

Imperial College of Science, Technology and Medicine
Department of Computing

Stochastic Performance Modeling of Distributed Data Processing Systems

Yicheng Gao

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

Statement of Originality

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

Copyright Declaration

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

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

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

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

Abstract

Microservice architectures are widely adopted in distributed data processing systems due to their scalability, flexibility, and fault tolerance. Different architectural designs can significantly influence key performance metrics, such as response time and throughput. Therefore, developing accurate performance models during the design phase is crucial to ensure that microservice architectures can meet performance requirements while operating efficiently. However, predicting performance in such distributed systems is challenging due to the complexity of layered architectures, where data flows through multiple components, creating complex latency interplay and workload-driven variability.

To address these challenges, we introduce a series of novel stochastic performance modeling techniques tailored for microservices operating within distributed data processing systems. First, we present LN, an open-source meta-solver for layered queueing network analysis, to more accurately capture the layered interactions and dependencies in microservice architectures. Building upon this, we propose ICQ, a novel class of stochastic models that enables joint analysis of data storage access workflows, caching, and queueing contention. Our evaluation, using real-world Azure traces, shows that ICQ can improve response times by up to 35% in edge computing systems compared to baseline heuristics. To further enhance data locality in distributed microservice architectures, we design a hierarchical caching model to optimize data placement across in-memory cache nodes and provide analytical solutions for various caching policies. Simulations using real-world Alibaba traces highlight the scalability and effectiveness of this approach, demonstrating up to a 21% reduction in miss ratio and a 37% decrease in mean response time in data mesh scenarios. Lastly, we address the challenges posed by batch arrivals in serverless edge computing by developing a deep surrogate model. Extensive experiments with AI-SPRINT traces validate the accuracy of our proposed model, achieving a mean absolute percentage error of 5.33%, thus offering a scalable and efficient approach to performance prediction for serverless applications.

Acknowledgements

First and foremost, I want to express my deepest gratitude to my supervisor, Giuliano Casale. His constant support and profound research expertise have been a guiding light throughout my PhD journey. He not only helped me navigate the complexities of my work but also gave me the encouragement and strength to persevere through challenging times, both in my studies and in my life.

I am truly grateful for the time I spent visiting Politecnico di Milano. I had the privilege of working under the guidance of Professor Danilo Ardagna, whose project provided me with invaluable insights into serverless applications, a core aspect of this thesis. His support and timely guidance were immensely helpful. I am also deeply thankful to Roberto and Enrico for their patience in walking me through the project details, assisting with data collection, and engaging in so many fruitful discussions. I will always fondly remember my time there with the DeepSE research group, not just for the work we did, but for those espresso breaks, amazing food, and spirited beach volleyball games.

I would like to extend my heartfelt gratitude to Rekha Singhal for being a wonderful collaborator on our paper focused on performance modeling of distributed data processing. Her industry expertise, particularly on data mesh, was invaluable and played a critical role in shaping key aspects of my research. I truly appreciated her support and guidance throughout the entire process.

I am also grateful to my incredible colleagues at QORE Lab. Their insightful discussions, valuable feedback, and idea-sharing have been instrumental in my growth. Alongside our academic collaboration, I will always treasure the memories we created together, from organizing the ICPE conference to enjoying meals and having fun at group events. These experiences made my PhD journey all the more enriching.

Finally, I want to express my heartfelt gratitude to my family and friends. Your love, patience, and unwavering belief in me have been my constant anchor throughout this journey. You were always there to lift me up during the tough times, and I truly could not have made it through without your support.

Dedication

To my father Chunsheng, who first showed me the Big Dipper, and who now shines as a star in the sky, forever lighting my way through the rest of my life.

To my partner Xin, whose bright laughter, courage, love, and care carried me through every joy and challenge of my PhD journey.

Contents

Statement of Originality	i
Abstract	iii
Acknowledgements	v
List of Acronyms	xxi
1 Introduction	1
1.1 Motivation and Objectives	1
1.2 Contributions	5
1.3 Publications	8
1.4 Thesis Organization	10
2 Background	12
2.1 Introduction	12
2.2 Microservices and Serverless Architectures	13
2.2.1 Overview	13

2.2.2	Related Work	18
2.3	Caching	21
2.3.1	Caching Policies	23
2.3.2	Hierarchical Caching Architecture	24
2.4	Extended Queueing Networks	26
2.4.1	Queueing Networks	26
2.4.2	Extended Queueing Networks Solution	30
2.5	Layered Queueing Networks	32
2.5.1	Layering Strategy	34
2.5.2	Submodel Generation	35
2.5.3	Performance Metrics	37
2.5.4	Fixed Point LQN Solution	38
2.6	Matrix Analytic Method	39
2.6.1	Phase-Type Distributions	39
2.6.2	Markovian Arrival Process (MAP)	43
2.6.3	Applications of MAM in Queueing Modeling	45
2.7	Summary	48
3	Algorithmic Framework for Layered Queueing Network Analysis	49
3.1	Introduction	49
3.2	LQN Meta-Solution	52
3.2.1	Motivating Example	52

3.2.2	LN Meta-Solver	54
3.3	Algorithms for Homogeneous Queueing Networks	56
3.3.1	CoMoM Algorithm for Homogeneous Models with Single Server Queues .	57
3.3.2	CoMoM Algorithm for Homogeneous Models with Multiserver Queues . .	62
3.3.3	Gaussian Quadrature Methods	65
3.3.4	Non-iterative MVA Approximation	68
3.4	Experiments and Case Studies	71
3.4.1	Experiments on Random Models	71
3.4.2	Case Study: Multi-Solver Based Layer Analysis	73
3.4.3	Case Study: Multi-Formalism Capabilities	76
3.5	Summary	78
4	Layered Caching-Queueing Modeling	80
4.1	Introduction	80
4.2	Motivating Examples	84
4.3	LQN Modeling of Distributed Workflows	84
4.3.1	System Model	85
4.3.2	LQN Modeling	86
4.3.3	Problem Formulation	87
4.3.4	Evaluation	88
4.4	Integrated Caching-Queueing Analysis	91
4.4.1	Layered Caching-Queueing Model	92

4.4.2	Analytical Solution for the ICQ model	93
4.5	Model Validation and Evaluation	96
4.5.1	Validation	96
4.5.2	Experimental Evaluation	100
4.6	Summary	106
5	Hierarchical Caching Modeling	107
5.1	Introduction	107
5.2	Caching Modeling	110
5.2.1	Hierarchical Data Distribution	110
5.2.2	Replacement Policies with Access Paths	111
5.3	Analytical Methods	112
5.3.1	Exact Solution for RR- $a(\mathbf{m})$ Policy	113
5.3.2	TTL Approximation for LRU- $a(\mathbf{m})$ Policy	118
5.4	Model Validation	120
5.4.1	Hierarchical Caching Model Validation	120
5.4.2	Validation Under Real Traffic	121
5.5	Application: Data Mesh	123
5.5.1	Introduction	123
5.5.2	Problem Statement	126
5.5.3	Experimental Evaluations	130
5.6	Summary	133

6	Surrogate Modelling for Tandem BMMPP/GI/c Queues	135
6.1	Introduction	135
6.2	System Characteristics	138
6.2.1	AI-SPRINT Applications and Profiling	138
6.2.2	Trace Collection and Analysis	141
6.3	Motivation	144
6.3.1	Traffic Decomposition Methods	144
6.3.2	Multiserver Approximation Analysis	145
6.3.3	Tandem Workflow Approximation	147
6.3.4	Issues with Approximation Methods	150
6.4	Surrogate Modeling	151
6.4.1	Performance Model	151
6.4.2	Deep Surrogate Model	154
6.5	Experimental Results	156
6.5.1	Simulation-based Evaluation	156
6.5.2	AI-SPRINT Application Trace	159
6.6	Summary	162
7	Conclusion	163
7.1	Summary of Thesis Achievements	163
7.2	Future Work	165

Bibliography	166
A LN Software Architecture Design	190

List of Tables

2.1	Open & Closed Queueing Networks	27
2.2	Single-class & Multi-class Queueing Networks	28
2.3	Common Properties of Queueing Buffers	28
2.4	Common Properties of Servers	29
3.1	Solver selection statistics for 20,000 randomly generated homogeneous models with FCFS and PS stations	53
3.2	LN automated paradigm selection heuristics in LINE 2.0.27	55
3.3	Relative error and runtime upon computing $\log G(m, N)$ exactly.	62
3.4	Relative error and runtime upon approximating $\log G(m, N)$	67
3.5	Parameter settings for generating random LQNs	72
3.6	Solution methods for solving LQN models	72
4.1	Parameter settings for the comparative experiments	90
4.2	Parameter settings for the evaluation of the ICQ model	103
5.1	Comparing the miss ratio difference between the analytical method and trace- driven simulation. β is the ratio of cache capacity to the number of items. . . .	122

6.1	Parameters for training data generation	158
6.2	Training hyperparameters	158

List of Figures

2.1	Context and techniques mapping to thesis chapters	14
2.2	Monolith architecture vs. Microservice architecture	15
2.3	Key components of Docker architecture	15
2.4	Key components of Kubernetes cluster architecture, including a master node and multiple worker nodes (Node 1, 2, ..., n) managing containerized workloads. . . .	16
2.5	An example of serverless event-driven E-commerce microservices architecture . .	18
2.6	Example of an LQN model	33
2.7	A layer of the LQN model with replication $m = 2$ and multiplicity $c = 3$	35
3.1	Mean absolute percentage error for X_r on 10,000 for (a) the non-iterative approximation (3.10) compared to (b) an asymptotic upper bound [DB78]. Figure (c) shows the ratio of runtime of (3.10) compared to AQL [Zah79], showing that the non-iterative approximation is much faster and the gap grows with more classes.	70
3.2	Performance of different solution methods in analyzing LQN models: (a) mean error, (b) median error, (c) max error	73
3.3	Leveraging LINE meta-solver capabilities in analyzing LQN models.	74
3.4	Example of a multi-formalism model containing a caching layer (middle task) . .	76

3.5	Capabilities of LINE in analyzing multi-formalism LQN models	78
4.1	A dependency graph of social network microservice architecture [GLD ⁺ 21,GZC ⁺ 19]	83
4.2	A use case of video content delivery chain in edge computing [Com18]	83
4.3	Edge computing system containing N users	85
4.4	An example of layered queueing networks modeling for distributed processing of microservices	86
4.5	An overview of the comparative procedure between the LQN-based model and the method in [ZXZ ⁺ 20]	89
4.6	Average gain of response time of the LQN-based model compared to the method in [ZXZ ⁺ 20] ($M = 2$)	90
4.7	Average gain of response time of the LQN-based model compared to the method in [ZXZ ⁺ 20] ($M = 5$)	91
4.8	An example of an LQN model containing one cache-task	93
4.9	Decomposition of the caching sub-model in the ICQ model (The top diagram shows the original caching sub-model, which is decomposed into the two sub- models below. The dashed boxes denote the boundaries of the upper and lower sub-models.)	94
4.10	A queueing Petri net used for validation via JMT	99
4.11	Comparison of average residence time between analytical and simulated methods	99
4.12	Characteristics of the selected random subset of the real-world Azure traces . . .	101
4.13	An example of the generalized ICQ model generated by LINE (The yellow triangles, pink triangles, red squares, blue circles and black hexagrams represent reference tasks, tasks, entries, activities and processors respectively.)	102

4.14	Total response time with respect to the increase of the number of users N (Azure dataset)	104
4.15	Total response time with respect to the increase of the number of services C (Azure dataset)	104
4.16	Sensitivity to p/q and C/M ratio with $\eta = 1$ (Azure dataset)	105
5.1	An example of a tree-structured cache for data mesh. The cache list $l = 0$ represents out-of-cache data items.	109
5.2	Comparing the miss ratio of parallel-structured caches between analytical and simulated methods	121
5.3	Comparing the miss ratio of tree-structured caches between analytical and simulated methods	121
5.4	Autocorrelation of the original and shuffled 2008 Youtube trace	122
5.5	Data mesh architecture	124
5.6	Core components of the data flow in each domain	124
5.7	Architectural designs for the storage of data products in data mesh	127
5.8	An example of the transformation from the data caching model to the integrated data caching and processing model	129
5.9	Statistical characteristics of the sampled Alibaba traces	131
5.10	Miss ratio and mean response time with respect to the increase of the number of data products (Alibaba dataset)	133
5.11	Miss ratio and mean response time with respect to the increase of the average call graph size (Alibaba dataset)	134
6.1	An example of an AI-SPRINT application for mask detection	138

6.2	Detailed logging process of OSCAR-P with execution stages and timestamps . .	140
6.3	Probability mass function of batch sizes across different core configurations . . .	141
6.4	Autocorrelation function of batch sizes across different core configurations	142
6.5	Autocorrelation function of inter-arrival times between batches	143
6.6	An example of a tandem queueing network with batch arrivals	145
6.7	Approximation error for multiserver scenarios	146
6.8	Approximation error for complete tandem workflows	148
6.9	Approximation error for tandem workflows with different arrival process fitting methods in the subsequent queues	149
6.10	Time complexity analysis for BMMPP/GI/1 models	151
6.11	Feedforward Neural Network Architecture	155
6.12	Comparison of the response time between JMT simulator and surrogate model .	159
6.13	Comparison of the response time between trace-based simulations and deep sur- rogate model, with shaded 95% confidence intervals shown for the surrogate predictions.	160
6.14	CDF of the prediction error of the deep surrogate model	161
A.1	Key architectural elements of LINE.	190

List of Acronyms

ACF	Autocorrelation Function
AI-SPRINT	Artificial Intelligence in Secure PRIVacy-preserving computing coNTinuum
AMVA	Approximate Mean Value Analysis
API	Application Programming Interface
AQL	Aggregate Queue Length
AWS	Amazon Web Services
BMAP	Batch Markovian Arrival Process
BMMPP/GI/c	Batch Markov-Modulated Poisson Process / General Independent / c-Server
CDF	Cumulative Distribution Function
CNN	Convolutional Neural Networks
CoMoM	Class-oriented Method of Moments
CTMC	Continuous-Time Markov Chain
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
DTMC	Discrete-Time Markov Chain
EQN	Extended Queueing Network
ESB	Enterprise Service Bus
ETAQA	Efficient Technique for Analyzing QBD Processes by Aggregation
ETL	Extract, Transform, and Load
FaaS	Function-as-a-Service
FCFS	First Come First Served
FGSD	Family of Graph Spectral Distances

FIFO	First-In-First-Out
Fluid	Mean-Field Approximations
FPI	Fixed Point Iteration
GA	Genetic Algorithm
GI	General Independent
HC	Hierarchical Data Caching
HCP	Hierarchical Data Caching and Processing
HOL	Head-Of-Line
HST	Half-Space Tree
IaaS	Infrastructure as a Service
ICQ	Integrated Caching and Queueing
i.i.d.	independent and identically distributed
IoT	Internet of Things
IRM	Independent Reference Model
JMT	Java Modelling Tools
K8s	Kubernetes
LCFS	Last Come First Served
LE	Logistic Expansion
LFU	Least Frequently Used
LN	Layered Network
LQN	Layered Queueing Network
LRU	Least-Recently-Used
MAP	Markovian Arrival Process
MAPE	Mean Absolute Percentage Error
MAM	Matrix Analytic Method
MDN	Mixture Density Network
MMPP	Markov Modulated Poisson Process
MNA	Matrix Network Analyzer
MVA	Mean Value Analysis

NC	Normalizing Constant
ODT-SC	Offloading Dependent Tasks with Service Caching
OSCAR	Open Source Serverless Computing for Data-Processing Applications
PDF	Probability Density Function
PEPA	Performance Evaluation Process Algebra
PH	Phase-Type
PMF	Probability Mass Function
PS	Processor Sharing
QBD	Quasi-Birth-Death (Process)
QN	Queueing Network
QNA	Queueing Network Analyzer
QPN	Queueing Petri Net
ReLU	Rectified Linear Unit
REST	Representational State Transfer
RFR	Random Forest Regression
RPC	Remote Procedure Call
RR	Random-Replacement
SaaS	Software as a Service
SCAR	Serverless Container-aware ARchitectures
SCV	Squared Coefficient of Variation
SIRO	Service In Random Order
SLA	Service Level Agreement
SMT	Satisfiability Modulo Theory
SOA	Service-Oriented Architecture
SPA	Singular Perturbation Approximation
SPN	Stochastic Petri Net
SSA	Stochastic Simulation
TTL	Time-To-Live
VM	Virtual Machine

Chapter 1

Introduction

1.1 Motivation and Objectives

The increasing availability of diverse communication networks has driven modern applications toward more distributed architectures [Lam11], spanning large-scale internet-based systems like cloud platforms (*e.g.*, Amazon Web Services [Ama15]) and streaming services (*e.g.*, Netflix [Lob19]), as well as localized networks of devices and sensors such as those found in industrial Internet of Things (IoT) or edge computing system. As these distributed data processing systems continue to grow in complexity and scale, the architectural styles of software development and deployment have evolved to meet the demands of supporting and maintaining such systems [SZY⁺16, SBB21].

This evolution has led to a paradigm shift from the traditional client-server model [IHM00], where clients initiate service requests to servers, with responses handled through request-response protocols, commonly utilizing Remote Procedure Calls (RPC), toward more flexible and scalable architectures. Service-oriented architecture (SOA) [PVDH07] emerged as a solution by modularizing application logic into loosely coupled, reusable services. Typically, SOA implementations rely on a centralized Enterprise Service Bus (ESB) to serve as the connection layer [SHLP05], routing user requests, transforming messages, and managing communications between services, ensuring interoperability across diverse platforms. However, the reliance on

the centralized ESB introduces scalability constraints and a single point of failure, complicating system management as service interactions increase.

In response to the limitations of SOA, microservices architecture emerged as a more decentralized and fine-grained alternative [AAE16]. This architecture decomposes applications into independent, autonomous services that communicate via lightweight Application Programming Interfaces (APIs), enabling each service to be developed, deployed, and scaled independently. Unlike SOA, where services are centrally managed through the ESB, microservices eliminate this dependency, promoting more direct service-to-service communication. Therefore, adopting the microservices architecture in distributed data processing systems offers several key advantages:

- **Scalability:** microservices architecture supports independent scaling of each service, either vertically or horizontally [GCW19]. Specifically, vertical scaling increases the resources allocated to an individual service instance, while horizontal scaling replicates multiple instances to manage higher workloads. This elastic scaling allows the system to adjust dynamically to varying workloads without impacting the overall system, optimizing resource utilization in distributed systems.
- **Flexibility:** microservices architecture allows independent development, deployment, updating, migration, and removal of individual services during runtime [Wol16]. This modular approach enhances flexibility and agility by enabling faster iterations, reducing downtime during updates, and facilitating parallel work across teams, thereby improving system adaptability to evolving workloads and requirements of distributed data processing.
- **Fault tolerance:** microservices architecture improves system resilience by isolating failures within individual services, ensuring that the failure of one component does not disrupt the entire system [BOP22]. This isolation allows other services to continue operating, maintaining system functionality even during service outages. Moreover, critical services can be deployed with redundancy, further bolstering reliability and ensuring uninterrupted performance in distributed data processing systems.

Although adopting microservices architectures provides significant benefits, the architectural decisions made during the design phase, such as service decomposition, interaction patterns, and data management, directly impact the overall system performance, especially when deployed in distributed systems. Improper architectural design can lead to excessive communication overhead, data inconsistencies, and bottlenecks in high-traffic services [SWG21], undermining both scalability and reliability.

Therefore, predicting performance during the design phase is crucial to ensure that the microservice architecture not only meets performance requirements such as response time and throughput, but also functions efficiently within the broader distributed system [MAKR19]. By building performance models early in the development cycle, architects can simulate different microservice architectures, evaluate potential bottlenecks, perform what-if analysis, and optimize resource allocation across distributed nodes. This proactive approach helps prevent costly refactorings later in development and ensures that the system can effectively handle varying workloads in distributed data processing systems.

However, accurately predicting the performance of microservices in distributed data processing systems presents several challenges due to their layered characteristics. Each layer, spanning application, middleware, and infrastructure, plays a distinct role, yet their tightly coupled dependencies complicate performance analysis. In such distributed environments, data often traverses multiple layers and distributed nodes, leading to the complicated interplay between data access workflows, caching mechanisms, and computational processes [GLD⁺21, GZC⁺19]. Additionally, varying workloads, especially batch arrivals, can cause queueing contention, which amplifies these interactions and exacerbates performance bottlenecks. These interdependencies introduce significant variability, making it difficult to predict system performance and ensure efficient resource allocation across distributed environments.

Based on these discussions, we define the following research objectives for this thesis.

- **RO1. Layered Queueing Network Modeling:** The first objective is to advance existing stochastic models for layered analysis, specifically Layered Queueing Networks

(LQNs), to more accurately capture the layered interactions and dependencies in microservice architectures.

- **RO1.1 LQN Meta-solver:** The first sub-objective is to offer multiple solution paradigms for each layer, enabling the dynamic selection of the most appropriate algorithm for analysis. This flexibility enhances accuracy while ensuring greater generality and extensibility.
- **RO1.2 Layer Solution:** The second sub-objective is to develop algorithms that improve upon existing techniques by enhancing accuracy or increasing efficiency in the analysis of each layer.
- **RO2. Layered Caching-Queueing Modeling:** The second objective is to develop a novel class of stochastic models that facilitates the joint analysis of data storage access workflows, caching, and queueing contention. This model is designed to enable a comprehensive analysis of data processing in microservices within distributed edge computing systems. It should capture not only computational aspects but also data access dynamics and the effects of cache misses on in-memory data storage nodes.
 - **RO2.1 LQN Modeling:** The first sub-objective is to demonstrate the advantages of employing LQN models for the distributed processing of computational services within microservice applications, thereby establishing a robust foundation for advanced analytical exploration.
 - **RO2.2 Integrated Analysis:** The second sub-objective is to generalize LQN models to incorporate caching characteristics, thereby establishing a novel framework for jointly modeling in-memory storage and computational processes of microservices in distributed systems.
- **RO3. Hierarchical Caching Modeling:** The third objective is to establish a hierarchical caching model that defines valid storage locations for data within a distributed network of in-memory nodes in microservice architectures. These models should then be solved considering common caching policies to ensure efficient data access and storage management.

- **RO3.1 Caching Modeling:** The first sub-objective is to formulate the caching modeling to describe the storage locations of data items within a hierarchically distributed network of in-memory storage nodes.
- **RO3.2 Analytical Methods:** The second sub-objective is to study the analytical solutions of the established hierarchical caching model considering different caching policies.
- **RO4. Surrogate Modeling:** The fourth objective is to build analytical performance models for serverless edge computing that account for the unique challenges of running microservices in resource-constrained edge environments. Specifically, we aim to design models that address the complexities of distributed workflow dependencies and the impact of batch job arrivals on system performance, providing accurate performance predictions at design time without relying on costly large-scale profiling experiments.
 - **RO4.1 Performance Model:** The first sub-objective is to formulate an analytical performance model to capture the impact of batch arrivals and distributed workflow dependencies in microservices, enabling precise performance predictions for serverless edge computing environments.
 - **RO4.2 Surrogate Model:** The second sub-objective is to develop surrogate models to efficiently approximate the performance metrics of the formulated analytical model, offering a scalable solution for fast and accurate predictions of microservice performance in design-time serverless edge computing systems.

1.2 Contributions

The contributions of this thesis are summarized as follows.

- To address our first research objective, **RO1**, we present the Layered Network solver (LN), an open-source meta-solver framework for LQN analysis that allows for the dynamic selection of the performance analysis paradigm to be iteratively applied to the

submodels arising from layer decomposition. We report experiments where this added flexibility helps us to reduce the LQN solution errors. Additionally, to support the accurate evaluation of the LQN submodels, we develop novel algorithms for homogeneous queueing networks consisting of an infinite server node and a set of identical queueing stations. In particular, (1) we introduce a method of moments algorithm that solves homogeneous models in linear time and extends it to multiserver stations, achieving significant speedups for large multiclass models; (2) we develop Gaussian quadrature methods for approximating normalizing constants, balancing accuracy and speed, especially for models with large populations; and (3) we propose a non-iterative Mean Value Analysis (MVA)-based approximation for fast model solutions, achieving superior accuracy compared to existing non-iterative approaches [DB78, LZGS84], though with slightly lower precision than iterative methods.

- To address our second research objective, **RO2**, we propose the Integrated Caching-Queueing (ICQ) model, a novel class of stochastic models that enables joint analysis of data storage access workflows, caching, and queueing contention. We first apply LQN modeling to microservice placement in edge computing, achieving a 16% improvement in system response time for stochastic workloads compared to the deterministic method in [ZXZ⁺20], highlighting the effectiveness of LQN modeling for distributed workflows. Then we generalize LQN models with caching components, enabling the analysis of data processing by means of workflow graphs that describe not only computation, but also data access dynamics and the effect of cache misses upon accessing an in-memory data storage node. The resulting ICQ model is analyzed by a novel fixed-point iteration method to approximate steady-state performance metrics. Finally, we use a Queueing Petri Net (QPN)-based model to validate accuracy and simulate real-world Azure traces to evaluate the ICQ model, demonstrating up to 35% improvement in response time and a 500MB reduction in memory usage compared to baseline heuristics.
- To address our third research objective, **RO3**, we design a hierarchical caching model to capture data locality in microservice architectures and develop analytical solutions to efficiently approximate the performance metrics of these models under various data

caching policies. First, we generalize existing product-form solutions for list-based caches to provide an analytical formalism to describe flexible data placement. In particular, we use the notion of cache list to model an in-memory storage node and our analytical formalism is extended to describe allowed locations where a data item can be stored in a hierarchically distributed network of in-memory storage nodes. The latter is achieved by introducing a notion of data item access path, which is essential to model organizational, regulatory (*e.g.*, privacy-related), or geographical constraints on data placement. Next, we study the exact solution for these models under random-replacement (RR) policy and adopt a time-to-live (TTL) approximation to analyze the models under least-recently-used (LRU) policy. Lastly, simulations using real-world Alibaba traces demonstrate the scalability and effectiveness of our proposed models, showing up to a 21% reduction in miss ratio and a 37% decrease in mean response time in the data mesh scenarios.

- To address our fourth research objective, **RO4**, we propose a surrogate modeling approach for performance prediction in microservices deployed in serverless edge computing systems. Initially, we analyze real-world trace characteristics and model the system as a tandem queueing network. A decomposition technique is applied to break down the tandem workflow into individual nodes, each of which is modeled by a batch Markov-modulated Poisson process / general independent / c-server (BMMPP/GI/c) queue to capture the effects of batch arrivals and workflow dependencies. Further, due to the computational complexity of solving this model, we propose a deep surrogate model, which efficiently approximates the performance metrics of the analytical model while maintaining high accuracy. Extensive simulations and real-world AI-SPRINT traces validate the accuracy of our proposed surrogate model, with the traces showing a strong alignment and achieving an overall Mean Absolute Percentage Error (MAPE) of 5.33%, demonstrating its suitability for scalable and efficient performance evaluation in serverless edge computing.

1.3 Publications

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

[GC22] Y Gao, G Casale. **JCSP: Joint Caching and Service Placement for Edge Computing Systems**. IEEE/ACM 30th International Symposium on Quality of Service (IWQoS), 2022.

In this paper, we propose ICQ, a novel class of stochastic models that enables joint analysis of data storage access workflows, caching, and queueing contention. My contributions include modeling distributed stochastic workflows using LQN, generalizing the LQN model to incorporate caching characteristics, developing a fixed-point iteration method to approximate steady-state performance metrics, validating the model based on Queueing Petri Nets, and evaluating its performance using real-world traces. These contributions are discussed in detail in Chapter 4 and address our second research objective **RO2**.

[CGNZ22] G Casale, Y Gao, Z Niu, L Zhu. **LN: A Meta-solver for Layered Queueing Network Analysis**. International Conference on Quantitative Evaluation of Systems (QEST), 2022.

This paper is a collaborative work by our research group (QORE Lab¹). In this paper, we presented the LN meta-solver, which enhances the analysis of LQNs by allowing the flexible combination of different solution paradigms. In this work, I contributed to the definition of the meta-solver and the integration of ICQ models into the framework. Part of my contribution is incorporated in Chapter 3 and addresses our first research objective **RO1**.

[GGS⁺23] AU Gias, Y Gao, M Sheldon, JA Perusquía, O O’Brien, G Casale. **SampleHST: Efficient On-the-Fly Selection of Distributed Traces**. IEEE/IFIP Network Operations and Management Symposium (NOMS), 2023.

This work is a collaborative project between our research group and an industry partner. It presents SampleHST, a novel approach for trace sampling in distributed tracing systems specif-

¹<https://qore.doc.ic.ac.uk>

ically designed for microservice architectures. The method employs Half Space Trees (HSTs) and online clustering to selectively store anomalous traces, significantly improving sampling efficiency. I contributed to developing and testing of the methodology, which aligns closely with the background of my PhD thesis and supports the context for our research objectives **RO1** and **RO2**.

[GCS24] Y Gao, G Casale, R Singhal. **Performance Modeling of Distributed Data Processing in Microservice Applications**. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS), 2024.

This paper is a collaborative work with Rekha Singhal from Tata Consulting Services. The journal version extends our earlier work in [GC22] by introducing hierarchical caching models for distributed data processing systems. I contributed to defining access paths for modeling data placement across cache nodes and developed analytical solutions for performance metrics under various caching policies. Simulations with real-world traces show significant performance improvements. These contributions are discussed in detail in Chapter 4 and 5 and address our research objectives **RO2** and **RO3**.

[CGNZ24] G Casale, Y Gao, Z Niu, L Zhu. **LN: A Flexible Algorithmic Framework for Layered Queueing Network Analysis**. ACM Transactions on Modeling and Computer Simulation (TOMACS), 2024.

This paper is a collaborative work by our research group (QORE Lab). It generalizes our previous work on LN [CGNZ22] by introducing a flexible meta-solver for LQNs and several new algorithms for homogeneous queueing networks. I mainly contributed to the meta-solution framework and the integration of the ICQ model, enhancing the performance and adaptability of the LN solver. Part of my contribution is incorporated in Chapter 3 and addresses our first research objective **RO1**.

[GGS⁺24] AU Gias, Y Gao, M Sheldon, JA Perusquía, O O’Brien, G Casale. **SampleHST-X: A Point and Collective Anomaly-Aware Trace Sampling Pipeline with Approximate Half Space Trees**. Journal of Network and Systems Management (JNSM), 2024.

This work is a collaborative project between our research group and an industry partner. The journal version extends our work in [GGS⁺23] by introducing SampleHST-X, which incorporates approximate Half Space Trees (HST) for reduced space requirements and a Family of Graph Spectral Distances (FGSD) component for identifying and sampling collective anomalies in addition to point anomalies. I contributed to developing and testing of the methodology, which aligns closely with the background of my PhD thesis and supports the context for our research objectives **RO1** and **RO2**.

In addition, we are in the process of preparing a paper submission that includes the contributions from Chapter 6 and addresses our research objective **RO4**.

1.4 Thesis Organization

The structure of the thesis is organized as follows.

- In Chapter 1, we present the motivation, research objectives, and main contributions of this thesis.
- In Chapter 2, we provide the background and preliminaries of techniques and concepts essential for understanding this thesis.
- In Chapter 3, we address our first research objective **RO1**. Here, we present LN, an open-source meta-solver framework for LQN analysis and develop novel algorithms for homogeneous queueing networks.
- In Chapter 4, we address our second research objective **RO2**. Here, we propose ICQ, a novel class of stochastic models that enables joint analysis of data storage access workflows, caching, and queueing contention.
- In Chapter 5, we address our third research objective **RO3**. Here, we design a hierarchical caching model to describe flexible data placement for microservices and give analytical solutions under different caching policies.

- In Chapter 6, we address our fourth research objective **RO4**. Here, we build surrogate models to approximate the performance of complex analytical models for microservices in serverless edge computing.
- In Chapter 7, we conclude by highlighting the thesis contributions and proposing future research directions.

Unless otherwise stated, confidence intervals are used throughout the thesis to quantify variability in results obtained from repeated runs or randomized instances. They are omitted from plots when visually negligible to improve clarity.

Chapter 2

Background

2.1 Introduction

This chapter provides the foundational background necessary to contextualize the contributions of this thesis, focusing on cloud-native architectures and their associated performance evaluation techniques. As cloud-native methodologies continue to reshape the landscape of modern distributed data processing systems, this thesis examines two key architectural paradigms: microservices [Zim17] and serverless computing [MB17]. Microservices offer scalability and flexibility by decomposing applications into fine-grained, independent services that can be developed, deployed, and scaled independently. On the other hand, serverless computing operates through an event-driven execution model in which functions are automatically triggered by specific events and dynamically scaled in response to workload demand, eliminating the need for infrastructure management. Together, these cloud-native architectures enable the development and deployment of flexible and efficient modern applications.

To facilitate the analysis and optimization of these cloud-native architectures, this thesis draws upon a suite of performance modeling and evaluation techniques. Figure 2.1 illustrates the interrelationships between these architectures and the performance evaluation methods that serve as the foundation for our proposed modeling approach. Among these foundational techniques, layered queueing networks (LQNs) [FAOW⁺08, SD11, AVZ19] build upon extended queueing

networks (EQNs) by introducing hierarchical abstractions, enabling the modeling of multi-layer systems where various layers interact to deliver services. Additionally, the matrix analytic method (MAM) is employed as a powerful solution technique [RS02a, RS07], particularly well-suited for deriving performance metrics from complex queueing models where exact analytical solutions are required. These techniques form the basis for the advanced modeling approach proposed in this thesis, which addresses the performance challenges in distributed systems with dynamic interactions and dependencies.

The rest of this chapter is organized as follows. Section 2.2 explores microservices and serverless architectures, highlighting their roles in distributed cloud-native systems. Section 2.3 provides essential background on caching mechanisms and policies. Sections 2.4 and 2.5 lay the foundation for the advanced modeling techniques presented in Chapters 3-5 by introducing LQNs. We begin with an overview of EQNs in Section 2.4 and then explain how they evolve into LQNs to accommodate the complexities of multi-tiered systems in Section 2.5. In Section 2.6, we introduce the MAM as a complementary solution technique to obtain precise performance metrics for queueing modeling, which will be applied in Chapter 6. Finally, we present the conclusion in Section 2.7.

2.2 Microservices and Serverless Architectures

2.2.1 Overview

Microservices and Containers

Since the concept of microservices architecture was first introduced in 2014 [Zim17], it has evolved as a main architectural style for modern software design and development, gaining widespread adoption across diverse application domains. Unlike traditional monolithic architecture, microservice architecture promotes the decomposition of applications into a collection of fine-grained, loosely coupled, autonomous services, as presented in Figure 2.2, each

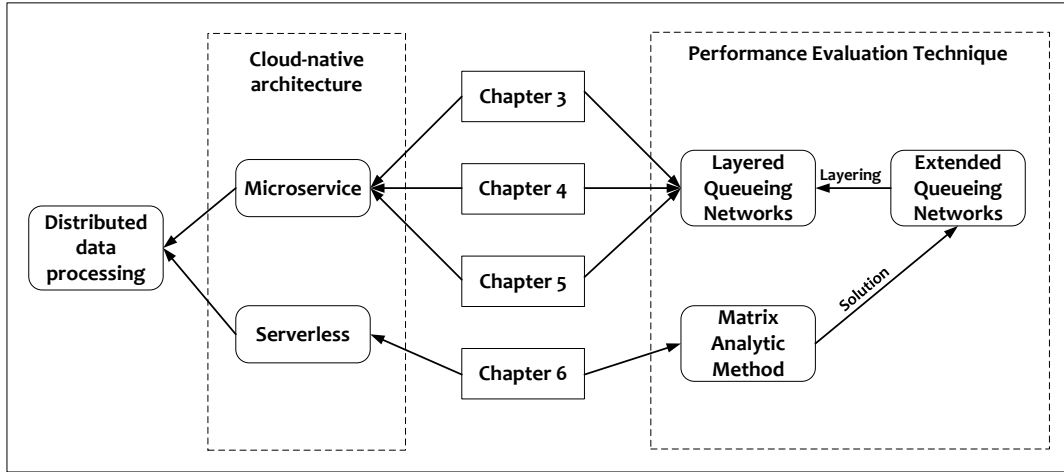


Figure 2.1: Context and techniques mapping to thesis chapters

dedicated to fulfilling a granular business functionality. These services are independently developed, deployed, and managed, and the communication between services using lightweight protocols like RESTful APIs [BHJ16, Sil16], enhancing greater flexibility, scalability, and adaptability [WGZ⁺19, CDT18]. Therefore, many leading companies, such as Amazon, Netflix, and Spotify, have adopted microservices architecture to enhance their system resilience and operational efficiency [NMMA16].

To effectively deploy microservices, container technology has emerged as one of the most widely adopted approaches. Containers provide an isolated, lightweight environment that packages a microservice along with its dependencies, libraries, and tools [APC⁺15], ensuring consistency across various deployment environments. One of the most widely used platforms for containerization is Docker [Mou15], which facilitates the entire process of creating, managing, and deploying containers. Specifically, Docker architecture is composed of three main elements: the Docker client, the Docker daemon, and the registry, as illustrated in Figure 2.3. The containerization process typically begins with the Docker client, where developers issue commands including [Mer14]:

- **docker build:** This command initiates the creation of a Docker image, which packages the application code, dependencies, and configurations required for the microservice.
- **docker push:** Once the image is built, it can be stored locally or pushed to a remote repository, such as Docker Hub or a private registry, making it accessible for other developers

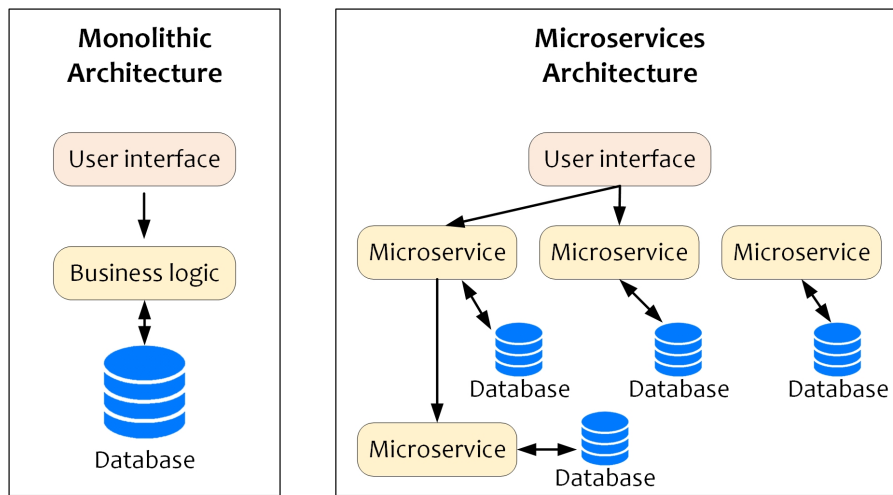


Figure 2.2: Monolith architecture vs. Microservice architecture

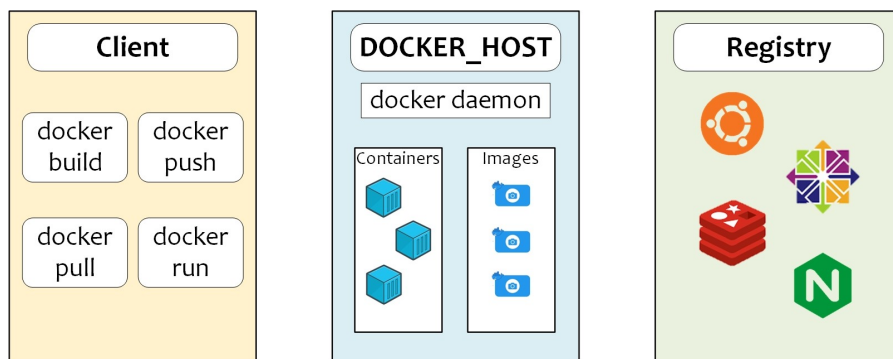


Figure 2.3: Key components of Docker architecture

or future deployments.

- **docker pull:** When the image is ready to be used, the Docker client can send a docker pull request to retrieve the image from the registry to the Docker host.
- **docker run:** This command instructs the Docker daemon on the host to launch a new container, which functions as a runtime instance of the Docker image.

The distinction between a Docker image, which encapsulates the service along with its dependencies, and a container, which represents the running instance of that image, greatly enhances both flexibility and scalability. In particular, it enables efficient scaling, as multiple containers can be instantiated from a single Docker image. Moreover, it ensures that microservices are portable across various environments, including cloud-based and on-premises infrastructures. By providing a consistent runtime environment and eliminating configuration discrepancies,

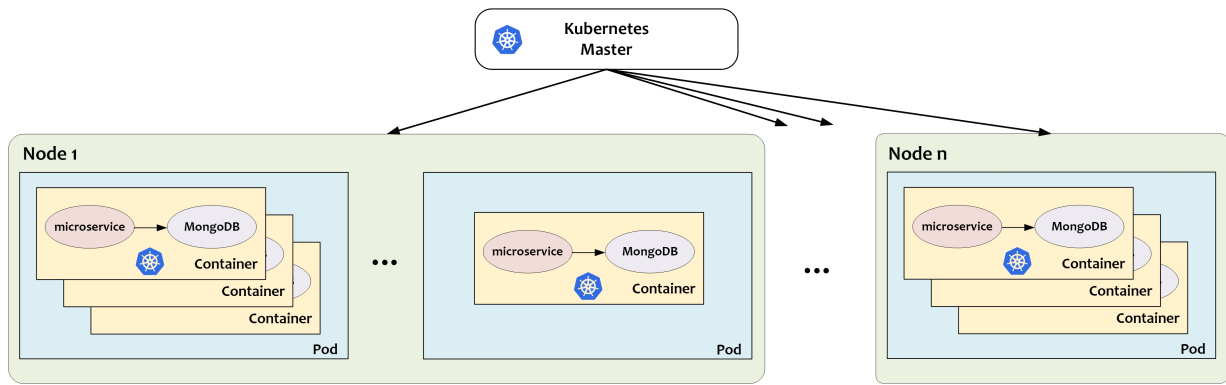


Figure 2.4: Key components of Kubernetes cluster architecture, including a master node and multiple worker nodes (Node 1, 2, ..., n) managing containerized workloads.

Docker simplifies the deployment process, enabling rapid scaling and improving overall operational efficiency.

While Docker simplifies the creation and deployment of individual containers, managing multiple containers at scale across distributed environments requires more advanced orchestration. Kubernetes (K8s) [Say18], an open-source container orchestration system, addresses this issue by automating the deployment, scaling, and management of containerized applications. Specifically, K8s abstracts the underlying infrastructure into a unified platform, which consists of two primary node types: master nodes and worker nodes [PMC21], as shown in Figure 2.4.

The master node governs the entire K8s cluster, managing essential tasks such as scheduling workloads, scaling applications, and monitoring the overall cluster state. It orchestrates container deployment across multiple worker nodes, ensuring the desired state of the application is maintained. Each worker node, also referred to as a *Node*, is a physical or virtual machine that executes containerized workloads.

When a user or automated system submits a deployment, the master node allocates pods to worker nodes based on resource availability and specified constraints. A *pod*, which represents the smallest deployable unit in K8s [Say18], typically contains one or more containers that share network and storage resources. Each worker node is tasked with running the allocated pods, ensuring that the containerized applications within them operate smoothly.

By decoupling microservices into containerized units and managing them at the Pod level,

K8s enables seamless scaling and efficient resource utilization across distributed systems. It not only simplifies container orchestration but also ensures resilience, self-healing, and efficient load-balancing, making it an essential tool for managing complex microservice architectures.

Serverless and FaaS

Serverless computing has become a widely adopted cloud-based application execution model [MB17], wherein developers do not need to manage underlying infrastructure such as virtual machines or containers. Instead, cloud providers handle server provisioning, scaling, and maintenance, allowing developers to focus exclusively on application logic. In this paradigm, resources are automatically allocated and scaled according to demand, and billing is based solely on the actual resources consumed, making serverless an attractive option for event-driven and cost-sensitive applications.

Within the serverless implementation, Function-as-a-Service (FaaS) plays a key role [UPK⁺21], which enables developers to deploy discrete units of code, called *functions*, that are executed in response to specific triggers or events, such as HTTP requests or file uploads. Unlike traditional microservices, which encapsulate larger services, FaaS allows for more granular execution [FJG20], where individual functions can be scaled and managed independently. This fine-grained approach maximizes resource efficiency, particularly for workloads with variable demand, and is especially advantageous for distributed systems. This is because, on one hand, functions can be deployed and executed across multiple nodes as needed. On the other hand, functions are instantiated only when necessary and terminated once execution is complete, further optimizing resource utilization.

To support such execution models, leading cloud providers such as Amazon Web Services (AWS Lambda)¹, Microsoft Azure (Azure Functions)², and Google Cloud (Google Cloud Functions)³ have developed comprehensive FaaS platforms. These services enable automatic scaling, monitoring, and billing based on the execution of functions rather than fixed infrastructure costs. In

¹<https://aws.amazon.com/lambda>

²<https://learn.microsoft.com/en-us/azure/azure-functions/>

³<https://cloud.google.com/functions>

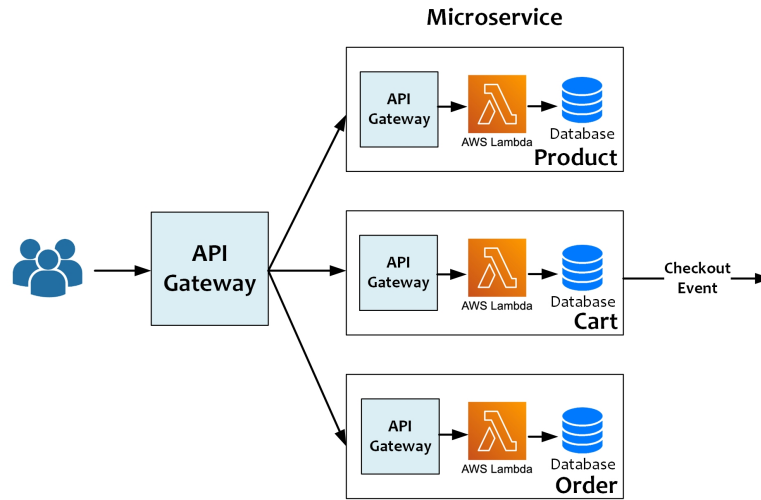


Figure 2.5: An example of serverless event-driven E-commerce microservices architecture

in addition to public cloud providers, open-source FaaS platforms, such as OpenFaaS⁴ and Apache OpenWhisk⁵, extend similar capabilities to both public and private cloud environments, making serverless computing accessible to a broader range of use cases.

Consequently, many microservice architectures are increasingly adopting serverless models to take advantage of the flexibility, scalability, and cost-efficiency of FaaS [LRC⁺18]. By leveraging FaaS, microservices can be further decoupled, allowing each service to scale independently and dynamically respond to demand. Figure 2.5 provides an example of this architecture, illustrating a serverless event-driven E-commerce microservices architecture supported by AWS Lambda. In this architecture, each microservice is supported by AWS Lambda, which handles specific tasks such as processing product data, managing user carts, and handling orders, providing high scalability, flexibility, and efficiency.

2.2.2 Related Work

The emerging microservice architecture has been increasingly adopted in modern application design and development due to its scalability, modularity, and flexibility. However, these benefits often introduce substantial performance overheads and resource management challenges. Consequently, addressing performance optimization has become a key focus in the design of

⁴<https://www.openfaas.com>

⁵<https://openwhisk.apache.org/>

microservice applications [BWB⁺19], particularly when deployed within distributed data processing systems [SJZY⁺16].

Many studies have explored the unique characteristics of microservice architectures, particularly emphasizing their decentralized and independent service structure. For instance, Heinrich *et al.* [HVHK⁺17] identify the limitations of traditional performance engineering techniques when applied to microservices due to their dynamic and distributed nature. They propose new research directions, including enhanced performance testing, continuous monitoring, and novel modeling approaches tailored to microservices. Similarly, Blinowski *et al.* [BOP22] conduct experiments across various deployment environments to compare the performance and scalability of monolithic and microservice architectures. Their findings reveal that microservices offer superior scalability, particularly in cloud environments. Al-Debagy *et al.* [ADM18] further extend this comparison by evaluating performance metrics such as throughput and response time, demonstrating that microservices excel in handling high concurrency levels, which highlights their scalability advantages over monolithic systems. In addition to scalability, Luo *et al.* [LXL⁺21] focus on microservice dependencies by analyzing over ten billion call traces from Alibaba clusters. They introduce a stochastic model to simulate dynamic dependencies, providing valuable insights into CPU and memory sensitivities that can optimize resource management in large-scale production environments. Ma *et al.* [MFC⁺19] contribute to the testing and validation of microservices with GSMART, a framework that leverages a graph-based service dependency model. The framework integrates behavior-driven development and word2vec for automated regression test selection, improving efficiency in complex microservice environments.

Building on the insights into microservice characteristics, numerous studies have focused on developing performance modeling techniques to address the unique challenges posed by microservices. Some research focuses on developing simulation-based tools and algorithms to optimize resource management. For instance, Jindal *et al.* [JPG19] tackle performance prediction and capacity planning for cloud-based microservices by introducing Terminus, a sandboxing tool that leverages regression models to predict individual microservice capacity, aiding in resource optimization. Sampaio *et al.* [SRBR19] propose REMaP, an adaptation mechanism that leverages

microservice affinities and resource usage history to optimize microservice placement employing heuristic algorithms and satisfiability modulo theory (SMT). Similarly, Bao *et al.* [BWB⁺19] develop a three-layer performance model for microservice-based cloud environments, which utilizes a heuristic scheduling algorithm based on a greedy recursive critical path method, focusing on optimizing workflow execution and scheduling.

Queueing-based models, a widely used subset of stochastic modeling, have been extensively applied to microservice performance analysis. Khazaei *et al.* [KMBL20] develop a queueing-theoretic model that captures both micro- and macro-level interactions, focusing on inter-service dependencies and communication patterns to predict performance metrics such as response time, availability, and scalability. Pinciroli *et al.* [PAT23] explore the performance impact of different design patterns in microservice systems by applying queuing networks (QNs) to model both microservice and system-level interactions, providing insights into how various design patterns influence performance.

Beyond queueing theory, other stochastic modeling approaches have also been applied to performance prediction in microservices. Pinheiro *et al.* [dSPPSM23] propose a framework based on stochastic Petri nets (SPNs) combined with random forest regression (RFR) for performance interference prediction and NSGA-II for multi-objective optimization, addressing resource allocation and real-time performance prediction in cloud-based microservice infrastructures. Khazaei *et al.* [KBBML16] present a continuous-time Markov chain (CTMC) model that captures the dynamic behavior of microservice provisioning in container and virtual machine (VM)-based cloud environments, evaluating metrics such as request rejection probability, response time, and cluster utilization. Gribaudo *et al.* [GIM18] employ Fluid Petri Nets to optimize performance, cost, and energy consumption in microservice-based architectures, with a focus on autoscaling strategies and interactions between microservices and infrastructure.

However, despite current advances in performance modeling for microservices, several critical research gaps remain. One major limitation is the lack of attention to data caching, which plays a crucial role in optimizing data access and minimizing latency in distributed microservice environments. The impact of caching on performance, especially in systems with complex data

flows, has been underexplored. Additionally, most current models do not adequately address the challenges introduced by batch arrivals, which can cause congestion and degrade performance, particularly in distributed systems. These gaps underscore the need for more comprehensive performance models to provide a more accurate and efficient analysis of microservice-based architectures.

To address these limitations, this thesis develops performance modeling techniques that capture key system behaviors overlooked in prior work. In particular, we propose a unified framework that models data caching and batch arrivals, which are critical for accuracy in distributed microservice environments. Unlike existing approaches based on queueing networks [PAT23, KMBL20], stochastic Petri nets [dSPPSM23], CTMCs [KBBML16], or fluid models [GIM18], which typically focus on isolated aspects such as communication patterns or autoscaling, our models integrate layered structures with caching-aware formulations and data-driven methods for highly dynamic workloads. This enhances both modeling granularity and applicability in real-world systems, as further discussed in Chapter 4.

2.3 Caching

Caching is a mechanism for storing copies of frequently accessed data, *e.g.*, images, files, or metadata, in memory nodes closer to consumers. By serving requests from nearby caches rather than remote origin sources, caching significantly improves system performance by reducing response time, backend load, and bandwidth consumption [MAN14]. A typical caching system comprises two core components: content items (also referred to as objects) and cache nodes.

The performance of a caching system is influenced by both content-level characteristics and cache-level configurations. On the content side, two key factors are object size and request popularity, which together determine the potential benefit of caching a given item. Frequently requested or large items, when cached effectively, contribute more to reducing system load and latency. In many real-world systems, item popularity has been observed to follow a *Zipf*

distribution, where the request probability of the i -th most popular item is given by:

$$P(i) = \frac{1/i^\alpha}{\sum_{j=1}^I 1/j^\alpha}, \quad \text{for } i = 1, \dots, I, \quad (2.1)$$

where I is the total number of items and α is the Zipf exponent. A larger value of α indicates that a few popular items dominate the request traffic, while a smaller α corresponds to a more uniform distribution over items. Empirical studies have found that Zipf-like distributions accurately model request patterns in various domains, including web traffic and content delivery networks [BCF⁺99].

On the cache side, important aspects include storage capacity, system architecture (*e.g.*, single-level or multi-level hierarchies), and the replacement policy that controls which items are evicted when space is limited. These dimensions collectively influence key performance metrics such as cache hit ratio, average response time, and backend traffic. Among these metrics, the *cache hit ratio* is widely used in both theoretical and empirical studies, as it captures the proportion of user requests that are successfully served from the cache. It is formally defined as:

$$p_{\text{hit}} = \frac{\text{number of cache hits}}{\text{total number of requests}}, \quad p_{\text{miss}} = 1 - p_{\text{hit}}. \quad (2.2)$$

A higher hit ratio typically indicates improved system efficiency by reducing access latency and backend load. The corresponding hit rate refers to the number of cache hits per unit time and is typically obtained by multiplying the hit ratio with the total request arrival rate.

Understanding how system-level configurations and access behaviors affect hit ratio and related metrics is a central concern in performance modeling. Prior studies have emphasized two aspects in this context: the cache replacement policy, which directly governs content retention and hit ratio [FRR12], and the architecture of system deployment, especially hierarchical caching structures [AFN16]. These two dimensions not only shape system behavior at different granularities but also introduce unique challenges in analytical modeling.

2.3.1 Caching Policies

Cache replacement policies determine which content to evict when a cache reaches its capacity. The aim is to retain items that are most likely to be requested again, thereby maximizing the hit ratio and improving performance. Common policies include:

- **Least Recently Used (LRU):** Evicts the item that has not been accessed for the longest time. LRU is widely adopted in practice due to its simplicity and effectiveness in exploiting temporal locality [FRR12]. Under the Independent Reference Model (IRM), the hit probability of item i under the LRU policy can be approximated by [CTW02]:

$$p_{\text{hit}}(i) = 1 - e^{-\lambda_i T}, \quad (2.3)$$

where λ_i is the request rate of item i , and T is the characteristic time representing the average time an item stays in the cache before eviction. The value of T is determined by the cache capacity C via the following equation:

$$\sum_{i=1}^I (1 - e^{-\lambda_i T}) = C. \quad (2.4)$$

This analytical result, known as Che's approximation [CTW02], provides a tractable estimate of LRU cache performance and is widely used in modeling studies.

- **Least Frequently Used (LFU):** Evicts items accessed the fewest number of times. LFU is suitable for workloads with stable popularity distributions but may be less adaptive to changing access patterns [JR04].
- **First-In First-Out (FIFO):** Evicts the oldest added item, irrespective of usage frequency. FIFO does not take item popularity or recency into account, which may limit its effectiveness under skewed access patterns [PB03].
- **Random Replacement (RR):** Randomly selects an item for eviction. RR is analytically tractable and serves as a useful baseline in theoretical models [GC22].

2.3.2 Hierarchical Caching Architecture

In distributed architectures, caches are frequently deployed in hierarchical structures across multiple tiers. In such multi-level caching schemes, lower-tier caches (closer to end users or services) serve local requests, while cache misses are forwarded to higher-tier caches or origin servers. This layered architecture allows cache nodes to collaborate, reducing redundancy and amplifying effective storage capacity [RSB01] and thus has been studied across various application domains. Rodriguez *et al.* [RSB01] provide a foundational comparison between hierarchical and flat architectures. Alouf *et al.* [AFN16] develop recursive models for DNS cache hierarchies, capturing per-level hit probabilities and routing behavior. More recent studies have focused on hierarchical caching in edge-based delivery systems. Wang *et al.* [WDH⁺21] propose a cost-driven caching model for edge content delivery networks, accounting for delay, popularity, and storage cost in multi-layer caching. Zeng *et al.* [ZZWW22] examine cooperative caching strategies in vehicular edge environments, modeling edge-cloud collaboration using an optimization framework to reduce network cost.

Several methodologies have been proposed to evaluate the caching performance of such systems, including simulation-based techniques [EE13] and analytical modeling using stochastic processes. Among these, analytical models such as those developed in [CG20] offer explicit performance metrics under formal assumptions, enabling tractable predictions across large parameter spaces. In particular, list-based caching policies such as FIFO and RR have been modeled using continuous-time Markov chains (CTMCs) under the Independent Reference Model (IRM), yielding closed-form expressions for hit and miss probabilities under controlled workloads. Compared to simulation, these models are less computationally intensive and offer structural insights.

For such analytical evaluation, the cache is represented as a set of lists connected in a tree structure, where each list $l = 1, 2, \dots, h$ has capacity m_l , and uncached items are represented by a virtual list $l = 0$ of size $m_0 = n - \sum_{l=1}^h m_l$. Here, n is the total number of distinct items in the system. The state vector $\mathbf{s} = [s(i, j)]$ captures the index of the item at position i in list

j , with each state corresponding to a valid list configuration. Item requests arrive as Poisson processes with rate $\lambda_{vk}(l)$ for stream v requesting item k in list l . Upon a hit in list l , item k may be moved to another list j with probability $c_{vk}(l, j)$. Replacement follows either the FIFO or RR policy, and list transitions respect a tree structure defined by a parent mapping $p(j)$, such that $p(p(\dots p(j))) = 0$.

The CTMC governing the replacement dynamics admits a product-form steady-state distribution [CG20]:

$$\pi(\mathbf{s}) = \frac{\prod_{j=0}^h \prod_{i=1}^{m_j} \gamma_{s(i,j)j}}{E(\mathbf{m})}, \quad (2.5)$$

where γ_{kj} is the access factor of item k to list j , defined recursively as:

$$\gamma_{kj} = \gamma_{kp(j)} \sum_{v=1}^u \lambda_{vk}(p(j)) c_{vk}(p(j), j), \quad (2.6)$$

and $E(\mathbf{m})$ is the normalizing constant. The marginal probability of item k residing in list l is:

$$\pi_{kl}(\mathbf{m}) = m_l \gamma_{kl} \frac{E_k(\mathbf{m} - 1_l)}{E(\mathbf{m})}, \quad (2.7)$$

where $E_k(\mathbf{m} - 1_l)$ is the normalizing constant when item k is removed from list l . From these probabilities, the miss ratio for item k is given by $\pi_{k0}(\mathbf{m}) = E_k(\mathbf{m})/E(\mathbf{m})$, and the miss rate for stream v requesting item k is:

$$M_{vk}(\mathbf{m}) = \lambda_{vk}(0) \pi_{k0}(\mathbf{m}) (1 - c_{vk}(0, 0)). \quad (2.8)$$

The total miss rate over all streams and items is:

$$M(\mathbf{m}) = \sum_{v,k} M_{vk}(\mathbf{m}). \quad (2.9)$$

These models provide a foundation for evaluating cache performance under hierarchical structures and non-uniform access patterns. However, existing formulations typically assume fixed

access paths and uniform propagation behavior. In this thesis, we extend these models to incorporate stream-specific routing constraints, configurable list interactions, and geographically-aware placement policies, enabling the analysis of more realistic distributed caching systems in microservice environments, as further discussed in Chapter 5.

2.4 Extended Queueing Networks

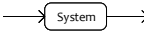
2.4.1 Queueing Networks

A queueing network consists of two elementary entities: jobs and service stations. Jobs, also known as customers or transactions, flow in and out of different service stations, constructing a connected network. It has manifold characteristics that determine the category of the queueing network. One characteristic is whether jobs are strictly retained in the network, which results in open and closed queueing networks, as presented in Table 2.1. Another characteristic is the class of the jobs, which differs in service time and routing probabilities. This leads to single-class and multi-class queueing networks, as shown in Table 2.2.

More than one service station, also known as resources or nodes, are contained in a queueing network to provide services requested by the jobs. Each single service station is composed of a queueing buffer and several servers, the common properties of which are summarised in Table 2.3 and Table 2.4. The service station is served as a queueing center when configuring finite servers. While it is served as a delay center without utilizing queueing buffers when configuring infinite servers, as there is no need for arrival jobs to wait in this case.

Based on such compositions, queueing networks are capable to represent real-life systems, such as web services and communication networks. This is because most practical systems adopt client-server architectures, similar to the queueing network models, in which the servers provide services consumed by the clients. Thus, queueing networks are widely used to model such systems and measure the performance analytically. Corresponding performance metrics include [LZGS84]:

Table 2.1: Open & Closed Queueing Networks

Queueing Network	Jobs			
	Characteristics	Population (number of jobs)	Workload	Workload Intensity
Open	can both enter and leave the network	varied	transaction workload 	λ - arrival rate
Closed	can neither enter nor leave the network	fixed	batch workload 	N - jobs
			terminal workload 	N - jobs Z - think time

- Utilization U , *i.e.*, the busy rate of resources, satisfying the Utilization Law $U = X \cdot S$, where X is the throughput and S is the mean service time.
- Throughput X , *i.e.*, the completion rate of jobs, satisfying Little's Law $N = X \cdot R$, where N is the average number of jobs in the system and R is the mean response time. The throughput is equal to the arrival rate when the system is in equilibrium.
- Mean response time R , *i.e.*, the average time spent in the system of each job. In closed queueing networks, it satisfies the Response Time Law $R = N/X - Z$, where Z is the think time. While in open queueing networks, there is no think time, and the response time is given by the standard Little's Law $R = N/X$.
- Mean queue length Q_i , *i.e.*, the average number of jobs at node i , including waiting or receiving service.

To obtain the performance metrics, the queueing network models can be solved by generating the underlying Continuous-time Markov Chain (CTMC). When the system is in equilibrium, the probabilities of each possible state of the CTMC, denoted as $\pi(\mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_M)$, can be leveraged to calculate all other performance measures. The primary method to compute the steady-state probabilities is utilizing the global balance equation, given by $\pi \mathbf{Q} = \mathbf{0}$, where π is the steady-state probability vector and $\mathbf{Q}$ is the infinitesimal generator matrix. This is, however, of high computational complexity, especially when applying to large networks. Therefore, there

Table 2.2: Single-class & Multi-class Queueing Networks

Queueing Network	Jobs	
	Single-class	Multi-class
Open	the overall arrival rate at node i $\lambda_i = \lambda_{oi} + \sum_{k=1}^M \lambda_k P_{ki}, i=1, \dots, M$	the arrival rate of class r at node i $\lambda_{ir} = \lambda_{o,ir} + \sum_{j=1}^R \sum_{k=1}^M \lambda_{kj} P_{kj,ir}$
Closed	number of jobs (constant) N	number of jobs (constant) $N = \sum_{i=1}^M \sum_{r=1}^R N_{ir}$ number of jobs of class r (not constant) $N_r = \sum_{i=1}^M N_{ir}$

Table 2.3: Common Properties of Queueing Buffers

Queueing Buffer		
Capacity	Quantity	Discipline
- Finite  - Infinite 	- Single list  - Multiple lists 	- FCFS (First Come First Served) - LCFS (Last Come First Served) - SIRO (Service In Random Order) - HOL (Head-of-Line FCFS priority) ...

are many alternative numerical techniques, the most common one of which is the product-form solution. A queueing network following particular rules, *e.g.*, Poisson arrival processes, exponentially distributed service time, and FCFS discipline, means that it satisfies local balance equations and has product-form solutions [BGDMT06]. Under the circumstances, the steady state probabilities of an open queueing network, according to Jackson's Theorem [BGDMT06], can be expressed as

$$\pi(\mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_M) = \pi_1(\mathbf{n}_1) \cdot \pi_2(\mathbf{n}_2) \cdots \pi_K(\mathbf{n}_M), \quad (2.10)$$

where $\pi_k(\mathbf{n}_k)$ is the marginal probability. Moreover, the steady state probabilities of a multi-class closed queueing network, according to the theorem of Gordon and Newell [BGDMT06], can be represented as

Table 2.4: Common Properties of Servers
Server

Quantity	Service Time Distribution
- Finite servers (Queueing Center)  - Infinite servers (Delay Center) 	- Deterministic distribution (D) - Exponential distribution (M) - Erlang distribution with k phases (E_k) - Hyperexponential distribution with k phases (H_k) ...

$$\begin{aligned}
 \pi(\mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_M) &= \frac{1}{G(N)} \prod_{i=1}^M P_i(\mathbf{n}_i) \\
 &= \frac{1}{G(N)} \prod_{i=1}^M n_i! \prod_{r=1}^R \frac{1}{n_{ir}!} \theta_{ir}^{n_{ir}},
 \end{aligned} \tag{2.11}$$

where θ_{ir} is the service demand of class r at node i . $G(N)$ is the normalizing constant, satisfying

$$G(N) = \sum_{\sum_{i=1}^M \mathbf{n}_i = N} \prod_{i=1}^M P_i(\mathbf{n}_i). \tag{2.12}$$

There exist multiple algorithms to calculate the $G(N)$, such as integral forms or recursive computations [Buz73, RK75, MM84, MM86, RW93, Cas17]. We illustrate here the integral forms sampling approach. Based on the gamma function, *i.e.*, $N! = \Gamma(N+1) = \int_0^{+\infty} x^N \cdot e^{-x} dx$, the equation (2.12) is equivalent to

$$G(N) = \sum_{\sum_{i=1}^M \mathbf{n}_i = N} \prod_{i=1}^M \int_0^{+\infty} x_i^{n_i} \cdot e^{-x_i} dx_i \prod_{r=1}^R \frac{1}{n_{ir}!} \theta_{ir}^{n_{ir}}. \tag{2.13}$$

As $\sum_{i=1}^M \sum_{r=1}^R n_{ir} = N$, $\sum_{i=1}^M n_{ir} = N_r$, the equation (2.13) can be transformed to the integral form [MM84]

$$G(N) = \prod_{r=1}^R \frac{1}{N_r!} \int_{x_1=0}^{+\infty} \cdots \int_{x_M=0}^{+\infty} e^{-\sum_i x_i} \prod_{r=1}^R (\theta_{1r} x_1 + \theta_{2r} x_2 + \cdots + \theta_{Mr} x_M)^{N_r} dx. \tag{2.14}$$

According to the multinomial theorem, the equation (2.14) is eventually expressed as a multi-

dimensional integral

$$G(N) = \prod_{r=1}^R \frac{1}{N_r!} \int_{\mathbf{x} \geq 0} e^{-\sum_i x_i} \prod_{r=1}^R \left(\sum_{i=1}^M \theta_{ir} x_i \right)^{N_r} dx. \quad (2.15)$$

By sampling points on the function in (2.15) and averaging, the normalizing constant $G(N)$ can be numerically computed [RW93].

2.4.2 Extended Queueing Networks Solution

Compared to standard product-form queueing networks [BCMP75], real-world applications often require extensions featured by more complex queueing networks, as described in [Cas20]. In these EQNs, jobs are probabilistically routed across a set of nodes, primarily queueing stations, where they receive service, possibly subject to contention by other jobs. Each job belongs to a *class*, *i.e.*, a type that defines its service, routing, and arrival characteristics at every node. The set of reachable classes for a job defines its *chain*. Typically, routing among the stations is probabilistic, and a job can *switch* class upon departing from a station, before reaching the next one, by traveling through specialized class-switching nodes.

Such EQNs include product-form queueing networks [BCMP75] and their extensions to model priorities, load-dependence, class-switching, and instantaneous transitions. A difficult problem is to obtain in such models station and system performance measures such as average queue lengths, utilization, response times, throughputs, and arrival rates. Several analytical and simulation-based methods have been proposed over the years to analyze EQN models of this kind. We briefly overview here the MVA, normalizing constant (NC), and Fluid paradigms. Throughout, we focus on models without caching and priorities, emphasizing instead analysis methods for networks with processor sharing (PS), FCFS, or infinite-server stations, which are the most common in LQN applications. In general, EQNs do not admit product-form solutions because their modeling features such as class switching and load dependence violate the assumptions required for product-form analysis.

Mean Value Analysis

The MVA paradigm is rooted in the arrival theorem, which offers a recurrence relation for mean response times in product-form queueing networks [RL80]. The subsequent development of the Linearizer [CN82] has led to many accurate approximation schemes for both product-form and EQN models. An overview of such methods, commonly referred to as approximate MVA algorithms, is available in [BGDMT06, §10]. Such methods are often fixed-point iterative approximations, which typically converge in milliseconds even on large-scale models. Due to this efficiency, LQN theory has been largely based on the approximate MVA theory [FAOW⁺08].

Normalizing Constant Methods

Probabilistic analyses typically require a different approach. The evaluation of marginal and joint distributions for the CTMC underpinning the queueing network normally requires to obtain the normalizing constant that ensures that state probabilities sum to one. A large number of normalizing constant (NC) methods, starting from the classic convolution algorithm [Buz73], are available. They include both recursive and asymptotic approximations, we point to [Cas17] for an overview. To our knowledge, NC methods have not been considered in past works on LQN analysis.

Mean-Field Approximation

We refer to the Fluid paradigm as the analysis style for queueing networks consisting of techniques derived from Kurtz’s mean-field approximation theory, see [BHL13] for a tutorial. Queueing network applications of these methods can scalably approximate large models, offering estimates for both transient and steady-state metrics [Tri13, PC17, RBÅC21]. However, such methods can face numerical difficulties in the presence of stiff ordinary equations as well as incur some inaccuracies in models with low parallelism levels. Recent works such as [GVH17, RBT22] are progressively reducing the severity of the last problem.

2.5 Layered Queueing Networks

Layered queueing networks (LQNs) are a rich class of extended queueing network models that has found application in software performance engineering, distributed system analysis, and modeling of stochastic networks, among other fields [FAOW⁺08, SD11, AVZ19]. Compared to ordinary queueing networks, the distinguishing feature of LQNs is the ability to model stochastic workflows and simultaneous resource possession. Due to their long history, a number of variants and specializations of these models have appeared in the literature, *e.g.*, networks with simultaneous resource possession [JL82], semi-open queueing networks [JH09], and stochastic rendezvous networks [WNPM95]. We here only introduce the essential background concepts on the LQN formalism and point the reader to [Woo13] for more details.

In an LQN, jobs issued by clients pay visits to a network of queueing resources following a workflow modeled as a directed acyclic graph (DAG). The DAG includes *calls* made by the job to *entries* exposed by *tasks* running on *host processors*. For example, an LQN model of a software system may use calls to describe service invocations, entries to describe service endpoints, tasks to describe software servers, and host processes to describe hardware resources. Informally, these concepts bear some resemblance to the standard queueing theoretic notions of routing (calls), classes (entries), and stations (tasks and host processors), as we discuss in more detail in Section 2.5.2, yet they enable the description of a more complex dynamics on layered network topology.

Upon invocation of an entry e through a call, a DAG-based workflow of one or more *activities* (*i.e.*, service phases) is executed until completion of the call. This workflow is called the *activity graph* bound to the entry e . Within this workflow, an activity may issue a synchronous call to an entry of another task while keeping a server in its parent task blocked, leading to simultaneous resource possession. Asynchronous calls and forwarding calls are also possible, we point to [Woo13] for additional details.

Figure 2.6 shows a small-scale example that summarizes the above definitions for the basic elements of LQN models: *tasks*, *host processors*, *entries*, and *activities*. Tasks are depicted

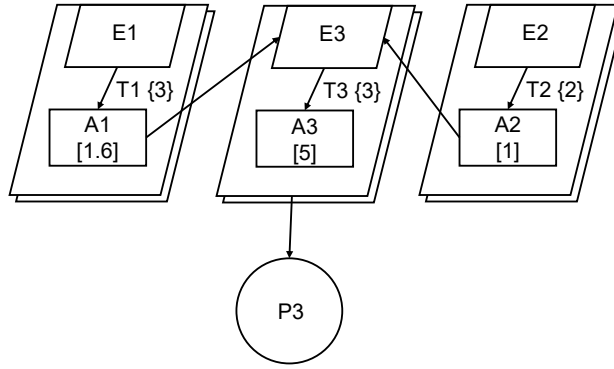


Figure 2.6: Example of an LQN model

as stacked parallelograms. A task runs on a single processor (*e.g.*, P3), represented by a circle. Specific services provided by a task are called entries and drawn as smaller parallelograms at the top of the task. Each rectangle denotes a particular activity performed during the execution of an entry. The number between square brackets specifies the service demand of the activity (*e.g.*, 1.6 for activity A1). Jobs in LQNs are generated by a special task termed the reference task, *e.g.*, tasks T1 and T2 in Figure 2.6 which model two classes of 3 and 2 users, respectively, both calling entry E3 exposed by task T3. The number between curly brackets represents the server multiplicity of the task, which is the maximum execution parallelism for a task, *e.g.*, T3 has a pool of 3 servers that can process calls to its entry E3. The processors of T1 and T2 are not drawn for conciseness. Larger models are typical in applications; we point to [RS95, FAOW⁺08, RCK⁺09] and the bibliography at [ann] for further examples.

Extensive prior work in the area has shown that an LQN can be accurately solved by iteratively evaluating a collection of ordinary mixed queueing networks, obtained using decomposition, until reaching a fixed-point iterative solution [FAOW⁺08]. Each decomposed submodel conceptually represents a layer of the system under study [RS95]. The collection of submodels forms an interactive ensemble, in the sense that the output parameters of a model for a lower layer, such as the response times at the queueing stations, become an input parameter of the upper layer models, typically service times at the queueing stations. We now briefly discuss the strategy to divide a given LQN into multiple layers and basic definitions concerning the obtained submodels that have been adopted in Chapter 3.

2.5.1 Layering Strategy

Layering strategies are methods for decomposing an LQN model into an ensemble of ordinary queueing networks. Prior work has noted a relatively weak sensitivity of the LQN solution accuracy based on the layering style [Fra00, §3.2]. In our proposed evaluation solver in Chapter 3, we adopt loose layering [Fra00, §3.2], which ensures that each layer includes a single queueing station, representing either a task or host processor, coupled with an infinite-server node to model the inter-request times of clients. The station may be replicated, with stations having identical demands.

For example, under loose layering decomposition, the model in Figure 2.6 yields four submodels: $T1 \rightarrow P1$, $T2 \rightarrow P2$, $T3 \rightarrow P3$, and $(T1, T2) \rightarrow T3$, where $\rightarrow$ indicates a client-server relationship. Note that each of these four models describes a single server in the LQN ($P1$, $P2$, $P3$, $T3$), thus corresponding submodels do not exist for the reference tasks ($T1$, $T2$), which are pure clients issuing calls to the system.

The rationale for choosing, unlike existing tools such as LQNS or DiffLQN, loose layering as the default strategy in our solver is that, while some queueing network algorithms such as MVA display linear complexity as a function of the number of stations m within a submodel, other paradigms rely on computations that have polynomial or exponential complexity with respect to m . While the asymptotic complexity of methods such as MVA is nearly unaffected⁶, if one solves m homogeneous models with a single type of station, rather than a monolithic model comprising m different queueing stations, methods of the latter kind are computationally more efficient in analyzing smaller models. For example, a CTMC solver may be fairly scalable in models with a single queue, but quickly incurs state space explosion with multiple stations. A similar issue is displayed by Monte Carlo sampling methods in the NC approach, which approximate a multidimensional integral in m dimensions and thus are quite sensitive to the growth of this parameter [WC13]. A drawback of loose layering is that certain features, such as heterogeneous load balancing or fork/join sections, are somewhat inconvenient to model, as the

⁶This applies to exact MVA methods. For approximate MVA methods, linear complexity holds at each iteration, but the number of iterations can be affected by m , although typically rather weakly.

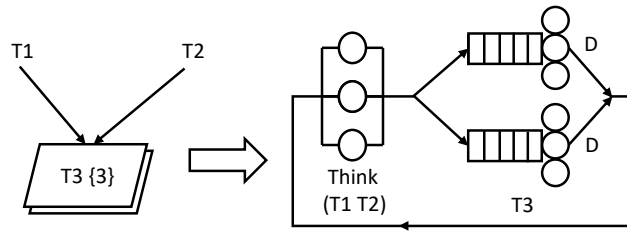


Figure 2.7: A layer of the LQN model with replication $m = 2$ and multiplicity $c = 3$.

participating stations may be scattered across different submodels, requiring specific solutions to cope with this difficulty⁷.

2.5.2 Submodel Generation

We now outline the mapping between LQN abstractions and the above models. Under loose layering, we abstract a layer l , which models a specific server in the LQN, as consisting of an EQN with m identical stations, modeling the server and its replicas, and a single infinite-server station, modeling the client think times. Thus, m denotes the replication factor of the server. We shall refer to such a queueing network as a *homogeneous* model. In more detail, the homogeneous models we consider are mixed queueing networks with m identical c -server queues and a delay (*i.e.*, infinite-server) node. Such networks can be mapped using standard techniques into closed queueing networks, after suitable correction of the service demands of the queueing stations [BGDMT06]. Jobs in class r have service demand D_r at the multiserver station and think time Z_r at the infinite-server station. We allow service distributions to be either phase type or degenerate distributions, in particular the *immediate* distribution, which models zero processing times. The handling of the latter typically relies in our solver on the stochastic complement technique that allows hiding selected transitions in a Markov chain [Mey89]; we point to [Cas09] for more details.

The number of servers in the queueing stations is equal to the multiplicity c of the server in

⁷For example, to address situations of this kind, if the response time of a station in layer l' needs to be evaluated within layer $l \leq l'$, it is possible to add a surrogate infinite server to layer l to represent such a station, iteratively synchronizing its think time with the response time observed in l' . Adding such a new infinite server station to layer l does not significantly change the submodel structure since these nodes can normally be aggregated with little approximation error.

that layer. Figure 2.7 gives the model of a layer for the LQN where T3 acts as a server. Clients issuing synchronous calls to layer l are represented as jobs initialized in a *reference class* at the delay. In the example of Figure 2.7, these may be classes T1 and T2. Subsequently, jobs cycle between the delay and the queues, switching their class to represent the specific tasks, entries, and activities that the clients visit (or invoke) during execution. For example, according to Figure 2.6, in the submodel for server T3, a job starting in reference class T1, will first switch to class E1, then to class A1, spending time in these classes at the delay station of the submodel. Then, the job will be routed to the server into a class labeled A1→E3, to distinguish it from similar calls to E3 issued by jobs starting in the other reference class T2, which is also present in the submodel. Calls issues from the latter would be in a class labeled A2→E3. Lastly, the job will complete at the station in class A1→E3 and return into class T1 at the delay.

Parameters such as think times and service demands are iteratively updated as per the method of layers [RS95]. For example, if certain sections of the client activity graph require access to another layer $l' \neq l$, the corresponding execution times are modeled as think times in the submodel for l that already incorporate the queueing contention in layer l' . Moreover, the probability of a client executing a particular entry is set proportional to the last throughput of this entry. We then update routing probabilities at each iteration, because not all solution paradigms are visit-based.

Clients that send asynchronous calls to layer l are instead represented as open Poisson arrival streams. The coexistence of open and closed classes, therefore, gives rise to mixed models, which can be reduced to closed ones by demand scaling [BGDMT06]. This allows us to analyze mixed models directly using solution paradigms for closed queueing networks.

Service classes are mapped to a set of R chains, obtained by computing the strongly connected components of the routing matrix. Each chain j represents a client task to the layer and has an associated number of jobs N_j that are initialized at the infinite-server station, starting in the chain reference class. Note that this does not lose information as it is possible to exactly recover the per-class performance metrics from the per-chain ones [BGDMT06, Zah79].

For ease of presentation, since a multi-chain model can always be reduced to a corresponding

multiclass model with R classes, one per chain, we shall use the terms “chain” and “class” interchangeably.

2.5.3 Performance Metrics

Performance metric computation is solver-dependent. We illustrate their definitions here taking the NC approach as a reference. We assume for simplicity that scheduling leads to product-form models and single server stations ($c = 1$). Moreover, we give definitions for cases where the station is a queueing resource, but similar definitions apply when this is instead an infinite server node. Approximations to handle other cases, such as multiclass FCFS scheduling, are discussed later.

Let $N = (N_1, \dots, N_R)$ be the population vector for a layer, $|N| = \sum_{r=1}^R N_r$, and recall that $G(m, N)$ is the normalizing constant of the state probabilities for the associated product-form model, which consists of m identical queueing stations and an infinite-server node. Denote by 1_r a row vector of all zeros with a one in the r -th dimension. We may exploit the following relations for the mean class- r throughput $X_r(N)$ and for the mean class- r queue length $Q_r(N)$ at any of the identical queueing stations:

$$X_r(N) = \frac{G(m, N - 1_r)}{G(m, N)}, \quad Q_r(N) = D_r \frac{G(m + 1, N - 1_r)}{G(m, N)}.$$

The system throughput $X_r(N)$ is assumed to be computed at a reference station, *i.e.*, the infinite server, for which we set the mean number of visits of class r to unity. Little’s law may then be combined with the previous relations to obtain other metrics at the queueing station, such as mean response times, mean residence times, and resource utilization [DB78, LZGS84]. Let V_r be the mean number of visits that a job in class r pays to the queueing station. This definition follows standard practice in closed queueing networks, where the number of visits is normalized relative to a designated reference station, allowing consistent derivation of throughput and residence time expressions [BGDMT06]. The class- r residence time $R_r(N)$ is the mean time accumulated by class r in visits to the queue prior to completing at the reference station,

i.e., $W_r(n) = Q_r/X_r$. The response time is then the average residence time per visit, *i.e.*, $W_r(N) = R_r(N)/V_r$. Lastly, the utilization is the percentage of time the processor is busy, given by $U_r(N) = X_r(N)D_r$. In the case of c -server stations, we assume $U_r(N)$ to be always re-scaled by c so as to range in $[0, 1]$, except if the station is an infinite server in which case we assume that $U_r(N) = Q_r(N)$, since $c \rightarrow \infty$.

Before discussing the novel algorithms proposed to compute these metrics, we remark that specific simplifications arise in evaluating normalizing constants for homogeneous models due to the structure of the product-form solutions. At first, if either $Z_r = 0 \wedge D_r = 0$ or $N_r = 0$ holds for a class r , then this class can be removed from the model as it does not contribute to the normalizing constant. Define $\mathcal{R}_D$ as the set of remaining classes for which $D_r = 0$. We note that the contribution of such classes to $G(m, N)$ is given exactly by a factor $\prod_{s \in \mathcal{R}_D} Z_s^{N_s}/N_s!$, which involves only elementary operations and can be efficiently computed in linear time [RL80]. Hence, every model can be reduced without loss of generality to one where all classes have $D_r > 0$, which we will assume throughout. This homogeneous structure not only simplifies the analytical formulation, but also aligns with real-world microservice architectures, where stateless service replicas are commonly deployed under symmetric configurations and accessed through uniform load balancing.

2.5.4 Fixed Point LQN Solution

Solutions of interactive models in an ensemble are reconciled through fixed-point iterations until performance metrics across the layers are consistent within a numerical tolerance. To this end, after topological sorting, the LQNS solver adopts an elevator algorithm whereby the graph that describes the client-server relationships is traversed top down and bottom up in alternation, thus cyclically inverting the order in which the layers are analyzed [Fra00]. This algorithm is also used in our LN solver to iterate over all the models within an ensemble.

Convergence testing in LN proceeds as follows. At each iteration, and for each submodel l , the largest change in per-class mean queue-length between iterations i and $i + 1$ is recorded as $\Delta_l^{i+1} = \max_r (Q_r^{(i)} - Q_r^{(i+1)})/N_r$, where $Q^{(i)}$ denotes the mean queue-length of class r obtained

at iteration i . If there are L submodels, then LN stops when $\Delta^{i+1} = \sum_{l=1}^L \Delta_l^{i+1}$ falls below a given tolerance. If this is not achieved within a maximum number of iterations, LN then assumes that input parameters are oscillating across iterations and applies a moving window to the residence time and through estimates at each layer to ease convergence. If this is not reached within a maximum number of iterations, the algorithm returns the current results.

2.6 Matrix Analytic Method

The Matrix Analytic Method (MAM) is a versatile analytical technique [He14] used to study a broad class of queueing models, particularly those featuring embedded Markov chains that extend beyond basic GI/M/1 and M/G/1 queues, representing more complex, two-dimensional generalizations. MAM leverages the structured nature of certain stochastic processes by representing their state transitions with matrices. This matrix representation allows for the systematic analysis and computation of steady-state probabilities and performance metrics. By decomposing complex state spaces into sub-blocks, MAM efficiently solves balance equations and derives key metrics through iterative algorithms and matrix operations.

In this section, we first explore the fundamental aspects of MAM, including Phase-Type (PH) distributions, Markovian Arrival Processes (MAP) and Batch Markovian Arrival Processes (BMAP). Following this, we demonstrate the applications of MAM in queueing modeling, including widely-used models such as MAP/PH/k and BMAP/PH/k queues, and discuss advanced methodologies for solving these models.

2.6.1 Phase-Type Distributions

Phase-type (PH) distributions [BGDMT06, He14] are an important class of probability distributions in stochastic modeling, extending the capabilities of the exponential distribution to encompass a wider variety of Markovian distributions. Defined as the time until absorption in a finite-state Markov process with a single absorbing state, PH distributions capture random

variables that describe a time-to-event through a series of exponential phases, each governed by its own rate parameter. This structure provides a detailed representation of complex system dynamics, making them indispensable to the MAM framework and enhancing its application across diverse stochastic systems. In this subsection, we introduce the most common PH distributions used in MAM, including exponential, Erlang, and hyper-exponential distributions.

Exponential Distribution

One of the most fundamental and widely studied continuous distributions is the exponential distribution. It is frequently used to model various stochastic processes due to its simplicity and the well-known memoryless property, which implies that the probability of an event occurring in the future is independent of all past events.

The exponential distribution can be represented within the MAM framework using the matrices $\mathbf{D}_0$ and $\mathbf{D}_1$. For an exponential distribution with rate parameter λ , the matrices are defined as follows:

$$\mathbf{D}_0 = [-\lambda], \quad \mathbf{D}_1 = [\lambda].$$

Here, $\mathbf{D}_0$ represents the rates of leaving states within the system, and $\mathbf{D}_1$ represents the rates of transitions leading to an absorbing state.

The cumulative distribution function (CDF) and the probability density function (PDF) of an exponentially distributed random variable X are given by:

$$F_X(x) = 1 - e^{-\lambda x}, \quad f_X(x) = \lambda e^{-\lambda x}, \quad \text{for } x \geq 0.$$

The mean $\mathbb{E}[X]$ and the variance $\text{Var}(X)$ of an exponentially distributed random variable X are given by:

$$\mathbb{E}[X] = \frac{1}{\lambda}, \quad \text{Var}(X) = \frac{1}{\lambda^2}.$$

The squared coefficient of variation (SCV) of an exponentially distributed random variable is 1, reflecting its constant coefficient of variation.

Erlang Distribution

The Erlang distribution is defined as the sum of k independent and identically distributed exponential random variables with a common rate parameter λ . This distribution, denoted as $E(\lambda, k)$, is particularly useful for modeling scenarios where the total time to an event consists of multiple, sequential phases, each with an exponential holding time.

The Erlang distribution can be represented within the MAM framework using the matrices $\mathbf{D}_0$ and $\mathbf{D}_1$. For an Erlang distribution with k phases and rate parameter λ , the matrices are defined as follows:

$$\mathbf{D}_0 = \begin{pmatrix} -\lambda & \lambda & 0 & \cdots & 0 \\ 0 & -\lambda & \lambda & \cdots & 0 \\ 0 & 0 & -\lambda & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -\lambda \end{pmatrix}, \quad \mathbf{D}_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ \lambda \end{pmatrix}.$$

Here, $\mathbf{D}_0$ captures the rates of leaving states within the system, with each state transitioning to the next at rate λ until the final state, while $\mathbf{D}_1$ represents the transition to the absorbing state at rate λ from the last phase.

The PDF and CDF of an Erlang distributed random variable X are given by:

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

The corresponding mean $\mathbb{E}[X]$ and the variance $\text{Var}(X)$ are given by:

$$\mathbb{E}[X] = \frac{k}{\lambda}, \quad \text{Var}(X) = \frac{k}{\lambda^2}.$$

The SCV of an Erlang distributed random variable is $\frac{1}{k}$, indicating lower variability as the number of phases k increases.

Hyper-Exponential Distribution

The hyper-exponential distribution is a finite mixture of k exponential distributions with different rate parameters λ_i , $i = 1, \dots, k$. It is characterized by its flexibility in modeling scenarios with high variability.

The hyper-exponential distribution can be represented within the MAM framework using the matrices $\mathbf{D}_0$ and $\mathbf{D}_1$. For a hyper-exponential distribution with k phases, the matrices are defined as follows:

$$\mathbf{D}_0 = \begin{pmatrix} -\lambda_1 & 0 & \cdots & 0 \\ 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -\lambda_k \end{pmatrix}, \quad \mathbf{D}_1 = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_k \end{pmatrix}.$$

Here, $\mathbf{D}_0$ captures the rates of leaving the states, each with its own rate parameter λ_i , while $\mathbf{D}_1$ represents the transitions leading to the absorbing state.

The PDF of a hyper-exponential distributed random variable X is given by:

$$f_X(x) = \sum_{i=1}^k q_i \lambda_i e^{-\lambda_i x}, \quad \text{for } x \geq 0,$$

where q_i are the mixing proportions satisfying $\sum_{i=1}^k q_i = 1$. The CDF of X is:

$$F_X(x) = 1 - \sum_{i=1}^k q_i e^{-\lambda_i x}, \quad \text{for } x \geq 0.$$

The mean $\mathbb{E}[X]$ and the variance $\text{Var}(X)$ of a hyper-exponential distributed random variable X are given by:

$$\mathbb{E}[X] = \sum_{i=1}^k \frac{q_i}{\lambda_i}, \quad \text{Var}(X) = \sum_{i=1}^k \frac{q_i}{\lambda_i^2} - \left(\sum_{i=1}^k \frac{q_i}{\lambda_i} \right)^2.$$

The SCV of the hyper-exponential distribution is always greater than one if $k > 1$, indicating higher variability compared to the exponential distribution.

2.6.2 Markovian Arrival Process (MAP)

MAP generalizes PH distributions to model more complex patterns, particularly those exhibiting correlations within time series data such as successive inter-arrival times [Asm00]. A MAP is defined by a pair of matrices, $\mathbf{D}_0$ and $\mathbf{D}_1$. The matrix $\mathbf{D}_0$ governs the rates of hidden transitions between transient states, with all off-diagonal entries being non-negative. The matrix $\mathbf{D}_1$ provides the rates of transitions that generate observable events, leading to the emission of a sample. The sum $\mathbf{Q} = \mathbf{D}_0 + \mathbf{D}_1$ forms an infinitesimal generator matrix that describes the overall transitions among the transient states.

Consider a MAP with $J = 2$ states as an illustrative example:

$$\mathbf{D}_0 = \begin{pmatrix} -\lambda_{11} & p_{12}^h \lambda_{11} \\ p_{21}^h \lambda_{22} & -\lambda_{22} \end{pmatrix}, \quad \mathbf{D}_1 = \begin{pmatrix} 0 & p_{12}^o \lambda_{11} \\ p_{21}^o \lambda_{22} & 0 \end{pmatrix}.$$

Here, λ_{ii} represents the rate of departing from state i , p_{ij}^h denotes the probability of a hidden transition from state i to state j , and p_{ij}^o denotes the probability of an observable transition from state i to state j .

For a sample s generated by a stationary MAP, the CDF is given by:

$$F(X) = 1 - \boldsymbol{\pi} \exp(\mathbf{D}_0 X) \mathbf{1},$$

where $\boldsymbol{\pi}$ is the left eigenvector of $\mathbf{P} = (-\mathbf{D}_0)^{-1} \mathbf{D}_1$, such that $\boldsymbol{\pi} \mathbf{P} = \boldsymbol{\pi}$, $\boldsymbol{\pi} \mathbf{1} = 1$, and $\mathbf{1}$ is a column vector of ones. The moments of the distribution can be expressed as:

$$\mathbb{E}[X^k] = k! \cdot \boldsymbol{\pi} (-\mathbf{D}_0)^{-k} \mathbf{1}, \quad k \geq 1.$$

The autocorrelation function (ACF) at lag k for a MAP is given by:

$$\rho_k = \frac{\mathbb{E}[X_0 X_k] - \mathbb{E}[X]^2}{\mathbb{E}[X^2] - \mathbb{E}[X]^2},$$

where $\mathbb{E}[X_0 X_k] = \boldsymbol{\pi} (-\mathbf{D}_0)^{-1} \mathbf{P}^k (-\mathbf{D}_0)^{-1} \mathbf{1}$ is the joint moments of the sample distribution. X_0 and X_k are samples that are $k \geq 1$ lags apart. If $\rho_k = 0$ for all k , the samples exhibit no correlations.

Markov Modulated Poisson Process (MMPP)

The Markov Modulated Poisson Process (MMPP) is a specific case of a MAP where the matrix $\mathbf{D}_1$ has non-zero entries exclusively on its diagonal [Asm00]. This configuration allows for the modeling of varying arrival rates with respect to the Markovian state, providing a more tractable yet flexible framework compared to general MAPs.

An example of MMPP considering $J = 2$ states is characterized by the following matrices:

$$\mathbf{D}_0 = \begin{pmatrix} -q_{12} - \lambda_1 & q_{12} \\ q_{21} & -q_{21} - \lambda_2 \end{pmatrix}, \quad \mathbf{D}_1 = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}.$$

Here, q_{ij} denotes the rate of transitioning from phase i to phase j , and λ_i is the arrival rate when the system is in phase i . The matrix $\mathbf{Q} = \mathbf{D}_0 + \mathbf{D}_1$ is the infinitesimal generator of the underlying Markov process, with each row summing to zero.

Batch Markovian Arrival Process (BMAP)

The Batch Markovian Arrival Process (BMAP) generalizes MAP to allow for simultaneous batch absorptions in the underlying Markov chain [Luc93]. A BMAP is characterized by a set of matrices $\{\mathbf{D}_0, \mathbf{D}_1, \dots, \mathbf{D}_b\}$, where $\mathbf{D}_0$ describes the transition rates between transient states when no batches are being processed. $\mathbf{D}_i$ represents the transition rates corresponding to batch arrivals of size i . The infinitesimal generator matrix $\mathbf{Q}$ is given by: $\mathbf{Q} = \mathbf{D}_0 + \mathbf{D}_1 + \mathbf{D}_2 + \dots + \mathbf{D}_b$.

Consider an example of a BMAP with $J = 3$ states and batch sizes ranging from 1 to 2. The matrices are given by:

$$\mathbf{D}_0 = \begin{pmatrix} -\lambda_1 & \lambda_1 p_{12}^{(0)} & \lambda_1 p_{13}^{(0)} \\ \lambda_2 p_{21}^{(0)} & -\lambda_2 & \lambda_2 p_{23}^{(0)} \\ \lambda_3 p_{31}^{(0)} & \lambda_3 p_{32}^{(0)} & -\lambda_3 \end{pmatrix}, \quad \mathbf{D}_1 = \begin{pmatrix} 0 & \lambda_1 p_{12}^{(1)} & 0 \\ 0 & 0 & \lambda_2 p_{23}^{(1)} \\ 0 & 0 & 0 \end{pmatrix}, \quad \mathbf{D}_2 = \begin{pmatrix} 0 & 0 & \lambda_1 p_{13}^{(2)} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Here, λ_i denotes the total rate of leaving state i , and $p_{ij}^{(k)}$ is the probability of a transition from state i to state j accompanied by a batch of size k , for $k = 0, 1, 2$.

2.6.3 Applications of MAM in Queueing Modeling

MAMs are powerful analytical methods used for queueing systems, especially when analyzed through their embedded Markov chains [RS02a]. These chains often exhibit a repetitive structure, which MAM efficiently handles using matrix-based techniques. By leveraging this repetitive nature, MAM enables precise computation of performance metrics and provides valuable insights into the system behavior. In this subsection, we first introduce widely used queueing models and subsequently present an advanced solution methodology known as ETAQA [RS02a, RS07].

Quasi-Birth-Death Processes

A Quasi-Birth-Death (QBD) process is a specialized Markov chain characterized by a structured transition matrix with distinct block patterns [BGDMT06]. The state space can be divided into levels, and each level l has n_l phases. In a QBD process, transitions are allowed only to the neighboring levels or within the same level, thus the transition matrix $\mathbf{Q}$ for a QBD process is organized as follows:

$$\mathbf{Q}_{QBD} = \begin{bmatrix} \mathbf{L} & \mathbf{F} & 0 & 0 & \dots \\ \mathbf{B} & \mathbf{L} & \mathbf{F} & 0 & \dots \\ 0 & \mathbf{B} & \mathbf{L} & \mathbf{F} & \dots \\ 0 & 0 & \mathbf{B} & \mathbf{L} & \ddots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix},$$

where $\mathbf{B}, \mathbf{L}, \mathbf{F}$ denote backward, local, and forward transition rates respectively.

M/G/1-type processes

If the arrivals can occur in bulk, such as in a MAP/PH/k or BMAP/PH/k queue, we define the embedded Markov chain as an M/G/1-type process [RS02a, RS07]. Their infinitesimal generator matrix can be presented as:

$$\mathbf{Q}_{M/G/1} = \begin{bmatrix} \mathbf{L} & \mathbf{F}^{(1)} & \mathbf{F}^{(2)} & \mathbf{F}^{(3)} & \dots \\ \mathbf{B} & \mathbf{L} & \mathbf{F}^{(1)} & \mathbf{F}^{(2)} & \dots \\ 0 & \mathbf{B} & \mathbf{L} & \mathbf{F}^{(1)} & \dots \\ 0 & 0 & \mathbf{B} & \mathbf{L} & \ddots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix},$$

where $\mathbf{Q}_{M/G/1}$ is an upper Hessenberg matrix.

ETAQA Methodology

In MAMs for M/G/1-type processes, the fundamental equation $\boldsymbol{\pi} \cdot \mathbf{Q}_{\text{M/G/1-type}} = \mathbf{0}$ is solved to determine the stationary probability vector $\boldsymbol{\pi}$. A crucial step in these methods is the computation of the auxiliary matrix $\mathbf{G}$, which captures the conditional probabilities associated with state transitions in the Markov chain. Various approaches have been proposed to solve $\mathbf{G}$, such as fixed-point iterations [FM98, Rhe10, BLM22], Newton's iteration methods [PTVH12], cyclic reduction algorithm [BMR00], and conditioning analysis [MSK18]. After obtaining $\mathbf{G}$, the stationary probability vector $\boldsymbol{\pi}$ is further determined via recursive relations involving $\mathbf{G}$, such as the well-established Ramaswami's recursive formula [Ram88] or by using aggregation-based methods like ETAQA [RS02b] with lower time and space complexity.

ETAQA [RS02a, RS07], which stands for *Efficient Technique for Analyzing QBD Processes by Aggregation*, is a method proposed for solving and computing various measures of interest for GI/M/1, M/G/1-type processes, and their intersection, *i.e.*, QBD processes. Unlike traditional techniques, ETAQA assumes that the state space S is partitioned into subsets $S^{(j)}$ and calculates the aggregate probability distribution for n classes of states. This approach transforms the problem into solving a finite system of $m + 2n$ linear equations, where m represents the number of states in the boundary portion of the process, and n denotes the number of states in each of the repetitive levels.

Specifically, for M/G/1-type processes, the block-partitioned infinitesimal generator $\mathbf{Q}_{\text{M/G/1}}$ defines a block partitioning of the stationary probability vector $\boldsymbol{\pi}$ as $\boldsymbol{\pi} = [\boldsymbol{\pi}^{(0)}, \boldsymbol{\pi}^{(1)}, \boldsymbol{\pi}^{(2)}, \dots]$, where each $\boldsymbol{\pi}_i$ corresponds to the probability distribution over the states in level i . This vector satisfies the equation $\boldsymbol{\pi} \mathbf{Q}_{\text{M/G/1}} = 0$. Given $\boldsymbol{\pi}$ and $\mathbf{Q}_{\text{M/G/1}}$, the system of linear equations $\mathbf{x} \mathbf{X} = [1, 0]$ admits a unique solution $\mathbf{x} = [\boldsymbol{\pi}^{(0)}, \boldsymbol{\pi}^{(1)}, \boldsymbol{\pi}^{(*)}]$, where $\boldsymbol{\pi}^{(*)}$ is the aggregate probability distribution for all levels $i \geq 2$, given by the sum $\boldsymbol{\pi}^{(*)} = \sum_{i \geq 2} \boldsymbol{\pi}^{(i)}$.

The key measure of interest that can be computed using ETAQA is the expected reward rate as:

$$r = \boldsymbol{\pi}^{(0)} \boldsymbol{\rho}^{(0)T} + \boldsymbol{\pi}^{(1)} \boldsymbol{\rho}^{(1)T} + \sum_{j=2}^{\infty} \boldsymbol{\pi}^{(j)} \boldsymbol{\rho}^{(j)T},$$

where $\boldsymbol{\rho}^{(j)}$ represents the reward vector associated with the states at level j of the system. In this context, the reward vector $\boldsymbol{\rho}^{(j)}$ assigns a value to each Markov state in level j that reflects the metric of interest, such as queue length, response time, or resource utilization, enabling a unified formulation of steady-state performance measures.

2.7 Summary

In this chapter, we have presented the essential background for the contributions in Chapters 3-6. First, in Section 2.2, we introduced key cloud-native concepts, including microservices, containers (Docker, Kubernetes), and serverless architectures (FaaS), alongside recent studies relevant to our research problems. Then, Sections 2.3, 2.4 and 2.5 laid the foundation for the performance models discussed in 3-5, focusing on caching, EQNs and LQNs. Finally, in Section 2.6, we demonstrated the MAM, which plays a critical role in the performance modeling techniques developed in Chapter 6.

Chapter 3

Algorithmic Framework for Layered Queueing Network Analysis

3.1 Introduction

Layering is a widespread feature of computer and communication networks, which can be analyzed using stochastic models such as layered queueing networks (LQNs) [WNPM95,FAOW⁺08]. LQNs build upon extended queueing network models to enable the analysis of simultaneous resource possession and stochastic call graphs in distributed systems. The underpinning stochastic models are typically analyzed by decomposition methods, which map the original LQN into an ensemble of interacting submodels, each being an extended mixed queueing network featuring open and closed job classes [Fra00]. The submodels can then be solved iteratively, updating their input parameters across the iterations until the performance metric estimates converge.

Several state-of-the-art solvers specific to LQNs have been proposed over the years, including for example LQNS [FAOW⁺08] and DiffLQN [WT16, Tri13]. LQNS is an established solver with many applications to real-world software engineering case studies. At heart, the tool applies to the LQN layers approximate mean-value analysis for extended queueing network models [FAOW⁺08]. The toolchain also offers LQN analysis based on generalized stochastic Petri nets via its `petri-rsvn` tool, which relies on the GreatSPN [ABB⁺16] tool, and via discrete-

event simulation, through the `lqsim` tool. We point to [FAOW⁺08] for a comparative analysis of LQNS with earlier tools.

DiffLQN is instead a solver that is based on the mean-field approximation theory developed in the context of Performance Evaluation Process Algebra (PEPA) models to scalably analyze LQNs [WT16, Tri13]. The mean-field fluid paradigm is particularly suited to the solution of large models, as it becomes asymptotically exact in layers with multiserver FCFS stations once the number of jobs and servers grows large in a fixed ratio. Subsequent work on mean-field approximations has further generalized the fluid solutions to processor sharing (PS) stations, class switching, random environments, response time percentiles [PC17], differentiated service weights [ZCP20], multiclass FCFS approximations [Cas20], and mixed models [RBÅC21].

The current research on LQNs emphasizes only a few, mainly MVA methods [FAOW⁺08] and, more recently, mean-field approximations [Tri13, PC17]. However, many performance evaluation paradigms have been proposed for evaluating the mixed queueing networks associated with each LQN submodel, ranging from stochastic simulation (SSA), continuous-time Markov chain analysis (CTMC), normalizing constant evaluation (NC), matrix analytic methods (MAM), mean-field approximations (Fluid), and mean-value analysis (MVA) [Cas17]. We argue that the availability of multiple solution paradigms motivates the development of a richer approach to solving LQNs. In particular, we show numerical evidence that adding the flexibility of dynamically selecting the most appropriate algorithm to analyze each submodel arising from LQN decomposition helps in reducing the maximum LQN solution error compared to using a single submodel evaluation paradigm.

To build upon this observation, we present LN, an open-source meta-solver framework for LQN analysis. We define a layered network meta-solver as a framework that builds upon a collection of independent solvers to analyze each network layer in isolation. A key goal of a meta-solver is to maintain abstraction in the interaction with these solvers. This enables greater generality, extensibility, and fewer dependence on solver-specific concepts and methods than in existing LQN solvers. LN builds its meta-solver capability on top of a suite of ordinary queueing network solvers implemented in the LINE software package, which offers several tens

of exact and approximate solution algorithms for extended queueing networks [Cas20]. In LN, the submodels passed to LINE solvers are generated using loose layering [Fra00, §3.2] and thus consist of a single infinite-server node and m replicated queueing stations, *i.e.*, stations having identical parameterization in every class of jobs. We refer to such models as *homogeneous* queueing networks. For such models, we develop specialized methods that are more accurate, or more efficient, than existing solution techniques for ordinary queueing networks, providing a case for systematically adopting in LN the loose layering decomposition style. While the homogeneous structure limits direct applicability to submodels with symmetric service stations, this assumption is realistic in many layered architectures with replicated resources, making it a practical and widely applicable modeling choice.

Our main contributions to the theory of homogeneous queueing networks are threefold. First, we propose a method of moments algorithm for solving *exactly* homogeneous models in linear time as the total population size grows. This is first developed for single server queueing stations. An extension to the multiserver case is also introduced, which can quickly solve large multiclass models with significant speedups compared to MVA. For example, we report on the ability of these algorithms to solve exactly multiclass models with thousands of jobs in milliseconds and without requiring exact algebra implementations, which is a crucial gain compared to the existing method of moment algorithms [Cas11, Cas09].

Next, we develop Gaussian quadrature methods for approximating the normalizing constants of homogeneous models. By controlling the order of the quadrature, these methods can trade accuracy for speed while remaining efficient in models with large population sizes. Moreover, integral expressions are more straightforward to use than recurrence relations on some models, such as those with fractional populations.

Lastly, we introduce a non-iterative MVA-based approximation for homogeneous models, which allows for a fast model solution in return for slightly degraded accuracy in the submodel evaluation. Although not as accurate as iterative methods, this new technique is significantly faster and offers much greater accuracy than comparable non-iterative approximations, such as asymptotic bounds [DB78, LZGS84], which are sometimes used to initialize LQN submodels.

The rest of the chapter is organized as follows. Section 3.2 gives motivation for meta-solution approaches in LQNs. Section 3.3 introduces novel solution algorithms for homogeneous queueing networks. Experiments and case studies are presented in Section 3.4. Finally, Section 3.5 is dedicated to the conclusions.

3.2 LQN Meta-Solution

3.2.1 Motivating Example

To compare different solution methods for solving closed queueing network models with varying characteristics, we investigate a set of 20,000 random models. Each model consists of a delay and a queue, with the scheduling strategy for each queue being randomly chosen as either FCFS or PS. The service time distribution for each job class is chosen uniformly at random to be either exponential, hyper-exponential, or Erlang. Note that the latter distributions admit product-form solution under PS but not under FCFS except for the case where exponential service times are identical [BCMP75], which occurs in our random models only in single-class networks. The models are designed to handle low, medium, and high loads, with the total job population ranging in $[1,10]$ (low), $[11,30]$ (medium), and $[31,40]$ (high), respectively.

Each model is solved using the MVA, NC, and Fluid solvers available in LINE¹ 2.0.27. To determine the optimal solution for each model, we measure the performance of each method using a combined metric that considers both accuracy and execution time, computed through a weighted geometric mean of the two factors with coefficients 0.75 and 0.25, respectively. Specifically, the error metric is the arithmetic mean of the absolute relative errors on the system response time and the system throughput, computed with respect to discrete-event simulation values. We use for the simulation the JSIMgraph tool, part of the Java Modeling Tool (JMT) suite [BCS07]. The solution method that exhibits the maximal accuracy metric is selected. The resulting distribution of the selected solution paradigm across all 20,000 models

¹Version available at: <https://github.com/imperial-qore/line-solver>

Table 3.1: Solver selection statistics for 20,000 randomly generated homogeneous models with FCFS and PS stations

<i>Solver</i>	<i>Server multiplicity</i>			<i>Class populations</i>			<i>Total percentage</i>
	<i>Low</i>	<i>Mid</i>	<i>High</i>	<i>Low</i>	<i>Mid</i>	<i>High</i>	
MVA	26.625	1.895	1.210	20.500	6.735	2.495	29.730
NC	35.655	22.830	11.580	45.190	18.465	6.410	70.065
Fluid	0.125	0.055	0.025	0.200	0.005	0.000	0.205

is illustrated in Table 3.1, which reports, for each solver, the percentage of models (out of 20,000) for which it achieves the lowest combined performance score, across different categories of server multiplicity and class population.

To enable finer-grained analysis, the models are grouped by server multiplicity and class population. Server multiplicity levels are categorized as low (1-2 servers), mid (3-5), and high (6 or more), while class population levels are defined as low (1-10 jobs), mid (11-30), and high (31-40). Solver selection is based on a composite score that combines accuracy and runtime, computed as a weighted geometric mean with weights 0.75 and 0.25, respectively. This choice reflects the primary focus on modeling accuracy, while ensuring that computational cost is also taken into account.

The results shown in Table 3.1 reveal that the majority of models in LINE are best solved by the NC and MVA methods, with Fluid being the best solver only in approximately 0.2% of models. Specifically, Table 3.1 indicates that the NC and MVA methods both exhibit comparable performance for small-scale models, with a difference of 9.03%. However, for medium-scale and large-scale models, the NC solver performs better than the MVA method. On the other hand, Table 3.1 demonstrates that the NC method outperforms the MVA method for models serving either low, medium, or high workloads, with an advantage of 24.69%, 11.73%, and 3.91%, respectively. The Fluid model is fairly accurate, but it is worse than MVA and NC in solution speed. This is because integrating the ODEs is fairly expensive, especially when the differential equations are stiff. For example, consider a cyclic closed queueing network model including a delay station with unit think times and two FCFS nodes. There are two classes of jobs with 10 jobs each. Class 1 has mean demand 1.5 at both queues, while class 2 has demand 3.0 at both stations. Averaging execution times over 100 repetitions, MVA and NC

complete in about 16 ms, while the Fluid solver takes about 42 ms and displays similar, but slightly lower, accuracy than MVA and NC, due to the fact that the model has single server stations and therefore does not perfectly align to the assumptions of Kurtz's theorem [PC17]. Nevertheless, the Fluid solver remains a useful tool for transient analysis and response time distribution evaluation, as illustrated later in Section 3.4.2.

It should be noted that the above conclusions hold for the specific implementations in the considered version of LINE. However, they illustrate the fact that no solver dominates the other, since each between MVA and NC can handle best a large fraction of models. This gives a case for developing automated methods for selecting the best algorithms to run within meta-solver approaches so as to leverage individual paradigm strengths.

In summary, the NC and MVA methods are generally better suited for solving most models, with the choice between the two methods depending on the expected load as well as the scale of the model. The Fluid method may be suitable for a small percentage of models, yet as we show later in Section 3.4, it still offers good average performance with low maximal error.

3.2.2 LN Meta-Solver

Stemming from the previous motivating example, the LINE Layered Network solver (LN) is a concrete implementation of a meta-solver approach to LQN analysis. All the solution paradigms available in LINE, including analytical methods (NC, MVA, MAM, and Fluid), discrete-event simulation (JMT, SSA), and CTMC analysis, can thus be leveraged to evaluate LQNs. Given an input LQN model, LN first produces an ensemble of submodels according to the principles discussed in Section 2.5.1. An initial assignment of performance metrics is made to each submodel assuming no queueing, after calculating essential quantities such as the visit ratios of client entries to tasks. Upper bounds to the multiplicity levels of the servers are also applied at this stage at infinite server stations to ensure that multiplicity coefficients are all finite. Then, a set of solvers is prepared for the analysis of the generated submodels.

After the initialization phase, LN starts to iteratively solve the ensemble of submodels, updating

Table 3.2: LN automated paradigm selection heuristics in LINE 2.0.27

<i>Paradigm</i>	<i>Selected if:</i>	<i>Main Methods</i>
NC	• Single class models • Multiclass models with single server queues • Small population models (including multiservers)	• CoMoM, Section 3.3.1 • Multiserver CoMoM, Section 3.3.2 • Gauss-Legendre, Section 3.3.3 • RD heuristic [CHO21]
MVA	• Multiclass models with multiserver queues • High or medium population models • Non-product-form models	• QD-AMVA [CPW15] • Linearizer [CHW75] • Non-iterative approximation, Section 3.3.4

the model parameters after each iteration. The order in which the submodels are processed is arbitrary, and a submodel may be skipped at a certain iteration if its parameter changes are negligible. In particular, the current implementation adopts the same ordering as in the LQN elevator algorithm [Fra00]. Each time the ensemble of submodels is analyzed and modified, a global update of the solver parameters ensues. The iterative solution procedure stops and returns the obtained results after detecting convergence of the performance metrics within a threshold or after exceeding an iteration limit.

Compared to existing solvers, such as LQNS, the main differences are the selection of the solver to be applied to each submodel and the range of algorithms within each solver. The process of selecting a solver for a submodel is either chosen manually by the end user or automated via heuristics. Table 3.2 illustrates the main heuristic selection criteria used in LN shipped with LINE version 2.0.27. Given that non-product-form models are difficult to characterize exactly, it is challenging to come up with universal rules to decide when a method is better than another; thus, the heuristic rules in the table summarize our empirical observations from the execution of the solvers on random instances. The conditions for selection are applied top-down; for example, a small population model will always be processed by the NC solver, also when it features multiclass multiserver stations.

Each solver offers a selection of several algorithms, a selection of the main methods is highlighted in the table. At present, LINE offers about 70 different algorithms across its solvers, each applicable to a specific family of submodels. An inner selection function is applied within each solver to choose which algorithm to run based on the characteristics of the model. Several algorithms are introduced in this chapter in the sections listed in the table. The logic for

selecting algorithms within each solver is hidden to the LN meta-solver, which interacts with the individual solvers only via abstract interfaces; we point to appendix A for more details on the specification of such interfaces. The selection rule of MVA prioritizes QD-AMVA on models where some classes have fractional populations with less than one job; otherwise, the Linearizer algorithm is chosen. Instead, the NC solver uses the Gauss-Legendre presented later in Section 3.3.3 as default and the Logistic Expansion proposed in [Cas17] for very large models. The CoMoM and multiserver CoMoM exact methods we propose later in Sections 3.3.1 and 3.3.2 can also be selected by the user for additional accuracy and the latter is the default for multiclass homogenous models with a multiserver queue.

In LINE MVA solver, Linearizer is applied by default to models with homogeneous single server queues. QD-AMVA is implemented similarly to [CPW15] and used by default to handle homogeneous multiserver PS queues, which are treated in this approximation as stations with load-dependent rates. QD-AMVA is also preferred to Linearizer to approximate models with very low population, including those with less than 1.0 job in some classes. This is because the QD-AMVA queue-length interpolation is class independent, therefore avoiding some inaccuracies that arise in this case when using Linearizer. Linearizer is instead used to approximate multiserver FCFS stations based on either Seidmann's approximation or the Rolia-Franks approximation [SSSO87, RS95, FAOW⁺08].

In LINE NC solver, the default methods are the algorithms presented in the next section, in particular, the Gauss-Legendre integration is used by default unless the model has multiserver stations in which case multiserver CoMoM is preferred. In addition to these, the RD method is used to handle the corrections required to handle load-dependent stations [CHO21].

3.3 Algorithms for Homogeneous Queueing Networks

From now on, we focus on LQN decomposition based on loose layering and present the algorithmic innovation developed to increase accuracy and performance of the solution algorithms for individual LQN layers.

3.3.1 CoMoM Algorithm for Homogeneous Models with Single Server Queues

Recall that mixed queueing network models can be mapped with suitable transformation to a model consisting only of closed classes [BGDMT06, §8.2.3]. A closed product-form network solution approach that demonstrates a specific advantage of relying on the NC paradigm instead than mean-value analysis is the Class-oriented Method of Moments (CoMoM) algorithm proposed in [Cas09] for exact analysis and several approximations derived from integral forms for the normalizing constant $G(m, N)$ that enable probabilistic analyses, *e.g.*, asymptotic expansions [MM84, Cas17]. We now describe the extensions developed for the two families of algorithms, corresponding to models with single server and multiserver queues respectively.

We now propose a method of moments algorithm that exactly solves homogeneous models in linear time as the total population size grows, assuming models with single server queueing stations. As opposed to existing methods such as MoM [Cas11] and CoMoM [Cas09] that can *theoretically* achieve linear complexity, the proposed technique is the first one that realizes this in concrete implementations by avoiding the use of exact arithmetic, which introduces overheads up to about log-linear in the total population size [Cas11, Cas09]. Moreover, it does so without solving systems of linear equations usually appearing in methods of moment algorithms. The result allows the efficient solution of submodels arising from loose layering in milliseconds. Also, it serves as a basis for efficient approximations for more complex networks with multiserver stations [SSSO87, CHO21] and for non-product-form models. We also discuss here an extension for the exact analysis of homogeneous models of multiserver nodes.

For a model with N jobs belonging to a fixed number of R classes, CoMoM implementations require approximately log-quadratic time and log-linear space in $|N|$ to obtain an *exact* solution, thus being more scalable than the exact MVA algorithms. The latter has a time and space complexity of $O(|N|^R)$. Moreover, contrary to other moment-based methods, CoMoM can avoid degeneracies when the model consists of one or more replicated queueing stations, as in the case of loose layering. An equivalent result is not currently available for MVA. Among

the complications, it is worth noting that MVA expressions such as the celebrated arrival theorem are bi-linear in their defining terms, mean queue lengths and mean throughputs, yielding systems of non-linear equations that are not as tractable as CoMoM linear matrix recurrence relation.

We enhance CoMoM by developing explicit solutions to its system of linear equations for homogeneous models. Such solutions apply to models with an arbitrary number of classes R . For a vector v with d dimensions, let $|v| = \sum_{i=1}^d v_i$ and define $\text{diag}(v)$ as a diagonal matrix with the elements of v placed on the main diagonal. We now give an exact result.

Theorem 1. *Consider a product-form queueing network model with R classes, having m identical single server queueing stations with service demand $D_r > 0$ in class r and an infinite-server node with think time Z_r in class r . Define the collection of normalizing constants*

$$g(m, N) = \begin{bmatrix} G(m, N) & G(m, N - 1_1) & \cdots & G(m, N - 1_{R-1}) \end{bmatrix},$$

and the basis

$$\Lambda(N) = \begin{bmatrix} g(m+1, N) & g(m, N) \end{bmatrix}^T.$$

Then the following matrix recurrence relation holds

$$\Lambda(N) = (\mathbf{F}_{1,R} + N_R^{-1} \mathbf{F}_{2,R}) \Lambda(N - 1_R), \quad (3.1)$$

in which

$$\mathbf{F}_{1,R} = \begin{bmatrix} D_R \mathbf{E}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad \mathbf{F}_{2,R} = \begin{bmatrix} m D_R \mathbf{S} & Z_R \mathbf{S} \\ m D_R \mathbf{I} & Z_R \mathbf{I} \end{bmatrix},$$

$$\mathbf{S} = -m^{-1} \begin{bmatrix} -|\tilde{N}| - m & \tilde{Z}^T \\ -\text{diag}(\tilde{D})^{-1} \tilde{N} & \text{diag}(\tilde{D})^{-1} \text{diag}(\tilde{Z}) \end{bmatrix},$$

where $\mathbf{E}_{1,1}$ is of order R with a single nonzero entry in position $(1, 1)$, $\tilde{N} = (N_1, \dots, N_{R-1})^T$, $\tilde{D} = (D_1, \dots, D_{R-1})^T$, $\tilde{Z} = (Z_1, \dots, Z_{R-1})^T$, $\mathbf{I}$ is the identity matrix of order R , and $\mathbf{0}$ is the zero matrix of order R .

Proof. For a homogeneous model, the CoMoM recurrence relation may be written as $\mathbf{A}\Lambda(N) = \mathbf{B}\Lambda(N - 1_R)$, where

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{0} & \mathbf{A}_{2,2} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{0} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}.$$

Let $\mathbf{0}_{I,J}$ indicate a block of zeros of size $I \times J$. Defining $\tilde{D} = (D_1, \dots, D_R)^T$, we have

$$\mathbf{A}_{1,1} = \begin{bmatrix} 1 & -\tilde{D}^T \\ \mathbf{0}_{R-1,1} & -m \operatorname{diag}(\tilde{D}) \end{bmatrix}, \quad \mathbf{A}_{1,2} = \begin{bmatrix} -1 & \mathbf{0}_{1,R-1} \\ \tilde{N} & -\operatorname{diag}(\tilde{Z}) \end{bmatrix}, \quad \mathbf{A}_{2,2} = N_R \mathbf{I},$$

$$\mathbf{B}_{1,1} = D_R \mathbf{E}_{1,1}, \quad \mathbf{B}_{2,1} = m D_R \mathbf{I}, \quad \mathbf{B}_{2,2} = Z_R \mathbf{I}.$$

The inverse of the block upper triangular matrix $\mathbf{A}$ is now computed as

$$\mathbf{A}^{-1} = \begin{bmatrix} \mathbf{A}_{1,1}^{-1} & \mathbf{S} \mathbf{A}_{2,2}^{-1} \\ \mathbf{0}_{R-1,1} & \mathbf{A}_{2,2}^{-1} \end{bmatrix},$$

with $\mathbf{S} = -\mathbf{A}_{1,1}^{-1} \mathbf{A}_{1,2}$. Observe first that

$$\mathbf{A}_{1,1}^{-1} = \begin{bmatrix} 1 & -m^{-1} e^T \\ \mathbf{0}_{R-1,1} & -m^{-1} \operatorname{diag}(\tilde{D})^{-1} \end{bmatrix}, \quad \mathbf{A}_{2,2}^{-1} = N_R^{-1} \mathbf{I},$$

where $e^T = \tilde{D}^T \operatorname{diag}(\tilde{D})^{-1} = (1, \dots, 1)$. Thus

$$\begin{aligned} \mathbf{S} &= - \begin{bmatrix} 1 & -m^{-1} e^T \\ \mathbf{0}_{R-1,1} & -m^{-1} \operatorname{diag}(\tilde{D})^{-1} \end{bmatrix} \begin{bmatrix} -1 & \mathbf{0}_{1,R-1} \\ \tilde{N} & -\operatorname{diag}(\tilde{Z}) \end{bmatrix} \\ &= -m^{-1} \begin{bmatrix} -|\tilde{N}| - m & \tilde{Z}^T \\ -\operatorname{diag}(\tilde{D})^{-1} \tilde{N} & \operatorname{diag}(\tilde{D})^{-1} \operatorname{diag}(\tilde{Z}) \end{bmatrix}. \end{aligned}$$

Note that $\mathbf{A}_{2,2}$ is the only block that depends on N_R . We can therefore write

$$\mathbf{A}^{-1} \mathbf{B} = \mathbf{F}_{1,R} + N_R^{-1} \mathbf{F}_{2,R},$$

where

$$\mathbf{F}_{1,R} = \begin{bmatrix} \mathbf{A}_{1,1}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} D_R \mathbf{E}_{1,1} & \mathbf{0} \\ mD_R \mathbf{I} & Z_R \mathbf{I} \end{bmatrix} = \begin{bmatrix} D_R \mathbf{E}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix},$$

$$\mathbf{F}_{2,R} = \begin{bmatrix} \mathbf{0} & \mathbf{S} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} D_R \mathbf{E}_{1,1} & \mathbf{0} \\ mD_R \mathbf{I} & Z_R \mathbf{I} \end{bmatrix} = \begin{bmatrix} mD_R \mathbf{S} & Z_R \mathbf{S} \\ mD_R \mathbf{I} & Z_R \mathbf{I} \end{bmatrix}.$$

□

Note in particular that the knowledge of $\Lambda(N)$ and $\Lambda(N - 1_R)$ is sufficient to determine the expression of all the mean performance metrics introduced in Section 2.5.3. Termination conditions for (3.1) are obtained noting that $G(\cdot, 0) = 1$ and that, whenever any element of N is negative, $G(\cdot, N) = 0$.

In terms of numerical stability, to prevent floating-point range exceptions, the proposed solution can either be computed using exact or multi-precision arithmetic. In practice, we have observed that scaling at each step the vector $\Lambda(N)$ so that $|\Lambda(N)| = 1$, and removing the effect of such scaling only in the final result, is sufficient to sanitize numerical problems in practical uses, except for negligible numerical fluctuations. This makes the theoretical and implementation complexity identical and, empirically, much faster than using exact arithmetic.

Lastly, we remark that the result in this section assumes integer populations since for fractional values an exact CTMC solution is undefined. In this case, we solve the models with integer populations $\lceil N \rceil$ and $\lfloor N \rfloor$ and linearly interpolate the mean performance metrics. For example, the mean class- r queue-length would be estimated as $Q_r(N) = Q_r(\lfloor N \rfloor) + \eta(Q_r(\lceil N \rceil) - Q_r(\lfloor N \rfloor))$, with $\eta = N - \lfloor N \rfloor$. Similar expressions can be used for the other metrics.

Numerical Example

To illustrate the performance of the enhanced CoMoM algorithm, we consider a challenging model with $R = 8$ classes, where $m = 1$, $Z_r = r$ and $D_r = 10^{-r}$, $r = 1, \dots, R$. Jobs are split equally across the classes. The exponential spacing of the demands and the large population make the analysis numerically challenging. We set a timeout of 10 seconds to solve a model. If the timeout is exceeded, the model is excluded from the results and treated as unsolved in our comparison. This threshold was empirically selected to balance evaluation throughput and coverage across a wide range of population sizes, and was sufficient to complete all models under the enhanced CoMoM algorithm.

The original CoMoM leverages exact arithmetic in Java, whereas the enhanced method implements in MATLAB the recursion we have proposed in Theorem 1 using standard floating-point arithmetic. Table 3.3 shows numerical results, which corroborate the high scalability of the enhanced CoMoM for homogeneous models. Results are obtained on an AMD Ryzen 7 2700X Processor with 64GB of RAM. Note that at population $|N| = 4000$ convolution becomes unviable due to excessive memory requirements, but since the normalizing constant reaches order 10^{-602} it would have exceeded the floating-point range during execution. Scaling methods for Convolution have been proposed in [Lam82], however, it is not difficult in our experience to generate examples of large models where this technique still cannot prevent floating-point range exceptions. Instead, the enhanced CoMoM can also solve the largest model with 10^6 jobs, agreeing within the first 11 digits of $\log G(m, N)$ with the results of the logistic expansion (LE) proposed in [Cas17], which is asymptotically exact. We have also observed in all cases that at least the first 6 digits of the mean per-class throughputs computed by the enhanced CoMoM were identical to the ones obtained by the Aggregate Queue Length (AQL) approximate MVA method [ZES88]. As no other exact method can reach this model scale, it is difficult to rigorously verify exactness, yet the result suggests no, or at least negligible, presence of error accumulation.

Table 3.3: Relative error and runtime upon computing $\log G(m, N)$ exactly.

<i>Classes</i>	<i>Total jobs</i>	<i>Method</i>	<i>Runtime [s]</i>
8	40	Convolution	0.0033
8	40	CoMoM (original)	0.0047
8	40	CoMoM (enhanced)	0.0014
8	400	Convolution	1.4201
8	400	CoMoM (original)	1.1433
8	400	CoMoM (enhanced)	0.0016
8	4000	Convolution	Memory exhausted
8	4000	CoMoM (original)	Timeout
8	4000	CoMoM (enhanced)	0.0017
8	10^6	Convolution	Memory exhausted
8	10^6	CoMoM (original)	Timeout
8	10^6	CoMoM (enhanced)	0.2591

3.3.2 CoMoM Algorithm for Homogeneous Models with Multiserver Queues

Using normalizing constants instead of MVA simplifies the calculation of probabilistic measures on each layer, as illustrated in this section. Thanks to loose layering, specialized results can be derived to allow simple computation of marginal probabilities in a layer, which enables the exact analysis of load-dependent service behavior, which encompasses multiserver stations as a special case [BGDMT06, CHO21]. We focus here in particular on computing using the CoMoM approach the marginal probability $\pi_N(m, n)$ that n jobs are queueing or receiving services at any of the m identical queueing stations. This is also equal to the probability that $\pi_N(m, |N| - n)$ jobs are waiting at the infinite server station.

Computing $\pi_N(m, n)$ is generally a difficult problem since with R classes there is a combinatorially large number of job mixes that result in the same total job population n at the m queueing nodes. In this case, our evaluation solver leverages a novel result, developed in the next theorem, which obtains marginal probabilities in $O(|N|^2)$ time and $O(|N|)$ space when applied to homogeneous models. For $m = 1$, this improves over MVA methods that require instead $O(|N|^R)$ time and space, while matching the complexity of CoMoM extension to marginal probabilities [Cas09, §VII], but without requiring the solution of a system of linear equations

as in the original CoMoM method. Other key novelties are that, unlike the original method, the expression below applies also to homogeneous models with $m > 1$ and with load-dependent service.

Theorem 2. *Consider a product-form queueing network model with R classes, having m identical c -server queueing stations with service demand $D_r > 0$ in class r , and an infinite-server node with think time Z_r in class r . Let $\pi_N(m, k)$ be the marginal probability that the m queueing stations have k resident jobs in total, $k = 0, \dots, |N|$. Define the following basis of unnormalized probabilities:*

$$\Pi(N) = G(m, N) \left[\pi_N(m, |N|), \dots, \pi_N(m, 0) \right]^T,$$

with $\Pi(0) = (0, \dots, 0, 1)^T$. Then $G(m, N) = |\Pi(N)|$ and the following exact recurrence relation holds

$$\Pi(N) = N_R^{-1} \mathbf{T}_R \Pi(N - \mathbf{1}_R), \quad (3.2)$$

in which

$$\mathbf{T}_R = \begin{bmatrix} Z_R & |N| \frac{D_R}{\mu(|N|)} & 0 & \dots & 0 \\ 0 & Z_R & (|N| - 1) \frac{D_R}{\mu(|N| - 1)} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & Z_R & \frac{D_R}{\mu(1)} \\ 0 & 0 & 0 & 0 & Z_R \end{bmatrix},$$

and where the load-dependent rate $\mu(n)$ can be computed as the throughput of a single-class product-form queueing network consisting of m identical c -server queueing stations, each having unit service demand.

Proof. Initially, consider a load-independent model with service rate $\mu(n) = 1$. Let the entries of $\Pi(N)$ be indicated with $\tilde{\pi}_N(n)$, $n = |N|, \dots, 0$. A probabilistic population constraint holds for homogeneous models with $m = 1$ [Cas09, Thm. 6]

$$N_R \tilde{\pi}_N(n) = Z_R \tilde{\pi}_{N - \mathbf{1}_R}(n) + n D_R \tilde{\pi}_{N - \mathbf{1}_R}(n - 1), \quad (3.3)$$

for all $n = 1, \dots, |N|$ and where $\tilde{\pi}_{N-1_R}(n-1) = 0$ if $n = 0$. With a load-dependent queueing station ($m = 1$), we generalize here the derivation in [Cas09] with similar passages to the following form

$$N_R \tilde{\pi}_N(n) = Z_R \tilde{\pi}_{N-1_R}(n) + n \frac{D_R}{\mu(n)} \tilde{\pi}_{N-1_R}(n-1), \quad (3.4)$$

where $\mu(n)$ is the load-dependent scaling at the queueing station. Note that the service rate $\mu(n)$ is the same irrespective of the number of classes R and the current class r , due to the structure of the product-form solution. Then, leveraging the flow-equivalent server theorem [CHW75], it can be computed as the throughput of an equivalent single-class model for the sub-network consisting only of the queueing stations. Organizing (3.4) in matrix form, we get (3.2).

□

Note that the service rates $\mu(n)$ can be pre-computed offline in $O(|N|^2)$ time, for example using the load-dependent MVA algorithm [BBA84], as their values are independent of R , D_r , and Z_r , $1 \leq r \leq R$. Note that this preliminary step has the same asymptotic complexity as the proposed algorithm for growing values of $|N|$.

Numerical Results

We have verified with numerical examples, comparing with the load-dependent MVALD-MX [BBA84], that the formulas in Theorem 2 match the exact solution while avoiding the exponential time and space requirements of traditional methods as the number of classes grows. Applying the same scaling as the one used for the multiserver CoMoM, we are able also to solve large models. For example, a model with $R = 8$ classes, $|N| = 40$ jobs, $m = 2$ identical PS queues with $c = 2$ servers, similar to the first row of Table 3.4, is solved exactly by MVALD-MX in 238 seconds, occupying about 76MB of memory, while the recurrence relation in Theorem 2 completes in about 1.5 milliseconds with negligible memory consumption.

3.3.3 Gaussian Quadrature Methods

As illustrated in the numerical examples, the CoMoM method for single server stations faces a slight increase in time requirements as the population grows. This also occurs as R increases, since the CoMoM basis has $2R$ elements. While tens or hundreds of milliseconds may be negligible for a single model, LQNs are solved iteratively and can feature many layers, hence solution times compound quickly. In large models, it can be useful to trade accuracy for speed.

A simple expression for the normalizing constant is given by the McKenna-Mitra integral form [MM84]. This is generally a multidimensional integral, with one dimension for each queueing station in the model. Thus, in a homogeneous model for a layer one would expect an m -dimensional integral. We show however that in homogeneous models this integral form takes the following simpler expression.

Theorem 3. *Under the same assumptions of Theorem 1, the normalizing constant of state probabilities for the queueing network model admits the following integral form*

$$G(m, N) = \frac{1}{(m-1)! \prod_{r=1}^R N_r!} \int_{u=0}^{+\infty} e^{-u} u^{m-1} \prod_{r=1}^R (Z_r + D_r u)^{N_r} du. \quad (3.5)$$

Proof. For a homogeneous model with m identical single server stations, the McKenna-Mitra integral takes the form

$$G(m, N) = \frac{1}{\prod_{r=1}^R N_r!} \int_{u_1=0}^{+\infty} \cdots \int_{u_m=0}^{+\infty} e^{-(u_1+\dots+u_m)} h(u_1 + \dots + u_m) du_1 \cdots du_m,$$

where $h(u) = \prod_{r=1}^R (Z_r + D_r u)^{N_r}$. We note that the multidimensional integral may be interpreted as computing $E[h(U_1 + \dots + U_m)]$ for the i.i.d. exponential random variables $U_i \sim \text{Exp}(1)$. The result then readily follows after noting that $U_1 + \dots + U_m$ is Erlang- m distributed with density $f(u) = \frac{1}{(m-1)!} u^{m-1} e^{-u}$. $\square$

The main difficulty associated with evaluating $G(m, N)$ directly is that (3.5) is prone to numerical difficulties. This is because quadratures do not operate directly in the log domain and

are therefore numerically sensitive to the magnitude of the factors under the integration sign, one being an exponentially decaying function (e^{-u}), the other being a polynomial of high order $|N| + m - 1$. A novel strategy developed in the NC solver to evaluate (3.5) is to use Gaussian quadrature methods coupled with the log-sum-exp trick [BHH21]. We have implemented both Gauss-Legendre and Gauss-Laguerre quadratures for (3.5), finding them empirically better suited at evaluating normalizing constants than MATLAB's default `integral` method and overall the best evaluation methods unless job populations are asymptotically large.

A Gauss-Laguerre quadrature of order K evaluates exponentially-weighted integrals using the approximation

$$\int_{x=0}^{\infty} e^{-x} f(x) dx \approx \sum_{k=1}^K w_k f(x_k), \quad (3.6)$$

where x_k denotes the k -th root of the Laguerre polynomial $L_K(x) = \sum_{j=0}^K \binom{K}{j} \frac{(-1)^j}{j!} x^j$, and with weights $w_k = x_k \left((k+1)^2 [L_{k+1}(x_k)]^2 \right)^{-1}$. Gauss-Legendre methods are similar but applicable to finite ranges $[a, b]$. Setting $a = 0$ and large b they can also help evaluate the normalizing constant. Their main benefit is that nodes and weights do not incur the same floating-point range exceptions as observed instead for Gauss-Laguerre quadratures of large order. We point to [JM18] for further details on Gauss-Legendre methods.

Generally, Gauss-Laguerre quadrature enables increasingly precise evaluations of (3.5) for growing values of its order K . However, it also faces numerical difficulties under a large population of jobs N , for which the quadrature weights and the integrand can display vastly different magnitudes. In such cases, we evaluate instead $\log G(m, N)$ in the quadrature summation by applying to the expression the log-sum-exp method, *e.g.*, the implementation in [BHH21].

Numerical Example

We consider the same models considered for CoMoM and numerically evaluate the integral form (3.5). The `integral` method of MATLAB is run with an absolute tolerance of 10^{-12} . Node and weights for the Gaussian quadratures are pre-computed offline: due to numerical instability, we can reach for the Gauss-Laguerre method up to order 300, while for Gauss-Legendre we could

Table 3.4: Relative error and runtime upon approximating $\log G(m, N)$.

<i>Classes</i>	<i>Total jobs</i>	<i>Method</i>	<i>Rel. error [%]</i>	<i>Runtime [s]</i>
8	40	MATLAB integral	0.0000	0.0006
8	40	Gauss-Legendre	0.0000	0.0004
8	40	Gauss-Laguerre	0.0000	0.0010
8	40	Logistic expansion	-0.1249	0.0012
8	400	MATLAB integral	0.0144	0.0005
8	400	Gauss-Legendre	-0.0001	0.0006
8	400	Gauss-Laguerre	-0.0001	0.0010
8	400	Logistic expansion	0.0033	0.0013
8	4000	MATLAB integral	Unstable	0.0008
8	4000	Gauss-Legendre	-0.0006	0.0021
8	4000	Gauss-Laguerre	-0.0006	0.0010
8	4000	Logistic expansion	0.0003	0.0013
8	10^6	MATLAB integral	Unstable	0.0008
8	10^6	Gauss-Legendre	-0.0592	0.0095
8	10^6	Gauss-Laguerre	0.2508	0.0011
8	10^6	Logistic expansion	0.0000	0.0013

pre-compute weights up to order 20,000 in the range $u \in [0, 10^6]$. We also include in the test the LE asymptotic expansion implemented in NC, which is a scalable method for models with few stations and many classes. The method applies a Laplace approximation to the simplex integral form for the normalizing constant in [Cas17]. Asymptotically the LE results are tight to the exact solutions.

The results are given in Table 3.4. Errors are computed using the exact CoMoM results. Overall, we see that Gauss-Legendre is typically sufficient except in large asymptotic models where LE solutions are closer to optimal. The lower performance of Gauss-Laguerre is interpreted as being due to the restriction of using up to 300 nodes and weights in the interpolation due to numerical instability in their computation. Since Gauss-Legendre quadratures of order $K = 2n - 1$ are exact for polynomials up to order n . Since the normalizing constant is the integral of a polynomial of order $n = |N| + m - 1$, it is possible to use the $K = 2|N| + 2m - 3$ order as a threshold for when the quadrature will cease to be exact and switch afterward to LE. For example, with Gaussian integration of order $n = 300$ and $m = 1$ the solution would switch to LE for $|N| \geq 6000$. Note that, on top of this, approximation errors are incurred in Table 3.4 by the log-sum-exp trick used for numerical stabilization, which explains why small errors are

incurred by Gauss-Legendre and Gauss-Laguerre also in cases where the quadrature should be exact. Another source of errors is that Gauss-Legendre requires a finite interval and has therefore been truncated to the range $u \in [0, 10^6]$, whereas the normalizing constant integral is defined in the range $u \in [0, \infty]$.

In summary, the numerical analysis reveals that a combination of Gauss-Legendre quadrature, for models with tens or hundreds of jobs, and LE, for larger models, provides an effective way to approximate homogeneous models.

Multiserver models. We remark that contrary to CoMoM, the above technique is not immediate to extend to multiserver stations. An integral form for this case is available in [MM86, Eq. (A20)], yet it is a multiple integral and depends on a Bessel function with oscillatory behavior, which renders the numerical quadrature more difficult. Seidmann's approximation may then be preferred to apply Gauss-Legendre and Gauss-Laguerre quadratures to models with multiserver stations [SSSO87]. This simple method replaces a c -server station with demands D_r with a sub-network consisting of a single server queueing station with demands D_r/c and an infinite server station having demands $D_r(c - 1)/c$. Under this transformation, as in the original system, a class- r job can traverse the two stations in D_r time overall when these are found both empty upon arrival. Moreover, the new infinite server station delay can be exactly aggregated within the pre-existing infinite server think time so that the model retains overall the same number of stations. This approximation style may be also instantiated with the multiserver CoMoM method of Section 3.3.2.

3.3.4 Non-iterative MVA Approximation

Lastly, we leverage the ability of our solver to encompass multiple solution paradigms to develop a fast, non-iterative, MVA-based approximation scheme for multiclass models. As mentioned in Section 3.2.2, non-iterative methods can be used for example to quickly initialize performance measures in submodels. We consider here single server stations and assume to approximate the multiserver case using Seidmann's approximation we have discussed before [SSSO87]. We also

take product-form assumptions, but the expressions may be extended to the FCFS case using the standard approximate MVA FCFS approximation and following similar passages.

For each class r , the standard MVA expression for the throughput in that class is

$$X_r(N) = \frac{N_r}{Z_r + mD_r + \sum_{s=1}^R D_r(N_s - \delta_{rs} - X_s(N - 1_r)Z_s)}, \quad (3.7)$$

where $\delta_{sr} = 1$ if $s = r$ and 0 otherwise, and we have used that the total population of jobs in the queue is $\sum_{s=1}^R (N_s - \delta_{rs} - X_s(N - 1_r)Z_s)$, since $X_s(N - 1_r)Z_s$ represents by Little's law the number of jobs of class s residing at the delay node (*i.e.*, the infinite-server think node) as seen at the instant of arrival at any of the queues in the homogeneous model.

To obtain a non-iterative formula, we use two approximations. First, we apply the interpolation [PKT90, p. 649]

$$X_r(N - 1_r) \approx \frac{N_r - 1}{N_r} X_r(N). \quad (3.8)$$

Next, we estimate the contributions of classes $s \neq r$ in (3.7) using twice the asymptotic upper bounds for the throughput [DB78, LZGS84], obtaining

$$\sum_{\substack{s=1 \\ s \neq r}}^R D_r X_s(N - 1_r) Z_s \approx B_r, \quad (3.9)$$

$$B_r = \sum_{\substack{s=1 \\ s \neq r}}^R D_r \frac{N_s Z_s}{Z_s + D_s(m + |N| - 1 - \sum_{c=1}^R Z_c(N_c - \delta_{cr}) / (Z_c + D_c(m + |N| - 2)))}.$$

Plugging (3.8) and (3.9) into (3.7) and re-arranging terms, these two approximations together make (3.7) a second-order univariate polynomial in the variable $X_r(N)$. This can therefore be solved explicitly, yielding the final expression

$$X_r(N) \approx \frac{Z_r - B_r + D_r|N| - \sqrt{B_r^2 + D_r^2|N|^2 + 2D_r|N|Z_r + Z_r^2 - 2B_rD_r|N| - 2B_rZ_r - 4N_rD_rZ_r}}{2D_rZ_r}. \quad (3.10)$$

The last expression can be computed in exactly R steps, one for each class r , thus with $O(1)$ complexity with respect to the total population size $|N|$.

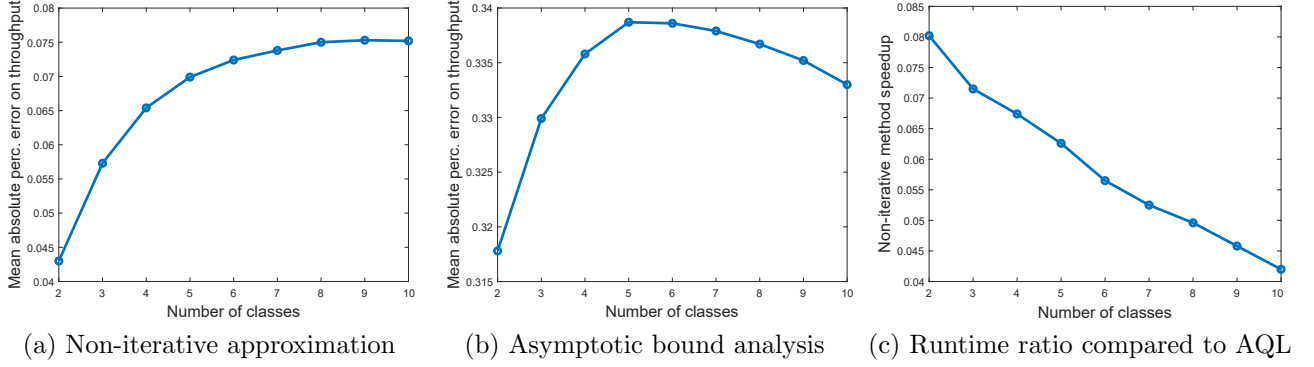


Figure 3.1: Mean absolute percentage error for X_r on 10,000 for (a) the non-iterative approximation (3.10) compared to (b) an asymptotic upper bound [DB78]. Figure (c) shows the ratio of runtime of (3.10) compared to AQL [Zah79], showing that the non-iterative approximation is much faster and the gap grows with more classes.

The mean queue-length at any of the m replicas is then simply $Q_r(N) = m^{-1}(N_r - Z_r X_r(N))$, since the population not resident at the delay will be, on average, equally distributed among the m queueing stations. Using X_r and Q_r for all classes, it is immediate from Little's law to obtain response times and utilization values.

Numerical Example

To illustrate the accuracy and speed of the proposed method, we consider the approximation of random closed product-form models for a varying number of classes R and jobs N . We set $m = 1$ station, $N = \lceil 1 + \beta 50 \rceil$, where β is a vector of non-negative uniform random numbers summing to 1.0, the demands D_r are chosen uniformly at random with mean 1.0, and the think times are also randomized as $Z_r = 70 + 5R \log_{10} RU$, where U is a uniform random number in $[0, 1]$. With this parameterization, we generate 10,000 models for each value of $R \in [2, 10]$ and compare (3.10) to the solution of the AQL approximate MVA algorithm, which is highly accurate [ZES88].

Figure 3.1a illustrates the approximation error of (3.10) as the number of classes increases. The average utilization across all the models generated by the particular choice of Z_r is such that it remains fairly constant as R increases, falling in the narrow range $[0.072, 0.080]$. The error grows slightly from 4.35% for $R = 2$ classes to about 7.51% with $R = 10$ classes, confirming the good

accuracy of the non-iterative approximation in (3.10) in light of its negligible computational cost. Figure 3.1b compares in the same experiment the performance of a standard asymptotic upper bound for the throughput [DB78], showing that the error is much larger than with (3.10), about 33% on average.

Figure 3.1c shows the ratio between the execution time of AQL and the execution time of the proposed non-iterative method. As shown, the computational gains grow roughly linearly with the number of classes. In our test, (3.10) on average took about 0.35ms to compute with $R = 10$ classes.

3.4 Experiments and Case Studies

The developments presented in the previous sections focus on improving the solution of a single layer and meta-solver methods. In this section, we first assess the computational gains arising from the meta-solver approach on a set of random instances evaluated for mean performance metrics. Then, we illustrate further advantages of the meta-solver approach in supporting advanced analyses that are difficult to handle in the MVA paradigm. Lastly, we provide a case study to demonstrate the extensibility of the meta-solver approach, which allows us to embed with solvers for other stochastic formalisms in LQN analysis, as illustrated by means of integrating caching and queueing analyses.

3.4.1 Experiments on Random Models

To evaluate the performance of LN, we generate a set of 30 random LQNs featuring a varying number of tasks, processors, layers, and synchronous calls. The detailed parameters of these models are listed in Table 3.5. In these models, we also assign random values to demands, multiplicities, and call means. Each LQN is solved with LN using the methods shown in Table 3.6.

A comparison of the performance of different solution methods in this experiment is then shown

Table 3.5: Parameter settings for generating random LQNs

<i>Factor</i>	<i>Range</i>	<i>Randomized?</i>
Clients	1	No
Layers	[1, 2]	No
Tasks	[2, 4]	No
Processors	[2, 4]	No
Client population	[5, 45]	Yes
Client think time	[10, 90]	Yes
Task multiplicity	[1, 9]	Yes
Processor multiplicity	[1, 9]	Yes
Demands	[1, 9]	Yes
Call means	[1, 9]	Yes

Table 3.6: Solution methods for solving LQN models

<i>Solution method</i>	<i>Description</i>
<i>ln.mva</i>	LN with approximate MVA in each layer.
<i>ln.nc.comom</i>	LN with the homogeneous CoMoM algorithm of Section 3.3.1 in each layer.
<i>ln.nc.gleint</i>	LN with the Gauss-Legendre integration of Section 3.3.3 in each layer.
<i>ln.fluid</i>	LN with the mean field approximation from [PC17] in each layer.
<i>ln.auto</i>	LN with the automated solver selection heuristic of Section 3.2.2 in each layer.

in Figure 3.2. The results highlight the mean, median, and maximum absolute error of the system throughput, as seen at the entry of the reference task that models the client workflow. The figure reveals that the heuristic proposed in Section 3.2.2 (*ln.auto*) outperforms all other LN solution methods, despite being a combination thereof. Specifically, *ln.auto* achieves a mean, median, and max error on the throughput seen by the client of 0.032, 0.022, and 0.171. We have observed that during the execution of *ln.auto* on the random LQNs, the heuristic chooses MVA about 9.6% times and NC the remaining times. Yet, it is interesting to note that, while CoMoM and the Gauss-Legendre methods often produce indistinguishable results, small deviations can still produce noticeable differences when looking at the iterative solution of an LQN. Yet, both methods appear to be useful since Gauss-Legendre is better in mean while CoMoM is better in median.

The Fluid, NC and MVA algorithms, albeit being in general very accurate, are less effective when used in isolation than when combined using the *ln.auto* method. This gives an indication of the gains resulting from using a meta-solver. We explain this due to accuracy variations in the solvers on submodels that require approximations falling outside product-form assump-

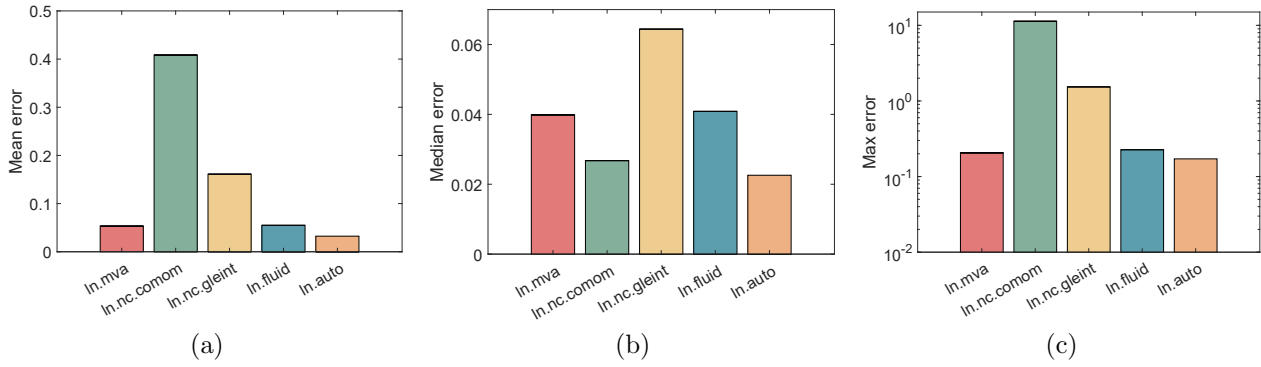


Figure 3.2: Performance of different solution methods in analyzing LQN models: (a) mean error, (b) median error, (c) max error

tions. The most common instance is a FCFS model with different (or non-exponential) service times, which is handled in the MVA solver using standard high-variability MVA approximations [Rei79]. This case is instead handled differently in the NC and Fluid solvers leveraging the decay-rate interpolation presented in [Cas20]. The latter is a heuristic that replaces FCFS queues with PS queues where demands are identical across the classes. The demands values are assigned by means of a non-iterative heavy-load diffusion approximation of the original FCFS queueing system [BGDMT06, §10.1]. The resulting model is then tractable using product-form solution algorithms. In practice, models exist where each of the three solvers is more accurate than the others, leading to a benefit in using a meta-solver.

We also remark that the Fluid solver performs better in this experiment than in Table 3.1 as the error metric here looks only at throughput accuracy without weighting in execution speed. Overall, these findings indicate that the proposed meta-solver heuristic of Section 3.2.2 can be useful for solving LQN models with heterogeneous submodels.

3.4.2 Case Study: Multi-Solver Based Layer Analysis

Once the LN iteration has reached a fixed-point for the input parameters of the submodels, multiple paradigms can be applied to obtain the metrics of interest. We illustrate this feature in the example shown in Figure 2.6, which describes a scenario where two job classes T1 and T2 require services from a server T3. Let S be a scale factor, in the model we assume $3S$ and

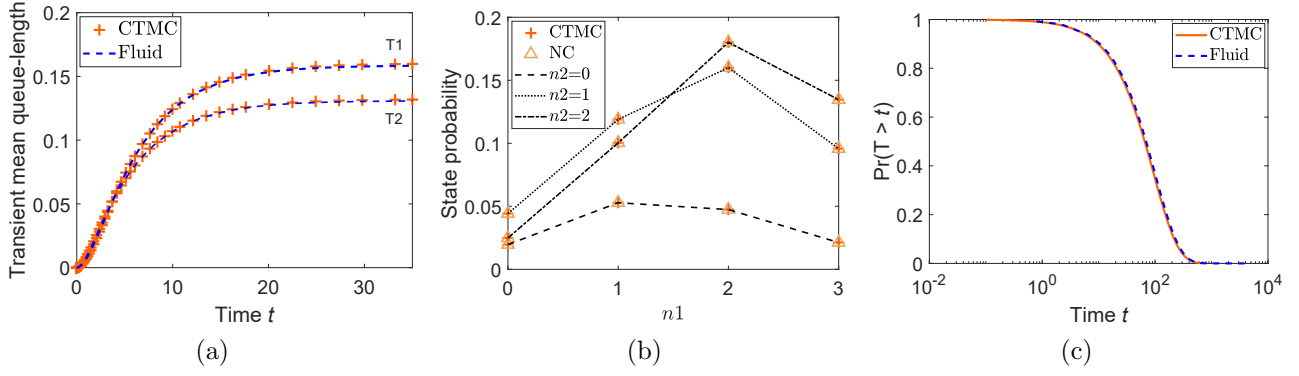


Figure 3.3: Leveraging LINE meta-solver capabilities in analyzing LQN models.

$2S$ jobs for *class1* and *class2* respectively, and T3 is an FCFS service station with multiplicity $c = 2S$.

We consider transient analysis, for which LINE provides multiple solver options. With the Fluid solver, one can compute transient metrics and passage time distributions in each layer. This solver performs mean-field approximations for PS nodes based on the theory presented in [PC17]. As mentioned, FCFS stations are also treated as PS nodes with service demands corrected through a decay-rate interpolation [Cas20]. Figure 3.3a shows the transient average queue length when the scale factor is set to $S = 1$ for the two job classes, representing T1 and T2 as clients, for the layer where T3 is modeled as a server. The plot shows a tight matching between the figures given by both CTMC and Fluid solvers. We can observe that the system reaches steady-state at around $t = 16$. In the figure, the Fluid solver looks very accurate for this non-saturated single queue scenario. Generally trends of CTMC and Fluid tend to be close under different parameterizations, but the two curves may not overlap at different loads and with varying number of servers.

The NC and CTMC paradigms allow the computation of joint and marginal state probabilities in each layer. This makes it possible to obtain probabilistic measures, which may be useful in parameter inference and buffer overflow analysis. Figure 3.3b displays the joint steady-state probabilities calculated by NC and CTMC solvers for the whole state space at T3, given the service demand of A3 as 50 and again a scale factor of $S = 1$. The results from both solvers are almost the same, but the speed of NC solver is much faster. In this example, the calculation

of each probability takes CTMC solver around 4 seconds while the NC solver takes less than 50 milliseconds.

It is also notable that, despite in this example the queue is multiserver FCFS with heterogeneous service times, the $M/G/k$ -diffusion approximation in NC allows us to compute the state probabilities very accurately, even if these do not admit a product-form solution. We have also attempted to scale up the same analysis to several tens of jobs, however the CTMC becomes infeasible. In these scenarios, the simulation cost becomes orders of magnitude slower than NC.

Beyond joint probabilities, LN allows us to compute response time percentiles at each layer using the Fluid and CTMC solvers. In the model, the operations of T3 are executed by the processor P3, here we use both solvers to obtain the response time distribution at the layer where T3 acts as a client to P3. Results for $S = 1$ are shown in Figure 3.3c, demonstrating agreement of the solutions. With $S = 200$ the infinitesimal generator matrix has an order of tens of thousands and too large for the CTMC solver, which is unable to produce an answer. Conversely, the Fluid solver has a cost invariant with the scale S and can solve the same model in less than a second, both for the cases $S = 200$ and $S = 1$.

We have repeated the last analysis only on the layer where T3 acts as a server. Since T3 is a multiclass FCFS station, the state space grows much faster than before for increasing values of S . For example, with $S = 50$ the CTMC is already intractable. Yet, the Fluid solver remains scalable, taking about 2 seconds to solve the model for $S = 50$. The increased time compared to the previous experiment is due to the additional iterations required by the $M/G/k$ -diffusion approximation. On this layer, it is also possible to use the CoMoM algorithm for multiserver stations presented in Section 3.3.2, which is exact and produces visually identical results to the Fluid solver. We find that NC solver based on CoMoM is more scalable than the Fluid solver, taking about 1.4 seconds with $S = 50$, using the response time distribution analysis method proposed in [MM86]. If one excludes the cost of computing the load-dependent scaling factors used in the technique proposed in [MM86], only 88 milliseconds are actually required by CoMoM to solve the model.

Recently, we have also demonstrated the possibility of using the queueing network solution

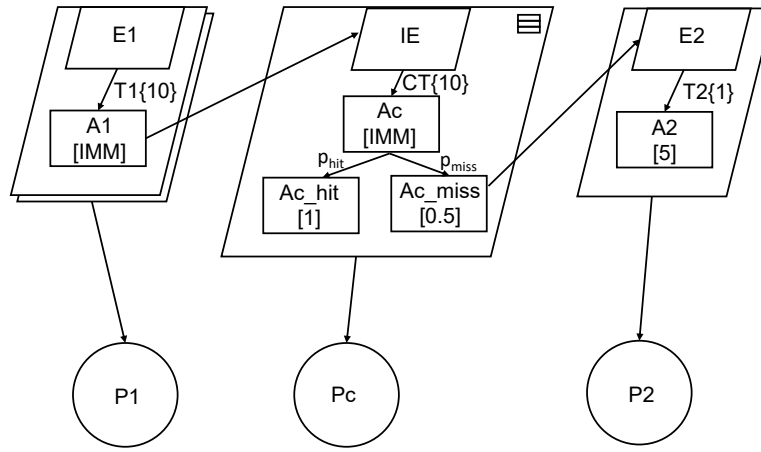


Figure 3.4: Example of a multi-formalism model containing a caching layer (middle task)

mixture density networks (MDNs) for response time distribution analysis [NC21]. The MDN-based approach considerably increases the precision of computing percentiles for LQN response times compared to analytical approximations, which are notoriously difficult for multiclass networks.

3.4.3 Case Study: Multi-Formalism Capabilities

LINE can analyze, analytically or via simulation, models with integrated queueing and caching formalisms. That is, the tool extends the LQN formalism, enabling the inclusion of *cache* nodes. When visiting a cache, a job reads an item according to some probability, resulting in either a cache hit or a cache miss. The subsequent processing activities can depend on whether the read outcome was a hit or miss. Cache reads also activate a replacement policy (*e.g.*, random replacement, FIFO, LRU) to evict infrequently used items. The LN solver features the ability to define caches in an LQN using specialized tasks and entries named `CacheTask` and `ItemEntry`, which capture the data access requirements of jobs traversing the LQN. More details can be found in [GC22] and Chapter 4.

LN can therefore also solve LQNs with caching, such as the three-layer model in Figure 3.4. The host processors P1, Pc, P2 adopt the processor-sharing scheduling policy. The number of users is represented by the multiplicity of task T1 and the number of jobs is represented by the multiplicity of task T2 and the cache task CT. A cache task is a novel LQN element

introduced by LN to describe data item reads from a cache, with different activities occurring based on whether a cache hit or a cache miss occurred. This is modeled using state-dependent class-switching.

In the example under study, item access probabilities obey a discrete uniform distribution and the cache is configured with a random replacement (RR) strategy. Arbitrary access probability distributions may be configured in LN and replacement strategies such as FIFO or LRU are supported. Jobs requested from T1 retrieve the items in the cache task CT. If the required items are cached, jobs will be processed by the *hit* activity with a probability of p_{hit} . Otherwise, jobs will be transformed to the *miss* activity with a probability of p_{miss} and be further processed by task T2.

To solve an LQN model containing caching, LN first decomposes the entire model into a group of layers, as illustrated in Section 2.5.1. For the layer without cache nodes, the solutions are given in Section 3.3. On the other hand, for the layer that involves a cache node, LN additionally divides the layer into two submodels. In the upper submodel, the cache node is isolated in an open model with Poisson arrivals, as shown in Figure 3.5a. In the lower submodel, the delay and the queueing station are contained in a closed queueing network with routing probabilities dynamically obtained from the p_{hit} and p_{miss} values obtained at the last iteration on the upper submodel. More details can be found in [GC22] and Chapter 4.

For this model that combines both the queueing and caching stochastic formalisms, numerical results given by LN are shown in Figure 3.5. Figure 3.5b demonstrates the miss probabilities against different numbers of items, which decrease with the improvement of the cache capacity. Figure 3.5c compares the throughput accuracy for cache solved by MVA and NC solver respectively. MVA analyzes caches by the fixed-point iteration method (FPI) proposed in [CG20], whereas NC implements the normalizing constant asymptotics proposed in the same paper. The solvers are both compared against an equivalent model constructed with a JMT model using both queueing and Petri net formalisms, similar to the validation model used in [GC22], but adapted to the example at hand. The results indicate high accuracy of both MVA and NC in capturing the cache layer throughput, which is generally a function of the cache hit ratio.

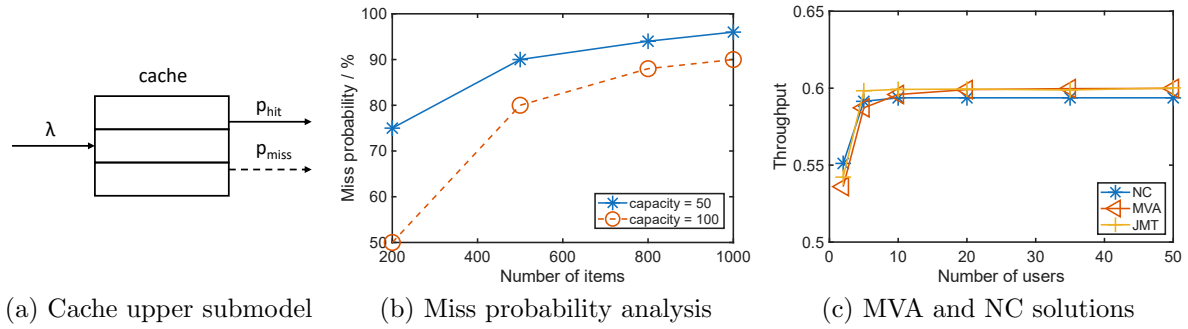


Figure 3.5: Capabilities of LINE in analyzing multi-formalism LQN models

3.5 Summary

In this chapter, we have presented a novel meta-solver framework to evaluate LQN models, discussing a concrete implementation in the LN solver, part of the LINE software package. The chapter argues that it is beneficial to consider multiple queueing network solution paradigms at once, to leverage their distinctive strengths. We have illustrated the concept also by introducing new analysis methods for homogeneous queueing networks, such as those obtained through LQN loose layering, in particular Gaussian integrals, a specialized and numerically-stable version of the exact CoMoM algorithm [Cas09], and a non-iterative mean-value analysis approximation.

These methods can solve submodels in milliseconds or less and improve both scalability and flexibility. In our experiments (Table 3.3), the enhanced CoMoM method solved models with up to 10^6 jobs across multiple classes, completing the exact computation of the normalizing constant in less than 0.3 seconds. This is among the largest exact product-form model evaluations reported to date.

No fundamental scalability issues were observed other than floating-point precision, which was addressed using a normalization step at each iteration. As a result, the solver can handle even larger models in principle, with the main limiting factors being memory and numerical range rather than algorithmic complexity.

In practice, the number of concurrent jobs per task in microservice-based systems usually ranges from a few tens to several hundreds [GZC⁺19], depending on system load and autoscaling

settings. The models evaluated in this chapter therefore provide sufficient capacity to cover both typical and stress-test scenarios in modern service-based applications.

Chapter 4

Layered Caching-Queueing Modeling

4.1 Introduction

Recent trends in software and system engineering have promoted the decomposition of monolithic applications into distributed architectures consisting of a collection of self-contained microservices. Each self-contained microservice can be independently developed and deployed in containers or virtual machines and interacts with the other microservices through lightweight mechanisms such as RESTful APIs [BHJ16, Sil16]. Compared to monolithic systems, a microservices architecture enables the dynamic deployment, migration, and removal of each service during runtime, enhancing flexibility, scalability, and adaptability [WGZ⁺19, CDT18], thus prompting their rapid adoption in industry [BOP22]. Such features, powered by containerization [PBSJ19], facilitate distributed processing of microservice applications, which can improve resource efficiency and resilience in various distributed systems, *e.g.*, mobile edge computing [WGZ⁺19] and stream processing [HH21].

However, distributed processing of microservice applications features complex and varying dependencies between in-memory storage and computational services [GZC⁺19] throughout the system. This complexity increases the difficulty for developers in performing capacity planning during the design phase of microservices, particularly when these services are deployed in distributed systems. Specifically, in-memory data storage services, *e.g.*, Memcached [LZS⁺21],

are usually modeled using stochastic caching analysis [GH15]. Instead, computational services perform different kinds of computations on the data, frequently resulting in contention among competing requests that may be modeled through stochastic queueing models [BGDMT06]. The complex inter-dependencies of computational processing, data placement, caching, and limited in-memory storage capacity poses a challenge in devising optimal resource management policies for such systems.

To design reliable performance predictions to drive the definitions of such policies, researchers have proposed many performance modeling methods. Several studies [YYG⁺19, CRF⁺18] leverage queueing models to make performance predictions, such as $M/M/c$ or $M/GI/1$ queues [YYG⁺19]. However, isolated queueing systems are insufficient to represent complex interdependencies between microservices, which can result in emerging phenomena, such as bottleneck switches [GCW19].

To address this issue, many studies [KSR⁺19, WDJ20, HLL⁺21, WLZ⁺22, WJY⁺22, FZC⁺21] adopt graph-based approaches to model computational service dependencies. For example, GrandSLAm [KSR⁺19] builds a directed acyclic graph (DAG) of microservices from a high-level language input to calculate partial deadlines for each microservice stage. Similarly, a microservice level parallelism applying the DAG modeling of requests for better resource utilization is proposed in [WLZ⁺22]. Wang *et al.* [WJY⁺22] use DAGs to model the microservices deployment problem. Fu *et al.* [FZC⁺21] utilize DAGs to model microservices and divide the graph into partitions for mapping to different nodes. In addition, Wang *et al.* [WDJ20] describe each workflow as a two-tuple and define a cost optimization problem with deadline constraints to optimize task scheduling of microservices. AlphaR [HLL⁺21] leverages a bipartite graph to extract the temporal characteristics of microservices. This however overlooks the role data placement, caching, and access locality plays in the end-to-end performance.

In the above-mentioned research, there are two limitations that we try to overcome in this chapter. First, graph-based modeling of computational services does not encompass scheduling strategies, resource contention, and stochasticity of workloads. These aspects can be however studied with stochastic performance models, such as LQNs [FAOW⁺08]. Next, current modeling

approaches dedicated to computational services neglect the significant role played by in-memory storage services in performance. This is a limitation since the invocation of services such as Memcached or other in-memory storage is nowadays fairly common to reduce response times. In some large-scale applications with complex dependencies, the call ratio of Memcached can reach 50% [LXL⁺21]. On the other hand, the processing time of in-memory storage services has a broad impact on overall efficiency. Some applications can spend up to 80% of their overall execution time reading data [DWT⁺18].

To aid in this, we present ICQ, a performance modeling framework to jointly analyze caching and queueing in layered systems. To implement this novel class of stochastic models, we generalize a class of models, *i.e.*, LQNs [FAOW⁺08], which are used in the performance engineering of layered service architectures. Our framework integrates for the first time caching characteristics in these models, enabling the analysis of data processing by means of workflow graphs that describe not only computation, but also data access dynamics and the effect of cache misses upon accessing an in-memory data storage node. The resulting class of generalized LQNs, called in this thesis integrated caching-queueing (ICQ) models, are analyzed by a novel fixed-point iteration method to approximate steady-state performance metrics.

To validate the accuracy of the proposed class of ICQ models, we simulate an advanced QPN based model¹ and implement the ICQ model within the LINE² tool [Cas20]. Extensive simulations based on real-world Azure traces are further conducted to evaluate the applicability of our proposed ICQ model. Results show that the ICQ model can find a better trade-off between system response time and memory consumption under a wide range of situations than the baselines. Overall, our contributions can be summarized as follows:

- a novel class of stochastic models to analyze the latency interplay of queueing for service access and cache hits/misses.
- a novel fixed-point iteration method to approximate steady-state performance metrics of the proposed ICQ model.

¹The validation datasets are available at <https://doi.org/10.5281/zenodo.6491327>

²The LINE tool is available at <http://line-solver.sourceforge.net/>

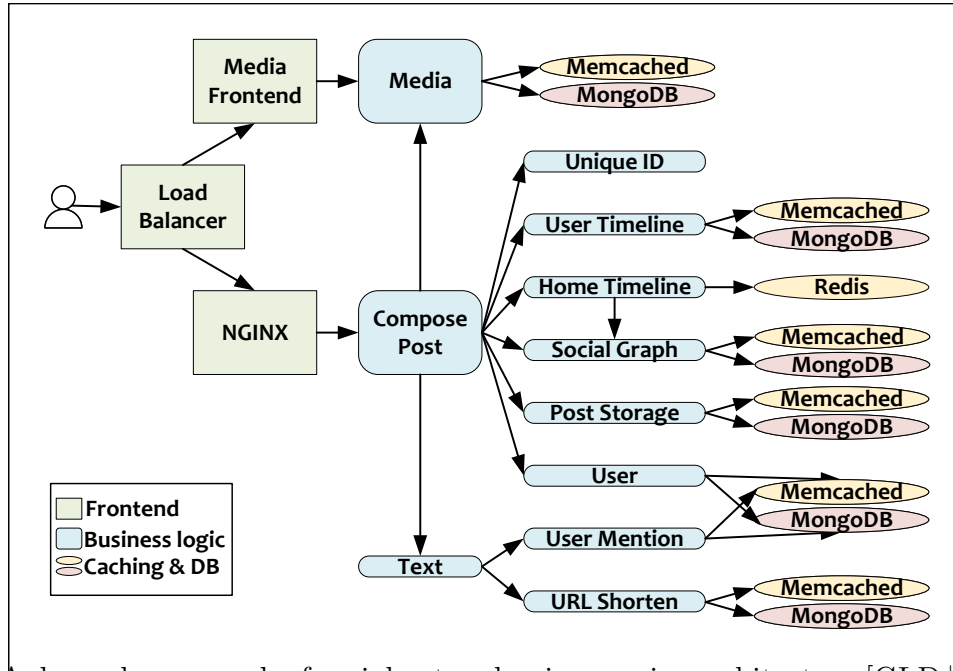
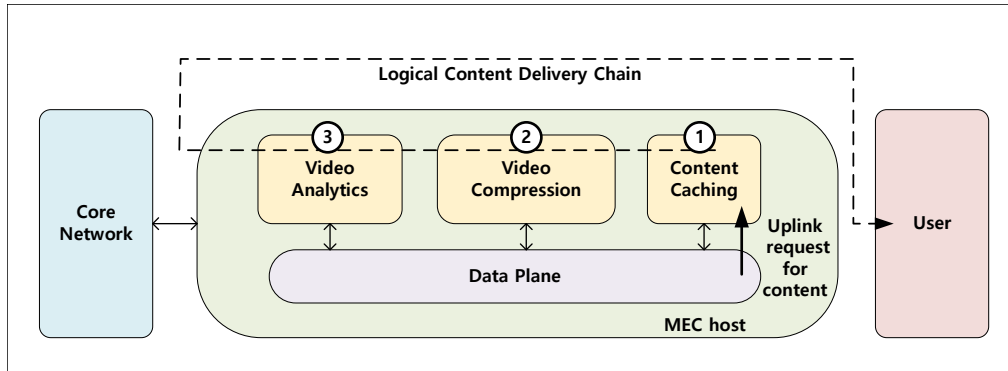
Figure 4.1: A dependency graph of social network microservice architecture [GLD⁺21, GZC⁺19]

Figure 4.2: A use case of video content delivery chain in edge computing [Com18]

- validations leveraging an advanced Queueing Petri Net based model, proving the high accuracy of the analytical model.
- extensive experiments based on real-world Azure traces, achieving up to 35% improvement in response time and 500MB reduction in memory usage than baseline heuristics.

The rest of the chapter is organized as follows: Section 4.2 shows the motivating examples for our work. Section 4.3 presents the LQN modeling of distributed workflows. Section 4.4 provides the ICQ model and gives the analytical solution to solve the model. Section 4.5 validates and evaluates the performance of the proposed model. The chapter is finally concluded with a summary of the contributions in Section 4.6.

4.2 Motivating Examples

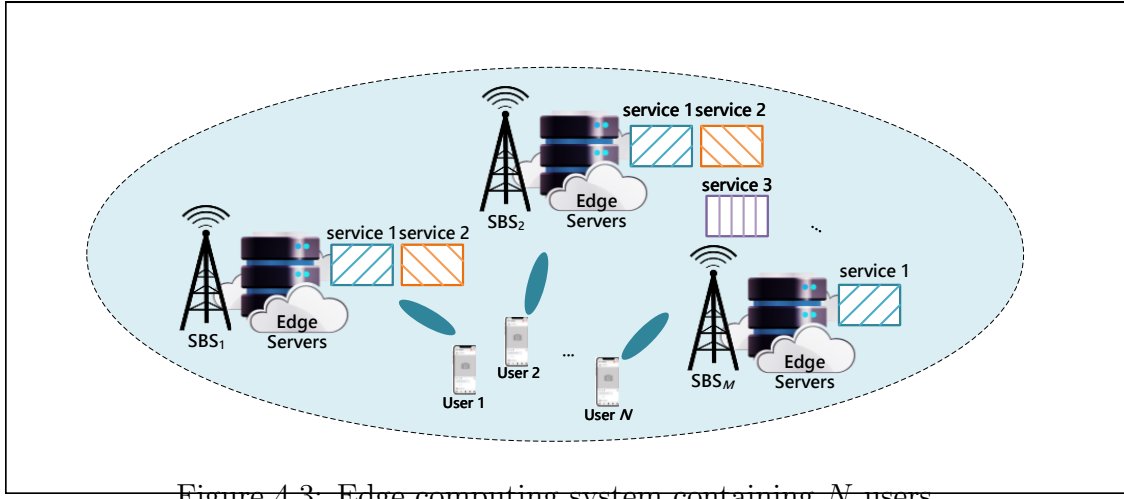
In the real world, most distributed data processing has complex dependencies between in-memory data storage and computational services. For example, Figure 4.1 illustrates an example of a social network architecture from [GLD⁺21, GZC⁺19], where the client requests are first forwarded to a front-end load balancer to be routed to multiple microservices. Subsequently, these requests are processed by a series of logic mid-tiers, encompassing an array of functions, such as creating posts, reading timelines, *etc.* Microservices-based architectures of this kind feature in-memory caching tiers, like Memcached and Redis, and persistent databases, such as MongoDB, to store critical data, such as posts, profiles, and media. This shows a concrete architecture where it is difficult to disentangle end-to-end performance by separating the analysis of computational and in-memory data storage services.

Another example is when processing video analytics in edge computing system [Com18], as shown in Figure 4.2, the request is first forwarded to the content caching function to retrieve the cached contents. After the identification, the request is further routed to video compression and video analytics services. Such processings can be distributed across different edge nodes, which feature the complicated inter-dependencies of computational processing, data placement, caching, and limited in-memory storage capacity in a distributed way.

Summarizing, the above examples offer a motivation for investigating the interaction between computed-oriented and data-oriented services in distributed systems in more depth. The complicated inter-dependency of queueing for service access and cache hits/misses calls for novel frameworks capable of jointly modeling the performance of in-memory storage and computational services in a distributed manner.

4.3 LQN Modeling of Distributed Workflows

Motivated by the above examples, we propose to establish a unified performance modeling approach for both computational and in-memory storage services. Therefore, in this section,

Figure 4.3: Edge computing system containing N users

we first introduce LQN modeling to serve as a basis to fit our purpose, which is proven to perform better in stochastic modeling of distributed workflows. This allows us to further integrate the description of both types of services to enable the modeling of systems such as those shown in Figure 4.1.

4.3.1 System Model

We consider a cluster of M edge nodes connecting N users in the edge computing system, as shown in Figure 4.3. Each edge node provides multiple services in edge servers co-located with the base stations. Each user offloads several jobs to the connected edge node to be served. The offloaded jobs may have precedences that must be processed in order. For example, a face recognition application [ZCPR03] can be regarded as consisting of five successive jobs, *i.e.*, object acquisition, face detection, pre-processing, feature extraction, and classification. Each job can be operated solely at the edge node that provides corresponding services. When the requested services are not provided in the connected edge node, the jobs need to be forwarded to other edge nodes that satisfy the requirements. Furthermore, different types of jobs at the same node also obey the sequential processing.

We assume the total number of jobs is K , which request a total number of C possible services. For job k at node m , $k = 1, 2, \dots, K, m = 1, 2, \dots, M$, we define two parameters as follows:

- *Service Placement Decision* p_{mk} . If node m provisions the service requested by job k , $p_{mk} = 1$. Otherwise, $p_{mk} = 0$. The set of all nodes that provision the service for job k is

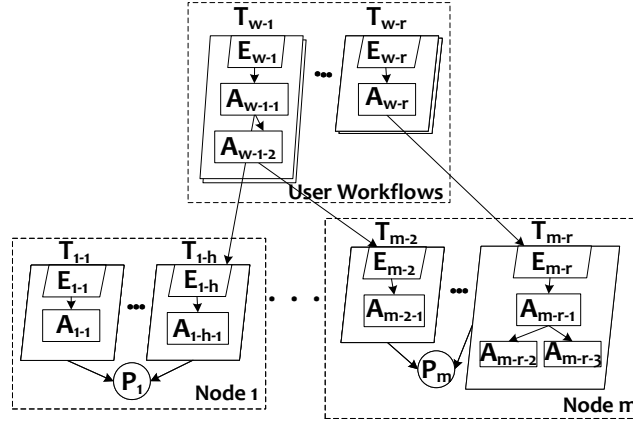


Figure 4.4: An example of layered queueing networks modeling for distributed processing of microservices

denoted as $\mathbf{F}_k = \{m | p_{mk} = 1, m = 1, 2, \dots, M\}$.

- *Mean service time t_{mk} .* This is the amount of computational processing requested by job k upon visiting node m . Different jobs may request different mean service time.

4.3.2 LQN Modeling

Figure 4.4 summarizes an example of the LQN model describing several workflows requested by clients running on a set of m edge nodes, where each node provides multiple services. A distributed architecture is seen as consisting of a set of nodes in a client-server relationship. Each node m is restricted to one of the following types:

- *Processor:* The computational resources on each node m are modeled as processor P_m , shown as a circle, obeying the processor-sharing scheduling policy.
- *Task:* The co-located servers are modeled as a set of tasks T_{mc} , shown as the larger parallelograms, each representing service c running on processor P_m . Each task has a queue to model first-come first-serve (FCFS) admission control and the service time follows a phase-type distribution.
- *Entry:* Each task publishes at least one entry E_{mc} , shown as a small parallelogram, which represents the endpoint for calls to service c . That is, upon invoking an entry E_{mc} , this will lead to a chain of activities being executed on the underpinning processor.

- *Activity*: Each entry invokes a sequence of activities A_{mci} , where $i = 1, \dots, I_c$ and I_c is the number of activities for service c . Activities are shown as a series of rectangles under the entries organized in the form of a DAG, to represent independent or dependent executions. Activities may trigger synchronous or asynchronous calls to entries, shown as arrows, resulting in complex service workflow.

The clients are modeled in the LQN formalism as *reference tasks*. A reference task is a modeling abstraction used to describe a workload class, in our case a client workflow DAG. That is, the workflows describing the client request chains, *e.g.*, T_{w-1} , are described as DAGs within the reference task, called activity graphs in LQN terminology, to represent the precedences between the jobs. The mean queue length of the infinite server on the reference task is defined as the multiplicity of the workflow. When requested jobs are assigned to different nodes, the activity graph will feature specific activities that make synchronous calls to corresponding nodes based on their precedence, such as A_{w-1-1} and A_{w-1-2} .

4.3.3 Problem Formulation

The LQN models we have described above in Chapter 3 allow us to obtain analytical estimates of throughputs, response times (*i.e.*, latency per unit invocation), mean queue lengths (*i.e.*, backlogs), and utilizations for each component type, for example using the algorithms given in [FAOW⁺08, CGNZ24]. We focus on response time throughout the rest of the chapter, but other reference metrics can be easily generated from the models.

Our optimization objective is to minimize the end-to-end total response time R of the LQN-based model, which is expressed as

$$R = \sum_{i=1}^C \frac{X_i}{X} R_i, \quad (4.1)$$

where X_i , R_i , and X are the throughput of job class i , the response time of job class i , and the total throughput summed over all service classes, respectively. The ratio X_i/X may be seen as related to the probability at steady-state where an arrival is of class i . Thus, the problem of

dependent job scheduling and service placement is formulated as

$$\begin{aligned}
& \min_{\mathbf{x}} && R(\mathbf{x}), \\
& s.t. && \sum_{m \in \mathbf{F}_k} x_{mk} = 1, \\
& && x_{mk} \in \{0, 1\},
\end{aligned} \tag{4.2}$$

where x_{mk} denotes the scheduling decision for job k at node m . The vector $\mathbf{x} = [x_{11}, \dots, x_{MK}]$ represents the set of scheduling decisions for all jobs and $R(\mathbf{x})$ is the corresponding system response time with decision vector $\mathbf{x}$. The first constraint guarantees that job k is solely served at one (and only one) feasible edge node provisioning the requested services. The second constraint ensures that if job k is scheduled to be served at node m , then $x_{mk} = 1$, or $x_{mk} = 0$ otherwise.

The optimization program in (4.2) is intended for design-time planning, enabling globally optimized placement decisions with minimal runtime overhead. While not suitable for online adaptation in its current form, the ICQ framework provides a foundation for future runtime extensions using surrogate models or incremental solvers.

4.3.4 Evaluation

We evaluate the merits of utilizing LQNs to model the microservice placement problem with a focus on computational services. We begin by comparing our established LQN-based model with the method in [ZXZ⁺20], which formulates an offloading dependent tasks with service caching (ODT-SC) problem. The ODT-SC problem considers deterministic scheduling with deterministic service time, while our scheme focuses on stochastic scheduling with random service time. Moreover, the optimization goal of the ODT-SC problem is to minimize the maximum makespan, *i.e.*, the sum of the start time of a job and its execution time, while our goal is to minimize the total response time. Despite the differences in the assumptions, we regard this baseline as a suitable comparison to just the benefits of stochastic LQN-based modeling of complex edge workflows.

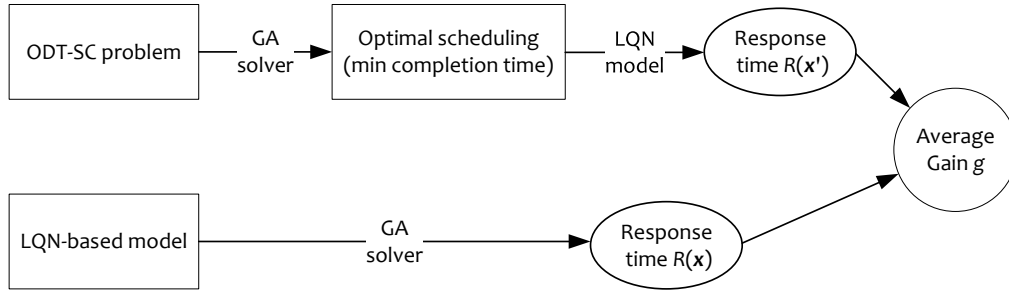


Figure 4.5: An overview of the comparative procedure between the LQN-based model and the method in [ZXZ⁺20]

For comparison, we set the mean service time for each job in our LQN-based model equal to the execution time in [ZXZ⁺20]. The optimization goal of the ODT-SC problem is reset to minimize the total completion time, which is the sum of the makespan of each job. Moreover, the method in [ZXZ⁺20] reformulates the ODT-SC problem into a convex programming to get feasible solutions. To be more general, we solve the ODT-SC problem by genetic algorithm (GA) without the need to relax the constraints. An overview of the comparative procedure of the two schemes is presented in Figure 4.5.

We first leverage the GA to address the ODT-SC issue, obtaining the optimal scheduling strategy $\mathbf{x}'$ of each job. Then the strategy is transformed into an LQN model to calculate the total response time $R(\mathbf{x}')$. The response time corresponding to the minimum completion time in each iteration is selected as the comparative object. Meanwhile, we equally exploit the GA to solve the proposed LQN-based model to search for the minimum total response time. Finally, we compare the two response times to observe the average gain g of our proposed LQN-based method, which is expressed as

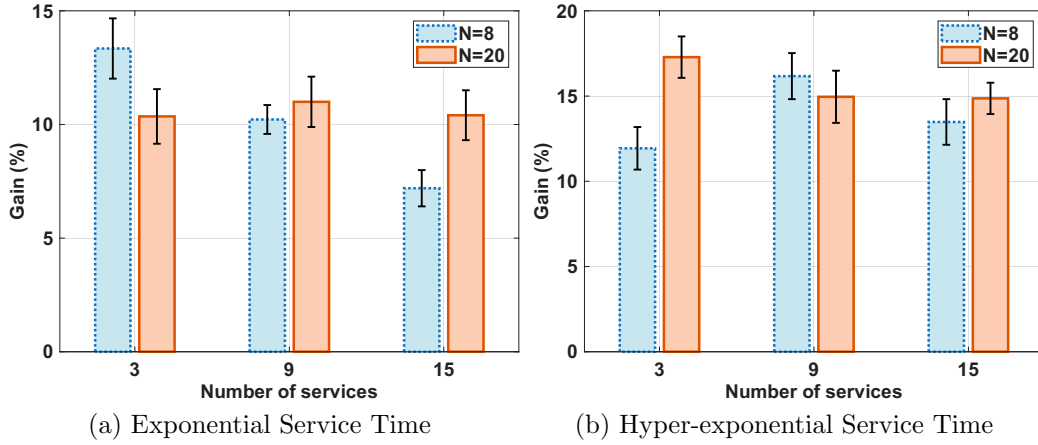
$$g = \frac{1}{I} \sum_{i=1}^I \frac{\| R_i(\mathbf{x}') - R_i(\mathbf{x}) \|}{R_i(\mathbf{x}')}, \quad (4.3)$$

where I is the total iterations.

In each iteration $i = 1, \dots, I$, to reduce the estimation error, we run Q replications of the GA solver to search for the optimal solutions. In each replication of the GA solver, the number of generations is denoted by G . The dimension of the design variable for the ODT-SC problem is $2K$, of which the first and second K elements are the start times and scheduling decisions of each

Table 4.1: Parameter settings for the comparative experiments

Definition	Parameter	Range of Value
number of edge nodes	M	2, 5
number of users	N	8, 20
number of services	C	3, 9, 15
number of iterations / replications	I / Q	30
number of generations	G	1000
number of decision variables	$nvars$	computed as $M \times N$

Figure 4.6: Average gain of response time of the LQN-based model compared to the method in [ZXZ⁺20] ($M = 2$)

job respectively. The service time is deterministic but varying in different kinds of services. While for the LQN-based model, the dimension of the design variable is the K scheduling decisions of each job. The service time of each job class is set to be exponentially distributed or hyper-exponentially distributed, of which the mean value is equal to the one in the ODT-SC problem.

A set of experiments are conducted to analyze the performance under different conditions. The setting of different parameters is shown in Table 4.1, where $nvars$ is the number of decision variables. The distribution of service time is assumed as exponential or hyper-exponential. For simplicity, the user workflow adopts chain structures for multiple services in the experiments, but it is easily extended to cases with parallel structures by LQNs.

The experimental results are presented in Figure 4.6 and 4.7, which can be seen that using

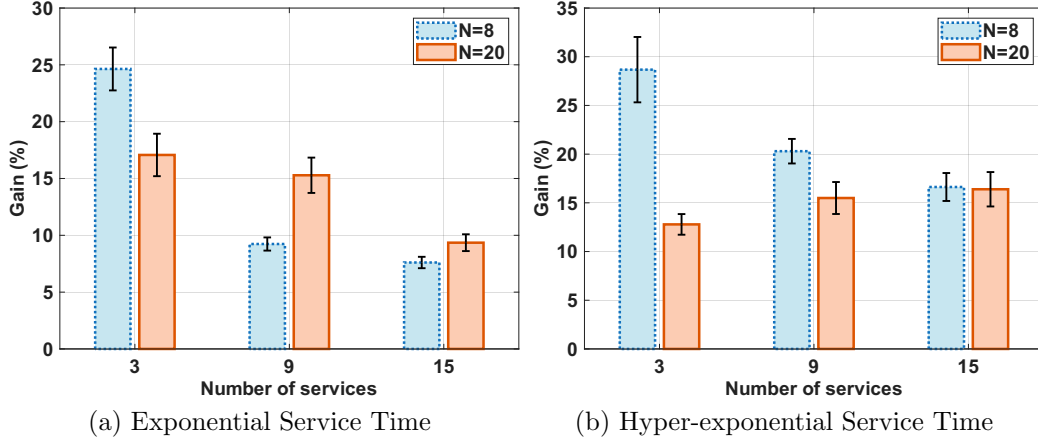


Figure 4.7: Average gain of response time of the LQN-based model compared to the method in [ZXZ⁺20] ($M = 5$)

LQN-based models yields a better average performance than the method in [ZXZ⁺20]. When $M = 2$, the gain of total response time is a minimum of 7%, an average of 12%, and a maximum of 17%. When $M = 5$, the gain of total response time is a minimum of 7%, an average of 16%, and a maximum of 28%. As a whole, the average gain improves with the increase of the number of edge nodes. Our results show that it is beneficial to apply LQNs to stochastic scheduling as opposed to heuristically apply a deterministic scheduling, *i.e.*, the solution to the ODT-SC problem. Therefore, it is meaningful to use the LQN-based model for further performance analysis of distributed processing of microservices in edge computing systems.

4.4 Integrated Caching-Queueing Analysis

In this section, we integrate caching into the layered queueing model to establish a new formalism for the modeling of in-memory and computational services. To tackle this, we first define two novel elements, *i.e.*, *cache-task* and *item-entry*, to declare caching entities in an LQN model. In particular, we present the extension for single-node caches ($h = 1$) as they are more widespread. Then we give the analytical solution for the resulting ICQ models to obtain accurate steady-state performance estimates.

4.4.1 Layered Caching-Queueing Model

To extend LQN models to include caching components, we first define two novel components as follows:

- *cache-task*: Each caching node is defined as a cache-task in the LQN model. Each cache-task offers the ability to access a collection of items through a cache, such as a key-value store. Cache-tasks have the basic properties of tasks, but add four specific properties for caching: the total number of items, the cache capacity, the cache partitioning into lists, and the cache replacement policy, *e.g.*, least recently used (LRU), least frequently used (LFU), first-in first-out (FIFO), and random replacement (RR).
- *item-entry*: The services provided by the caching module are defined as item-entries in the LQN model. Each item-entry represents a collection of items accessible through the cache-task. Item-entries are similar to standard entries, but add the property of the popularity of the items, *e.g.*, Zipf or custom distribution.

Upon issuing a call to an item-entry exposed by a cache-task, a client obtains the requested object either after a cache hit or a cache miss. The hit/miss selection depends on the cached contents and reflects as a choice of a different branch in the DAG bound to the item-entry. This is modeled through a novel precedence relationship, *i.e.*, an activity graph edge. We name this precedence relationship as *cache-access* for the cache hit and miss activities under each item-entry, which can be used to specify the branching in the activity graph. The *cache-access* transforms an incoming job into a hit or miss job, sending out the transformed job to the corresponding outgoing link with a certain probability.

The cache replacement policy of each cache-task adopts by default the RR policy, which randomly selects a candidate item to replace when a cache miss occurs. The RR policy can offer the same steady-state performance of the time-to-live (TTL) mechanism [FNNT14] with exponential timers. Hence it is a reasonable approximation of the real-world TTL with better analytical tractability. Other policies, such as LRU or CLIMB, can be easily obtained based on our framework.

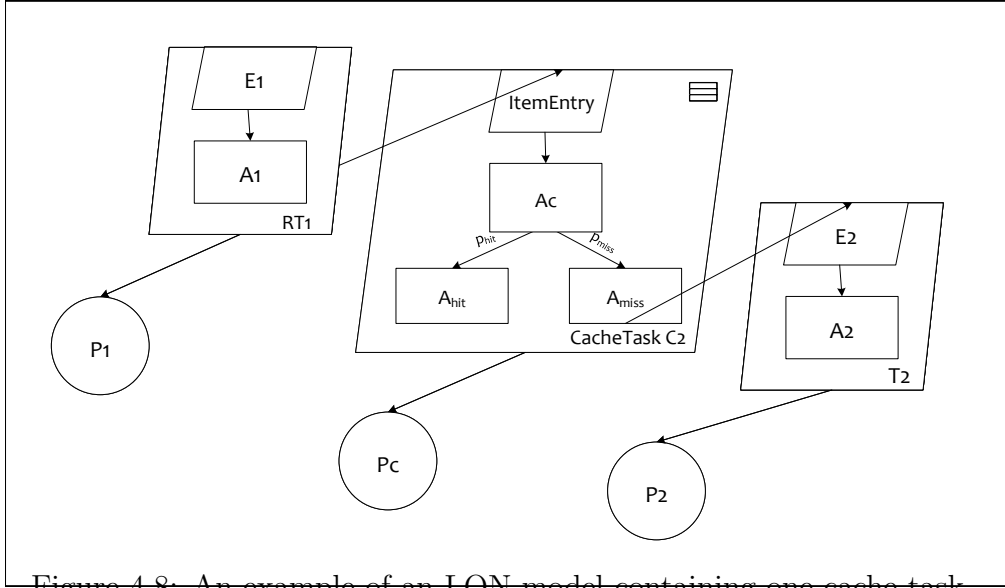


Figure 4.8: An example of an LQN model containing one cache-task

Based on such extensions, we generalize the LQN model by enabling caching nodes and denote the model as an ICQ model. Figure 4.8 presents an example of an ICQ model containing one cache-task to model the media workflow in Figure 4.1. The top-right symbol in the task parallelogram distinguishes a cache-task from an ordinary task. When a request issued by the media front-end arrives, the activity A_0 will issue calls to Memcached to retrieve cached contents. We model the probability that the item is in the Memcached with p_{hit} and out of Memcached with $p_{miss} = 1 - p_{hit}$. If a cache hit occurs, the cache hit activity A_{hit} is selected with p_{hit} to issue calls to Media services for further processing. If a cache miss occurs, the cache miss activity A_{miss} is then selected with p_{miss} to issue calls to MongoDB services to retrieve the contents. This generated architecture allows joint modeling of caching and queueing into the same layered model. It is important to note that p_{hit} and p_{miss} need *not* to be specified by the end user. They are automatically determined through the solution of the stochastic models underpinning the ICQ model, which also relies on a stochastic equilibrium analysis of the cache replacement policy, as described in the next section.

4.4.2 Analytical Solution for the ICQ model

To solve the ICQ model, the first step is to decompose the entire model into a group of sub-models. Each sub-model may be seen as a mixed queueing network that contains clients and servers. The term mixed refers to the simultaneous presence of open and closed service classes.

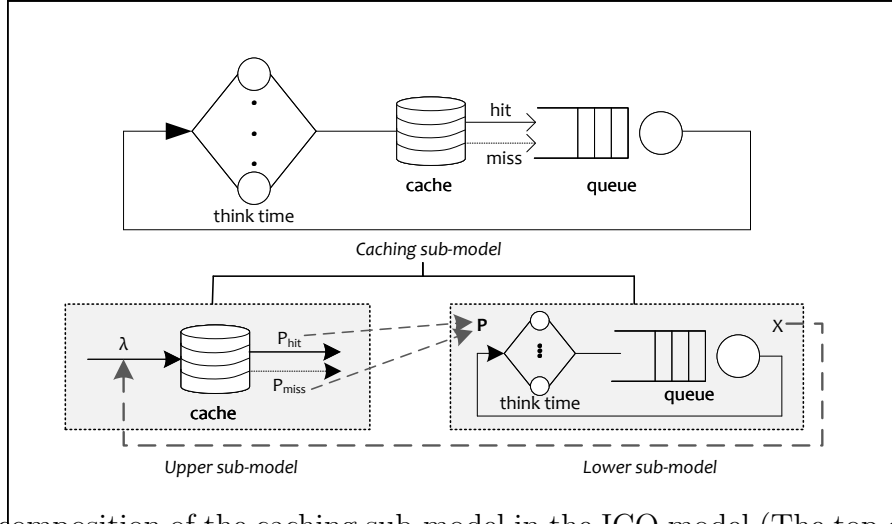


Figure 4.9: Decomposition of the caching sub-model in the ICQ model (The top diagram shows the original caching sub-model, which is decomposed into the two sub-models below. The dashed boxes denote the boundaries of the upper and lower sub-models.)

The former represents asynchronous requests from clients, while the latter represents synchronous ones (*i.e.*, closed since issued from a finite connection pool). In particular, when a queueing network contains the cache module, the workload is synchronous, which means the issue of the next request to the cache is required to take place after the reply of the last request issued to the cache. This assumption reflects common patterns in microservice architectures, where cache accesses are typically performed via blocking RPCs or synchronous APIs. In such cases, the service waits for the cache response before issuing the next request, justifying the use of a closed network representation. While asynchronous interactions may occur in systems using non-blocking or message-based cache access, these can be incorporated into the ICQ framework by extending the interaction model, leveraging the flexibility of LQNs.

To model these dynamics, we additionally divide the caching sub-model into two sub-models, as shown in Figure 4.9. In the upper sub-model, the cache module is isolated in an open queueing network with asynchronous Poisson streams. While in the lower sub-model, the delay and the queueing station are contained in a closed queueing network. To solve the caching sub-model, we generalize the analytical method for non-layered queueing networks from [CG20] to the multi-layer setting of ICQs.

Cache requests arrive in the upper sub-model as multiclass Poisson streams with rate $\lambda_v = \sum_r \lambda_{vr}$. Each stream maps to one of the chains in the original caching sub-model. We use an initial guess of the arrival rate $\lambda_v^{(0)}$, $v = 1, \dots, u$ and of the hit and miss rates of the cache node,

i.e., $p_{miss}^{(v,t)}$ and $p_{hit}^{(v,t)}$. The obtained values in each iteration are leveraged in [CG20] to update the arrival rate $\lambda^{(t+1)}$ by the fixed point iteration:

$$\lambda_v^{(t+1)} = \frac{s_v \lambda_v^{(t)}}{\lambda_v^{(t)} \theta_t + p_{miss}^{(v,t)} \theta_m + p_{hit}^{(v,t)} \theta_h}, \quad (4.4)$$

where $\theta_t, \theta_m, \theta_h$ respectively are the mean delay between consecutive requests to the cache, and the mean delay due to miss and item fetch. Here, s_v is the maximum number of pending requests of stream v to the cache, *i.e.*, the total population of jobs in the caching sub-model. The iteration process continues until the value of λ_v converges. In our experiments, the fixed-point iteration for λ_v consistently converges within a small number of steps, and the convergence behavior is aligned with that reported in [CG20].

The hit and miss rates at each iteration are used to parameterize the queueing network of the *lower* sub-model as routing probabilities. That is jobs representing calls that experience cache hits will dynamically switch after visiting the cache to a *hit service class*, which will be processed by the queueing station according to the activities specified in the activity graph for hits. Similarly, calls that experience misses will switch to a *miss service class* representing the workflow in the activity graph for misses. This class-switching based separation of the hit and miss calls enables the modeling of the different response times experienced by the two classes of requests, which depend on whether the items they accessed were cached or not. At the end of the chain of operations that follow hit/miss, both classes of jobs will be merged back into the original class and return to the user as a response for that class.

The solution to the upper sub-model is reconciled with the lower sub-model iteratively, as presented in line 6-15 in Algorithm 1. After an initialization of the arrival rate $\lambda_v^{(0)}$ (*e.g.*, to a lower bound such as $1/(\theta_t + \theta_m + \theta_h)$), miss probability $p_{miss}^{(v,0)}$ (*e.g.*, set to $\frac{1}{h+1}$, with h denoting the number of cache nodes in the model), and hit probability $p_{hit}^{(v,0)} = 1 - p_{miss}^{(v,0)}$, the iterative approach begins by solving the upper sub-model with (4.4) to calculate $\lambda_v^{(n)}$, $n \geq 1$. The hit and miss probabilities, *i.e.*, $p_{hit}^{(v,n)}$ and $p_{miss}^{(v,n)}$, are then solved with approximation methods, *e.g.*, the fixed-point iteration, with the detailed derivation provided later in Equations (5.3) and (5.7). These probabilities are passed to the lower sub-model to set the routing matrix. Subsequently,

the lower sub-model undergoes analysis with a queueing network solver method $f(s)$, *e.g.*, approximate mean value analysis (AMVA) [BGDMT06], to obtain the mean throughput $X_v^{(n)}$. The computed throughput is then passed to the upper sub-model as the next arrival rate $\lambda_v^{(n+1)}$. This iterative process continues until the arrival rate λ_v converges.

The entire procedure to solve the ICQ model containing multiple cache-tasks is presented in Algorithm 1, where $f(\cdot)$ is the custom solver for the sub-model s that returns throughputs, queue-lengths, response times, and utilizations. Specifically, when presented with an input of the ICQ model containing cache-tasks, our approach initially decomposes the entire model into sub-models using loose layering [Fra00]. In cases where a layer encompasses a caching module, we further partition the caching sub-model into upper and lower sub-models, updating parameters at each iteration such that the solution of the upper sub-model is reconciled with that of the lower sub-model. The iteration process continues until the value of λ converges. Alternatively, for layers without a caching module, we employ custom solvers, such as AMVA, to address the respective sub-model accordingly in line 17-18, which follows the standard LQN solution algorithm explained in [Fra00]. We note that such sub-models, which do not include caching, can also be solved using the LN meta-solver for automatic method selection. In this part, we employ custom solvers, such as AMVA, for consistency and interpretability, though the LN meta-solver is equally applicable. Based on this, we can obtain the mean performance metrics of the model, such as response times and throughputs.

4.5 Model Validation and Evaluation

4.5.1 Validation

We validate the generalized ICQ model by Java Modelling Tools (JMT) [BCS09], which is an open-source suite for performance evaluation. Specifically, we use the JSIMgraph discrete-event simulator, which facilitates the analysis of queueing networks and stochastic Petri nets. Stochastic Petri nets [BCS09] describes a set of tokens, equated in our model with jobs or

Algorithm 1 Solution algorithm for the ICQ model**Input:** ICQ model with cache-tasks, Sub-model solution algorithm $f(\cdot)$

```

1: Decompose the ICQ model into  $S$  sub-models
2: while not converged or not reached iteration limit do
3:   for  $s \leftarrow 1$  to  $S$  do
4:     if  $s$  is caching sub-model then
5:       Decompose sub-model  $s$  into two sub-models.
6:       Initialize  $\lambda_v^{(0)} \leftarrow \frac{1}{\theta_t + \theta_m + \theta_h}$ ,  $p_{miss}^{(v,0)} \leftarrow \frac{1}{h+1}$ , and  $p_{hit}^{(v,0)} = 1 - p_{miss}^{(v,0)}$  in the upper sub-model.
7:        $n = 1$ 
8:       while  $\|\lambda_v^{(n)} - \lambda_v^{(n-1)}\| \geq \delta$  do
9:         Solve the upper sub-model for  $\lambda_v^{(n)} = \frac{s_v \lambda_v^{(n-1)}}{\lambda_v^{(n-1)} \theta_t + p_{miss}^{(v,n-1)} \theta_m + p_{hit}^{(v,n-1)} \theta_h}$ .
10:        Using  $\lambda_v^{(n)}$ , obtain  $p_{hit}^{(v,n)}$  and  $p_{miss}^{(v,n)}$ . (See Equations (5.3) and (5.7) for details.)
11:        Pass  $p_{hit}^{(v,n)}$  and  $p_{miss}^{(v,n)}$  to the lower sub-model to set its routing probabilities.
12:        Solve the lower sub-model with  $f$ .
13:        Return the throughput returned by  $f$  to the upper sub-model as the next arrival rate  $\lambda_v^{(n+1)}$ .
14:         $n = n + 1$ 
15:       end while
16:     else
17:       Obtain the response time for sub-model  $s$  using the solution algorithm  $f$ .
18:       Update in all other layers the think time for requests that originate from sub-model  $s$ .
19:     end if
20:   end for
21: end while
22: Obtain using  $f$  response times, residence times, throughputs, and utilizations at each layer.
Output: ICQ response times, residence times, throughputs, and utilizations.

```

data items, circulating across places, represented diagrammatically as circles. Transitions, represented as boxes, are defined by exponential timers that, upon elapsing, pull jobs from places connected to their input and deposit a set of newly generated tokens to the places connected to their output.

As the RR replacement policy involves a synchronized movement of randomly chosen items among lists, this can be encoded in terms of transitions that pull and deposit tokens from the cache, each representing a data item access. The support in JMT for queueing networks allows us instead to characterize the queueing behaviour and CPU consumption associated to cache service. The joint assessment of caching and queueing then follows by simulating a hybrid model combining both queueing stations and stochastic Petri nets. From this, we can get an accurate estimate of the expected performance of an ICQ model, against which we compare

our FPI-based implementation described in Algorithm 1.

In particular, we consider a validation model containing the upper two layers of the model shown in Figure 4.8. In the upper layer, the underlying processor P_1 adopts the processor sharing strategy and the number of users is represented by the multiplicity of the reference task RT_1 . The think time is set as an Immediate transition, which fires instantaneously once its enabling condition is satisfied, to ensure that the client issues without delay the request to the cache-task.

In the lower layer, the capacity of the cache-task is set as 1 with a total number of 3 items. All the items obey discrete uniform distribution and take the RR strategy. The underlying processor P_c also employs the processor sharing strategy and the number of tokens in this layer is denoted by the multiplicity of the cache-task. The service rate of hit and miss activities on processor P_c is exponentially distributed with a parameter of μ_h and μ_m respectively.

The validation model is mapped to the JMT model presented in Figure 4.10. The jobs first originate from the delay node and are stored at the place of *upperlayer*. Then, utilizing Petri net transition, *i.e.*, waiting for the tokens to be available in the inbound places and firing a given number of tokens in the output, we enable the transition 2 by 1 job from the upper layer and 1 token, to fire 1 job into the lower layer. To determine a cache hit or miss in the lower layer, we regard the cached items as different job classes. Thus, the fired job class can switch into any of the items at the *ClassSwitch 1* node according to the probability distribution. Based on the state of the stored items in *Cache* and *OutOfCache* places, the transition 1 determines whether to fire a hit or miss job class into the queue. After different services for the hit or miss job at the queue node, the job transforms back to the original job class at the *ClassSwitch 2* node and enables the transition 3. Finally, the transition 3 fires 1 job to the place of *Access Tokens* and 1 job to the delay node to execute iteratively.

We simulate the above model with JMT to validate the analytical model. When considering an equal number of users and tokens, we assume the total users as 2, 5, 10, 20, 50, and 100. When considering the number of tokens is half that of the users, we assume the total users as 2, 4, 8, 10, 40, and 100. The service rate μ_h and μ_m are set as 1 and 0.5 respectively. The

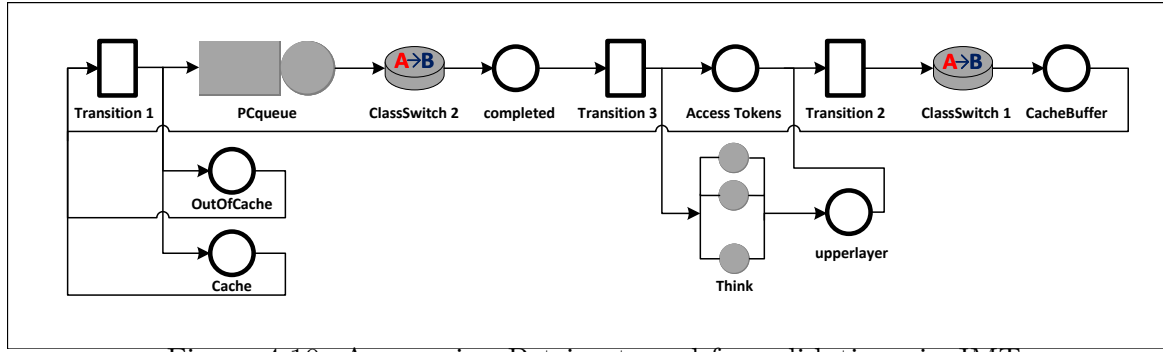


Figure 4.10: A queueing Petri net used for validation via JMT

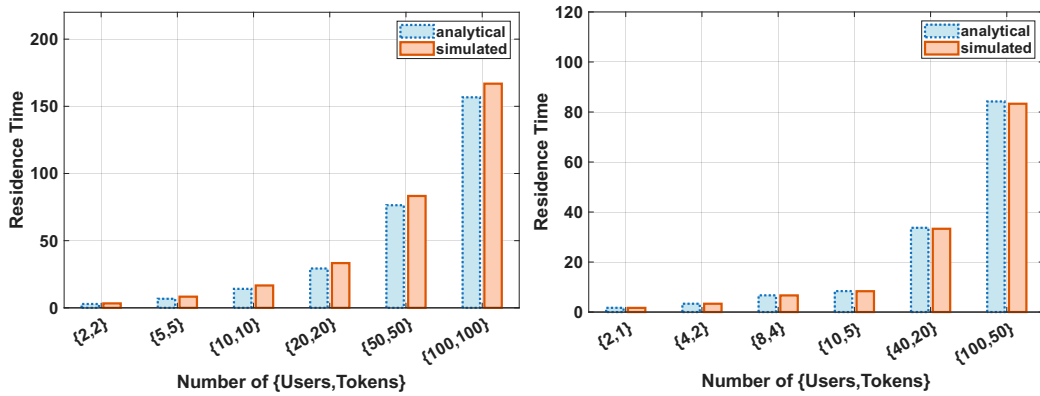


Figure 4.11: Comparison of average residence time between analytical and simulated methods

performance metric we select is the residence time at the queue. This corresponds to the total time between successive visits to the think time station, since the cache replacement transitions are instantaneous.

The results indicate the differences in residence time between the analytical and simulated model under different combinations of numbers of users and tokens, as shown in Figure 4.11. It shows that the differences are not exceeding 8.26% of residence time in all cases, which proves the high accuracy of the generalized LCQ model. The simulation model adheres to rules governing cache replacement algorithms, ensuring accurate emulation of cache behavior. Our approach, while not directly employing CTMC due to the potential for state space explosion, draws upon its principles to closely mirror these dynamics with a notable degree of accuracy.

4.5.2 Experimental Evaluation

Problem Statement

The combination of microservice architecture with edge computing has attracted much attention [WGZ⁺19]. In practice, the processing of many microservice applications requires both computation and data at edge nodes. To evaluate the performance of applications of this kind, we illustrate the benefits of our methodology by considering a cluster of M edge nodes connecting N users in the edge computing system. Each edge node provides multiple services in co-located edge servers and is equipped with a shared cache node that stores data for these services.

Each user offloads several jobs to the connected edge node to be served. We assume the maximum number of running jobs to be K , and the total number of services to be C . When a job is processed, it first checks if the required data is available in the shared cache node. The shared cache node follows the RR replacement policy and has a capacity of q . If so, the data is retrieved from the cache, and then the job is served by the corresponding services provided by the edge node. If not, the data has to be retrieved from other cache nodes or remote data centers with a longer processing time before being served.

In the design and deployment of such microservice placement applications, minimizing computational overhead while satisfying the constrained resources is paramount. To address this, we consider the optimization program same in (4.2), but adopting our proposed ICQ model, aiming at predicting the minimized response time cost incurred by the application. This serves as a baseline for efficient evaluation of design-time applications, addressing data caching and processing simultaneously. The optimization process is conducted as part of the initial design phase or during periodic design updates, depending on factors such as changes in application requirements. The program complexity is dominated by the number of variables, which grows as $O(M \cdot K)$.

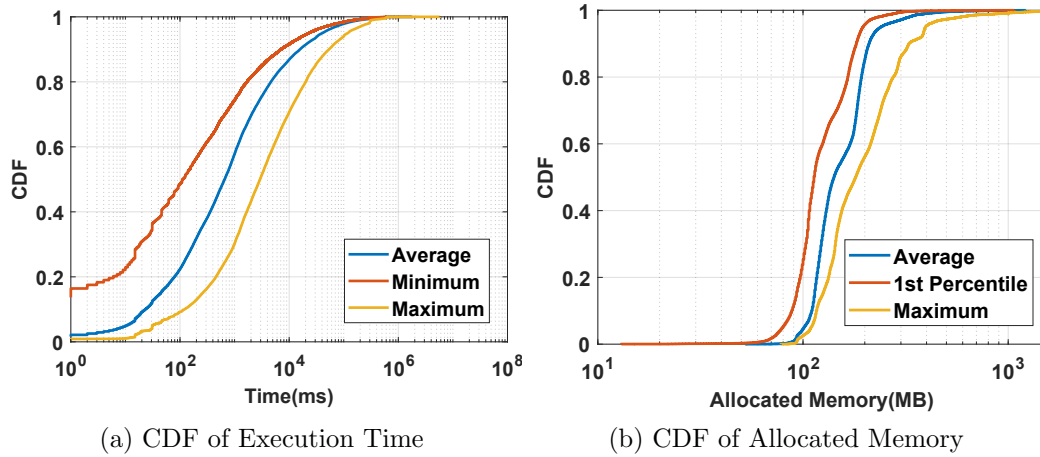


Figure 4.12: Characteristics of the selected random subset of the real-world Azure traces

Evaluation Results

To evaluate our proposed ICQ models to real workloads, we perform a trace-driven simulation to demonstrate the system applicability in principle to production systems. The trace is collected from a subset of applications running on Azure Functions in 2019 [SFG⁺20]. We focus on a random subset of the trace, which contains 34385 application ids and the corresponding function ids belonging to the same application. The invocation rate and the distribution of execution time for each function as well as the distribution of memory usage for each application are also given in the trace.

Figure 4.12(a) shows the cumulative distribution of the minimum, average, and maximum execution time for all functions. We can see that 50% of the functions have average and maximum execution time less than 0.7s and 3s. 90% of the functions execute at most 50s and 95% of the functions take less than 50s on average. Figure 4.12(b) presents the cumulative distribution of the 1st percentile, average, and maximum virtual memory reserved for all applications. Each application calls for one or more functions. We can observe that 99% of the applications consume no more than 400MB on average and 99% of the applications are allocated at most 1000MB.

To simulate the trace-driven data, we implement our proposed ICQ model by LINE [Cas20]. An example of the generalized ICQ model generated by LINE is shown in Figure 4.13. The

Table 4.2: Parameter settings for the evaluation of the ICQ model

Definition	Parameter	Range of Value
number of edge nodes	M	2, 4, 8, 16
number of users	N	5, 15, 25
number of services	C	10, 20, 40
node cache capacity (MB)	q	250, 750, 1024
average total size of items (GB)	p	0.5, 1, 2, 4, 8
Zipf parameter	η	0.6, 1.0, 1.4
number of replications	Q	30

For each combination of the parameters, we generate 30 models with different random seeds. In each model, different services on each node are added with cache-tasks that specify the size of total items and the allocated cache capacity for each service. The sum of service capacities is equal to the node cache capacity, which guarantees the maximum utilization of the constrained storage for each edge node. We employ the total system response time as the performance measure and compare our scheme to the case of *no cache* and *prefetch all*. The *no cache* scheme deploys no caching module that all the jobs need to wait for the requested contents from origin servers before being processed locally. While the *prefetch all* scheme indicates each edge node caches all the items with a full utilization of the local memory.

From Figure 4.14 (a) and (b), we can see that the total response time increases with the growth of the number of users under same conditions, which attributes to the cumulative queueing time. With same number of users, *no cache* scheme as the baseline exhibits the highest response time, because jobs need to obtain required contents from the origin server. Conversely, *prefetch all* scheme exhibits the lowest response time but ignoring the large memory consumption. Compared to the *prefetch all* scheme, although our proposed scheme shows an increase of the response time at most 35% and 30%, it reduces the memory usage by 500MB and 250MB respectively. This is because our proposed system optimizes the allocation of the cache capacity for each service based on the job scheduling and service placement strategies on different edge nodes, which achieves to maximize the utilization of limited edge resources. Overall, the total response time declines significantly when the number of nodes rises, which is

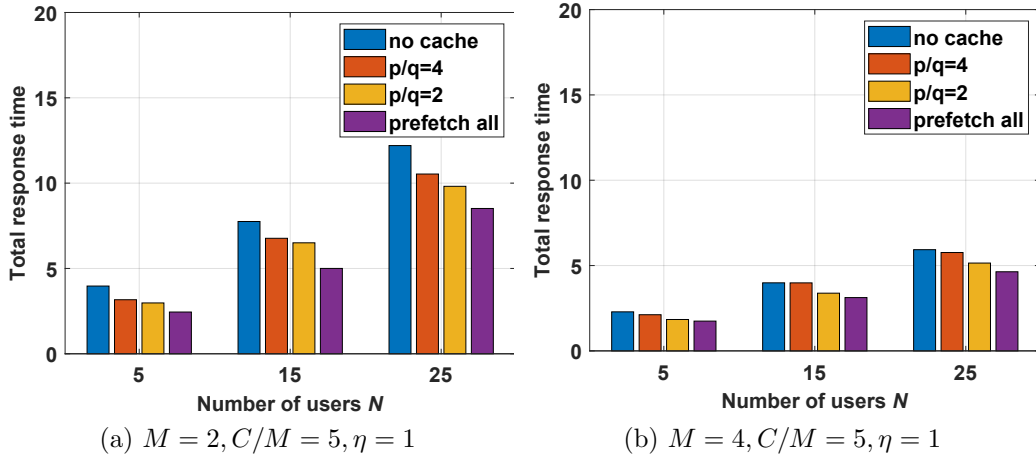


Figure 4.14: Total response time with respect to the increase of the number of users N (Azure dataset)

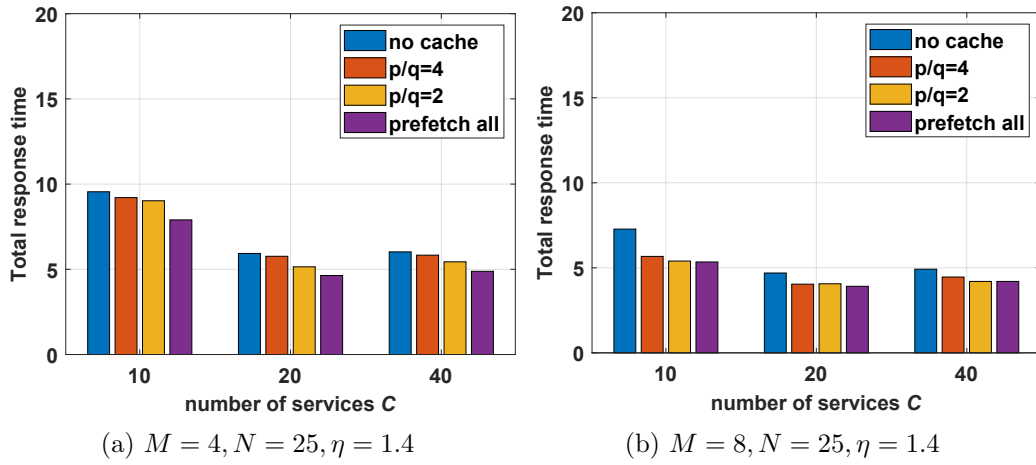


Figure 4.15: Total response time with respect to the increase of the number of services C (Azure dataset)

benefit from more concurrent servers that bring down the waiting time.

From Figure 4.15 (a) and (b), we can see that the total response time decreases with the growth of the number of services, but tends to be stable after C reaches 20. The decreasing trend in the first stage is the case when the number of services is less than the number of users. Under the circumstances, the possibility of users to request for the same service is higher, which results in the longer queueing time for the same server. As the number of services grows to become equal to or larger than the number of users, the possibility of calling the same service reduces, thus leading to the flat trend in the later stages. With same number of services, our proposed scheme only sees an increase of the response time at most 24% and 14%, but with a less memory usage of 500MB and 250MB compared to the *prefetch all* scheme.

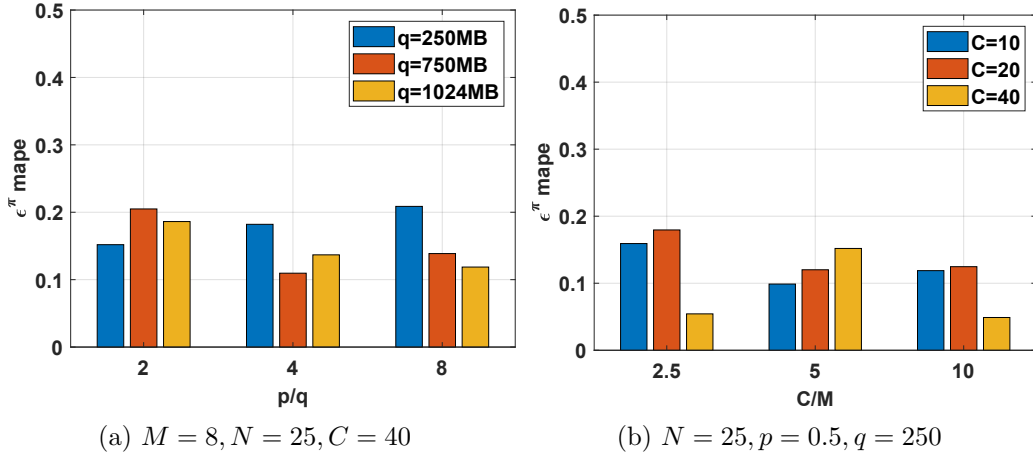


Figure 4.16: Sensitivity to p/q and C/M ratio with $\eta = 1$ (Azure dataset)

Further, we analyze the sensitivity to p/q and C/M ratio on these experiments. We focus on the miss ratio and define the performance metric as

$$\epsilon_{mape}^\pi = \frac{1}{Q} \frac{1}{M} \frac{1}{C} \sum_{k=1}^Q \sum_{j=1}^M \sum_{i=1}^C \left| 1 - \frac{\hat{\pi}_{i,j}}{\pi_{i,j}} \right|, \quad (4.5)$$

where $\hat{\pi}_{i,j}$ is the estimated miss ratio for service i at node j . The services that are not requested by users on each node are not included. From Figure 4.16 (a), we can see that the average percentage error ϵ_{mape}^π for p/q ratio approximately ranges between 0.1 and 0.2. When the node cache capacity is high, the error metric decreases with the size of total items increases. From Figure 4.16 (b), we can observe that the average percentage error ϵ_{mape}^π for C/M ratio approximately ranges between 0.05 and 0.15. The results demonstrate the robustness of our proposed scheme in effectively managing substantial data volumes and diverse levels of edge computing processing capabilities. While Figure 4.16 focuses on $\eta = 1$, additional experiments with $\eta = 0.6$ and $\eta = 1.4$ yield similar error levels. This observation indicates that the accuracy of the model remains stable across different degrees of popularity skew.

Overall, with the increase of the number of users, our proposed scheme indicates an increase of the response time at most 35% and 30%, but reduces the memory usage by 500MB and 250MB, compared to the *prefetch all* scheme. With the increase of the number of services, our proposed scheme shows an increase of the response time at most 24% and 14%, but a less memory usage of 500MB and 250MB, compared to the *prefetch all* scheme. This proves our proposed method

is better than simpler heuristics for resource allocation in edge computing system.

Concerning time and space requirements for the execution of our proposed method, the above experiments are conducted on an AMD EPYC 7302P 16-Core Processor. For each replication, the estimate of memory usage and execution time are no more than 4.45MB and 0.2s, which reflects our proposed method is scalable for multiple workloads.

4.6 Summary

In this chapter, we established a new class of performance models, termed integrated caching-queueing (ICQ) models, that generalize layered queueing networks with caching nodes, enabling the joint modeling of in-memory storage and computation services. We provided an analytical solution for numerically analyzing the proposed class of ICQ models. We further validated the generalized ICQ model by a QPN-based model with JMT and implemented the proposed class of ICQ models within the LINE tool. To evaluate the performance of our proposed ICQ model, we have conducted extensive trace-driven simulations to evaluate the performance of our proposed ICQ model, which shows a better trade-off between the response time and memory consumption.

Chapter 5

Hierarchical Caching Modeling

5.1 Introduction

Caching is a widely used technique to store copies of frequently accessed data items, such as images, files, and scripts, on servers that are closer to end users, thereby reducing latency and improving performance [MAN14]. When a user requests an item that is available in the cache, it results in a cache hit, enabling fast retrieval from the local cache rather than from the original data source. Cache effectiveness is typically measured by the hit rate, which reflects the ratio of cache hits to the total number of requests over a period. Conversely, a cache miss occurs when the requested item is not found in the cache, leading to a costly retrieval from a remote server. These basic caching mechanisms help to reduce response times, alleviate network load, and decrease bandwidth consumption, making them crucial for performance optimization in distributed systems.

In microservice architectures, the role of caching becomes even more critical due to the decentralized and distributed nature of these systems [GD18]. Unlike monolithic architectures where data access and processing occur within a single, tightly coupled system, microservices are independent units that may be spread across geographically distributed nodes. Each service may maintain its own cache, often for its localized data access needs. However, such single-layer caching methods can fall short in optimizing overall system performance. On the one hand,

different services may require access to shared data distributed across multiple caches, which can introduce data inconsistencies across the system. On the other hand, this distributed access can significantly increase network overhead, leading to bottlenecks and latency in data retrieval. Thus, it is essential to ensure that data is efficiently accessible at the required locations when needed, while minimizing latency across a distributed environment [NZP⁺19].

Hierarchical caching [CG20] offers a solution to this problem by structuring cache nodes in a layered architecture across different nodes in the network. In a hierarchical caching system, caches are placed at multiple layers, allowing data to be accessed more efficiently based on its location and access patterns. This approach significantly enhances data locality and reduces the need for repeated retrievals from remote servers, which is particularly beneficial in distributed data processing systems such as data mesh [AL22], where microservices from different domains frequently access domain-specific data products. By establishing a hierarchical caching structure, microservices can retrieve data from the closest cache node in the hierarchy, reducing response times and improving overall system performance.

An illustrative example of hierarchical caching in a microservice architecture is shown in Figure 5.1. This figure depicts a simplified tree-structured caching model designed for a data mesh scenario, where different cache nodes correspond to various in-memory storage locations, such as Memcached or Redis. These caches are organized hierarchically, with data stored closer to the services that require it, thus reducing access times. This abstraction of the data mesh architecture as a tree allows for tractable reasoning on data placement and access paths, capturing essential aspects of access locality and capacity constraints.

To enable the analysis of such hierarchical caching, we propose a novel analytical model, which offers a flexible formalism for describing data placement across distributed cache nodes. The model defines in-memory storage nodes as cache lists and introduces the concept of data item access paths, which are crucial for enforcing organizational, regulatory (*e.g.*, privacy-related), and geographical constraints on data placement. We derive exact solutions for the proposed model under random replacement (RR) policy and a time-to-live (TTL) approximation for the least recently used (LRU) policy. We conduct extensive experiments to validate and evaluate

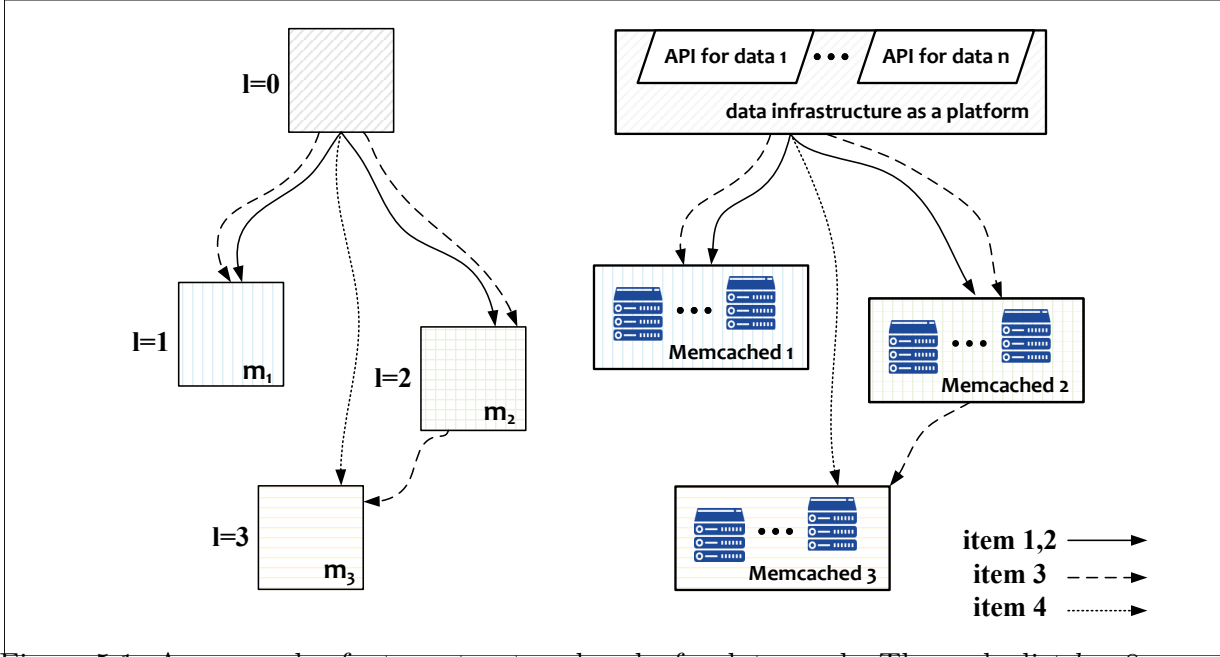


Figure 5.1: An example of a tree-structured cache for data mesh. The cache list $l = 0$ represents out-of-cache data items.

our hierarchical caching model. Overall, our contributions can be summarized as follows:

- We present a generalized analytical model for hierarchical caching, designed to optimize data placement in distributed microservice architectures.
- We formulate a product-form solution for RR policies and a TTL approximation for LRU policy analysis, effectively capturing cache behaviors in hierarchical systems.
- We validate the accuracy and efficiency of the model through simulations and real-world YouTube traces.
- We evaluate the proposed models through real-world Alibaba traces, showing a performance improvement of up to 37% in data mesh scenarios.

The rest of this chapter is organized as follows. In Section 5.2, we present the fundamentals of caching modeling, focusing on hierarchical data distribution and exploring different replacement policies. Section 5.3 introduces the analytical methods employed in this chapter, where we provide the exact solution for the $RR-a(\mathbf{m})$ policy and discuss the TTL approximation for the $LRU-a(\mathbf{m})$ policy. In Section 5.4, we validate the hierarchical caching model through simu-

lations and real-world YouTube traces. Section 5.5 explores the application of the hierarchical caching model in data mesh architectures. Finally, Section 5.6 presents a summary.

5.2 Caching Modeling

5.2.1 Hierarchical Data Distribution

In the analytical cache models developed in this chapter, we consider a total of M cache nodes distributed in h layers arranged in a tree topology. Each cache node has a capacity of m_l data items, $l = 1, 2, \dots, M$, and each data item can be stored at only one cache node at a time. The total cache capacity is $m = \sum_{l=1}^M m_l$ items. In total, data consists of I items accessible through requests, *e.g.*, via REST APIs. The size of each data item is denoted as $s_i, i = 1, 2, \dots, I$, and the total size of data items is $n = \sum_{i=1}^I s_i$. Node $l = 0$ is a virtual cache node representing all remote data sources, *i.e.*, virtually storing all data items outside the cache nodes, with a total capacity of $m_0 = n - m$ items. If a requested item is not found in any reachable cache node along the access path, the request is forwarded to node $l = 0$, which emulates a fallback to the origin server, as observed in practical systems.

We describe the data distribution across the cache nodes using Markov chains, however, this implies that representing item sizes presents difficulties, since the analysis normally either loses product-form solution or becomes computationally difficult [BGDMT06]. Hence, in what follows, we take the common assumption that the cache model abstracts a simplified scenario where items have identical sizes ($s_i = 1, \forall i$). Thus, we keep track of the presence of an item at a particular node, but not of its size. This restriction is lifted later by accounting for total available cache capacity directly in the optimization programs used to distribute data.

In accordance with the independent reference model (IRM), we specify the requests to data items by the users as u independent request streams. Stream v is assumed to be a Poisson process with rate $\lambda_{vk}(l)$ for data item k stored in cache node l , where $v = 1, \dots, u; k = 1, \dots, I; l = 1, \dots, M$. These exogenous requests may originate at any cache node, including both leaf

and intermediate nodes, depending on the geographical or organizational scope of the stream.

5.2.2 Replacement Policies with Access Paths

Cache nodes in our model can exchange data items to improve the locality of access. A data item can be stored at a single cache node at a time. Each cache node l has only one parent cache node that can exchange data items with, but a set of child nodes, denoted by sets $p(l)$ and $\mathbf{b}(l)$ respectively. If $\mathbf{b}(l) = \emptyset$, node l is a leaf node.

The movement of items across the hierarchical cache is regulated by the cache replacement policy. Multiple replacement policies exist for individual caches, *e.g.*, random replacement (RR), first in first out (FIFO), least recently used (LRU). If the cache is partitioned in nodes (*i.e.*, lists) and there is control over the probability that an item moves from a parent node to its children, the generalizations of these replacement policies are called RR-c($\mathbf{m}$), FIFO-c($\mathbf{m}$), LRU-c($\mathbf{m}$) [CG20].

Due to their practicality, we focus on RR-c($\mathbf{m}$) and LRU-c($\mathbf{m}$) policies and propose in this chapter to generalize them to describe scenarios such as the one in Figure 5.1, in which items may go directly from the source (node $l = 0$) to any of the cache nodes (*e.g.*, node 2). This is not allowed in existing policies such as RR-c($\mathbf{m}$) and LRU-c($\mathbf{m}$), but it is a requirement in systems where data may be requested directly from the data sources at any location, *e.g.*, close to the team using that data product.

To resolve these issues, we introduce the concept of access path $\mathcal{P}_{vi}(j)$, *i.e.* a sequence of nodes visited by a data item i that starts outside the cache until it is promoted to node j under requests from stream v only. If data item i cannot reach node j under requests coming from stream v , we set $\mathcal{P}_{vi}(j) = \emptyset$. With the definition of access paths, the resulting variants of the above replacement policies are denoted as RR-a($\mathbf{m}$) and LRU-a($\mathbf{m}$). The two policies are defined as follows.

In both policies, if a cache hit for item k in a non-leaf node l occurs, then item k swaps its position with an item in one of its child nodes j in accordance with the following rule:

- $\text{RR-}a(\mathbf{m})$: a child node j is chosen with probability $c_{vk}(l, j), j \in \mathbf{b}(l)$ and an item k' within j is chosen uniformly at random for eviction. Item k then replaces item k' in node j and, vice versa, item k' is stored in place of item k in node l .
- $\text{LRU-}a(\mathbf{m})$: similar to $\text{RR-}a(\mathbf{m})$, except that item k' is chosen as the least recently used item in node j .

If a cache miss occurs, then item k evicts an item from node l with probability $c_{vk}(0, l)$ to replace it. Note that the ability to specify $c_{vk}(0, l)$ in our models goes beyond caches studied under $\text{RR-c}(\mathbf{m})$ and $\text{LRU-c}(\mathbf{m})$, where items triggering an eviction are restricted to enter the cache at node $l = 1$.

An example of access paths can be illustrated by the case presented in Figure 5.1 with $\mathbf{m} = (1, 1, 1)$ and $n = 4$. In the diagram, an arrow for data item i between nodes l and l' indicates that there exists a stream v such that $c_{vi}(l, l') > 0$. In-memory storage locations that are not reachable according to the access path determined by the geography-based constraints are not listed. According to our definitions, there are access paths, *e.g.*, $\mathcal{P}_{v3}(3) = \{(0, 2), (2, 3)\}$ and $\mathcal{P}_{v4}(3) = \{(0, 3)\}$, and unreachable paths, *e.g.*, $\mathcal{P}_{v1}(3) = \mathcal{P}_{v2}(3) = \mathcal{P}_{v4}(1) = \mathcal{P}_{v4}(2) = \emptyset$.

5.3 Analytical Methods

To analyze the performance of the class of hierarchical cache models we consider, we first generalize an exact solution in [CG20] for the $\text{RR-}a(\mathbf{m})$ policy to enable each data item to directly enter cache nodes at different layers. Then, considering the high computational complexity of calculating large-scale LRU caches, we generalize existing time-to-live (TTL) approximations to the $\text{LRU-}a(\mathbf{m})$ policy, leveraging the fact that prior work has obtained asymptotically exact results for $\text{LRU}(\mathbf{m})$ in linear caches [GVH16].

5.3.1 Exact Solution for RR- $a(\mathbf{m})$ Policy

We first focus on RR- $a(\mathbf{m})$ policy and leverage continuous-time Markov chain (CTMC) to model the replacement policy. We define the state vector $\mathbf{s} = [s(k, j)]$ to represent the index of the data item stored in the k -th position of cache node j . The reachable state space is $\mathcal{S}$ ($\mathbf{s} \in \mathcal{S}$). Each state represents a unique placement of data items across the cache nodes, ensuring that each item is stored at only one node.

Example. Consider a system with two cache nodes, where node 1 has capacity $m_1 = 2$ and node 2 has capacity $m_2 = 1$. Assume three data items (IDs 1, 2, 3) are to be placed across the two cache nodes without duplication. A state can be denoted as

$$\mathbf{s} = [\mathbf{s}(1, 1), \mathbf{s}(2, 1), \mathbf{s}(1, 2)],$$

where $\mathbf{s}(1, 1)$ and $\mathbf{s}(2, 1)$ represent the indices of the data items stored at the first and second positions of cache node 1, respectively, and $\mathbf{s}(1, 2)$ represents the index of the data item stored at the single position of cache node 2.

The state space $\mathcal{S}$ consists of all permutations of the three data items across the available cache positions. There are $3! = 6$ valid configurations:

$$\begin{aligned} \mathcal{S} = \{ & [1, 2, 3], \quad [1, 3, 2], \quad [2, 1, 3], \\ & [2, 3, 1], \quad [3, 1, 2], \quad [3, 2, 1] \}. \end{aligned}$$

Thus, we set $\pi(\mathbf{s})$ the equilibrium probability of the model. A product-form result now holds.

Theorem 4. Assume that if $\mathcal{P}_{vk}(j) = \emptyset$ then $\lambda_{vk}(j) = 0$ and otherwise that access paths are stream-independent, i.e., $\mathcal{P}_{vk}(j) = \cup_{v=1}^u \mathcal{P}_{vk}(j) = \mathcal{P}_k(j)$. Then, RR- $a(\mathbf{m})$ policy admits the equilibrium distribution

$$\pi(\mathbf{s}) = \frac{\prod_{j=0}^M \prod_{k=1}^{m_j} \alpha_{s(k,j)j}}{E(\mathbf{m})}, \quad (5.1)$$

for all recurrent states $\mathbf{s} \in \mathcal{S}$, where we define the normalizing constant

$$E(\mathbf{m}) = \sum_{\mathbf{s} \in \mathcal{S}} \prod_{j=0}^M \prod_{k=1}^{m_j} \alpha_{s(k,j)j}, \quad (5.2)$$

and where

$$\alpha_{kj} = \begin{cases} \prod_{(l,l') \in \mathcal{P}_k(j)} \sum_{v=1}^u \lambda_{vk}(l) c_{vk}(l, l') & \text{if } j > 0 \\ 1 & \text{if } j = 0 \end{cases}$$

for all items $k = 1, \dots, n$ and cache nodes $j = 0, \dots, M$.

Proof. The argument relies on showing the reversibility of the underlying Markov process, which can be done following very similar steps as the proof of Theorem 1 in [CG20]. The latter proves the product-form solution of RR- $c(\mathbf{m})$, for which the product-form result assumes caches where the parent relationships between nodes (*i.e.*, cache lists) are statically defined and stream-independent. This is also ensured by the assumptions of Theorem 4 as access paths to items are stream-independent. Moreover, while the proof given in [CG20] is for a model that assumes that items enter at cache list 1, the proof does not rely on this assumption, hence the same argument proves the reversibility of the Markov process underlying the RR- $a(\mathbf{m})$ policy. □

We refer to α_{kj} as the *cost factor* for item k to access node j . The cost factor differs from the access cost in that it accounts for both the costs and the item request rates involved in item i movement from outside the cache into node j . In this way, the normalizing constant $E(\mathbf{m})$ can be computed same as the methods in [CG20]. An illustrative example is given as follows.

We consider again the model shown on the left of Figure 5.1, assuming $n = 4$ items, $M = 3$ nodes, and $\mathbf{m} = (1, 1, 1)$. We consider $u = 2$ streams that both access items at rate $\lambda_{vi}(j)$. If $\mathcal{P}_{vi}(j) = \emptyset$, item i cannot reach node j under requests from stream v and we indicate this by setting $\lambda_{vi}(j) = 0$. In this setting, we index the lists by $j = 0, 1, 2, 3$, where $j = 0$ denotes the virtual out-of-cache list, and $j = 1, 2, 3$ correspond to the three physical cache nodes, each with

unit capacity. Each item is stored in exactly one of these lists at any given time. Matrices λ_1 and λ_2 shown below specify the request rates per stream v , item i , and list j :

$$\lambda_1 = [\lambda_{1i}(j)] = \begin{bmatrix} 4 & 10 & 8 & 0 \\ 9 & 10 & 4 & 0 \\ 6 & 6 & 9 & 3 \\ 5 & 0 & 0 & 8 \end{bmatrix}, \quad \lambda_2 = [\lambda_{2i}(j)] = \begin{bmatrix} 8 & 3 & 2 & 0 \\ 6 & 2 & 4 & 0 \\ 2 & 8 & 5 & 2 \\ 10 & 0 & 0 & 7 \end{bmatrix}.$$

Each matrix $[\lambda_{vi}(j)]$ is organized such that rows index items $i = 1, 2, 3, 4$, and columns index target lists $j = 0, 1, 2, 3$. The entry $\lambda_{vi}(j)$ represents the request rate from stream v for item i when it is located in list j . A zero value indicates that item i cannot be requested at list j under stream v . For example, $\lambda_{14}(0) = 5$ indicates that stream 1 requests item 4 at a rate of 5 requests/second when it resides in list 0.

Let $C_{vi} = [c_{vi}(l, j)]$ be the access cost matrix for stream v and item i . Each matrix C_{vi} specifies the access costs from source list l to target list j ($l, j = 0, 1, 2, 3$). A non-zero entry indicates an allowed transition along the access-path graph, whereas a zero entry corresponds to a disallowed move. For instance, $C_{14}(0, 3) = 1$ implies that item 4 can be inserted into list 3 from list 0, while all other zero entries in C_{14} reflect restricted access paths.

$$C_{11} = \begin{bmatrix} \frac{1}{2} & \frac{1}{6} & \frac{1}{3} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad C_{12} = \begin{bmatrix} 0 & 1 & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad C_{13} = \begin{bmatrix} 0 & \frac{7}{10} & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad C_{14} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$C_{21} = \begin{bmatrix} \frac{2}{3} & \frac{1}{9} & \frac{2}{9} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad C_{22} = \begin{bmatrix} 0 & 1 & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad C_{23} = \begin{bmatrix} 0 & \frac{7}{10} & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad C_{24} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

With the request rate and cost matrices specified, we now characterize the dynamics of the caching process using a CTMC defined over the recurrent states of the system. We enumerate all recurrent states according to a fixed descending lexicographic ordering. Each state $\mathbf{s}$ is represented as a tuple $[\mathbf{s}(0), \mathbf{s}(1), \mathbf{s}(2), \mathbf{s}(3)]$, where $\mathbf{s}(j)$ indicates the index of the item stored in list j .

Although there are $4! = 24$ possible permutations of 4 items across the four lists, the access-path constraints (as illustrated in Figure 5.1) restrict the reachable state space. Specifically, a state is admissible only if every item placement obeys these access-path constraints. Under these conditions, exactly six states are recurrent and reachable:

State 1: $[3,2,1,4]$, State 2: $[3,1,2,4]$, State 3: $[2,3,1,4]$,

State 4: $[2,1,3,4]$, State 5: $[1,3,2,4]$, State 6: $[1,2,3,4]$.

Thus, the irreducible Markov process for the recurrent states has an infinitesimal generator

$$Q = \begin{bmatrix} -8 & 0 & \frac{28}{5} & 0 & 0 & \frac{12}{5} \\ 0 & -8 & 0 & \frac{12}{5} & \frac{28}{5} & 0 \\ 15 & 0 & -\frac{39}{2} & 0 & \frac{9}{2} & 0 \\ 0 & \frac{9}{2} & 0 & -\frac{39}{2} & 0 & 15 \\ 0 & \frac{16}{9} & \frac{32}{9} & 0 & -\frac{16}{3} & 0 \\ \frac{32}{9} & 0 & 0 & \frac{16}{9} & 0 & -\frac{16}{3} \end{bmatrix}.$$

The rows and columns of the matrix Q are indexed by these states in the given order. Off-diagonal elements $Q(i, j)$ represent transition rates from state i to state j , while diagonal elements are set such that each row sums to zero. For example, $Q(6, 1) = \frac{32}{9}$ is the transition rate between state $[1, 2, 3, 4]$ and state $[3, 2, 1, 4]$. This transition corresponds to a request for item 1, which is initially outside the cache (stored in list 0) in state 6. Upon this request, item 1 is inserted into list 2, replacing item 3, which is evicted to list 0. The transition is allowed by the access paths defined in Figure 5.1, specifically via the path $(0 \rightarrow 2)$. The transition rate reflects the cumulative effect of such insertions across both request streams.

We determine numerically from the Markov process the following equilibrium state probabilities

$$\boldsymbol{\pi} = \begin{bmatrix} 0.3635 & 0.0545 & 0.1357 & 0.0291 & 0.1718 & 0.2254 \end{bmatrix},$$

where $\boldsymbol{\pi} = [\pi(\mathbf{s})]$. We now verify that this is the same result that one would obtain using the product-form expression. Using the definitions in Theorem 4 we first compute the following cost factors

$$\alpha = [\alpha_{ij}] = \begin{bmatrix} 1 & \frac{16}{9} & \frac{32}{9} & 0 \\ 1 & 15 & \frac{9}{2} & 0 \\ 1 & \frac{28}{5} & \frac{12}{5} & \frac{84}{5} \\ 1 & 0 & 0 & 15 \end{bmatrix},$$

where rows index items $i = 1, \dots, n$ and columns index lists $j = 0, \dots, M$.

With the above parameterization, we obtain from (5.1) the following state probabilities:

<i>state $\mathbf{s}$</i>	<i>probability $\pi(\mathbf{s})$</i>	<i>value</i>
[3, 2, 1, 4]	$\alpha_{21}\alpha_{12}\alpha_{43}/E(\mathbf{m})$	$\frac{1200}{3301} = 0.3635$
[3, 1, 2, 4]	$\alpha_{11}\alpha_{22}\alpha_{43}/E(\mathbf{m})$	$\frac{180}{3301} = 0.0545$
[2, 3, 1, 4]	$\alpha_{31}\alpha_{12}\alpha_{43}/E(\mathbf{m})$	$\frac{448}{3301} = 0.1357$
[2, 1, 3, 4]	$\alpha_{11}\alpha_{32}\alpha_{43}/E(\mathbf{m})$	$\frac{96}{3301} = 0.0291$
[1, 3, 2, 4]	$\alpha_{31}\alpha_{22}\alpha_{43}/E(\mathbf{m})$	$\frac{567}{3301} = 0.1718$
[1, 2, 3, 4]	$\alpha_{21}\alpha_{32}\alpha_{43}/E(\mathbf{m})$	$\frac{810}{3301} = 0.2254$

where we omitted the terms for virtual list 0 since $\alpha_{30} = \alpha_{20} = \alpha_{10} = 1$. The values in the table match the ones obtained numerically for $\boldsymbol{\pi}$.

The result generalizes the prior work on tree-structured caches accounting for the access paths of items. In particular, the above result states that if the locations at which a data item can be stored are the same for all users of that data item, then it is possible to tractably

analyze the caching model using a product-form solution. The performance measure for the analytical model is denoted by the total miss ratio $p_{miss}^{(v)}$ for stream v , which can be iteratively approximated by the fixed point iteration method (FPI) [CG20] as:

$$p_{miss}^{(v)} = 1 - \sum_{k,j} \pi_{kj}^{(t)}, \quad k = 1, 2, \dots, n; j = 1, \dots, M, \quad (5.3)$$

where t is an iteration number and $\pi_{kj}^{(t)} = \alpha_{kj} \xi_j^{(t)} \left(1 + \sum_{l=1}^h \alpha_{kl} \xi_l^{(t)}\right)^{-1}$ represents the probability of item k residing in list j at equilibrium and $\xi_j^{(t)} = m_j \left(\sum_{k=1}^n \alpha_{kj} (1 - \sum_{j=1}^h \pi_{kj}^{(t-1)})\right)^{-1}$. Initially, FPI sets $\pi_{kj}^{(0)} = 0$ for all items k if $j > 0$, and $\pi_{k0}^{(0)} = 1$ otherwise. The iteration continues until either the miss ratio $p_{miss}^{(v)}$ has converged or the maximum number of iterations has been reached. The corresponding hit ratio for stream v is given by $p_{hit}^{(v)} = \lambda_v - p_{miss}^{(v)}$.

5.3.2 TTL Approximation for LRU- $a(\mathbf{m})$ Policy

We now focus on the LRU- $a(\mathbf{m})$ replacement policy. Such caches are difficult to analyze even in the basic LRU($\mathbf{m}$) setting, for which prior work has used TTL approximations [GVH16]. The TTL approximation assumes an infinite capacity for each node to approximate the LRU($\mathbf{m}$) policy, thus evicting the data items based on the expiration time rather than the usage history. In this section, we illustrate how to extend the formulas in [GVH16] to encompass access paths, thus generalizing the approximation to the LRU- $a(\mathbf{m})$ policy, *i.e.*, to non-uniform access costs and access paths.

Under TTL approximation, each cache node l stores data items for a deterministic time period of duration T_l , *i.e.*, the time-to-live value. If a data item k is not accessed until the expiration of its time to live, the item moves from node l to its parent node $p(l)$ and the corresponding cost factor is

$$\alpha_k(l, p(l)) = e^{-\sum_{v=1}^u \lambda_{vk} T_l}, \quad l = 1, \dots, M. \quad (5.4)$$

If data item k is requested before exceeding T_l in non-leaf node l , it is promoted to its child node j with probability $c_{vk}(l, j)$. Otherwise, it keeps itself in leaf node j and resets its time-to-live.

The cost factor of data item k moving from current node l to node j is represented as

$$\alpha_k(l, j) = 1 - e^{-\sum_{v=1}^u c_{vk}(l, j) \lambda_{vk}(l) T_l}, \quad l = 0, \dots, M, \quad (5.5)$$

where $l = p(j)$ or $l = j$ is a leaf node. $c_{vk}(l, j)$ is the access probability of data item k to move from node l to node j , satisfying $\sum_{j \in \mathcal{b}(l)} c_{vk}(l, j) = 1$ and $0 \leq c_{vk}(l, j) \leq 1$. When $l = j$ is a leaf node, $c_{vk}(l, j) = 1$.

Based on the above definition of non-uniform access, we utilize an $M + 1$ states discrete-time Markov chain (DTMC) to model the LRU- $a(\mathbf{m})$ replacement policy for data item k at time t^n , denoted by $(Y_k)_n, n \geq 0$, where Y_k is the node id keeping data item k . The state space is represented by $\{0, \dots, M\}$ and we thus give the steady state of data item k in cache node j as

$$\pi_{k,j} = \frac{\prod_{(l,l') \in \mathcal{P}_k(j)} \alpha_k(l, l')}{\Omega(\mathbf{m})}, \quad (5.6)$$

where $\Omega(\mathbf{m})$ is the normalizing constant ensuring that $\sum_{k,j} \pi_{k,j} = 1$, and $\mathcal{P}_k(j)$ denotes all possible paths for data item k to access node j . The result generalizes prior work in [GVH16] to analyze the steady state of caches with non-uniform access paths for data items, thus offering a solution for layered data access under LRU- $a(\mathbf{m})$ policy.

Further, to solve the deterministic time T_l that provides an approximation to LRU- $a(\mathbf{m})$, we apply a set of fixed point equations for $T_l, l = 1, \dots, M$, similar to the iterative expressions in [GVH16, CTW02]. The equations utilize the average time that data item k spends in node j to derive the probability of data item k residing in node $j \geq 1$ at a random instant. These probabilities are summed up to equal the capacity of each cache node. The key performance measure for the model is denoted by the total miss ratio $p_{miss}^{(v)}$ for stream v as:

$$p_{miss}^{(v)} = \sum_k \lambda_{vk}(0) \left(1 - \sum_{j=1}^M \gamma_{k,j} \right), \quad (5.7)$$

where $\gamma_{k,j} = \pi_{k,j} \bar{t}_{k,j} \left(\sum_{i=0}^M \pi_{k,i} \bar{t}_{k,i} \right)^{-1}$ denotes the probability of item k in node $j \geq 1$ at a random instant and $\bar{t}_{k,j}$ is the average time of item k spent in node j . When $j = 0$, $\bar{t}_{k,j} = \lambda_{vk}^{-1}(j)$. When

$j = 1, 2, \dots, M$, it is $\bar{t}_{k,j} = (1 - e^{-\lambda_{vk}(j)})(\lambda_{vk}(j))^{-1}$. The corresponding hit ratio is then given by $p_{hit}^{(v)} = \lambda_v - p_{miss}^{(v)}$.

5.4 Model Validation

5.4.1 Hierarchical Caching Model Validation

We validate our proposed analytical model for hierarchical cache modeling considering the LRU- $a(\mathbf{m})$ replacement policy. For caches employing the RR- $a(\mathbf{m})$ replacement policy, we can draw upon the validation process outlined in [CG20], leveraging FPI and singular perturbation approximation (SPA). Since our model generalizes prior research [GVH16], which has achieved asymptotically exact results for linear caches, we extend the validation to encompass parallel and tree structures respectively, catering to a wider array of practical scenarios.

For parallel-structured caches, which can be regarded as a special case of tree-structured caches. We consider a total number of 10, 100, 500, and 1000 data items. The ratio of the total cache capacity to total data items β is set as 0.4 and 0.8. We assume that data item popularity follows a Zipf distribution with parameter 0.8, which falls within the recommended range of 0.64 to 0.85 for modeling request patterns, as identified in [BCF⁺99, RM21], and which has been widely adopted in trace-driven caching studies [GH15, CG20]. The results of comparing the miss ratio for different cases are illustrated in Figure 5.2. We can observe that the errors between simulated and analytical results are no larger than 5% for all cases. Moreover, the error declines with larger-scale data items under the same number of lists or β , which is conducive to analyzing practical situations.

Next, for tree-structured caches, *i.e.*, different lists are connected linearly and/or in parallel, we consider a total number of 10, 100, 500, and 1000 data items obeying Zipf distribution with a parameter of 0.8. The ratio β is also set as 0.4 and 0.8. From Figure 5.3, it can be seen that, for all cases, the errors between simulated and analytical results do not exceed 6%. Under the same number of lists, the error is declined with the increase of the number of data items. Under

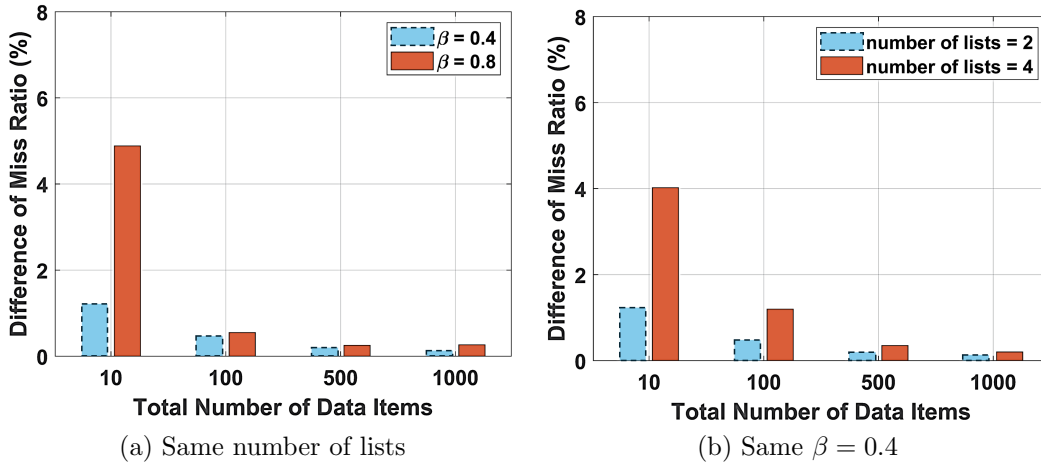


Figure 5.2: Comparing the miss ratio of parallel-structured caches between analytical and simulated methods

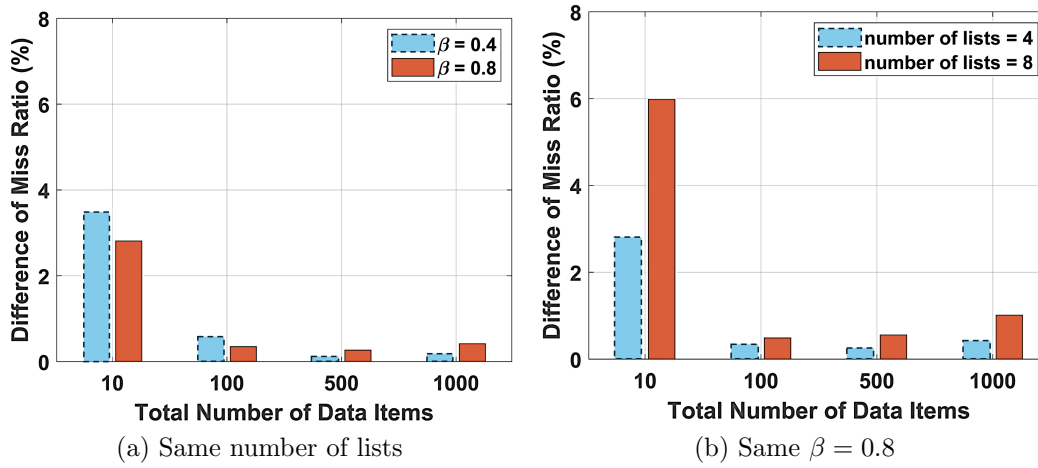


Figure 5.3: Comparing the miss ratio of tree-structured caches between analytical and simulated methods

the same ratio β , the error is increased when there are more lists.

Overall, our analytical model for hierarchical caches shows high accuracy and generality in various caching architectures.

5.4.2 Validation Under Real Traffic

In this section, we illustrate our analytical model using trace-driven simulation. Specifically, we use the temporally-correlated 2008 YouTube trace from [ZSGK09], comprising 611,968 requests. This is a representative of real multimedia content that may realistically be cached near

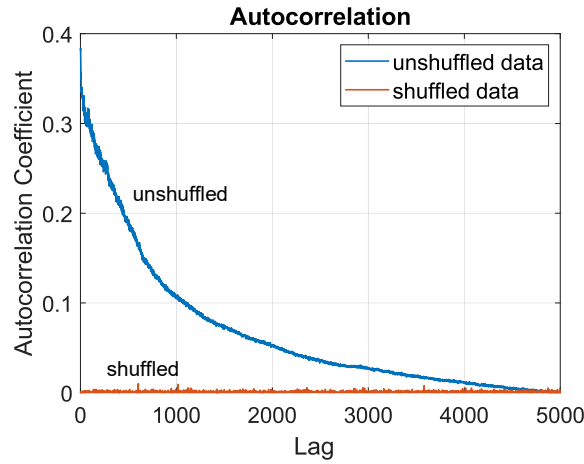


Figure 5.4: Autocorrelation of the original and shuffled 2008 Youtube trace

Table 5.1: Comparing the miss ratio difference between the analytical method and trace-driven simulation. β is the ratio of cache capacity to the number of items.

number of lists	list capacity $m_l, l = 1, 2, \dots$	β	Miss Ratio			Absolute Difference		Relative Difference	
			analytical	simulated (unshuffled)	simulated (shuffled)	unshuffled	shuffled	unshuffled	shuffled
2	1516	0.01	0.4517	0.4783	0.4535	0.0266	0.0018	5.88%	0.39%
4	758		0.4411	0.4725	0.4569	0.0314	0.0157	7.11%	3.55%
2	3032	0.02	0.4310	0.4613	0.4336	0.0303	0.0026	7.03%	0.60%
4	1516		0.4184	0.4541	0.4402	0.0357	0.0218	8.53%	5.21%
2	7583	0.05	0.3907	0.4253	0.3954	0.0346	0.0047	8.85%	1.20%
4	3791		0.3756	0.4166	0.4081	0.0410	0.0325	10.91%	8.65%
2	15166	0.10	0.3453	0.3832	0.3525	0.0379	0.0072	10.97%	2.08%
4	7583		0.3291	0.3741	0.3703	0.0450	0.0412	13.67%	12.51%

the edge, *e.g.* for download by user mobile devices. The chosen trace encompasses 303,332 distinct items and originates from 16,337 client IPs, each mapped to a single stream v . The autocorrelation of the unshuffled and shuffled Youtube trace data is shown in Figure 5.4, showing a high positive correlation for the original (unshuffled) data. The shuffled trace retains the same distribution of inter-arrival times as in the original trace, but removes the autocorrelations, thus helping to assess the impact of temporal dependence alone on the applicability of our models, which rely on IRM assumptions.

Our simulations employ a cache with either linear or tree structure under the LRU- $a(\mathbf{m})$ replacement policy. Specifically, for tree structures with 4 nodes, direct connections exist between lists 1-2, 1-3, and 2-4. The ratio β of total cache capacity to the number of readable data items is set at 0.01, 0.02, 0.05, and 0.10. Results are given in Table 5.1.

For each simulation, we calculate the total miss ratio and compare it with the estimates generated by our analytical model. Across all scenarios, the average *difference* in miss ratio for

unshuffled data stands at 0.0353, while for shuffled data, it reduces to 0.0159. These results highlight that the proposed model can display modest differences in miss ratio also under significant violations of the IRM assumption.

5.5 Application: Data Mesh

Following the validation of our hierarchical caching models under both synthetic workloads and real-world traffic patterns, we now explore their deployment in a concrete system setting. Specifically, we examine how these models can be leveraged within data mesh architectures, where the decentralization of data ownership and the heterogeneity of access patterns present new challenges for effective cache coordination and resource allocation.

5.5.1 Introduction

Traditionally, many organizations adopt a monolithic data lake collecting operational data from different domain teams. The data team, responsible for the centralized data lake, is expected to process all the big data to generate deep insights, which can allow multiple teams to make competitive data-driven decisions. However, the efficiency of the data team is hindered. On one hand, the data team needs to adjust data pipelines endlessly for processing variable operational data from different domain teams. On the other hand, the data team spends a lot of time learning domain knowledge so as to make useful analytics.

To resolve the issue, data mesh, a novel paradigm of data architecture, has recently gained attention [Deh19]. It decentralizes data management by shifting the responsibility from a centralized data team to distributed domain teams. Each domain team takes responsibility for their own data, including processing source data, publishing data products, *etc.* This enables each domain team to leverage its specialized knowledge and expertise to deliver more insightful and valuable views [AL22].

A data mesh architecture mainly focuses on 2 components [MCS22]: domains and data infras-

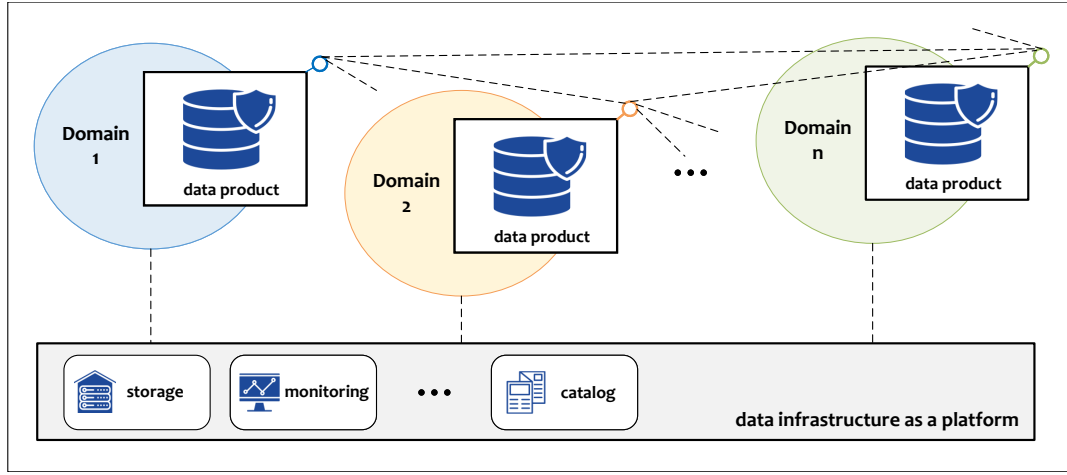


Figure 5.5: Data mesh architecture

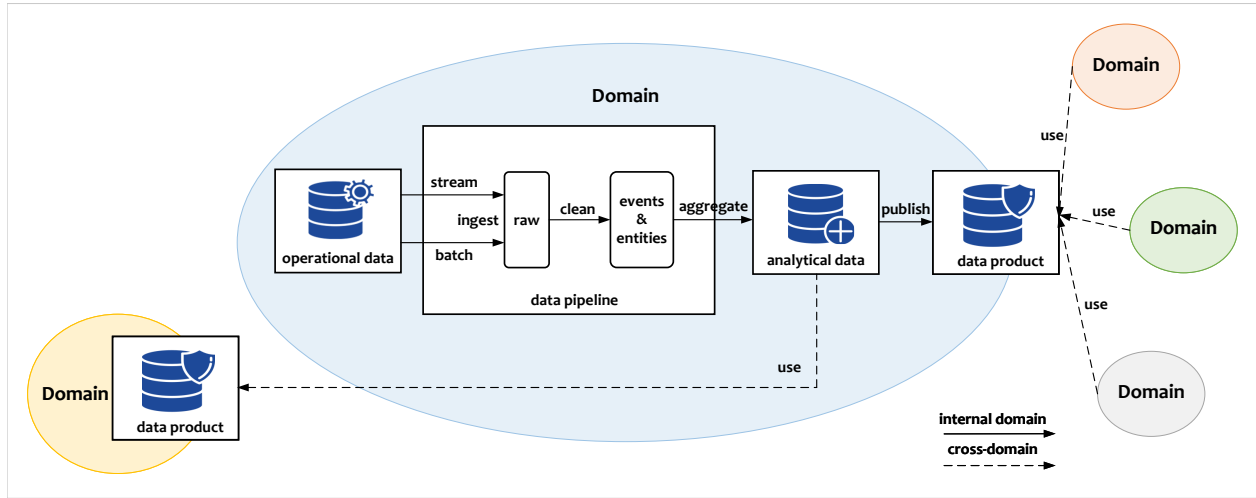


Figure 5.6: Core components of the data flow in each domain

tructure as a platform, as shown in Figure 5.5. Specifically, domains take responsibility for their own data, including processing source data to be analytical data, publishing data products to meet the needs of other domains, *etc.* The data infrastructure is built as a shared and harmonized self-serve platform, which manages source data and provides necessary solutions for each domain.

(1) Domain

In each domain, the data flow consists of 4 core components: operational data, data pipelines, analytical data, and data products, as shown in Figure 5.6.

- First, self-contained systems of each domain, such as web applications and necessary infrastructures, generate operational data.
- Second, data pipelines ingest the generated operational data for a series of processing,

which is generally composed of extract, transform, and load (ETL). The extraction part, *i.e.*, ingestion, can be done in 2 ways: (a) batch processing, in which source data is collected periodically; (b) stream processing, in which data is sourced and manipulated as soon as it is created. After the extraction, the transformation part is tasked with managing ingestion, cleaning, standardization, sorting, deduplication, *etc.*, to generate data assets to be eventually loaded.

- Third, the analytical dataset aggregates not only the processed operational data but also relevant data products provided by other domains. The domain team can apply various methodologies, such as visualization tools and machine learning methods, to these analytical data to make advanced analytics.
- Last, each domain team publishes its own data products based on the analytical data. The published data products can be seen as data sets and accessed by other domains, similar to an API. The forms of the provided data products of each domain are expected to be standardized, such as a CSV file on an AWS S3 bucket, via a federated governance guild.

(2) Data infrastructure as a platform

The self-serve data platform manages data and provides domain-agnostic functionality enabling each domain team to perform analytics and build data products. The shared data platform is expected to support each domain team internally and crosswise:

- Internal-domain: analytical capability. This enables each domain to build analytical models to analyze their own data so as to make data-driven intelligent decisions. To fulfill this, the data platform needs to manage domain-oriented data and provides solutions for each domain.
- Cross-domain: data product capability. This enables each domain not only to build its own data products but also to access data products of other domains. To implement this, on one hand, the data platform should support each domain team to build, publish, and monitor their own data products in separate areas. On the other hand, the platform should set entry points (*e.g.*, APIs) for all decentralized data

products so that each domain can discover and access data products of each other.

5.5.2 Problem Statement

Microservices architecture aligns well with a distributed data mesh approach [MSB⁺23], where each microservice can represent a specific domain team and encapsulate the logic and functionality related to that data domain. By allowing domain teams to independently develop, deploy, and manage their data services, this architecture empowers them to handle specific data processing, transformation, and integration tasks with greater autonomy. This decentralization is particularly advantageous when handling geographically distributed data, where compliance, security, and organizational constraints often restrict access to certain data products.

To optimize data access, frequently queried data products are copied from data buckets and stored in caches (*e.g.*, Redis or Memcached). As presented in Figure 5.7, different architectural designs of the storage services can be considered on top of this model:

- single-layer architecture: a centralized layer for caches with multiple data buckets located in different regions.
- hierarchical architecture: multi-level layers for caches, that may be distributed at different locations (*e.g.*, network edges) based on user requirements, with multiple data buckets located in different regions.

Considering the different requirements of each domain team, how to properly access and process data products is a major challenge. To tackle this, we present optimization strategies tailored for data product management within the framework of our proposed hierarchical data distribution model. Our goal is to minimize operational costs during the design phase of data mesh applications. Such activities are normally conducted prior to implementation, when the optimal architecture needs to be identified by system engineering teams. A key benefit of our approach compared to simulation is to enable a faster evaluation of individual scenarios by means of our proposed model.

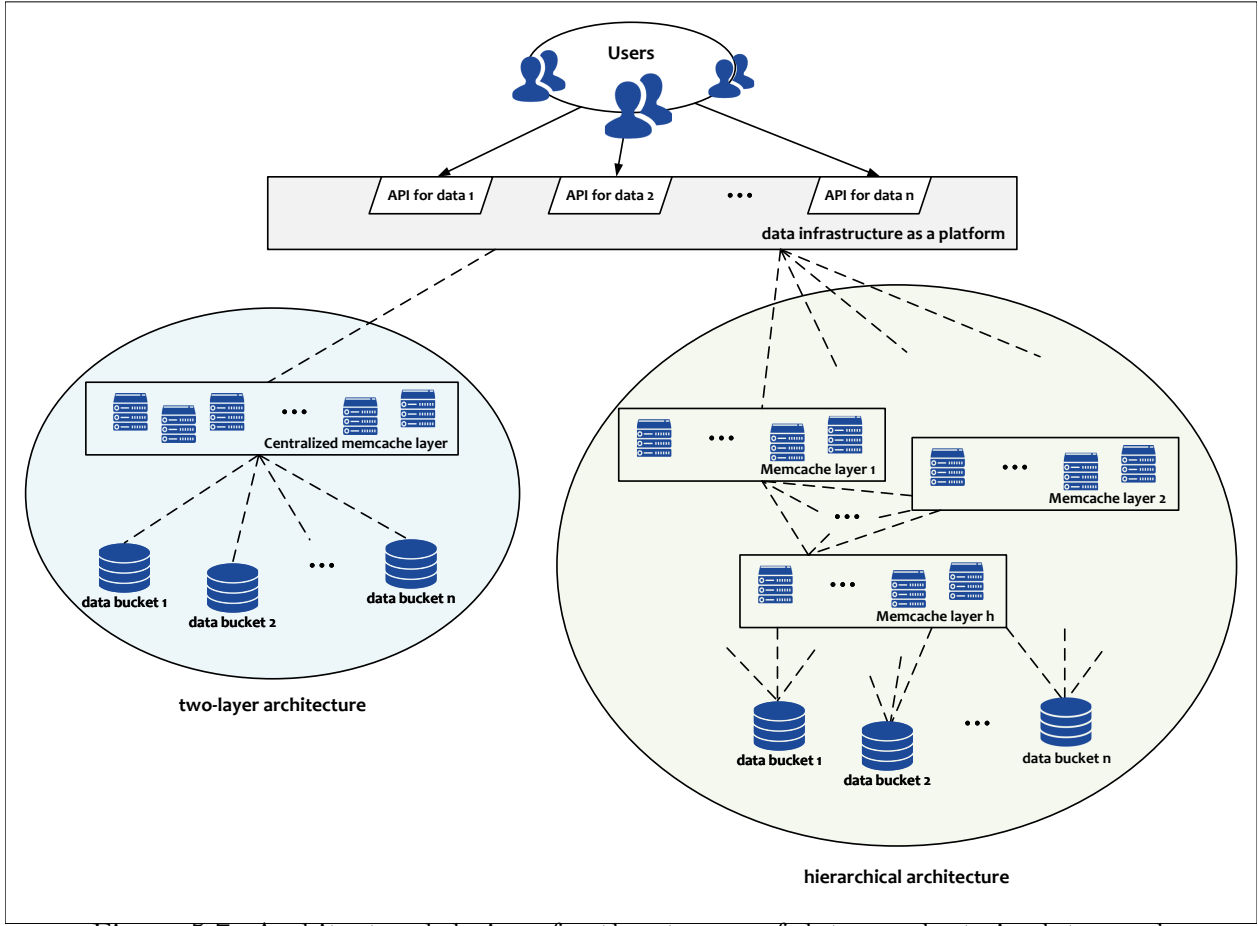


Figure 5.7: Architectural designs for the storage of data products in data mesh

Hierarchical Data Caching

For the hierarchical data caching (HC), our goal is to minimize the total operational cost $C_1(\mathbf{x})$, which is set as the total miss ratio of all data products. We define $\mathbf{x} = [x_{11}, \dots, x_{ij}, \dots, x_{IM}]$ as the **caching plan**, where x_{ij} is the access probability of data product $i = 1, \dots, I$, to be cached at cache node $j = 1, \dots, M$, satisfying $0 \leq x_{ij} \leq 1$.

Using the previous definitions, we can define the total operational cost $C_1(\mathbf{x})$, *i.e.*, the total miss ratio under caching plan $\mathbf{x}$, as:

$$C_1(\mathbf{x}) = 1 - \mathbf{Q}\boldsymbol{\gamma}(\mathbf{x}), \quad (5.8)$$

where matrix $\mathbf{Q} = \{q_{ij}, i = 1, \dots, I, j = 1, \dots, M\}$ and vector $\boldsymbol{\gamma}(\mathbf{x}) = \{\gamma_{ij}, i = 1, \dots, I, j = 1, \dots, M\}$ describe the request rates and the probability of each data product at each cache node under the caching plan $\mathbf{x}$, respectively. Thus, the problem of configuring the data caching

layers is formalized as

$$\begin{aligned}
& \min_{\mathbf{x}} && C_1(\mathbf{x}), \\
& s.t. && \sum_i s_i \gamma_{ij} \leq m_j, \quad \forall j \\
& && \sum_{j \in \mathbf{a}_i} x_{ij} = 1, \quad 0 \leq x_{ij} \leq 1, \forall i
\end{aligned} \tag{5.9}$$

where s_i denotes the size of data item i , and the model generalizes prior assumptions by allowing heterogeneous item sizes, *i.e.*, $s_i \neq 1$, and $\mathbf{a}_i$ represents the set of cache nodes where item i can be stored. The first constraint guarantees that the total size of cached data products at each cache node is no larger than the node capacity. The second constraint limits the sum of the probabilities of data product i to access all child nodes that meet the geographical constraints equal to 1. The complexity of the program depends on the optimization algorithm chosen for its solution and it is driven by the number of variables, which is $O(I \cdot M)$, and the $O(M)$ constraints. Upon using a metaheuristic, such as GA, implemented by embedding the constraint evaluation in the objective calls, this overall results in an estimated complexity of $O(P \cdot G \cdot I \cdot M)$, where P is the GA population and G the number of GA generations. In our experiments, the population P is proportional to the number of variables, making the solution approach overall quadratic in $I \cdot M$. In cases where this exceeds the desired complexity, keeping a fixed GA population as the problem size grows can help to reduce the computational burden.

Hierarchical Caching and Processing

For hierarchical data caching and processing networks (HCP), our optimization goal is to minimize the mean service level agreement (SLA) deviation of all data products $C_2(\mathbf{x})$, *i.e.*,

$$C_2(\mathbf{x}) = \frac{1}{I} \sum_i \max\{0, R_i(\mathbf{x}) - SLA_i\}, \tag{5.10}$$

where the SLA deviation of each data product is the difference between the actual response time $R_i(\mathbf{x})$ occurring for data product i by configuring the system with plan $\mathbf{x}$ and the SLA target SLA_i , which is defined as the maximum allowable mean response time for serving data

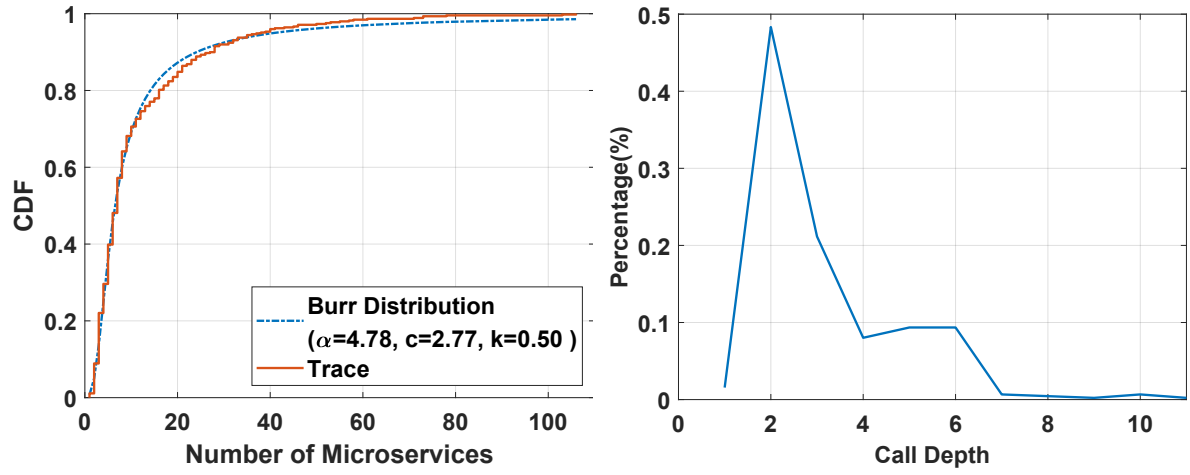
the performance of data product i . In such a model, users that request data product i are denoted in the ICQ by a dedicated reference task, *i.e.*, $RT1$. Such task models the process of requesting data product from the data mesh system through activity, *i.e.*, $A1$. The activity in the ICQ model issues a synchronization call to the cache node that stores the data product i , based on the caching plan $\mathbf{x}$. The remaining data products $i' \neq i$ are also modeled in $ICQ(i)$, however, this is done in an aggregate fashion by mapping them to the workflow of a client that describes the arrival rate of requests to the cache node they are stored into.

Within such clients, a request to data product i' will be issued in accordance with the total probability of all data products in this group and the relative frequency of requests of i' with respect to this group. Through these mappings, $ICQ(i)$ provides sufficient details to collect response time predictions for accesses to data product i , while only providing aggregate performance measures for the remaining data products. This is generally preferable than modeling all individual data products in the same model, which would result in a large number of job classes in the underpinning queueing models, facing scalability issues. Instead, based on the collection of models $ICQ(i)$, we can calculate the mean response time of processing each data product i with lower computational cost.

Continuing the description of the transformation, cache nodes are described in the ICQs as cache tasks, such as $CT1$ and $CT2$, with the same settings in the original model used for the caching plan generation. Attached to each cache task is a workflow that specifies how the cache handles the hit and misses for the requested products. If the requested data product is cached, a cache hit is conducted at the cache node to process the request, *e.g.*, $ACh1$. If the requested data product is not cached, a cache miss occurs and the node forwards the request to data buckets for further processing, *e.g.*, via the call from $ACm1$ to $Aback$.

5.5.3 Experimental Evaluations

We perform a trace-driven simulation to evaluate our proposed analytical solutions in the data mesh scenario. The trace is collected from a subset of microservices running on Alibaba production clusters over ten thousand bare-metal nodes during twelve hours in 2021 [LXL⁺21].



(a) CDF of the number of microservices of sampled traces (b) Call depth distribution of sampled traces

Figure 5.9: Statistical characteristics of the sampled Alibaba traces

The subset contains 449 trace ids, each of which represents a call graph consisting of multiple calls between different microservices. The total number of involved microservices is 737, which can be categorized into two types: stateful services, *e.g.*, databases and Memcached, stateless services, *i.e.*, data processing for different purposes. The call rate and CPU utilization of each container serving for the corresponding microservice are given.

Figure 5.9(a) shows the cumulative distribution of the number of microservices for all sampled traces. We can see that the CDF follows a Burr distribution [LXL⁺21] with parameters of $\alpha = 4.78, c = 2.77, k = 0.50$. 50% of the traces have less than 10 microservices and 90% of the traces contain no more than 30 microservices. Figure 5.9(b) presents the distribution of the call depth, *i.e.*, the length of the longest path in a call graph, of all sampled traces. We can observe that 80% of the traces have a call depth of 4 at most and only 2% of the traces possess a call depth over 7.

The parameter settings for simulating the sampled Alibaba traces are configured based on the statistical characteristics. The number of microservices in each call graph is ranging between 2 and 20, while the number of data products is set to 10, 50, 100, and 500. The cached data products follow a Zipf distribution with a parameter value of 0.8, and their total size is proportional to the allocated cache capacity. Each cache node adopts the LRU replacement policy.

For each combination of the parameters, we first optimize the HC program to obtain a caching plan $\mathbf{x}$ with minimized total miss ratio. According to the caching plan $\mathbf{x}$, we then transform the plan to ICQ models to calculate the mean response time of processing all data products. Further, with the same hierarchical caching architecture, we optimize the HCP program to derive a caching plan $\mathbf{x}'$ with minimized mean SLA derivation. We then utilize the new caching plan to calculate the total miss ratio. Thus, we employ the total miss ratio and mean response time as the performance measure and compare the two optimization programs under different cases.

From Figure 5.10, it is evident that our analytical solution-based optimization of the HC program exhibits a better mean response time for larger scale call graphs while the HCP program performs better with smaller scale. Specifically, the HC program achieves a reduction in miss ratio of 21% on average, accompanied by a significantly lower mean response time of 37% when the average size of call graphs is 15. Conversely, for the average graph size of 5, the HCP program demonstrates an average improvement of 17% in mean response time, albeit at the expense of minor disadvantages in total miss ratio. Also, in such cases, approximately 16% of the services in the HC program fails to meet the SLA requirements. These results highlight the potential benefits of employing analytical solutions to optimize HCP programs for smaller scale call graphs, ensuring compliance with SLA demands. While for systems with higher complexity, leveraging HC programs offers a compelling approach to maintain exceptional performance levels.

From Figure 5.11, we can observe that, the HCP program outperforms better than the HC program across different scales of data products. Specifically, when considering $I = 50$ and $I = 500$, as the average graph size increases, the HCP program can yield a reduction of up to 24% and 10% in mean response time, respectively. Moreover, it is worth noting that as the average graph size scales up to 12, although the HCP program demonstrates a modest 2% improvement in mean response time compared to the HC program, 26% of the services in the HC program violates the SLA requirements. These findings demonstrate the efficacy of leveraging analytical models to optimize HCP programs across a range of data product variations.

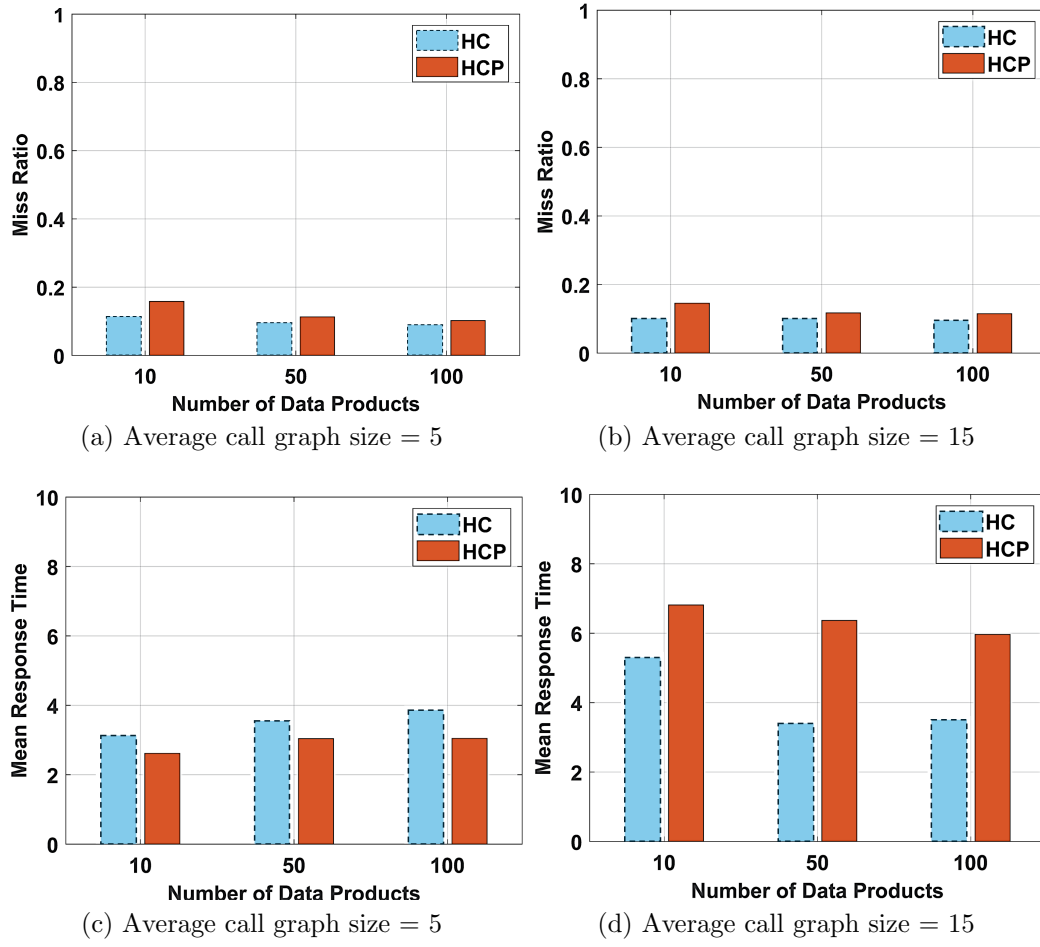


Figure 5.10: Miss ratio and mean response time with respect to the increase of the number of data products (Alibaba dataset)

In summary, employing our proposed analytical solutions to optimize the HCP program demonstrates better performance in smaller scale call graphs across various data product scales, exhibiting a mean response time improvement of up to 24%. On the other hand, the optimization of the HC program showcases higher effectiveness for larger scale call graphs, achieving a significant reduction in miss ratio of 21% and an enhancement in mean response time of 37%.

5.6 Summary

In this chapter, we introduced the hierarchical caching model as an effective mechanism for optimizing data access in distributed microservice architectures. We began by discussing the principles of hierarchical data distribution and explored replacement policies using access paths.

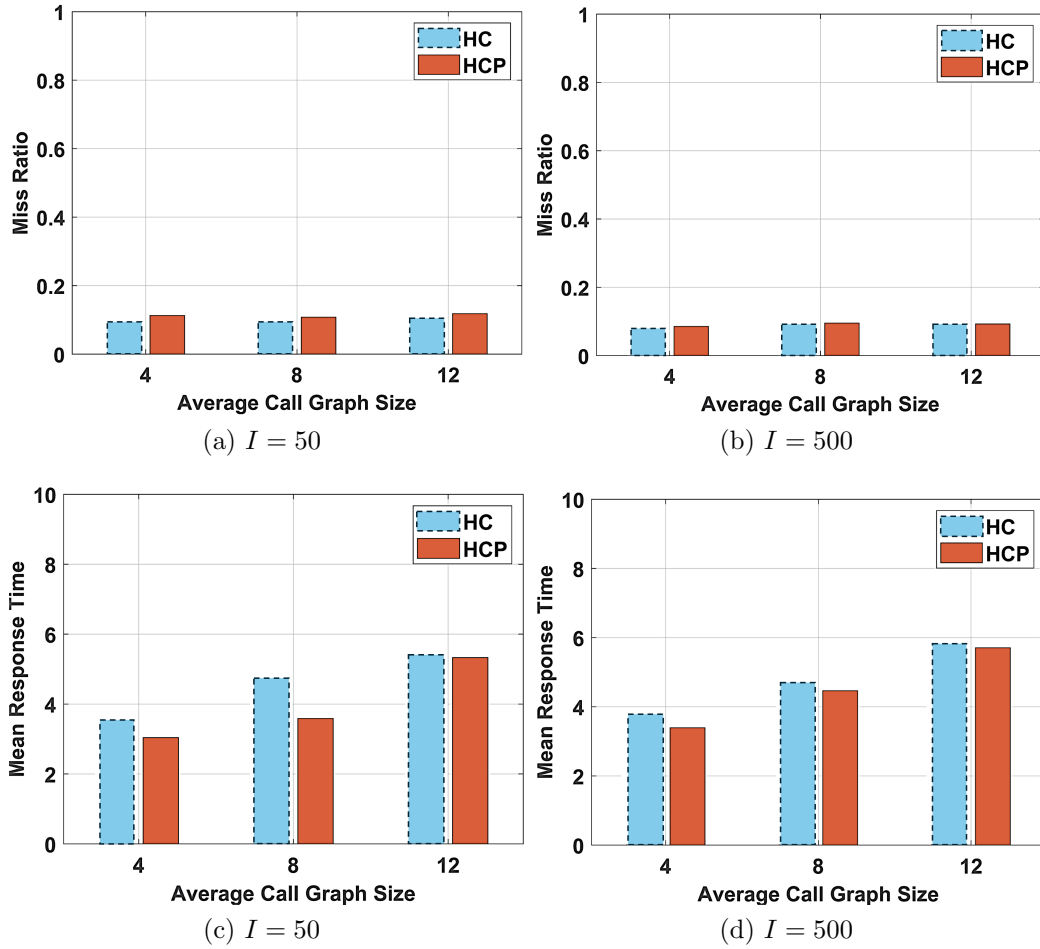


Figure 5.11: Miss ratio and mean response time with respect to the increase of the average call graph size (Alibaba dataset)

We then developed analytical methods to model cache behaviors, presenting exact solutions for $RR-a(\mathbf{m})$ policies and applying TTL approximations for $LRU-a(\mathbf{m})$ policies. Our models were validated through simulations and real-world traffic scenarios, demonstrating their applicability in capturing caching performance. Finally, we applied the hierarchical caching model to data mesh architectures, showcasing its effectiveness in optimizing data placement and reducing access latency. These results underscore the importance of hierarchical caching in improving the performance of microservice-based distributed systems, providing a robust framework for future research and practical applications in this field.

Chapter 6

Surrogate Modelling for Tandem BMMPP/GI/c Queues

6.1 Introduction

Edge computing has emerged as a key paradigm for meeting localized computing demands by distributing computing and storage resources at the network edge, such as wireless access points, enabling high bandwidth and low latency. However, since each edge node has limited resources, it is significant to employ appropriate service models to ensure effective edge computing. Common service models utilized in edge computing include Infrastructure as a Service (IaaS), Software as a Service (SaaS), and Function as a Service (FaaS).

IaaS, which provisions resources at the granularity of virtual machines, can lead to significant resource wastage, particularly since many edge computing applications like Internet-of-Things (IoT) services often exhibit periodic or intermittent request patterns [CZ16]. On the other hand, SaaS offers more fine-grained resource provisioning but comes at the expense of programmability and customization [PWCL22]. Different from IaaS or SaaS, FaaS utilizes lightweight containers that are instantiated only in response to specific events and terminated once the task is completed. This event-driven model minimizes resource wastage by dynamically scaling resources according to actual demand. Furthermore, FaaS offers greater programmability and

customization, as it allows developers to define and deploy custom functions with fine-grained control over application logic without being tied to predefined software capabilities.

Motivated by the benefits of FaaS, the integration of FaaS paradigm with edge computing, *i.e.*, serverless edge computing, has garnered significant attention from both academia and industry. Research primarily focuses on designing platforms and architectures that adapt serverless computing to edge scenarios [RMNB21]. For instance, the authors in [NRS⁺17] propose a unified data analytics platform that extends serverless computing to the edge, facilitating seamless resource and analytics management across cloud and edge infrastructures. In [CCP21], the authors propose a decentralized framework for serverless edge computing in the IoT environment, designed to efficiently dispatch stateless tasks to in-network executors while minimizing response times and maintaining fairness across varying network conditions. During such design phases, given the resource limitations and heterogeneity of edge environments, how to guarantee the system performance for low response time is a critical challenge [XTQ⁺21].

Many studies adopt learning-based approaches to analyze and predict the performance of applications running in edge computing systems [LSX⁺19, YCX⁺20, MWL⁺20, BONC21, TXY⁺22]. For example, in [YCX⁺20], a neural network model, built with denoising auto-encoder and fuzzy clustering for QoS prediction in mobile edge environments, is designed to improve the prediction accuracy while addressing the overfitting problem. In [BONC21], five widely used machine learning models, such as support vector regression and random forest, are considered for predicting Convolutional Neural Networks (CNN) inference execution time on Edge GPUs. However, when applying learning-based methods in the design phase of serverless edge computing systems, extensive large-scale profiling experiments are required to collect sufficient training data, leading to significant operational costs. Additionally, gathering adequate data in the design phase may be particularly challenging due to the limited availability of system performance metrics at this stage.

Alternatively, some researchers consider analytical performance models for FaaS platforms, allowing application developers and serverless operators to perform capacity planning and system analysis rapidly, without the need for costly and large-scale experimental setups [MK20a]. The

authors in [MK20b] and [MK20a] develop transient and steady-state analytical performance models based on M/G/m/m queueing systems to make capacity predictions for serverless computing platforms. The authors in [GC20] present a sizing method, called COCOA, leveraging the LQN model and M/M/k setup models to derive performance estimates for provisioning FaaS systems that meet service-level agreements.

While these existing analytical models are well-suited for performance prediction of serverless computing, they overlook the fact that offloaded jobs often arrive in batches [HLC20], leading to significant challenges due to the limited computational resources available at the edge. These batch arrivals complicate resource allocation and queueing dynamics, often resulting in performance bottlenecks. Therefore, efficient performance models are crucial to capture the constraints and operational dynamics of serverless edge computing systems.

To address these issues, we propose a surrogate modeling approach for predicting the performance of microservices in serverless edge computing systems. We first model the system as a tandem queueing network and apply the decomposition technique to break down the workflow into individual components. Each component is then analyzed in isolation using BMPP/GI/c queues to capture the effects of batch arrivals. Due to the complexity of solving such tandem models analytically, we further develop a deep surrogate model to approximate the performance metrics. Finally, extensive simulations validate the accuracy and robustness of our proposed model. Overall, we summarize our contributions as follows:

- We investigate the workflow characteristics of AI-SPRINT application traces, offering an in-depth analysis of batch arrival patterns in serverless edge computing systems.
- We formulate a performance model based on tandem queueing networks with batch arrivals to capture the impacts of batch job patterns and distributed workflow dependencies.
- To address the computational overhead of analytical methods, we propose a deep surrogate model to approximate the performance metrics of tandem workflows.
- We validate our performance model through extensive simulations with both synthetic workloads and real-world AI-SPRINT traces, demonstrating that our surrogate model

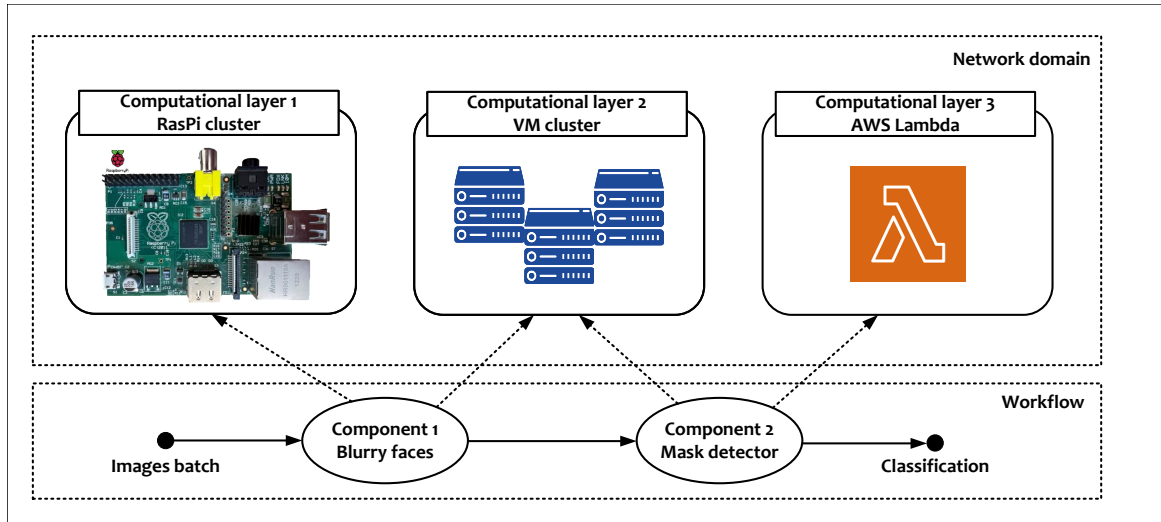


Figure 6.1: An example of an AI-SPRINT application for mask detection

achieves strong alignment with actual performance data, yielding a MAPE of 5.33%.

The rest of the chapter is organized as follows. Section 6.2 introduces the AI-SPRINT applications and presents a detailed trace analysis. Section 6.3 gives the motivation for modeling with tandem queueing networks and outlines the approach for their approximation. Section 6.4 presents the performance model and deep surrogate modeling approach. Experimental results are discussed in Section 6.5 and conclusions are provided in Section 6.6.

6.2 System Characteristics

6.2.1 AI-SPRINT Applications and Profiling

AI-SPRINT (“Artificial Intelligence in Secure PRiVacy-preserving computing coNTinuum”) is a comprehensive framework designed to facilitate the development and deployment of AI applications across a distributed computing continuum, encompassing both edge devices and cloud infrastructures [SAM⁺21, FSA23]. The framework addresses the critical need to balance performance, accuracy, and security within AI-driven workflows.

Within the AI-SPRINT framework, an application consists of multiple interconnected Python components, each performing a specific task within the workflow. The execution of these

components is event-driven and orchestrated by Open Source Serverless Computing for Data-Processing Applications (OSCAR) [PRN⁺19], an open-source platform that supports the FaaS model. Each component is triggered by file uploads, and the output of one component automatically initiates the next in the sequence.

To ensure execution consistency and reliability, each component is encapsulated within a Docker container. These containers are deployed using the Serverless Container-aware ARchitectures (SCAR) framework, which facilitates the execution of Docker containers on AWS Lambda, thereby enabling the application to utilize serverless computing resources efficiently. The architecture supports a diverse range of computational resources, from edge devices like Raspberry Pis to cloud-based virtual machines, allowing for optimized resource allocation tailored to the computational demands of each component.

An example of an AI-SPRINT application is depicted in Figure 6.1, which features a two-component workflow for mask detection executed across multiple computing layers. The first component, *Blurry Faces*, processes video input by extracting frames every five seconds, detecting faces using a neural network, and anonymizing them through pixel modification. The anonymized frames are then passed to the second component, *Mask Detector*, which detects the presence of masks on the faces and annotates the images with corresponding labels and confidence scores. This application exemplifies the use of edge computing, where video preprocessing is performed on a cluster of Raspberry Pi devices to ensure data privacy by keeping sensitive information at the edge. The anonymized frames are then uploaded and analyzed in the cloud, allowing for efficient and secure mask detection across various urban areas.

To systematically profile AI-SPRINT applications across diverse hardware configurations, an advanced auto-profiling tool integrated with the OSCAR framework named OSCAR-P is further employed [GGF⁺23]. The profiling process initiates with the specification of detailed hardware parameters and application requirements, such as node configurations, memory capacity, processing power, Docker images, and parallelism levels. OSCAR-P then conducts experiments by dynamically adjusting these parameters and executing the application under various scenarios. After each test run, OSCAR-P retrieves and processes the execution logs, providing a compre-

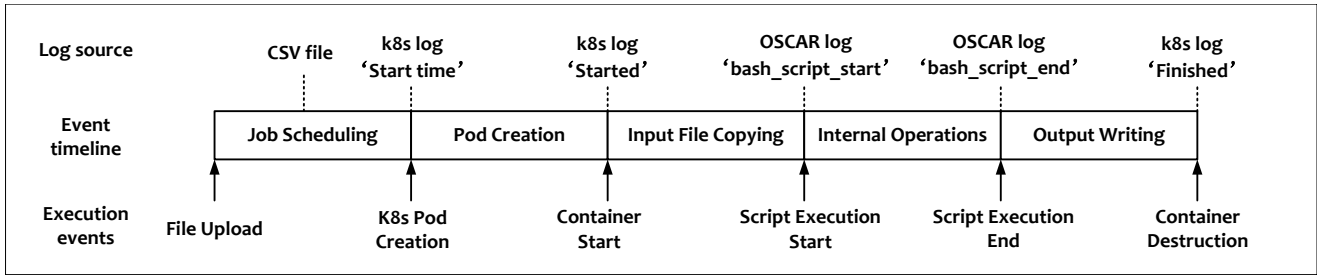


Figure 6.2: Detailed logging process of OSCAR-P with execution stages and timestamps

hensive record of the timing, resource usage, and overall performance of each component.

Figure 6.2 illustrates the detailed logging process managed by OSCAR-P, highlighting the various stages of execution along with their corresponding timestamps from multiple sources. The timeline begins with the upload of the input file that triggers component execution, such as an image or video frame in the mask detection workflow, then progresses through job scheduling, pod creation, and container operations, and concludes with the destruction of the container. This logging process applies, for instance, to the mask detection workflow introduced earlier in Figure 6.1, where OSCAR-P records the full sequence of container lifecycle events for each individual component, including Blurry Faces and Mask Detector. The logs, collected from both Kubernetes (k8s) and OSCAR, offer a thorough analysis of the overhead, computation, and write-back phases for each component, thereby providing in-depth insights into the application performance across different resource configurations.

For such AI-SPRINT applications, predicting performance during the design phase is crucial, particularly when estimating the response time across various configurations, as it enables effective capacity planning, resource management, and maintaining quality of service. However, this task is challenging due to the complexity of distributed, multi-component workflows. Accurately predicting the execution time of components deployed across a computing continuum and understanding the complex interactions between these components is difficult. Moreover, the design phase often lacks sufficient data, making data-driven approaches both time-consuming and costly to implement [MVdLvN16]. To address these challenges, we propose an analytical approach that captures workflow characteristics to develop a performance model. Once established, such models are typically quick to evaluate and can often be used without the need for extensive application execution.

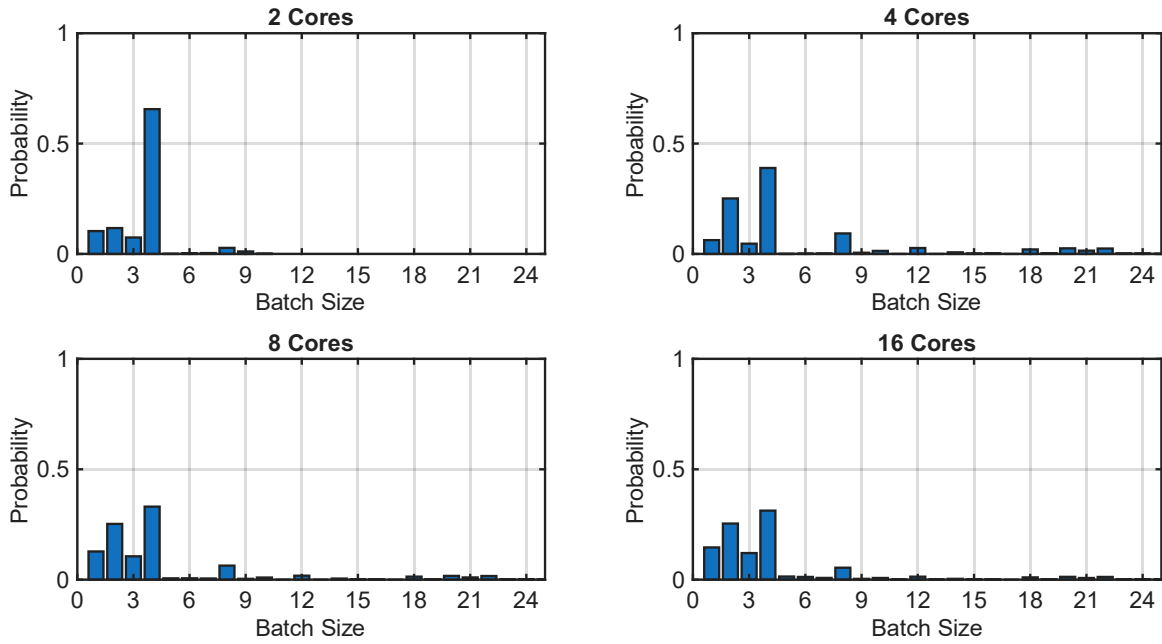


Figure 6.3: Probability mass function of batch sizes across different core configurations

6.2.2 Trace Collection and Analysis

Here, we use OSCAR-P to collect traces for analyzing workflow characteristics. The mask detection applications, as depicted in Figure 6.1, are evaluated using multiple VMs equipped with Intel Xeon processors featuring four cores and 8 GB of memory each. The trace spans a duration of six days, from November 27, 2023, at 10:30 AM to December 2, 2023, at 3:01 PM. During this period, a series of experiments are conducted to capture a comprehensive assessment of the system under a wide range of operational conditions.

Input Traffic Configuration: The experiments consider different numbers of 5-second videos as input for the mask detection application. Each set of input data is processed three times on VMs to ensure robust and consistent results. To simulate diverse real-world usage scenarios, the input traffic is configured to arrive in batches, with varying throughput rates that reflect different numbers of requests per second, ranging from 0.02414 to 1.86216. This variability is achieved by maintaining a batch size of 4 and altering the inter-arrival time between batches, allowing for a detailed analysis of the system performance under various input traffic conditions.

VMs Configuration: The VMs configuration in these experiments is critical for understanding how the number of cores influences the performance of the mask-detection application. In

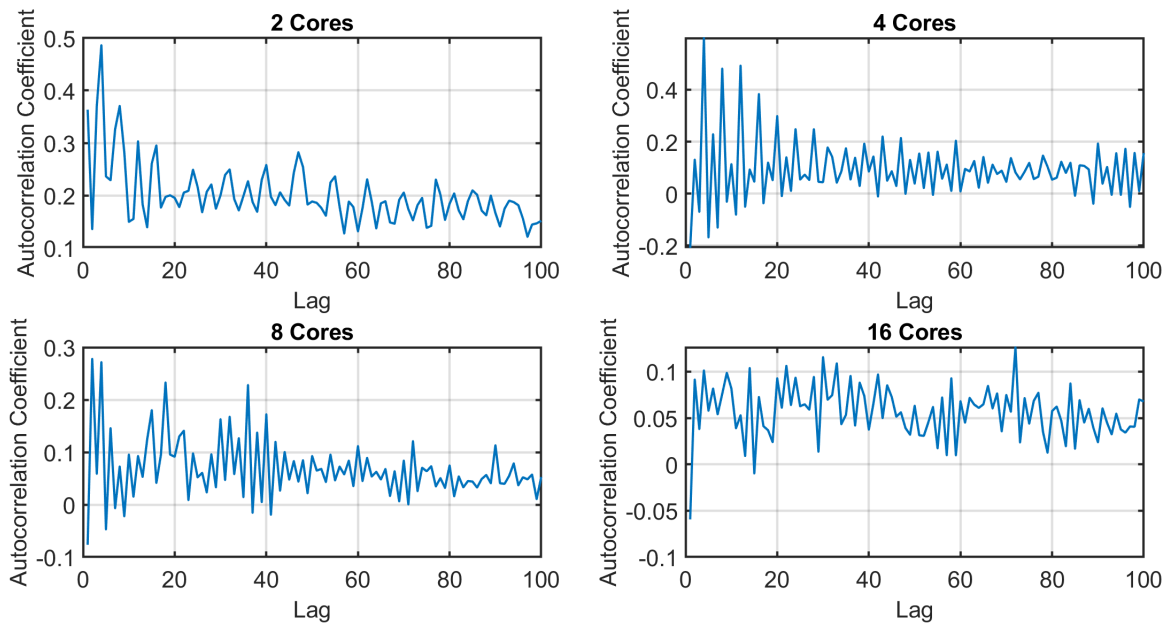


Figure 6.4: Autocorrelation function of batch sizes across different core configurations

this context, each node refers to an individual VM within the cluster, equipped with a specific number of cores, which are the fundamental CPU processing units. The number of cores on a node determines its capacity to run multiple tasks concurrently. Each component of the application operates within an isolated container, and since all components are single-core, each container requires one core to function. Thus, the parallelism level, defined as the maximum number of concurrent containers or instances that can run on a node, is directly determined by the number of cores available. In this setup, VMs are configured with 2, 4, 8, and 16 cores and service times for all components remain relatively stable across these configurations.

With these predefined configurations, AI-SPRINT application traces are collected from log files for each job across a varying number of cores. Both the full workflow and its individual components are profiled. Each log file records essential information, including job IDs, component names, job creation times, job completion times, parallelism level, and other relevant metrics. To capture the characteristics of these collected traces, we analyze each single component by its probability distribution and temporal dependencies, enabling more accurate system performance predictions.

As part of this analysis, we observe that both batch sizes and inter-arrival times in our traces show stable variability over time and no visible trend, indicating a lack of strong non-stationary

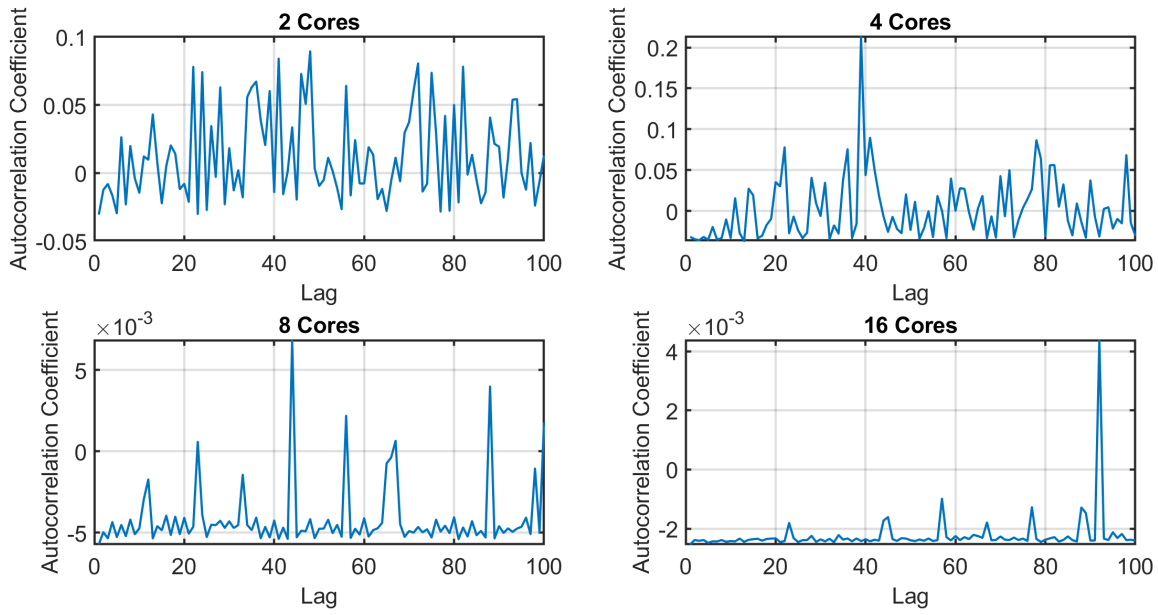


Figure 6.5: Autocorrelation function of inter-arrival times between batches

behavior. Based on these observations, we treat the input series as weakly stationary, which is a common assumption in queueing analysis under similar conditions. This allows us to use the autocorrelation function (ACF) to quantify the degree of short-range temporal dependence. Specifically, we utilize the ACF to analyze the temporal dependencies within the traces. The ACF quantifies the degree of correlation between observations in a time series X_t at different time lags, which is denoted by:

$$\rho(k) = \frac{\text{Cov}(X_t, X_{t+k})}{\text{Var}(X_t)}, \quad (6.1)$$

where $\text{Cov}(X_t, X_{t+k})$ represents the covariance between the values of the time series X at times t and $t + k$, and $\text{Var}(X_t)$ denotes the variance of X_t . The term lag k refers to the number of time steps by which the series is shifted. For example, a lag of 1 compares each value with the following value, thereby capturing the immediate temporal correlation.

For each individual component, Figure 6.3 illustrates that the probability mass function (PMF) spans batch sizes from 1 to 25, with a notable concentration around batch sizes of 4. Although the system is configured for batches of 4 jobs, Kubernetes scheduling policies, such as resource allocation, pod distribution, and job queueing, can lead to variations in the actual batch sizes

observed. These factors can create discrepancies between the intended batch size and what occurs during execution.

Furthermore, Figure 6.4 shows that the autocorrelation of batch sizes is weak and diminishes rapidly, implying that the sizes of consecutive batches are independent. In Figure 6.5, the periodic spikes observed in the ACF of the inter-arrival times suggest the presence of recurring scheduling patterns. However, these periodic spikes diminish rapidly, indicating that the inter-arrival times between batches are close to independent over lags.

6.3 Motivation

Given the characteristics of the AI-SPRINT application workflow, we propose modeling the system as a tandem queueing network, where each component operates as a sequential queueing system, an example of which is illustrated in Figure 6.6.

6.3.1 Traffic Decomposition Methods

To address state spaces explosion and complex interactions within such tandem networks, we apply the traffic decomposition approach. This technique enables the analysis of each node in isolation by treating the output from one queue as the arrival process for the next. Many decomposition methods have been developed to manage complex workflows, the key of which is how to characterize internal traffic between nodes. One well-known method is the Queueing Network Analyzer (QNA) [Whi83], which offers a parametric decomposition approach that approximates traffic flows using mean and variability metrics. Built on QNA, a Matrix Network Analyzer (MNA) is proposed [LC24], which leverages MAM and PH distributions to capture both first- and second-order moments, improving accuracy for systems with complex traffic patterns and phase-type queues. Additionally, a moments-based approximation [HHT10] is designed for the decomposition process by incorporating higher-order moments, allowing for a more detailed representation of inter-node traffic. By leveraging decomposition, we can

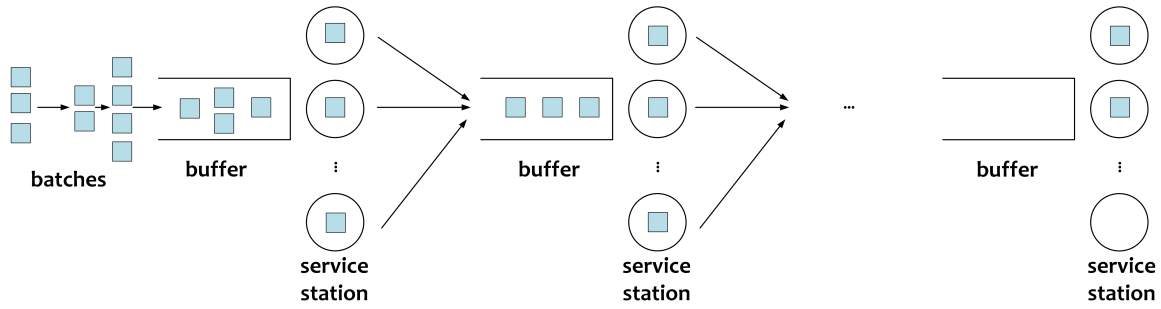


Figure 6.6: An example of a tandem queueing network with batch arrivals

efficiently estimate key performance metrics, such as mean waiting times, queue lengths, and loss probabilities of the system, while ensuring computational feasibility.

Within this framework, our primary focus is modeling each component individually to accurately capture the overall system dynamics. Given the trace characteristics of each individual component, particularly the batch arrivals with uncorrelated inter-arrival times and batch sizes, we propose to capture the dynamics of each component by employing the BMMPP/GI/ c model. These models are typically classified as M/G/ c -type models and characterized by an upper block Hessenberg structure in their infinitesimal generator $Q_{\text{M/G}/c\text{-type}}$, where only the first block below the main diagonal contains non-zero elements, while all subsequent lower blocks are zero matrices.

6.3.2 Multiserver Approximation Analysis

For each individual component processed with multiserver queues, existing matrix analytic research primarily focuses on solvers designed for single-server systems, *i.e.*, M/G/1-type models, leaving a gap in scalable solutions for complex multiserver scenarios. To approximate the BMMPP/GI/ c model, we initially focus on the response time as an illustrative metric and model the system as an equivalent single-server queue with an adjusted service rate. Our proposal is to adapt an approximation method proposed in the context of product-form queueing network theory to multiserver matrix analytic queueing systems, namely the approximation proposed in [SSSO87] that we describe below.

Specifically, we first model the BMMPP/GI/ c system as a single-server queue with a service

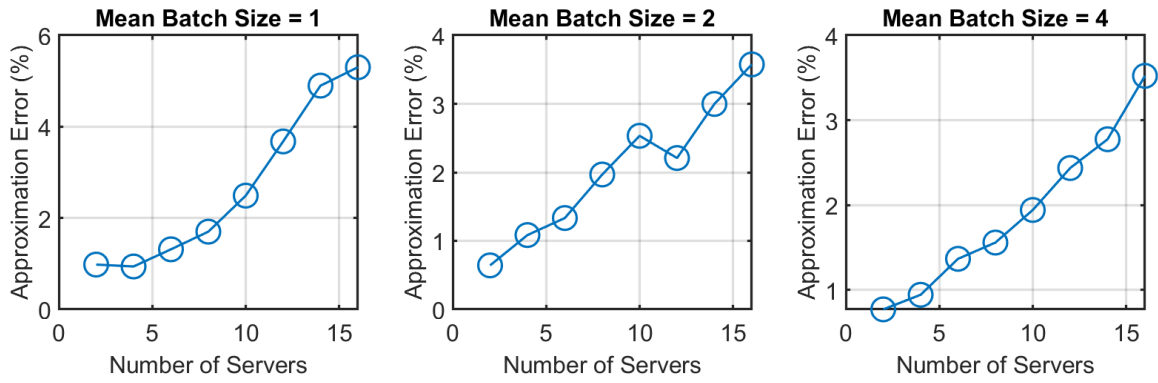


Figure 6.7: Approximation error for multiserver scenarios

rate of $c\mu$, where μ is the service rate of each individual server and c is the number of parallel servers. This represents an equivalent rate of service of the station when all its servers are busy. The response time R_s for this single-server system can be obtained using matrix analytic methods for the BMMPP/GI/1 queue. Second, an adjustment is needed to account that in certain periods of time the server will not be fully loaded. The correction factor is expressed as $\alpha = \frac{c-1}{c} \times \frac{1}{\mu}$. This factor ensures that for a job that arrives when the station is idle, the total response time will be

$$\frac{1}{c\mu} + \frac{c-1}{c} \times \frac{1}{\mu} = \frac{1}{\mu}$$

as intended. Summarizing, the response time $\widetilde{R}_m$ can be approximated as [SSSO87]:

$$\widetilde{R}_m = R_s + \frac{c-1}{c} \times \frac{1}{\mu}. \quad (6.2)$$

The merit of our proposal to apply this approach to matrix analytic queues is that it allows for an efficient approximation of the response time of the BMMPP/GI/c system by resorting to solution methods that are available for BMMPP/GI/1 queues. The approximation error σ is evaluated as:

$$\sigma = \frac{|\widetilde{R}_m - R_m|}{R_m} \times 100\%, \quad (6.3)$$

where R_m is the response time obtained through a BMMPP/GI/c simulator.

To determine the accuracy of the proposed approximations in a way that abstracts the accuracy

of the associated single-server solver, we test the proposed method first using simulation. As illustrated in Figure 6.7, the approximation error remains minimal across different numbers of servers and mean batch sizes. For scenarios with a small number of servers (*e.g.*, $c \leq 5$), the error is consistently below 2%, demonstrating that the approximation is highly accurate in these settings. As the number of servers increases, the error gradually escalates but the deviation remains limited to 4-6%. These results indicate that the approximation based on solvers for single-server systems provides a precise solution for BMMPP/GI/ c systems.

6.3.3 Tandem Workflow Approximation

In the full application workflow modeled as a tandem queueing network, the departure process of one component directly serves as the arrival process for the subsequent component. While using the BMMPP/GI/1 based approximation in the previous section has proven effective for approximating individual isolated queues, extending this approach to a complete workflow with multiple interconnected queues presents significant challenges. These challenges arise from the dependencies between successive components, where the departure processes of multiserver queues are difficult to obtain using basic BMMPP/GI/1 techniques.

To approximate the performance of the full tandem workflow, we initially leverage MAMsolver [RS02a, RS07] to solve each individual component within the network. MAMsolver is tailored to handle M/G/1-type, GI/M/1-type, and QBD processes. It employs matrix-analytic methods to compute key stationary performance metrics such as the probability vector, queue length distribution, waiting time, and higher-order moments. Additionally, MAMsolver integrates ETAQA techniques to enhance computational efficiency, offering an alternative solution for reducing time and space complexity when determining these performance measures.

Moreover, the departure process of each component is derived with simulations to analyze moments and ACF of the inter-departure times, and then passed as the arrival process for the subsequent component in the tandem workflow. For all subsequent components, the arrival process is defined by the departure process of the preceding component. This sequential analysis

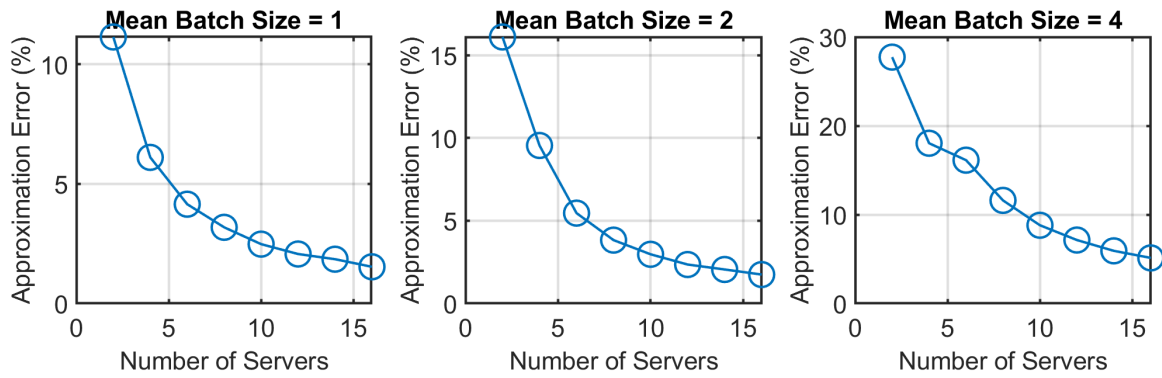


Figure 6.8: Approximation error for complete tandem workflows

allows us to approximate the performance of the entire tandem workflow, despite the challenges posed by inter-component dependencies.

To simplify the analysis, we first approximate the inter-departure times by fitting them to an Erlang- k distribution, which captures the first two moments. The results in Figure 6.8 show that the approximation error for the complete workflow remains below 27.73% across all configurations. As the mean batch size increases, the error tends to rise, particularly in scenarios with fewer servers. For smaller batch sizes, the error decreases rapidly, falling below 5% as the number of servers increases, indicating a high level of accuracy. While the maximum error reaches 27.73% in some configurations, this reflects a worst-case scenario under high batch sizes and limited server capacity. The model still captures key performance trends and provides useful estimates for early-stage system design. In queueing performance modeling, such levels of approximation are considered acceptable when closed-form analysis is infeasible and simulation is computationally expensive.

As the number of servers increases beyond 15, the system becomes less congested and queueing effects are reduced, making the approximation increasingly accurate. A notable difference between Figures 6.7 and 6.8 lies in the direction of error change with respect to the number of servers. This contrast arises from the structural differences between the two settings: Figure 6.7 evaluates a single multiserver queue in isolation, where increasing the number of servers introduces more complex service interactions that reduce the accuracy of the single-server-based approximation. In contrast, Figure 6.8 considers the full tandem workflow, where additional servers mitigate bottlenecks across components, reduce queueing delays, and ultimately improve

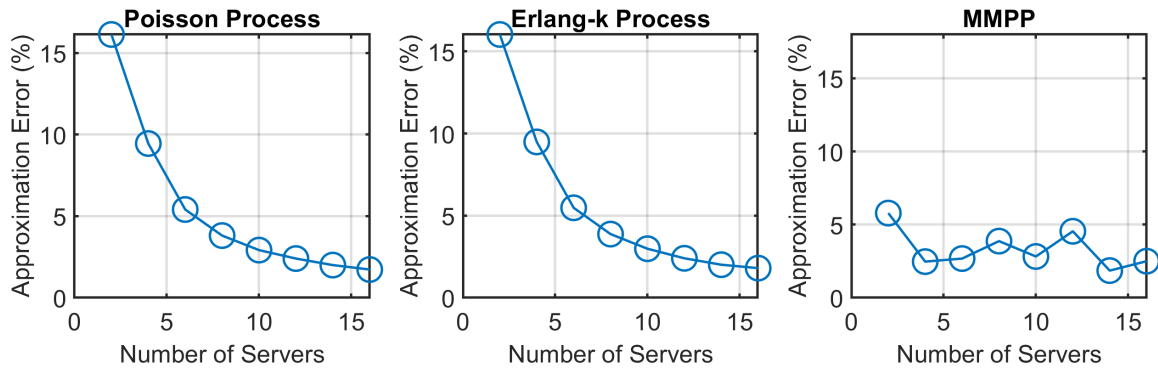


Figure 6.9: Approximation error for tandem workflows with different arrival process fitting methods in the subsequent queues

the overall approximation.

Further, we compare different methods of fitting the arrival process for the subsequent queues within the tandem workflow, specifically under a batch size of 2. The results in Figure 6.9 compare the approximation errors across tandem workflows modeled with three different types of arrival processes for the subsequent queues: Poisson, Erlang-k, and MMPP. The MMPP model, which accounts for the first three moments and ACF, demonstrates higher performance, with the approximation error consistently staying below 5.8% across different configurations. This improvement confirms that a more expressive model of the arrival process leads to higher accuracy when modeling tandem systems. The use of Erlang-k in Figure 6.8 reflects a deliberate modeling trade-off between accuracy and traceability, as it efficiently captures the first two moments and provides a lightweight approximation with minimal computational overhead. As shown in Figure 6.9, switching to MMPP further improves accuracy, albeit at the cost of increased modeling and simulation complexity, allowing flexibility depending on the application performance requirements.

While our current evaluation focuses on a representative two-stage workflow, the proposed decomposition approach is designed to scale to longer tandem systems. Since each component is analyzed independently with updated arrival processes, the approximation error is expected to grow gradually rather than accumulate linearly. A more comprehensive assessment of scalability with respect to the number of queues is left as a direction for future work.

6.3.4 Issues with Approximation Methods

While existing BMMPP/GI/1 solvers provide accurate approximations of multiserver systems and enable measurement of full tandem workflows, both simulation-based and analytical approaches exhibit notable limitations. The primary drawback lies in their high computational complexity when applied to complex configurations. Simulation-based methods, although versatile, become computationally intensive and time-consuming as the number of servers or batch sizes increases, rendering them impractical for real-time analysis or scenarios requiring frequent evaluations. On the other hand, analytical solutions leverage the repetitive block structure characteristic of such models, enabling the application of matrix-analytic techniques [Neu21], a well-established framework for the exact analysis of such systems. However, in large-scale systems with complex dynamics, these methods may fail to capture all relevant variations accurately.

To demonstrate the computational demands, we employ MAMsolver as a case study for analyzing the BMMPP/GI/1 model. Specifically, we examine the time complexity associated with varying numbers of phases in the arrival process, averaging 1,000 samples for each configuration. As depicted in Figure 6.10, the computational overhead escalates significantly with an increasing number of phases in the arrival process, highlighting scalability issues, particularly in scenarios characterized by low variability. If both the arrival and service processes are considered, the computational burden can exceed 10^6 seconds, further illustrating the limitations of this approach in handling complex systems. This approach becomes particularly costly in design studies, where thousands of models may need to be solved.

Moreover, the single-server approximation approaches fail to model the departure processes of individual components in multiserver configurations. This limitation is particularly problematic in tandem workflows, where the departure process of one queue directly influences the performance of subsequent queues. The lack of modeling of these departure processes hinders the effectiveness of decomposition techniques for the analysis of the entire workflow.

Given the computational limitations of existing methods, there is a critical need for models that

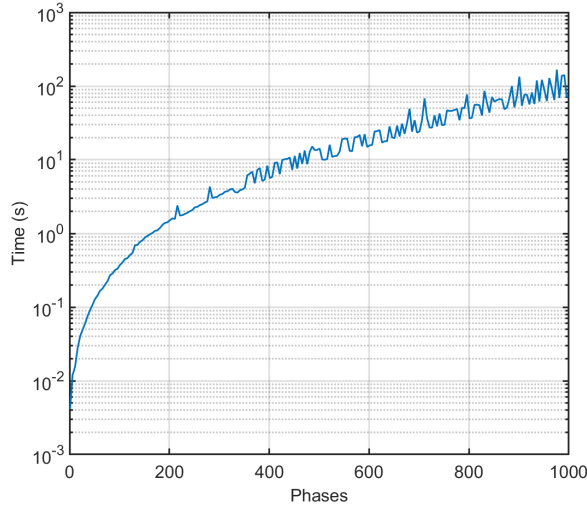


Figure 6.10: Time complexity analysis for BMMPP/GI/1 models

can achieve both accuracy and efficiency for multiserver scenarios. To address these challenges, we propose leveraging surrogate modeling techniques. Surrogate models are simplified, data-driven approximations that capture the essential behavior of complex systems while efficiently reducing computational overhead. By offering a scalable solution that combines computational efficiency with precision, these models can address critical issues related to time complexity, complex multiserver configurations, and the complex departure processes in interconnected systems. The following section provides a detailed exploration of this approach.

6.4 Surrogate Modeling

6.4.1 Performance Model

Since we apply the decomposition technique to the entire application workflow, this section provides a detailed performance modeling of each individual component. Specifically, we utilize a two-state Batch Markov Modulated Poisson Process, denoted as BMMPP(2), to model the batch arrival process of each component. The intensity matrices of BMMPP(2) can be characterized by $\mathbf{D}_b$, $b = 0, 1, \dots, B$, where B is the maximum batch size, as follows:

The intensity matrix $\mathbf{D}_0$ governs transitions without arrivals:

$$\mathbf{D}_0 = \begin{bmatrix} -\lambda_1 & \lambda_{12} \\ \lambda_{21} & -\lambda_2 \end{bmatrix}, \quad (6.4)$$

where λ_{12} and λ_{21} represent the transition rates between the two states. The intensity matrices $\mathbf{D}_b$, $b = 1, \dots, B-1$, describe transitions associated with batch arrivals of size b :

$$\mathbf{D}_b = \begin{bmatrix} \lambda_1 p_{11b} & 0 \\ 0 & \lambda_2 p_{22b} \end{bmatrix}, \quad (6.5)$$

where p_{11b} and p_{22b} are the probabilities of observing a batch arrival of size b in states 1 and 2, respectively. The intensity matrix $\mathbf{D}_B$ is defined as:

$$\mathbf{D}_B = \begin{bmatrix} -\lambda_1 - \lambda_{12} - \sum_{i=1}^{B-1} \lambda_1 p_{11i} & 0 \\ 0 & -\lambda_2 - \lambda_{21} - \sum_{i=1}^{B-1} \lambda_2 p_{22i} \end{bmatrix}. \quad (6.6)$$

Thus, the infinitesimal generator $\mathbf{Q}_{BMMPP(2)}$ of the underlying Markov process is given by:

$$\mathbf{Q}_{BMMPP(2)} = \sum_{b=0}^B \mathbf{D}_b. \quad (6.7)$$

Given the uncorrelated nature of the trace data, we consider a variant of the BMMPP(2) where the batch size probabilities are assumed to be identical across states, *i.e.*, $p_{iib} = p_{jjb}$, for any $i \neq j$. Under this assumption, the intensity matrices can be expressed more simply as [YLRC19]:

$$\mathbf{D}_0 = \mathbf{Q} - \mathbf{\Delta}(\boldsymbol{\delta}), \quad (6.8)$$

$$\mathbf{D}_b = \mathbf{\Delta}(\boldsymbol{\delta}) \cdot \mathbf{\Delta}(p_b), \quad b \geq 1, \quad (6.9)$$

where $\mathbf{\Delta}(\boldsymbol{\delta})$ is a diagonal matrix with $\boldsymbol{\delta} = [\lambda_1, \lambda_2]^T$, and $\mathbf{\Delta}(p_b)$ is a diagonal matrix with the batch probabilities p_b as its diagonal elements, satisfying $\sum_{b=1}^B p_b = 1$.

By assuming no correlation between inter-arrival times and batch sizes, this variant, denoted

as $\text{BMMPP}_v(2)$ and also known as a MAP with independent and identically distributed (i.i.d.) batch arrivals, provides a tractable approach for modeling the batch arrival process of each application component. Additionally, the service time distribution is modeled using a General Independent (GI) distribution, offering flexibility to capture a wide range of service behaviors, and the model incorporates c servers to account for the parallel processing capabilities of each component.

Regarding the embedded Markov chain of the $\text{BMMPP}_v(2)/\text{GI}/c$ queue we study, each state in the chain is represented by three elements: the number of jobs in the system, the phase of the arrival process, and the phase of the service process. Both the arrival process and service process consist of multiple exponential phases in series. Due to the structural characteristics of this Markov chain, the state space can be partitioned into boundary and repetitive blocks. Consequently, the infinitesimal generator matrix $\mathbf{Q}$ can be constructed as follows:

$$\mathbf{Q}_{\text{BMMPP}_v(2)/\text{GI}/c} = \begin{bmatrix} \mathbf{B}_0 & \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{B}_3 & \cdots \\ \mathbf{C}_0 & \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots \\ 0 & \mathbf{A}_0 & \mathbf{A}_1 & \mathbf{A}_2 & \cdots \\ 0 & 0 & \mathbf{A}_0 & \mathbf{A}_1 & \ddots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}, \quad (6.10)$$

where $\mathbf{A}_k$, $k = 0, 1, 2, \dots$, and $\mathbf{C}_0$ denote the transitions for repetitive states, and $\mathbf{B}_k$, $k = 0, 1, 2, \dots$, represent the transitions for boundary states.

Specifically, $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_k$, $k \geq 2$, represent the backward, local, and forward transitions, respectively, as follows:

$$\mathbf{A}_0 = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ r\mu & \cdots & 0 \end{bmatrix}, \quad (6.11)$$

$$\mathbf{A}_1 = \begin{bmatrix} -\beta\lambda + r\mu & r\mu & \cdots & 0 \\ 0 & -\beta\lambda + r\mu & r\mu & \cdots \\ 0 & 0 & -\beta\lambda + r\mu & r\mu & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots \end{bmatrix}, \quad (6.12)$$

$$\mathbf{A}_k = \begin{bmatrix} P_k\lambda & 0 & 0 & \cdots \\ 0 & P_k\lambda & 0 & 0 & \cdots \\ 0 & 0 & P_k\lambda & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots \end{bmatrix}. \quad (6.13)$$

Here, β and r denote the number of phases in the arrival and service processes, respectively, and the expressions $\beta\lambda$ and $r\mu$ correspond to the total transition rates across the respective phase-type structures. This compact notation facilitates the representation of transitions in the generator blocks. P_k denotes the probability mass function of the batch size. $\mathbf{C}_0 = [0, 0, \dots, r\mu]^T$ represents the backward transition for the first repetitive block. Moreover, $\mathbf{B}_0$ and $\mathbf{B}_k$ denote the local and forward transitions for the boundary states, where $\mathbf{B}_0 = -\lambda$ and $\mathbf{B}_k = [\cdots 0, 0, P_k\lambda, 0, 0, \cdots]$.

6.4.2 Deep Surrogate Model

To solve the BMMPP_v(2)/GI/c queue in a more efficient way than with traditional matrix analytic methods, we further propose a deep surrogate model. This model is tailored to learn the system behavior and approximate the performance metrics of each component in multiserver systems. The model takes high-level statistical features of arrival and service processes as input, and outputs response time moments to approximate the performance. Specifically, it employs a feedforward neural network architecture with multiple hidden layers of fully connected neurons as shown in Figure 6.11. Each hidden layer applies Rectified Linear Unit (ReLU) activation functions to introduce nonlinearity, enhancing the capability to learn complex mappings between input features and performance metrics.

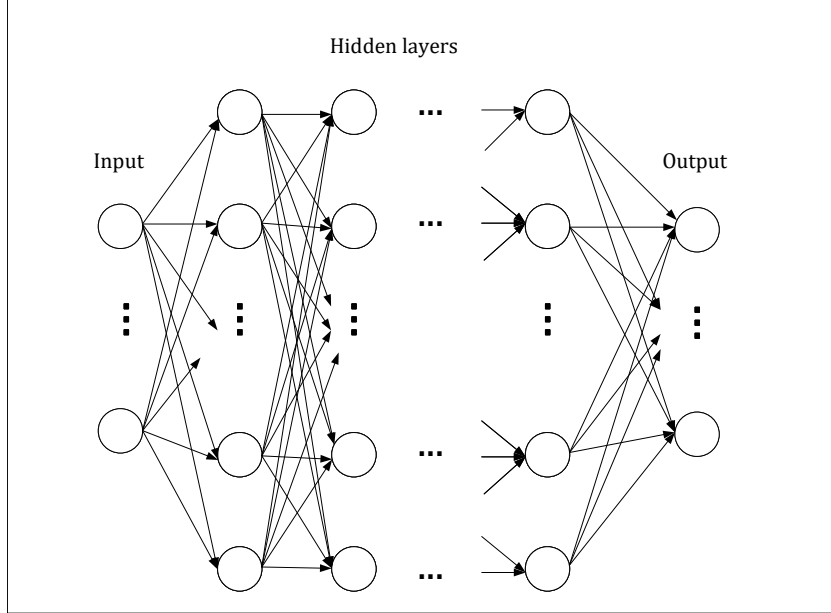


Figure 6.11: Feedforward Neural Network Architecture

Training Data and Normalization

Training data for the proposed deep surrogate model is generated from simulations of BMMPP/PH/c systems. The dataset includes critical features:

- Statistical moments of arrival processes E_{a1}, E_{a2}, E_{a3} ;
- PMF of batch arrival sizes P_{bs} ;
- Service process parameters E_{s1}, E_{s2} ;
- Number of servers c .

The target outputs are performance metrics, including moments of response times. To ensure effective training and enhance model performance, both the input features and output targets are normalized using min-max scaling:

$$X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}, \quad (6.14)$$

where X represents the original data values, $X_{\min}$ and $X_{\max}$ are the minimum and maximum values in the dataset, respectively.

Training Objectives and Loss Function

The training process focuses on minimizing the Mean Absolute Percentage Error (MAPE) between the predicted and actual performance metrics. The MAPE is defined as:

$$\text{MAPE} = \frac{1}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right| \times 100\%, \quad (6.15)$$

where y_i denotes the actual performance metrics, $\hat{y}_i$ represents the predicted values, and n is the number of samples. An adaptive optimization algorithm adjusts the learning rate dynamically to ensure efficient convergence. The network undergoes training for a specified number of epochs, with a batch size optimized for both computational efficiency and memory management, facilitating robust learning and accurate approximation of the performance metrics. This setup follows a supervised learning paradigm, where the ground-truth performance metrics used as targets are obtained from simulation during training.

6.5 Experimental Results

6.5.1 Simulation-based Evaluation

In this section, we evaluate our proposed deep surrogate model for full tandem workflows using data simulated by the JMT tool [BCS09]. For each component in the tandem workflow, we model it as a BMAP/GI/c system, where BMAP generalizes BMMPP for more complex dynamics.

BMAP Sample Generation

First, we generate samples of BMAP to simulate the arrival process for each component in the tandem queueing network. For the first component, by configuring the mean inter-arrival time

between batches and the squared coefficient of variation (SCV), we characterize the BMAP using the following parameters:

- **Mean inter-arrival time, E_1 :** The average time between consecutive batch arrivals.
- **Second moment, E_2 :**

$$E_2 = (1 + \text{SCV}) \times E_1^2. \quad (6.16)$$

- **Third moment, E_3 :** The third moment E_3 is set to the minimum value possible for the underlying MAP using the formulas in [HHG06]. Specifically, we let $r_i = \mathbb{E}[X^i] = E_i(i = 1, 2, 3)$, and define the squared coefficient of variation as $h_2 = \frac{r_2 - r_1^2}{r_1^2}$, the third standardized moment as $h_3 = \frac{r_3 - 3r_2r_1 + 2r_1^3}{r_1^3}$. [HHG06] shows that the minimal feasible value for $h_3(h_2 > 0)$ is $h_3^{\min} = h_2^2$. Since

$$r_3 = r_1^3 h_3 + 3r_1 r_2 - 2r_1^3,$$

we have

$$E_3^{\min} = E_1^3 h_2^2 + 3E_1 E_2 - 2E_1^3 = \frac{2E_2^2}{E_1^2} + 1.$$

- **Lag-2 autocorrelation, γ_2 :** The autocorrelation at a lag of two time units.

Given these parameters, we generate BMAP samples, each consisting of 10^6 requests. For each subsequent component in the tandem workflow, we use the JMT tool to log the departure times of each request from the preceding component. These logged departure times are then used to model the arrival process for the next component in the tandem network.

Training Data Generation

With the input of BMAP samples for each component, we further simulate the mean response time of each queue with the JMT tool. In parallel, we perform a full workflow simulation using

Table 6.1: Parameters for training data generation

Parameter	Range/Set
Mean inter-arrival time (E_1)	[50, 100] (steps of 1)
SCV	{1, 2, 4, 8, 16, 32, 64}
Second central moment (E_2)	Computed using Equation 6.16
Lag-2 autocorrelation (γ_2)	{0.0, 0.75, 0.9, 0.95, 0.99}
Batch size (S)	{2, 4, 8, 16, 32, 64}
Service rate (μ)	Erlang distribution with shape parameter μ_p (1-5) and scale parameter μ_a (0.1-0.5)
Number of cores (c)	{1, 2, 4, 8, 16}

Table 6.2: Training hyperparameters

Hyperparameter	Value
Learning rate	0.001
Batch size	512
Epochs	100
Optimizer	Adam
Loss function	MAPE

the JMT tool, with the arrival process of the first component as input, allowing for assessing the overall system performance. To generate the training data for the deep surrogate model, we further detail the configuration by simulating a range of scenarios using the JMT tool in Table 6.1. Note that in cases where the SCV is low, the impact of the ACF is typically marginal and often negligible; therefore, we focus our configuration on scenarios where $SCV \geq 1$.

Training Process

We train the deep surrogate model using the following architecture and configurations. The Deep Neural Network (DNN) consists of four hidden layers with 128, 64, 32, and 16 neurons respectively, using ReLU activation functions for non-linearity. The final layer outputs the moments of response time prediction for each component of the tandem workflow. We implement the model using PyTorch and optimize it using the Adam optimizer. Table 6.2 details the hyperparameters used in training the deep surrogate model.

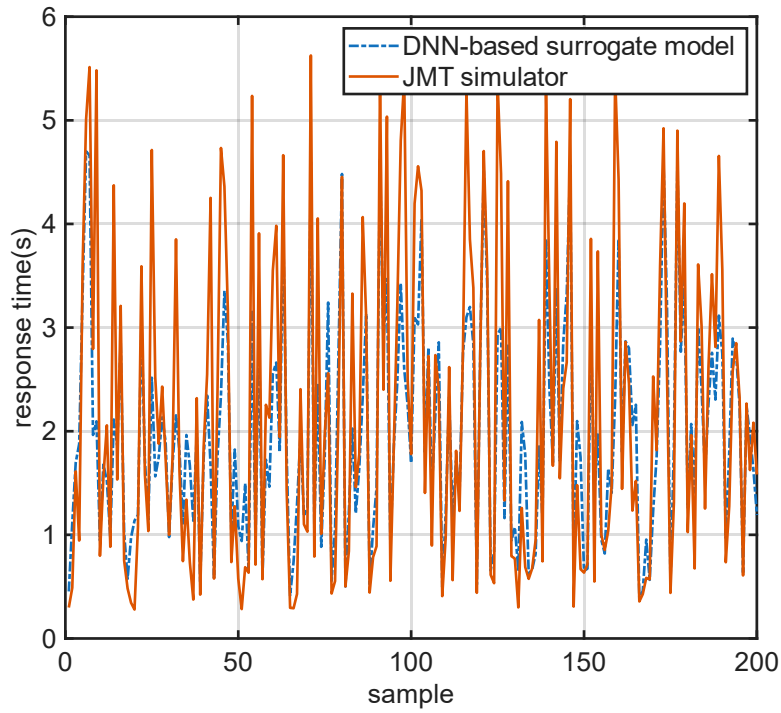


Figure 6.12: Comparison of the response time between JMT simulator and surrogate model

Experimental Evaluation

The effectiveness of our proposed deep surrogate model is evaluated by comparing its aggregated predictions with the response times obtained from the full tandem workflow simulations conducted in the JMT tool. As shown in Figure 6.12, the evaluation results present an average MAPE of 19.04% across all samples, demonstrating the accuracy of our proposed surrogate model in predicting the performance of tandem queueing systems.

6.5.2 AI-SPRINT Application Trace

To evaluate the effectiveness of our proposed deep surrogate model for real-world tandem workflows, we compare its performance against the trace-driven simulation using traces collected from AI-SPRINT applications operating within serverless edge computing systems. The experimental setup includes a total of 216 different configurations, varying by the number of cores and job arrival rates. The trace data collected under each configuration spans a range of 32 to 3,968 requests. The inter-arrival times between batches within these traces exhibit low

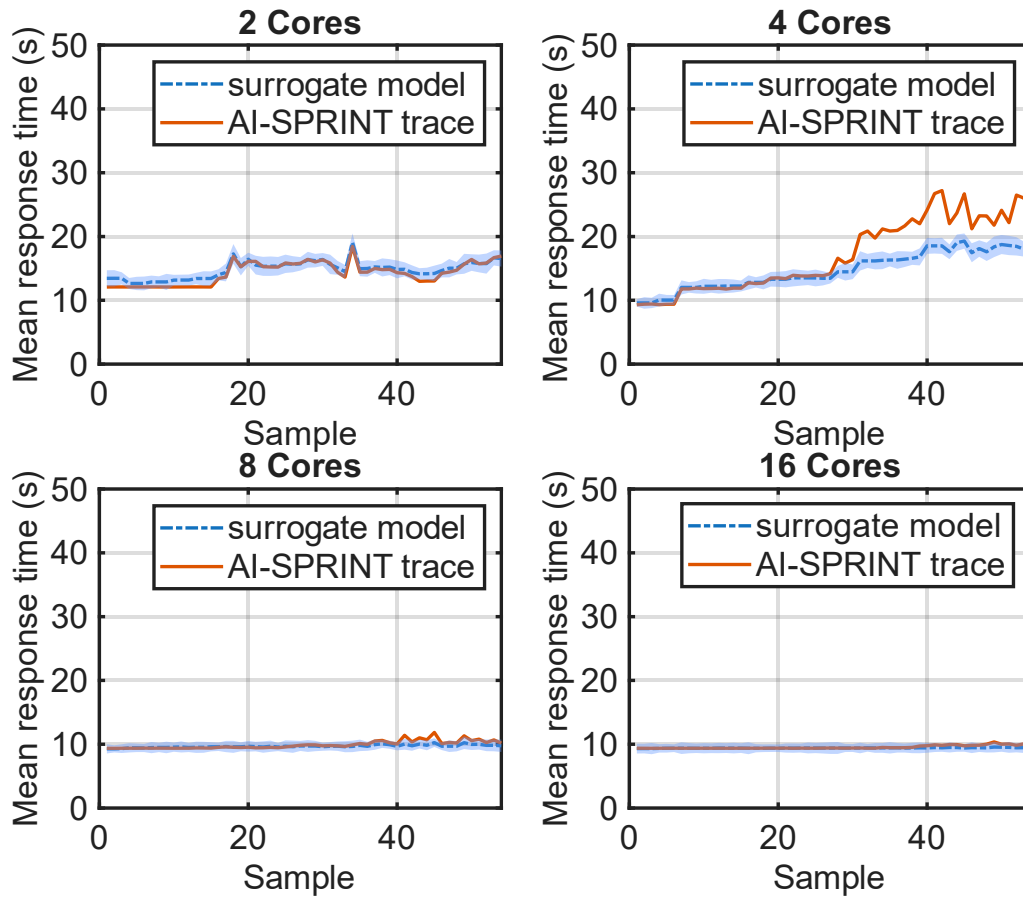


Figure 6.13: Comparison of the response time between trace-based simulations and deep surrogate model, with shaded 95% confidence intervals shown for the surrogate predictions.

variability, allowing us to approximate them using an Erlang distribution. This approximation is well-suited for time-to-event random variables with low variability in Markovian modeling, as the Erlang distribution accurately captures the first two moments, *i.e.*, the mean and the variance, effectively in this context.

For both the trace-driven simulation and our proposed deep surrogate model, the input parameters include the number of cores, the first two moments of the inter-arrival and service time distributions, and the pmf of batch sizes. In the trace-driven simulation, the complete response time for each sample is simulated based on the trace data, providing a comprehensive measure of response times for the entire tandem workflow. In the deep surrogate model, it predicts the moments of response time for each component in the tandem workflow. These component-level moments are then aggregated to estimate the overall response time for the entire system.

Figure 6.13 presents the comparison of response times between the surrogate model and the

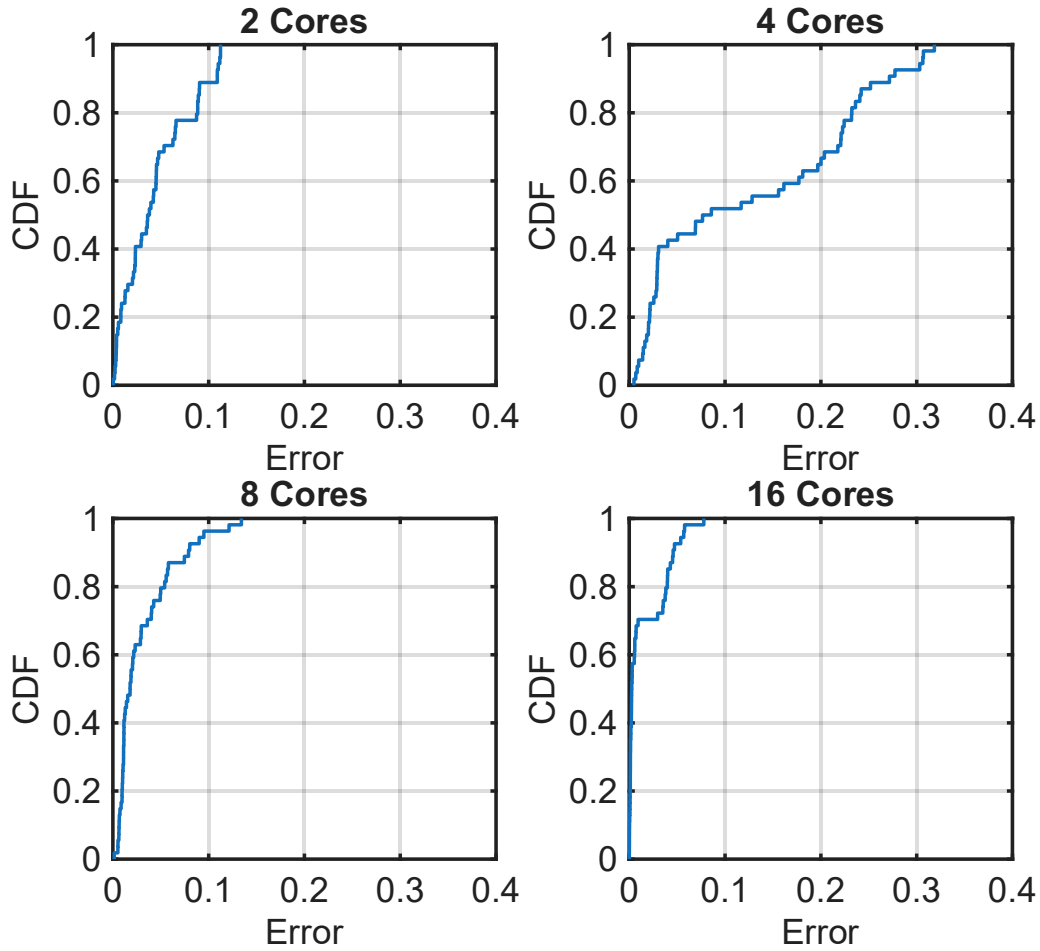


Figure 6.14: CDF of the prediction error of the deep surrogate model

AI-SPRINT trace across four core configurations. The surrogate model demonstrates a strong alignment with the AI-SPRINT trace, achieving an overall MAPE of 5.33%, indicating its effectiveness in capturing the system behavior. For the 2-core, 8-core, and 16-core configurations, response times are consistently below 20 seconds showing minimal deviation. In the 4-core configuration, a slight fluctuation is observed, where the AI-SPRINT trace shows an increase in response times to around 20-25 seconds, while the surrogate model captures the overall trend but with a degree of error. This discrepancy is likely due to sudden bursts of high load in the real system, managed by Kubernetes, leading to temporary resource contention and scheduling delays.

We conjecture that during the 4-core experiments, the system may have entered an unusual state midway through the test, differing from the typical conditions observed in other tests. However, we could not firmly confirm this as we did not have direct access to the internals. Indeed, the

strong alignment in the first half of the experiment with 4 cores shows that under normal conditions this case can also be matched fairly accurately. The overall alignment underscores the robustness of the model and its ability to generalize across diverse configurations, accurately predicting system performance in real-world scenarios.

Figure 6.14 shows the CDF of the prediction error (MAPE) of the proposed surrogate model under different core configurations. The CDF of the error across 2, 8, and 16-core configurations demonstrates that the surrogate model consistently achieves high accuracy, with most errors remaining below 0.1. These cases reflect strong alignment with the AI-SPRINT trace, indicating the robustness of the model under dynamic environments. The 4-core configuration stands out as an outlier, with maximum errors extending up to 30%, which are still acceptable from a practical standpoint. It should be noted that, nevertheless, the median error is far lower and equal to 10%. Overall, the CDF analysis demonstrates that our proposed surrogate model achieves high accuracy across different configurations.

6.6 Summary

In this chapter, we proposed a surrogate modeling approach for performance prediction in serverless edge computing systems. We began by analyzing AI-SPRINT application traces, with a focus on batch arrival patterns. Next, we formulated a tandem queueing network model and applied a decomposition technique to analyze each component in isolation using BMMPP/GI/c queues. To address the computational complexity of these analytical methods, we developed a deep surrogate model that efficiently approximates key performance metrics, such as mean response time. Finally, we conducted extensive simulations to validate the surrogate model, using both synthetic workloads and real-world AI-SPRINT traces, demonstrating that the surrogate model achieves high accuracy and effectively captures the performance characteristics of serverless edge computing systems.

Chapter 7

Conclusion

In this chapter, we summarize the contributions made in each of the thesis chapters and outline future research directions based on the current findings.

7.1 Summary of Thesis Achievements

In this thesis, we tackle critical challenges in performance modeling within distributed microservice architectures and serverless computing systems, proposing novel analytical approaches to accurately capture the dynamic behavior and interactions in these complex environments. Our research objectives **RO1-RO4** have been systematically explored through the development of innovative methodologies and models. Below, we summarize the achievements across Chapters 3 to 6.

We address our first research objective, **RO1**, in Chapter 3, which focuses on improving layered queueing network (LQN) modeling for distributed microservice architectures. We divide **RO1** into two sub-objectives: **RO1.1**, the development of a meta-solver for LQNs, and **RO1.2**, improving the efficiency and accuracy of layer solutions. To address **RO1.1**, we have introduced LN, an open-source meta-solver framework for LQN analysis. This framework supports dynamic selection of performance analysis paradigms for each layer. Through this approach, we

significantly reduce LQN solution errors and improve the generality of the models. For **RO1.2**, we have developed novel algorithms to improve the analysis of homogeneous queueing networks. These include (1) a method-of-moments algorithm that efficiently solves homogeneous models in linear time, (2) an algorithm for multiserver stations that achieves significant speedups for large multiclass models, (3) Gaussian quadrature methods for approximating normalizing constants, balancing accuracy and computational speed, and (4) a non-iterative MVA-based approximation that delivers high accuracy with less computational complexity than existing iterative methods.

Our second research objective, **RO2**, is explored in Chapter 4. This chapter introduces the integrated caching-queueing (ICQ) model, which enables joint analysis of data access patterns, caching, and queueing contention in edge computing environments. We divide **RO2** into two sub-objectives: **RO2.1**, modeling LQN for distributed workflows, and **RO2.2**, generalizing LQN to incorporate caching characteristics. To address **RO2.1**, we have applied LQN modeling to microservice placement in edge computing. This approach leads to a 16% improvement in system response time for stochastic workloads, proving the effectiveness of LQN modeling in distributed workflows. To address **RO2.2**, we have further generalized the LQN model to include caching components, enabling the analysis of data processing workflows that capture computation and data access dynamics simultaneously. The resulting ICQ model is evaluated with Azure real-world traces, demonstrating a 35% improvement in response time and a significant reduction in memory usage.

In Chapter 5, we tackle our third research objective, **RO3**, by focusing on the development of hierarchical caching models aimed at enhancing data locality within distributed microservice architectures. We divide **RO3** into two sub-objectives: **RO3.1**, the formulation of a hierarchical caching model, and **RO3.2**, the development of analytical solutions under different caching policies. To address **RO3.1**, we have developed a hierarchical caching model that accurately describes storage locations for data items within a distributed network of in-memory storage nodes. To address **RO3.2**, We have developed exact solutions for the caching models under random-replacement (RR) policy and applied a time-to-live (TTL) approximation for the least-recently-used (LRU) policy. Our simulations, utilizing real-world Alibaba traces, demon-

strate the effectiveness of this approach, resulting in a 21% reduction in miss ratio and a 37% improvement in response time.

Lastly, we address **RO4** in Chapter 6, where we introduce a surrogate modeling approach for performance prediction in microservices operating within serverless edge computing environments. We break **RO4** into two sub-objectives: **RO4.1**, formulating an analytical performance model, and **RO4.2**, developing a deep surrogate model. To address **RO4.1**, we have formulated an analytical performance model to capture the effects of batch arrivals and distributed workflow dependencies in serverless edge computing systems. To address **RO4.2**, we have proposed a deep surrogate model that approximates the performance metrics of the analytical model with high accuracy. Extensive simulations using AI-SPRINT traces validate this approach, achieving a MAPE of 5.33%, demonstrating its suitability for fast and scalable performance evaluation in serverless edge computing environments.

7.2 Future Work

Based on the contributions presented in this thesis, there are several potential directions for extending this research. These opportunities primarily focus on advancing performance modeling techniques and enhancing management strategies for cloud-native applications.

First, with respect to the LQN meta-solver development in Chapter 3, future work may explore solver-independent methods for LQN interlock correction. This could involve the creation of generalized techniques that address interlock issues without being tied to specific solvers, thereby improving the robustness and adaptability of the LQN solutions. In addition, learning-based techniques could be developed to automate the solver selection process. By moving away from the current reliance on heuristic rules, such as those outlined in Table 3.1, this approach would allow for the dynamic adaptation of solver strategies, especially as new solvers and solution methods are introduced. This would ultimately lead to more efficient, scalable, and flexible performance analysis in complex microservice and serverless environments.

Building on the contributions from Chapters 4 and 5, future work could focus on enhancing run-

time optimization techniques for data caching and processing within distributed microservice environments. By developing adaptive runtime strategies, the system could monitor performance metrics such as cache hit rates, queue lengths, and system latency in real-time, allowing for dynamic adjustments to caching and queueing parameters. This could involve predictive algorithms or machine learning models that anticipate workload fluctuations and optimize cache allocations and queue handling proactively. Additionally, by continuously evaluating cache performance and dynamically distributing data across caches, the system could minimize response times, improve resource utilization, and maintain low latency in varying workload conditions. Such advancements would significantly enhance the efficiency of data processing and caching in distributed systems, ensuring optimal system performance and scalability.

For Chapter 6, future research could extend the surrogate modeling techniques to more complex system topologies beyond the current tandem configurations. Specifically, exploring non-tandem topologies such as mesh or tree-structured networks would allow for a broader range of microservice and serverless computing architectures to be accurately modeled. Additionally, incorporating multiclass batch arrivals into the surrogate models could further enhance their applicability by addressing more diverse workload patterns, where different classes of tasks arrive simultaneously with varying resource demands. These extensions would broaden their applicability within cloud-native and serverless environments, enabling more precise and scalable performance predictions for a wider range of workloads.

Bibliography

- [AAE16] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.
- [ABB⁺16] Elvio G. Amparore, Gianfranco Balbo, Marco Beccuti, Susanna Donatelli, and Giuliana Franceschinis. *30 Years of GreatSPN*, pages 227–254. Springer, 2016.
- [ADM18] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.
- [AFN16] Sara Alouf, Nicaise Choungmo Fofack, and Nedko Nedkov. Performance models for hierarchy of caches: Application to modern DNS caches. *Performance Evaluation*, 97:57–82, 2016.
- [AL22] Santiago Andrés Azcoitia and Nikolaos Laoutaris. A survey of data marketplaces and their business models. *ACM SIGMOD Record*, 51(3):18–29, 2022.
- [Ama15] EC Amazon. Amazon web services. Available in: <http://aws.amazon.com/es/ec2/>(November 2012), 39, 2015.
- [ann] Annotated bibliography of papers related to layered queueing. <http://www.sce.carleton.ca/rads/lqns/papers/>. Accessed: 2023-02-23.

- [APC⁺15] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015.
- [Asm00] Søren Asmussen. Matrix-analytic models and their analysis. *Scandinavian Journal of Statistics*, 27(2):193–226, 2000.
- [AVZ19] Angelos Avelklouris, Maria Vlasidou, and Bert Zwart. Bounds and limit theorems for a layered queueing model in electric vehicle charging. *Queueing Systems*, 93(1-2):83–137, 2019.
- [BBA84] Steven C Bruell, Gianfranco Balbo, and PV Afshari. Mean value analysis of mixed, multiple class BCMP networks with load dependent service stations. *Performance Evaluation*, 4(4):241–260, 1984.
- [BCF⁺99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM’99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999.
- [BCMP75] Forest Baskett, K Mani Chandy, Richard R Muntz, and Fernando G Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM (JACM)*, 22(2):248–260, 1975.
- [BCS07] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. The JMT simulator for performance evaluation of non-product-form queueing networks. In *40th Annual Simulation Symposium (ANSS’07)*, pages 3–10. IEEE, 2007.
- [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.
- [BHH21] Pierre Blanchard, Desmond J Higham, and Nicholas J Higham. Accurately computing the log-sum-exp and softmax functions. *IMA Journal of Numerical Analysis*, 41(4):2311–2330, 2021.
- [BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.
- [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.
- [BLM22] Dario A Bini, Guy Latouche, and Beatrice Meini. A family of fast fixed point iterations for M/G/1-type Markov chains. *IMA Journal of Numerical Analysis*, 42(2):1454–1477, 2022.
- [BMR00] Dario A Bini, Beatrice Meini, and V Ramaswami. Analyzing M/G/1 paradigms through QBDs: The role of the block structure in computing the matrix G. *Advances in Algorithmic Methods for Stochastic Models*, pages 73–86, 2000.
- [BONC21] Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. Performance prediction for convolutional neural networks on edge GPUs. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, pages 54–62, 2021.
- [BOP22] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.
- [Buz73] Jeffrey P Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, 1973.

- [BWB⁺19] Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2114–2129, 2019.
- [Cas09] Giuliano Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models. *IEEE Transactions on Software Engineering*, 35(2):162–177, 2009.
- [Cas11] Giuliano Casale. Exact analysis of performance models by the method of moments. *Performance Evaluation*, 68(6):487–506, 2011.
- [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):1–25, 2017.
- [Cas20] Giuliano Casale. Integrated performance evaluation of extended queueing network models with LINE. In *2020 Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, 2020.
- [CCP21] Claudio Cicconetti, Marco Conti, and Andrea Passarella. A decentralized framework for serverless edge computing in the Internet of Things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2021.
- [CDT18] Tomas Cerny, Michael J Donahoo, and Michal Trnka. Contextual understanding of microservice architecture: Current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45, 2018.
- [CG20] Giuliano Casale and Nicolas Gast. Performance analysis methods for list-based caches with non-uniform access. *IEEE/ACM Transactions on Networking*, 29(2):651–664, 2020.
- [CGNZ22] Giuliano Casale, Yicheng Gao, Zifeng Niu, and Lulai Zhu. LN: A meta-solver for layered queueing network analysis. In *International Conference on Quantitative Evaluation of Systems*, pages 232–254. Springer, 2022.

- [CGNZ24] Giuliano Casale, Yicheng Gao, Zifeng Niu, and Lulai Zhu. LN: A flexible algorithmic framework for layered queueing network analysis. *ACM Transactions on Modeling and Computer Simulation*, 34(3):1–26, 2024.
- [CHO21] Giuliano Casale, Peter G. Harrison, and Wai Hong Ong. Facilitating load-dependent queueing analysis through factorization. *Performance Evaluation*, 152:102241, 2021.
- [CHW75] K. Mani Chandy, Ulrich Herzog, and Lin Woo. Parametric analysis of queueing networks. *IBM Journal of Research and Development*, 19(1):36–42, 1975.
- [CN82] K Mani Chandy and Doug Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *Communications of the ACM*, 25(2):126–134, 1982.
- [Com18] Multi-Access Edge Computing. Multi-access edge computing (MEC); Phase 2: Use cases and requirements. *MEC/001_099/002/02.01.01_60/gs-MEC002v020101p. pdf*, 2018.
- [CPW15] Giuliano Casale, Juan F Pérez, and Weikun Wang. QD-AMVA: Evaluating systems with queue-dependent service requirements. *Performance Evaluation*, 91:80–98, 2015.
- [CRF⁺18] Jaime Correia, Fábio Ribeiro, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Response time characterization of microservice-based systems. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–5, 2018.
- [CTH14] Giuliano Casale, Mirco Tribastone, and Peter G. Harrison. Blending randomness in closed queueing network models. *Performance Evaluation*, 82:15–38, 2014.
- [CTW02] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

- [CZ16] Mung Chiang and Tao Zhang. Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.
- [DB78] Peter J. Denning and Jeffrey P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, 1978.
- [Deh19] Zhamak Dehghani. How to move beyond a monolithic data lake to a distributed data mesh. *Martin Fowler’s Blog*, 2019.
- [dSPPSM23] Thiago Felipe da Silva Pinheiro, Paulo Pereira, Bruno Silva, and Paulo Maciel. A performance modeling framework for microservices-based cloud infrastructures. *The Journal of Supercomputing*, 79(7):7762–7803, 2023.
- [DWT⁺18] Bin Dong, Teng Wang, Houjun Tang, Quincey Koziol, Kesheng Wu, and Suren Byna. ARCHIE: Data analysis acceleration with array caching in hierarchical storage. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 211–220. IEEE, 2018.
- [EE13] Hala ElAarag and Hala ElAarag. A quantitative study of web cache replacement strategies using simulation. *Web Proxy Cache Replacement Strategies: Simulation, Implementation, and Performance Evaluation*, pages 17–60, 2013.
- [FAOW⁺08] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2008.
- [FJG20] Chen-Fu Fan, Anshul Jindal, and Michael Gerndt. Microservices vs serverless: A performance comparison on a cloud-native web application. In *CLOSER*, pages 204–215, 2020.
- [FM98] Paola Favati and Beatrice Meini. Relaxed functional iteration techniques for the numerical solution of M/G/1 type Markov chains. *BIT Numerical Mathematics*, 38(3):510–526, 1998.

- [FNNT14] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.
- [Fra00] Gregory Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton University, 2000.
- [FRR12] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. In *2012 24th International Teletraffic Congress (ITC 24)*, pages 1–8. IEEE, 2012.
- [FSA23] Federica Filippini, Hamta Sedghani, and Danilo Ardagna. SPACE4AI-R: A run-time management tool for AI applications component placement and resource scaling in computing continua. In *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, pages 1–7, 2023.
- [FZC⁺21] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. QoS-aware and resource efficient microservice deployment in cloud-edge continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 932–941. IEEE, 2021.
- [GC20] Alim Ul Gias and Giuliano Casale. COCOA: Cold start aware capacity planning for function-as-a-service platforms. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020.
- [GC22] Yicheng Gao and Giuliano Casale. JCSP: Joint caching and service placement for edge computing systems. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2022.
- [GCS24] Yicheng Gao, Giuliano Casale, and Rekha Singhal. Performance modeling of distributed data processing in microservice applications. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2024.

- [GCW19] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004. IEEE, 2019.
- [GD18] Yu Gan and Christina Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.
- [GGF⁺23] Enrico Galimberti, Bruno Guindani, Federica Filippini, Hamta Sedghani, Danilo Ardagna, Germán Moltó, and Miguel Caballer. OSCAR-P and aMLLibrary: performance profiling and prediction of computing continua applications. In *ACM/SPEC ICPE 2023 Companion*, pages 139–146, 2023.
- [GGS⁺23] Alim Ul Gias, Yicheng Gao, Matthew Sheldon, José A Perusquía, Owen O’Brien, and Giuliano Casale. SampleHST: Efficient on-the-fly selection of distributed traces. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2023.
- [GGS⁺24] Alim Ul Gias, Yicheng Gao, Matthew Sheldon, José A Perusquía, Owen O’Brien, and Giuliano Casale. SampleHST-X: A point and collective anomaly-aware trace sampling pipeline with approximate half space trees. *Journal of Network and Systems Management*, 32(3):1–38, 2024.
- [GH15] N. Gast and B. Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 123–136. ACM, 2015.
- [GIM18] Marco Gribaudo, Mauro Iacono, and Daniele Manini. Performance evaluation of replication policies in microservice based architectures. *Electronic Notes in Theoretical Computer Science*, 337:45–65, 2018.
- [GLD⁺21] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ML-driven performance debugging in microservices. In

- Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.
- [GVH16] Nicolas Gast and Benny Van Houdt. Asymptotically exact TTL-approximations of the cache replacement algorithms LRU(m) and h-LRU. In *2016 28th International Teletraffic Congress (ITC 28)*, volume 01, pages 157–165, 2016.
- [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):1–28, 2017.
- [GZC⁺19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.
- [He14] Qi-Ming He. *Fundamentals of matrix-analytic methods*, volume 365. Springer, 2014.
- [HH21] Sören Henning and Wilhelm Hasselbring. Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures. *Big Data Research*, 25:100209, 2021.
- [HHG06] Armin Heindl, Gábor Horváth, and Karsten Gross. Explicit inverse characterizations of acyclic maps of second order. In *Formal Methods and Stochastic Models for Performance Evaluation: Third European Performance Engineering Workshop, EPEW 2006, Budapest, Hungary, June 21-22, 2006. Proceedings 3*, pages 108–122. Springer, 2006.
- [HHT10] András Horváth, Gábor Horváth, and Miklós Telek. A joint moments based analysis of networks of MAP/MAP/1 queues. *Performance Evaluation*, 67(9):759–778, 2010.

- [HLC20] Jiwei Huang, Songyuan Li, and Ying Chen. Revenue-optimal task scheduling and resource management for IoT batch jobs in mobile edge computing. *Peer-to-Peer Networking and Applications*, 13(5):1776–1787, 2020.
- [HLL⁺21] Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Shaolei Ren, Jingwen Leng, Quan Chen, and Minyi Guo. AlphaR: Learning-powered resource management for irregular, dynamic microservice graph. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 797–806. IEEE, 2021.
- [HNN⁺18] Dinh Thai Hoang, Dusit Niyato, Diep N Nguyen, Eryk Dutkiewicz, Ping Wang, and Zhu Han. A dynamic edge caching framework for mobile 5G networks. *IEEE Wireless Communications*, 25(5):95–103, 2018.
- [HVHK⁺17] Robert Heinrich, André Van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance engineering for microservices: Research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 223–226, 2017.
- [IHM00] L Ismail, D Hagimont, and J Mossiere. Evaluation of the mobile agents technology: Comparison with the client/server paradigm. *Information Science and Technology (IST)*, 19, 2000.
- [JH09] Jing Jia and Sunderesh S Heragu. Solving semi-open queuing networks. *Operations research*, 57(2):391–401, 2009.
- [JL82] Patricia A. Jacobson and Edward D Lazowska. Analyzing queueing networks with simultaneous resource possession. *Communications of the ACM*, 25(2):142–151, 1982.
- [JM18] Fredrik Johansson and Marc Mezzarobba. Fast and rigorous arbitrary-precision computation of Gauss-Legendre quadrature nodes and weights. *SIAM Journal on Scientific Computing*, 40(6):C726–C747, 2018.

- [JPG19] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32, 2019.
- [JR04] Predrag R Jelenković and Ana Radovanović. Least-Recently-Used caching with dependent requests. *Theoretical Computer Science*, 326(1-3):293–327, 2004.
- [KBBML16] Hamzeh Khazaei, Cornel Barna, Nasim Beigi-Mohammadi, and Marin Litoiu. Efficiency analysis of provisioning microservices. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 261–268. IEEE, 2016.
- [KMBL20] Hamzeh Khazaei, Nima Mahmoudi, Cornel Barna, and Marin Litoiu. Performance modeling of microservice platforms. *IEEE Transactions on Cloud Computing*, 10(4):2848–2862, 2020.
- [KSR⁺19] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. GrandSLAM: Guaranteeing SLAs for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.
- [Lam82] Simon S. Lam. Dynamic scaling and growth behavior of queueing network normalization constants. *Journal of the ACM*, 29(2):492–513, 1982.
- [Lam11] Winfried Lamersdorf. Paradigms of distributed software systems: Services, processes and self-organization. In *International Conference on E-Business and Telecommunications*, pages 33–40. Springer, 2011.
- [LC24] Zhuoyuan Li and Giuliano Casale. Matrix network analyzer: A new decomposition algorithm for phase-type queueing networks (work in progress paper). In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, pages 34–39, 2024.
- [Lob19] Ramon Lobato. *Netflix nations: The geography of digital distribution*. New York University Press, 2019.

- [LRC⁺18] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.
- [LSX⁺19] Zhizhong Liu, Quan Z Sheng, Xiaofei Xu, Dianhui Chu, and Wei Emma Zhang. Context-aware and adaptive QoS prediction for mobile edge computing services. *IEEE Transactions on Services Computing*, 15(1):400–413, 2019.
- [Luc93] David M Lucantoni. The BMAP/G/1 queue: A tutorial. In *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 330–358. Springer, 1993.
- [LXL⁺21] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.
- [LZGS84] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: Computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.
- [LZS⁺21] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. *arXiv preprint arXiv:2103.00170*, 2021.
- [MAKR19] Raja Mubashir Munaf, Jawwad Ahmed, Faraz Khakwani, and Tauseef Rana. Microservices architecture: Challenges and proposed conceptual design. In *2019 International Conference on Communication Technologies (ComTech)*, pages 82–87. IEEE, 2019.
- [MAN14] Mohammad Ali Maddah-Ali and Urs Niesen. Fundamental limits of caching. *IEEE Transactions on Information Theory*, 60(5):2856–2867, 2014.

- [MB17] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.
- [MCS22] Inês Araújo Machado, Carlos Costa, and Maribel Yasmina Santos. Data mesh: Concepts and principles of a paradigm shift in data architectures. *Procedia Computer Science*, 196:263–271, 2022.
- [Mer14] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.
- [Mey89] Carl D. Meyer. Stochastic complementation, uncoupling Markov chains, and the theory of nearly reducible systems. *SIAM Review*, 31(2):240–272, 1989.
- [MFC⁺19] Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, I-Hsiu Liu, and Ci-Wei Lan. Graph-based and scenario-driven microservice analysis, retrieval, and testing. *Future Generation Computer Systems*, 100:724–735, 2019.
- [MK20a] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, 10(4):2834–2847, 2020.
- [MK20b] Nima Mahmoudi and Hamzeh Khazaei. Temporal performance modelling of serverless computing platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 1–6, 2020.
- [MM84] James McKenna and Debasis Mitra. Asymptotic expansions and integral representations of moments of queue lengths in closed Markovian networks. *Journal of the ACM*, 31(2):346–360, 1984.
- [MM86] Debasis Mitra and James McKenna. Asymptotic expansions for closed Markovian networks with state-dependent service rates. *Journal of the ACM*, 33(3):568–592, 1986.

- [Mou15] Adrian Mouat. *Using Docker: Developing and deploying software with containers*. O'Reilly Media, Inc., 2015.
- [MSB⁺23] Francisco Morais, Nuno Soares, João Bessa, José Vicente, Pedro Ribeiro, José Machado, and Ricardo J Machado. Converging data mesh and microservice principles into a unified logical architecture. In *International Conference on Intelligent Systems in Production Engineering and Maintenance*, pages 300–314. Springer, 2023.
- [MSK18] Jie Meng, Sang-Hyup Seo, and Hyun-Min Kim. Condition numbers and backward error of a matrix polynomial equation arising in stochastic models. *Journal of Scientific Computing*, 76:759–776, 2018.
- [MVdLvN16] Souley Madougou, Ana Varbanescu, Cees de Laat, and Rob van Nieuwpoort. The landscape of GPGPU performance modeling tools. *Parallel Computing*, 56:18–33, 2016.
- [MWL⁺20] Yiming Miao, Gaoxiang Wu, Miao Li, Ahmed Ghoneim, Mabrook Al-Rakhami, and M Shamim Hossain. Intelligent task prediction and computation offloading based on mobile-edge cloud computing. *Future Generation Computer Systems*, 102:925–931, 2020.
- [NC21] Zifeng Niu and Giuliano Casale. A mixture density network approach to predicting response times in layered systems. In *2021 29th International Symposium On Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2021.
- [Neu21] Marcel F Neuts. *Structured stochastic matrices of M/G/1 type and their applications*. CRC Press, 2021.
- [NMMA16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: Aligning principles, practices, and culture*. O'Reilly Media, Inc., 2016.

- [NRS⁺17] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.
- [NZP⁺19] Evangelos Ntontos, Uwe Zdun, Konstantinos Plakidas, Daniel Schall, Fei Li, and Sebastian Meixner. Supporting architectural decision making on data management in microservice architectures. In *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings 13*, pages 20–36. Springer, 2019.
- [PAT23] Riccardo Pincioli, Aldeida Aleti, and Catia Trubiani. Performance modeling and analysis of design patterns for microservice systems. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 35–46. IEEE, 2023.
- [PB03] Stefan Podlipnig and Laszlo Böszörményi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.
- [PBSJ19] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2019.
- [PC17] Juan F. Pérez and Giuliano Casale. LINE: Evaluating software applications in unreliable environments. *IEEE Transactions on Reliability*, 66(3):837–853, 2017.
- [pic] A guide to 5G small cells and macrocells. <https://www.essentracomponents.com/en-gb/news/guides/guide-to-5g-small-cells-and-macrocells>. Accessed: 2021-12-31.
- [PKT90] Krishna R Pattipati, Michael M Kostreva, and John L Teele. Approximate mean value analysis algorithms for queuing networks: Existence, uniqueness, and convergence results. *Journal of the ACM (JACM)*, 37(3):643–673, 1990.

- [PMC21] Aneta Poniszewska-Marańda and Ewa Czechowska. Kubernetes cluster for automating software production environment. *Sensors*, 21(5):1910, 2021.
- [PRN⁺19] Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, and Germán Moltó. On-premises serverless computing for event-driven data processing applications. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 414–421. IEEE, 2019.
- [PTVH12] Juan F Pérez, Miklós Telek, and Benny Van Houdt. A fast Newton’s iteration for M/G/1-type and GI/M/1-type Markov chains. *Stochastic Models*, 28(4):557–583, 2012.
- [PVDH07] Mike P Papazoglou and Willem-Jan Van Den Heuvel. Service oriented architectures: Approaches, technologies and research issues. *The VLDB journal*, 16:389–415, 2007.
- [PWCL22] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. Retention-aware container caching for serverless edge computing. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1069–1078. IEEE, 2022.
- [Ram88] V Ramaswami. A stable recursion for the steady state vector in Markov chains of M/G/1 type. *Stochastic Models*, 4(1):183–188, 1988.
- [RBÅC21] Johan Ruuskanen, Tommi Berner, Karl-Erik Årzén, and Anton Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *Performance Evaluation*, 151:102231, 2021.
- [RBT22] Francesca Randone, Luca Bortolussi, and Mirco Tribastone. Jump longer to jump less: Improving dynamic boundary projection with h-scaling. In Erika Ábrahám and Marco Paolieri, editors, *Quantitative Evaluation of Systems*, pages 150–170, Cham, 2022. Springer International Publishing.
- [RCK⁺09] Jerry Rolia, Giuliano Casale, Diwakar Krishnamurthy, Stephen Dawson, and Stephan Kraft. Predictive modelling of SAP ERP applications: Challenges and

- solutions. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 1–9, 2009.
- [Rei79] Martin Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE Transactions on Communications*, 27(8):1199–1209, 1979.
- [Rhe10] Noah H Rhee. Note on functional iteration technique for M/G/1 type Markov chains. *Linear Algebra and Its Applications*, 432(4):1042–1048, 2010.
- [RK75] Martin Reiser and Hisashi Kobayashi. Queuing networks with multiple closed chains: Theory and computational algorithms. *IBM Journal of Research and Development*, 19(3):283–294, 1975.
- [RL80] Martin Reiser and Stephen S Lavenberg. Mean-value analysis of closed multichain queueing networks. *Journal of the ACM (JACM)*, 27(2):313–322, 1980.
- [RM21] Estefanía Recayte and Andrea Munari. Caching at the edge: Outage probability. In *2021 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2021.
- [RMNB21] Sebastián Risco, Germán Moltó, Diana M Naranjo, and Ignacio Blanquer. Serverless workflows for containerised applications in the cloud continuum. *Journal of Grid Computing*, 19:1–18, 2021.
- [RS95] Jerome A. Rolia and Kenneth C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, 1995.
- [RS02a] Alma Riska and Evgenia Smirni. MAMsolver: A matrix analytic methods tool. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 205–211. Springer, 2002.
- [RS02b] Alma Riska and Evgenia Smirni. M/G/1-type Markov processes: A tutorial. *Performance Evaluation of Complex Systems: Techniques and Tools: Performance 2002 Tutorial Lectures*, pages 36–63, 2002.

- [RS07] Alma Riska and Evgenia Smirni. ETAQA solutions for infinite Markov processes with repetitive structure. *INFORMS Journal on Computing*, 19(2):215–228, 2007.
- [RSB01] Pablo Rodriguez, Christian Spanner, and Ernst W Biersack. Analysis of web caching architectures: Hierarchical and distributed caching. *IEEE/ACM Transactions On Networking*, 9(4):404–418, 2001.
- [RW93] Keith W Ross and Jie Wang. Asymptotically optimal importance sampling for product-form queuing networks. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 3(3):244–268, 1993.
- [SAM⁺21] Hamta Sedghani, Danilo Ardagna, Matteo Matteucci, Giulio Angelo Fontana, Giacomo Verticale, Fabrizio Amarilli, Rosa Badia, Daniele Lezzi, Ignacio Blanquer, André Martin, et al. Advancing design and runtime management of AI applications with AI-SPRINT (position paper). In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1455–1462. IEEE, 2021.
- [Say18] Gigi Sayfan. *Mastering Kubernetes: Master the art of container management by using the power of Kubernetes*. Packt Publishing Ltd, 2018.
- [SD11] Yasir Shoaib and Olivia Das. Web application performance modeling using layered queueing networks. *Electronic Notes in Theoretical Computer Science*, 275:123–142, 2011.
- [SFG⁺20] Mohammad Shahradd, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Bantum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423*, 2020.

- [SHLP05] M-T Schmidt, Beth Hutchison, Peter Lambros, and Rob Phippen. The enterprise service bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, 2005.
- [Sil16] Alan Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.
- [SJZY⁺16] Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 318–325, 2016.
- [SLP10] Connie U. Smith, Catalina M. Lladó, and Ramon Puigjaner. Performance model interchange format (PMIF 2): A comprehensive approach to queueing network model interoperability. *Performance Evaluation*, 67(7):548–568, 2010.
- [SMBB21] Isak Shabani, Endrit Mëziu, Blend Berisha, and Tonit Biba. Design of modern distributed systems based on microservices architecture. *International Journal of Advanced Computer Science and Applications*, 12(2), 2021.
- [SRBR19] Adalberto R Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S Rosa. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, 10:1–30, 2019.
- [SSSO87] Abraham Seidmann, Paul J Schweitzer, and Sarit Shalev-Oren. Computerized closed queueing network models of flexible manufacturing systems: A comparative evaluation. *Large Scale Systems*, 12(2):91–107, 1987.
- [SWG21] Christoph Schröer, Sven Wittfoth, and Jorge Marx Gómez. A process model for microservices design and identification. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, pages 1–8. IEEE, 2021.
- [SZY⁺16] Tasneem Salah, M Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. The evolution of distributed systems towards microservices

- architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 318–325. IEEE, 2016.
- [Tri13] Mirco Tribastone. A fluid model for layered queueing networks. *IEEE Transactions on Software Engineering*, 39(6):744–756, 2013.
- [TXY⁺22] Qinqin Tang, Renchao Xie, Fei Richard Yu, Tianjiao Chen, Ran Zhang, Tao Huang, and Yunjie Liu. Distributed task scheduling in serverless edge computing networks for the Internet of Things: A learning approach. *IEEE Internet of Things Journal*, 9(20):19634–19648, 2022.
- [UPK⁺21] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.
- [WC13] Weikun Wang and Giuliano Casale. Bayesian service demand estimation using Gibbs sampling. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 567–576. IEEE, 2013.
- [WDH⁺21] Yang Wang, Hao Dai, Xinxin Han, Pengfei Wang, Yong Zhang, and Chengzhong Xu. Cost-driven data caching in edge-based content delivery networks. *IEEE Transactions on Mobile Computing*, 22(3):1384–1400, 2021.
- [WDJ20] Sheng Wang, Zhijun Ding, and Changjun Jiang. Elastic scheduling for microservice applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):98–115, 2020.
- [WGZ⁺19] Shangguang Wang, Yan Guo, Ning Zhang, Peng Yang, Ao Zhou, and Xuemin Shen. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. *IEEE Transactions on Mobile Computing*, 20(3):939–951, 2019.

- [Whi83] Ward Whitt. The queueing network analyzer. *The Bell System Technical Journal*, 62(9):2779–2815, 1983.
- [WJY⁺22] Chenyang Wang, Bosen Jia, Hao Yu, Xiuhua Li, Xiaofei Wang, and Tarik Taleb. Deep reinforcement learning for dependency-aware microservice deployment in edge computing. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 5141–5146, 2022.
- [WLZ⁺22] Xinkai Wang, Chao Li, Lu Zhang, Xiaofeng Hou, Quan Chen, and Minyi Guo. Exploring efficient microservice level parallelism. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 223–233. IEEE, 2022.
- [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.
- [Wol16] Eberhard Wolff. *Microservices: Flexible software architecture*. Addison-Wesley Professional, 2016.
- [Woo13] Murray Woodside. Tutorial introduction to layered modeling of software performance. <http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialh.pdf>, 2013. Accessed: 2022-05-08.
- [WT16] Tabea Waizmann and Mirco Tribastone. DiffLQN: Differential equation analysis of layered queuing networks. In *Companion Publication for ACM/SPEC on International Conference on Performance Engineering*, pages 63–68. ACM, 2016.
- [XTQ⁺21] Renchao Xie, Qinqin Tang, Shi Qiao, Han Zhu, F. Richard Yu, and Tao Huang. When serverless computing meets edge computing: Architecture, challenges, and open issues. *IEEE Wireless Communications*, 28(5):126–133, 2021.
- [YCX⁺20] Yuyu Yin, Zengxu Cao, Yueshen Xu, Honghao Gao, Rui Li, and Zhida Mai. QoS prediction for service recommendation with features learning in mobile edge

- computing environment. *IEEE Transactions on Cognitive Communications and Networking*, 6(4):1136–1145, 2020.
- [YLRC19] Yoel G Yera, Rosa E Lillo, and Pepa Ramírez-Cobo. Fitting procedure for the two-state batch Markov modulated Poisson process. *European Journal of Operational Research*, 279(1):79–92, 2019.
- [YYG⁺19] Yinbo Yu, Jianfeng Yang, Chengcheng Guo, Hong Zheng, and Jiancheng He. Joint optimization of service request routing and instance placement in the microservice system. *Journal of Network and Computer Applications*, 147:102441, 2019.
- [Zah79] John Zahorjan. An exact solution method for the general class of closed separable queueing networks. *ACM SIGSIM Simulation Digest*, 11(1):107–112, 1979.
- [ZCP20] Lulai Zhu, Giuliano Casale, and Iker Perez. Fluid approximation of closed queueing networks with discriminatory processor sharing. *Performance Evaluation*, 139:102094, 2020.
- [ZCPR03] Wenyi Zhao, Rama Chellappa, P Jonathon Phillips, and Azriel Rosenfeld. Face recognition: A literature survey. *ACM Computing Surveys (CSUR)*, 35(4):399–458, 2003.
- [ZES88] John Zahorjan, Derek L Eager, and Hisham M Sweillam. Accuracy, speed, and convergence of approximate mean value analysis. *Performance Evaluation*, 8(4):255–270, 1988.
- [Zim17] Olaf Zimmermann. Microservices tenets: Agile approach to service development and deployment. *Computer Science-Research and Development*, 32:301–310, 2017.
- [ZSGK09] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Characteristics of YouTube network traffic at a campus network—measurements, models, and implications. *Computer networks*, 53(4):501–514, 2009.

- [ZXZ⁺20] Gongming Zhao, Hongli Xu, Yangming Zhao, Chunming Qiao, and Liusheng Huang. Offloading dependent tasks in mobile edge computing with service caching. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1997–2006. IEEE, 2020.
- [ZZWW22] Feng Zeng, Kanwen Zhang, Lin Wu, and Jinsong Wu. Efficient caching in vehicular edge computing based on edge-cloud collaboration. *IEEE Transactions on Vehicular Technology*, 72(2):2468–2481, 2022.

Appendix A

LN Software Architecture Design

Fig. A.1 illustrates the key architectural elements of LINE, including the `NetworkStruct` data structure, and the `Network`, `NetworkSolver`, `LayeredNetwork`, `EnsembleSolver` and `LayeredNetworkSolver` classes.

The `Network` object summarizes the model characteristics and acts as its persistence layer. The object is generated by the user either through a domain-specific language offered by LINE [Cas20] or via model-to-model transformations from other formats (*e.g.*, JMT's XML [BCS07], PMIF [SLP10]). Besides the model specification, a `Network` object can cache the model state space, its initial state, and retain information needed for the traffic equations in state-dependent models.

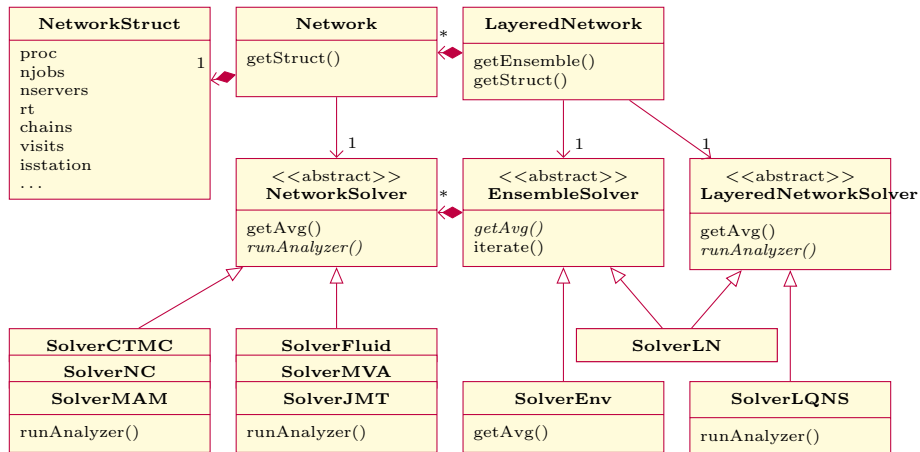


Figure A.1: Key architectural elements of LINE.

Each **Network** object can synthesize via the **getStruct** method a **NetworkStruct** data structure. The latter includes key model parameters, such as representations of service and arrival processes, job populations, and the number of servers, among others. In addition, the data structure includes the routing table, the associated chains, and the average number of visits that each class pays to each node. **NetworkStruct** also offers indexing functions, that allow for example to differentiate between *stations*, where jobs can reside, and *nodes*, which are elements of the network traversed with zero service time (*e.g.*, a fork).

The **NetworkSolver** object encodes a solver type, of which the aforementioned six LINE solvers are specific instances. The main role of this class is to ensure consistent computation of performance results, adopting identical conventions for reporting per-class and per-chain results to the end user. Operational relationships are also applied by this object to derive certain performance metrics from the ones returned by the solvers, *e.g.* arrival rates from throughputs [DB78].

Each **NetworkSolver** object is equipped with a **getAvg** method that returns mean performance metrics for the model in a tabular format. The method invokes via the **runAnalyzer** method one of the solution methods offered by that solver, which operates solely on the **NetworkStruct** data structure. Model transformations that alter the model topology are conducted within **runAnalyzer**. An example is tagging a job class, which is used in response time distribution analysis.

The **EnsembleSolver** specifies the life-cycle for an iterative solution method that works on an ensemble of **Network** objects. This class allows binding a **NetworkSolver** to each particular **Network** in the ensemble, applying actions before, in-between, and after each iteration, and verifying convergence. It also harmonizes the presentation of ensemble-level results to the end-user. Besides LN, the Env solver is another example of **EnsembleSolver**, wherein iteration is used to analyze random environments [CTH14].

The LN solver is a special instance of **EnsembleSolver**, operating on an ensemble consisting of the LQN layers. The **LayeredNetwork** class encompasses the objects that form an LQN, such as the **Entry**, **Task**, **Host**, and **Activity** classes.

The `LayeredNetwork` class offers a `getEnsemble` method that generates, and stores within the `LayeredNetwork` object, the ensemble of `Network` models, each mapping to a distinct LQN layer. Similarly to `Network`, this class also exposes a `getStruct` method that builds a static data structure of the LQN parameters.

The LN solver, implemented in the `SolverLN` class, is specified parametrically in terms of any of the LINE solvers, or a custom combination thereof. For example, the end-user may require to use LINE's simulators on layers that include non-Markovian service distributions (*e.g.*, Pareto) and MVA otherwise.