymbol{N}-\boldsymbol{e}_k$. A particularly simple yet reasonably accurate form of (2.82) is defined by

$$Q_i(\boldsymbol{N}-\boldsymbol{e}_k) \approx \frac{N_k-1}{N_k} Q_{i,k}(\boldsymbol{N}) + \sum_{l=1, l\neq k}^{K} Q_{i,l}(\boldsymbol{N}). \tag{2.83}$$

Replacing (2.71) with (2.83), we can rewrite (2.79) as

$$Rd_{i,k}(\boldsymbol{N}) \approx \begin{cases} D_{i,k} & \text{if } i \in \mathcal{D}, \\ D_{i,k}\left(1 + \dfrac{N_k-1}{N_k} Q_{i,k}(\boldsymbol{N}) + \sum\limits_{l=1, l\neq k}^{K} Q_{i,l}(\boldsymbol{N})\right) & \text{otherwise.} \end{cases} \tag{2.84}$$

Algorithm 2.7 formulates the AMVA algorithm for closed QNs [Bar79, Sch79], which performs a fixed-point iteration procedure that equally allocates the population of each class to all the stations at the beginning and produces the approximate solution by iteratively applying (2.84), (2.81) and (2.67) until the specified tolerance is satisfied between two successive iterations.

Algorithm 2.7: The AMVA algorithm for closed QNs [Bar79, Sch79].

Input:

- Number of stations: M
- Number of classes: K
- Set of delay stations: $\mathcal{D}$
- Set of queueing stations: $\mathcal{Q}$
- Population: N_k
- Mean think time: Z_k
- Mean service demand: $D_{i,k}$
- Iteration tolerance: ϵ

Output:

- Mean queue length: $Q_{i,k}(\boldsymbol{N})$
- Mean residence time: $Rd_{i,k}(\boldsymbol{N})$
- Mean system throughput: $X_k(\boldsymbol{N})$

1: For every station i and every class k, initialize the mean queue length $Q_{i,k}(\boldsymbol{N})$ and its last copy $Q'_{i,k}(\boldsymbol{N})$ to be N_k/M and 0 respectively
2: While the difference metric $\max_{i\in[1..M],k\in[1..K]} |Q_{i,k}(\boldsymbol{N}) - Q'_{i,k}(\boldsymbol{N})|$ is greater than ϵ,
3: For every station i and every class k, back up the mean queue length $Q_{i,k}(\boldsymbol{N})$ as $Q'_{i,k}(\boldsymbol{N})$
4: For every station i and every class k, compute the mean residence time $Rd_{i,k}(\boldsymbol{N})$ from (2.84)
5: For every class k, compute the mean system throughput $X_k(\boldsymbol{N})$ from (2.81)
6: For every station i and every class k, compute the mean queue length $Q_{i,k}(\boldsymbol{N})$ from (2.67)

Algorithm 2.8: The MVA algorithm for mixed QNs [ZW81].

Input:

- Number of stations: M
- Number of classes: K
- Set of delay stations: $\mathcal{D}$
- Set of queueing stations: $\mathcal{Q}$
- Set of open classes: $\mathcal{O}$
- Set of closed classes: $\mathcal{C}$
- Arrival rate: λ_k
- Population: N_k
- Mean think time: Z_k
- Mean service demand: $D_{i,k}$
- Iteration tolerance: ϵ

Output:

- Mean queue length: $Q_{i,k}(\boldsymbol{I})$
- Mean residence time: $Rd_{i,k}(\boldsymbol{I})$
- Mean system throughput: $X_k(\boldsymbol{I})$

1: For every open class k, compute the mean system throughput $X_k(\boldsymbol{I})$ from (2.73)
2: For every station i and every open class k, compute the mean utilization $U_{i,k}(\boldsymbol{I})$ from (2.66)
3: For every station i and every closed class k, inflate the service demand $D_{i,k}$ as per (2.85)
4: Run Algorithm 2.6 to evaluate the performance measures for the closed classes
5: For every station i and every open class k, compute the mean residence time $Rd_{i,k}(\boldsymbol{I})$ and the mean queue length $Q_{i,k}(\boldsymbol{I})$ from (2.86) and (2.67) respectively

As illustrated in Algorithm 2.8, the MVA algorithm for mixed QNs [ZW81] essentially combines those for open and closed QNs, i.e. Algorithms 2.19 and 2.20, to evaluate the performance measures for different types of classes separately. The mean system throughputs and the mean utilizations for the open classes are first obtained by applying (2.73) and (2.66), respectively. Since the mean total utilization for the open classes at station i is $\sum_{l\in\mathcal{O}} U_{i,l}(\boldsymbol{I})$, the fraction of service capacity left for the closed classes there is $1 - \sum_{l\in\mathcal{O}} U_{i,l}(\boldsymbol{I})$. The effects of the open classes on the performance measures for the closed classes can be represented by inflating the service demand for each closed class k at station i by a factor $1/(1 - \sum_{l\in\mathcal{O}} U_{i,l}(\boldsymbol{I}))$:

$$D'_{i,k} = \frac{D_{i,k}}{1 - \sum\limits_{l\in\mathcal{O}} U_{i,l}(\boldsymbol{I})} \quad \text{for } k \in \mathcal{C}. \tag{2.85}$$

After that, the MVA algorithm for closed QNs is used to evaluate the performance measures for the closed classes as if there were no open classes. The technique of eliminating some job classes by inflating the service demands for the rest is referred to as *load concealment*, which is helpful

in reducing the complexity of a model. With the knowledge of the mean queue lengths for the closed classes and the mean utilizations for the open classes, we can determine the residence time for an open class k at station i as follows:

$$Rd_{i,k}(\boldsymbol{I}) = \begin{cases} D_{i,k} & \text{if } i \in \mathcal{D}, k \in \mathcal{O}; \\ \dfrac{D_{i,k}\left(1 + \sum\limits_{l \in \mathcal{C}} Q_{i,l}(\boldsymbol{I})\right)}{1 - \sum\limits_{l \in \mathcal{O}} U_{i,l}(\boldsymbol{I})} & \text{if } i \in \mathcal{Q}, k \in \mathcal{O}. \end{cases} \tag{2.86}$$

(2.86) is derived similarly to (2.76) except that an extra term, $\sum_{l \in \mathcal{C}} Q_{i,l}(\boldsymbol{I})$, representing the mean total queue length for the closed classes at station i is included from the third equality of (2.75) onward. At last, the mean queue lengths for the open classes are obtained according to (2.67). Note that it is more preferable to run the AMVA algorithm for closed QNs, i.e. Algorithm 2.7, on Line 4 if the number of closed classes is large. This results in Algorithm 2.8 becoming the AMVA algorithm for mixed QNs.

2.3.4 Layered Queueing Networks

Layered queueing networks (LQNs) are a canonical form of extended queueing networks developed for modeling systems with nested simultaneous resource possession. In ordinary queueing networks, hardware resources such as CPUs and disks are represented as stations while the software structure is implicitly taken into account in the service demand at each station. LQNs instead abstract a system as a layered hierarchy of interactive software entities hosted on hardware entities. This layered hierarchy arises from client-server relationships commonly found in a multi-tier architecture. The capabilities to separate software and hardware entities as well as to capture the layers of the system architecture make LQNs a modeling formalism particularly suitable for describing modern software systems.

The primitive constructs of LQNs include *tasks*, *processors*, *entries* and *activities*. A task represents a software entity and runs on a single processor, which characterizes a hardware entity. Services provided by the software entity are modeled as the entries of the task. Every

entry comprises a graph of activities that mimic the execution steps of the modeled service. These activities may send synchronous or asynchronous requests to entries of other tasks. As summarized in Table 2.6, a convenient graphical notation is available for LQNs. Figure 2.6 gives a concrete example of an LQN to demonstrate its usage. This model consists of three tasks, `t1`, `t2`, `t3`, and two processors, `p1`, `p2`. Task `t1` occupies processor `p1`. Tasks `t2` and `t3` share processor `p2`. An entry is defined for each of the three tasks, i.e. `e1`, `e2`, `e3`. The activity graphs of both entries `e1` and `e3` have only one activity: `a1`, `a5`. There are by contrast three activities in that of entry `e2`: `a2`, `a3`, `a4`. During the execution, activities `a1` and `a4` issue synchronous and asynchronous requests respectively to entries `e2` and `e3`. The constructs and features of LQNs are elaborated in the next part.

Table 2.6: The graphical notation of LQNs.

Representation	**Definition**
(Stacked) Parallelogram	Task, interpreted as a software entity
(Stacked) Circle	Processor, interpreted as a hardware entity
Internal Parallelogram	Entry, interpreted as a service provided by a software entity
Internal Rectangle	Activity, interpreted as an execution step of a service
Curly Brackets	Multiplicity of a task or processor
Square Brackets	Mean service time of an activity
Round Brackets*	Mean number of requests from an activity to an entry
Big Z	Mean think time of a reference task
Solid Arrow†	Send requests to an entry/Run on a processor/Perform an activity
Dashed Arrow	Reply to a synchronous request

* If ":D" is appended to the value between the brackets, the pattern in which requests are sent is deterministic. Otherwise, it is stochastic.

† If the end of the arrow is not filled, requests sent to the entry are asynchronous. Otherwise, they are synchronous.

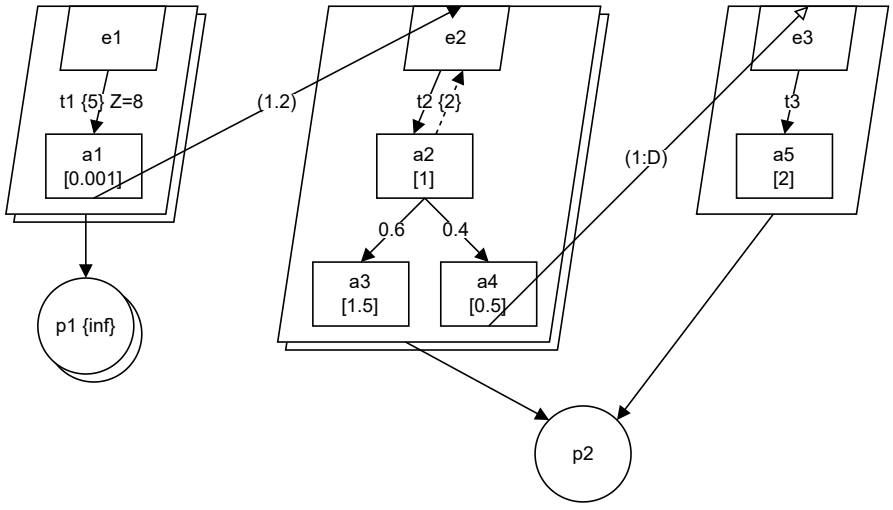


Figure 2.6: A concrete example of an LQN.

By default, a task handles only one request at a time and buffers extra incoming requests in a queue. Assigning a multiplicity to it enables the task to process as many requests as the assigned multiplicity in parallel. Tasks may schedule requests according to three disciplines:

- *First-in-first-out.* Requests are served in the order of their arrival.
- *Preemptive-resume priority.* Requests are served in the order of their priority. New requests with higher priority will preempt those currently in service.
- *Head-of-line priority.* Requests are served in the order of their priority. New requests with higher priority will not preempt those currently in service.

Every LQN contains a special type of task, termed the *reference task*, which typically has a single entry and never accepts requests from other tasks. A reference task characterizes a group of clients with statistically identical behavior. Besides the multiplicity, a mean think time needs to be specified for a reference task. The multiplicity determines the population of clients while the mean think time indicates how long a client is expected to spend between two consecutive interactions. Thus, reference tasks naturally model closed workloads. Open workloads can be generated by letting a reference task send asynchronous requests to normal tasks.

Despite representing hardware entities, processors behave similarly to normal tasks except that they do not depend on any other tasks or processors. A processor also has a multiplicity and can deal with the number of requests up to its multiplicity at the same time. Extra incoming requests are buffered in a queue as well. In addition to first-in-first-out, preemptive-resume priority and head-of-line priority, three more scheduling disciplines are available for processors:

- *Random.* Requests are served in a random order.
- *Processor sharing.* All the requests are served simultaneously, each receiving an equal fraction of the service capacity.
- *Generalized processor sharing.* Requests are divided into different groups. Each group is assigned a positive weight and receives a weighted fraction of the service capacity, which is shared equally among requests within the group.

An entry may accept either synchronous or asynchronous requests. In the former case, each request must be replied to or forwarded to an entry of another task. The behavior of an entry is specified in terms of an activity graph. This is useful to depict a sophisticated workflow. Entries are conceptually considered as comprising two phases. The first phase, known as the *service phase*, spans all activities necessary for replying to a request. The second phase, called the *autonomous phase*, covers remaining activities required to complete the request and is executed in parallel with the request sender.

Each activity is associated with a mean service time, which quantifies the amount of service that it demands from the hosting processor on average. More than one request may be sent during the execution of an activity. Two request patterns, stochastic and deterministic, are applicable. If the request pattern of the activity is stochastic, a random number of requests are sent to each target entry, following a geometric distribution with the specified mean. Otherwise, the activity always sends a fixed number of requests to the target entry.

The activity graph of an entry is formed by connecting its activities with *precedences*. There are essentially two categories of precedences, namely *pre-precedences* and *post-precedences*. The former are used to merge multiple executions while the latter are present to split a single execution. To connect two sets of activities, one needs to define a pair of pre-precedence and post-precedence, and then attaches the source and target sets of activities to them respectively. Table 2.7 lists precedences supported by LQNs.

Table 2.7: Precedences supported by LQNs.

Category	**Name**	**Description**
Pre-Precedence	Sequence	Transfer control from an activity
	And-Join	Merge all the concurrent executions
	Quorum-Join	Merge part of concurrent executions
	Or-Join	Merge branch executions
Post-Precedence	Sequence	Transfer control to an activity
	And-Fork	Split into concurrent executions
	Or-Fork	Split into branch executions
	Loop	Repeat subsequent executions

The analytical approach for solving an LQN entails first decomposing the model into different layers. A common strategy to do so is *loose layering*, where each layer is defined by a single

downstream task or processor. After that, the resulting layers are sorted in the topological order and transformed into ordinary queueing networks. The solution to the LQN is obtained by iteratively analyzing these submodels top down and bottom up until the inputs and outputs of all the layers become consistent. Several LQN solvers are available out there. The oldest and perhaps most famous one is LQNS;[23] which uses the MVA/AMVA algorithms for submodel analysis. Details of the solution techniques for LQNs are beyond the scope of this thesis. We refer the reader to [Fra00] for more information.

2.4 RADON Modeling Profile

Approaches proposed in Chapters 3 and 4, i.e. RADF and RDOF, have been implemented as the architecture decomposition and deployment optimization features of the RADON decomposition tool.[11] The input and output of the tool are TOSCA models in conformance to RADON Particles;[24] a unified modeling profile that broadens the scope of TOSCA YAML Profile v1.3[25] to serverless computing. In what follows, we introduce the basics of the TOSCA language and the extension to RADON Particles in support of the two approaches.

2.4.1 Basics of TOSCA Language

TOSCA is an OASIS[26] standard language for modeling cloud applications. TOSCA models may be encoded in two general formats, XML and YAML. The latter has a much easier syntax than the former, thus being chosen as the encoding format of RADON Particles. A TOSCA model is defined as a *service template* that describes the topology of a composite application and its orchestration workflows in a declarative fashion. It can be packaged together with accompanying artifacts, such as binaries, scripts and configuration files, into a so-called *cloud service*

[23] http://www.sce.carleton.ca/rads/lqns/
[24] https://github.com/radon-h2020/radon-particles
[25] https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html
[26] https://www.oasis-open.org/

archive and then executed by a TOSCA-compliant orchestrator to automate the deployment and management of the application.

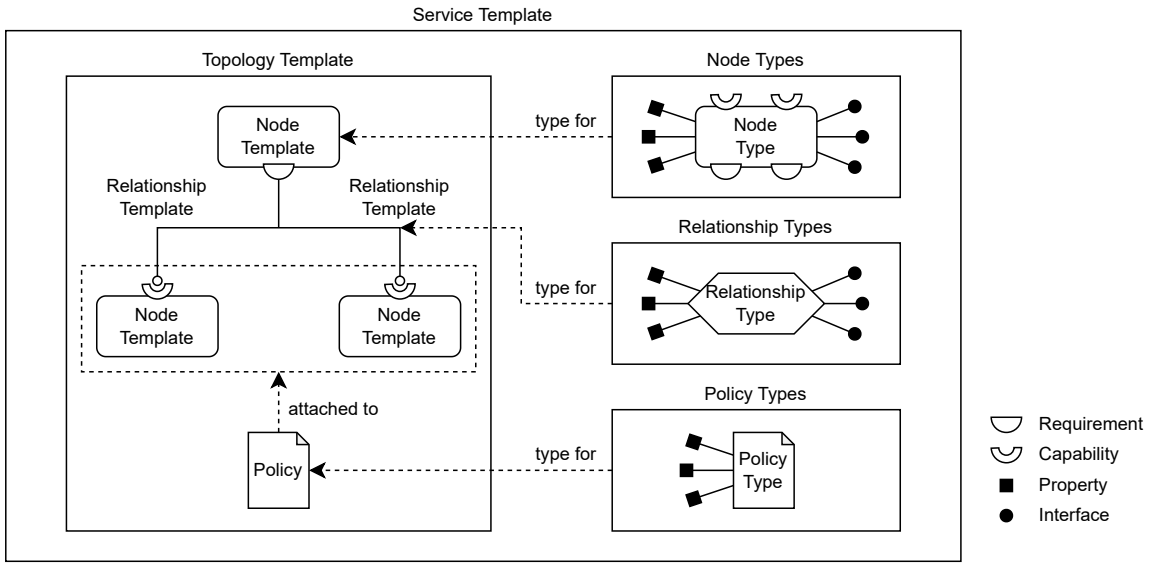


Figure 2.7: The structure of a service template.

Figure 2.7 illustrates the structure of a service template, which contains a *topology template* and the definitions of all the types in use. There are three categories of building blocks in TOSCA: *nodes*, *relationships* and *policies*. The topology template depicts a colored graph where each vertex is a *node template* representing a component of the application and each edge is a *relationship template* representing an association of one application component with another. In essence, a node template is of a specific type, which defines its *properties* and *interfaces*. This is also true for relationship templates. A property (e.g. the `component_version` of a `SoftwareComponent`) carries a requested or desired value to be achieved. An interface (e.g. the `lifecycle.Standard` of any node) arranges a set of workflow operations to be performed. Conceptually, TOSCA types are comparable to classes in object-oriented programming whilst the templates correspond to objects created from those classes. TOSCA innovates the concepts of *requirements* and *capabilities* to explicitly regulate the establishment of relationships between nodes. For example, a `Database` has a `host` requirement for the `Compute` capability of a `DBMS`, and this must be fulfilled through a `HostedOn` relationship. Policies are present in TOSCA to enable the specification of non-functional requirements, such as security, availability, scalability and performance. Unlike node or relationship templates, they have properties but no interfaces. A policy can be attached to a single node or a group of nodes defined in the topology template.

2.4.2 Extension to RADON Particles

The RADON project delivers a holistic DevOps framework for designing, prototyping, deploying, testing, verifying and evolving serverless applications. RADON Particles, the unified modeling profile offered by the RADON framework, in particular helps to realize the integration of RADF and RDOF in an end-to-end manner and the evaluation of them on various use cases from industry. To support the two approaches, we have extended RADON Particles with the following features:

- Incorporation of modeling constructs provided by LQNs;
- Definition of monoliths and microservices;
- Enhanced modeling of serverless applications;
- Specification of reference workloads, either open or closed;
- Specification of performance requirements, e.g. the mean response time.

Data Types

Several new data types are defined to incorporate LQN constructs, namely entries, activities, precedences and requests. They underlie the tool-specific extension made to node, relationship and policy types that are detailed subsequently.

Random Variable. Temporal information is expressed in LQNs as non-negative random variables. To represent such a quantity, we define the `RandomVariable` data type, as shown in Listing 2.1. One can adjust the `mean` and `scv` of a `RandomVariable`, i.e. the expectation and squared coefficient of variation. The second property is only reserved for now due to the limitation of the LQN solver invoked by the RADON decomposition tool. Notably, a zero `mean` implies a zero `scv` and therefore should also be avoided. A workaround is to use a small value like 0.001.

Listing 2.1: The data type for representing a non-negative random variable.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.RandomVariable:
```

```
    derived_from: tosca.datatypes.Root
    properties:
      mean:
        type: float
        required: true
        default: 1.0
        constraints:
          - greater_or_equal: 0.0
      scv:
        type: float
        required: false
        constraints:
          - greater_or_equal: 0.0
```

Entry. We compare each node to a task, which contains a set of entries. The `Entry` data type shown in Listing 2.2 is defined for representing an entry of a node. There are four properties for an `Entry`:

- `description`, a `string` specifying a concise description of the entry;
- `class_trace`, a `list` of `string`s specifying the subset of classes observed in the execution traces of the entry;
- `activities`, a `map` of `Activity`s specifying the activities of the entry;
- `precedences`, a `list` of `Precedence`s specifying precedences connecting the activities of the entry.

Listing 2.2: The data type for representing an entry of a node.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Entry:
    derived_from: tosca.datatypes.Root
    properties:
      description:
        type: string
        required: false
      class_trace:
        type: list
        required: false
        entry_schema:
          type: string
      activities:
        type: map
        required: false
```

```
        entry_schema:
          type: radon.datatypes.Activity
      precedences:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Precedence
```

Activity. An entry comprises a graph of activities. Listing 2.3 shows the `Activity` data type defined for representing an activity of an entry. The properties of an `Activity` are as follows:

- `service_time`, a `RandomVariable` specifying the service time of the activity;
- `bound_to_entry`, a `boolean` specifying whether the entry is executed starting with the activity;
- `replies_to_entry`, a `boolean` specifying whether the entry sends back a response after the activity (only used when the entry accepts synchronous requests);
- `request_pattern`, a `string` specifying the pattern of requests sent by the activity.

Listing 2.3: The data type for representing an activity of an entry.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Activity:
    derived_from: tosca.datatypes.Root
    properties:
      service_time:
        type: radon.datatypes.RandomVariable
        required: false
      bound_to_entry:
        type: boolean
        required: false
      replies_to_entry:
        type: boolean
        required: false
      request_pattern:
        type: string
        required: false
        constraints:
          - valid_values: [ stochastic, deterministic ]
```

Precedence. The activities of an entry are connected by precedences. To represent a precedence among activities, we define the `Precedence` data type, as shown in Listing 2.4. Below are the properties of a `Precedence`:

- `type`, a `string` specifying the type of the precedence;
- `pre_activities`, a `list` of `string`s specifying the set of activities connected from by the precedence;
- `post_activities`, a `list` of `string`s specifying the set of activities connected to by the precedence;
- `parameters`, a `list` of `float`s specifying the parameters of the precedence.

Table 2.8 summarizes valid settings for these properties.

Listing 2.4: The data type for representing a precedence among activities.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Precedence:
    derived_from: tosca.datatypes.Root
    properties:
      type:
        type: string
        required: true
        default: sequence
        constraints:
          - valid_values: [ sequence, and-fork, and-join, or-fork, or-join,
            loop ]
      pre_activities:
        type: list
        required: true
        entry_schema:
          type: string
      post_activities:
        type: list
        required: true
        entry_schema:
          type: string
      parameters:
        type: list
        required: false
        entry_schema:
          type: float
          constraints:
            - greater_or_equal: 0.0
```

Interaction. We model interactions between nodes by blending the concept of requests into relationships. The `Interaction` data type shown in Listing 2.5 is defined for representing an interaction of a relationship. An `Interaction` has five properties:

Table 2.8: Valid settings for the properties of a `Precedence`.

Type	Pre-Activities	Post-Activities	Parameters	
			Number*	Description
`sequence`	single	single	0	N/A
`and-fork`	single	multiple	0	N/A
`and-join`	multiple	single	1	Number of pre-activities that must be completed before the join
`or-fork`	single	multiple	n	Probability that each of the n post-activities is chosen as the next one to perform
`or-join`	multiple	single	0	N/A
`loop`	single	multiple	$n-1$	Number of times that each of the first $n-1$ post-activities is repeated

* n denotes the number of post-activities.

- `type`, a `string` specifying the type of the interaction;
- `source_activity`, a `string` specifying the source activity of the interaction;
- `target_entry`, a `string` specifying the target entry of the interaction;
- `number_of_requests`, a `float` specifying the number of requests sent during the interaction (or its mean if the source activity issues requests in a stochastic pattern);
- `network_delay`, a `RandomVariable` specifying the network delay of each request sent during the interaction.

Listing 2.5: The data type for representing an interaction of a relationship.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Interaction:
    derived_from: tosca.datatypes.Root
    properties:
      type:
        type: string
        required: true
        default: synchronous
        constraints:
          - valid_values: [ synchronous, asynchronous ]
      source_activity:
        type: string
        required: true
      target_entry:
        type: string
        required: true
      number_of_requests:
        type: float
        required: true
```

```
        default: 1.0
        constraints:
          - greater_or_equal: 0.0
      network_delay:
        type: radon.datatypes.RandomVariable
        required: false
```

Node Types

Nodes of the same abstract type often behave similarly across different cloud platforms. We define new node types or adding properties to existing ones at the abstract level to convey the inputs and outputs of RADF and RDOF. Particularly, a common property named `entries` is introduced to these node types, making it possible to capture the node behavior in the LQN fashion.

Monolith. Listing 2.6 shows the node type defined for representing a monolith. This is the normative `WebApplication` node type enriched with seven properties:

- `name`, a `string` specifying the name of the monolith;
- `granularity`, a `string` specifying the granularity of the decomposition solution to the monolith (e.g. `coarse-grained` for a microservices-based solution);
- `word_embedding`, a `string` specifying the word embedding to analyze the semantics of operations provided by the interface of the monolith;
- `clustering_algorithm`, a `string` specifying the clustering algorithm to identify candidate microservices from the monolith;
- `clustering_parameters`, a `map` of `string`s specifying the parameters of the clustering algorithm manually;
- `validity_index`, a `string` specifying the validity index to parameterize the clustering algorithm automatically.
- `entries`, a `map` of `Entry`s specifying the set of entries that characterize operations provided by the interface of the monolith.
- `core_elements`, a `list` of `string`s specifying the core elements of the monolith.

Valid settings for the clustering properties are summarized in Table 2.9. To represent the server of monoliths, we also extend the normative `WebServer` node type, as shown in Listing 2.7.

Listing 2.6: The abstract node type for representing a monolith.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.WebApplication:
    derived_from: tosca.nodes.WebApplication
    properties:
      name:
        type: string
        required: false
      granularity:
        type: string
        required: false
        constraints:
          - valid_values: [ monolithic, coarse-grained, fine-grained ]
      word_embedding:
        type: string
        required: false
        constraints:
          - valid_values: [ esa, word2vec, glove, fasttext ]
      clustering_algorithm:
        type: string
        required: false
        constraints:
          - valid_values: [ hierarchical, kmedoids, dbscan, spectral ]
      clustering_parameters:
        type: map
        required: false
        entry_schema:
          type: string
      validity_index:
        type: string
        required: false
        constraints:
          - valid_values: [ gamma, silhouette ]
      entries:
        type: map
        required: false
        entry_schema:
          type: radon.datatypes.Entry
      core_elements:
        type: list
        required: false
        entry_schema:
          type: string
    requirements:
```

```
    - host:
        capability: tosca.capabilities.Compute
        node: radon.nodes.abstract.WebServer
        relationship: tosca.relationships.HostedOn
        occurrences: [ 1, 1 ]
```

Table 2.9: Valid settings for the clustering properties of a `WebApplication`.

Clustering Algorithm	Clustering Parameter		
	Key Name	Default Value*	Description
hierarchical	linkage	–	Definition of the inter-cluster distance
	K	–	Number of clusters to discover
kmedoids	K	–	Number of clusters to discover
	R	–	Number of times to repeat the clustering
dbscan	epsilon	–	Neighborhood of a data point
	m	1	Minimum neighbors of a core point
spectral	K	–	Number of clusters to discover
	sigma	0.6006	Scaling factor of the Gaussian kernel
	R	–	Number of times to repeat the clustering

* If not explicitly specified, parameters without a default value are automatically determined by optimizing the validity index.

Listing 2.7: The abstract node type for representing the server of monoliths.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.WebServer:
    derived_from: tosca.nodes.WebServer
    properties:
      name:
        type: string
        required: false
    capabilities:
      host:
        occurrences: [ 1, UNBOUNDED ]
        valid_source_types: [ radon.nodes.abstract.WebApplication ]
        type: tosca.capabilities.Compute
```

Microservice. The `ContainerApplication` and `ContainerRuntime` node types shown in Listings 2.8 and 2.9 are defined for representing a microservice and the runtime of microservices, respectively. They are derived from the normative `Container.Application` and `Container.Runtime` node types. Five properties, `name`, `granularity`, `entries`, `core_elements` and `global_utilities`, are made available for a `ContainerApplication`. The first four are similar to those of a `WebApplication`. The last one specifies utilities that the microservice shares with others.

Listing 2.8: The abstract node type for representing a microservice.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.ContainerApplication:
    derived_from: tosca.nodes.Container.Application
    properties:
      name:
        type: string
        required: false
      granularity:
        type: string
        required: false
        constraints:
          - valid_values: [ coarse-grained, fine-grained ]
      entries:
        type: map
        required: false
        entry_schema:
          type: radon.datatypes.Entry
      core_elements:
        type: list
        required: false
        entry_schema:
          type: string
      global_utilities:
        type: list
        required: false
        entry_schema:
          type: string
    requirements:
      - host:
          capability: tosca.capabilities.Compute
          node: radon.nodes.abstract.ContainerRuntime
          relationship: tosca.relationships.HostedOn
          occurrences: [ 1, 1 ]
      - storage:
          capability: tosca.capabilities.Storage
          occurrences: [ 0, 1 ]
      - network:
          capability: tosca.capabilities.Endpoint
          occurrences: [ 0, 1 ]
```

Listing 2.9: The abstract node type for representing the runtime of microservices.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.ContainerRuntime:
    derived_from: tosca.nodes.Container.Runtime
```

```
    properties:
      name:
        type: string
        required: false
    capabilities:
      host:
        occurrences: [ 1, UNBOUNDED ]
        valid_source_types: [ radon.nodes.abstract.ContainerApplication ]
        type: tosca.capabilities.Compute
```

Serverless Function. The extended `Function` node type for representing a serverless function is shown in Listing 2.10. We restrict the `entries` of a `Function` to a `map` of a single `Entry`. The rest properties, `core_elements`, `local_utilities` and `global_utilities`, specify the core elements of the function, utilities that it shares locally with other functions and those shared globally among different microservices, respectively.

Listing 2.10: The abstract node type for representing a serverless function.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.Function:
    derived_from: tosca.nodes.Root
    properties:
      name:
        type: string
        required: true
      entries:
        type: map
        required: false
        constraints:
          - length: 1
        entry_schema:
          type: radon.datatypes.Entry
      core_elements:
        type: list
        required: false
        entry_schema:
          type: string
      local_utilities:
        type: list
        required: false
        entry_schema:
          type: string
      global_utilities:
        type: list
        required: false
```

```
        entry_schema:
          type: string
    requirements:
      - host:
          capability: tosca.capabilities.Container
          node: radon.nodes.abstract.CloudPlatform
          relationship: tosca.relationships.HostedOn
          occurrences: [ 1, 1 ]
      - invoker:
          capability: radon.capabilities.Invocable
          relationship: radon.relationships.Triggers
          occurrences: [ 0, UNBOUNDED ]
      - endpoint:
          capability: tosca.capabilities.Endpoint
          relationship: radon.relationships.ConnectsTo
          occurrences: [ 0, UNBOUNDED ]
    capabilities:
      invocable:
        occurrences: [ 0, UNBOUNDED ]
        type: radon.capabilities.Invocable
```

Object Storage. Listing 2.11 shows the extended `ObjectStorage` node type for representing an object storage. An `ObjectStorage` can have an arbitrary `map` of `Entry`s. However, the key name of each `Entry` must be prefixed with the name of the operation that it is associated with, e.g. "get", "get_image" or "getImage". This enables the RADON decomposition tool to compute the operating cost of the `ObjectStorage` for deployment optimization. The supported operations currently include GET, PUT and DELETE.

Listing 2.11: The abstract node type for representing an object storage.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.ObjectStorage:
    derived_from: tosca.nodes.Storage.ObjectStorage
    properties:
      entries:
        type: map
        required: false
        entry_schema:
          type: radon.datatypes.Entry
    requirements:
      - host:
          capability: tosca.capabilities.Container
          node: radon.nodes.abstract.CloudPlatform
          relationship: tosca.relationships.HostedOn
```

```
          occurrences: [ 1, 1 ]
      - invoker:
          capability: radon.capabilities.Invocable
          node: radon.nodes.abstract.Function
          relationship: radon.relationships.Triggers
          occurrences: [ 0, UNBOUNDED ]
```

Reference Workload. We define a new node type called `Workload` for representing a generic workload. As shown in Listing 2.12, the `entries` of a `Workload` is restricted to a `map` of a single `Entry`. It may `Triggers` any `Invocable` node, e.g. a `Function`, or `ConnectsTo` any `Endpoint` node, e.g. an `ObjectStorage`. To represent a specific workload, we further derive the `OpenWorkload` and `ClosedWorkload` node types. Listings 2.13 and 2.14 show the two child node types, respectively. One can set the `interarrival_time` of an `OpenWorkload` or the `population` and `think_time` of a `ClosedWorkload`. The `interarrival_time` and `think_time` properties are both `RandomVariables`.

Listing 2.12: The abstract node type for representing a generic workload.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.Workload:
    derived_from: tosca.nodes.Root
    properties:
      name:
        type: string
        required: false
      entries:
        type: map
        required: false
        constraints:
          - length: 1
        entry_schema:
          type: radon.datatypes.Entry
    requirements:
      - invoker:
          capability: radon.capabilities.Invocable
          relationship: radon.relationships.Triggers
          occurrences: [ 0, UNBOUNDED ]
      - endpoint:
          capability: tosca.capabilities.Endpoint
          relationship: radon.relationships.ConnectsTo
          occurrences: [ 0, UNBOUNDED ]
```

Listing 2.13: The abstract node type for representing an open workload.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.workload.OpenWorkload:
    derived_from: radon.nodes.abstract.Workload
    properties:
      interarrival_time:
        type: radon.datatypes.RandomVariable
        required: true
```

Listing 2.14: The abstract node type for representing a closed workload.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.workload.ClosedWorkload:
    derived_from: radon.nodes.abstract.Workload
    properties:
      population:
        type: integer
        required: true
        default: 1
        constraints:
          - greater_or_equal: 1
      think_time:
        type: radon.datatypes.RandomVariable
        required: true
```

Relationship Types

Some relationships imply the existence of an interaction channel that bridges between source and target nodes. Apart from allowing the specification of an `entries` property for nodes, we also introduce a common property named `interactions` to relationship types. The `Triggers` and `ConnectsTo` relationship types shown in Listings 2.15 and 2.16 are examples of this enhancement. They are used for representing an invocation trigger and an endpoint connection respectively at either abstract or concrete level. The `interactions` of a relationship is typically defined as a flexible `list` of `Interaction`s.

Listing 2.15: The relationship type for representing an invocation trigger.

```
tosca_definitions_version: tosca_simple_yaml_1_3
relationship_types:
```

```
  radon.relationships.Triggers:
    derived_from: tosca.relationships.ConnectsTo
    properties:
      events:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Event
      interactions:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Interaction
    valid_target_types: [ radon.capabilities.Invocable ]
```

Listing 2.16: The relationship type for representing an endpoint connection.

```
tosca_definitions_version: tosca_simple_yaml_1_3
relationship_types:
  radon.relationships.ConnectsTo:
    derived_from: tosca.relationships.ConnectsTo
    properties:
      interactions:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Interaction
```

Policy Types

To represent a generic performance requirement, we extend the normative `Performance` policy type as shown in Listing 2.17. One can configure the `target_entries` that the requirement is imposed on as well as the `upper_bound` and `lower_bound` that their performance should satisfy. Listings 2.18 and 2.19 show two child policy types, `MeanResponseTime` and `MeanTotalResponseTime`, derived for representing performance requirements on the mean response time and the mean total response time respectively. The main difference between these policy types is that the former applies to each target node or entry individually while the latter applies to all the target nodes or entries together. It is worth pointing out that in certain cases, the mean total response time is equivalent to the mean system response time.

Listing 2.17: The policy type for representing a generic performance requirement.

```
tosca_definitions_version: tosca_simple_yaml_1_3
policy_types:
  radon.policies.Performance:
    derived_from: tosca.policies.Performance
    properties:
      target_entries:
        type: list
        required: false
        entry_schema:
          type: string
      lower_bound:
        type: float
        required: false
        constraints:
          - greater_or_equal: 0.0
      upper_bound:
        type: float
        required: false
        constraints:
          - greater_or_equal: 0.0
```

Listing 2.18: The policy type for representing a performance requirement on the mean response time.

```
tosca_definitions_version: tosca_simple_yaml_1_3
policy_types:
  radon.policies.performance.MeanResponseTime:
    derived_from: radon.policies.Performance
```

Listing 2.19: The policy type for representing a performance requirement on the mean total response time.

```
tosca_definitions_version: tosca_simple_yaml_1_3
policy_types:
  radon.policies.performance.MeanTotalResponseTime:
    derived_from: radon.policies.Performance
```

Chapter 3

RADF: Architecture Decomposition for Function as a Service

3.1 Introduction

Monolithic applications combine everything, including user interface, business logic, data access and service integration, within a single program. Despite being simple to get started with, this architectural pattern suffers several drawbacks whose severity grows as the features of the application evolve over time [New15, Ric18]. Firstly, it is hard for developers to keep a detailed insight into a large monolith, causing the maintenance of source code to be tedious and error-prone. Secondly, applications designed in a monolithic architecture may only be scaled as a whole, which is a costly response to typically unbalanced and fluctuating workloads. Thirdly, the development of a monolithic application becomes cumbersome once its size reaches a certain level, since the tight coupling among various modules prevents multiple teams from working independently.

All these issues are well addressed in the serverless architecture, where an application consists of stateless functions that implement the main business logic, together with backend-as-a-service (BaaS) components, for example object storages, databases and message queues, that fulfill specific product needs [RC17, JSSS+19]. In most cases, one function is associated with

a definite and unique role, thus being very easy to understand and maintain. The scaling of functions is conducted separately according to their own workloads, which is more efficient than simply duplicating instances of the whole application. Additionally, different development teams—or even individual developers—can merely focus on the bundle of functions assigned to themselves without concerning much about the others.

Migrating monolithic applications to the serverless architecture is generally difficult, as exemplified by case studies reported in [GHK+20], [LPS+20] and [STM21]. The major challenge lies in how to decompose a monolith into serverless functions that satisfy the specified functional and non-functional requirements. As a common functional requirement, the behavior of the application should remain invariant after the migration. The non-functional requirements define qualities expected of a sound solution, such as coupling, cohesion, performance, cost, security and privacy. Software designers have to make a wide range of decisions during the decomposition process in order to meet both functional and non-functional requirements. Unfortunately, there is a lack of systematic approaches to this problem, and the qualities of the final solution are largely subject to human experience.

To facilitate migration to the serverless architecture, we propose in this chapter a semi-automatic approach called RADF[15] for refactoring a monolithic application into one compatible with the FaaS paradigm. The proposed approach divides the entire process into two stages, i.e. coarse- and fine-grained decompositions. It identifies candidate microservices by clustering API operations in the first stage and composes each identified microservice with serverless functions in the second stage. We have implemented RADF as the architecture decomposition feature of the RADON decomposition tool.[11] The input and output of the tool are TOSCA models [BBKL14], which can be consumed by a TOSCA-compliant orchestrator for increased automation. An evaluation of the proposed approach has been carried out in terms of its capability for microservice identification and feasibility for code refactoring. The evaluation results indicate that the decomposition solution given by RADF has lower coupling and relatively balanced cohesion against baselines found in prior art.

The rest of the chapter is organized as follows. Section 3.2 reviews the related work to show the motivation behind RADF. Section 3.3 describes a running example to be used afterward. An overview of the refactoring strategy and the reference workflow is present in Section 3.4. Techniques for interface analysis and microservice identification are elaborated in Sections 3.5 and 3.6, respectively. Section 3.7 outlines a practical procedure for code refactoring. Section 3.8 introduces the implementation of the proposed approach. The evaluation experiments and results are reported in Section 3.9. Section 3.10 is dedicated to the conclusions.

3.2 Related Work

Automatic decomposition of a monolith into microservices has gained extensive attention in the past few years. Gysel et al. [GKGZ16] offered a tool named Service Cutter, which constructs a weighted undirected graph capturing the domain model and coupling information of a software system, and derives service decomposition from the graph with the Girvan-Newman [NG04] or the epidemic label propagation algorithm [RAK07]. Baresi et al. [BGDR17] introduced the idea of identifying microservices by means of interface analysis. Specifically, they generate candidate microservices by mapping operations defined in the API specification onto the concepts in a reference vocabulary. Mazlami et al. [MCL17] suggested three coupling strategies, namely logical, semantic and contributor, to obtain a graph representation of an application based on its revision history or source code as well as a minimum spanning tree clustering algorithm to extract candidate microservices from the graph. De Alwis et al. [DABPF18] presented function splitting heuristics for identifying consumer-oriented parts of an enterprise system that could be suitably re-engineered as microservices, with respect to business object subtypes or common execution fragments. A data flow-driven approach was proposed by Li et al. [LZJ+19]. This approach takes a well-defined diagram of processes and data stores depicting the business logic of an application and finds candidate microservices by grouping the processes and closely related data stores into individual modules. Jin et al. [JLC+19] contributes a service identification approach that discovers functional atoms from the execution traces of different test cases and resorts to the non-dominated sorting genetic algorithm II [DPAM02] to search for a partition of

the functional atoms with the optimal structural and conceptual intra- and inter-connectivities. A graph neural network with the ability to dilute outliers was applied by Desai et al. [DBT21] to decomposing monoliths. They create the call graph of an application through static analysis of the source code and populate it with attributes that incorporate execution traces of the API entry points and inheritance dependencies among the classes. Kalia et al. [KXK+21] developed Mono2Micro. This tool acquires candidate microservices by performing hierarchical clustering on the classes according to direct and indirect call relations and patterns observed in the execution traces of different use cases. A multi-model-based approach to microservice identification was proposed by Daoud et al. [DEMF+21], who retrieve control, data and semantic dependencies from the business process model of an application and put highly-dependent activities into the same microservice using a collaborative clustering algorithm.

There are a couple of publications devoted to automatic decomposition of a monolith into serverless functions. To convert Java code into Lambda[3] functions, Spillner and Dorodko [SD17] devised a FaaSification pipeline comprising six steps, specifically analysis, decomposition, translation, compilation, upload and verification, and implemented a tool called Podilizer to support automation of this pipeline. They also offered Lambada [Spi17], another tool that follows a similar idea but operates on Python code. De Carvalho and de Araújo [dCdA19] developed NodeJS2FaaS for migrating Node.js code to various FaaS compute services including AWS Lambda[3], Google Cloud Functions[4] and Microsoft Azure Functions[5]. NodeJS2FaaS performs the migration in five steps: extraction, normalization, assembly, compression and publication. The aforementioned three approaches have two common weaknesses. Code dependencies are computed via static analysis. Advanced language features, for example reflection, dynamic class loading and dependency injection, are consequently left aside, leading to discrepancies between the detected and the actual dependencies. Besides, these approaches essentially carry out mechanical transformation without taking into account the business logic of the application, which is an important factor in making decomposition decisions. Noticing the current gap, we propose RADF as an alternative approach for decomposing a monolith at the function level. RADF adopts a two-stage refactoring strategy, where the decomposition process is divided into coarse- and fine-grained stages. This simplifies the refactoring procedure into smaller steps. Moreover,

RADF produces decomposition solutions based on business logic inherent in the interface of the application, leveraging the state-of-the-art techniques for natural language processing and dynamic tracing.

3.3 Running Example

We consider the famous Cargo Shipping system[27] as a running example to demonstrate RADF in the subsequent sections. The Cargo Shipping system is a sample Java application introduced by Evans [Eva04] to illustrate domain-driven design, a software development approach that maps software artifacts onto business domain concepts defined by human experts. Prior art such as [GKGZ16], [BGDR17], [LZJ+19] and [DEMF+21] has used this application to evaluate their decomposition approaches, which potentially provides baselines for us to compare RADF with these approaches at the level of microservices.

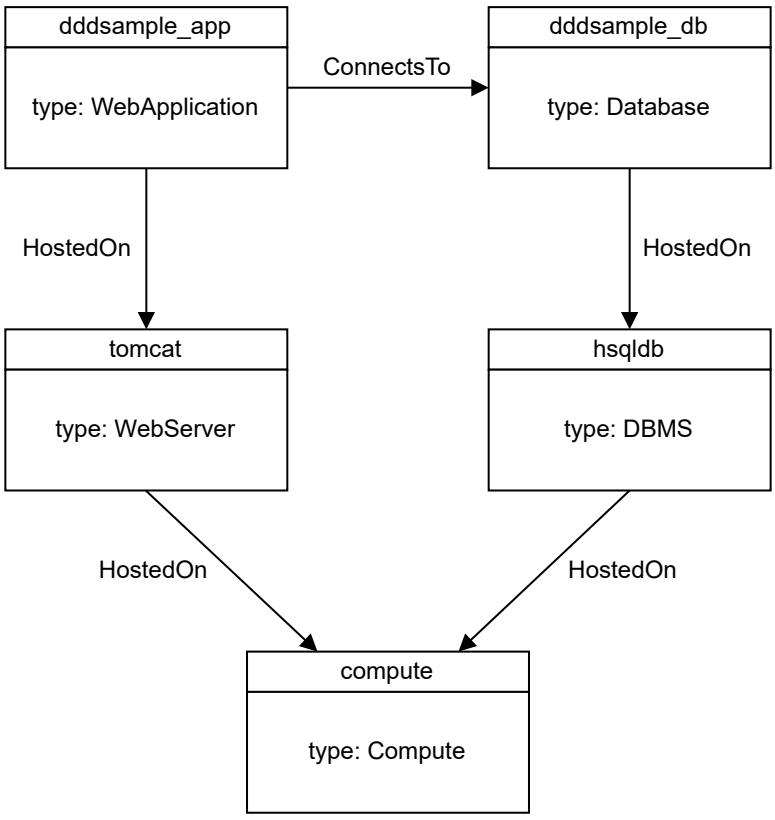


Figure 3.1: The TOSCA model of the Cargo Shipping system.

[27] `https://github.com/citerus/dddsample-core`

Figure 3.1 illustrates the TOSCA model of the Cargo Shipping system, which has a typical monolithic architecture consisting of a web application named `dddsample_app` and a database named `dddsample_db`. The two components are deployed respectively on an Apache Tomcat[28] web server and a HyperSQL[29] database management system (DBMS) operating on some compute node. At runtime, the `dddsample_app` application accesses the `dddsample_db` database for persistent data storage, reading and writing information about cargoes, voyages, locations and handling events.

The Cargo Shipping system provides functionalities for managing and tracking cargoes. These functionalities are implemented conceptually by two subsystems, which we refer to as Cargo Admin and Cargo Tracking respectively. The Cargo Admin subsystem is present for the system manager to book a new cargo, show the details of the registered cargoes, pick a destination or select an itinerary for a cargo. The Cargo Tracking subsystem updates the status of every cargo according to the submitted handling reports, and allows a customer to inspect the handling history of a cargo with a given tracking ID.

Suppose that a software designer wants to migrate the Cargo Shipping system to the serverless architecture. They may ask themselves the following list of questions before taking action:

- What does the code structure of the decomposition solution look like?
- How many functions do I need to preserve the behavior of the application?
- What are the role and boundary associated with each function?

The last question is the hardest one, which involves numerous decisions on the assignment of classes. The goal of RADF is to find answers to these questions in a systematic way.

3.4 Overview

To help with a better understanding of RADF in detail, we draw a preliminary overview of the proposed approach in this section, particularly focusing on the refactoring strategy adopted for

[28] `https://tomcat.apache.org/`
[29] `http://hsqldb.org/`

the decomposition of a monolith into serverless functions as well as the reference workflow to complete this task.

3.4.1 Refactoring Strategy

The interface of an application defines a set of specific operations that interpret and execute the business logic. We think a pair of API operations correlated if they coordinate to serve the same business capability or subdomain. To produce a decomposition solution with loose coupling and tight cohesion, candidate microservices can be identified as small self-contained services that are responsible for different groups of correlated API operations. A natural way to compose such a service in a FaaS-compatible manner is to handle one API operation with a dedicated serverless function.

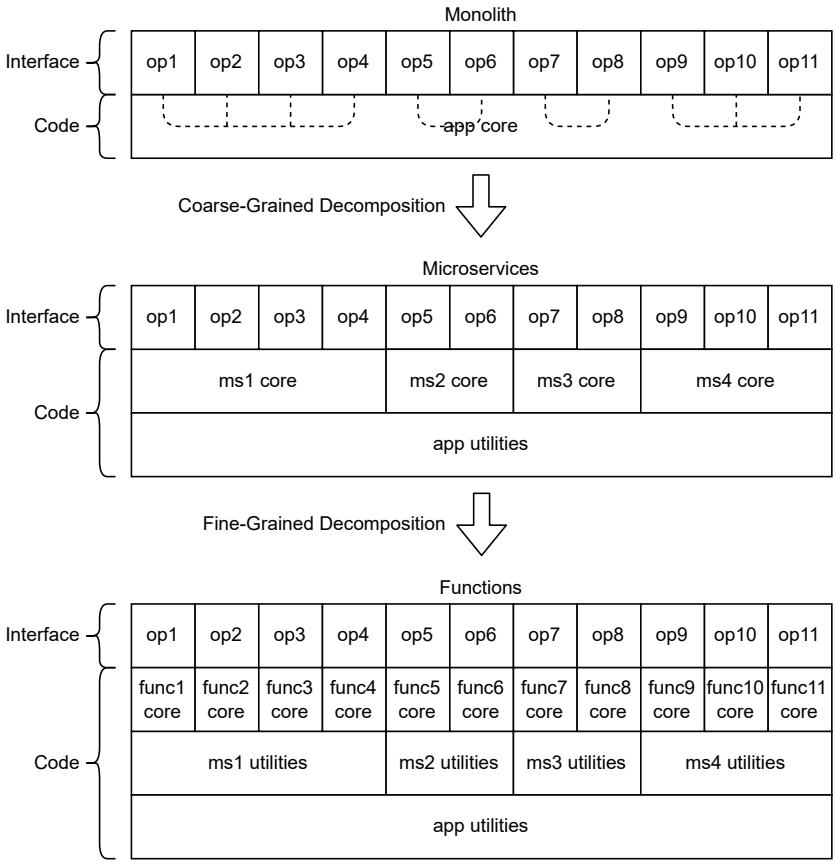


Figure 3.2: The refactoring strategy for decomposing a monolith into serverless functions.

Building on these notions, the refactoring strategy illustrated in Figure 3.2 is adopted to decompose a monolith into serverless functions. RADF divides the entire process into two stages,

namely coarse- and fine-grained decompositions. All the classes are initially assumed to form a single layer: the application core. In the first stage, the monolith is decomposed into microservices by partitioning its interface into groups of correlated operations, each of which implies a candidate microservice. Classes implementing the application are separated into the core elements of the microservices and global utilities shared among them. This leads to a code structure with two layers called microservice cores and application utilities respectively. In the second stage, every microservice is further decomposed into as many serverless functions as API operations in the corresponding group. Classes belonging to the microservice are separated into the core elements of the serverless functions and local utilities shared among them. A three-layer code structure consisting of function cores, microservice utilities and application utilities is thus obtained.

Dividing the decomposition process into the coarse- and fine-grained stages reduces the difficulty of the refactoring procedure and enables the reuse of ideas and techniques from prior work on microservice identification. Microservices and serverless functions arising from this refactoring strategy also follow the single responsibility principle, which can enhance the quality of the resultant decomposition solution. Since the interface of the application remains unchanged, its behavior is preserved from a client perspective. No additional modifications are demanded on the client side after the refactoring.

3.4.2 Reference Workflow

Figure 3.3 shows the reference workflow adopted to decompose a monolith into serverless functions. RADF produces the decomposition solution to an application through semantic and structural analysis of its interface. As a requisite, the software designer must provide an API documentation, preferably compliant with a certain standard like the OpenAPI specification.[30] The semantic analysis starts by retrieving the descriptions of API operations. The keywords of the descriptions are then extracted and used to compute the semantic similarities. As for the structural analysis, functional tests are performed to collect the execution traces of API

[30] `https://swagger.io/specification/`

operations. The structural similarities are computed from the traces. Afterward, candidate microservices are identified by clustering API operations based on their semantic and structural similarities. A number of serverless functions are created to compose each identified microservice. One handles an individual API operation. Finally, the source code of the application is refactored accordingly.

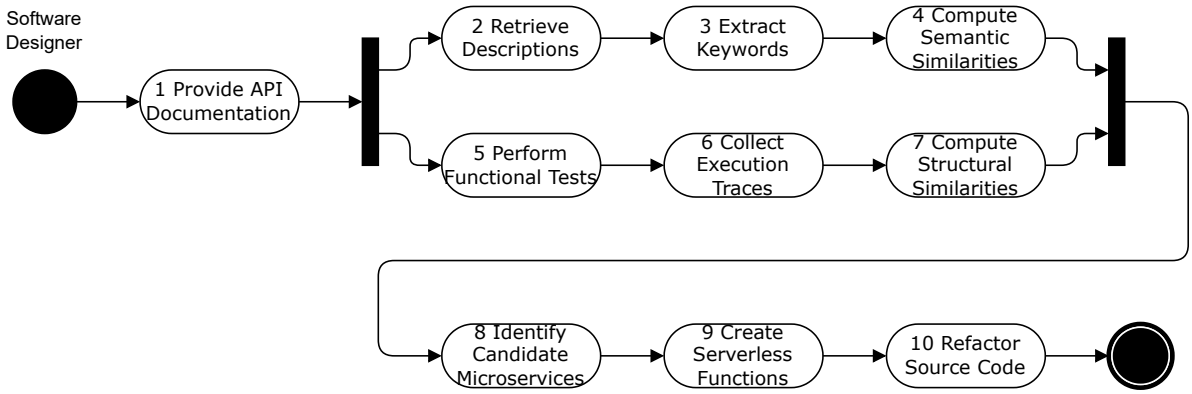


Figure 3.3: The reference workflow for decomposing a monolith into serverless functions.

The API documentation is essential for the above reference workflow, but not all the applications have a well-documented interface. The Cargo Shipping system is such an example. In this case, the software designer ought to scan the interface of the application and gather adequate information for both semantic and structural analysis. In the semantic analysis, a concise description is required for every API operation. The software designer could simply use the name of the corresponding entry point or derive one from their understanding of the business logic. Technical details about API operations are needed in the structural analysis to perform functional tests and collect execution traces. These include the path, method, parameters, body and response as well as the entry point. Tables 3.1 and 3.2 report the definitions of API operations exposed by the Cargo Shipping system.

3.5 Interface Analysis

This section introduces techniques used to analyze the semantics and structure of an API operation. Similarity measures that we devise for quantifying to what extent a pair of API operations are semantically and structurally correlated are also given, in company with their aggregated form.

Table 3.1: Definitions of API operations exposed by the Cargo Shipping system (Part 1).

ID	Path	Method	Parameter/Body/Response*
1	/admin/registration	GET	
			unlocodes: string array locations: object array
2	/admin/register	POST	
			originUnlocode: string destinationUnlocode: string arrivalDeadline: string
			trackingId: string
3	/admin/list	GET	
			cargoList: object array
4	/admin/show	GET	trackingId: string
			cargo: object
5	/admin/selectItinerary	GET	trackingId: string
			routeCandidates: object array cargo: object
6	/admin/assignItinerary	POST	
			trackingId: string legs: object array
7	/admin/pickNewDestination	GET	trackingId: string
			locations: object array cargo: object
8	/admin/changeDestination	POST	trackingId: string unlocode: string
9	/ws/RegisterEvent	POST	
			arg0: object
10	/track	GET	
			trackCommand: object
11	/track	POST	
			trackingId: string
			cargo: object

* The MIME types of the body and response for each API operation except 9 are `application/x-www-form-urlencoded` and `text/html`, respectively. In particular, the API operation 9 follows the SOAP protocol to exchange messages, thereby using `text/xml` as the MIME types of both body and response.

Table 3.2: Definitions of API operations exposed by the Cargo Shipping system (Part 2).

ID	Description	Entry Point	
1	Book a new cargo	se.citerus.dddsample. interfaces.booking.web. CargoAdminController	String registration(HttpServletRequest, HttpServletResponse, Map<String, Object>)
2	Register a cargo		void register(HttpServletRequest, HttpServletResponse, RegistrationCommand)
3	List all the cargoes		String list(HttpServletRequest, HttpServletResponse, Map<String, Object>)
4	Show a cargo		String show(HttpServletRequest, HttpServletResponse, Map<String, Object>)
5	Select an itinerary		String selectItinerary(HttpServletRequest, HttpServletResponse, Map<String, Object>)
6	Assign an itinerary		void assignItinerary(HttpServletRequest, HttpServletResponse, RouteAssignmentCommand)
7	Pick a new destination		String pickNewDestination(HttpServletRequest, HttpServletResponse, Map<String, Object>)
8	Change the destination		void changeDestination(HttpServletRequest, HttpServletResponse)
9	Submit a handling report	com.aggregator. HandlingReportService	void submitReport(HandlingReport)
10	Get the handling history	se.citerus.dddsample. interfaces.tracking. CargoTrackingController	String get(Map<String, Object>)
11	Show the handling history		String onSubmit(HttpServletRequest, TrackCommand, Map<String, Object>, BindingResult)

3.5.1 Semantic Similarity

In an API documentation, the semantics of an operation is expressed in the form of a natural language description, from which a human, usually a software developer, can infer valuable information about the operation such as what it does, how it works and when to use it. A pair of API operations are found correlated typically when their descriptions embody similar semantics. As a result, the semantic similarity is possibly an effective measure for discovering correlated API operations.

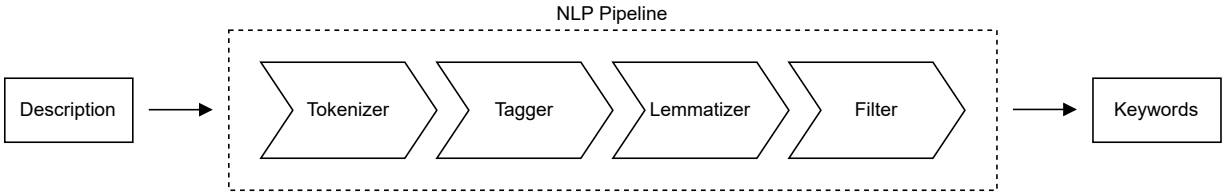


Figure 3.4: The NLP pipeline for extracting keywords from the description of an API operation.

To extract keywords from the description of an API operation, we build a natural language processing (NLP) pipeline as illustrated in Figure 3.4. This NLP pipeline consists of four components, namely tokenizer, tagger, lemmatizer and filter. The tokenizer aims to segment words, numbers and punctuation marks in the input description, which is the very first step of almost any NLP procedure [JM09]. Each word is then assigned a part-of-speech label by the tagger and normalized to its citation form by the lemmatizer. After that, the filter removes any words not labeled as adjectives, nouns or proper nouns. Those left are regarded as the keywords of the description. Take the API operation 1 of the Cargo Shipping system for example. Keywords extracted from the corresponding description, i.e. “Book a new cargo”, are “new” and “cargo”. There are many advanced methods available for keyword extraction [FNAD20]. Nevertheless, simply selecting nouns and noun phrases as the keywords is sufficient in our case, given that the description of an API operation is relatively short, one or two sentences in general.

Thanks to the recent development of techniques such as word2vec [MCCD13], GloVe [PSM14] and fastText [BGJM17], the semantics of a word can be accurately represented as a dense vector. Vector space models learned using these techniques are termed word embeddings. They

Table 3.3: The TF-IDF weighting scheme for computing the semantic vector of API operation i.

Definition*	**Description**	**Purpose**
$\mathrm{tf}(t, d_i) = \ln(\lvert\{w \in d_i \mid w = t\}\rvert) + 1$	Sublinear TF weight of keyword t in the description d_i of the API operation	Scale up the effect of recurring keywords
$\mathrm{idf}(t, \mathcal{D}) = \ln\left(\frac{\lvert\mathcal{D}\rvert}{\lvert\{d \in \mathcal{D} \mid t \in d\}\rvert}\right) + 1$	Empirical IDF weight of keyword t on the set $\mathcal{D}$ of descriptions in the API documentation	Scale down the effect of domain keywords
$\mathrm{idf}(t, \mathcal{A}) = \ln\left(\frac{\lvert\mathcal{A}\rvert}{\lvert\{a \in \mathcal{A} \mid t \in a\}\rvert}\right)$	Standard IDF weight of keyword t on the set $\mathcal{A}$ of articles in the English Wikipedia	Scale down the effect of general keywords

* The curly brackets in the definition of $\mathrm{tf}(t, d_i)$ symbolize bags; those in the definitions of $\mathrm{idf}(t, \mathcal{D})$ and $\mathrm{idf}(t, \mathcal{A})$ symbolize sets.

naturally satisfy additive compositionality, allowing the representation of a text by summing up the semantic vectors of all the words. Plain summation however does not capture the fact that different words contribute unequally to the overall semantics of the text. For example, "cargo" is a more important keyword than "new" in describing the API operation 1 of the Cargo Shipping system. The same problem is confronted in the task of information retrieval, where term frequency (TF) and inverse document frequency (IDF) weights come to aid [MRS08]. Inspired by this solution, we apply word embedding along with TF-IDF weighting and compute the semantic vector of API operation i as

$$\boldsymbol{v}_i = \sum_{t \in q_i} \mathrm{tf}(t, d_i)\mathrm{idf}(t, \mathcal{D})\mathrm{idf}(t, \mathcal{A})\boldsymbol{u}_t, \tag{3.1}$$

where d_i is the description of the API operation, q_i is the set of keywords extracted from d_i, $\mathcal{D}$ is the set of descriptions in the API documentation, $\mathcal{A}$ is the set of articles in the English Wikipedia, and $\boldsymbol{u}_t$ is the semantic vector of keyword t. Table 3.3 reports the TF-IDF weighting scheme adopted to compute the semantic vector of API operation i. The definitions of $\mathrm{tf}(t, d_i)$ and $\mathrm{idf}(t, \mathcal{D})$ follow practical forms[31] seen in industry rather than theoretical forms shown in textbooks. It is notable that the combination of word embedding and TF-IDF weighting has

[31] `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html`

led to success in various NLP tasks such as text classification [LZZ15]. Given the semantic vectors of two operations i and j, we use the half-wave rectified cosine of the included angle $\theta_{i,j}$ to measure their similarity:

$$s_{i,j}^{\text{sem}} = \max\{0, \cos\theta_{i,j}\} = \max\left\{0, \frac{\boldsymbol{v}_i \cdot \boldsymbol{v}_j}{\|\boldsymbol{v}_i\|_2 \|\boldsymbol{v}_j\|_2}\right\}, \tag{3.2}$$

which differs from the cosine similarity in the sense that opposite semantics are interpreted as being completely unrelated.

3.5.2 Structural Similarity

The business logic of a modern application is programmed in an object-oriented fashion, where business data and rules centering around a specific type of object are encapsulated as the fields and methods of the corresponding class respectively. It is often the case that a pair of correlated API operations depend on largely overlapping subsets of classes and exhibit similar structures. Therefore, the structural similarity may also be an effective measure for discovering correlated API operations.

There are basically two strategies, static and dynamic, to explore the structure of an API operation. The static strategy parses the source code into an abstract syntax tree and probes the structure of the API operation by iterating over all the nodes. The dynamic strategy carries out a functional test on the API operation and draws the structure from execution traces collected in different test cases. Both strategies have their own issue. Consider the problematic scenario shown in Figure 3.5. This is a real scenario encountered during the refactoring of the Cargo Shipping system. `CargoRepository` is an abstract class extended by `CargoRepositoryHibernate` and `CargoRepositoryInMem`. Objects of the two concrete classes are constructed at runtime through dependency injection and accessed by the API operation 1 and a unit test respectively via references of the abstract class. On one hand, it is impossible to verify whether the API operation depends on `CargoRepositoryHibernate` or `CargoRepositoryInMem` statically. On the other hand, the dependency of the API operation on `CargoRepositoryHibernate` is determinable dynamically, but `CargoRepository` would be

neglected as the class hierarchy of an object is normally not traced. We choose the dynamic strategy to explore the structures of API operations more accurately. The refactoring procedure described in Section 3.7 fills missing classes in the decomposition solution resorting to static dependency analysis.

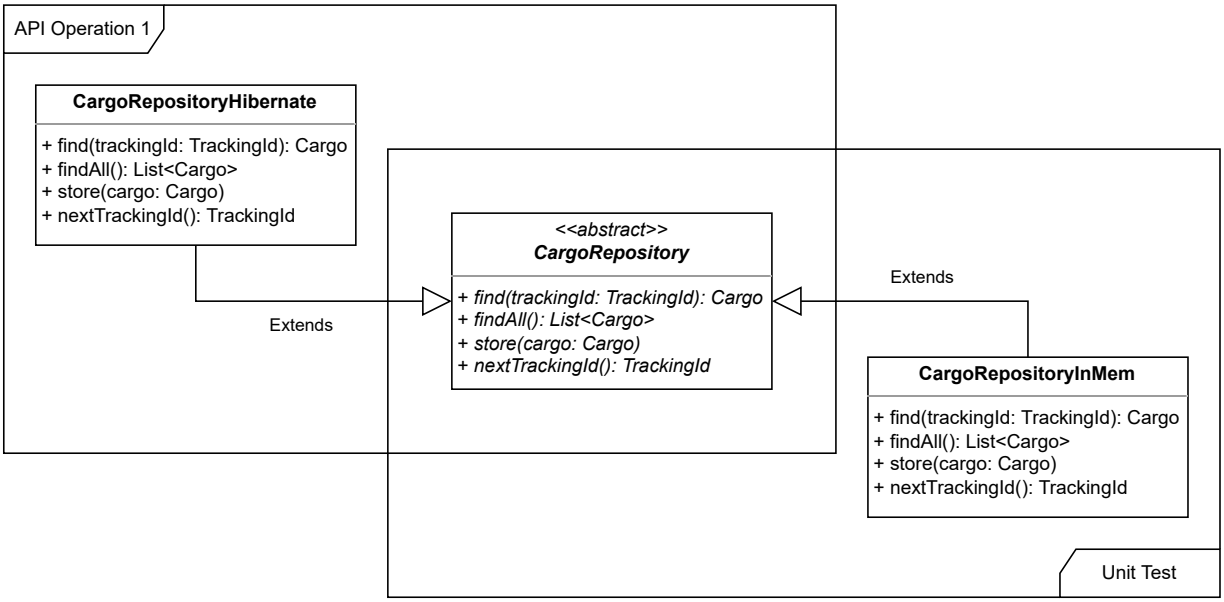


Figure 3.5: A scenario that cannot be properly dealt with by either static or dynamic strategy.

We are particularly interested in the subset of classes observed in the execution traces of each API operation, and call it the class trace for conciseness. Let $\mathcal{T}_i$ denote the class trace of API operation i. We define the structural similarity between two API operations i and j by

$$s_{i,j}^{\text{str}} = \frac{|\mathcal{T}_i \cap \mathcal{T}_j|}{\min\{|\mathcal{T}_i|, |\mathcal{T}_j|\}}, \tag{3.3}$$

which ranges from 0 if $\mathcal{T}_i \cap \mathcal{T}_j = \varnothing$ to 1 if $\mathcal{T}_i \subset \mathcal{T}_j$ or vice versa, measuring to what extent the smaller of $\mathcal{T}_i$ and $\mathcal{T}_j$ is contained in the larger. The similarity measure proposed in (3.3) is more sensitive than the Jaccard index in detecting a common API design pattern where an interaction between a client and the application is realized by using a pair of API operations to exchange forms. Take the API operations 10 and 11 of the Cargo Shipping system for example. These two API operations jointly enable a customer to inspect the handling history of a cargo with a given tracking ID. The former gets an empty form for the customer to enter the tracking ID of a cargo, while the latter shows the handling history of that cargo upon form submission. Table 3.4 compares the Jaccard index and the proposed measure on the API operations 10 and

11 of the Cargo Shipping system. Although the two API operations are obviously correlated, the Jaccard index of their class traces is merely 0.12. By contrast, the proposed measure results in a value of 1.00.

Table 3.4: Comparison of the Jaccard index and the proposed measure on the API operations 10 and 11 of the Cargo Shipping system.

Class Trace		**Jaccard Index**	**Proposed Measure**
API Operation 10	**API Operation 11**		
CargoTrackingController TrackCommand	CargoTrackingController TrackCommand ... (15 other classes)	0.12	1.00

3.5.3 Aggregation

We identify candidate microservices as groups of correlated API operations by means of clustering. Typical clustering algorithms assume the availability of a similarity or distance for each pair of data points. Thus, we aggregate the semantic and structural similarities between two API operations i and j as

$$s_{i,j} = \alpha s_{i,j}^{\text{sem}} + (1 - \alpha) s_{i,j}^{\text{str}}, \tag{3.4}$$

where α is a trade-off factor adjustable from 0 to 1. One may set α to be greater than 0.5 if the API documentation provides a precise description for every operation. A value of less than 0.5 is instead preferable in the case of an application whose source code is well structured. Since the similarity measure given by (3.4) is bounded between 0 and 1, we further define a distance measure as its complement to 1:

$$d_{i,j} = 1 - s_{i,j}. \tag{3.5}$$

Table 3.5 reports pairwise distances obtained by analyzing the API operations of the Cargo Shipping system with a word2vec model [MCCD13] and a trade-off factor α of 0.5. To show the implication in Table 3.5, we represent the API operations as data points in a two-dimensional space through non-metric multidimensional scaling [Kru64]. It is easy to find from Figure 3.6 four groups of correlated API operations: $\{1, 2, 3, 4\}$, $\{5, 6\}$, $\{7, 8\}$ and $\{9, 10, 11\}$, each corresponding to a candidate microservice for the Cargo Shipping system. This partition exactly matches the one suggested by experienced software designers in the evaluation experiments.

Table 3.5: Pairwise distances between the API operations of the Cargo Shipping system.

ID	1	2	3	4	5	6	7	8	9	10	11
1	0.0000	0.1615	0.3012	0.2240	0.5401	0.5401	0.3239	0.5153	0.6105	0.8565	0.6690
2	0.1615	0.0000	0.2403	0.1250	0.5030	0.4736	0.5020	0.4191	0.4848	0.8611	0.5670
3	0.3012	0.2403	0.0000	0.0933	0.3802	0.4236	0.4276	0.4496	0.4632	0.8782	0.5449
4	0.2240	0.1250	0.0933	0.0000	0.4147	0.5085	0.3844	0.4816	0.4922	0.8611	0.5486
5	0.5401	0.5030	0.3802	0.4147	0.0000	0.0577	0.3498	0.3199	0.6524	0.9476	0.6420
6	0.5401	0.4736	0.4236	0.5085	0.0577	0.0000	0.4287	0.2886	0.5774	0.9476	0.6420
7	0.3239	0.5020	0.4276	0.3844	0.3498	0.4287	0.0000	0.0922	0.6143	0.9531	0.6753
8	0.5153	0.4191	0.4496	0.4816	0.3199	0.2886	0.0922	0.0000	0.5660	0.9673	0.6548
9	0.6105	0.4848	0.4632	0.4922	0.6524	0.5774	0.6143	0.5660	0.0000	0.5965	0.2076
10	0.8565	0.8611	0.8782	0.8611	0.9476	0.9476	0.9531	0.9673	0.5965	0.0000	0.0000
11	0.6690	0.5670	0.5449	0.5486	0.6420	0.6420	0.6753	0.6548	0.2076	0.0000	0.0000

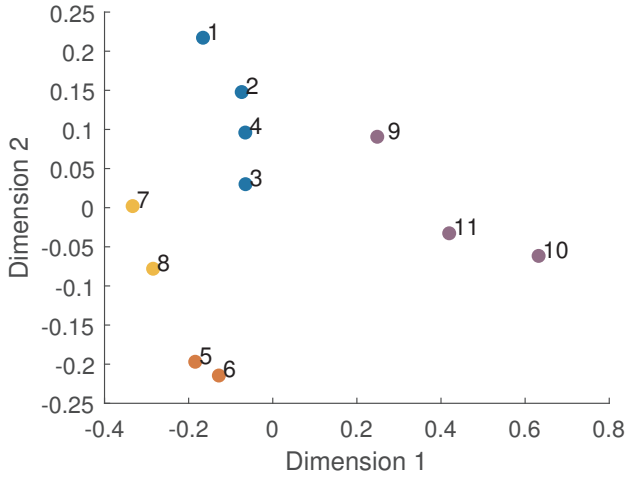


Figure 3.6: API operations of the Cargo Shipping system in a two-dimensional representation.

3.6 Microservice Identification

After the interface analysis, candidate microservices can be automatically identified by clustering API operations. There are plenty of clustering algorithms available in the literature [XT15]. We adapt hierarchical [Joh67], K-medoids [KR90], DBSCAN [EKSX96] and spectral [SM00] clustering algorithms for microservice identification, because these four algorithms do not have restrictions on the definition of similarities or distances between data points. Parameterization of the clustering algorithms is automated by optimizing an internal validity index, i.e. one computed based on the data set. Many internal validity indices have been devised, including Caliński-Harabasz [CH74], Dunn [Dun74], gamma [BH75], Davies-Bouldin [DB79], silhouette [Rou87], Krzanowski-Lai [KL88] and CDbw [HV08]. Unfortunately, most of them are not suitable in our case:

- the Caliński-Harabasz, Dunn and Davies-Bouldin indices are observed to be insensitive on the given data sets;
- the Krzanowski-Lai and CDbw indices may only be applied to data points explicitly defined on an vector space.

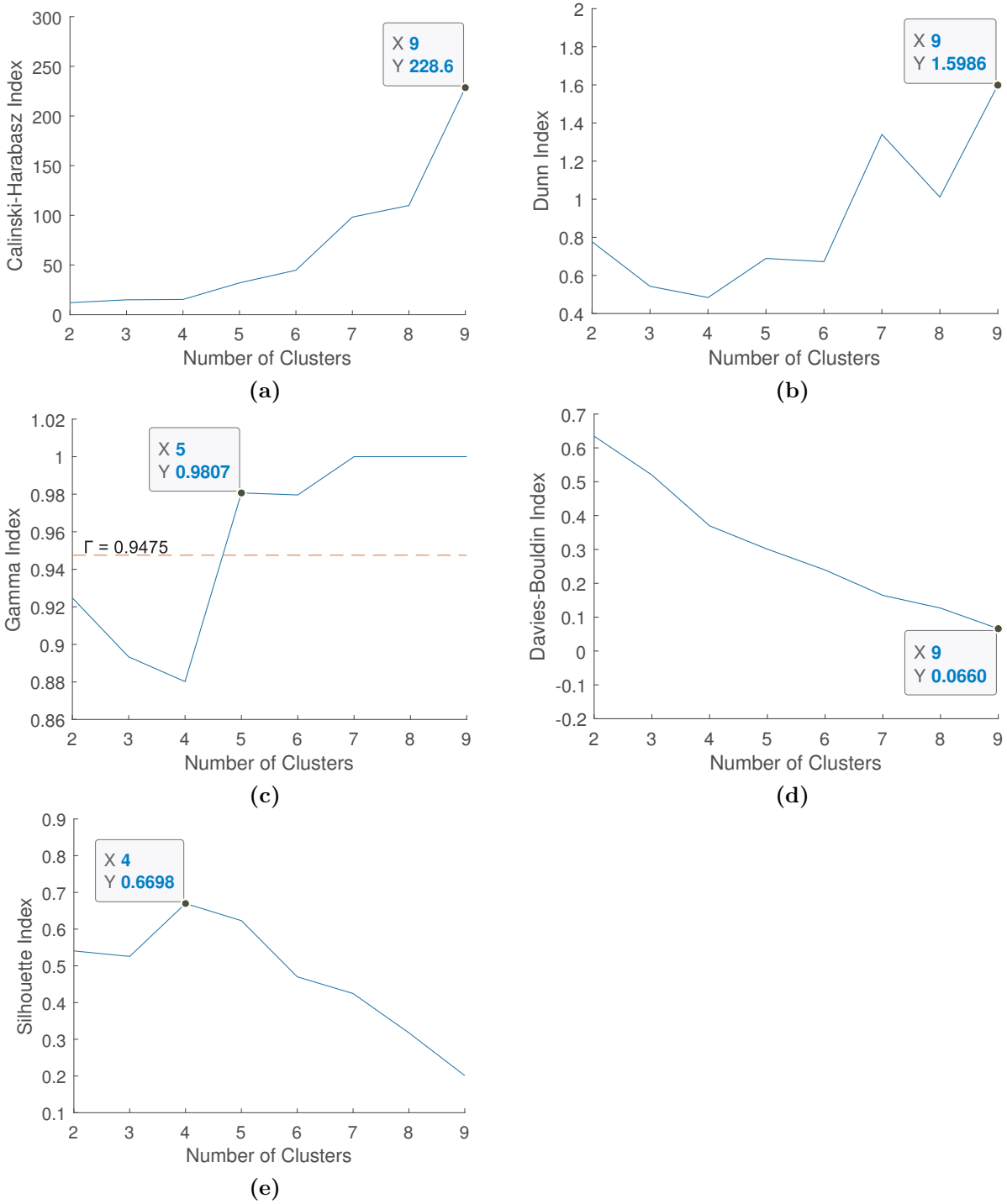


Figure 3.7: Caliński-Harabasz, Dunn, gamma, Davies-Bouldin and silhouette indices of partitions resulting from hierarchical clustering on the API operations of the Cargo Shipping system.

To evidence the first argument, we conducted the single-linkage hierarchical clustering on the API operations of the Cargo Shipping system, and computed the Caliński-Harabasz, Dunn, gamma, Davies-Bouldin and silhouette indices of the resultant partition as the number K of clusters to discover incremented from 2 to 9. These indices except Davies-Bouldin are optimal at the maximum value. In particular, we empirically set a threshold of 0.9475 for the gamma index and calculate the silhouette index according to (2.18). It can be seen from Figure 3.7 that the Caliński-Harabasz, Dunn and Davies-Bouldin indices seem to favor as many clusters as possible while the gamma and silhouette indices exhibit good effectiveness in seeking the right number of clusters. We therefore opt for the latter two indices.

3.7 Code Refactoring

In what follows, we outline a practical procedure for refactoring a monolithic application into one compatible with the FaaS paradigm. This procedure applies the two-stage refactoring strategy discussed in Section 3.4.1 and attains the target code structure at each decomposition stage based on the identified microservices and the class traces of API operations. Classes that can hardly be captured at runtime, such as pure abstract and exception classes, are also taken into account via static dependency analysis. The procedure consists of five sequential steps. For the sake of clarification, the last four steps are formulated in Algorithms 3.1, 3.2, 3.3 and 3.4 respectively. Table 3.6 summarizes notation used therein.

Table 3.6: Notation used in Algorithms 3.1, 3.2, 3.3 and 3.4.

Symbol	Definition
i, j	API operation indices
k, l	Microservice indices
p, q	Class indices
M	Number of API operations
K	Number of Microservices
N	Number of Classes
η	Proportion of microservices by which a global utility is depended on

$\mathcal{T}_i$	Set of classes observed in the execution traces API operation i
$\mathcal{O}_k$	Set of correlated API operations assigned to microservice k
$\mathcal{D}_\mathrm{I}$	Set of inheritance dependencies between classes
$\mathcal{D}_\mathrm{A}$	Set of access (non-inheritance) dependencies between classes
$\mathcal{U}_\mathrm{G}$	Set of global utilities shared among all the microservices
$\mathcal{E}_k$	Set of core elements implementing microservice k
$\mathcal{U}_k$	Set of local utilities shared among serverless functions composing microservice k
$\mathcal{E}_i$	Set of core elements implementing serverless function i
$\mathcal{L}$	Mutable set acting like a linked list
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}$	Temporary sets computed along the way
$\mathcal{S}(\cdot)$	An arbitrary member in a given set $\mathcal{S}$

Step 1. Split the entry point classes of the application so that fields and methods related to each API operation are encapsulated in a separate class. For example, the `CargoTrackingController` class of the Cargo Shipping system provides entry points for both API operations 10 and 11, and thus needs to be split into two classes possibly named `CargoTrackingGetController` and `CargoTrackingOnSubmitController`. This may demand significant effort for certain applications, subject to how tightly fields and methods within the same entry point class are coupled.

Step 2. For every class observed in the execution traces of API operations, decide whether it is a global utility or a core element of a microservice. We define a global utility as a class on which the dependence of microservices reaches a specified proportion η. Any class not satisfying this condition is regarded as a core element of the microservice with the most number of API operations dependent on it. When multiple alternatives are available, the one with the highest ratio of such API operations is chosen.

Algorithm 3.1: Deciding whether an observed class is a global utility or a core element of a microservice.

Input: M, K, η, $\mathcal{T}_i$, $\mathcal{O}_k$
Output: $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$

1: $\mathcal{U}_\mathrm{G} := \varnothing$
2: **for** $k \in [1\,..\,K]$
3: $\quad \mathcal{E}_k := \varnothing$
▷ Set of classes observed in the execution traces of API operations
4: $\mathcal{X} := \bigcup_{i=1}^{M} \mathcal{T}_i$
5: **for** $p \in \mathcal{X}$

▷ Set of microservices with API operations dependent on class p
$\mathcal{Y} := \{k \mid \exists i, i \in \mathcal{O}_k, p \in \mathcal{T}_i\}$
if $|\mathcal{Y}| \geq \eta K$
  $\mathcal{U}_\mathrm{G} := \mathcal{U}_\mathrm{G} \bigcup \{p\}$
else
  ▷ Set of microservices with the most number of API operations dependent on class p
  $\mathcal{Z} := \mathrm{argmax}_{k \in [1..K]} |\{i \mid i \in \mathcal{O}_k, p \in \mathcal{T}_i\}|$
  if $|\mathcal{Z}| > 1$
    ▷ Set of microservices with the highest ratio of API operations dependent on class p
    $\mathcal{Z} := \mathrm{argmax}_{k \in \mathcal{Z}} |\{i \mid i \in \mathcal{O}_k, p \in \mathcal{T}_i\}| / |\mathcal{O}_k|$
  $k := \mathcal{Z}(\cdot)$
  $\mathcal{E}_k := \mathcal{E}_k \bigcup \{p\}$

Step 3. For every class belonging to a microservice, decide whether it is a local utility of the microservice or a core element of a serverless function. We deem a class to be a core element of a serverless function if no API operations except the one handled by that serverless function depends on it. Otherwise, the class is put into local utilities shared among serverless functions comprising the microservice.

Algorithm 3.2: Deciding whether an observed class is a local utility of a microservice or a core element of a serverless function.

Input: M, K, $\mathcal{T}_i$, $\mathcal{E}_k$
Output: $\mathcal{U}_k$, $\mathcal{E}_i$

for $k \in [1\,..\,K]$
  $\mathcal{U}_k := \varnothing$
for $i \in [1\,..\,M]$
  $\mathcal{E}_i := \varnothing$
for $k \in [1\,..\,K]$
  for $p \in \mathcal{E}_k$
    ▷ Set of API operations assigned to microservice k and depending on class p
    $\mathcal{X} := \{i \mid i \in \mathcal{O}_k, p \in \mathcal{T}_i\}$
    if $|\mathcal{X}| > 1$
      $\mathcal{U}_k := \mathcal{U}_k \bigcup \{p\}$
    else
      $i := \mathcal{X}(\cdot)$
      $\mathcal{E}_i = \mathcal{E}_i \bigcup \{p\}$

Step 4. Incorporate classes that are not captured at runtime. These are mainly pure abstract and exception classes, which normally do not appear in the execution traces. We resort to static analysis to detect both inheritance and access (non-inheritance) dependencies between classes, and prioritize the former over the latter in making decomposition decisions. Any class not covered so far is treated as a global utility if it is depended on by another global utility

or the core elements of more than one microservice. Otherwise, we assign the class to the single microservice with classes dependent on it, and decide whether it is a local utility of the microservice or a core element of a serverless function following rules similar to those used in Step 3.

Algorithm 3.3: Incorporating classes that are not captured at runtime.[32]

Input: M, K, N, $\mathcal{T}_i$, $\mathcal{D}_\mathrm{I}$, $\mathcal{D}_\mathrm{A}$, $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$
Output: $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$

▷ Set of classes not observed in the execution traces of API operations
$\mathcal{L} := [1\,..\,N] \setminus \bigcup_{i=1}^{M} \mathcal{T}_i$
$size := 1$
while $|\mathcal{L}| < size$
  $size := |\mathcal{L}|$
  for $p \in \mathcal{L}$
    ▷ Set of classes inherited from class p
    $\mathcal{X} := \{q \mid (q, p) \in \mathcal{D}_\mathrm{I}\}$
    if $\mathcal{X} = \varnothing$
      ▷ Set of classes accessing class p
      $\mathcal{X} := \{q \mid (q, p) \in \mathcal{D}_\mathrm{A}\}$
    if $\mathcal{X} = \varnothing$ **or** $\mathcal{L} \bigcap \mathcal{X} \neq \varnothing$
      continue
    $\mathcal{L} := \mathcal{L} \setminus \{p\}$
    ▷ Set of microservices with classes dependent on class p
    $\mathcal{Y} := \{k \mid \mathcal{E}_k \bigcap \mathcal{X} \neq \varnothing\}$
    if $\mathcal{U}_\mathrm{G} \bigcap \mathcal{X} \neq \varnothing$ **or** $|\mathcal{Y}| > 1$
      $\mathcal{U}_\mathrm{G} := \mathcal{U}_\mathrm{G} \bigcup \{p\}$
    else
      $k := \mathcal{Y}(\cdot)$
      $\mathcal{E}_k := \mathcal{E}_k \bigcup \{p\}$
      if $\mathcal{U}_k \bigcap \mathcal{X} \neq \varnothing$
        $\mathcal{U}_k := \mathcal{U}_k \bigcup \{p\}$
      else
        ▷ Set of serverless functions composing microservice k and containing classes
        ▷ dependent on class p
        $\mathcal{Z} := \{i \mid i \in \mathcal{O}_k, \mathcal{E}_i \bigcap \mathcal{X} \neq \varnothing\}$
        if $|\mathcal{Z}| > 1$
          $\mathcal{U}_k := \mathcal{U}_k \bigcup \{p\}$
        else
          $i := \mathcal{Z}(\cdot)$
          $\mathcal{E}_i := \mathcal{E}_i \bigcup \{p\}$

Step 5. Address the inheritance and reverse dependency issue. It is unavoidable to introduce the following unrealistic dependencies in the previous steps:

[32] This algorithm tends to leave aside classes within an inheritance or access cycle.

- inheritance dependencies between the core elements of different microservices;
- reverse dependencies from the global utilities to the core elements of microservices.

Such dependencies can be resolved by iteratively moving the required classes into the global utilities.

Algorithm 3.4: Addressing the inheritance and reverse dependency issue.

Input: M, K, $\mathcal{D}_{\mathrm{I}}$, $\mathcal{D}_{\mathrm{A}}$, $\mathcal{U}_{\mathrm{G}}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$
Output: $\mathcal{U}_{\mathrm{G}}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$

▷ Set of inheritance dependencies between the core elements of different microservices
1: $\mathcal{X} := \{(q,p) \mid (q,p) \in \mathcal{D}_{\mathrm{I}} \wedge (\exists (l,k), q \in \mathcal{E}_l, p \in \mathcal{E}_k, l \neq k)\}$
2: **while** $\mathcal{X} \neq \varnothing$
3: **for** $(q,p) \in \mathcal{X}$
4: $k := \{k \mid p \in \mathcal{E}_k\}(\cdot)$
5: $\mathcal{E}_k := \mathcal{E}_k \setminus \{p\}$
6: **if** $p \in \mathcal{U}_k$
7: $\mathcal{U}_k := \mathcal{U}_k \setminus \{p\}$
8: **else**
9: $i := \{i \mid p \in \mathcal{E}_i\}(\cdot)$
10: $\mathcal{E}_i := \mathcal{E}_i \setminus \{p\}$
11: $\mathcal{U}_{\mathrm{G}} := \mathcal{U}_{\mathrm{G}} \bigcup \{p\}$
12: $\mathcal{X} := \{(q,p) \mid (q,p) \in \mathcal{D}_{\mathrm{I}} \wedge (\exists (l,k), q \in \mathcal{E}_l, p \in \mathcal{E}_k, l \neq k)\}$
▷ Set of reverse dependencies from the global utilities to the core elements of microservices
13: $\mathcal{Y} := \{(q,p) \mid (q,p) \in \mathcal{D}_{\mathrm{I}} \bigcup \mathcal{D}_{\mathrm{A}} \wedge q \in \mathcal{U}_{\mathrm{G}} \wedge (\exists k, p \in \mathcal{E}_k)\}$
14: **while** $\mathcal{Y} \neq \varnothing$
15: **for** $(q,p) \in \mathcal{Y}$
16: **repeat** 4–11
17: $\mathcal{Y} := \{(q,p) \mid (q,p) \in \mathcal{D}_{\mathrm{I}} \bigcup \mathcal{D}_{\mathrm{A}} \wedge q \in \mathcal{U}_{\mathrm{G}} \wedge (\exists k, p \in \mathcal{E}_k)\}$

3.8 Tool Support

We have developed the architecture decomposition feature of the RADON decomposition tool[11] to facilitate the adoption of RADF for refactoring a monolithic application into one consisting of serverless functions. Figure 3.8 illustrates the implementation of this feature. Given the TOSCA model of a monolithic application, the tool relies on an integrated YAML processor to read the service template into a MATLAB structure and generates a so-called topology graph through model-to-model transformation. This graph encompasses nodes, relationships and policies defined in the service template. A decomposition problem is then constructed from the topology graph and solved by running clustering algorithms made available in the statistics

& machine learning toolbox of MATLAB.[33] Particularly, the NLP pipeline and word embeddings for semantic analysis are built on the utilities of the MATLAB text analytics toolbox.[34] After the decomposition process is completed, the original service template will be updated accordingly.

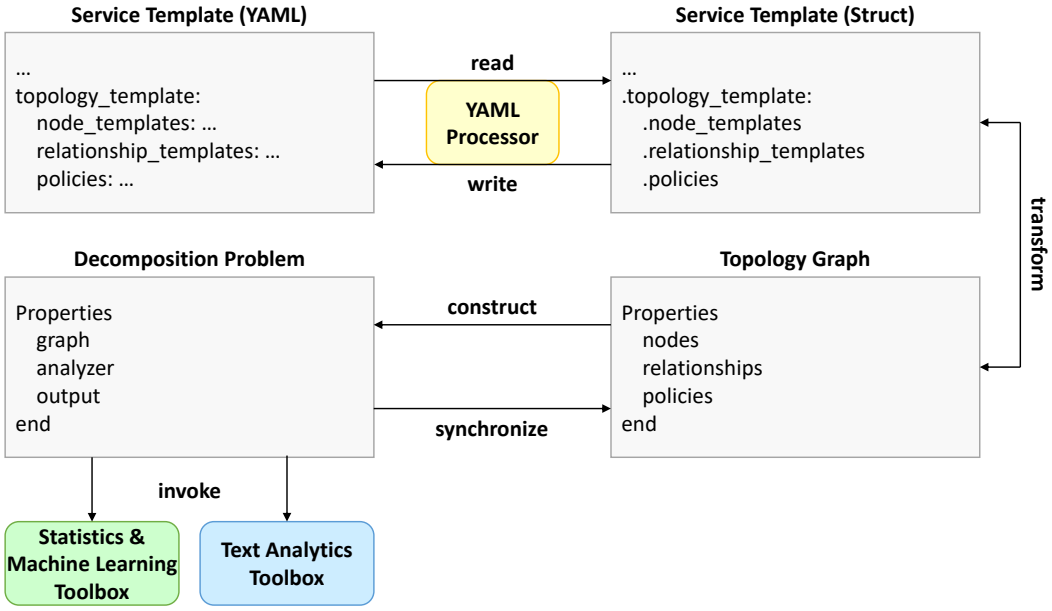


Figure 3.8: The architecture decomposition feature of the RADON decomposition tool.

3.9 Evaluation

This section reports an evaluation of RADF apropos its capability of identifying candidate microservices. Furthermore, we have evaluated the feasibility of RADF for code refactoring on the Cargo Shipping system. The final solution is compared with baselines obtained by applying other decomposition approaches.

3.9.1 Capability for Microservice Identification

Besides the Cargo Shipping system, six additional applications were selected from the GitHub codebase to evaluate the capability of RADF for microservice identification, including Pet

[33] https://www.mathworks.com/help/stats/index.html
[34] https://www.mathworks.com/help/textanalytics/index.html

Store,[35] Pet Clinic,[36] Polls App,[37] Spring React Blog,[38] Spring Boot Blog[39] and Shopping Cart.[40] These are all Java applications designed in a monolithic architecture. As a preparation, the interfaces of the applications were scanned and documented in a format similar to Tables 3.1 and 3.2. Two experienced software designers were invited to identify candidate microservices for each application according to the definitions of API operations and their understanding of source code. After that, a discussion took place to reach a consensus on the identified microservices, which formed the gold standard for evaluation.

Following the reference workflow of RADF, we conducted functional tests on API operations and collected execution traces with the BTrace tool.[41] The class trace of an API operation was acquired by putting together classes appearing in its execution traces. A TOSCA model annotated with the descriptions and class traces of API operations was created for every application and decomposed by the RADON decomposition tool at the microservice level using different clustering algorithms and validity indices. We also considered the special scenario where the parameters of clustering algorithms are manually set by human experts based on a priori knowledge. This helps to reveal the best performance of a clustering algorithm in the presence of human experts and the effectiveness of a validity index in automatically parameterizing a clustering algorithm.

RADF advocates analyzing the semantics of API operations via word embedding. In the experiments, we leveraged a word2vec model, GoogleNews-vectors-negative300 [MCCD13], pretrained on part of the Google News data set. The trade-off factor α was fixed at 0.5 all the time to balance between semantic and structural similarities. Table 3.7 lists the parameter settings of clustering algorithms for coarse-grained decomposition. We regarded the definition of the inter-cluster distance $D(C_k, C_l)$ as a parameter of hierarchical clustering due to its great impact on the behavior of the algorithm. The initialization of K-medoids is randomized. To guarantee reproducibility, we run the K-medoids algorithm multiple times and kept the partition with

[35] https://github.com/mybatis/jpetstore-6
[36] https://github.com/spring-petclinic/spring-framework-petclinic
[37] https://github.com/callicoder/spring-security-react-ant-design-polls-app
[38] https://github.com/keumtae-kim/spring-boot-react-blog
[39] https://github.com/reljicd/spring-boot-blog
[40] https://github.com/reljicd/spring-boot-shopping-cart
[41] https://github.com/btraceio/btrace

the smallest total deviation TD. The number R of times to repeat the clustering was increased exponentially from 5 to 80 until the total deviation TD no longer improved. A similar setup was enacted on the K-means algorithm for spectral clustering as well. The minimum neighbors m of a core point was specified as 1 in DBSCAN, which was to prevent non-core points from being classified as noise and consequently excluded out of any clusters. For the spectral clustering algorithm, the scaling factor σ of the Gaussian kernel was set to 0.6006. This choice would result in a distance $d(x_i, x_j)$ greater than 0.5 being mapped onto an edge weight $w(x_i, x_j)$ less than $1 - d(x_i, x_j)$ and vice versa.

Table 3.7: Parameter settings of clustering algorithms for coarse-grained decomposition.

Clustering Algorithm	**Clustering Parameter**	**Optional Values**
Hierarchical	Definition of inter-cluster distance: $D(C_k, C_l)$	(2.5)*, (2.6)*, (2.7)*
	Number of clusters to discover: K	$[2\,..\,M-1]^{\dagger}$
K-Medoids	Number of clusters to discover: K	$[2\,..\,M-1]^{\dagger}$
	Number of times to repeat the clustering: R	5, 10, 20, 40, 80
DBSCAN	Neighborhood of a data point: ϵ	$[0.5, 0]$
	Minimum neighbors of a core point: m	1
Spectral	Number of clusters to discover: K	$[2\,..\,M-1]^{\dagger}$
	Scaling factor of the Gaussian kernel: σ	0.6006
	Number of times to repeat the clustering: R	5, 10, 20, 40, 80

* These corresponds to the single, complete and average linkages, respectively.
† M denotes the total number of data points.

The computational complexity of RADF for microservice identification is subject to the scale of the application to be migrated as well as the word embedding, clustering algorithm and validity index in use, thereby being difficult to analyze. To understand RADF in this respect, we recorded average times spent in searching for the coarse-grained decompositions under different combinations of clustering algorithms and validity indices, as shown in Figure 3.9. The experiments were carried out on an Ubuntu server 20.04 LTS with 8 Intel(R) Xeon(R) CPU E3-1280 v5 @ 3.70 GHz and 64 GB RAM. It can be seen that RADF finds a coarse-grained decomposition solution on average in less than 1 s, which is almost negligible. Choosing the K-medoids or spectral clustering algorithm to automatically identify candidate microservices makes RADF slightly slower than in other cases, as the clustering is repeated many times to remove randomness inherent in the algorithm.

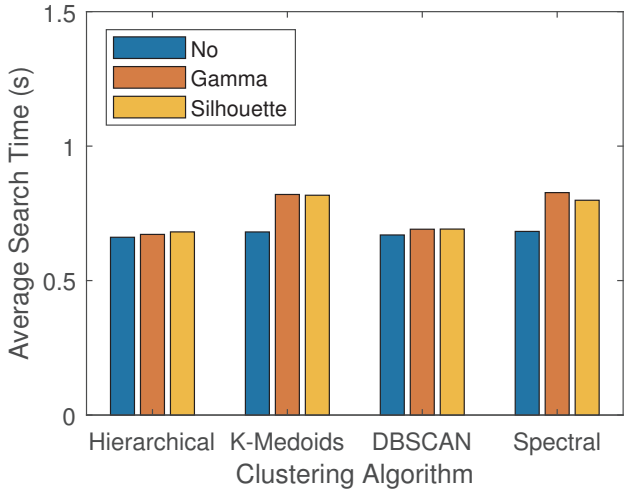


Figure 3.9: Average times spent in searching for the coarse-grained decomposition solutions under different combinations of clustering algorithms and validity indices.

Coarse-grained decomposition solutions generated under different combinations of clustering algorithms and validity indices were assessed against the gold standard in terms of the F1 score originating from [LA99]. Let C_k be the group of API operations assigned to microservice k in the gold standard, and let $C'_{k'}$ be that associated with microservice k' in a generated solution. We computed the precision, recall and F1 score of $C'_{k'}$ with respect to C_k as

$$P(C_k, C'_{k'}) = \frac{|C_k \cap C'_{k'}|}{|C'_{k'}|}, \tag{3.6}$$

$$R(C_k, C'_{k'}) = \frac{|C_k \cap C'_{k'}|}{|C_k|}. \tag{3.7}$$

$$F_1(C_k, C'_{k'}) = \frac{2P(C_k, C'_{k'})R(C_k, C'_{k'})}{P(C_k, C'_{k'}) + R(C_k, C'_{k'})}. \tag{3.8}$$

$P(C_k, C'_{k'})$ is the ratio of the shared API operations to those in $C'_{k'}$, while $R(C_k, C'_{k'})$ is the ratio of the shared API operations to those in C_k. $F_1(C_k, C'_{k'})$ is the harmonic mean of the two measures. The overall F1 score of the generated solution was defined by

$$F_1 = \sum_{k=1}^{K} \frac{|C_k|}{M} \max_{k' \in [1..K']} F_1(C_k, C'_{k'}), \tag{3.9}$$

where K and K' are the numbers of groups in the gold standard and the generated solution respectively. (3.9) essentially matches every group in the gold standard to one in the generated solution with the highest F1 score against it, and takes the weighted average of such F1 scores across all the groups in the gold standard.

Table 3.8: F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices.

Clustering Algorithm	**Validity Index**	**Application***							**Average**
		CS	PS	PC	PA	SRB	SBB	SC	
Hierarchical	No	1.0000	1.0000	1.0000	0.9117	1.0000	1.0000	1.0000	0.9874
	Gamma	0.9455	1.0000	1.0000	0.8788	0.7569	0.9121	0.9048	0.9140
	Silhouette	1.0000	1.0000	1.0000	0.9067	1.0000	0.9333	0.7857	0.9465
K-Medoids	No	1.0000	1.0000	1.0000	0.9067	1.0000	1.0000	1.0000	0.9867
	Gamma	0.8935	1.0000	1.0000	0.8788	1.0000	0.9455	1.0000	0.9597
	Silhouette	1.0000	1.0000	1.0000	0.9067	1.0000	0.9333	0.7460	0.9409
DBSCAN	No	1.0000	0.9216	1.0000	0.8788	1.0000	0.9455	1.0000	0.9637
	Gamma	0.9455	0.9216	1.0000	0.8788	1.0000	0.9455	1.0000	0.9559
	Silhouette	1.0000	0.9216	1.0000	0.7303	1.0000	0.9455	0.8095	0.9153
Spectral	No	1.0000	1.0000	1.0000	1.0000	1.0000	0.9333	0.9048	0.9769
	Gamma	0.9455	1.0000	1.0000	0.8788	1.0000	0.8788	0.9048	0.9440
	Silhouette	1.0000	1.0000	1.0000	1.0000	1.0000	0.9333	0.7460	0.9542

* CS: Cargo Shipping, PS: Pet Store, PC: Pet Clinic, PA: Polls App, SRB: Spring React Blog, SBB: Spring Boot Blog, SC: Shopping Cart.

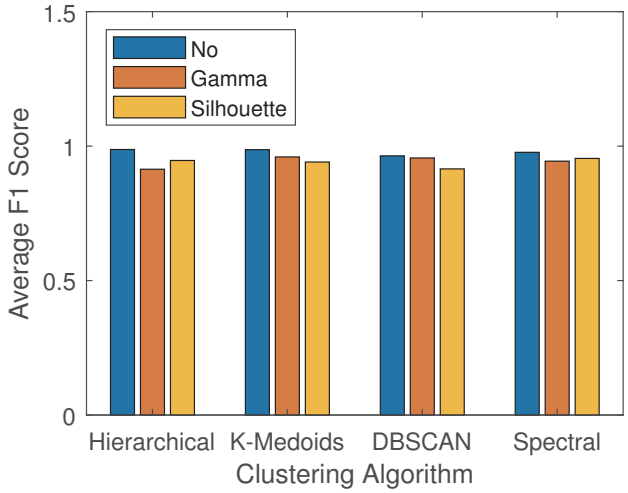


Figure 3.10: Average F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices.

Table 3.8 reports the F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices. Average values present in Table 3.8 are also visualized in Figure 3.10 for comparison. In the manual scenario, both hierarchical clustering and K-medoids yield the same candidate microservices as in the gold standard for six out of the seven applications. At the third place is the spectral clustering algorithm, which only fails in the last two cases. DBSCAN achieves an average F1 score of 0.9637 despite being the worst under manual parameterization. In the automatic scenario, the silhouette index typically

outperforms the gamma index for the first six applications but works much more poorly for the last one. This defeat is mainly because the Shopping Cart application has some standalone API operations that should been assigned to dedicated microservices. By convention, the silhouette value of a data point degenerates to 0 if it belongs to a singleton cluster. Maximizing the silhouette index tends to produce microservices with more than one API operation and thus eventuates in a bad decomposition solution to the application. Building on the above findings, Table 3.9 summarizes the preferable combinations of clustering algorithms and validity indices for coarse-grained decomposition.

Table 3.9: Preferable combinations of clustering algorithms and validity indices for coarse-grained decomposition.

Parameterization	**Standalone API Operations**	**Clustering algorithm**	**Validity Index**
Manual	No	Hierarchical/K-medoids/Spectral	No
	At least one	Hierarchical/K-medoids/DBSCAN	
Automatic	No	Hierarchical/Spectral	Silhouette
	At least one	K-medoids/DBSCAN	Gamma

3.9.2 Feasibility for Code Refactoring

In the aforementioned experiments, we identified four candidate microservices, namely Cargo, Planning, Location and Tracking, for the Cargo Shipping system using the hierarchical clustering algorithm and the silhouette index. The identified microservices were consistent with those suggested by the software designers and respectively responsible for four groups of API operations: $\{1, 2, 3, 4\}$, $\{5, 6\}$, $\{7, 8\}$ and $\{9, 10, 11\}$. To evaluate the feasibility of RADF for code refactoring, we employed again the RADON decomposition tool to further decompose the resultant TOSCA model at the function level, which took merely 0.0088 s on the same server and led to a fine-grained decomposition solution where each API operation was handled by an individual serverless function. The classes of the Cargo Shipping system were then reorganized by practicing the procedure described in Section 3.7 with a static code analysis tool called Sonargraph Architect.[42] In particular, we set the dependence proportion η to 1 so that a class was viewed as a global utility if and only if all the microservices expected its availability.

[42] `https://www.hello2morrow.com/products/sonargraph/architect9`

Table 3.10: Quality metrics for assessing the decomposition solution to the Cargo Shipping system.

Quality	Metric	Description
Coupling	Afferent Coupling (AC) [Mar03]	AC is defined as the number of classes in other modules, that depend on classes in this module, and thus quantifies the module's responsibilities to other modules.
	Efferent Coupling (EC) [Mar03]	EC is defined as the number of classes in other modules, that classes in this module depend on, and thus quantifies the module's requirements on other modules.
	Instability (I) [Mar03]	I is calculated as the ratio of EC to the sum of AC and EC, and measures the resilience of a module to change. It varies from 0 to 1, with values of 0 and 1 signifying completely stable and unstable modules respectively.
Cohesion	Relational Cohesion (RC) [Lar12]	RC refers to the ratio between the numbers of dependencies and classes within a module. Instance creation, class inheritance, method invocation and field access among others are all counted as dependencies.

Coupling and cohesion are two types of qualities that software designers are most concerned about in architecture decomposition. Coupling is the degree to which two modules are mutually dependent, whereas cohesion is the degree to which one module forms a logically atomic unit. In an ideal architectural design, all the modules should be not only loosely coupled from one another but also tightly cohesive on their own. We assessed the decomposition solution to the Cargo Shipping system according to four quality metrics, afferent coupling (AC) [Mar03], efferent coupling (EC) [Mar03], instability (I) [Mar03] and relational cohesion (RC) [Lar12], as detailed in Table 3.10. The same application and quality metrics had been used in prior work to evaluate other decomposition approaches including Service Cutter [GKGZ16], concept mapping [BGDR17], data flow-driven [LZJ+19] and multi-model-based [DEMF+21]. We selected these as the baseline approaches and collated their evaluation results from the literature to compare RADF with them.

Tables 3.11 and 3.12 report the quality metrics of decomposition solutions obtained by applying RADF and the baseline approaches to the Cargo Shipping system, respectively. As can be seen, the average afferent coupling among microservices arising from RADF is 2.25, which is 82.7% lower than the best value attained by previous approaches, i.e. 13.00 through concept mapping.

Table 3.11: Quality metrics of the decomposition solution obtained by applying RADF to the Cargo Shipping system.

Approach	**Microservice**	**Quality Metric**					
		AC	EC	I	RC	ACF	ECF
RADF	Cargo	2	7	0.78	11.7	0.50	1.75
	Planning	4	2	0.33	12.1	2.00	1.00
	Location	3	2	0.40	10.9	1.50	1.00
	Tracking	0	1	1.00	11.4	0.00	0.33
	Average	2.25	3.00	0.63	11.53	1.00	1.02

Table 3.12: Quality metrics of decomposition solutions obtained by applying the baseline approaches to the Cargo Shipping system.

Approach	**Microservice**	**Quality Metric**			
		AC	EC	I	RC
Service Cutter [GKGZ16]	Location	14	1	0.07	21.5
	Tracking	11	3	0.21	4.3
	Voyage & Planning	15	4	0.21	16.7
	Average	13.33	2.67	0.16	14.17
Concept Mapping [BGDR17]	Planning	16	2	0.11	20.3
	Product	13	4	0.24	1.8
	Tracking	15	5	0.25	14.9
	Trip	8	2	0.20	0.0
	Average	13.00	3.25	0.20	9.25
Data Flow-Driven [LZJ+19]	Cargo	13	4	0.24	1.8
	Planning	10	3	0.23	11.5
	Location	15	1	0.06	21.5
	Tracking	16	5	0.24	14.1
	Average	13.50	3.25	0.19	12.21
Multi-Model-Based [DEMF+21]	Preparation	23	0	0.00	14.0
	Handling	18	3	0.14	11.0
	Planning & Tracking	16	2	0.11	23.0
	Delivery	12	8	0.40	5.0
	Return	7	5	0.42	0.0
	Average	15.20	3.60	0.21	10.60

Although the solution produced by RADF looks the most unstable, it is by no means inferior to the baselines given that the high instability is largely due to a reduction in the afferent coupling instead of a growth in the efferent coupling. Unlike other approaches, RADF does not cause certain microservices to be loosely cohesive. Consider the worst cases in this respect. The Tracking microservice yielded by Service Cutter for example has a rational cohesion of 4.3, whereas that of the Location microservice resulting from RADF is 10.9. Since the goal of RADF is to decompose a monolith into serverless functions, Table 3.11 provides two additional

quality metrics, afferent coupling per function (ACF) and efferent coupling per function (ECF), as the indicators of how many incoming and outgoing dependencies still need to be resolved for each function.

3.10 Conclusion

A semi-automatic approach to architecture decomposition, named RADF, is proposed in this chapter. The proposed approach adopts a two-stage strategy to refactor a monolithic application into one that consists of serverless functions. In the first stage, we identify candidate microservices by analyzing the semantic and structural similarities between API operations and clustering them into separate groups accordingly. As for the second stage, a number of serverless functions are created to compose each identified microservice, one handling a single API operation. We have implemented RADF as part of a DevOps methodology following OASIS TOSCA and compared it with previous decomposition approaches in refactoring the well-known Cargo Shipping system. The proposed approach brings in a decomposition solution with lower coupling and relatively balanced cohesion against the baselines.

Chapter 4

RDOF: Deployment Optimization for Function as a Service

4.1 Introduction

Serverless computing promotes the idea of entirely bypassing user involvement in resource provision and management [BCC+17]. As a realization of serverless computing, function as a service (FaaS) allows developers to compose a cloud application with stateless functions that are executed and scaled automatically in response to the associated events [FIMS17]. Take the most prominent FaaS compute service—AWS Lambda[3]—for example. At runtime, Lambda keeps monitoring every type of event and dispatches the occurring events to their handler functions. A Lambda function has zero or more instances. New instances will be created to deal with extra incoming events, while existent instances will be deleted after being idle for some time. Resources available for the application are thus automatically increased and decreased on demand at the function level.

Thanks to this fine-grained autoscaling mechanism, public cloud providers charge their FaaS offerings based on a pay-as-you-go cost model [CIMS19]. The billing of a FaaS compute service is generally by a dimension of resource consumption, e.g. the memory and execution time of serverless functions, rather than by a dimension of resource allocation, e.g. the number and

size of virtual machines. Such a pay-as-you-go cost model is particularly economical when the application to be deployed is typically under a bursty workload. The main incentive for the adoption of serverless computing is indeed to save operating costs, which accounts for 47% applications investigated in [ESVE+21]. To this end, one needs to configure each function appropriately. An over-resourced deployment scheme may lead to a significant rise in operating costs, whereas an under-resourced deployment scheme possibly brings about a serious violation of performance requirements, e.g. service-level agreements.

How to minimize the total operating cost of a serverless application while satisfying the specified performance requirements is one of the remaining challenges in the context of DevOps for serverless computing [YBLW19]. Addressing this challenge calls for a performance model that can capture the behavior of a serverless application and an optimization algorithm that can find the optimal deployment scheme. Both are non-trivial and necessary. A serverless application consists of not only functions but also various backend-as-a-service (BaaS) components. To predict its performance and operating costs, the performance model should preserve details about the individual components of the application and interactions between these components. Parameters with impact on the performance and operating cost of a serverless function only have integer values. As a result, the optimization algorithm should solve an integer nonlinear programming problem in a reasonable time.

To facilitate cost optimization of serverless applications, we propose in this chapter a model-driven approach to deployment optimization for FaaS, RDOF;[16] which uses a layered queueing network (LQN) [WNPM95, RS08] to predict performance measures and a genetic algorithm (GA) [Gol89] to optimize deployment schemes. The evaluation results indicate that the relative error of the LQN in performance prediction is below 10% on Amazon Web Services (AWS) and the saving rate of RDOF is above 10% against a baseline approach that enforces an identical configuration for all the functions. The proposed approach has been implemented as the deployment optimization feature of the RADON decomposition tool;[11] whose input and output are TOSCA models [BBKL14] that can be consumed by a TOSCA-compliant orchestrator for increased automation.

The rest of the chapter is organized as follows. Section 4.2 describes a serverless application and reviews the related work to show the motivation behind RDOF. This application will act as a running example in subsequent sections. Sections 4.3 and 4.4 discuss the details of the performance model and the optimization approach, respectively. An introduction to the implementation of RDOF is given in Section 4.5. Section 4.6 reports an evaluation in terms of the accuracy of the performance model and the utility of the optimization approach. The applicability of RDOF is demonstrated in Section 4.7 through an industrial case study. Section 4.8 is dedicated to the conclusions.

4.2 Motivation

This section starts with an investigation of FaaS compute services available on the public cloud platforms. Then, we describe a typical serverless application to be used as a running example in the following sections and review related work on cost optimization of serverless applications to show the motivation for proposing RDOF.

4.2.1 FaaS Compute Services

AWS Lambda[3], Google Cloud Functions[4] and Microsoft Azure Functions[5] are three representative FaaS compute services on public cloud platforms. The resource provision and management of a Lambda function is fully automated. However, users can control the autoscaling of the function through configuration of memory and concurrency. These two parameters are both critical

Table 4.1: Maximum memory and concurrency imposed by AWS Lambda, Google Cloud Functions and Microsoft Azure Functions.

Parameter Limit	**AWS Lambda**	**Google Cloud Functions**	**Microsoft Azure Functions**
Maximum Memory	3008 MB per function instance	2048 MB per function instance	1536 MB per app instance
Maximum Concurrency	1000 per account (upgradable)	1000 per function	Unbounded

to the performance and operating cost of a serverless application. Table 4.1 lists maximum memory and concurrency imposed by the three FaaS compute services.

When a function is invoked, AWS Lambda and Google Cloud assign an event to an instance of that function. Each instance may only handle one event at a time. If all the instances of the function are busy, a new instance is created to process the event. Resources allocated to the new instance, including both memory and CPUs, are in proportion to the memory of the function, while the maximum number of instances that the function can have is specified by its concurrency.

Differently on Microsoft Azure, functions that belongs to the same function app are bundled together. A system component, called the scale controller, monitors the rate of incoming events to each function and heuristically decides whether to scale up or down the number of app instances. Each app instance may process multiple events simultaneously. Thus, there is no set limit on the number of concurrent executions.

The running example present in the next section is based on AWS Lambda, which in our experience is currently the most mature and popular among the three FaaS compute services. We validate the applicability of RDOF to real systems on this example. It is however worth pointing out that RDOF is extensible to Google Cloud Functions due to the similarity between the underlying FaaS implementations on AWS and Google Cloud.

4.2.2 Running Example

Lambda works along with other AWS services such as Simple Queue Service (SQS), API Gateway and Simple Storage Service (S3) to invoke functions. S3 is an object storage service with scalability, high availability and low latency. It allows AWS users to store untyped binary objects in a hierarchical, file-system-like structure. Each object in S3 is identified by a unique key and located in a virtual component called the bucket.

Figure 4.1 illustrates the TOSCA model of a serverless application, Thumbnail Generation, which is composed of two S3 buckets named `uploads` and `thumbnails` as well as a Lambda func-

tion named `create_thumbnail`. When an image is put into the `uploads` bucket, S3 produces an event to invoke the `create_thumbnail` function asynchronously. The `create_thumbnail` function then gets the image from the `uploads` bucket, resizes it into a thumbnail and puts the result into the `thumbnails` bucket.

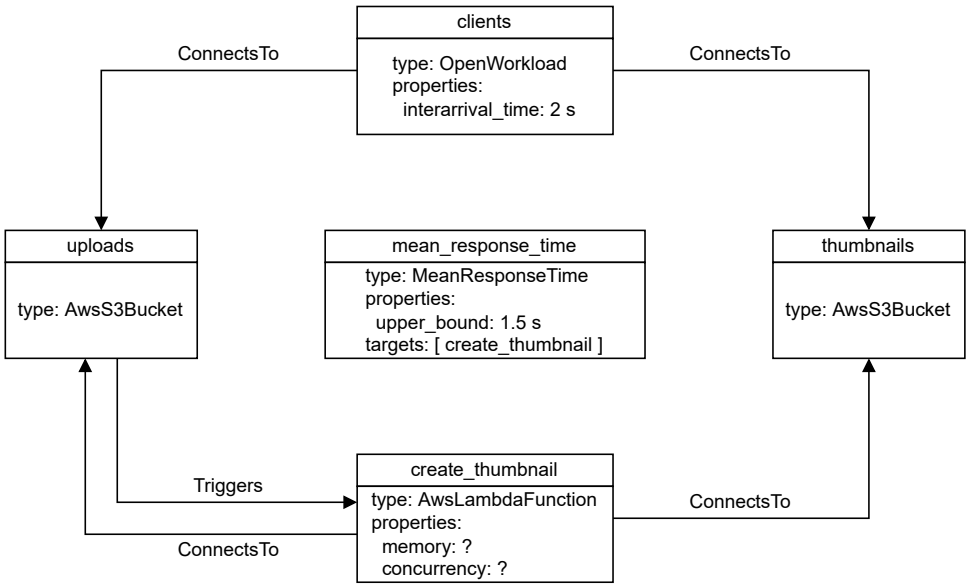


Figure 4.1: The TOSCA model of the Thumbnail Generation application.

As defined by the `clients` node, the reference workload of the Thumbnail Generation application is an infinite stream of clients with a mean interarrival time of 2 s. We assume that every client carries out a sequence of activities: (i) put an image into the `uploads` bucket; (ii) wait 10 s for the thumbnail to be ready; (iii) get the result from the `thumbnails` bucket; (iv) delete the image from the `uploads` bucket; (v) delete the result from the `thumbnails` bucket. The mean response time of the `create_thumbnail` function is expected to be less than 1.5 s. This performance requirement is specified through the `mean_response_time` policy.

To show the motivation behind RDOF, we used the performance and cost models present in Sections 4.3.1 and 4.4.1 respectively to predict the mean response time of the `create_thumbnail` function and to calculate the total operating cost of the Thumbnail Generation application for different memories but an infinite concurrency under the reference workload. As shown in Figure 4.2, the performance requirement can be satisfied with a minimum total operating cost of 0.030 $/h when the memory of the function is set to 256 MB. Although a memory configuration of 3008 MB is also feasible, the total operating cost increases by around 45%. The goal of RDOF is to solve this kind of deployment optimization problem.

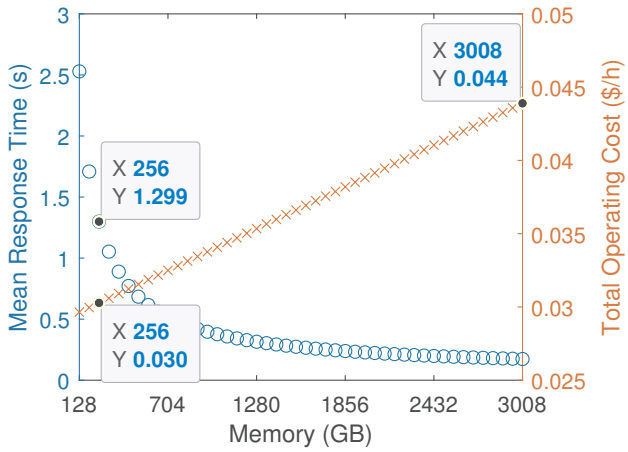


Figure 4.2: The mean response time of the `create_thumbnail` function versus the total operating cost of the Thumbnail Generation application.

4.2.3 Related Work

Merely a few publications attempt to tackle cost optimization of serverless applications. Tarek et al. [ESNA18] presented Costless, an approach for optimizing the price of a serverless workflow hosted across edge and cloud resources through function fusion and placement. They devised a price and an execution time model for such an workflow and found the optimal solution as the constrained shortest path of a so-called cost graph. Mahajan et al. [MFMR19] considered the scenario where a serverless function is combined with virtual machines to process an open workload. They represented the serverless function and each virtual machine respectively as an infinite-server and a single-server station with Poisson arrivals, and derived in closed form the best split of the workload that minimizes the cost. A framework named COSE was presented by Akhtar et al. [ARIM20] for optimizing the cost of running a function chain. COSE is composed of a performance modeler that uses Bayesian optimization to learn the relation between the performance/cost and memory of a function, and a config finder that obtains the best configuration for a chain of functions via integer linear programming. Lin and Khazaei [LK20] provided analytical models for predicting the end-to-end mean response time and cost of a serverless workflow under a memory configuration, and introduced the probability refined critical path greedy algorithm, a heuristic with four strategies aiming at the best performance under a given budget or the minimum cost for a desired performance. The previous approaches do not explicitly take into account BaaS components, which are an essential part of many serverless

applications and have non-negligible effects on both performance and operating costs. Also, none of the approaches can determine the concurrency needed by a serverless function. This parameter is not directly related to the operating cost of the function, but reserving a concurrency helps to prevent the function from being throttled by a surge of invocations to the others. Noticing the current gap, we propose RDOF as an alternative approach for optimizing the deployment of serverless applications. RDOF relies on an LQN to predict the performance of an application comprising serverless functions and various BaaS components. To reduce its computational complexity, the deployment optimization problem is divided into memory allocation and concurrency determination subproblems. RDOF solves these two subproblems resorting to a GA for nonlinear integer programming.

4.3 Performance Prediction

In what follows, we present the LQN used to predict the performance of serverless applications, with a particular focus on how to model all the components appearing in the running example and interactions between them. A service time estimation method, which enables the accurate parameterization of the LQN, is also introduced.

4.3.1 Layered Queueing Network

LQNs are a class of extended queueing networks suitable for describing software systems designed in a multi-tier architecture [WNPM95, RS08]. An LQN can be thought of as a directed graph, where each vertex represents either a software or a hardware entity and each edge represents a dependency of one entity on another. In LQNs, software entities are termed tasks while hardware entities are termed processors. Each task is hosted on a single processor and contains a set of entries, which often characterizes the interface of the software entity. Entries are defined as consisting of activities. An activity corresponds to a specific execution step and is allowed to send synchronous or asynchronous requests to entries of other tasks. Details about the notation and features of LQNs has been provided in Section 2.3.4.

The performance of the Thumbnail Generation application is essentially determined by the behavior of the `clients` node, the `uploads` and `thumbnails` buckets, the `create_thumbnail` function as well as the remote interactions between them. Workloads, either open or closed, can be represented in LQNs using a primitive construct called the reference task. We elaborate below the LQN structures for modeling S3 buckets, Lambda functions and remote interactions. Each task in these structures is hosted on a dedicated processor with the same multiplicity, and can thus be simply regarded as a station in ordinary queueing networks. For conciseness, we omit both textual and graphical depiction of processors. Some LQN structures contain immediate activities that demand zero service time from the hosting processor. Since LQNs with deterministic service times are not analytically tractable, every immediate activity is assigned a small mean service time of δS as an approximation.

Modeling S3 Buckets

We model an S3 bucket in LQNs as a task with an infinite multiplicity and are particularly concerned about three operations involved in the Thumbnail Generation application, i.e. GET, PUT and DELETE. As illustrated in Figure 4.3, each operation is represented by an individual entry of the task. Because how long an S3 bucket spends in responding to a request is opaque, the first activity of all the entries is defined to be an immediate activity. We take this into account in the network delay of the associated interaction. Upon completion of a PUT operation, the new object may not be instantly accessible from an S3 bucket. A second activity is introduced to exactly capture this two-phase behavior. The third activity is needed only when the bucket is configured to invoke a Lambda function afterward.

Modeling Lambda Functions

Figure 4.4 shows the LQN structure for modeling a Lambda function, which is a task with a multiplicity equal to the concurrency of the function. A single entry is defined in the task to depict the execution logic of the function as an activity graph. Several points are worthy of notice in using such an LQN structure to model a Lambda function:

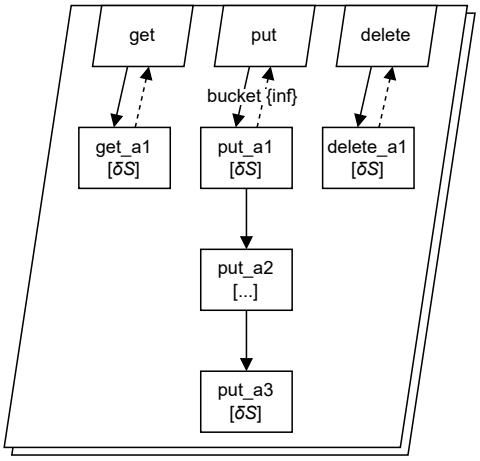


Figure 4.3: The LQN structure for modeling an S3 bucket.

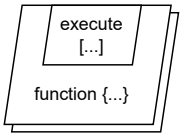


Figure 4.4: The LQN structure for modeling a Lambda function.

- *Cold start.* A function instance is deleted by Lambda after it has been idle or existent for a long time [MCPB20]. Then, a new instance may be created to process an incoming event, and an extra time is needed to establish the runtime environment of that instance. This is known as cold start. We ignore the effects of cold start in the proposed LQN structure by assuming the reference workload of the application is intensive such that the interarrival times of events to any instance almost never exceed the idle time of Lambda functions.
- *Service times.* Since the memory of a function specifies the amount of memory and CPU resources available for its instances, the mean service times of activities in the entry are inversely proportional to the configured memory when running an instance of the function does not require more memory resources. This relation not only enables us to accurately estimate the mean service times of function activities through linear regression across different memories but also tells how the optimization procedure should modify the LQN with respect to the memory configuration of functions.
- *Throttling and retrial.* If a function does not have enough concurrency to process all the incoming events, additional invocations are throttled. Lambda manages an event queue for each function and retries throttled invocations after a delay. The proposed LQN structure does not capture the retrial behavior of a function and therefore is valid only if invocations to the function are rarely throttled. This condition is equivalent to that requests to the task representing the function are almost never queued.

Modeling Remote Interactions

Remote interactions between a pair of source and target nodes are modeled by a task with an infinite multiplicity, as illustrated in Figure 4.5. Each entry of the task represents a single interaction, which may be either synchronous or asynchronous. Figure 4.6 shows the entry for modeling a synchronous interaction. Both requests and responses between the source activity and the target entry are forwarded by this entry. The network delay is incorporated into the activity of the entry as a service time, which results in it being divided in a balanced manner between any pair of request and response. Figure 4.7 shows the entry for modeling an asynchronous interaction. Only requests from the source activity to the target entry are forwarded by this entry. The first activity of the entry incorporates the network delay, while the second one is present to forward requests. Therefore, the network delay is fully pinned to any request.

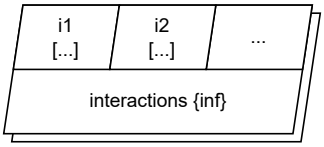


Figure 4.5: The LQN structure for modeling a group of remote interactions.

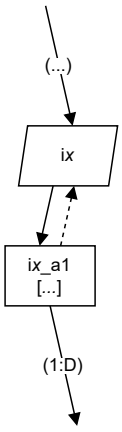


Figure 4.6: The entry for modeling a synchronous interaction.

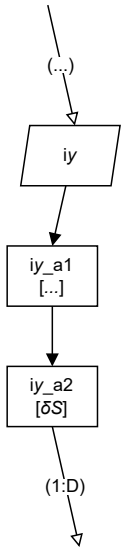


Figure 4.7: The entry for modeling an asynchronous interaction.

4.3.2 Service Time Estimation

Two methods are commonly used for service time estimation, namely utilization-based [SCBK15] and response-time-based [KPSCD09]. As mentioned in the last section, we model a Lambda function as a multi-server task where each server is corresponding to a function instance. Because the instances of a Lambda function are distributed among anonymous hosts that may change over time, it is difficult to measure quantities required by these methods. Lambda allocates dedicated resources to every instance of a function. The mean service times of activities performed by Lambda functions are invariant to the intensity of the workload. Thus, we can in fact estimate them through direct measurement. This simple method is also applicable to S3 buckets and remote interactions, which are modeled as infinite-server tasks that only introduce pure delays.

However, the service time of a function activity may not always directly measurable. Take the `create_thumbnail` function for example. This function carries out three sequential activities: (i) get the image from the `uploads` bucket; (ii) resize the image into a thumbnail; (iii) put the result into the `thumbnails` bucket. The execution time of the first activity includes the service time spent in completing the activity itself and the waiting time spent in accessing the `uploads` bucket, which are difficult to measured separately. Let $E_{i,k,r}$, $S_{i,k,r}$ and $W_{i,k,r}$ be the mean execution, service and waiting times of activity r in entry k at node i, respectively. In general, we have

$$E_{i,k,r} = S_{i,k,r} + W_{i,k,r}. \tag{4.1}$$

The mean service time of activity r at a function node i is in inverse proportion to the configured memory:

$$S_{i,1,r} = \frac{K_{i,1,r}}{m_i} \quad \text{for } i \in \mathcal{F}, \tag{4.2}$$

where $K_{i,1,r}$ is the corresponding proportionality constant, m_i is the memory of the function node i, and $\mathcal{F}$ is the set of function nodes. We reduce the entry index k to 1 in (4.2) because each function node has only one entry. The same convention will apply implicitly hereafter. It follows from (4.1) to (4.2) that

$$E_{i,1,r} = \frac{K_{i,1,r}}{m_i} + W_{i,1,r} \quad \text{for } i \in \mathcal{F}. \tag{4.3}$$

(4.3) and (4.2) provides a two-step method for service time estimation of a Lambda function. One can estimate the proportionality constant and the mean waiting time of an activity by fitting (4.3) through linear regression across different memories and then calculate the mean service time of the activity for a particular memory from (4.2).

4.4 Deployment Optimization

The approach for optimizing the deployment of serverless applications is elaborated in this section. Specifically, we formulate the deployment optimization problem and show how to divide it into two subproblems, namely memory allocation and concurrency determination, that can be solved efficiently.

4.4.1 Problem Formulation

The operating cost of a Lambda function is split into the costs of invocation requests, execution duration and data transfer. Lambda counts a request every time the function is invoked. The duration of a function execution is rounded up to the nearest multiple of 100 ms, and the price depends on the memory configuration. Data transferred to or from outside the AWS region where the function resides is charged as well.

Let $E_{i,k}$ and $X_{i,k}$ be the mean execution time and throughput of entry k at node i, respectively. We calculate the operating cost of a function node i as

$$C_i = (P^{\mathrm{FI}} + P^{\mathrm{FE}} m_i (E_{i,1} + 0.1)) X_{i,1} \quad \text{for } i \in \mathcal{F}, \tag{4.4}$$

where P^{FI} and P^{FE} are the prices for function invocation and execution. (4.4) does not take into account the cost of data transfer, which is often negligible compared to the others. Notably, the cost of execution duration given in (4.4) is an upper bound since the expectation of a rounded random variable is generally undecidable.

Object storages like S3 buckets is charged according to operation requests, space usage and data transfer. The operating cost of a storage node i is given by

$$C_i = \sum_k P_k^{\mathrm{SO}} X_{i,k} \quad \text{for } i \in \mathcal{S}, \tag{4.5}$$

where P_k^{SO} is the price for storage operation k, and $\mathcal{S}$ is the set of storage nodes. Assuming that objects will not be kept for a long time, we drop the costs of space usage and data transfer in (4.5). This assumption is true for the Thumbnail Generation application, as described in Section 4.2.

Let $C = \sum_i C_i$ be the total operating cost of the application to be deployed, and let $\boldsymbol{m} = (m_i)$ and $\boldsymbol{c} = (c_i)$ be the vectors of the configured memory and concurrency at each node i. We formulate the deployment optimization problem for an application composed of serverless functions and object storages as follows:

$$\begin{aligned} \min_{\boldsymbol{m},\boldsymbol{c}} \quad & C(\boldsymbol{m}, \boldsymbol{c}), \\ \text{s.t.} \quad & \boldsymbol{m} \in \{(m_i) \mid \forall i \in \mathcal{F}, m_i \in M\}, \\ & \boldsymbol{c} \in \{(c_i) \mid \forall i \in \mathcal{F}, c_i \in \mathbb{Z}_{>0}\}, \\ & \boldsymbol{a} \leq \boldsymbol{G}(\boldsymbol{m}, \boldsymbol{c}) \leq \boldsymbol{b}, \end{aligned} \tag{4.6}$$

where M is the set of memory options for function nodes, $\mathbb{Z}_{>0}$ is the set of positive integers, and $\boldsymbol{G}$ is the vector of quantities, e.g. the mean response time of a function node i, that are expected to be bounded between $\boldsymbol{a}$ and $\boldsymbol{b}$. (4.6) considers the concurrency of a function to be unlimited as the maximum concurrency imposed by a FaaS compute service is normally upgradeable or large enough.

4.4.2 Complexity Reduction

As shown in the previous section, the deployment optimization of a serverless application is an integer nonlinear programming problem, whose search space is extremely large even if the application comprises a few functions. Unlike the memory, the concurrency of a function does not contribute to its operating cost. An under-resourced concurrency configuration however can

significantly affect the performance of the function due to extra delays arising from throttling and retrial of invocations. The optimal deployment scheme should maximize the concurrency of each function as necessary. Based on this argument, we divide the overall optimization problem into two subproblems to reduce the computational complexity of searching for the optimal solution:

- *Memory allocation subproblem.* Solve for $\boldsymbol{m}^*$ such that

$$\begin{aligned} \min_{\boldsymbol{m}} \quad & C(\boldsymbol{m}, \boldsymbol{c}), \\ \text{s.t.} \quad & \boldsymbol{m} \in \{(m_i) \mid \forall i \in \mathcal{F}, m_i \in M\}, \\ & \boldsymbol{c} \in \{(c_i) \mid \forall i \in \mathcal{F}, c_i = \infty\}, \\ & \boldsymbol{a} \leq \boldsymbol{G}(\boldsymbol{m}, \boldsymbol{c}) \leq \boldsymbol{b}. \end{aligned} \tag{4.7}$$

- *Concurrency determination subproblem.* For each function node i, solve for c_i^* such that

$$\begin{aligned} \min_{c_i} \quad & c_i, \\ \text{s.t.} \quad & \boldsymbol{m} \in \{(m_j) \mid \forall j \in \mathcal{F}, m_j = m_j^*\}, \\ & \boldsymbol{c} \in \{(c_j) \mid \forall (j \in \mathcal{F} \wedge j \neq i), c_j = \infty\}, \\ & B_i(\boldsymbol{m}, \boldsymbol{c}) \leq \delta B, \end{aligned} \tag{4.8}$$

  where B_i is the mean buffer length of the function node i, and δB is a small upper bound for the mean buffer lengths of function nodes.

We obtain from (4.7) the optimal memory allocation by setting the concurrencies of all the functions to infinity. The solution to (4.7) is then used in (4.8) to determine the optimal concurrency of each function individually while the concurrencies of the others are fixed at infinity. We rely on the last constraint in (4.8) to ensure that invocations to functions are seldom throttled. This constraint is justifiable by presuming a function with infrequent throttling to behave like a task with a typically empty buffer. Let b_i be the buffer length of a function node i. The probability of requests to the function node i being queued has the following relation with its mean buffer length:

$$\mathbb{P}(b_i \geq 1) = \sum_{n=1}^{\infty} \mathbb{P}(b_i = n) \leq \sum_{n=1}^{\infty} \mathbb{P}(b_i = n) = \mathbb{E}(b_i) = B_i. \tag{4.9}$$

Applying (4.9) and (4.8) yields

$$\mathbb{P}(b_i \geq 1) \leq B_i \leq \delta B. \tag{4.10}$$

By equating a throttled invocation with a queued request, we know from (4.10) that the probability of invocations to the function node i being throttled would be nearly zero if its mean buffer length is restricted to a small upper bound δB.

Both memory allocation and concurrency determination subproblems are integer nonlinear programming problems. GAs are a powerful technique to deal with this kind of optimization problem. The search space of the memory allocation subproblem grows exponentially with the number of functions: $|M|^{|F|}$. To speed up the optimization procedure, we first use a derivative-based algorithm to find the continuous optimal solution, from which an initial population range is decided for GAs to search more effectively for the discrete optimal solution.

4.5 Tool Support

We have developed the deployment optimization feature of the RADON decomposition tool[11] to facilitate the adoption of RDOF for minimizing the total operating cost of a serverless application under the specified performance requirements. Figure 4.8 illustrates the implementation of this feature. With the help of a built-in YAML processor, the tool reads the service template of a given TOSCA model representing a serverless application into a MATLAB structure. This structure is further transformed into an extended directed graph incorporating nodes, relationships and policies, which we call a topology graph. An optimization problem is then constructed from the resultant topology graph and solved using the GA solver provided by the global optimization toolbox of MATLAB.[43] Particularly, the LQN for performance prediction is instantiated and analyzed via the LQNS solver of the LINE engine[13]. After the optimization process is completed, the original service template will be updated accordingly.

[43] `https://www.mathworks.com/help/gads/`

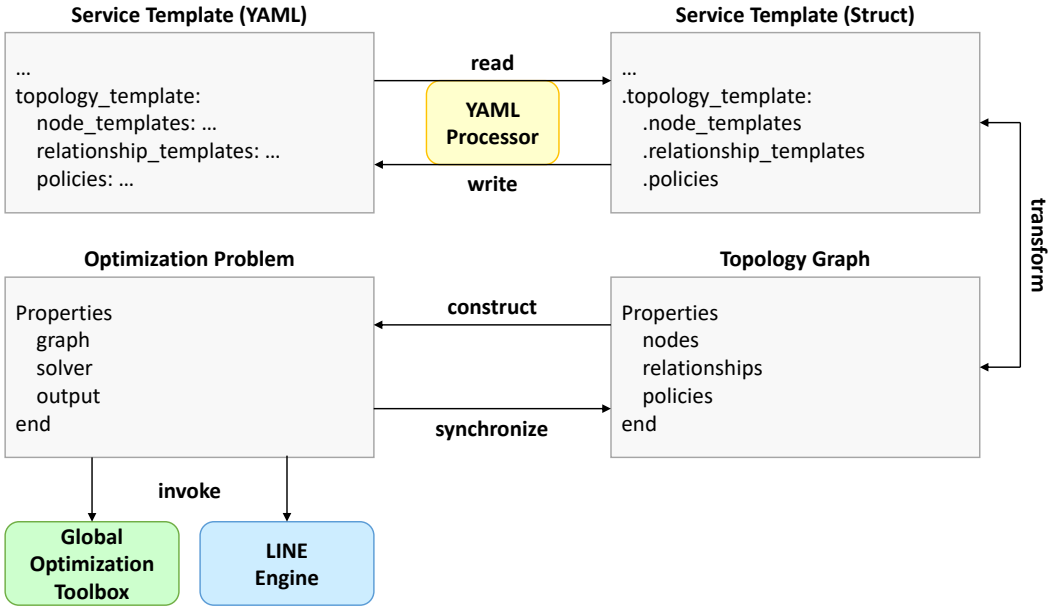


Figure 4.8: The implementation of the deployment optimization feature in the RADON decomposition tool.

4.6 Evaluation

In this section, we present an evaluation of RDOF with respect to the accuracy of the performance model and the utility of the optimization approach. The accuracy analysis is based on an actual deployment of the Thumbnail Generation application. As for the utility assessment, we compare RDOF with a baseline approach on a set of derived TOSCA models.

4.6.1 Accuracy of Performance Model

A benchmark of the Thumbnail Generation application was created to evaluate the accuracy of the LQN. We collected a set of 50 flower images from the Unsplash gallery[44] and used the Locust framework[45] to simulate a single client that repeatedly picked an image at random and put it to the `uploads` bucket. The memory and concurrency of the `create_thumbnail` function were set to 128 MB and 1. Service time estimation was conducted through direct measurement within a relative error of 5% at the 99% confidence level. For activities whose service times are not directly measurable, i.e. the first and third activities of the `create_thumbnail` function, we

[44] `https://unsplash.com`
[45] `https://locust.io`

estimated their mean execution and waiting times under two memory configurations, 128 MB and 320 MB, and then calculated their mean service times by applying (4.3) and (4.2).

The accuracy of the LQN was evaluated in terms of its relative error in predicting the mean response time of the `create_thumbnail` function. To this end, three groups of experiments were designed by varying the memory and concurrency of the function as well as the interarrival time of the reference workload. We parameterized the LQN accordingly based on the created benchmark and solved it analytically to predict the mean response time of the function. Particularly, the mean service times of function activities for different memories was estimated using (4.2), and the small mean service time δS assigned to every immediate activity was chosen to be 0.001 s. The relative errors of the predicted values were computed against those measured from the real system under the same statistical criterion as for the benchmark.

Accuracy evaluation results for the three groups of experiments are reported in Tables 4.2, 4.3 and 4.4. The relative error of the LQN in predicting the mean response time of the `create_thumbnail` function is no more than 10% in all the cases. Additionally, to what extent invocations to the function would be throttled is well anticipated by the degree to which requests to the corresponding task are queued. We fix the small upper bound δB for the mean buffer lengths of functions at 0.01 in the implementation of RDOF. It follows from (4.10) that the probability of invocations to any function being throttled would be less than or equal to 1% and thus sufficiently small to guarantee the accuracy of the LQN.

Table 4.2: Accuracy evaluation results for different memories but the same concurrency under the reference workload.

Memory	**Concurrency**	**Predicted Value**	**Measured Value**	**Relative Error**	**Task Queueing**	**Function Throttling**
192 MB	4	1.720 s	1.605 s	7.2%	$\leq 3.0\%$	$\leq 3.0\%$
256 MB		1.304 s	1.218 s	7.1%	$\leq 1.0\%$	$\leq 1.0\%$
320 MB		1.054 s	1.008 s	4.6%	$\leq 0.3\%$	$\leq 0.3\%$

Table 4.3: Accuracy evaluation results for the same memory but different concurrencies under the reference workload.

Memory	**Concurrency**	**Predicted Value**	**Measured Value**	**Relative Error**	**Task Queueing**	**Function Throttling**
256 MB	3	1.335 s	1.219 s	9.5%	$\leq 3.0\%$	$\leq 3.0\%$
	4	1.304 s	1.218 s	7.1%	$\leq 1.0\%$	$\leq 1.0\%$
	5	1.300 s	1.210 s	7.4%	$\leq 0.3\%$	$\leq 0.3\%$

Table 4.4: Accuracy evaluation results for different interarrival times under the optimal configuration (256 MB and 4).

Interarrival Time	Predicted Value	Measured Value	Relative Error	Task Queueing	Function Throttling
1 s	1.332 s	1.247 s	6.8%	$\leq 3.0\%$	$\leq 3.0\%$
2 s	1.304 s	1.218 s	7.1%	$\leq 1.0\%$	$\leq 1.0\%$
3 s	1.301 s	1.214 s	7.2%	$\leq 0.3\%$	$\leq 0.3\%$

4.6.2 Utility of Optimization Approach

We evaluated the utility of RDOF in terms of its saving rate against a baseline approach that can easily find a feasible solution. Similar to RDOF, this approach also divides the overall problem into two subproblems, memory allocation and concurrency determination. It sets the concurrencies of all the functions to infinity and increases their memories from the minimum up to the satisfaction of the specified performance requirements. The concurrency configuration is then obtained by incrementing the concurrencies of the functions from 1 until the mean buffer length of every function is no more than the small upper bound δB.

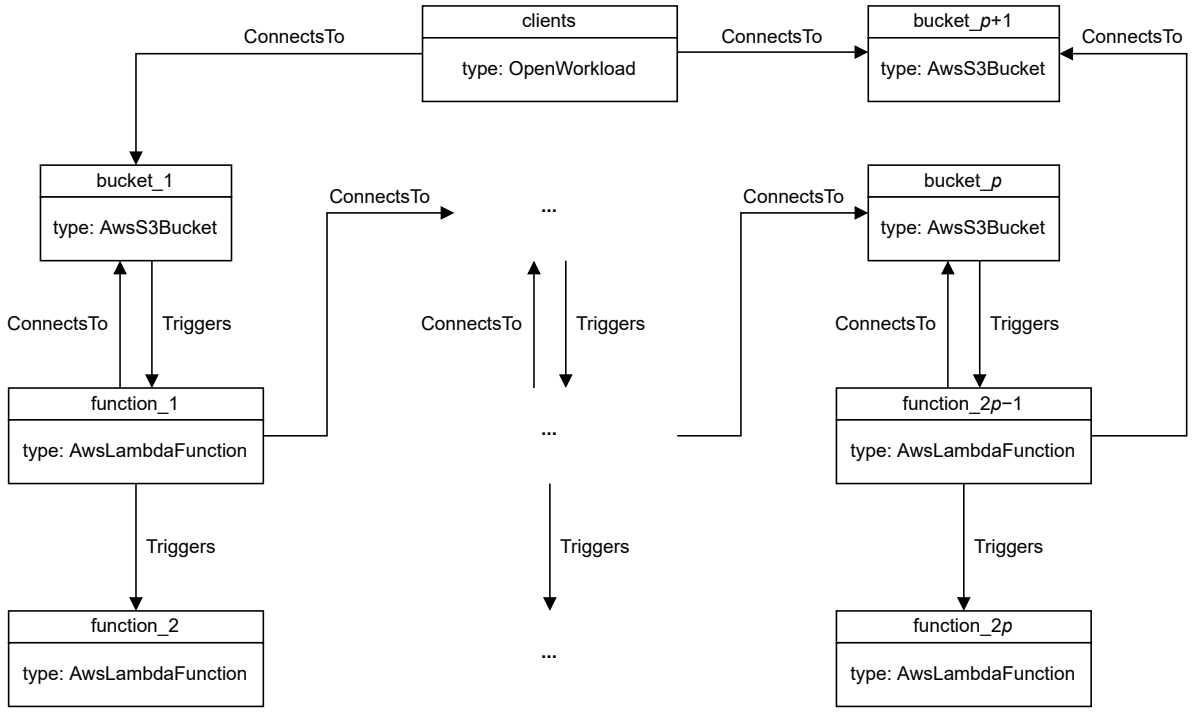


Figure 4.9: The topology of the derived TOSCA models.

The utility evaluation of RDOF was carried out on a set of 90 TOSCA models derived from that of the Thumbnail Generation application. Figure 4.9 illustrates the topology of these

TOSCA models, each defining a chain of p main functions with a synchronous invocation to their respective auxiliary function. A bucket was added in between two adjacent main functions to connect the output of the former to the input of the latter. The reference workload was the same as before except that the time spent by a client in waiting for the result was prolonged to $20p$ s. We reused the benchmark of the Thumbnail Generation application to parameterize the LQN. In particular, the mean time spent by a function in processing an object was uniformly sampled from a range of $[1, 4]$ s.[46] A upper bound of $3p$ s was imposed on the mean total response time of the main functions. Multiple models were generated by ascending the number p of main functions exponentially from 1 to 4 and the random seed from 1 to 30. Table 4.5 lists the parameter settings of the derived TOSCA models.

Table 4.5: Parameter settings of the derived TOSCA models.

Parameter	**Optional Values**
Number of main/auxiliary functions (denoted by p)	1, 2, 4
Number of functions	$2p$
Number of buckets	$p + 1$
Mean interarrival time of clients	2 s
Time spent by a client in waiting for the result	$20p$ s
Mean time spent by a function in processing an object	$[1, 4]$ s
Upper bound for the mean total response time	$3p$ s
Random seed	1, 2, . . . , 30

Experiments for the utility evaluation were performed on an Ubuntu server 20.04 LTS with 8 Intel(R) Xeon(R) CPU E3-1280 v5 @ 3.70 GHz and 64 GB RAM. In each experiment, RDOF and the baseline approach were used to respectively produce an optimal and a feasible deployment scheme for a derived TOSCA model. We recorded times spent in searching for the two solutions, and computed the saving rate of the optimal solution against the feasible one. Utility evaluation results for the derived TOSCA models with different numbers of functions are reported in Table 4.6. RDOF generally runs much slower than the baseline approach. This is however as expected, given the difficulty of the memory allocation subproblem. Deployment schemes found by RDOF save on average over 10% operating costs compared to those arising from the baseline approach, which is significant in practice.

[46] Given a derived TOSCA model, we generated the mean times spent by the functions in processing an object through the MATLAB command `round(3*gallery('uniformdata',[scale,1],seed)+1,3)`, where `scale` is the number of functions and `seed` is the random seed in use.

Table 4.6: Utility evaluation results for the derived TOSCA models with different numbers of functions.

Number of Functions	Optimization Approach			Baseline Approach			Average Saving Rate
	Average Search Time	Average Total Memory	Average Total Concurrency	Average Search Time	Average Total Memory	Average Total Concurrency	
2	8.3 min	894 MB	7.5	0.3 min	508 MB	10.1	11.7%
4	22.7 min	1734 MB	15.8	0.4 min	1024 MB	22.7	13.9%
8	177.9 min	3465 MB	32.0	0.7 min	2082 MB	47.5	14.9%

It is counter-intuitive that RDOF tends to allocate more memory to functions, since the operating cost of a function grows with the configured memory as in (4.4). To dispel the doubt, we inspected the obtained optimal and feasible solutions to the derived TOSCA model with two functions and random seed 1. As shown in Table 4.7, although the deployment scheme resulting from RDOF needs more memory, it actually costs 12.9% less than the one yielded by the baseline approach. Notably in the optimal solution, most of the memory is devoted to the auxiliary function, leading to a relatively short mean execution time. The rationale behind such a decision is that the auxiliary function is invoked by the main function in a synchronous manner and the execution time of the former is charged one more time as part of the execution time of the latter.

Table 4.7: The obtained optimal and feasible solutions to the derived TOSCA model with two functions and random seed 1.

Solution	Node	Memory	Concurrency	Mean Execution Time	Mean Throughput	Operating Cost*
Optimal	`function_1`	256 MB	5	2.491 s	0.500 r/s	5.499×10^{-6} \$/s
	`function_2`	1024 MB	2	0.389 s	0.500 r/s	4.176×10^{-6} \$/s
	Total	1280 MB	7	–	–	9.675×10^{-6} \$/s
Feasible	`function_1`	320 MB	6	2.945 s	0.500 r/s	7.771×10^{-6} \$/s
	`function_2`	320 MB	6	1.245 s	0.500 r/s	3.343×10^{-6} \$/s
	Total	640 MB	12	–	–	1.111×10^{-5} \$/s

* The prices for function invocation and execution are 2×10^{-7} \$/r and 1.628×10^{-8} \$/(MB·s), respectively.

4.7 Case Study

To demonstrate its applicability, RDOF has been adopted to deployment optimization of a use case from industry: Natural Language Processing (NLP) Pipeline. This application con-

sists of a chain of two Lambda functions, `ner` and `pos`, connected through three S3 buckets, `pipeline_in`, `pipeline_mid` and `pipeline_out`. Figure 4.10 illustrates the TOSCA model of the NLP Pipeline application. The reference workload of the application is defined by the `clients` node, which puts posts as JSON files into the `pipeline_in` bucket with a mean inter-arrival time of 0.1 s. Once a new JSON file is available, the `ner` function is invoked to get that file from the `pipeline_in` bucket and recognizes various named entities in the post. The JSON file is then updated and put into the `pipeline_mid` bucket. Following a similar procedure, the `pos` function parses the post and enriches the JSON file with part-of-speech tags. The final result will be archived in the `pipeline_out` for future analysis. The `ner` and `pos` functions are also responsible for deleting the original JSON files from their input buckets. As specified by the `total_mean_response_time` policy, the total mean response time of the `ner` and `pos` functions is expected to be no more than 5 s.

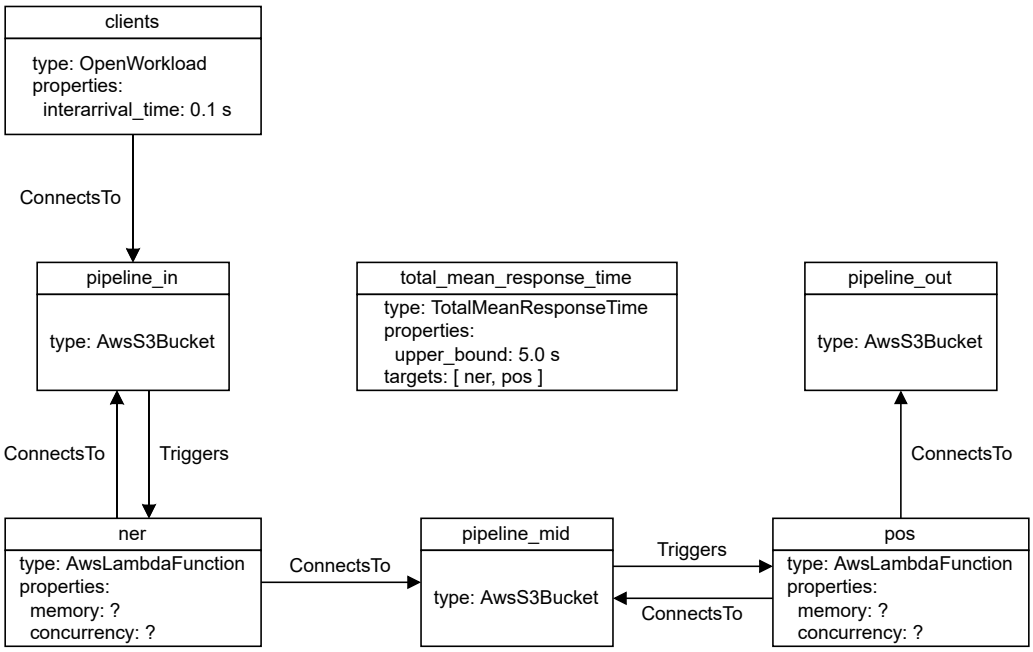


Figure 4.10: The TOSCA model of the NLP Pipeline application.

We first created a benchmark of the NLP Pipeline application to parameterize the LQN. A set of 850 posts was collected, and the Locust framework was used to simulate a synchronized client that randomly picked a post and put it as a JSON file into the `pipeline_in` bucket only after the result of the previous post was ready in the `pipeline_out` bucket. We set the memory and concurrency of both `ner` and `pos` functions to be 256 MB and 1, because at least

256 MB memory is needed to run an instance of the two functions. This introduced an extra constraint in the memory allocation subproblem. We estimated the mean service times of the activities through direct measurement within a relative error of 5% at the 99% confidence level. In particular, an additional measurement under a memory configuration of 768 MB was carried out to compute the mean service times of function activities by applying (4.3) and (4.2). The small mean service time δS of every immediate activity was chosen to be 0.001 s.

The RADON decomposition tool is then used to optimize the deployment of the NLP Pipeline application. The optimal memory and concurrency configurations of the `ner` and `pos` functions are found to be 512 MB, 41 and 576 MB, 37 respectively. To validate this solution, we configure the actual deployment of the application on AWS accordingly and carry out a load test using again the Locust framework to simulate the reference workload. The total mean response time of the two functions is measured to be 4.529 s under the same statistical criterion as for the benchmark, thus satisfying the specified performance requirement. In addition, the value predicted by the LQN is 4.960 s with a relative error of 9.5%.

4.8 Conclusion

A model-driven approach to deployment optimization for FaaS, named RDOF, is proposed in this chapter. RDOF can find the optimal deployment scheme of a serverless application that minimizes the total operating cost under the specified performance requirements. We model the performance of serverless applications using an LQN, which shows a relative error of less than 10% on AWS. To reduce computational complexity, the deployment optimization problem is divided into two subproblems so that the memory and concurrency configurations of functions are decided separately. RDOF has been compared with a baseline approach that obtains a feasible solution by treating all the functions identically, and achieves a saving rate of greater than 10% against this simple approach in the evaluation experiments. For conciseness, the scope of discussion in the chapter is confined to only two applications comprising Lambda functions and S3 buckets. The applicability of RDOF have however been validated on more industrial

use cases, such as Routes Calculator and Todo List API, which leverage various cloud services including Google Map Directions,[47] Amazon API Gateway[48] and Amazon DynamoDB.[49]

[47] https://developers.google.com/maps/documentation/directions
[48] https://aws.amazon.com/api-gateway/
[49] https://aws.amazon.com/dynamodb/

Chapter 5

Fluid Approximation of Closed Queueing Networks with Discriminatory Processor Sharing

5.1 Introduction

The vast majority of general-purpose operating systems, such as Linux, schedule multiple tasks on a single processor in a time-sharing manner such that each task in turn gains a time slice of the processor per cycle. Similarly in packet-switched communication networks, the bandwidth of a link is distributed among all flows passing through it, and accordingly the rate reserved for each flow varies with respect to the total number of flows on the link. A number of scheduling disciplines are available in queueing theory to abstract these resource allocation mechanisms. The best-known one is perhaps the egalitarian processor-sharing (EPS) discipline, which assumes that all the jobs equally share the service capacity of the system. In spite of its mathematical tractability, the EPS model is not appropriate to describe systems that differentiate between jobs with heterogeneous characteristics. Two multi-class variants of EPS have thus been developed, namely generalized processor sharing (GPS) [PG93, PG94] and discriminatory processor sharing (DPS) [Kle67]. The GPS discipline guarantees each running class a minimum

portion of the service capacity. As for DPS, the ration of the service capacity for a class is not guaranteed but rather depends on the number of jobs currently in each class.

The DPS discipline is widely applied in performance modeling of computer and communication systems. It naturally idealizes the weighted round-robin algorithm, which is often implemented in time-sharing operating systems to schedule prioritized tasks [SG98]. Such an idealization is reasonable if the time slice gained by each task per cycle is small compared to its total processing time and if the temporal overhead spent in switching between two consecutive tasks is negligible.

Modern communication networks also adapt the quality of service for different groups of customers. This feature is well captured by the DPS discipline. For example, a single-station DPS model is used in [MR00] to evaluate the flow-level performance of unequal bandwidth sharing in the Internet. More applications of DPS can be found in [BT01], [AJK04], [CvdBB$^+$05], [HT05] and [MC17]. Additionally, we mention [AABNQ05], [AAA06] and [AAB$^+$07] as surveys on the DPS discipline overviewing relevant applications and results in the literature.

No product-form solution has been established for general queueing models with DPS. That is, the steady-state probabilities are not known to be decomposable into simple products, as instead available for EPS by the BCMP theorem [BCMP75]. Partly motivated by the need for insights into the performance of modern communication networks, much effort has been put into understanding the behavior of single-station DPS models. There is however a shortage of results on the DPS discipline for closed queueing networks (QNs), which are useful to model, for example, complex software systems with admission control or threading limits on the interactions between separate software layers [RS95]. To fill this gap, we propose a fluid approach for analyzing the transient and steady-state behavior of closed QNs that consist of delay and DPS stations. Our reference model accommodates for an arbitrary routing topology among the stations. It also assumes the service time distributions for each class to be of phase type and allows a job to switch from one class to another upon service completion at a station.

The proposed approach is based on Kurtz's theorem [Kur70], whereby a closed QN could be casted as a *density-dependent* Markov process and solved by a system of ordinary differential

equations (ODEs). As such, the discrete population of jobs in the model is approximated as a continuous amount of fluid. The ODE system resulting from fluid approximation of a QN with DPS is discontinuous when a DPS station is empty. To address this technical issue, we modify the considered model by the notion of *self-looping* jobs [CeSL89], i.e. additional jobs that permanently loop at a queueing station, consuming a fraction of the service capacity. Self-looping jobs have recently proven to be helpful in simplifying analysis of product-form closed QNs [Cas17]. Unlike previous work, we show how this notion is applicable to closed QNs with DPS, for which the product-form solution is unknown yet. Such a modification results in a stable and accurate fluid approximation to the original model under the condition that all the DPS stations are heavily loaded. This has been validated by simulation on typical examples for transient analysis and on a large set of model instances for steady-state analysis.

One key benefit of transient analysis techniques is that they often imply a feasible way to compute the response time distribution of a tagged job at an individual station or for the entire system. Along with the proposed approach, we introduce a refined method for approximating response time distributions at the station level and further extend it to enable approximation of system response time distributions with respect to a delay station. This method natively copes with class switching by the concept of *chains* [Mun73] and normally outperforms prior art [PC17] with significantly higher accuracy for general phase-type service time distributions.

Capacity planning is an essential but difficult task for private cloud users who want to host a serverless application on physical infrastructure owned or rented by themselves. The main difficulty lies in understanding how the performance of the application changes with the workload intensity and allocated resources. To help in tackling this problem, we have carried out a case study in which the fluid approach is applied to performance prediction of serverless functions on a private cloud. A closed QN composed of delay and EPS stations is devised to model the performance of the OpenFaaS[6] framework on a Docker[10] swarm. We have evaluated the accuracy of the model, showing that its relative error in predicting the mean response times of OpenFaaS functions is below 10%. As a side effect of the extra self-looping jobs, the resultant accuracy may be further improved when the model is solved with the fluid approach.

The rest of the chapter is organized as follows. Section 5.2 outlines related work on the DPS discipline. Our reference model is defined in Section 5.3, whereas the fluid approach is formulated in Section 5.4. Section 5.5 describes the method for approximating response time distributions. Section 5.6 presents the experiment settings and results of the validations. The case study on performance modeling of the OpenFaaS framework is included in Section 5.7. Section 5.8 is dedicated to the conclusions. Table 5.1 summarizes contextual notation used throughout this chapter.

Table 5.1: Contextual notation used throughout Chapter 5.

Symbol	Definition
i, j	Station indices
k, l	Class indices
r, s	Phase indices
q	Chain index
M	Number of stations
K	Number of classes
N	Total population of jobs
$\mathcal{D}$	Set of delay stations
$\mathcal{Q}$	Set of DPS stations
${R_{i,k}}^*$	Number of phases composing the service time distribution of a class-k job at station i
$\mu_{i,k,r}$	Service rate of a class-k job in phase r at station i
$\alpha_{i,k,r}$	Probability that a class-k job at station i starts in phase r
$P_{i,k,(r,s)}$	Probability that a class-k job at station i shifts from phase r to phase s
$\beta_{i,k,r}$	Probability that a class-k job at station i ends in phase r
${w_{i,k}}^*$	Service weight of a class-k job at station i
$P_{(i,k),(j,l)}$	Probability that a class-k job at station i switches to class l and moves to station j
$P_{(i,k,r),(j,l,s)}$	Probability that a class-k job in phase r at station i proceeds to station j as a class-l job in phase s
$n_{i,k,r}$	Number of class-k jobs in phase r at station i
$x_{i,k,r}$	Amount of class-k fluid in phase r at station i
${N_k}^\dagger$	Population of class-k jobs
$S_{i,k}$	Mean service time of a class-k job at station i

$D_{i,k}$	Mean service demand of a class-k job at station i
$\tilde{Q}_{i,k}$	Mean queue length of class-k jobs at station i
$\tilde{Rp}_{i,k}$	Mean response time of a class-k job at station i
$\tilde{Rd}_{i,k}$	Mean residence time of a class-k job at station i
$\Phi_{i,k}$	Response time CDF of a class-k job at station i
N_q	Population of chain-q jobs
$\tilde{Q}_{q\text{-}i}$	Mean queue length of chain-q jobs with respect to station i
$\tilde{X}_{q\text{-}i}$	Mean throughput of chain-q jobs with respect to station i
$\tilde{Rp}_{q\text{-}i}$	Mean response time of a chain-q job with respect to station i
$\Phi_{q\text{-}i}$	Response time CDF of a chain-q job with respect to station i

* $R_{i,k}$ and $w_{i,k}$ may be ambiguous for a station–class pair (i,k). In these cases, our convention is to let $R_{i,k}=0$ and $w_{i,k}=0$ respectively.

† N_k becomes deterministic only when there is no class switching, i.e. $P_{(i,k),(j,l)}=0$ for all classes $k,l\in[1\,..\,K]$ such that $k\neq l$.

5.2 Related Work

DPS was initially suggested by Kleinrock to model time-sharing computer systems with priority groups [Kle67]. Under the DPS discipline, each class-k job in the system is assigned a non-negative weight w_k, and the service capacity is divided among all the jobs in proportion to their assigned weights. If there are $n_l(t)$ class-l jobs in the system at time t, each class-k job receives an instantaneous fraction $w_k/\sum_l w_l n_l(t)$ of the service capacity. One can therefore effectively control the completion rates of different classes of jobs by adjusting the class-dependent weights. EPS is actually a special case of DPS where every job is assigned an equal weight.

After the work of Kleinrock, the research community mainly focused on single-station DPS models with Poisson arrivals. Fayolle et al. [FMI80] obtained the conditional mean response times as the solution to a system of integro-differential equations. When the service times follow exponential distributions, this yields a rather simple formula, and the unconditional mean response times satisfy a linear equation system. Rege and Sengupta [RS94] proved a decomposition theorem, which says that the response time distribution of a job seeing n jobs in the system upon arrival can be decomposed into $n+1$ independent components given by

a system of non-linear integral equations. For exponentially distributed service times, Rege and Sengupta again [RS96] found the moments of the joint queue length distribution as the solutions to linear simultaneous equations and proved a heavy traffic limit theorem for the joint queue length distribution. These results were further extended to the case of phase-type service time distributions by van Kessel et al. [vKNQB04]. Avrachenkov et al. [AABNQ05] showed that the unconditional mean response times are finite as long as the system is stable. They also established a conservation law for the mean total residual service time and obtained the asymptotes of the conditional mean response times in closed form, assuming that the service time distributions have finite second-order moments. Borst et al. [BvOZ05] explored the asymptotic bounds of the response time distributions for heavy-tailed service time distributions. They showed that the service time and response time distributions for a class exhibit similar tail behavior if the service times of both specific and arbitrary jobs are regularly varying of index $v > 2$. Notably in the case of exponentially distributed service times, Kim and Kim [KK04] obtained a system of partial differential equations for the hybrid transforms of the response time distribution for each class and the joint queue length distribution. They found by applying this equation system that the moments of the response time distributions satisfy linear simultaneous equations. Based on Kim and Kim's work, a moment-generating algorithm for iteratively computing the higher-order moments of the response time distributions was devised by Chis and Harrison [CH16]. Izagirre et al. [IAV16] meanwhile derived closed-form approximations to the conditional and unconditional mean response times, which are applicable for general service time distributions with finite second-order moments. The approximation becomes exact in certain scenarios and is uniformly bounded in the second-order moments of the service time distributions.

Publications on closed QNs with DPS are instead limited in number. Lazowska et al. [LZGS84] brought in an equation for computing the mean residence time of a job at a DPS station. This extends the *mean value analysis* (MVA) algorithm [RL80] with support for the DPS discipline. Mitra and Weiss [MW89] studied a model comprising a delay and a DPS station where the think and service times are exponentially distributed, and obtained a system of ordinary differential equations for the transient mean queue lengths through asymptotic analysis. Under the

heavy traffic assumption, they gave explicit expressions for the steady-state values and a simple procedure for determining time constants that govern the convergence rate to the steady state. Morrison [Mor91] reviewed the same model while assuming moderately heavy traffic. He found two-term asymptotic approximations to the mean queue length and the mean response time for each class as well as the leading term in the asymptotic approximation to the joint queue length distribution, which is a zero-mean multivariate Gaussian distribution around a line through the origin. In contrast to the MVA algorithm for DPS, the fluid approach proposed in this chapter allows not only steady-state but also transient analysis of closed QNs with DPS. The latter in particular makes it possible to approximate response time distributions at either station or system level. Compared to results derived by Mitra et al., the proposed approach deals with a more general reference model that features an arbitrary routing topology, phase-type service time distributions and class switching.

5.3 Model Definition

We consider a closed QN that consists of M stations and K classes. There is a total population of N jobs circulating among the stations, each of which belongs to one of the classes at any point in time. The service time of a class-k job at station i follows a phase-type distribution composed of $R_{i,k}$ phases. Each phase r represents an exponentially distributed amount of service time with rate $\mu_{i,k,r}$ and is associated with probability $\alpha_{i,k,r}$ of being chosen as the initial one. On completion of its service in phase r, the job either shifts to phase s with probability $P_{i,k,(r,s)}$ or ends in phase r with probability $\beta_{i,k,r} = 1 - \sum_{s=1}^{R_{i,k}} P_{i,k,(r,s)}$. The type of a station can be either delay or DPS. We denote by $\mathcal{D}$ and $\mathcal{Q}$ the sets of delay and DPS stations, respectively. A delay station has an infinite number of servers, thus allocating a dedicated server to each incoming job. In our case, a DPS station refers to a single-server queueing station that implements the DPS discipline. Each class-k job at a DPS station i is assigned a non-negative weight $w_{i,k}$ and receives a fraction $w_{i,k} / \sum_{l=1}^{K} w_{i,l} \sum_{s=1}^{R_{i,l}} n_{i,l,s}(t)$ of the server at time t, where $n_{i,l,s}(t)$ is the instantaneous number of class-l jobs in phase s at the station. Once the service of a class-k job at station i finishes, it switches to class l and moves to station j with probability $P_{(i,k),(j,l)}$.

The routing probability matrix $\boldsymbol{P} = [P_{(i,k),(j,l)}] \in \mathbb{R}^{MK \times MK}$ is required to meet the assumption of a multi-chain closed QN (see Section 5.5.1). For convenience, we denote by $P_{(i,k,r),(j,l,s)}$ the probability of a class-k job completed in phase r at station i proceeding to station j as a class-l job in phase s:

$$P_{(i,k,r),(j,l,s)} = \begin{cases} P_{i,k,(r,s)} + \beta_{i,k,r} P_{(i,k),(i,k)} \alpha_{i,k,s} & \text{if } i = j \wedge k = l, \\ \beta_{i,k,r} P_{(i,k),(j,l)} \alpha_{j,l,s} & \text{otherwise.} \end{cases} \tag{5.1}$$

The former piece of (5.1) is due to the fact that when i, j and k, l index the same station and the same class respectively, the job may shift directly from phase r to phase s or alternatively end in phase r, loop back to the station and restart in phase s.

5.4 Fluid Approximation

Our reference model can be readily mapped onto a continuous-time Markov chain (CTMC) for which a steady-state probability vector uniquely exists. However, the state space of this CTMC is extremely large even for a medium-sized model instance. Fluid approximation treats the evolution of a discrete-state stochastic process in a continuous fashion, thus being an effective means to avoid the state space explosion [BHLM13]. Thanks to results obtained by Kurtz [Kur70], the expected sample path of a model with a Markovian description can be approximated by the solution to a system of ODEs, provided that it satisfies a number of relatively mild technical conditions.

Featuring an arbitrary routing topology, the considered model may be instantiated as a simple closed QN where a delay station interacts with a DPS station. As shown in [MW89], the ODE system for such a model instance is discontinuous when the DPS station is empty, and consequently does not admit a unique solution. This is a common problem in fluid approximation of QNs with DPS. To deal with it, we modify the original model by adding a self-looping job at each DPS station. Recall that a job is said to be self-looping if it always loops back to the same station upon service completion [CeSL89]. To apply this notion, we assign the self-looping job

at a DPS station i a service weight

$$w_{i,0} = g(\boldsymbol{w}_i), \tag{5.2}$$

where $g : \mathbb{R}^K \to \mathbb{R}$ is a function of the local service weight vector $\boldsymbol{w}_i = (w_{i,k}) \in \mathbb{R}^K$ with two properties:

$$g(\boldsymbol{w}_i) \geq 0 \quad \text{if } \forall k \in [1\,..\,K], w_{i,k} \geq 0, \tag{5.3}$$

$$g(\boldsymbol{w}_i) = a \quad \text{if } \forall k \in [1\,..\,K], w_{i,k} = a. \tag{5.4}$$

The first one (5.3) is necessary for a valid service weight, whilst the second one (5.4) is needed to accommodate the EPS discipline, which treats all the jobs inclusive of the extra self-looping job equally as if they have the same service weight at the station.

The state of the modified model at time t is represented by the instantaneous number of jobs in each phase, i.e. a vector $\boldsymbol{n}(t) = (n_{i,k,r}(t)) \in \{\boldsymbol{n} \in \mathbb{N}^R \mid \sum_{i=1}^{M} \sum_{k=1}^{K} \sum_{r=1}^{R_{i,k}} n_{i,k,r} = N\}$, where $R = \sum_{i=1}^{M} \sum_{k=1}^{K} R_{i,k}$ is the total number of phases. Once the service of a class-k job in phase r at station i finishes, it may proceed to station j as a class-l job in phase s. This means that the model jumps from state $\boldsymbol{n}$ to state $\boldsymbol{n} - \boldsymbol{e}_{i,k,r} + \boldsymbol{e}_{j,l,s}$, where $\boldsymbol{e}_{i,k,r}$ is a vector of all zeros except for a one in the (i,k,r)th entry, and $\boldsymbol{e}_{j,l,s}$ has a similar meaning. The set of possible state changes in the model is thus $L = \{-\boldsymbol{e}_{i,k,r} + \boldsymbol{e}_{j,l,s} \mid i,j \in [1\,..\,M], k,l \in [1\,..\,K], r \in [1\,..\,R_{i,k}], s \in [1\,..\,R_{j,l}]\}$. The transition rate of the model from state $\boldsymbol{n}$ to state $\boldsymbol{n} + \boldsymbol{l}$ can be written as

$$q_{\boldsymbol{n},\boldsymbol{n}+\boldsymbol{l}} = f(\boldsymbol{n}, \boldsymbol{l}), \tag{5.5}$$

where $f : \mathbb{R}^R \times \mathbb{Z}^R \to \mathbb{R}$ is a function defined by

$$f(\boldsymbol{x}, \boldsymbol{l}) = \begin{cases} \mu_{i,k,r} P_{(i,k,r),(j,l,s)} x_{i,k,r} & \text{if } \boldsymbol{l} \in L \wedge i \in \mathcal{D}, \\ \mu_{i,k,r} P_{(i,k,r),(j,l,s)} \dfrac{w_{i,k} x_{i,k,r}}{\sum_{k'=1}^{K} w_{i,k'} \sum_{r'=1}^{R_{i,k'}} x_{i,k',r'} + g(\boldsymbol{w}_i)} & \text{if } \boldsymbol{l} \in L \wedge i \in \mathcal{Q}, \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

Given a state change $\boldsymbol{l} \in L$, the function $f(\boldsymbol{x}, \boldsymbol{l})$ is either linear or rational in the state argument $\boldsymbol{x}$ with a domain $\{\boldsymbol{x} \in \mathbb{R}^R\}$ or $\{\boldsymbol{x} \in \mathbb{R}^R \mid \sum_{k'=1}^{K} w_{i,k'} \sum_{r'=1}^{R_{i,k'}} x_{i,k',r'} + g(\boldsymbol{w}_i) \neq 0\}$, depending on whether i indexes a delay or a DPS station. Such functions are known to exhibit continuity,

boundedness and Lipschitz continuity on any non-empty compact and convex subset of their domains. As for a state change $\boldsymbol{l} \notin L$, the function $f(\boldsymbol{x}, \boldsymbol{l})$ is constantly zero-valued for all states $\boldsymbol{x} \in \mathbb{R}^R$. It is obvious that the above properties hold on the entire domain $\{\boldsymbol{x} \in \mathbb{R}^R\}$ in this case. Therefore, $f(\boldsymbol{x}, \boldsymbol{l})$ can be thought of as a function of the state argument $\boldsymbol{x}$ that is continuous, bounded and Lipschitz continuous on any non-empty compact and convex subset of its global domain $D = \{\boldsymbol{x} \in \mathbb{R}^R \mid \forall i \in \mathcal{Q}, \sum_{k'=1}^{K} w_{i,k'} \sum_{r'=1}^{R_{i,k'}} x_{i,k',r'} + g(\boldsymbol{w}_i) \neq 0\}$.

We now present formal results that underlie the fluid approach. The proof of these results essentially follows Kurtz's theorem and either explicitly or implicitly relies on the properties of the function $f(\boldsymbol{x}, \boldsymbol{l})$.

Lemma 5.1. *Let $\{\boldsymbol{n}^{[v]}(t) \mid v \in \mathbb{N}^*\}$ be the sample paths of an infinite sequence of closed QNs whose vth element is the modified model scaled with a total population of vN jobs as well as v times service rates and v self-looping jobs at each DPS station i. The sequence $\{\boldsymbol{n}^{[v]}(t) \mid v \in \mathbb{N}^*\}$ constitutes a density-dependent family of CTMCs.*

Proof. The elements of the sequence $\{\boldsymbol{n}^{[v]}(t) \mid v \in \mathbb{N}_{>0}\}$ represent closed QNs with a multiple scale of the modified model. According to the scaling scheme, the set of possible state changes in each of these models remains invariant, i.e. $L^{[v]} = L = \{-\boldsymbol{e}_{i,k,r} + \boldsymbol{e}_{j,l,s} \mid i, j \in [1\,..\,M], k, l \in [1\,..\,K], r \in [1\,..\,R_{i,k}], s \in [1\,..\,R_{j,l}]\}$, and the transition rate of the vth model from state $\boldsymbol{n}$ to state $\boldsymbol{n} + \boldsymbol{l}$ is given by

$$
\begin{aligned}
q^{[v]}_{\boldsymbol{n},\boldsymbol{n}+\boldsymbol{l}} &= \begin{cases}
\mu_{i,k,r} P_{(i,k,r),(j,l,s)} n_{i,k,r} & \text{if } \boldsymbol{l} \in L^{[v]} \wedge i \in \mathcal{D}, \\
v\mu_{i,k,r} P_{(i,k,r),(j,l,s)} \dfrac{w_{i,k} n_{i,k,r}}{\sum_{k'=1}^{K} w_{i,k'} \sum_{r'=1}^{R_{i,k'}} n_{i,k',r'} + v g(\boldsymbol{w}_i)} & \text{if } \boldsymbol{l} \in L^{[v]} \wedge i \in \mathcal{Q}, \\
0 & \text{otherwise;}
\end{cases} \\
&= \begin{cases}
v\mu_{i,k,r} P_{(i,k,r),(j,l,s)} \dfrac{n_{i,k,r}}{v} & \text{if } \boldsymbol{l} \in L \wedge i \in \mathcal{D}, \\
v\mu_{i,k,r} P_{(i,k,r),(j,l,s)} \dfrac{w_{i,k} \frac{n_{i,k,r}}{v}}{\sum_{k'=1}^{K} w_{i,k'} \sum_{r'=1}^{R_{i,k'}} \frac{n_{i,k',r'}}{v} + g(\boldsymbol{w}_i)} & \text{if } \boldsymbol{l} \in L \wedge i \in \mathcal{Q}, \\
0 & \text{otherwise.}
\end{cases}
\end{aligned}
\tag{5.7}
$$

Comparing (5.7) with (5.6) yields

$$q^{[v]}_{\boldsymbol{n},\boldsymbol{n}+\boldsymbol{l}} = vf\left(\frac{\boldsymbol{n}}{v}, \boldsymbol{l}\right). \tag{5.8}$$

(5.8) implies that for any state change $\boldsymbol{l}$, the transition rate of the vth model in state $\boldsymbol{n}$ can be expressed as v times that of the first model in state $\boldsymbol{n}/v$. The sequence $\{\boldsymbol{n}^{[v]}(t) \mid v \in \mathbb{N}_{>0}\}$ therefore constitutes a density-dependent family of CTMCs [Kur70]. □

Lemma 5.2. *Define a vector field* $\boldsymbol{F} : \mathbb{R}^R \to \mathbb{R}^R$ *by*

$$\boldsymbol{F}(\boldsymbol{x}) = \sum_{\boldsymbol{l}} \boldsymbol{l} f(\boldsymbol{x}, \boldsymbol{l}). \tag{5.9}$$

Specifically,

$$F_{i,k,r}(\boldsymbol{x}) = \sum_{j=1}^{M}\sum_{l=1}^{K}\sum_{s=1}^{R_{j,l}}(f(\boldsymbol{x}, -\boldsymbol{e}_{j,l,s} + \boldsymbol{e}_{i,k,r}) - f(\boldsymbol{x}, -\boldsymbol{e}_{i,k,r} + \boldsymbol{e}_{j,l,s})). \tag{5.10}$$

The initial value problem (IVP)

$$\begin{cases} \dfrac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{F}(\boldsymbol{x}(t)), \\ \boldsymbol{x}(0) = \boldsymbol{n}(0) \end{cases} \tag{5.11}$$

has a unique solution $\boldsymbol{x}(t)$ *in any finite time horizon* T.

Proof. The modified model is a closed QN with a total population of N jobs across all the phases. The entries of its sample path $\boldsymbol{n}(t)$ must be non-negative and sum up to N:

$$\begin{cases} n_{i,k,r}(t) \geq 0, \\ \sum\limits_{i=1}^{M}\sum\limits_{k=1}^{K}\sum\limits_{r=1}^{R_{i,k}} n_{i,k,r}(t) = N. \end{cases} \tag{5.12}$$

This is also true for each element of the normalized sequence $\{\boldsymbol{n}^{[v]}(t)/v \mid v \in \mathbb{N}_{>0}\}$ and every candidate solution $\boldsymbol{x}(t)$ to the IVP (5.11):

$$\begin{cases} \dfrac{n^{[v]}_{i,k,r}(t)}{v} \geq 0, \\ \sum\limits_{i=1}^{M}\sum\limits_{k=1}^{K}\sum\limits_{r=1}^{R_{i,k}} \dfrac{n^{[v]}_{i,k,r}(t)}{v} = N; \end{cases} \tag{5.13}$$

$$\begin{cases} x_{i,k,r}(t) \geq 0, \\ \sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}} x_{i,k,r}(t) = N. \end{cases} \tag{5.14}$$

The former piece of (5.14) holds because

$$x_{i,k,r}(0) = n_{i,k,r}(0) \geq 0 \tag{5.15}$$

and

$$\begin{aligned} &\left.\frac{dx_{i,k,r}(t)}{dt}\right|_{x_{i,k,r}(t)=0} = F_{i,k,r}(\boldsymbol{x}(t))|_{x_{i,k,r}(t)=0} \\ &= \sum_{j=1}^{M}\sum_{l=1}^{K}\sum_{s=1}^{R_{j,l}}(f(\boldsymbol{x}(t), -\boldsymbol{e}_{j,l,s} + \boldsymbol{e}_{i,k,r}) - f(\boldsymbol{x}(t), -\boldsymbol{e}_{i,k,r} + \boldsymbol{e}_{j,l,s}))\Big|_{x_{i,k,r}(t)=0} \\ &= \sum_{j=1}^{M}\sum_{l=1}^{K}\sum_{s=1}^{R_{j,l}} f(\boldsymbol{x}(t), -\boldsymbol{e}_{j,l,s} + \boldsymbol{e}_{i,k,r})\Big|_{x_{i,k,r}(t)=0} \geq 0. \end{aligned} \tag{5.16}$$

(5.15) and (5.16) result from (5.11), (5.12), (5.10) and (5.6). The latter piece of (5.14) is due to the fact that

$$\sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}} x_{i,k,r}(0) = \sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}} n_{i,k,r}(0) = N \tag{5.17}$$

and that

$$\begin{aligned} &\frac{d\sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}} x_{i,k,r}(t)}{dt} = \sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}} \frac{dx_{i,k,r}(t)}{dt} = \sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}} F_{i,k,r}(\boldsymbol{x}(t)) \\ &= \sum_{i,j=1}^{M}\sum_{k,l=1}^{K}\sum_{r=1}^{R_{i,k}}\sum_{s=1}^{R_{j,l}}(f(\boldsymbol{x}(t), -\boldsymbol{e}_{j,l,s} + \boldsymbol{e}_{i,k,r}) - f(\boldsymbol{x}(t), -\boldsymbol{e}_{i,k,r} + \boldsymbol{e}_{j,l,s})) = 0. \end{aligned} \tag{5.18}$$

(5.17) and (5.18) derive from (5.11), (5.12) and (5.10).

Consider an open set $E = \{\boldsymbol{x} \in \mathbb{R}^{R}_{>-\epsilon} \mid |\sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}} x_{i,k,r} - N| < 1\}$, where $0 < \epsilon < \min_{i\in\mathcal{Q}} g(\boldsymbol{w}_i)/\sum_{k'=1}^{K} w_{i,k'}R_{i,k'}$. We know from (5.14) that

$$\boldsymbol{x}(t) \in E \quad \text{for } t \in [0, T]. \tag{5.19}$$

The closure of the set E is a non-empty compact and convex subset of the set D, which is the global domain of the function $f(\boldsymbol{x}, \boldsymbol{l})$. As in (5.10), each entry of the vector field $\boldsymbol{F}(\boldsymbol{x})$ is a linear combination of the function $f(\boldsymbol{x}, \boldsymbol{l})$ at relevant state changes $\boldsymbol{l} \in L$. The vector field

$\boldsymbol{F}(\boldsymbol{x})$ thus retains the continuity, boundedness and Lipschitz continuity of the function $f(\boldsymbol{x}, \boldsymbol{l})$ on the set E:

$$\lim_{\boldsymbol{x}' \to \boldsymbol{x}} \boldsymbol{F}(\boldsymbol{x}') = \boldsymbol{F}(\boldsymbol{x}) \quad \text{for } \boldsymbol{x} \in E; \tag{5.20}$$

$$\|\boldsymbol{F}(\boldsymbol{x})\| \leq M_E \quad \text{for } \boldsymbol{x} \in E; \tag{5.21}$$

$$\|\boldsymbol{F}(\boldsymbol{x}') - \boldsymbol{F}(\boldsymbol{x})\| \leq K_E \|\boldsymbol{x}' - \boldsymbol{x}\| \quad \text{for } \boldsymbol{x}', \boldsymbol{x} \in E. \tag{5.22}$$

Norms on a finite-dimensional vector space are necessarily equivalent. There exist two constants $c_1, c_2 > 0$ such that for any vector $\boldsymbol{a} \in \mathbb{R}^R$,

$$c_1 \|\boldsymbol{a}\|_1 \leq \|\boldsymbol{a}\| \leq c_2 \|\boldsymbol{a}\|_1. \tag{5.23}$$

Let $\boldsymbol{x}_0$ be an arbitrary state satisfying (5.14). From (5.23), (5.20), (5.21) and (5.22), we can find a closed neighborhood of the state $\boldsymbol{x}_0$, $B_0 = \{\boldsymbol{x} \in \mathbb{R}^R \mid \|\boldsymbol{x} - \boldsymbol{x}_0\| \leq c_1\epsilon/2\} \subset \{\boldsymbol{x} \in \mathbb{R}^R \mid \|\boldsymbol{x} - \boldsymbol{x}_0\|_1 \leq \epsilon/2\} \subset E$, on which the vector field $\boldsymbol{F}(\boldsymbol{x})$ is continuous and has a bound $M_{B_0} \leq M_E$ as well as a Lipschitz constant $K_{B_0} \leq K_E$. It then follows by the Picard–Lindelöf theorem (Theorem 8.13 in [KP10]) that the ODE system in (5.11) is uniquely solvable under the initial condition $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ on the time interval $[t_0, t_0 + \Delta]$, where $\Delta = c_1\epsilon/2M_E \leq c_1\epsilon/2M_{B_0}$. This result is applicable to a finite sequence of intersecting time intervals $\{[\tau\Delta, (\tau+1)\Delta] \mid \tau \in \mathbb{N}_{<T/\Delta}\}$. The IVP (5.11) therefore has a unique solution $\boldsymbol{x}(t)$ in any finite time horizon T. □

Theorem 5.1. *Given the same initial state*

$$\frac{\boldsymbol{n}^{[v]}(0)}{v} = \boldsymbol{n}(0), \tag{5.24}$$

the normalized sequence $\{\boldsymbol{n}^{[v]}(t)/v \mid v \in \mathbb{N}_{>0}\}$ *converges to the solution* $\boldsymbol{x}(t)$ *to the IVP (5.11) in the sense that for any small number* $\delta > 0$,

$$\lim_{v \to \infty} \mathbb{P}\left(\sup_{t \in [0,T]} \left\| \frac{\boldsymbol{n}^{[v]}(t)}{v} - \boldsymbol{x}(t) \right\| > \delta \right) = 0. \tag{5.25}$$

Proof. Applying the triangle inequality and (5.23) gives

$$\|-\boldsymbol{e}_{i,k,r} + \boldsymbol{e}_{j,l,s}\| \leq \|-\boldsymbol{e}_{i,k,r}\| + \|\boldsymbol{e}_{j,l,s}\| \leq c_2\|-\boldsymbol{e}_{i,k,r}\|_1 + c_2\|\boldsymbol{e}_{j,l,s}\|_1 = 2c_2. \tag{5.26}$$

From (5.6), we can show that

$$\begin{aligned}\sup_{\boldsymbol{x}\in E}\sum_{\boldsymbol{l}}\|\boldsymbol{l}\|f(\boldsymbol{x},\boldsymbol{l}) &= \sup_{\boldsymbol{x}\in E}\sum_{i,j=1}^{M}\sum_{k,l=1}^{K}\sum_{r=1}^{R_{i,k}}\sum_{s=1}^{R_{j,l}}\|-\boldsymbol{e}_{i,k,r}+\boldsymbol{e}_{j,l,s}\|f(\boldsymbol{x},-\boldsymbol{e}_{i,k,r}+\boldsymbol{e}_{j,l,s})\\ &\le \sup_{\boldsymbol{x}\in E}\sum_{i,j=1}^{M}\sum_{k,l=1}^{K}\sum_{r=1}^{R_{i,k}}\sum_{s=1}^{R_{j,l}}2c_2 f(\boldsymbol{x},-\boldsymbol{e}_{i,k,r}+\boldsymbol{e}_{j,l,s})<\infty.\end{aligned} \tag{5.27}$$

where the first inequality results from (5.26), and the second one is due to the boundedness of the function $f(\boldsymbol{x},\boldsymbol{l})$ on set E. Applying (5.6) and (5.26), we also have

$$\lim_{d\to\infty}\sup_{\boldsymbol{x}\in E}\sum_{\|\boldsymbol{l}\|>d}\|\boldsymbol{l}\|f(\boldsymbol{x},\boldsymbol{l}) = \lim_{d\to\infty}\sup_{\boldsymbol{x}\in E}\sum_{\|-\boldsymbol{e}_{i,k,r}+\boldsymbol{e}_{j,l,s}\|>d}\|-\boldsymbol{e}_{i,k,r}+\boldsymbol{e}_{j,l,s}\|f(\boldsymbol{x},-\boldsymbol{e}_{i,k,r}+\boldsymbol{e}_{j,l,s})=0. \tag{5.28}$$

By Kurtz's theorem (Theorem 3.1 in [Kur70]) together with Lemmas 5.1 and 5.2, it follows from (5.22), (5.27), (5.28), (5.19) and (5.24) that the normalized sequence $\{\boldsymbol{n}^{[v]}(t)/v \mid v\in\mathbb{N}_{>0}\}$ converges to the solution $\boldsymbol{x}(t)$ to the IVP (5.11) in the sense of (5.25). □

Corollary 5.1. *Suppose that the condition of Theorem 5.1 is satisfied. The expectation of the normalized sequence* $\{\boldsymbol{n}^{[v]}(t)/v \mid v\in\mathbb{N}_{>0}\}$ *converges to the solution* $\boldsymbol{x}(t)$ *to the IVP (5.11) in any finite time horizon* T*:*

$$\lim_{v\to\infty}\mathbb{E}\left(\frac{\boldsymbol{n}^{[v]}(t)}{v}\right)=\boldsymbol{x}(t)\quad\text{for } t\in[0,T]. \tag{5.29}$$

Proof. Observe that

$$0\le\mathbb{P}\left(\left\|\frac{\boldsymbol{n}^{[v]}(t)}{v}-\boldsymbol{x}(t)\right\|>\delta\right)\le\mathbb{P}\left(\sup_{t'\in[0,T]}\left\|\frac{\boldsymbol{n}^{[v]}(t')}{v}-\boldsymbol{x}(t')\right\|>\delta\right)\quad\text{for } t\in[0,T]. \tag{5.30}$$

By the squeeze theorem, we obtain from (5.30) and (5.25)

$$\lim_{v\to\infty}\mathbb{P}\left(\left\|\frac{\boldsymbol{n}^{[v]}(t)}{v}-\boldsymbol{x}(t)\right\|>\delta\right)=0\quad\text{for } t\in[0,T]. \tag{5.31}$$

Applying the triangle inequality, (5.23), (5.13) and (5.14) leads to

$$\begin{aligned}\left\|\frac{\boldsymbol{n}^{[v]}(t)}{v}-\boldsymbol{x}(t)\right\| &\le \left\|\frac{\boldsymbol{n}^{[v]}(t)}{v}\right\|+\|\boldsymbol{x}(t)\| \le c_2\left\|\frac{\boldsymbol{n}^{[v]}(t)}{v}\right\|_1+c_2\|\boldsymbol{x}(t)\|_1\\ &= c_2\sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}}\frac{n^{[v]}_{i,k,r}(t)}{v}+c_2\sum_{i=1}^{M}\sum_{k=1}^{K}\sum_{r=1}^{R_{i,k}}x_{i,k,r}(t)=2c_2N.\end{aligned} \tag{5.32}$$

By a corollary to the dominated convergence theorem (Corollary 1 to Theorem 1.C in [Spa05]),

it follows from (5.31) and (5.32) that

$$\lim_{v\to\infty} \mathbb{E}\left(\frac{\boldsymbol{n}^{[v]}(t)}{v}\right) = \mathbb{E}(\boldsymbol{x}(t)) = \boldsymbol{x}(t) \quad \text{for } t \in [0, T]. \tag{5.33}$$

The expectation of the normalized sequence $\{\boldsymbol{n}^{[v]}(t)/v \mid v \in \mathbb{N}_{>0}\}$ therefore converges to the solution $\boldsymbol{x}(t)$ to the IVP (5.11) in any finite time horizon T. □

Corollary 5.1 implies that as the scaling factor v increments, the expected normalized sample path $\mathbb{E}(\boldsymbol{n}^{[v]}(t)/v)$ of the vth model gradually approaches the solution $\boldsymbol{x}(t)$ to the fluid ODE system. As stated in Lemma 5.1, we scale the total population of jobs as well as service rates and self-looping jobs at each DPS station. Since this scaling scheme preserves the completion rate per job for all the classes and all the stations, it is reasonable to approximate the expected sample path $\mathbb{E}(\boldsymbol{n}(t))$ of the first model by the solution $\boldsymbol{x}(t)$ to the fluid ODE system:

$$\mathbb{E}(\boldsymbol{n}(t)) \approx \boldsymbol{x}(t) \quad \text{for } t \in [0, T]. \tag{5.34}$$

The derivation of (5.34) relies on the presence of self-looping jobs, which are however not included in the definition of our reference model. Replacing the sample path $\boldsymbol{n}(t)$ with the one of the original model would potentially require every DPS station i to be under a heavy load such that

$$\sum_{k=1}^{K} w_{i,k} \sum_{r=1}^{R_{i,k}} \mathbb{E}(n_{i,k,r}(t)) \gg g(\boldsymbol{w}_i) \quad \text{for } t \in [0, T]. \tag{5.35}$$

Any function $g(\boldsymbol{w}_i)$ in compliance with (5.3) and (5.4) gives an optional form for the service weight $w_{i,0}$ of the extra self-looping job at station i. We recommend adopting the arithmetic mean over the entries of the local service weight vector $\boldsymbol{w}_i$:

$$w_{i,0} = g(\boldsymbol{w}_i) = \frac{\sum_{k=1}^{K} w_{i,k}}{K}, \tag{5.36}$$

so that the self-looping job acts as an average job across all the classes at the station. Validation results reported in Section 5.6 are based on this intuitive choice. Other forms, such as $\min_{k\in[1..K]} w_{i,k}$ and $\max_{k\in[1..K]} w_{i,k}$, are generally feasible but observed to be less accurate for typical examples used in the validation experiments.

From (5.34), we know that the solution $\boldsymbol{x}(t)$ to the fluid ODE system forms an approximation to the expected sample path $\mathbb{E}(\boldsymbol{n}(t))$ of our reference model on any finite time interval $[0, T]$. The transient mean queue length of class-k jobs at station i can thus be written as

$$Q_{i,k}(t) = \sum_{r=1}^{R_{i,k}} \mathbb{E}(n_{i,k,r}(t)) \approx \sum_{r=1}^{R_{i,k}} x_{i,k,r}(t) \quad \text{for } t \in [0, T]. \tag{5.37}$$

To obtain the steady-state value, (5.34) needs to be extended at infinite time $t \to \infty$:

$$\lim_{t\to\infty} \mathbb{E}(\boldsymbol{n}(t)) \approx \lim_{t\to\infty} \boldsymbol{x}(t) = \tilde{\boldsymbol{x}}, \tag{5.38}$$

where $\tilde{\boldsymbol{x}}$ is the steady-state solution to the fluid ODE system, also known as the attractor [GVH17]. Applying (5.38) yields

$$\tilde{Q}_{i,k} = \sum_{r=1}^{R_{i,k}} \lim_{t\to\infty} \mathbb{E}(n_{i,k,r}(t)) \approx \sum_{r=1}^{R_{i,k}} \tilde{x}_{i,k,r}. \tag{5.39}$$

The proof of (5.38) requires showing that the fluid ODE system admits a unique steady-state solution $\tilde{\boldsymbol{x}}$:

$$\lim_{t\to\infty} \boldsymbol{x}(t) = \tilde{\boldsymbol{x}} \tag{5.40}$$

and that the expected normalized sample path $\mathbb{E}(\boldsymbol{n}^{[v]}(t)/v)$ converges to the solution $\boldsymbol{x}(t)$ to the fluid ODE system at infinite time $t \to \infty$:

$$\lim_{t\to\infty} \lim_{v\to\infty} \mathbb{E}\left(\frac{\boldsymbol{n}^{[v]}(t)}{v}\right) = \lim_{t\to\infty} \boldsymbol{x}(t), \tag{5.41}$$

which are difficult. In practice, we find the steady-state solution $\tilde{\boldsymbol{x}}$ by evolving the fluid ODE system until time t_∞ where the norm $\|d\boldsymbol{x}(t)/dt\|$ of the derivative nearly vanishes. Although this is not completely rigorous from a theoretical point of view, no failed cases have been encountered in the large set of model instances created for validation.

5.5 Application

The fluid approach features the ability to analyze the transient behavior of closed QNs with DPS. This could be useful, for example, to capture the dynamics of a random environment

between successive stage changes [CTH14]. Transient analysis also enables approximation of response time distributions at a station where the service of an incoming job starts immediately after arrival [PC17]. In what follows, we introduce a refined method and its extension for approximating response time distributions at individual stations and for the entire system, building on the concept of chains [Mun73].

5.5.1 Concept of Chains

The routing probability matrix $\boldsymbol{P} = [P_{(i,k),(j,l)}] \in \mathbb{R}^{MK \times MK}$ can be regarded as defining a discrete-time Markov chain (DTMC) whose states are station–class pairs of form (i, k). Assuming the routing DTMC to be decomposable into ergodic subchains, Muntz [Mun73] showed in the product-form case that a closed QN with multiple chains is equivalent to one with the same number of classes. The states of an ergodic DTMC are reachable from one other, recurrent non-null and aperiodic. If we represent the routing DTMC as a weighted directed graph, then each chain manifests itself as a strongly connected subgraph. Since all the chains are completely isolated, it is impossible for a job to switch between different chains. Let $\mathcal{V}_q$ and $\mathcal{E}_q$ be the sets of station–class pairs and connections in chain q, respectively. The population of chain-q jobs is thus

$$N_q = \sum_{(i,k) \in \mathcal{V}_q} \sum_{r=1}^{R_{i,k}} n_{i,k,r} \tag{5.42}$$

For convenience, we denote by $\mathcal{C}_{i,q}$ the set of classes that may be visited by a chain-q job at station i, and by $\mathcal{C}_q$ the set of all classes that may be visited by a chain-q job, i.e. $\mathcal{C}_{i,q} = \{k \mid (i,k) \in \mathcal{V}_q\}$ and $\mathcal{C}_q = \bigcup_{i=1}^{M} \mathcal{C}_{i,q}$. The concept of chains is essential not only in that our reference model is defined as a multi-chain closed QN but also in that it underlies the method for approximating response time distributions.

5.5.2 Response Time Distributions

To approximate the response time distribution for class k at station i, the method present in [PC17] tags all class-k jobs at that station and restore them upon service completion. Conse-

quently, the duration that a class-k job is being tagged at station i corresponds to its residual sojourn time, which differs from its response time unless the service time of the job is memoryless, i.e. exponentially distributed. Our method overcomes this limitation by adding a start phase in the service time distribution to tag class-k jobs that have just arrived at the station.

First, we evaluate the mean service time of a class-k job at station i by applying the moment formula for phase-type distributions:

$$S_{i,k} = \boldsymbol{\alpha}_{i,k}(-\boldsymbol{T}_{i,k})^{-1}\mathbf{1}_{i,k}, \tag{5.43}$$

where $\boldsymbol{\alpha}_{i,k} = (\alpha_{i,k,r}) \in \mathbb{R}^{1\times R_{i,k}}$ and $\boldsymbol{T}_{i,k} = [T_{i,k,(r,s)}] \in \mathbb{R}^{R_{i,k}\times R_{i,k}}$ are the initial probability vector and the transition rate matrix that represent the service time distribution for class k at station i, and $\mathbf{1}_{i,k} \in \mathbb{N}^{R_{i,k}\times 1}$ is a column vector of all ones. The (r,s)th entry of the transition rate matrix $\boldsymbol{T}_{i,k}$ is given by

$$T_{i,k,(r,s)} = \begin{cases} -\mu_{i,k,r} & \text{if } r = s, \\ \mu_{i,k,r}P_{i,k,(r,s)} & \text{otherwise.} \end{cases} \tag{5.44}$$

Next, we modify the service time distribution for class k at station i so that any class-k arrival at the station starts in an additional phase 0 with probability $\alpha_{i,k,0} = 1$ and shifts from that phase to phase r with probability $P_{i,k,(0,r)} = \alpha_{i,k,r}$. Phase $(i,k,0)$ therefore gathers up all class-k jobs having just arrived at station i. To limit the side effects of the modification, the service rate of a job in phase $(i,k,0)$ is set as $\mu_{i,k,0} = 1/\eta S_{i,k}$, where η is a small positive coefficient.[50] After that, we build a fluid ODE system that incorporates the modified service time distribution, and compute its steady-state solution $\tilde{\boldsymbol{x}}' \in \mathbb{R}^{R'}$ under the initial condition $\boldsymbol{x}'(0) \in \mathbb{R}^{R'}$, where $R' = R + 1$. The initial condition $\boldsymbol{x}'(0)$ arises from the initial state $\boldsymbol{n}(0)$ by inserting zero entries. Then, we create an auxiliary class k' for tagging class-k jobs. Class k' has the same service time distribution and the same service weight as class k at station i. The probability of a class-k' job at station i being forwarded to station j as a class-l job is set to $P_{(i,k'),(j,l)} = P_{(i,k),(j,l)}$. A tagged class-$k$ job is thereby restored immediately after departure from station i. Finally, we build another fluid ODE system that incorporates the auxiliary class,[51]

[50] The positive coefficient η cannot be chosen to be arbitrarily small as this will end up with a stiff fluid ODE system and a large computational error.

[51] Auxiliary classes are in fact a tool for tagging jobs that satisfy certain conditions. The service weights for such classes should not be taken into account in that of the self-looping job at any DPS station.

and compute its transient solution $\boldsymbol{x}''(t) \in \mathbb{R}^{R''}$ under the initial condition $\boldsymbol{x}''(0) \in \mathbb{R}^{R''}$, where $R'' = R + R_{i,k'} + 2$. The initial condition $\boldsymbol{x}''(0)$ derives from the resultant steady-state solution $\tilde{\boldsymbol{x}}'$ by inserting zero entries except that $x''_{i,k,0}(0) = 0$ and $x''_{i,k',0}(0) = \tilde{x}'_{i,k,0}$.

Let $\tilde{rp}_{i,k}$ be the response time of a class-k job at station i. $\tilde{rp}_{i,k}$ is a random variable of which the cumulative distribution function (CDF) can be written as

$$\Phi_{i,k}(t) = \mathbb{P}(\tilde{rp}_{i,k} \leq t) = 1 - \mathbb{P}(\tilde{rp}_{i,k} > t). \tag{5.45}$$

Suppose that (i,k) is a station–class pair in chain q, i.e. $(i,k) \in \mathcal{V}_q$. Because any class-k job at station i belongs to chain q and because all chain-q jobs are indistinguishable from one another, the probability of the response time of a class-k job at station i being longer than time span t is equal to that of the nth job in chain q staying at station i as a class-k job during time interval $[0,t]$ after being observed in phase $(i,k,0)$ at time 0:

$$\mathbb{P}(\tilde{rp}_{i,k} > t) \approx \mathbb{P}(\forall u \in [0,t], y_{q,n}(u) \in \mathcal{P}_{i,k} \,|\, y_{q,n}(0) = (i,k,0)), \tag{5.46}$$

where $y_{q,n}(u)$ is the phase visited by the nth job in chain q at time u, and $\mathcal{P}_{i,k}$ is the set of phases in the service time distribution for class k at station i, i.e. $\mathcal{P}_{i,k} = \{(i,k,r) \mid r \in [0\,..\,R_{i,k}]\}$. Particularly, the approximate equality in (5.46) is due to the addition of phase $(i,k,0)$. The count of events that a chain-q job is in phase $(i,k,0)$ at time 0 and stays as a class-k job at station i during time interval $[0,t]$ is exactly the number of class-k' jobs at station i at time t. That is,

$$\sum_{n=1}^{N_q} \mathbb{1}_{\{\forall u \in [0,t], y_{q,n}(u) \in \mathcal{P}_{i,k} \,\wedge\, y_{q,n}(0) = (i,k,0)\}} = \sum_{r=0}^{R_{i,k'}} n''_{i,k',r}(t). \tag{5.47}$$

Taking the expectation on both sides of (5.47) and dividing the results by N_q leads to

$$\begin{aligned}
&\mathbb{P}(\forall u \in [0,t], y_{q,n}(u) \in \mathcal{P}_{i,k} \,\wedge\, y_{q,n}(0) = (i,k,0)) \\
&= \mathbb{E}(\mathbb{1}_{\{\forall u \in [0,t], y_{q,n}(u) \in \mathcal{P}_{i,k} \,\wedge\, y_{q,n}(0) = (i,k,0)\}}) \\
&= \frac{1}{N_q} \mathbb{E}\left(\sum_{n=1}^{N_q} \mathbb{1}_{\{\forall u \in [0,t], y_{q,n}(u) \in \mathcal{P}_{i,k} \,\wedge\, y_{q,n}(0) = (i,k,0)\}} \right) \\
&= \frac{\mathbb{E}\left(\sum\limits_{r=0}^{R_{i,k'}} n''_{i,k',r}(t) \right)}{N_q} = \frac{\sum\limits_{r=0}^{R_{i,k'}} \mathbb{E}(n''_{i,k',r}(t))}{N_q}.
\end{aligned} \tag{5.48}$$

Substituting t with 0 in (5.48) and noticing that all class-k' jobs at station i are in phase 0 at time 0, we get

$$\begin{aligned}
&\mathbb{P}(y_{q,n}(0) = (i,k,0)) \\
&= \mathbb{P}(y_{q,n}(0) \in \mathcal{P}_{i,k} \wedge y_{q,n}(0) = (i,k,0)) \\
&= \frac{\sum_{r=0}^{R_{i,k'}} \mathbb{E}(n''_{i,k',r}(0))}{N_q} = \frac{\mathbb{E}(n''_{i,k',0}(0))}{N_q}.
\end{aligned} \tag{5.49}$$

It follows from (5.45), (5.46), (5.48), (5.49) and (5.34) that

$$\begin{aligned}
\Phi_{i,k}(t) &= 1 - \mathbb{P}(\tilde{rp}_{i,k} > t) \\
&\approx 1 - \mathbb{P}(\forall u \in [0,t], y_{q,n}(u) \in \mathcal{P}_{i,k} \mid y_{q,n}(0) = (i,k,0)) \\
&= 1 - \frac{\mathbb{P}(\forall u \in [0,t], y_{q,n}(u) \in \mathcal{P}_{i,k} \wedge y_{q,n}(0) = (i,k,0))}{\mathbb{P}(y_{q,n}(0) = (i,k,0))} \\
&= 1 - \frac{\sum_{r=0}^{R_{i,k'}} \mathbb{E}(n''_{i,k',r}(t))}{\mathbb{E}(n''_{i,k',0}(0))} \approx 1 - \frac{\sum_{r=0}^{R_{i,k'}} x''_{i,k',r}(t)}{x''_{i,k',0}(0)}.
\end{aligned} \tag{5.50}$$

(5.50) provides an approximation to the response time CDF for class k at station i from the transient solution $\boldsymbol{x}''(t)$.

By adding a start phase, the introduced method is applicable for general phase-type service times. It is worth pointing out that representative service time distributions, including exponential, Erlang, hypoexponential and Coxian, often contain a start phase. In such cases, the additional phase can be replaced with that phase to slightly improve the accuracy of the method. We implicitly exploit this advantage in the validation experiments for response time distribution analysis (see Section 5.6.5).

5.5.3 System Response Time Distributions

System performance measures of a closed QN with class switching are evaluated per chain with respect to a reference point or station [RL80]. We only consider the second case, where the system response time of a job is defined as the time that it spends between departure from and next arrival at the reference station. Like the original one [PC17], our method is further

extensible to approximate the response time distribution for chain q with respect to a delay station D. The basic idea is to add an end phase in the service time distribution for each class $k \in \mathcal{C}_{D,q}$ at station D to tag chain-q jobs that are about to depart from the station and restore them upon next arrival.

First, we evaluate the mean response time of a chain-q job with respect to station D by applying Little's law to chain q at the system level:

$$\tilde{R}p_{q\text{-}D} = \frac{\tilde{Q}_{q\text{-}D}}{\tilde{X}_{q\text{-}D}}, \tag{5.51}$$

where $\tilde{Q}_{q\text{-}D}$ and $\tilde{X}_{q\text{-}D}$ are the mean queue length and the mean throughput of chain-q jobs with respect to station D. The former can be obtained from (5.39) as

$$\tilde{Q}_{q\text{-}D} = N_q - \sum_{k \in \mathcal{C}_{D,q}} \tilde{Q}_{D,k} \approx N_q - \sum_{k \in \mathcal{C}_{D,q}} \sum_{r=1}^{R_{D,k}} \tilde{x}_{D,k,r}. \tag{5.52}$$

Applying Little's law to each class $k \in \mathcal{C}_{D,q}$ at station D together with (5.39) and noticing that station D is of delay type yields the latter:

$$\tilde{X}_{q\text{-}D} = \sum_{k \in \mathcal{C}_{D,q}} \tilde{X}_{D,k} = \sum_{k \in \mathcal{C}_{D,q}} \frac{\tilde{Q}_{D,k}}{\tilde{R}p_{D,k}} \approx \sum_{k \in \mathcal{C}_{D,q}} \frac{\sum_{r=1}^{R_{D,k}} \tilde{x}_{D,k,r}}{S_{D,k}}. \tag{5.53}$$

Next, we modify the service time distribution for each class $k \in \mathcal{C}_{D,q}$ at station D so that any class-k departure from the station shifts from phase r to an additional phase $R_{D,k} + 1$ with probability $P_{D,k,(r,R_{D,k}+1)} = \beta_{D,k,r}$ and ends in that phase with probability $\beta_{D,k,R_{D,k}+1} = 1$. The service rate of a job in phase $(D, k, R_{D,k} + 1)$ is set as $\mu_{D,k,R_{D,k}+1} = 1/\eta\tilde{R}p_{q\text{-}D}$. Since η is a small positive coefficient,[50] phase $(D, k, R_{D,k} + 1)$ gathers up all class-q jobs about to depart from station D as a class-k job. After that, we build a fluid ODE system that incorporates the modified service time distributions, and compute its steady-state solution $\tilde{\boldsymbol{x}}' \in \mathbb{R}^{R'}$ under the initial condition $\boldsymbol{x}'(0) \in \mathbb{R}^{R'}$, where $R' = R + |\mathcal{C}_{D,q}|$. The initial condition $\boldsymbol{x}'(0)$ arises from the initial state $\boldsymbol{n}(0)$ by inserting zero entries. Then, we create $|\mathcal{C}_q|$ auxiliary classes for tagging chain-q jobs. Each auxiliary class k' has the same service time distribution and the same service weight as a class $k \in \mathcal{C}_q$ at every station i such that (i, k) is a station–class pair in chain q, i.e. $(i, k) \in \mathcal{V}_q$. For each connection $((i, k), (j, l)) \in \mathcal{E}_q$, the probability of a class-k' job

at station i being forwarded to station $j = D$ as a class-l job is set as $P_{(i,k'),(D,l)} = P_{(i,k),(D,l)}$, while that of it being forwarded to station $j \neq D$ as a class-l' job is set as $P_{(i,k'),(j,l')} = P_{(i,k),(j,l)}$. A tagged chain-$q$ job is thereby not restored until immediately before next arrival at station D. Finally, we build another fluid ODE system that incorporates the auxiliary classes,[51] and compute its transient solution $\boldsymbol{x}''(t) \in \mathbb{R}^{R''}$ under the initial condition $\boldsymbol{x}''(0) \in \mathbb{R}^{R''}$, where $R'' = R + \sum_{(i,k)\in\mathcal{V}_q} R_{i,k'} + 2|\mathcal{C}_{D,q}|$. The initial condition $\boldsymbol{x}''(0)$ derives from the resultant steady-state solution $\tilde{\boldsymbol{x}}'$ by inserting zero entries except that $x''_{D,k,R_{D,k}+1}(0) = 0$ and $x''_{D,k',R_{D,k'}+1}(0) = \tilde{x}'_{D,k,R_{D,k}+1}$ for each class $k \in \mathcal{C}_{D,q}$ and its corresponding auxiliary class k'.

Let $\tilde{rp}_{q\text{-}D}$ be the response time of chain q with respect to station D. Following a derivation similar to that of (5.50) leads to

$$\Phi_{q\text{-}D}(t) = \mathbb{P}(\tilde{rp}_{q\text{-}D} \leq t) \approx 1 - \frac{\sum\limits_{k\in\mathcal{C}_{D,q}} x''_{D,k',R_{D,k'}+1}(t) + \sum\limits_{\substack{(i,k)\in\mathcal{V}_q \\ i\neq D}} \sum\limits_{r=1}^{R_{i,k'}} x''_{i,k',r}(t)}{\sum\limits_{k\in\mathcal{C}_{D,q}} x''_{D,k',R_{D,k'}+1}(0)}. \tag{5.54}$$

Given the transient solution $\boldsymbol{x}''(t)$, we can approximate the response time CDF for chain q with respect to station D by applying (5.54). Although some service time distributions, such as exponential, Erlang and hypoexponential, already contain an end phase, adding a new one is often inevitable since the introduced method requires the mean service time in that phase to be small compared to the mean system response time.

5.6 Validation

This section reports the validation of the fluid approach against simulation for both transient and steady-state analysis of mean queue lengths. In the transient validation, we focus on typical examples, given that estimating the transient performance measures of a QN through simulation is time-consuming. As for the steady-state validation, the fluid approach is compared with the approximate mean value analysis (AMVA) algorithm for DPS [LZGS84] on a large set of model instances. We have also validated the refined method for approximating response time distributions against simulation at both station and system levels on typical examples, comparing it with the original method [PC17].

5.6.1 Model Parameter Settings

The validation experiments were based on a set of 4914 model instances, which were created by altering the topology ($P_{(i,k),(j,l)}$), service time distributions ($R_{i,k}$, $\alpha_{i,k,r}$, $P_{i,k,(r,s)}$, $\beta_{i,k,r}$), number of stations (M), number of classes (K), total population of jobs (N), service rates ($\mu_{i,k,r}$), service weights ($w_{i,k}$) and random seed ($\mu_{i,k,r}$, $w_{i,k}$). The topology of a model instance was either serial or parallel with a delay station, as illustrated in Figures 5.1 and 5.2. In the serial topology, stations inclusive of the delay station were connected one after another, whilst in the parallel topology, DPS stations stood side by side and interacted with the delay station. Because the steady-state mean queue lengths in our reference model appear insensitive to different types of service time distributions, we chose the service times to be exponentially distributed for all the model instances but considered typical examples with Coxian service time distributions when necessary. Each model instance consisted of 3, 5 or 9 stations and 2, 4 or 8 classes with a total population of 128, 256 or 512 jobs, which were divided equally among the classes and preloaded at the delay station. For simplicity, the probability of a departure from the delay station moving to any DPS station was set the same, and no jobs were allowed to switch between any two classes so that every class constituted a separate chain. Service rates for a model instance were either balanced or unbalanced. In the balanced case, we let the service rates at all the DPS stations be 5.0. As for the unbalanced case, the service rates at each DPS station were set

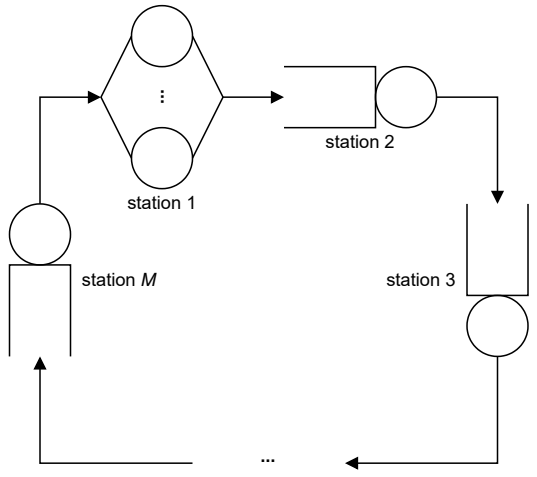


Figure 5.1: The serial topology.

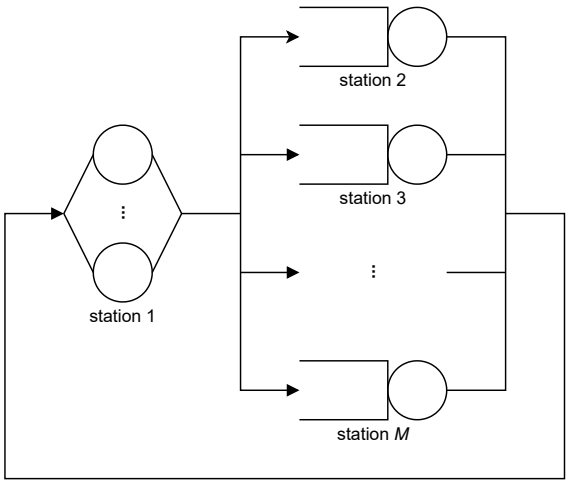


Figure 5.2: The parallel topology.

to a collection of K values sampled from a uniform distribution with interval $[1.0, 9.0]$.[52] Service weights for a model instance were generated in a similar way, except that the unbalanced ones were totally randomized across different DPS stations.[52] When the service rates or the service weights were unbalanced, multiple model instances were obtained by incrementing the random seed from 1 to 30. Particularly, the service rates at the delay station were always balanced and determined such that the mean utilization of every DPS station in the model instance would be 0.9375 if there was a total population of 128 jobs with balanced service rates and service weights. This was to ensure that (5.35) held for most of the model instances in the steady state. Table 5.2 lists model parameter settings for the validations.

Table 5.2: Model parameter settings for the validations.

Dimension	**Optional Values**	**Related Parameters**
Topology	Serial, Parallel	$P_{(i,k),(j,l)}$
Service Time Distributions	Exponential, Coxian*‡	$R_{i,k}$, $\alpha_{i,k,r}$, $P_{i,k,(r,s)}$, $\beta_{i,k,r}$
Number of Stations	3†, 5, 9†	M
Number of Classes	2†, 4, 8†	K
Total Population of Jobs	128†, 256, 512†	N
Service Rates	Balanced: 5.0†, Unbalanced: $[1.0, 9.0]$	$\mu_{i,k}$
Service Weights	Balanced: 5.0†, Unbalanced: $[1.0, 9.0]$	$w_{i,k}$
Random Seed	1, 2†, ..., 30†	$\mu_{i,k}$, $w_{i,k}$

*†‡ These values were exclusive to the validations for transient, steady-state and response time distribution analysis, respectively.

5.6.2 Tool Parameter Settings

We used MATLAB as the platform to run the validation experiments on an Ubuntu server 20.04 LTS with 8 Intel(R) Xeon(R) CPU E3-1280 v5 @ 3.70 GHz and 64 GB RAM. The fluid ODE system for each model instance was solved numerically with ode23, a non-stiff low-order ODE solver built in MATLAB. To decide the truncation point t_∞, we controlled the execution of ode23 precisely by specifying a stop event. Two open-source tools, JMT [BCS09] and LINE [PC17], were also invoked from MATLAB. JMT is an integrated framework for perfor-

[52] Given a model instance with `M` stations and `K` classes, we generated unbalanced service rates and service weights at the DPS stations through the MATLAB commands `8*gallery('uniformdata',[1,K],seed)+1` and `8*gallery('uniformdata',[M-1,K],10000+seed)+1` respectively, where `seed` is the random seed of that model instance.

mance evaluation, capacity planning and workload characterization of computer and communication systems based on queueing theory. A suite of six Java applications are available in JMT, including a discrete-event simulator called JSIM. In a recent release of JMT (v1.0.2), an extension to JSIM has been developed to support simulation of DPS stations. LINE is a MATLAB toolbox for performance and reliability analysis of software applications, business processes and computer networks that can be described by queueing models. The AMVA algorithm for DPS suggested by Lazowska et al. [LZGS84] has recently been implemented in the MVA solver of LINE (v2.0.0). Additionally, LINE provides a programming interface for building QNs and exporting them to JSIM. Table 5.3 lists tool parameter settings for the validations.

Table 5.3: Tool parameter settings for the validations.

Solver	Critical Parameter	Specified Value
MATLAB - ode23	Relative Tolerance	10^{-9}
	Absolute Tolerance	10^{-9}
	Time Span	$[0, 10^5]$
	Stop Event I	$\|d\boldsymbol{x}(t)/dt\|_\infty \leq 10^{-3}$ for 100 time steps*†
	Stop Event II	$1 - \Phi_{i,k}(t) \leq 10^{-3\ddagger}$; $1 - \Phi_{q\text{-}D}(t) \leq 10^{-3\ddagger}$
JMT - JSIM	Random Seed	Random
	Maximum Simulation Time	t_∞[*]; Infinity[†]; $800t_\infty$[‡]
	Automatic Stop	Disabled
	Maximum Samples	$2^{31} - 1$[*‡]; $1.6 \times 10^8/K$[†]
LINE - MVA	Iteration Tolerance	10^{-6}
	Maximum Iterations	10^3

*†‡ These values are exclusive to the validations for transient, steady-state and response time distribution analysis, respectively.

5.6.3 Transient Analysis

The accuracy of the fluid approach for transient analysis was evaluated in terms of the following error metric:

$$\max_{i\in[1..M],k\in[1..K]} \frac{|Q_{i,k}^{\text{app}}(t) - Q_{i,k}^{\text{sim}}(t)|}{N_k} \quad \text{for } t \in [0, T], \tag{5.55}$$

where "app" and "sim" refer to the approximated and simulated results respectively. This metric indicates the maximum percentage of misplaced jobs arising from fluid approximation relative to simulation at finite time $t \in [0, T]$.

The validation experiments for transient analysis were performed on four typical examples. Two were selected from the considered set of model instances, each having a different topology, exponential service time distributions, 5 stations, 4 classes, a total population of 256 jobs as well as unbalanced service rates and service weights that were generated with random seed 1. The other two were created by changing the service time distributions to 2-phase Coxian with coefficient of variation 2.

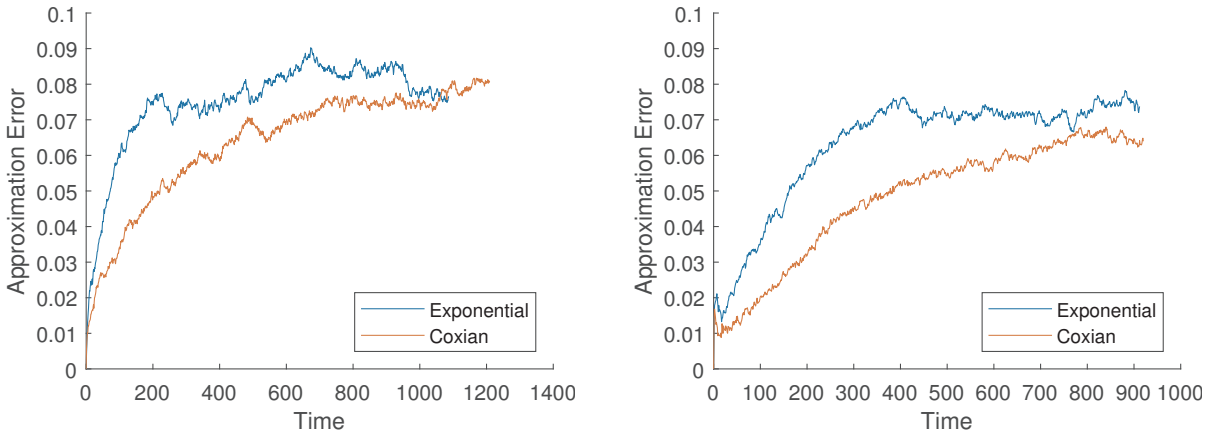
To solve the fluid ODE system for each example, we used ode23 to evolve it until the maximum norm $\|d\boldsymbol{x}(t)/dt\|_\infty$ of the derivative was no more than 10^{-3} for 100 time steps. The transient mean queue lengths were calculated from the numerical solutions across all the time steps by applying (5.37). We set the maximum simulation time of JSIM to the truncation point t_∞, and obtained the simulated results by averaging the sample paths of queue lengths collected in 2000 repeated simulations. The time spent by the fluid approach in computing transient mean queue lengths for each example is reported in Table 5.4. It takes longer to analyze the examples with Coxian service time distributions, because the solution to the fluid ODE system for such an example has more entries.

Table 5.4: The time spent by the fluid approach in computing transient mean queue lengths for each example.

Topology	Service Time Distributions	Computation Time
Serial	Exponential	16.3 s
	Coxian	72.1 s
Parallel	Exponential	13.0 s
	Coxian	64.7 s

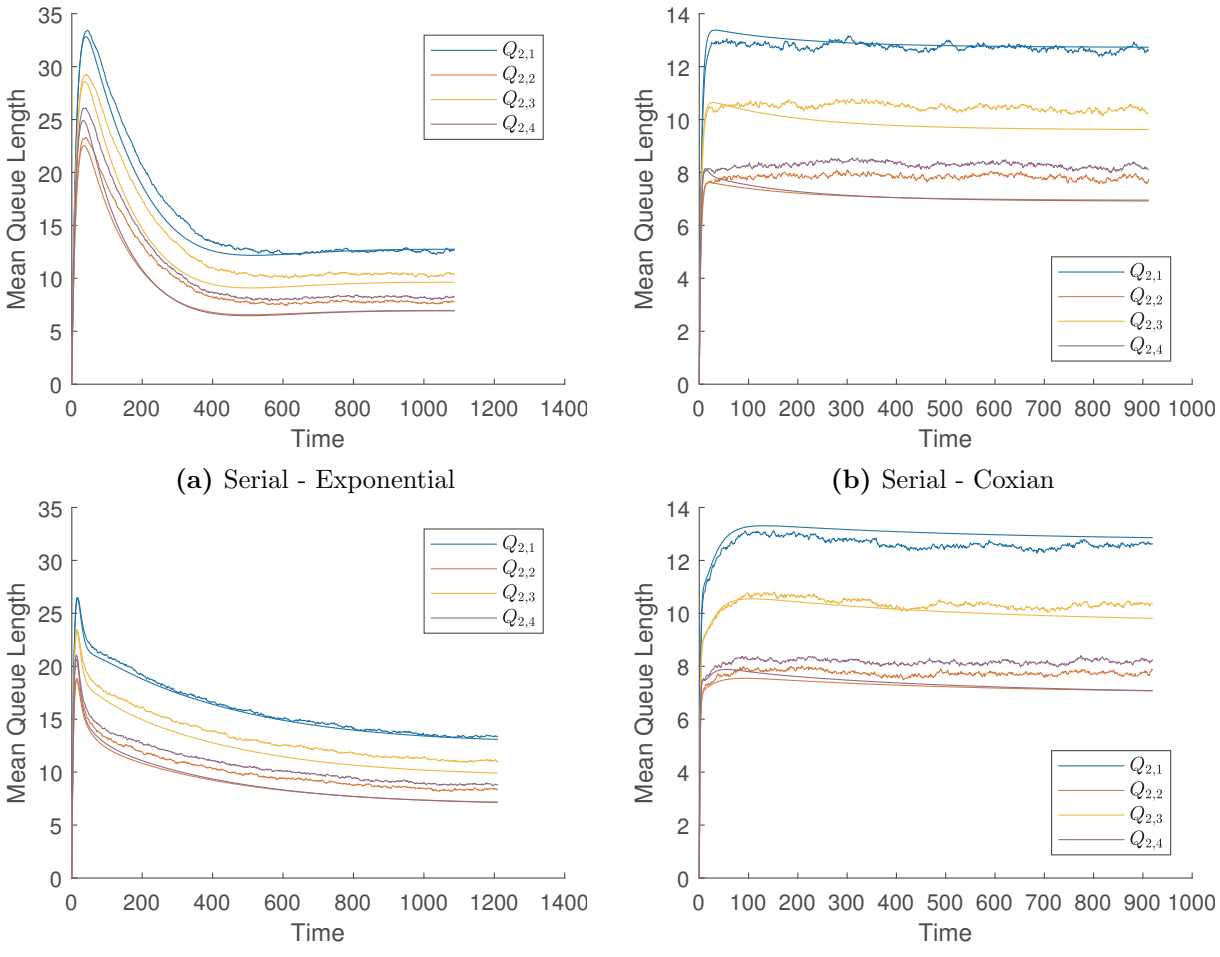
Figure 5.3 shows the error in using the fluid approach to approximate transient mean queue lengths for each example, which increases over time but converges to a limit below 0.1. Thus, the accuracy of the fluid approach for steady-state analysis (see Section 5.6.4) could be thought of as a reference for transient analysis. The approximated and simulated transient mean queue lengths at the first DPS station for each example are shown in Figure 5.4. Comparing Figures 5.4a and 5.4c with 5.4b and 5.4d respectively, we observe that the types of the service time distributions, more technically their higher-order moments, may impact on the transient

mean queue lengths in a model instance but are seemingly a minor factor when it comes to steady-state analysis.



(a) Serial **(b)** Parallel

Figure 5.3: The error in using the fluid approach to approximate transient mean queue lengths for each example.



(a) Serial - Exponential **(b)** Serial - Coxian

(c) Parallel - Exponential **(d)** Parallel - Coxian

Figure 5.4: The approximated and simulated transient mean queue lengths at the first DPS station for each example.

5.6.4 Steady-State Analysis

We evaluated the accuracy of the fluid approach for steady-state analysis according to an error metric defined in [CN82]:

$$\max_{i\in[1..M],k\in[1..K]} \frac{|\tilde{Q}_{i,k}^{\text{app}} - \tilde{Q}_{i,k}^{\text{sim}}|}{N_k}, \tag{5.56}$$

which is a natural extension of (5.55) at infinite time $t \to \infty$. As a comparison, the accuracy of the AMVA algorithm for DPS [LZGS84] was also evaluated using the same metric. This algorithm computes the mean residence time of a class-k job at a DPS station i in a closed QN as

$$\tilde{R}d_{i,k}(\boldsymbol{N}) \approx D_{i,k}\left(1 + \frac{\sum_{l=1}^{K} w_{i,l}\tilde{Q}_{i,l}(\boldsymbol{N}-\boldsymbol{e}_k)}{w_{i,k}}\right), \tag{5.57}$$

where $D_{i,k}$ is the mean service demand of a class-k job at the station i, and $\tilde{Q}_{i,l}(\boldsymbol{N}-\boldsymbol{e}_k)$ is the mean queue length of class-l jobs at the station when the model has one less class-k job.

Table 5.5: The average times spent by the fluid approach and the AMVA algorithm in computing steady-state mean queue lengths for the considered set of model instances.

Parameter Dimension	**Parameter Value**	**Average Computation Time**	
		Fluid	AMVA
Number of Stations	$M = 3$	13.8 s	0.3 s
	$M = 5$	42.7 s	0.3 s
	$M = 9$	188.3 s	0.3 s
Number of Classes	$K = 2$	15.4 s	0.2 s
	$K = 4$	56.4 s	0.3 s
	$K = 8$	173.3 s	0.5 s
Total Population of Jobs	$N = 128$	67.3 s	0.1 s
	$N = 256$	80.3 s	0.3 s
	$N = 512$	97.5 s	0.6 s
Service Rates - Service Weights	B - B	67.8 s	0.4 s
	B - U	91.7 s	0.3 s
	U - B	64.6 s	0.3 s
	U - U	89.3 s	0.3 s

The validation experiments for steady-state analysis were carried out on the entire set of model instances. We used the same stop event as for transient analysis to control the execution of ode23. The steady-state mean queue lengths were calculated from the numerical solution at

the truncation point t_∞ by applying (5.39). Those to be compared with were given by the MVA solver of LINE. We generated the simulated results through a long simulation in which JSIM collected $1.6 \times 10^8/K$ samples to estimate each queue length measure. The average times spent by the fluid approach and the AMVA algorithm in computing steady-state mean queue lengths for all the model instances are reported in Table 5.5. As expected, the computational complexity of the fluid approach depends mainly on the numbers of stations and classes. The reason why the fluid approach analyzes a model instance much slower than the AMVA algorithm is that it has to find the expected sample path all the way down to the truncation point t_∞.

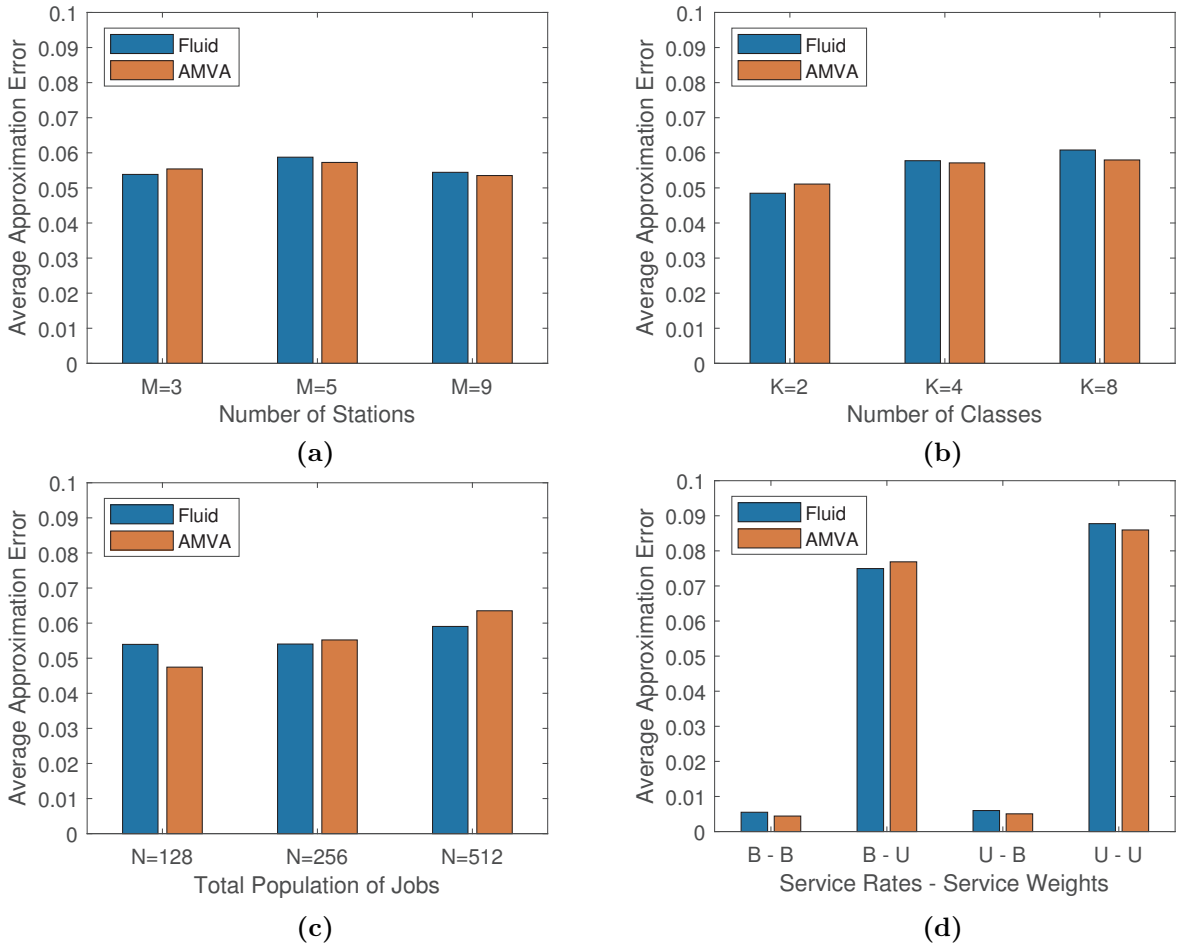


Figure 5.5: The average errors in using the fluid approach and the AMVA algorithm to approximate steady-state mean queue lengths for the considered set of model instances.

Figure 5.5 shows the average errors in using the fluid approach and the AMVA algorithm to approximate steady-state mean queue lengths for all the model instances. The two techniques perform similarly along different dimensions of the model parameter space with overall average errors of 0.0557 and 0.0554, which are almost identical. As shown in Figure 5.5c, the fluid

approach is initially worse but gets increasingly better than the AMVA algorithm as the total population of jobs grows. This reflects a desirable feature of fluid approximation. We further notice from Figure 5.5d that average errors arising from the fluid approach and the AMVA algorithm are below 0.01 when the service weights are balanced, which means that both techniques are highly accurate for model instances where DPS reduces to EPS.

5.6.5 Response Time Distribution Analysis

The 95th percentiles of response time distributions are commonly used in the service-level agreement of a real-world system to specify the required quality of service. Therefore, we focused particularly on this performance measure and evaluated the accuracy of the refined method for approximating response time distributions at individual stations in terms of the following error metric:

$$\max_{i\in[1..M],k\in[1..K]} |\Phi_{i,k}^{\text{app}}((\Phi_{i,k}^{\text{sim}})^{-1}(0.95)) - 0.95|. \tag{5.58}$$

This metric represents the maximum difference between the approximated and simulated station response time CDFs at the 95th percentile. The accuracy of the method at the system level was evaluated according to a similar error metric:

$$\max_{q\in[1..U]} |\Phi_{q\text{-}D}^{\text{app}}((\Phi_{q\text{-}D}^{\text{sim}})^{-1}(0.95)) - 0.95|. \tag{5.59}$$

For comparison, we also evaluated the accuracy of the original method [PC17] using the same metrics.

The validation experiments for response time distribution analysis were conducted on the same examples as for transient analysis. To approximate a station response time distribution, we used ode23 to evolve the fluid ODE system that incorporated the auxiliary class until the complementary CDF $1 - \Phi_{i,k}(t)$ decayed to 10^{-3}. Each system response time distribution was obtained similarly with respect to the delay station. We produced the simulated results based on the samples of response times collected by JSIM during a simulation time of $800t_\infty$ so that the effects of the initial bias were negligible. The times spent by the original and the refined

method in computing station and system response time distributions for each example are reported in Tables 5.6 and 5.7. At the station level, the refined method runs faster than the original one in the case of Coxian service time distributions. As for the system level, it normally needs a slightly longer time to converge.

Table 5.6: The times spent by the original and the refined method in computing station response time distributions for each example.

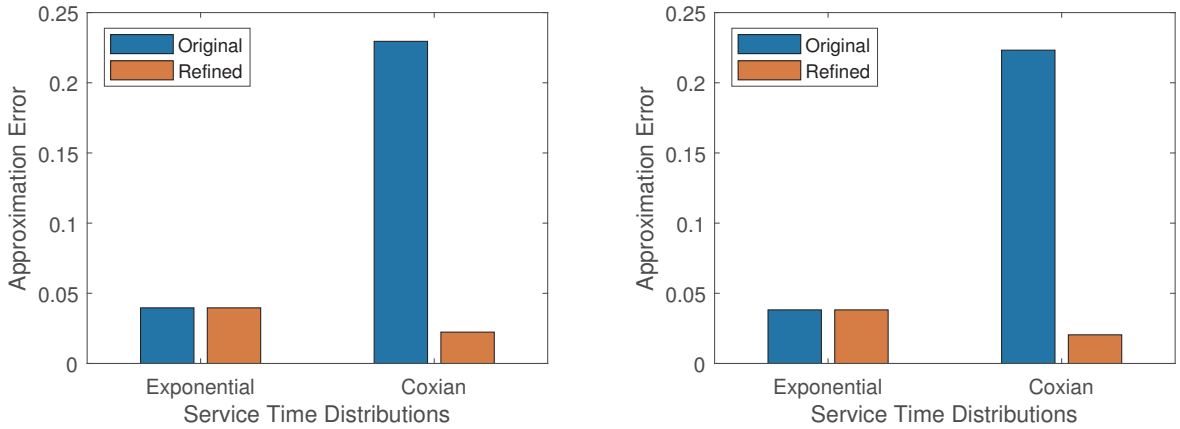
Topology	**Service Time Distributions**	**Computation Time**	
		Original	Refined
Serial	Exponential	232.6 s	232.0 s
	Coxian	1332.2 s	1062.1 s
Parallel	Exponential	221.0 s	221.0 s
	Coxian	1276.2 s	1019.3 s

Table 5.7: The times spent by the original and the refined method in computing system response time distributions for each example.

Topology	**Service Time Distributions**	**Computation Time**	
		Original	Refined
Serial	Exponential	106.6 s	141.6 s
	Coxian	473.6 s	509.0 s
Parallel	Exponential	84.9 s	110.6 s
	Coxian	449.7 s	463.7 s

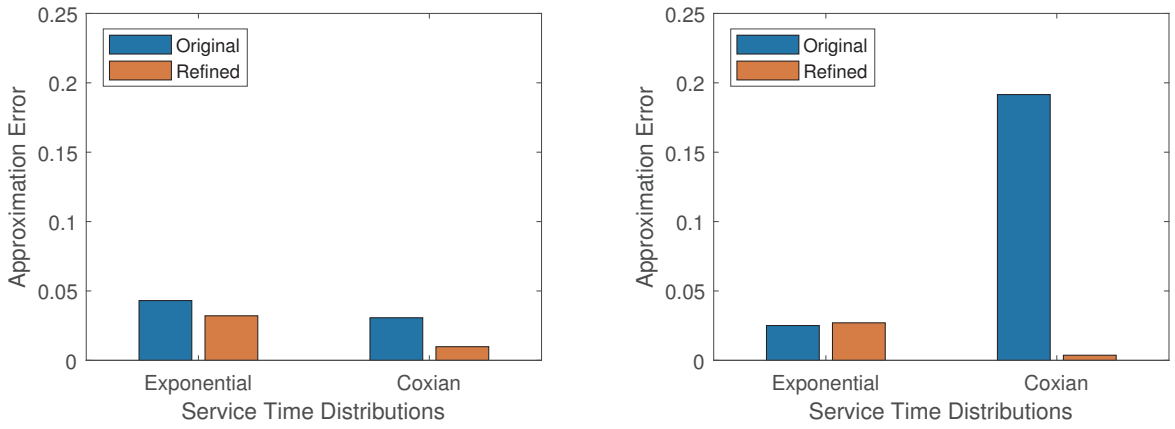
Figures 5.6 and 5.7 show the errors in using the original and the refined method to approximate station and system response time distributions for each example. The refined method generally exhibits higher accuracy than the original one, with an error below 0.05 at both station and system levels. The two methods perform identically when the service time distributions are exponential, because they coincide with each other in this case. By contrast, the refined method is much more accurate for the examples with Coxian service time distributions, where the assumption of the original method fails. Both methods provide good approximations at the system level in the case of a serial topology. The original method is slightly better than the refined one for the parallel example with exponentially distributed service times, which is as expected in the sense that the end phase added by the refined method for each chain at the delay station would introduce a small extra error. Figure 5.8 shows the approximated and

simulated response time CDFs of a class-1 job at the first DPS station for the serial examples, while those of a chain-1 job with respect to the delay station for the parallel examples are shown in Figure 5.9. The latter are actually the resultant system response time CDFs for class 1 as there is no class switching in the examples. It can be seen from the two figures that the approximated results given by the refined method agree well with the simulated results at both station and system levels. This is however not always the case for the original method, especially when the service times are not exponentially distributed.



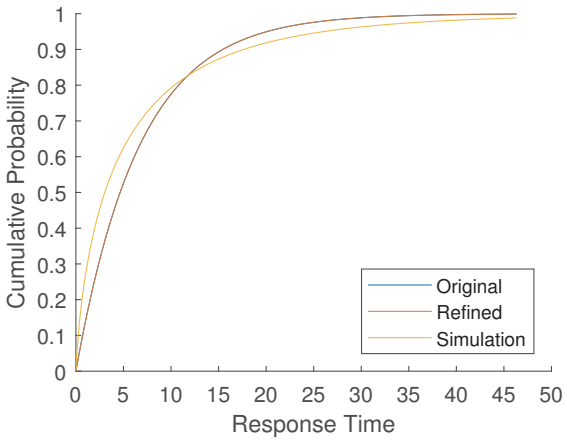
(a) Serial **(b)** Parallel

Figure 5.6: The errors in using the original and the refined method to approximate station response time distributions for each example.

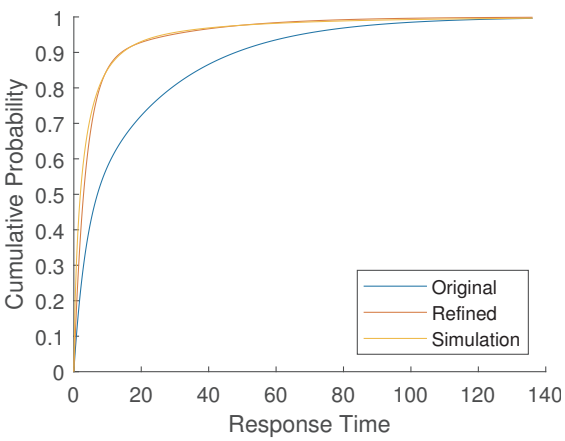


(a) Serial **(b)** Parallel

Figure 5.7: The errors in using the original and the refined method to approximate system response time distributions for each example.

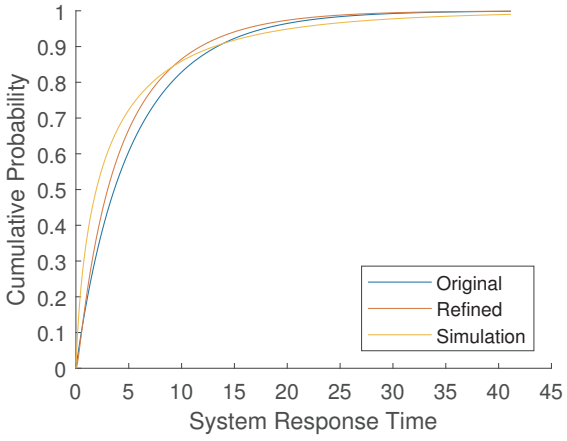


(a) Exponential

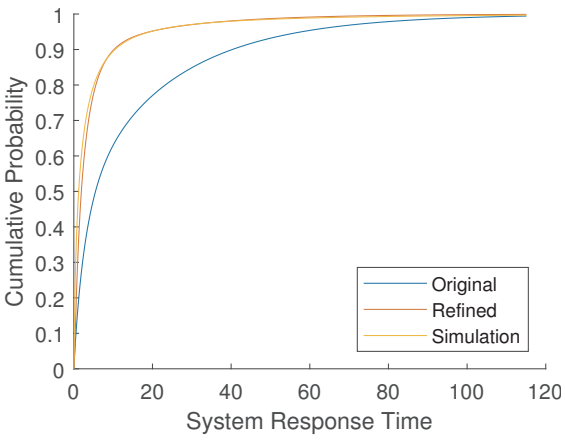


(b) Coxian

Figure 5.8: The approximated and simulated response time CDFs of class-1 jobs at the first DPS station for serial examples.



(a) Exponential



(b) Coxian

Figure 5.9: The approximated and simulated response time CDFs of chain-1 jobs with respect to the delay station for parallel examples.

5.6.6 Summary

Based on the above findings, we believe that the fluid approach can deliver good results for both transient and steady-state analysis. No comparable baselines other than simulation are used in the transient validation since the proposed approach is the only technique that can analyze the transient behavior of our reference model. In the steady-state validation, the accuracy of the fluid approach is observed to be similar to that of the AMVA algorithm, which to our knowledge is the best technique available for steady-state analysis of QNs with DPS. The validation for response time distribution analysis is also important, demonstrating the applicability of the refined method for approximating response time distributions at both station and system levels

as well as its improvement upon the original method in dealing with general phase-type service time distributions.

5.7 Case Study

OpenFaaS[6] and OpenWhisk[7] are two FaaS-oriented frameworks with growing visibility in the open-source community. A case study that applies the fluid approach to predicting the mean response times of OpenFaaS functions is reported in this section. Under certain CPU constraints, we devise a closed QN that comprises delay and EPS stations for modeling the performance of OpenFaaS on a Docker swarm. The relative error of the devised model is no more than 10% and may be somewhat compensated by the additional self-looping jobs if the fluid approach is used to solve it.

5.7.1 OpenFaaS Framework

The OpenFaaS framework integrates various components, including the FaaS provider, the API gateway, NATS[53] and Prometheus,[54] to support deployment, invocation, monitoring and autoscaling of serverless functions. It can be installed on either a Kubernetes[9] cluster or a Docker[10] swarm. In the latter case, both components and functions are deployed as replicated services that are reachable from each other through an overlay network. Each function is associated with a pair of RESTful endpoints exposed by the API gateway. One can thus invoke the function either synchronously or asynchronously by calling the appropriate RESTful endpoint. Modeling asynchronization involves isolating the client and server sides and representing the requests using open classes. For simplicity, we focus on synchronous invocations here.

Figure 5.10 illustrates interactions within the API gateway and an OpenFaaS function during a synchronous invocation. Once the corresponding RESTful endpoint is called, the API gateway forwards the HTTP request to the function. A running process named watchdog works as a

[53] `https://nats.io/`
[54] `https://prometheus.io/`

Web server that listens to the HTTP port of the function and forks a new process to handle each incoming request. The handler process takes the request body as input and produces the response body as output. Data exchange between the watchdog and handler processes are via the standard streams. After acquiring the output of the handler process, the watchdog process sends an HTTP response to the API gateway, which in turn replies to the RESTful call accordingly.

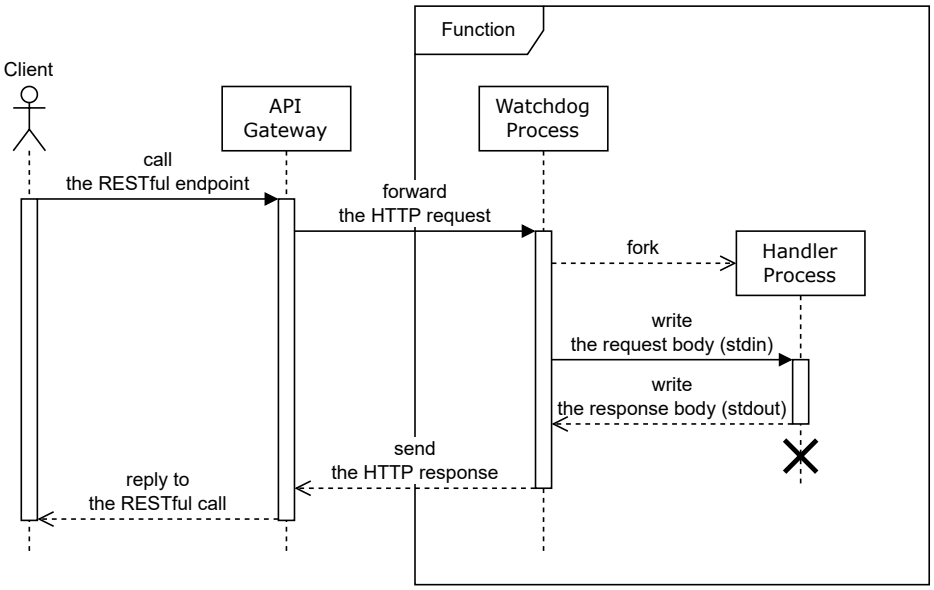


Figure 5.10: Interactions within the API gateway and an OpenFaaS function during a synchronous invocation.

5.7.2 Performance Model

When installed on a Docker swarm, OpenFaaS deploys a serverless function as a service with possibly multiple replicas. Each replica is a copy of the function hosted in a dedicated container. The API gateway addresses a function using a virtual IP shared among its replicas. Thanks to the load balancing feature of Docker, HTTP requests forwarded by the API gateway are distributed equally across the replicas of the function. The Docker engine relies on the built-in cgroup controllers of the Linux kernel for resource management. It essentially creates a cgroup for each container and put all the processes there into that cgroup. As a result, the watchdog and handler processes in a function replica can be thought of as being scheduled on the allocated amount of CPUs in an EPS manner.

Table 5.8: Runtime options for controlling the CPU resources of a Docker container.

Option Flag	Default Value	Description
`--cpus`	c^*	Specify the amount of CPUs available for a container
`--cpu-period`	100000	Specify the CPU scheduling period (µs)
`--cpu-quota`	$-1^\dagger$	Specify the CPU quota available for a container (µs)
`--cpuset-cpus`	1–c^*	Specify the set of CPUs that a container can use
`--cpu-shares`	1024	Specify the weight of a container for sharing CPUs‡

* c is the number of CPUs on the host machine.
† A value of -1 means infinity for `--cpu-quota`.
‡ This actually leads to the GPS scheduling policy.

Table 5.8 lists runtime options for controlling the CPU resources of a Docker container. To model the performance of OpenFaaS, we use two of these options, `--cpuset-cpus` and `--cpu-quota`, to implement the following CPU constraints on the replicas of the framework components and the deployed functions:

- the component and function replicas execute on exclusive sets of CPUs;
- each function replica executes on a specific CPU;
- multiple function replicas share a CPU according to the assigned CPU quotas.

The first constraint can minimize performance interference between the component and function replicas. The second and third constraints aim to override the default multi-server GPS behavior of the underlying scheduler by binding each function replica to a single CPU and meanwhile imposing a hard limit on the available CPU quota.

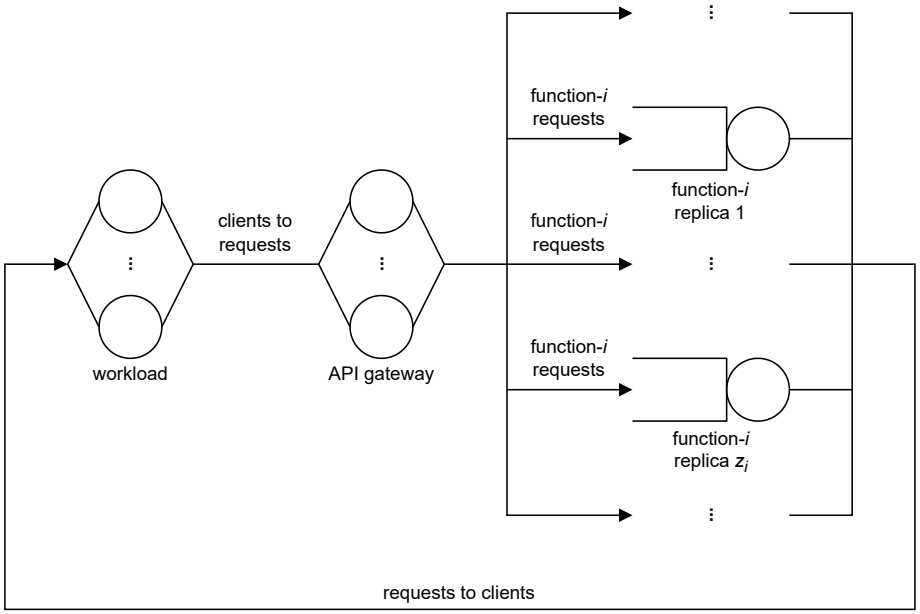


Figure 5.11: The closed QN for modeling the performance of OpenFaaS.

Figure 5.11 shows the closed QN for modeling the performance of OpenFaaS. Suppose that there are m functions and that function i has z_i replicas. Then, the model is composed of $\sum_{i=1}^{m} z_i + 2$ stations and $m+1$ classes. Two delay stations, named workload and API gateway, characterize the think times of clients and the network delays of requests respectively. The queueing stations in parallel adopt the EPS scheduling policy and mimic how requests are processed by the function replicas. One class represents clients who invoke the functions synchronously, whilst each of the rest represents requests to a specific function. After the think time elapses, a client becomes a function-i request following a parametric discrete distribution. The API gateway then forwards the request to any replica of function i with an equal probability, i.e. $1/z_i$. Upon processing completion, the request turns back to the client. We assume the service time distribution for every class at every station to be exponential, which would not affect the generality of the model for steady-state analysis, as discussed in the Section 5.6.3. To capture the sharing of a CPU among multiple function replicas, the mean processing time of a request at each function replica is inflated by a factor of the CPU period over the assigned CPU quota. This technique is based on the same idea as load concealment, used in the MVA algorithm for mixed QNs [ZW81], and has recently been applied by Gias et al. [GCW19] to predicting the performance of microservices deployed on a Docker swarm.

5.7.3 Evaluation Experiments

An evaluation of the performance model for OpenFaaS was conducted on two sample functions provided in the OpenFaaS function store, namely SentimentAnalysis[55] and FaceDetectOpenCV[56]. The former helps to quantify the polarity and subjectivity of a given text, while the latter can recognize and locate human faces in a given image. As a preparation of the experiments, we created a Docker swarm and installed the OpenFaaS framework on an Ubuntu server 20.04 LTS with 32 Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10 GHz and 16 GB RAM. The CPU set for the framework components was restricted to the CPUs 6 to 10 of the server. 50 movie reviews and 50 portrait photos were also selected from the IMDb dataset [MDP+11] and the LFW dataset [HMBLM07] respectively as the inputs of the two functions.

[55] `https://github.com/openfaas/store-functions/tree/master/sentimentanalysis`
[56] `https://github.com/alexellis/facedetect-openfaas`

To benchmark the SentimentAnalysis function, we configured it with only one replica that was bound to the CPU 11, and used the Locust framework[45] to simulate a single client that repeatedly picked a movie review at random and called the RESTful endpoint of the function. The mean processing time of a request was estimated by applying the utilization law. Specifically, we measured the mean utilizations of the CPU and the mean throughputs of requests under different think time settings, and fit (2.48) to the resultant values through least square regression. The mean network delay of a request was calculated by subtracting the estimated mean processing time from the mean response time measured when the think time was set to 1 s. All the measured values were guaranteed within a relative error of 3% at the 99% confidence level. We benchmarked the FaceDetectOpenCV function in a similar way except invoking it with the portrait photos. The benchmarking results of requests to the two OpenFaaS functions are reported in Table 5.9.

Table 5.9: Benchmarking results of requests to the two OpenFaaS functions.

Function	**Model Parameter**	**Estimated Value**
SentimentAnalysis	Mean Processing Time	0.9609 s
	Mean Network Delay	0.0777 s
FaceDetectOpenCV	Mean Processing Time	0.2776 s
	Mean Network Delay	0.0793 s

The performance model for OpenFaaS was evaluated in terms of its accuracy in predicting the mean response times of the SentimentAnalysis and FaceDetectOpenCV functions. We performed three experiments with increasing populations: 8, 12 and 16 clients. Each client thought on average 2 s and proceeded as a request to the two functions with probabilities 0.4 and 0.6. Two replicas were set up for both functions, one on the CPU 11 and the other on the CPU 12. The CPU quotas of replicas belonging to the functions were specified as 66667 and 33333, respectively. We parameterized the performance model accordingly based on the benchmarking results. For comparison, the mean response times of the functions were predicted by solving the model both simulatively and analytically using JSIM and the fluid approach. We computed the relative errors of the simulative and the analytical values against those measured from the real system, whose workload was generated via the Locust framework. The same

statistical criterion as for the benchmarking results were adopted to control the number of samples collected to calculate each of the simulative and the measured values.

Table 5.10: Evaluation results of the performance model for OpenFaaS.

Population	Function	Simulative Value	Analytical Value	Measured Value
8	SentimentAnalysis	2.8094 s (−9.3%)	3.2089 s (+3.6%)	3.0969 s
	FaceDetectOpenCV	1.5086 s (−0.8%)	1.6450 s (+8.2%)	1.5205 s
12	SentimentAnalysis	4.0030 s (−7.7%)	4.5693 s (+5.4%)	4.3374 s
	FaceDetectOpenCV	1.9809 s (−7.2%)	2.1055 s (−1.3%)	2.1336 s
16	SentimentAnalysis	5.4586 s (−6.2%)	6.2187 s (+6.9%)	5.8184 s
	FaceDetectOpenCV	2.4554 s (−5.5%)	2.5579 s (−1.5%)	2.5977 s

Table 5.10 reports the evaluation results of the performance model for OpenFaaS. The relative error of the model is less than 10% according to simulated values, and solving the model with the fluid approach typically leads to a more accurate prediction. This is because the number of context switches required to complete a request increases with respect to the population, which makes its actual mean processing time slightly longer than the estimation. On the other hand, the extra self-looping jobs added by the fluid approach happens to mitigate this misalignment to some extent.

5.8 Conclusion

A fluid approach to transient and steady-state analysis of closed QNs comprising delay and DPS stations is proposed in the chapter. Our reference model has a rather general definition, featuring an arbitrary routing topology, phase-type service time distributions and class switching. Particularly, we manage to remove the discontinuity of the fluid ODE system by the notion of self-looping jobs [CeSL89]. The validation of the proposed approach is performed on a large set of model instances for steady-state analysis and on typical examples for transient analysis. Building on the concept of chains [Mun73], a refined method is also introduced for approximating response time distributions at individual stations and for the entire system through transient analysis. It is remarkably more accurate than prior art [PC17] in the case of

non-exponentially distributed service times. In the case study, a closed QN with EPS is devised for modeling the performance of the OpenFaaS framework on a Docker swarm. The relative error of the model is below 10%, and a better result could be obtained by solving it with the fluid approach.

Chapter 6

Conclusion

6.1 Summary of Thesis Achievements

In the thesis, we have investigated approaches to three challenges posed by the current shift in cloud computing from traditional paradigms toward function as a service (FaaS), i.e. migration of a monolith to the serverless architecture, cost optimization of serverless applications and performance modeling of FaaS-oriented frameworks. The motivation behind this work is to help software vendors unlock the advantages of the FaaS paradigm and relevant technologies in their products.

A semi-automatic approach called RADF is proposed in Chapter 3 for decomposing the architecture of a monolithic application. It enacts a two-stage refactoring strategy, where the entire decomposition process is divided into the coarse- and fine-grained stages. The coarse-grained decomposition aims to identify candidate microservices by clustering the API operations according to their semantic and structural similarities. As for the fine-grained decomposition, each identified microservice is built using serverless functions such that one function implements an associated API operation. We have evaluated RADF in terms of its capability for microservice identification and feasibility for code refactoring. In the evaluation experiments, RADF obtains a decomposition solution with lower coupling and relatively balanced cohesion compared to baselines arising from previous approaches. Although it is not tied to a particular

programming language, RADF relies on language-specific tools to conduct dynamic tracing and static dependency analysis for a given application. In addition, manual effort is still needed afterward to resolve residual dependencies between functions and instantiate the final decomposition solution on the target cloud.

We propose in Chapter 4 a model-driven approach, RDOF, for optimizing the deployment of an application comprising serverless functions and object storages. This approach uses a layered queueing network (LQN) to predict the performance of the application and a genetic algorithm (GA) to search for the optimal deployment scheme that can minimize the total operation cost of the application while satisfying the specified performance requirements under the reference workload. We have evaluated RDOF with respect to the accuracy of the performance model and the utility of the optimization algorithm on Amazon Web Services (AWS). The evaluation results indicate that the relatively error of the LQN for performance prediction is less than 10% and the saving rate of the GA for cost optimization is greater than 10%. RDOF may be extended to Google Cloud, but its applicability to different cloud services is hard limited by the available constructs and features of the LQN formalism. Despite being able to handle the memory allocation and concurrency determination subproblems, the GA normally spends a very long time in searching for the optimal solution. RDOF is thus not a suitable approach in the case of online deployment optimization, where speed is crucial.

Chapter 5 focuses on closed queueing networks (QNs) with discriminatory processor sharing (DPS), to which a product-form solution is yet unknown. General features such as an arbitrary routing topology, phase-type service time distributions and class switching are taken into account in the model definition. A fluid approach to transient and steady-state analysis of this reference model is proposed and exhibits an accuracy comparable to that of the approximate mean value analysis algorithm for DPS in the latter case. Moreover, we introduce a refined method for approximating system and station response time distributions through transient analysis. The method considerably improves upon prior work when the service times of jobs are non-exponentially distributed. A performance model for the OpenFaaS framework on a Docker swarm is devised and can predict the mean response times of the deployed functions within a relative error of no more than 10%. Two preconditions must be satisfied to adapt

the performance model along with the fluid approach for use. On one hand, CPU constraints should be imposed as expected. On the other hand, each function should be under a heavy load. The former is to ensure the validity of the performance model whereas the latter is to guarantee the accuracy of the fluid approach.

RADF and RDOF have been implemented as the architecture decomposition and deployment optimization features of the RADON decomposition tool[11], respectively. To this end, we have conceived novel tool-specific definitions of data, node, relationship and policy types, which enrich the ecosystem of the TOSCA language. The fluid approach for analyzing closed QNs with DPS has been supported by the corresponding solver of the LINE engine.[13] Both of these tools are free for teaching and research purposes.

6.2 Future Work

There is still a wide range of interesting points left to be explored for future work. In the last section, we present several enhancements and applications that are in our mind the most promising for each of the proposed approaches.

RADF: Architecture Decomposition for Function as a Service

The semantic vector of an API operation is computed by combining static word embedding with TF-IDF weighting. However, a more precise representation could be acquired using a dynamic word embedding technique such as BERT [DCLT19] or GPT [RNSS18], which incorporates the current context of each word into the resultant vector. RADF quantifies the correlation between two API operations based on their semantic and structural similarities. It may be helpful to also take into account the so-called data similarity, a measure of to what extent a pair of operations access the same portion of data. As observed in the experiments, the partition that maximizes the silhouette or gamma index is not always the best one that we can find manually. Approximating data points in an orthogonal vector space and optimizing a more sophisticated validity index, for example Krzanowski-Lai [KL88] and CDbw [HV08], is a possible solution to mitigate this disagreement.

RDOF: Deployment Optimization for Function as a Service

Serverless functions exhibit special behavior that may remarkably affect their performance. When a new function instance is created, an extra time is needed to establish the runtime environment of that instance. Also, additional invocations are throttled and retried after a delay if a function does not have enough concurrency. The former is called cold start while the latter is referred to as throttling and retrial. Neither of them can be captured by the LQN used for performance prediction. To cope with this difficulty, we assume the reference workload to be intensive and enforce a small mean buffer length on each function. A more advanced performance model may be introduced to cover the general case. RDOF is applicable to AWS Lambda and extensible to Google Cloud Functions because the underlying implementations of the two FaaS compute services are similar. However, deployment optimization for Microsoft Azure Functions is still an open problem.

Fluid Approximation of Closed Queueing Networks with Discriminatory Processor Sharing

In the validation experiments, the service weight of a self-looping job is chosen as the arithmetic mean over the entries of the local service weight vector. There possibly exist other forms that can lead to higher accuracy for the majority of model instances. Although we have validated the fluid approach against simulation for both transient and steady-state analysis, a supplementary study is desired to uncover the side effects of self-looping jobs preloaded at the DPS stations. The results of the transient validation suggest that steady-state mean queue lengths in the considered model are insensitive to the higher-order moments of the service time distributions. This is an interesting observation to be verified theoretically in a rigorous manner. Furthermore, the fluid approach may be used to provide transient solutions required for analyzing the dynamic behavior of a random environment [CTH14] in which the operating model is a closed QN with DPS.

Bibliography

[AAA06] Eitan Altman, Konstantin Avrachenkov, and Urtzi Ayesta. A Survey on Discriminatory Processor Sharing. *Queueing Systems*, 53(1):53–63, 2006.

[AAB+07] Samuli Aalto, Urtzi Ayesta, Sem Borst, Vishal Misra, and Rudesindo Núñez-Queija. Beyond Processor Sharing. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):36–43, 2007.

[AABNQ05] Konstantin Avrachenkov, Urtzi Ayesta, Patrick Brown, and Rudesindo Núñez-Queija. Discriminatory Processor Sharing Revisited. In *Proceedings of the 24th IEEE International Conference on Computer Communications*, pages 784–795. IEEE, 2005.

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

[AJK04] Eitan Altman, Tania Jimenez, and Daniel Kofman. DPS Queues with Stationary Ergodic Service Times and the Performance of TCP in Overload. In *Proceedings of the 23rd IEEE International Conference on Computer Communications*, pages 975–983. IEEE, 2004.

[ARIM20] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. COSE: Configuring Serverless Functions Using Statistical Learning. In *Proceedings of the 39th IEEE International Conference on Computer Communications*, pages 129–138. IEEE, 2020.

[Bar79] Yonathan Bard. Some Extensions to Multiclass Queueing Network Analysis. In *Proceedings of the 3rd International Symposium on Modelling and Performance Evaluation of Computer Systems*, pages 51–62. North-Holland, 1979.

[BBG10] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing: Principles and Paradigms*. John Wiley & Sons, 2010.

[BBKL14] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

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

[BCMP75] Forest Baskett, Kanianthra M. Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *Journal of the ACM*, 22(2):248–260, 1975.

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

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

[BGDR17] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices Identification through Interface Analysis. In *Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing*, pages 19–33. Springer, 2017.

[BGJM17] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the ACL*, 5:135–146, 2017.

[BH75] Frank B. Baker and Lawrence J. Hubert. Measuring the Power of Hierarchical Cluster Analysis. *Journal of the ASA*, 70(349):31–38, 1975.

[BHLM13] Luca Bortolussi, Jane Hillston, Diego Latella, and Mieke Massink. Continuous Approximation of Collective System Behaviour: A Tutorial. *Performance Evaluation*, 70(5):317–349, 2013.

[BP66] Leonard E. Baum and Ted Petrie. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *Annals of Mathematical Statistics*, 37(6):1554–1563, 1966.

[BT01] Tian Bu and Don Towsley. Fixed Point Approximations for TCP Behavior in an AQM Network. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):216–225, 2001.

[BvOZ05] Sem Borst, Dennis van Ooteghem, and Bert Zwart. Tail Asymptotics for Discriminatory Processor-Sharing Queues with Heavy-Tailed Service Requirements. *Performance Evaluation*, 61(2):281–298, 2005.

[Cas17] Giuliano Casale. Accelerating Performance Inference over Closed Systems by Asymptotic Methods. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):8:1–8:25, 2017.

[CAvdH+20] Giuliano Casale, Matej Artač, Willem-Jan van den Heuvel, André van Hoorn, Pelle Jakovits, Frank Leymann, Mike Long, Vasilis Papanikolaou, Domenico Presenza, Alessandra Russo, Satish N. Srirama, Damian A. Tamburri, Michael Wurster, and Lulai Zhu. RADON: Rational Decomposition and Orchestration for Serverless Computing. *Software-Intensive Cyber-Physical Systems*, 35(1):77–87, 2020.

[CeSL89] Adrian E. Conway, Edmundo S. e Silva, and Stephen S. Lavenberg. Mean Value Analysis by Chain of Product Form Queueing Networks. *IEEE Transactions on Computers*, 38(3):432–442, 1989.

[CGZ19] Giuliano Casale, Alim U. Gias, and Lulai Zhu. Deliverable 3.2 - Decomposition Tool I. Technical report, RADON Consortium, 2019.

[CGZ20] Giuliano Casale, Alim U. Gias, and Lulai Zhu. Deliverable 3.3 - Decomposition Tool II. Technical report, RADON Consortium, 2020.

[CH74] Tadeusz Caliński and Jerzy Harabasz. A Dendrite Method for Cluster Analysis. *Communications in Statistics: Theory and Methods*, 3(1):1–27, 1974.

[CH16] Tiberiu Chis and Peter Harrison. Higher Response Time Moments for M/M/1 Discriminatory Processor Sharing Queues. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 145–148. ICST, 2016.

[CIMS19] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The Rise of Serverless Computing. *Communications of the ACM*, 62(12):44–54, 2019.

[CN82] Kanianthra M. Chandy and Doug Neuse. Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems. *Communications of the ACM*, 25(2):126–134, 1982.

[CTH14] Giuliano Casale, Mirco Tribastone, and Peter G. Harrison. Blending Randomness in Closed Queueing Network Models. *Performance Evaluation*, 82:15–38, 2014.

[CvdBB+05] Sing K. Cheung, Jan L. van den Berg, Richard J. Boucherie, Remco Litjens, and Frank Roijers. An Analytical Packet/Flow-Level Modelling Approach for Wireless LANs with Quality-of-Service Support. In *Proceedings of the 19th International Teletraffic Congress*, pages 1651–1662. Beijing University of Posts and Telecommunications Press, 2005.

[CvHW+20] Matija Cankar, André van Hoorn, Michael Wurster, Vladimir Yussupov, and Lulai Zhu. Deliverable 4.4 - RADON Models II. Technical report, RADON Consortium, 2020.

[DABPF18] Adambarage A. C. De Alwis, Alistair Barros, Artem Polyvyanyy, and Colin Fidge. Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems. In *Proceedings of the 16th International Conference on Service-Oriented Computing*, pages 37–53. Springer, 2018.

[DB78] Peter J. Denning and Jeffrey P. Buzen. The Operational Analysis of Queueing Network Models. *ACM Computing Surveys*, 10(3):225–261, 1978.

[DB79] David L. Davies and Donald W. Bouldin. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, 1979.

[DBT21] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, pages 72–80. AAAI, 2021.

[dCdA19] Leonardo R. de Carvalho and Aletéia P. F. de Araújo. Framework Node2FaaS: Automatic NodeJS Application Converter for Function as a Service. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, pages 271–278. SciTePress, 2019.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186. ACL, 2019.

[DEMF+21] Mohamed Daoud, Asmae El Mezouari, Noura Faci, Djamal Benslimane, Zakaria Maamar, and Aziz El Fazziki. A Multi-Model Based Microservices Identiftcation Approach. *Journal of Systems Architecture*, 118:102200, 2021.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and Thirunavukarasu Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[DS12] Morris H. DeGroot and Mark J. Schervish. *Probability and Statistics*. Addison-Wesley, 2012.

[Dun74] Joseph C. Dunn. Well-Separated Clusters and Optimal Fuzzy Partitions. *Journal of Cybernetics*, 4(1):95–104, 1974.

[DWC10] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud Computing: Issues and Challenges. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 27–33. IEEE, 2010.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI, 1996.

[Elm90] Jeffrey L. Elman. Finding Structure in Time. *Cognitive Science*, 14(2):179–211, 1990.

[ESNA18] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *Proceedings of the 3rd IEEE/ACM Symposium on Edge Computing*, pages 300–312. IEEE, 2018.

[ESVE+21] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Transactions on Software Engineering*, 2021.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[FGZ+18] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. Performance Evaluation of Heterogeneous Cloud Functions. *Concurrency and Computation: Practice and Experience*, 30(23):e4792, 2018.

[FIMS17] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. *arXiv preprint arXiv:1708.08028*, 2017.

[FMI80] Guy Fayolle, Isi Mitrani, and Roudolf Iasnogorodski. Sharing a Processor among Many Job Classes. *Journal of the ACM*, 27(3):519–532, 1980.

[FMW+13] Greg Franks, Peter Maly, Murray Woodside, Dorina C. Petriu, Alex Hubbard, and Martin Mroz. *Layered Queueing Network Solver and Simulator User Manual*. Department of Systems and Computer Engineering, Carleton University, 2013.

[FNAD20] Nazanin Firoozeh, Adeline Nazarenko, Fabrice Alizon, and Béatrice Daille. Keyword Extraction: Issues and Methods. *Natural Language Engineering*, 26(3):259–291, 2020.

[Fra00] Roy G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton University, 2000.

[GBG+18] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning Word Vectors for 157 Languages. In *Proceedings of the 11th International Conference on Language Resources and Evaluation*, pages 3483–3487. ELRA, 2018.

[GCW19] Alim U. Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven autoscaling for microservices. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, pages 1994–2004. IEEE, 2019.

[GHK+20] Alireza Goli, Omid Hajihassani, Hamzeh Khazaei, Omid Ardakanian, Moe Rashidi, and Tyler Dauphinee. Migrating from Monolithic to Serverless: A FinTech Case Study. In *Companion of the 11th ACM/SPEC International Conference on Performance Engineering*, pages 20–25. ACM, 2020.

[GKGZ16] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service Cutter: A Systematic Approach to Service Decomposition. In *Proceedings of*

the 5th European Conference on Service-Oriented and Cloud Computing, pages 185–200. Springer, 2016.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[GVH17] Nicolas Gast and Benny Van Houdt. A Refined Mean Field Approximation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):33:1–33:28, 2017.

[Har54] Zellig S. Harris. Distributional Structure. *Word*, 10(2):146–162, 1954.

[HMBLM07] Gary B. Huang, Marwan Mattar, Tamara Berg, and Eric Learned-Miller. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical report, University of Massachusetts, 2007.

[HT05] Yezekael Hayel and Bruno Tuffin. Pricing for Heterogeneous Services at a Discriminatory Processor Sharing Queue. In *Proceedings of the 4th IFIP International Conference on Networking*, pages 816–827. Springer, 2005.

[HV08] Maria Halkidi and Michalis Vazirgiannis. A Density-Based Cluster Validity Approach Using Multi-Representatives. *Pattern Recognition Letters*, 29(6):773–786, 2008.

[IAV16] Ane Izagirre, Urtzi Ayesta, and Ina M. Verloop. Sojourn Time Approximations for a Discriminatory Processor Sharing Queue. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(1):5:1–5:30, 2016.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[JLC+19] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service Candidate Identification from Monolithic Systems Based on Execution Traces. *IEEE Transactions on Software Engineering*, 47(5):987–1007, 2019.

[JM09] Dan Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 2009.

[Joh67] Stephen C. Johnson. Hierarchical Clustering Schemes. *Psychometrika*, 32(3):241–254, 1967.

[Jon72] Karen S. Jones. A Statistical Interpretation of Term Specificity and Its Application in Retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[JSSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca A. Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[KK04] Jeongsim Kim and Bara Kim. Sojourn Time Distribution in the M/M/1 Queue with Discriminatory Processor-Sharing. *Performance Evaluation*, 58(4):341–365, 2004.

[KL88] Wojtek J. Krzanowski and Y. T. Lai. A Criterion for Determining the Number of Groups in a Data Set Using Sum-of-Squares Clustering. *Biometrics*, 44(1):23–34, 1988.

[Kle51] Stephen C. Kleene. Representation of Events in Nerve Nets and Finite Automata. Technical report, RAND Corporation, 1951.

[Kle56] Stephen C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, 34:3–42, 1956.

[Kle67] Leonard Kleinrock. Time-Shared Systems: A Theoretical Treatment. *Journal of the ACM*, 14(2):242–261, 1967.

[KP10] Walter G. Kelley and Allan C. Peterson. *The Theory of Differential Equations: Classical and Qualitative*. Springer, 2010.

[KPSCD09] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. Estimating Service Resource Consumption from Response Time Measurements. In *Proceedings of the 4th ICST International Conference on Performance Evaluation Methodologies and Tools*, pages 1–10. ICST, 2009.

[KR90] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 1990.

[Kru64] Joseph B. Kruskal. Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[Kur70] Thomas G. Kurtz. Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes. *Journal of Applied Probability*, 7(1):49–58, 1970.

[KXK+21] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1214–1224. ACM, 2021.

[LA99] Bjornar Larsen and Chinatsu Aone. Fast and Effective Text Mining Using Linear-Time Document Clustering. In *Proceedings of the 5th ACM International Conference on Knowledge Discovery and Data Mining*, pages 16–22. ACM, 1999.

[LAK17] Euripidis Loukis, Spyros Arvanitis, and Niki Kyriakou. An Empirical Investigation of the Effects of Firm Characteristics on the Propensity to Adopt Cloud Computing. *Information Systems and e-Business Management*, 15(4):963–988, 2017.

[Lar12] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Pearson, 2012.

[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.

[LMP01] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289. Morgan Kaufmann, 2001.

[LPS+20] Robin Lichtenthäler, Mike Prechtl, Christoph Schwille, Tobias Schwartz, Pascal Cezanne, and Guido Wirtz. Requirements for a Model-Driven Cloud-Native Migration of Monolithic Web-Based Applications. *Software-Intensive Cyber-Physical Systems*, 35(1):89–100, 2020.

[LRC+18] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *Proceedings of the 6th IEEE International Conference on Cloud Engineering*, pages 159–169. IEEE, 2018.

[Luh57] Hans P. Luhn. A Statistical Approach to Mechanized Encoding and Searching of Literary Information. *IBM Journal of Research and Development*, 1(4):309–317, 1957.

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

[LZJ+19] Shanshan Li, He Zhang, Zijia Jia, Zheng Li, Cheng Zhang, Jiaqi Li, Qiuya Gao, Jidong Ge, and Zhihao Shan. A Dataflow-Driven Approach to Identifying Microservices from Monolithic Applications. *Journal of Systems and Software*, 157:110380, 2019.

[LZZ15] Joseph Lilleberg, Yun Zhu, and Yanqing Zhang. Support Vector Machines and Word2vec for Text Classification with Semantic Features. In *Proceedings of the 14th IEEE International Conference on Cognitive Informatics and Cognitive Computing*, pages 136–140. IEEE, 2015.

[MA98] Daniel A. Menasce and Virgilio A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice Hall, 1998.

[Mac67] James MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[Mar03] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[MC17] Karsten Molka and Giuliano Casale. Energy-Efficient Resource Allocation and Provisioning for In-Memory Database Clusters. In *Proceedings of the 15th IFIP/IEEE Symposium on Integrated Network and Service Management*, pages 19–27. IEEE, 2017.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv preprint arXiv:1301.3781*, 2013.

[MCL17] Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of Microservices from Monolithic Software Architectures. In *Proceedings of the 24th IEEE International Conference on Web Services*, pages 524–531. IEEE, 2017.

[MCPB20] Lisa Muller, Christos Chrysoulas, Nikolaos Pitropakis, and Peter J. Barclay. A Traffic Analysis on Serverless Computing Based on the Example of a File Upload Stream on AWS Lambda. *Big Data and Cognitive Computing*, 4(4):38, 2020.

[MDP+11] Andrew Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the ACL*, pages 142–150. ACL, 2011.

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

[Mor91] John A. Morrison. Asymptotic Analysis of a Large Closed Queueing Network with Discriminatory Processor Sharing. *Queueing Systems*, 9(1):191–213, 1991.

[MR00] Laurent Massoulié and James W. Roberts. Bandwidth Sharing and Admission Control for Elastic Traffic. *Telecommunication Systems*, 15(1):185–201, 2000.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.

[Mun73] Richard R. Muntz. Poisson Departure Process and Queueing Networks. In *Proceedings of the 7th Annual Princeton Conference on Information Sciences and Systems*, pages 435–440. Princeton University Press, 1973.

[MW89] Debasis Mitra and Alan Weiss. A Closed Network with a Discriminatory Processor-Sharing Server. *ACM SIGMETRICS Performance Evaluation Review*, 17(1):200–208, 1989.

[NDMG+16] Joakim Nivre, Marie-Catherine De Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. Universal Dependencies v1: A Multilingual Treebank Collection. In *Proceedings of the 10th International Conference on Language Resources and Evaluation*, pages 1659–1666. ELRA, 2016.

[New15] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[NG04] Mark E. J. Newman and Michelle Girvan. Finding and Evaluating Community Structure in Networks. *Physical Review E*, 69(2):026113, 2004.

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

[PG93] Abhay K. Parekh and Robert G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[PG94] Abhay K. Parekh and Robert G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994.

[Por80] Martin F. Porter. An Algorithm for Suffix Stripping. *Program*, 14(3):130–137, 1980.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 3483–3487. ACL, 2014.

[RAK07] Usha N. Raghavan, Réka Albert, and Soundar Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3):036106, 2007.

[RC17] Michael Roberts and John Chapin. *What is Serverless?* O'Reilly Media, 2017.

[Ric18] Chris Richardson. *Microservices Patterns: With Examples in Java.* Simon & Schuster, 2018.

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

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. Technical report, OpenAI, 2018.

[Ros14] Sheldon M. Ross. *Introduction to Probability Models.* Academic Press, 2014.

[Rou87] Peter J. Rousseeuw. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[RS94] Kiran M. Rege and Bhaskar Sengupta. A Decomposition Theorem and Related Results for the Discriminatory Processor Sharing Queue. *Queueing Systems*, 18(3):333–351, 1994.

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

[RS96] Kiran M. Rege and Bhaskar Sengupta. Queue-Length Distribution for the Discriminatory Processor-Sharing Queue. *Operations Research*, 44(4):653–657, 1996.

[RS08] Jerome A. Rolia and Kenneth C. Sevcik. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2008.

[SCBK15] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, 92:51–71, 2015.

[Sch79] Paul J. Schweitzer. Approximate Analysis of Multiclass Closed Networks of Queues. In *Proceedings of the International Conference on Stochastic Control and Optimization*, pages 25–29, 1979.

[SD17] Josef Spillner and Serhii Dorodko. Java Code Analysis and Transformation into AWS Lambda Functions. *arXiv preprint arXiv:1702.05510*, 2017.

[SG98] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 1998.

[SM00] Jianbo Shi and Jitendra Malik. Normalized Cuts and Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[Spa05] James C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. John Wiley & Sons, 2005.

[Spi17] Josef Spillner. Transformation of Python Applications into Function-as-a-Service Deployments. *arXiv preprint arXiv:1705.08169*, 2017.

[STM21] Adam Stafford, Farshad Ghassemi Toosi, and Anila Mjeda. Cost-Aware Migration to Functions-as-a-Service Architecture. In *Companion of the 15th European Conference on Software Architecture*. CEUR-WS, 2021.

[TWYZ19] Damian A. Tamburri, Michael Wurster, Vladimir Yussupov, and Lulai Zhu. Deliverable 4.3 - RADON Models I. Technical report, RADON Consortium, 2019.

[vKNQB04] Gijs van Kessel, Rudesindo Núñez-Queija, and Sem Borst. Asymptotic Regimes and Approximations for Discriminatory Processor Sharing. *ACM SIGMETRICS Performance Evaluation Review*, 32(2):44–46, 2004.

[VL07] Ulrike Von Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems*, pages 6000–6010. Curran, 2017.

[WLZ+18] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 133–146. USENIX, 2018.

[WNPM95] Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. *IEEE Transactions on Computers*, 44(1):20–34, 1995.

[XT15] Dongkuan Xu and Yingjie Tian. A Comprehensive Survey of Clustering Algorithms. *Annals of Data Science*, 2(2):165–193, 2015.

[YBLW19] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Michael Wurster. A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 229–240. ACM, 2019.

[ZCP20] Lulai Zhu, Giuliano Casale, and Iker Perez. Fluid Approximation of Closed Queueing Networks with Discriminatory Processor Sharing. *Performance Evaluation*, 139:102094, 2020.

[ZGTC21] Lulai Zhu, Giorgos Giotis, Vasilis Tountopoulos, and Giuliano Casale. RDOF: Deployment Optimization for Function as a Service. In *Proceedings of the 14th IEEE International Conference on Cloud Computing*, pages 508–514. IEEE, 2021.

[ZTC22] Lulai Zhu, Damian A. Tamburri, and Giuliano Casale. RADF: Architecture Decomposition for Function as a Service. Manuscript in preparation for publication, 2022.

[ZW81] John Zahorjan and Eugene Wong. The Solution of Separable Queueing Network Models Using Mean Value Analysis. *ACM SIGMETRICS Performance Evaluation Review*, 10(3):80–85, 1981.