IMT School for Advanced Studies, Lucca
Lucca, Italy

Automatic and Accurate Performance Prediction in Distributed Systems

PhD Program in Systems Science

Track in Computer Science and Systems Engineering

XXXIII Cycle

By

Giulio Garbi

2023

The dissertation of Giulio Garbi is approved.

IMT School for Advanced Studies Lucca
2023

Contents

List of Figures

List of Tables

Acknowledgements

Chapter 4 text is based on in-progress work: Giulio Garbi, Emilio Incerto, and Mirco Tribastone. “Service Demands Estimation in Layered Queueing Networks” .

Vita

August 5, 1992	Born, Vercelli, Italy
2014	Bachelor in Computer Science Final mark: 110/110 cum laude (“*110/110 e lode*”) Università degli Studi di Torino, Turin, Italy
2016	Master in Computer Science Final mark: 110/110 summa cum laude (“*110/110 e lode con menzione per l’eccezionale curriculum*”) Università degli Studi di Torino, Turin, Italy
2017–2020	PhD Scholarship IMT Scuola Alti Studi Lucca, Lucca, Italy
2020–2021	Frontier Proposal Fellowship and Research Fellowship IMT Scuola Alti Studi Lucca, Lucca, Italy
February 2022	Erasmus+ for Traineeship DiffBlue, Oxford, UK

Publications

1. Chapter 3 text is based on Giulio Garbi, Emilio Incerto, and Mirco Tribastone, "Learning Queuing Networks by Recurrent Neural Networks," in *ICPE*, 2020.
2. Chapter 5 text is based on Giulio Garbi, Emilio Incerto, and Mirco Tribastone."μP: A Development Framework for Predicting Performance of Microservices by Design," in *IEEE CLOUD*, 2023.

Presentations

1. Giulio Garbi, Emilio Incerto and Mirco Tribastone, "Machine Learning Software Performance Models," at *InfQ Meeting*, Caserta, Italy, 2019.
2. Giulio Garbi, Emilio Incerto and Mirco Tribastone, "Machine Learning Software Performance Models," at *SIESTA International Summer School on Software Engineering*, Termoli (Campobasso), Italy, 2019.
3. Giulio Garbi, Emilio Incerto and Mirco Tribastone, "Learning Queuing Networks by Recurrent Neural Networks," at *ICPE*, Online event, 2020. Available online at `https://www.youtube.com/watch?v=LxTaV0FezgI`.

Abstract

System performance is getting attention by industry as it affects user experience, and much research focused on performance evaluation approaches. Profiling is the most straightforward approach to performance evaluation of software systems, despite being limited to shallow analyses. Conversely, software performance models excel in representing complex interactions between components. Still, practitioners do not integrate performance models in the software development cycle, as the learning curve is too steep, and the approaches do not adapt well to incremental development practices. In this thesis, we propose three approaches towards automatic learning of performance models. The first approach employs a Recurrent Neural Network (RNN) to extract a full Queueing Network (QN) model of the system; the second one calibrates a Layered Queueing Network (LQN) using an RNN; the third one presents μP, a framework that allows the user to develop microservice systems and obtain the corresponding LQN model from source code analysis. We considered the microservices architecture as it is embraced by influential players (e.g., Amazon, Netflix). Those approaches have two advantages: *i*) minimal user intervention to flatten the learning curve; *ii*) continuous synchronization between software and performance model, such as each software development iteration is reflected on the model. We validated our approaches on several benchmarks taken from the literature. The models we generate can be queried to predict the system behavior under conditions significantly different from the learning setting, and the results show sensible advancements in the quality of the predictions.

Chapter 1

Introduction

1.1 Motivation

Performance engineering has attracted much attention in the industry. Indeed, performance metrics such as throughput and response time are essential factors in users' willingness to interact with a software system, besides the standard functional requirements (e.g., correctness). Among the wealth of statements, we point out the reaction of two big players: Walmart (in 2012) is concerned that "We are not the fastest retail site on the internet today" [Bix12], and "page speed will be a ranking factor for mobile searches" in Google since 2018 [Goo18].

Performance engineering literature generally describes complex systems through models, which use mathematical abstractions to represent the various aspects of the system component's interactions [Bol+05; Ste07]. We can divide performance models into two categories: *black-box* and *white-box* performance models.

Black-box models describe the systems as a whole (hence they do not require in-deep knowledge of the system) but are limited to shallow analyses. Those models are generally easy to construct as they do not require in-deep knowledge of the system. The most known approach to black-box modeling is *profiling*. Profiling is performed by stimulating the system under different conditions (e.g., inputs, configuration files) and observing their effects on the system's performance. Tools like *Gprof* analyze the observed data and highlight the critical points that affect the overall system performance. Profiling can easily be automated as it does not require expertise in performance evaluation. However, it cannot generalize its findings beyond the tested inputs; therefore, it cannot be used in predictive analysis [ZH12].

Conversely, white-box models excel in representing complex interactions between components but require proficiency in the performance evaluation domain to obtain accurate results. Among white-box models, we cite Stochastic Petri Nets, Stochastic Process Algebras, and Queuing Networks [CMI11]. Unlike profiling, the explicit modeling of each component allows for predictive analysis. This allows for more extensive analysis in contexts not seen at the model construction phase, e.g., we can see how the system reacts under different load levels or when reallocating resources (*what-if* analysis). Unfortunately, this comes at the price of more demanding mathematical skills (to develop, test, and validate the models) and deep knowledge of the software under study. Those steep entry barriers prevent a broader usage in software development practices [WFP07].

Automatic performance model generation lessens the entry barrier by embedding performance annotations in software modeling languages but does not integrate seamlessly with modern software development practices. The most known example is MARTE [Obj07] extension to UML, which generates software artifacts (i.e., a code skeleton) and their associated performance models [Bal+04; Koz10]. This approach accounts only for the first artifact generation: subsequent modifications to the software require the developer to manually reflect the changes on the performance model [Gar+13]. Hence, this approach is unsuitable for current software development techniques (e.g., continuous integration, AGILE practices) that dictate short and frequent code revision cycles.

This thesis proposes three advancements to the state-of-the-art automatic performance model generation of distributed multithreaded systems starting from minimal but targeted profiling information. We will employ profiling at different levels of granularity to produce white-box performance models that keep a relation between the parts of the systems under analysis and the corresponding model parts. Those approaches combine the advantages of the techniques described in this section:

- the *simplicity* of profiling, applied to individual components of the system rather than the system as a whole;
- the *prediction power* of white-box models, as we describe the components individually;
- the *usability* of automatic modeling, as we do not require user intervention in the model construction procedure;
- the *incrementality* of new software development practices, as we can limit the model construction procedure to the modified part through the model-to-system component relation.

1.2 Main contribution

Here, we will describe the three approaches at the core of this thesis. These approaches share a common design pattern, shown in Figure 1. The approaches start from a distributed system and define a *targeted profiling* scheme that extracts the required data. The data is then processed by a *machine learning* algorithm that produces a *performance model*. We then evaluate the model quality by considering several scenarios the machine learning algorithm does not see, e.g., different load or distributed system deployment (*what-if configuration*). These modifications are applied both on the model and the original system, and we compare the predicted *performance metrics* (e.g., throughput, response time, and CPU utilization) with the ones observed under the new configuration.

We now move to a more detailed description of the methodologies. In the first approach, *Learning Queuing Networks by Recurrent Neural Networks* [GIT20], a Recurrent Neural Network (RNN) analyzes the number of pending requests (*queue length*) at each system component to produce a Queueing Network (QN). QN models attracted attention in the performance engineering literature as they are versatile in representing various

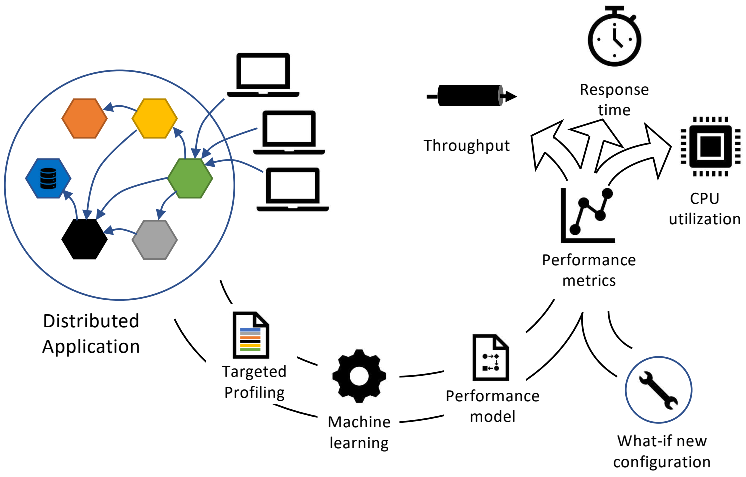
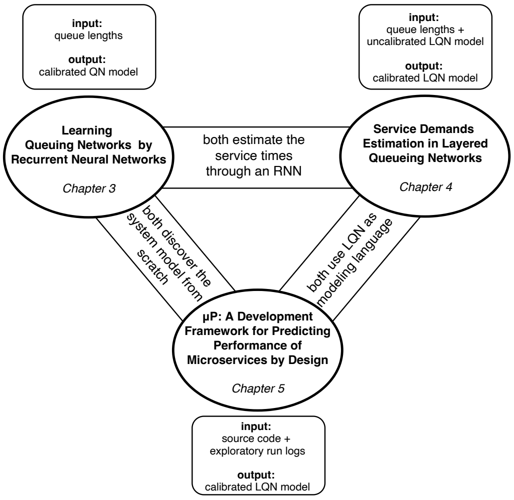


Figure 1: Design pattern of the approaches presented in this thesis.

systems, e.g., component-based systems [Koz10], web services [DI04], and adaptive systems [Arc+15; ITT17]. We refer the reader to Section 2.1 for a more complete description of the QN formalism. The only input of this approach is the components' queue lengths, obtainable through OS system calls; therefore, it is suitable for systems where the source code can not be modified.

In the second approach, *Service Demands Estimation in Layered Queueing Networks* [GIT], a RNN analyzes each system component to calibrate a Layered Queueing Network (LQN) that models the system. The LQN formalism [FAW+09] is an extension of QN that better captures the synchronization patterns between system components. Section 2.2 gives a broader description of the LQN formalism. This approach differs from the previous one as it calibrates a hand-made model rather than generating one, but it can capture both synchronous and asynchronous communication.

The third approach, *μP: A Development Framework for Predicting Performance of Microservices by Design* [GIT23], defines a framework (called μP)



input:
queue lengths
output:
calibrated QN model
input:
queue lengths +
uncalibrated LQN model
output:
calibrated LQN model
Learning Queuing Networks by Recurrent Neural Networks
Chapter 3
both estimate the service times through an RNN
Service Demands Estimation in Layered Queueing Networks
Chapter 4
both discover the system model from scratch
both use LQN as modeling language
μP: A Development Framework for Predicting Performance of Microservices by Design
Chapter 5
input:
source code +
exploratory run logs
output:
calibrated LQN model

Figure 2: Relations between the three approaches

to develop microservice systems with an associated performance model without further user input beyond writing the source code. Microservice systems are made of many simple components, usually deployed on cloud architectures, that interact with each other through a restful API [RR08]. We targeted those systems as they have industrial applications, e.g., Amazon, Netflix, and Spotify embraced the microservice architecture [Blo13; sta15; Gol15]. Section 1.3 will provide more insights into the microservice architecture. The proposed framework (called μP) provides an API that the developers use to code their system (similar to the popular Node.js or Java Spring Boot ones) and a set of point-and-click tools to extract the system model, modify the model, and predict the response time and cloud utilization under the new running conditions. Specifically, this approach produces an LQN performance model by analyzing access logs generated by μP's profiling infrastructure (included in the framework and entirely transparent for the developer).

Figure 2 exposes the relations between the three approaches. The least intrusive approach is the first one (see Chapter 3), albeit more limited in applicability. This approach requires only the queue lengths of each system component and produces a calibrated QN model of the system. It discovers the system model from scratch alike the third approach, but, similarly to the second approach, it uses an RNN to calibrate the model. This approach uses the QN formalism, which cannot capture some synchronization patterns found in distributed software systems.

The second approach (see Chapter 4) uses the more expressive LQN formalism and requires the user to provide an uncalibrated model of the system as well as the queue lengths of each component. This approach can be applied to existing software, as the queue lengths can be seamlessly monitored through OS calls. The user-provided uncalibrated model guides the construction of the RNN, which discovers the service times that best fit the queue lengths, leading to the calibrated model.

The third approach (see Chapter 5) starts from the source code of the microservice system and the logs of some exploratory runs of the system to produce a calibrated LQN model. In this approach, the user has to write the system using the μP framework; therefore it is more suited to new systems. Unlike the second approach, it does not require a user-provided description of the model, but it extracts the structure of the system from the source code. After that, μP uses the exploratory runs to extract the service times of the components.

1.3 The microservice architecture: an interesting case study

Among distributed systems, microservice architectures make a good case study for this thesis due to their popularity and unclear effect on system performance.

A microservice system comprises several small loosely coupled components called microservices [Ric18]. Each microservice performs a small set of functions (called endpoints) and relies on collaboration to perform its scope. Microservices collaborate by exchanging messages using language-agnostic protocols like HTTP or JSON. Microservices allow for a highly flexible deployment, where they might be added or removed at runtime; hence they do not know the network location of each other [Ric18].

Microservice architectures are popular for the extra-functional properties they guarantee [Ent19; Mod14; Gro19]. Performance is one of the considered properties, but it is hard to quantify how the deployment or the communication patterns influence it [Llo+18]. For example, in a survey of 17 in-depth interviews with software practitioners [BFW+19], only half of the interviewed subjects have a favorable opinion on the influence of the Microservice Architecture (MSA) on performance, while the other half remain neutral. With μP, we aim to close this gap by explicitly modeling the relationship between deployment, communication patterns, and observed performance.

1.4 Thesis organization

The remainder of the thesis is organized as follows. Chapter 2 presents the background of this thesis, i.e., the mathematical abstractions used in the following chapters, as well as a review of related works. Chapter 3 presents the QN estimation approach (*Learning Queuing Networks by Recurrent Neural Networks*). Chapter 4 discusses the LQN calibration approach (*Service Demands Estimation in Layered Queueing Networks*). Chapter 5 shows the μP framework and how it can be used to develop real systems (*μP: A Development Framework for Predicting Performance of Microservices by Design*). Finally, Chapter 6 wraps up and presents some future lines of research.

Chapter 2

Background

This section will briefly introduce the tools used in the technical part of this thesis (i.e., Chapters 3, 4, and 5). Section 2.1 presents the Queueing Networks (QN) performance model and its *fluid approximation*. Section 2.2 introduces the Layered extension on the QN model. Section 2.3 briefly introduces the Neural Networks, used in Chapters 3 and 4. Section 2.4 concludes with some lines of research related to this thesis.

2.1 Queueing Networks

The Queueing Networks (QN) model represents a system as a set of *stations*, each one performing a specific work upon request by *clients*. Stations perform their work through a number of indistinguishable servers, each one performing the station's work independently to other servers. For this thesis we assume that the time needed to complete the i-th station work follows an exponential distribution whose average is $1/\mu_i$. Clients represent customers of the system. Each client asks for work to a station, waits for a free server (on a FCFS policy) if there is none, and then get the required service by one server. We call *queue length* of a station i the number of clients that are being serviced or wait for a free server at i.

In this thesis, we study *closed* systems, where the number of clients is constant throughout time. This contrasts to *open* systems, where clients continuously enter and leave the system. More precisely, in a closed system, clients who got the requested service always ask for work at another station chosen probabilistically, while in open systems, they might (probabilistically) choose to exit the system. The closed system assumption is not restrictive: in our approaches, we use the clients to stress the systems' performance and probabilistic behavior, which does not depend on the client's identity. Rather, this assumption proves useful for two reasons. Firstly, in the RNN-based methods, we can simplify the learning error function (see Sections 3.1.4 and 4.2.2, where the denominator of the formula is constant). Secondly, this allows the stress of the systems under heavy load without incurring the effects of a potentially unbounded number of clients. In a closed system, load is realized by having many clients. Once the system is saturated, the clients need more time to traverse the network; therefore, their request rate (i.e., how many new clients perform a new request per unit of time) is indirectly regulated so the system can handle them. In an open system, load is realized by raising the request rate of the system. If the system is in severe saturation, the arrival rate is larger than the exit rate (the number of clients served per unit of time). Since there is no link between exit and request rate, the number of clients in the system grows indefinitely and leads to spurious behavior (e.g., connection timeouts, OS running out of resources).

Formally, we define a closed QN by the following:

- N: the number of clients in the network;
- M: the number of stations;
- $\mathrm{s} = (s_1, \ldots, s_M)$: the vector of concurrency levels, where s_i gives the number of servers at station i, with $1 \leq i \leq M$;
- $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_M)$: the vector of *service rates*, i.e., $1/\mu_i > 0$ is the mean service demand at station i, with $1 \leq i \leq M$;
- $\mathbf{P} = (\mathbf{P}_{i,j})_{1 \leq i,j \leq M}$: the *routing probability matrix*, where each element $\mathbf{P}_{i,j} \geq 0$ gives the probability that a client goes to station j upon completion at station i;

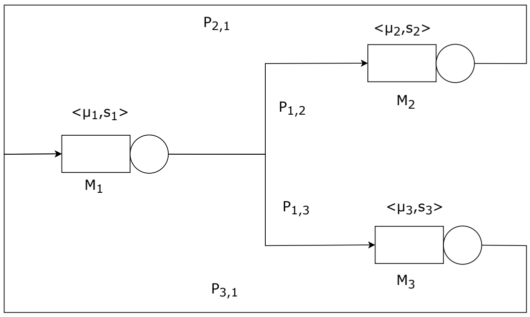


Figure 3: Load balancing example

- $\mathbf{x}(0) = (\mathbf{x}_1(0), \ldots, \mathbf{x}_M(0))$: the *initial condition*, i.e., $\mathbf{x}_i(0)$ is the number of clients at station i at time 0.

We remark that, in a closed QN, $\mathbf{P}$ is a *stochastic* matrix (i.e., each row sums up to 1) because clients that got served always ask for another service (potentially the same if the client got served by station i and $\mathbf{P}_{i,i} > 0$).

QNs have also a graphical representation. Each station is represented with a rectangle followed by a circle, annotated with its concurrency level and its service rate. The routing probability matrix is represented with directed arcs, where the arc connecting station i with station j bears the label $\mathbf{P}_{i,j}$.

Figure 3 represents the running example we use through this section. It represents a load-balancing system with 3 stations. Requests from station ***1*** are forwarded to stations **2** and ***3*** with probabilities $\mathbf{P}_{1,2}$ and $\mathbf{P}_{1,3}$, respectively; then clients return to station ***1***. In the following, stations **2** and ***3*** are the computation units, while ***1*** is the dispatching unit.

2.1.1 Markov chain semantics

Given a QN, we can define a continuous-time Markov chain (CTMC) that represents the evolution of the queue length at each station over time. The CTMC state is $\vec{X} = (X_1, \ldots, X_M)$, where X_i is the queue length of station i. For each station i at most s_i clients can be served concurrently:

if $X_i \leq s_i$, all the X_i clients proceed with rate μ_i; otherwise, s_i clients are served with rate μ_i while the remainder $(X_i - s_i)$ wait for a free server. After the service of station i, clients go to station j with probability $\mathbf{P}_{i,j}$.

We model this evolution through a Markov population process (MPP). This representation has two advantages over the traditional CTMC one: i) it is more compact (i.e., we do not need to explicitly represent the evolution for each possible state); ii) it has a direct mapping on the learning algorithm presented in Chapter 3. MPP's transitions are represented by jump vectors and associated transition functions defined over $\vec{X}$ [Bor+13]. The jump vector $h^{(ij)}$ represents the movement of one client from station i to station j. More formally, the jump vector $h^{(ij)}$ is the evolution from the state $\vec{X}$ to state $\vec{X} + h^{(ij)}$, where

$$\vec{X} + h^{(ij)} = (X_1, \ldots, X_i - 1, \ldots, X_j + 1, \ldots, X_M).$$

Function q represents the transition rate between two states. We write $h^{(ij)}$'s rate as $q(\vec{X}, \vec{X} + h^{(ij)})$.

Given this, the CTMC is defined by

$$q(\vec{X}, \vec{X} + h^{(ij)}) = \mathbf{P}_{i,j} \mu_i \min(X_i, s_i), \quad i, j = 1, \ldots, M. \tag{2.1}$$

In the running example of Figure 3, the jump vectors are

$$\begin{aligned} h^{(12)} &= (-1, +1, 0) & \qquad h^{(13)} &= (-1, 0, +1) \\ h^{(21)} &= (+1, -1, 0) & \qquad h^{(31)} &= (+1, 0, -1) \end{aligned}$$

where the first row represents a client moving from the dispatching unit towards one computation unit, and vice-versa for the second row. The transition rates are:

$$\begin{aligned} q(\vec{X}, \vec{X} + h^{(12)}) &= \mathbf{P}_{1,2} \mu_1 \min(X_1, s_1) \\ q(\vec{X}, \vec{X} + h^{(13)}) &= \mathbf{P}_{1,3} \mu_1 \min(X_1, s_1) \\ q(\vec{X}, \vec{X} + h^{(21)}) &= \mathbf{P}_{2,1} \mu_2 \min(X_2, s_2) \\ q(\vec{X}, \vec{X} + h^{(31)}) &= \mathbf{P}_{3,1} \mu_3 \min(X_3, s_3) \end{aligned}$$

Given the initial condition $\mathbf{x}(0)$ and the transition function of Equation (2.1), the CTMC is completely defined and simulable [Gil07], albeit

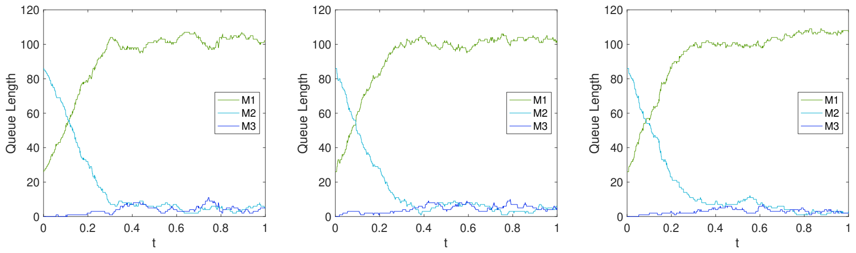


Figure 4: Three simulations of the running example's CTMC.

untractable. Namely, the number of equations grows combinatorially with the number of clients and stations, as it considers each possible division of N clients among M stations. The following section introduces an approximation to avoid this problem.

In Figure 4, we show three independent simulations in the time interval $[0, 1]$ of the running example CTMC, where we set $s_1 = 1000$, $s_2 = 30$, $s_3 = 25$, $\mu_1 = 1$, $\mu_2 = \mu_3 = 11$. The simulations start with $\mathbf{x}(0) = (26, 86, 0)$. On the x axis, we have the simulation time, while the y axis has the queue lengths of the three stations M_1 (green line), M_2 (cyan line), and M_3 (blue line).

2.1.2 Fluid Approximation

QNs admit a tractable approximation as a system of ordinary differential equations (ODEs) called *fluid approximation*. The fluid approximation represents the system using M ordinary differential equations, regardless of the number of clients. Among the applications of fluid approximation, we list network protocols [CBL15] and load balancing strategies [GB10; Xie+15].

We consider the average effect of clients flowing into and out of the station for each station. The evolution of each station is described in an equation. Given a station k, its equation contains the jump vectors $h^{(ik)}$ and $h^{(kj)}$ that represent the arrival of one client from station i and the exit of one client towards station j, respectively, weighted by the transition rates ($q(\vec{X}, \vec{X} + h^{(ik)})$ and $q(\vec{X}, \vec{X} + h^{(kj)})$, respectively). The ODE system is written as:

$$\frac{d\mathbf{x}_k(t)}{dt} = \sum_{h^{(ij)}} h_k^{(ij)} q(\vec{x}(t), \vec{x}(t) + h^{(ij)}),\ k = 1, \ldots, M. \qquad (2.2)$$

where $\vec{x} = (\vec{x}_1, \ldots, \vec{x}_M)$ are the variables of the fluid approximation, and $h_k^{(ij)}$ is the k-th coordinate of the jump vector. We recall that, given the jump vector $h^{(ij)}$:

$$h_k^{(ij)} = \begin{cases} -1 & \text{if } i = k \\ +1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

The solution for each coordinate $\mathbf{x}_k(t)$ is the approximation of the average queue length of station k at time t according to the CTMC semantics [Bor+13]. This approximation is asymptotically exact [Kur70], i.e., with enough servers and clients circulating the system, the ODE solution and the expected queue length of the stochastic process become indistinguishable.

The fluid approximation for the load balancer running example (Figure 3) is:

$$\begin{aligned}
\frac{d\mathbf{x}_1(t)}{dt} &= -\boldsymbol{\mu}_1 \min(\mathbf{x}_1(t), \mathbf{s}_1) + \boldsymbol{\mu}_2 \min(\mathbf{x}_2(t), \mathbf{s}_2) + \\
&\quad + \boldsymbol{\mu}_3 \min(\mathbf{x}_3(t), \mathbf{s}_3) \\
\frac{d\mathbf{x}_2(t)}{dt} &= -\boldsymbol{\mu}_2 \min(\mathbf{x}_2(t), \mathbf{s}_2) + \mathbf{P}_{1,2}\boldsymbol{\mu}_1 \min(\mathbf{x}_1(t), \mathbf{s}_1) \\
\frac{d\mathbf{x}_3(t)}{dt} &= -\boldsymbol{\mu}_3 \min(\mathbf{x}_3(t), \mathbf{s}_3) + \mathbf{P}_{1,3}\boldsymbol{\mu}_1 \min(\mathbf{x}_1(t), \mathbf{s}_1)
\end{aligned}$$

In Figure 5, we show the fluid approximation integration against one CTMC simulation, the average of 10 and 100 simulations, respectively. We kept the parameters of the previous section, i.e., $s_1 = 1000$, $s_2 = 30$, $s_3 = 25$, $\mu_1 = 1$, $\mu_2 = \mu_3 = 11$. The solid lines represent the CTMC-derived data, while the dashed lines depict the fluid approximation. The more simulations are considered, the more the fluid approximation matches the observations.

From Equation (2.2) we can derive other performance metrics, e.g., throughput, utilization, and response time. In [TGH12; Tri+12] there is a study of these results in a process algebra [Hil96].

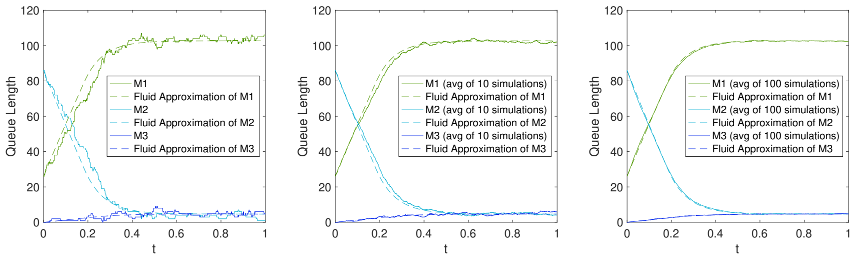


Figure 5: Fluid approximation (dashed line) against one CTMC simulation, the average of 10 and 100 simulations (solid lines).

This thesis focuses on QNs without *self-loops* (i.e., clients that request another service from the same station immediately after leaving it). In the fluid approximation, we can choose the self-loop probabilities freely if we tune the remaining elements of the probability matrix and the service rates. More formally, we rewrite Equation (2.2) to single out the rates due to self loops $\mathbf{P}_{k,k}$:

$$\frac{d\mathbf{x}_k\left(t\right)}{dt} = \sum_{i\neq k} \mathbf{P}_{i,k}\boldsymbol{\mu}_i \min(\mathbf{x}_i\left(t\right), \mathbf{s}_i) + (\mathbf{P}_{k,k} - 1)\boldsymbol{\mu}_k \min(\mathbf{x}_k\left(t\right), \mathbf{s}_k) \quad (2.3)$$

and prove the following theorem:

Theorem 2.1.1. *For each* $\pi \in [0,1)^M$*, stochastic matrix* $\mathbf{P}$ *and* $\boldsymbol{\mu} \geq 0$ *where* (2.3) *holds, there exist* $\hat{\mathbf{P}}$ *and* $\hat{\boldsymbol{\mu}}$ *such as for each k:*

1. $\frac{d\mathbf{x}_k(t)}{dt} = \sum_{i\neq k} \hat{\mathbf{P}}_{i,k}\hat{\boldsymbol{\mu}}_i \min(\mathbf{x}_i\left(t\right), \mathbf{s}_i) + (\hat{\mathbf{P}}_{k,k} - 1)\hat{\boldsymbol{\mu}}_k \min(\mathbf{x}_k\left(t\right), \mathbf{s}_k)$;

2. $\hat{\mathbf{P}}_{k,k} = \pi_k$;

3. $\sum_i \hat{\mathbf{P}}_{k,i} = 1$;

4. $\forall i\ \hat{\mathbf{P}}_{k,i} \geq 0$;

5. $\hat{\boldsymbol{\mu}}_k \geq 0$.

Proof. Available in Appendix A. □

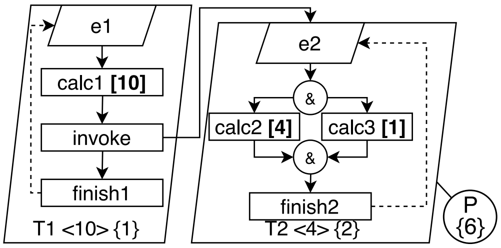


Figure 6: Simple LQN example.

This means that fluid approximation does not distinguish a network with self-loops from another without self-loops having the remaining parameters tuned accordingly.

We now rewrite (2.3) to remove the self-loops:

$$\frac{d\mathbf{x}_k(t)}{dt} = \sum_{i \neq k} \mathbf{P}_{i,k} \boldsymbol{\mu}_i \min(\mathbf{x}_i(t), \mathbf{s}_i) - \boldsymbol{\mu}_k \min(\mathbf{x}_k(t), \mathbf{s}_k) \tag{2.4}$$

2.2 Layered Queueing Networks

The approaches presented in Chapters 4 and 5 use an extension of the traditional QN model called Layered Queueing Network (LQN) [FAW+09]. This section will briefly describe this extension together with its graphical representation.

Figure 6 shows a simple LQN model. LQN models are composed of *tasks*, depicted as large parallelograms (named T1 and T2 in Figure 6), each one representing a set of correlated behaviors. Each task is available in multiple identical and independent copies according to its *replication factor* (10 and 4, respectively, for T1 and T2). *Task multiplicity* represents the threadpool size, which says how many software threads can perform work in each replica concurrently (1 for T1 and 2 for T2).

Processors represent hardware resources, i.e., the hardware resources that physically execute each thread. Processors are represented with circles, bearing the number of available hardware threads (6 for P). Processors are linked to the tasks they are executing with solid lines (processor P linked to task T2). In the LQN model, if a task is not linked to a processor, it is assumed to have a dedicated processor with enough cores to execute all its threads concurrently without contention on the hardware resources (T1).

An *entry* is a type of service that a task can perform. The LQN graphical notation represents an entry with a small box at the top of the task's parallelogram (in Figure 6, they are e1 and e2). The behavior of each entry is described by a direct acyclic graph (DAG) of *activities* and control *nodes*.

Activities are used to describe (i) computational work (i.e., CPU usage, like calc1, calc2, and calc3 in Figure 6) with its average duration written between square brackets; (ii) synchronous calls where the execution moves to the called entries and the caller retains the thread (e.g., in Figure 6 the activity invoke calls the e2 entry); (iii) asynchronous calls where the caller releases the thread before making the invocation. Solid arrows connect activities to the called entries, while dashed arrows are used for asynchronous calls. There are special activities without CPU demand used to represent the termination of an entry's logic (finish1 and finish2 in Figure 6).

Control nodes are points where the execution flow branches concurrently (AND-nodes, symbol '&', see e2's logic in Figure 6) and probabilistically (OR-nodes, symbol '+'). Finally, join nodes (symbol '&') identify locations where concurrent branches synchronize.

We will use the following functions, some taken from [Tri13].

- $rate(a)$ for each activity a returns the inverse of the average time needed to complete action a on a's processor.

- $prob(d, i)$ for each OR-node d is the probability that the successor of d is Γ_d^i's root.

- $proc(a)$ returns a's processor. It corresponds to $act^{-1}(a)$ in [Tri13].

2.3 Neural Networks

This section briefly explores the machine learning tool used in Chapters 3 and 4: the *Neural Network* (NN). NN is a machine learning local optimization tool that takes inspiration from how the mammals' nervous systems are organized [GKP19]. NNs take several inputs and produce one or more outputs by continuously combining them. NNs are organized in *layers*. A layer takes several inputs and computes each output by applying a non-linear differentiable function to a (weighted) linear combination of its inputs. Layers form a chain: the outputs of one layer are the inputs of the following one, with the first layer taking directly the NN inputs and the last layer being the final output of the NN. The more layers there are in an NN, the more expressive the outputs of the NN (i.e., they can represent complex functions).

The optimization procedure tweaks each layer's linear combination *weights* to find the values that minimize the *error* function (i.e., a derivable function that measures how far the NN outputs from the desired ones). The learning procedure is guided by a set of input and matching outputs, usually randomly partitioned into two parts: the *training set* and *validation set*. The splits for training–validation set is usually 80%–20% or 90%–10% of the provided data. The optimization procedure is iterative: in each iteration (called *epoch* in NN jargon), it applies the learning algorithm and computes the error function on the validation set. The optimization stops when the error function on the validation set is not decreasing over the epochs.

The learning algorithm dictates how the weights are updated during the epoch. The most common one is Gradient Descent, which uses the gradient of the error function applied to the training data w.r.t. the weights. In this thesis, we will use the Adam algorithm [KB15], which is derived from Gradient Descent, and is more suited towards large datasets and many weights to learn (like the problems we will explore in Chapters 3 and 4).

Learning algorithms' behaviors are usually tuned using *hyperparameters*, i.e., values set before the optimization procedure that condition how the search proceeds. One common hyperparameter (used by Adam) is the *learning rate*, which conditions how much the weights can be changed at every iteration. A small learning rate makes the search generally more precise but slower and more prone to finding local minima in the error function; a larger learning rate leads to a faster search, but it might skip large zones of the search space (that might contain the optimal weights). Hyperparameters tuning is a complex task requiring considerable work to attain effective NN optimization procedures.

2.4 Related work

In this section, we discuss techniques related to the following lines of research: performance prediction from programs, generation of performance models from programs, and estimation of parameters in QNs.

2.4.1 White-box Performance Prediction

White-box performance models are applied to systems where we know their inner details, except for some non-functional parameters. White-box techniques (the ones we consider in this thesis) keep a piecewise association between the model and the system (i.e., they have explanatory power) [Val+17].

PerfPlotter uses probabilistic symbolic execution [GDV12] to analyze the system's source code and extract a probability distribution on the desired performance metric (e.g, response time) [CLL16]. PerfPlotter is limited to single-threaded applications (therefore, it cannot capture thread contention) and does not produce a predictive model.

Some works extract performance models from execution logs, similarly to μP (see Chapter 5). In [IWF07], the authors define an algorithm that explores the execution log, observes how the components interact, and maps those interactions on the LQN formalism to produce a model. In [BHK11], the procedure targets component-based systems. Those approaches still require user intervention, and their models do not evolve with the system development. Instead, μP defines comprehensive logging that tracks the components' performance and the overall system infrastructure, starting from the source code only. Since the model is defined using only the execution log and the source code, we can say that the model evolves along with the code. The code-to-model link allows for ad-hoc logging, which improves prediction accuracy. For instance, in [BHK11], authors report that the performance models obtained via their approach have a mean percentage error of response time prediction greater than 50% in cases of high load (i.e., average utilization close to 50%). In Section 5.3 we show that μP can produce very accurate models from a similar system reporting prediction errors less than 10% under extreme operating conditions, i.e., with average utilization of the bottleneck microservice very close to 100%. Similarly, in [COQ21], a maximum relative prediction error of 21% is reported when the learned models are used to detect the saturation point of Teastore [KES+18]. Still, in Section 5.3, μP estimates performance quantities with less than 10% of error in all case studies, including Teastore.

Research in QN parameters estimation usually focuses on the service demands estimation problem when the system is in steady-state [Spi+15]. A system is in a steady state when its evolution does not depend on its initial condition: this usually happens after enough time passes from its startup. The precise quantification of this time depends on the system under study. The steady-state assumption is very powerful, leading to many analytical results for QNs [Bol+05]. Many results from the literature work under the steady-state assumption, using various techniques: linear regression [Pac+08], quadratic programming [INT18], non-linear optimization [Men08; AM17], clustering regression [CDS10], independent component analysis [Sha+08], pattern matching [CS14], Gibbs sampling [WC13; SJ11], and maximum likelihood [Wan+16].

There has been some research also in the LQN direction, using the Kalman Filters to estimate time-dependent parameters monitoring utilization and response times of the system [ZWL08]. It has been extended to predict ahead-of-time parameters evolution [ZLW11], or to attain QoS targets with a control-based approach that updates system parameters [Zhe+05].

The approach we propose for QN estimation (see Chapter 3) advances the state of the art as it estimates both service demands and QN routing probabilities (i.e., its topology). Moreover, we drop the steady-state assumption as the RNN uses the fluid approximation (see Section 3.1), and actually, the RNN works on traces that include the transient dynamics. Another work that uses the fluid approximation for service-demand estimation is [INT18].

Some works use data that is not readily available in a real use-case, e.g. [Liu+06; CDS10; Sha+08; Kal+11; CS14; ZWL08; ZLW11]. In a virtualized context, typical in Platform-as-a-Service environments, metrics like utilization might not be available as they depend on the specific hardware running the system, which contradicts the virtualization paradigm. The three approaches presented in this thesis use quantities that do not require explicit hardware support (i.e., the queue length obtained by querying the OS networking stack) or use execution logs.

2.4.2 Black-box Performance Prediction

Black-box approaches build performance models using only external information of the system but are inaccurate when predicting unseen contexts. Black-box approaches complement the role of the white-box techniques. Rather than keeping a piecewise relation between the system and the model, black-box models estimate the overall input-performance relation. Those limitations make black-box models computationally easier to manage than white-box ones. Still, black-box approaches are unsuitable when the model is used to predict the system in conditions very far from the ones observed in the learning phase (e.g., the testing conditions used in this thesis).

Even though black-box techniques are the focus of this thesis, we will give a brief overview of them for completeness reasons. Black-box models proved effective in specific contexts, e.g., variability-intensive systems (i.e, where the performance depends on the specific configuration) [Sie+12; Sie+15], online data-intensive systems [SW18], and in modeling data dependencies [Gra+20].

Many techniques are used to generate black-box models: machine learning [Guo+13; Sie+15; Val+17; Jam+18; RL19; GLD+21; Gro+21a], regression [JPG19; Gro+21b] and specifically linear regression [Sie+12], markov processes [Kha+16; Kha+20], and reduction to other known problems (e.g. the multiple-choice knapsack problem [BWB+19]).

2.4.3 Program-driven Generation of Performance Models

In the literature, program-driven generation of performance models has been less explored than the model-driven approach [CMI11]. Brosig et al. derive a component-based performance model from applications running on the Java EE platform [BKK09; BHK11]. Hrischuk et al. extracts LQN software performance models from distributed applications that communicate via remote procedure calls [HRW95]. Tarvo and Reiss analyze task-oriented applications to generate discrete-event simulation models [TR14]. In task-oriented applications, the workload is divided into tasks (indivisible jobs) assigned to a pool of worker threads. Tarvo and Reiss justified the simulation model approach to avoid explicitly modeling performance-related phenomena such as queuing effects, inter-thread synchronization, and hardware contention. The approaches proposed in this thesis measure and include those phenomena in the analytical model, thus forfeiting the need for a simulation model.

Chapter 3

Learning Queueing Networks by Recurrent Neural Networks

This chapter focuses on automatically extracting a white-box QN model from a distributed system.

As detailed in Section 2.1, a QN model describes a system as a set of users (*clients*) that probabilistically visit a set of resources (*stations*) that perform some specific work upon request. Each station is described by the time needed to perform their work (*service demand*) and the maximum number of requests that can be served at the same time (*concurrency level*). Clients are characterized by how they probabilistically access the resources (*routing matrix*).

Only part of the QN parameters is easily deducible from a distributed system deployment. For example, the concurrency level of each machine is the number of cores on the processor chip or the number of assigned CPU cores to the virtual machine for bare-metal or virtualized setups, respectively. Instead, tracking the service demands and how clients access the resources (the routing matrix) is harder.

In this chapter, we will construct a full QN model given the set of resources and their concurrency level. We will discover the resources' service demands and the clients' routing matrix, observing the evolution of resource queue lengths over time.

Many approaches exist to learn a performance model from data (Section 2.4 gives an overview). Among QN fitting approaches, many works estimated the service demands only ([Spi+15]), while we did not find approaches that estimate service demands and the topology (i.e., the routing matrix) together. The main obstacle is that this is a *nonlinear problem*. Indeed, the service demands and the routing matrix are multiplied together in the dynamical equations that describe the QN evolution (see Equation (2.1) in Section 2.1.1), and both need to be fitted. Moreover, we also face the scalability problem, as the exact equations that rule the QN grow combinatorially with the number of resources and clients in the system (see Section 2.1.2).

Our proposal solves both problems, and it is scalable. Section 3.1 describes a new Recurrent Neural Network (RNN) architecture that encodes the Fluid Approximation of the QN dynamics in an *interpretable fashion*: each RNN weight (concurrency levels, routing probabilities, and service rates) is associated to the QN parameter. Fluid Approximation solves the scalability problem by keeping only one equation per station while giving an accurate average queue length estimation (see Section 2.1.2 for more details). Still, the Fluid Approximation leads to a non-linear problem. For this reason, we use RNNs to estimate service demands and the routing matrix, as they proved effective in fitting non-linear systems [Mit97].

The connection between ODEs and RNNs is not new in the literature [Pea89; Che+18], and some approaches use Neural Networks to extract black-box performance models (see Section 2.4.2 for more details). Still, we did not find applications of customized RNN architectures that extract a white-box performance model. We think that this chapter fills an interesting niche as it leverages the best of three worlds:

- *neural networks* are a versatile tool in model fitting, with a lively research community;
- the *custom architecture* encodes the knowledge we have on the sys-

tem dynamics in the RNN, which in turn requires less training data to obtain a reliable estimation;

- *white-box performance models* allow for accurate what-if analysis: for each change in the system, we know which parameter must be altered to obtain a prediction under the new setting.

The key technical contribution of this approach is to define a direct association between the Fluid Approximation and the RNN activation functions and layers. To the best of our knowledge, this is the first approach that brings together the learning capability of machine learning and the expressiveness of analytical performance models, confirming that *"AI will be at the core of performance engineering"* [Lit19].

The remainder of this chapter is organized as follows. Section 3.1 describes how we encode the Fluid Approximation in an RNN cell, and how we define the error function used both in learning and in the experimental phase. Section 3.2 applies the approach on synthetic benchmarks on randomly generated QNs and a real web application developed using the load balancing architectural style. We test the model using what-if analysis, i.e., we compare the system dynamics under unseen configurations with one predicted by the model when applying the same modifications. We experienced prediction errors under 10% across a validation set of 2000 instances that changed the system workload, number of servers, and routing probabilities.

3.1 Learning methodology

This section proposes an RNN architecture that encodes the Fluid Approximation dynamics in an *interpretable fashion*, i.e., the quantities to estimate (routing probabilities, service rates) as RNN weights.

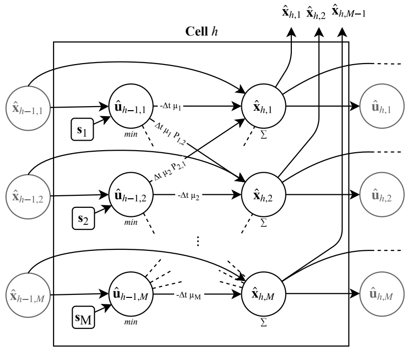


Figure 7: RNN encoding

3.1.1 ODE Discretization

Our encoding works on a time-discretized version of the Fluid Approximation. Firstly, we rewrite the Fluid Approximation in matrix notation:

$$\frac{d\mathbf{x}(t)}{dt} = -\boldsymbol{\mu} \min(\mathbf{x}(t), \mathbf{s}) + \mathbf{P}^T \boldsymbol{\mu} \min(\mathbf{x}(t), \mathbf{s})$$

where $\mathbf{x}(t)$ is the M-dimensional vector of each station queue lengths at time t. We now introduce a finite-step approximation for a small time duration Δt:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot \big(-\boldsymbol{\mu} \min(\mathbf{x}(t), \mathbf{s}) + \boldsymbol{\mu}\mathbf{P} \min(\mathbf{x}(t), \mathbf{s})\big)$$

that can be rewritten as

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot \mathbf{u}(t) \cdot (\boldsymbol{\mu} \odot (\mathbf{P} - \mathbf{I})) \tag{3.1}$$

where $\mathbf{u}(t) = \min(\mathbf{x}(t), \mathbf{s})$, $\mathbf{I}$ is the identity matrix of appropriate dimension, and $\odot$ is the operator where if $\mathbf{C} = \mathbf{a} \odot \mathbf{B}$, then $\mathbf{C}_{i,j} = \mathbf{a}_i \cdot \mathbf{B}_{i,j}$.

3.1.2 RNN Encoding

Equation (3.1) can be directly encoded in the RNN. The first layer of the RNN is an M-dimensional input layer $\hat{\mathrm{x}}_0$ that corresponds to the queue lengths' initial condition. After that, there are $H - 1$ cells, where the

h-th cell computes the evolution of the queue lengths at time $h\Delta t$, denoted by $\hat{\mathbf{x}}_h$ (see Fig. 7). Using the time-discretized version of the Fluid Approximation (Equation (3.1)), the h-th cell computes $\hat{\mathbf{x}}_h = \hat{\mathbf{x}}_{h-1} + \Delta t \cdot \hat{\mathbf{u}}_{h-1} \cdot (\boldsymbol{\mu} \odot (\mathbf{P} - \mathbf{I}))$, where $\hat{\mathbf{u}}_{h-1}$ estimates $\mathbf{u}\left((h-1)\Delta t\right)$ as $\hat{\mathbf{u}}_{h-1} = \min\left(\mathbf{s}, \hat{\mathbf{x}}_{h-1}\right)$.

The weights to learn are the matrix $\mathbf{P}$ (made of $M(M-1)$ weights, since the diagonal is empty as we avoid self-loops) and the vector $\boldsymbol{\mu}$ (made of M weights).

Since we are interested in a white-box model, we introduce some feasibility constraints that force the learning algorithm to explore valid values (i.e., values that do not violate physical constraints). Specifically, we require that $\mathbf{P}$ is a stochastic matrix (each element is non-negative and each row sums up to 1), that there are no self-loops ($\mathbf{P}_{i,i} = 0$ for each i), and the rates are non-negative ($\boldsymbol{\mu} \geq 0$). We impose the stochasticity of $\mathbf{P}$ by clamping the learned values in the range $[0, \infty)$ and dividing the values in each row by their sum. By restricting to valid values, we keep the link between learned values and their physical counterparts in reality (*explainable machine learning* [SWM17]).

Since we have a white-box model, we can forecast how the system would behave if we change part of it (*what-if analysis*) as we can replicate the same alterations on the learned values. A traditional approach without constraints generates a black-box model where the learned values might not be valid (e.g., a negative probability). Non-valid values break the link, so it's unclear how to replicate the system changes on the learned values. In the latter case, we must learn the model from scratch at each system change.

As a running example, we will use the load balancer system presented in Section 2.1, Figure 3. The RNN encoding for the h-th cell (i.e., the queue length transient evolution at time $h\Delta t$) of our running example is:

$$
\begin{aligned}
\hat{\mathbf{u}}_{h-1,1} &= \max(\mathbf{s}_1, \hat{\mathbf{x}}_{h-1,1}) \\
\hat{\mathbf{u}}_{h-1,2} &= \max(\mathbf{s}_2, \hat{\mathbf{x}}_{h-1,2}) \\
\hat{\mathbf{u}}_{h-1,3} &= \max(\mathbf{s}_3, \hat{\mathbf{x}}_{h-1,3}) \\
\hat{\mathbf{x}}_{h,1} &= \hat{\mathbf{x}}_{h-1,1} + \Delta t\,(-\boldsymbol{\mu}_1\hat{\mathbf{u}}_{h-1,1} + \boldsymbol{\mu}_2\mathbf{P}_{2,1}\hat{\mathbf{u}}_{h-1,2} + \boldsymbol{\mu}_3\mathbf{P}_{3,1}\hat{\mathbf{u}}_{h-1,3}) \\
\hat{\mathbf{x}}_{h,2} &= \hat{\mathbf{x}}_{h-1,2} + \Delta t\,(\boldsymbol{\mu}_1\mathbf{P}_{1,2}\hat{\mathbf{u}}_{h-1,1} - \boldsymbol{\mu}_2\hat{\mathbf{u}}_{h-1,2} + \boldsymbol{\mu}_3\mathbf{P}_{3,2}\hat{\mathbf{u}}_{h-1,3}) \\
\hat{\mathbf{x}}_{h,3} &= \hat{\mathbf{x}}_{h-1,3} + \Delta t\,(\boldsymbol{\mu}_1\mathbf{P}_{1,3}\hat{\mathbf{u}}_{h-1,1} + \boldsymbol{\mu}_2\mathbf{P}_{2,3}\hat{\mathbf{u}}_{h-1,2} - \boldsymbol{\mu}_3\hat{\mathbf{u}}_{h-1,3})
\end{aligned}
$$

3.1.3 Input data

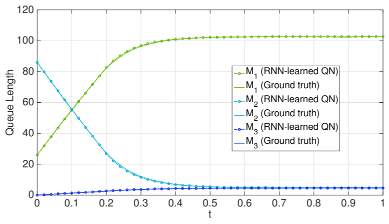
We train the RNN over a set of traces. Each trace is made of H vectors: $\tilde{\mathbf{x}}_0, \tilde{\mathbf{x}}_1, ..., \tilde{\mathbf{x}}_{H-1} \in \mathbb{R}^M_{\geq 0}$. The i-th component $\tilde{\mathbf{x}}_{h,i}$ of the vector $\tilde{\mathbf{x}}_h$ represents a sample of the queue length of station i at time $h \cdot \Delta t$. Since the Fluid Approximation estimates the average queue lengths, it should match the average of several independent executions that start with the same initial condition. At the same time, we need traces that start with different initial conditions to explore the distinct behaviors of the system.

3.1.4 Learning function

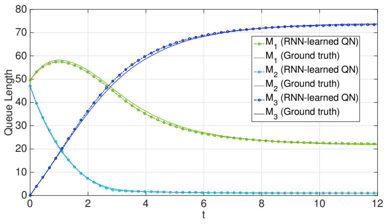
The learning error function err measures the maximum number of ill-positioned clients in the prediction across each trace's time interval. More precisely, err minimizes the maximum relative error between the queue lengths estimated by the RNN according to the discretized Fluid Approximation, $\hat{\mathbf{x}}_h$, and the measurements $\tilde{\mathbf{x}}_h$. We define err as:

$$err = \frac{\max_{h=1}^{H-1} \|\tilde{\mathbf{x}}_h - \hat{\mathbf{x}}_h\|}{2N} \cdot 100 \tag{3.2}$$

where $\|\cdot\|$ is the L1 norm. Since we are considering closed QNs with a fixed number of N circulating clients (see Section 2.1), the quantity $\|\tilde{\mathbf{x}}_h - \hat{\mathbf{x}}_h\|/(2N)$ measures the proportion of clients that are "misplaced" (i.e., predicted to be in a different station) at each time step. Since a misplaced client is counted twice (once when missing in a queue and once when is extra in another queue), we divide the norm by 2.



(a) Training trace ($err = 0.69\%$) with initial population $x(0) = (26, 86, 0)$ and concurrency levels (s_1=1000, s_2=30, s_3=25).



(b) What-if trace ($err = 1.49\%$) having unseen concurrency levels (s_1=1000, s_2=6, s_3=1) and initial population $x(0) = (49, 47, 0)$.

Figure 8: Simulations of the queue lengths using the learned parameters and the ground-truth ones for the running example.

We apply the methodology to our running example (the load balancer of Figure 3). Suppose that the original system has $\mathrm{s} = (1000, 30, 25)$ and $\boldsymbol{\mu} = (1, 11, 11)$, and $\mathbf{P}$ is set at

$$\mathbf{P} = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Using the experimental setup (discussed in the next section), we generated a training dataset of 50 traces, each with a different randomly generated initial population vector. Each trace was the average of 500 independent simulations considering the transient evolutions of the queue lengths. Figure 8 compares the queue lengths predicted with the parameters learned using the RNN vs the ground-truth ones. We observe high accuracy on the training set: for example, Figure 8a shows a trace taken from it. We now use the learned parameters to predict (what-if analysis) a load balancer with the same structure and service rates but different concurrency levels ($\mathrm{s} = (1000, 6, 1)$). We generate the what-if traces with $\mathrm{s} = (1000, 6, 1)$ and predict using the learned QN with the concurrency levels set at $\mathrm{s} = (1000, 6, 1)$. Figure 8b shows one trace from the what-if set and its predicted dynamics. The marked lines represent the simulations using the learned parameters, while solid lines use the ground-truth parameters. The prediction is accurate despite bottleneck shifts and longer transient dynamics compared to the ones used for learning (e.g. the one in Figure 8b).

3.2 Numerical Evaluation

This section evaluates our approach's accuracy using synthetic benchmarks and a real case study. The RNN architecture is implemented using the Keras framework [Cho+15] with the TensorFlow backend [MAA15]. We learned the models using a machine with the Intel(R) Xeon(R) CPU E7-4830 v4 at 2.00GHz with 500 GB of RAM, running 4.15.0-55-generic Linux kernel.

3.2.1 Synthetic case studies

Training phase In the synthetic case studies, we considered ten randomly generated QNs, five with $M = 5$ and five with $M = 10$ stations. For each QN, we generate at random a $M \times M$ stochastic routing matrix without self loops, M service rates in the interval $[4.0, 30.0]$, and M integer concurrency levels in the interval $\{15, 16, \dots, 30\}$. For each of those QNs, we generate 100 traces with a distinct initial population vector, where each station has a random number of clients in $\{0, \dots, 40\}$. Therefore, the total number of clients varies in the trace set between 1 and $40 \cdot M$ (we excluded the population vector with no clients). Each trace is then the average of 500 independent stochastic simulations (generated using Gillespie's algorithm [Gil07]) that start with the same population vector.

The traces set is split into 50 for the training and 50 for validation (chosen at random). During the learning phase, we use the Adam algorithm [KB15] with a learning rate equal to 0.05 until the error computed on the validation set does not improve by at least 0.01% in the last 50 iterations. The average learning time is 74 minutes for the 5-station QNs and 86 minutes for the 10-station QNs.

Discretization methodology We want to discuss two important hyper-parameters: the time horizon T of the stochastic simulations and the discretization interval Δt. Those two parameters contribute to the depth (number of cells) of the RNN as $T = (H - 1)\Delta t$. T and Δt must be chosen with care, as they impact the duration of the learning phase and the quality of the model:

- if T is too short, the trace might not expose the full dynamics of the network;
- if T is too long, a significant part of the trace is in a steady state, where there is no information about the network dynamics, and lengthens the learning time without improving the model;
- if Δt is too short, the RNN is unnecessarily deep, and the learning time increases;
- if Δt is too long, the discretization might skip important dynamics affecting the model quality.

We set $T = 10$ and $\Delta t = 0.01$ in our synthetic experiments, hence $H = 1000$.

Testing phase: what-if analysis To test the predictive power of the learned parameters, we perform the so-called "what-if" analysis. What-if analysis tests the learned models against new conditions unseen at learning time. What-if analysis mimics a real use of the learned model and evaluates if it captured the spirit of the system rather than just echoing the observed traces. Firstly, we test the predictive power under different client populations and the concurrency levels seen during learning. Then, we test unseen concurrency levels with the client populations used in the learning phase.

What-if analysis over client population In this paragraph, we experiment, for each QN, with 100 new (i.e, not used in the learning phase) initial population vectors and compare the traces generated using the

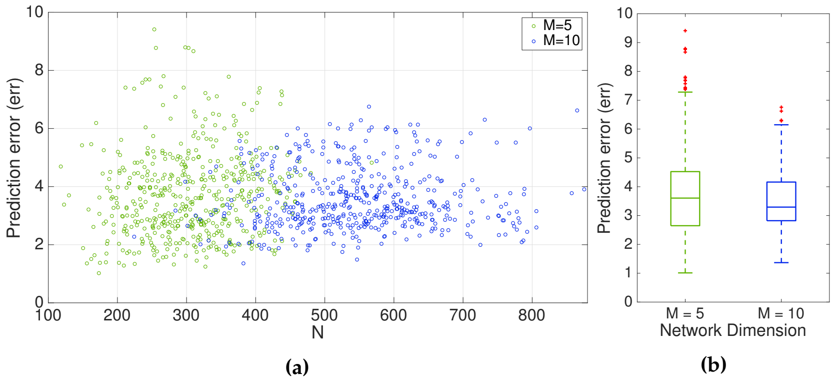
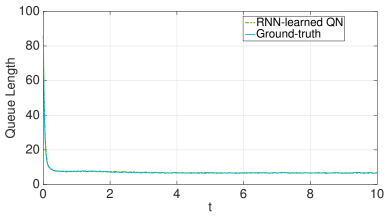


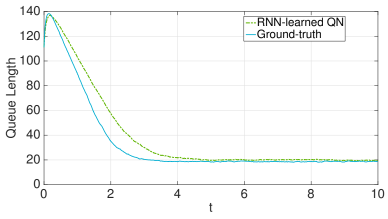
Figure 9: What-if instances for synthetic case studies. a) Prediction error over the number of clients in each instance. b) Prediction error statistics for each network size.

learned parameters (routing matrix and service rates) against the ground-truth ones. As with the learning phase, a trace is an average of 500 independent stochastic simulations that start with the prescribed initial condition.

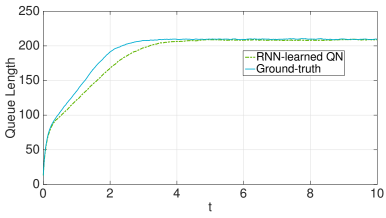
In the scatterplot of Figure 9a, for each instance, we plot the prediction error (err, see Equation (3.2)) over the number of clients circulating the system. Green markers correspond to $M = 5$ cases, while blue ones are the $M = 10$ cases. In all cases, we observe errors under 10%. The box plots in Figure 9b show the prediction error distribution for each network size. Box plots are drawn as follows: the line inside the box is the median error, the box plots' upper and lower sides are the 25th and 75th percentiles, and the upper and lower limit of the dashed line represent the extreme points not to be considered outliers, and in red we draw the outliers (12 with $M = 5$, 4 with $M = 10$). Those box plots show no statistically significant difference in the prediction quality between the two QN sizes. Figure 10 plots the predicted and ground-truth traces for the instance with the most significant prediction error ($err = 9.41\%$). This case has 5 stations and starts with the (unseen) population vector (86,111,13,15,28). The solid line represents the ground-truth dynamics,



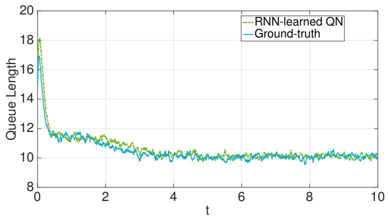
(a) Station 1



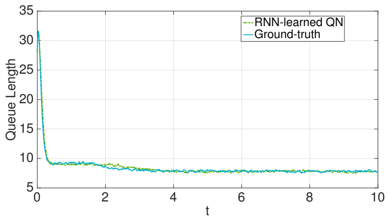
(b) Station 2



(c) Station 3



(d) Station 4



(e) Station 5

Figure 10: Ground-truth queue lengths plotted against the learned QN on the case with the highest error among the synthetic what-if over client population.

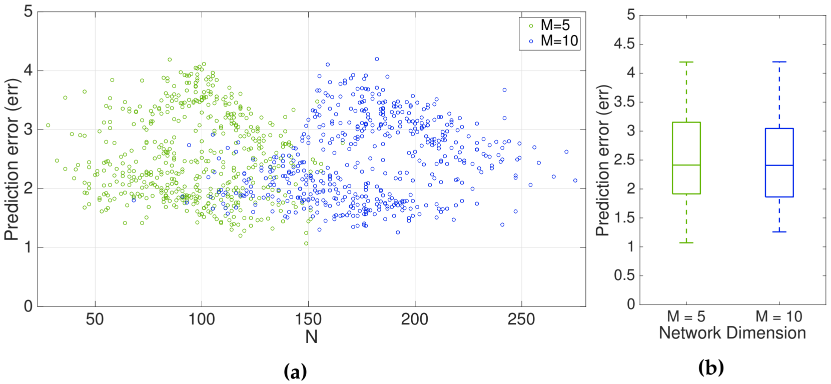
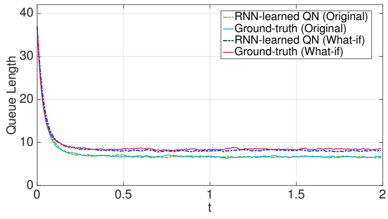


Figure 11: a) Prediction error of the what-if instances over the concurrency level of the most utilized station. b) Prediction error statistics for $M = 5$ and $M = 10$.

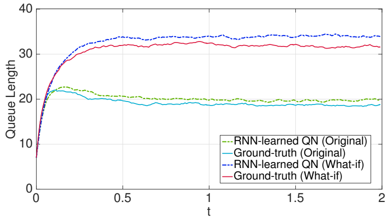
while the dashed line represents the evolution predicted by the RNN-learned QN. Still, this case shows a good generalizing power for all the station dynamics.

What-if over concurrency levels We validate the predictive power of the learned QN under varying concurrency levels. For each of the ten QNs, we found the station with the highest ratio between the steady-state queue length and its concurrency level (*bottleneck*) and added multiples of 20 servers to this station until it was not the bottleneck anymore. Then we compared the traces using the learned routing matrix and service rates against the ground-truth routing matrix and service rates, using the new concurrency levels and the population vectors used at learning time. We kept the error function used in learning (see Equation (3.2)).

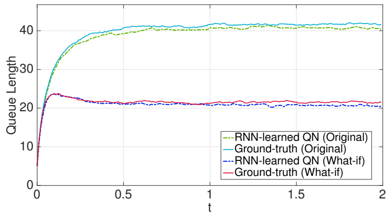
Figure 11a shows the prediction error of this experiment, exposing prediction errors under 5% across all instances. Even in this setting, Figure 11b does not report a significant difference in the prediction error for the two network sizes. Figure 12 compares the queue lengths of the what-if instance (after bottleneck switch) that reported the maximum prediction error (i.e., 4.5%) against the original ones (before bottleneck



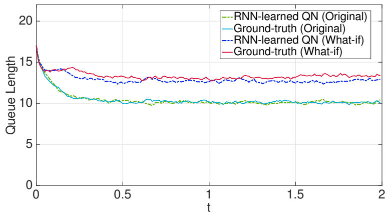
(a) Station 1



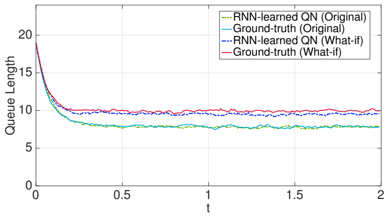
(b) Station 2



(c) Station 3



(d) Station 4



(e) Station 5

Figure 12: Ground-truth and predicted queue lengths on the test case with the maximum prediction error in the what-if over concurrency levels.

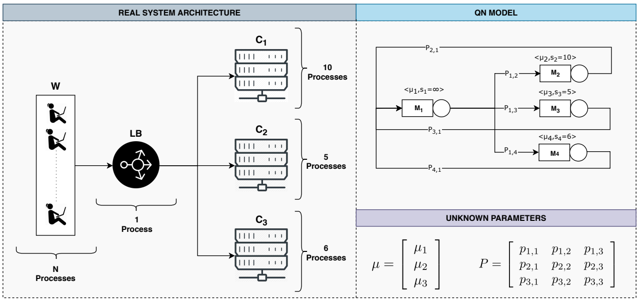


Figure 13: Case study architecture.

switch) that had a prediction error of 3.1%. The cyan line are the original conditions' ground-truth dynamics, while the green lines plot the RNN prediction. The red line shows the ground-truth traces after the concurrency increase in the bottleneck station from 17 to 37 servers, while the blue line depicts the RNN prediction under such change. Although the bottleneck switch from station 3 to station 2 also significantly impacted the other stations' dynamics, the prediction is good and fully captured the spirit of the what-if.

This result shows that the models we obtained are versatile enough to provide insights in situations where the original QN was not exercised. Therefore, the learning procedure produces an effective white-box model, i.e., we do not need to repeat the learning procedure if the system changes, but we apply the modifications directly to the model.

3.2.2 Real case study

Setup This section presents an in-house developed web application emulating a load-balancing system whose load is configurable. Figure 13 depicts each node's system architecture and concurrency level. It contains five components.

- **W** is the reference station. It is a multi-threaded Python program where each of the N clients is assigned a process that iteratively waits for an exponentially distributed time (whose parameter is supplied at deploy time) and then issues a request to the load balancer.

- **LB** is the load balancer. It is implemented in Node.js [TV10] and routes the user requests with equal probability to one of the three replicas of the web-server. The implementation allows for different probabilities (e.g., 1/4 to **C1** and **C2**, 1/2 to **C3**) by setting the weights accordingly.

- **C1**, **C2**, and **C3**, are the three replicas of the web server that serve the request from the user. They are implemented using the multi-threaded NodeJs Clusters[1], and the service time is extracted from an exponential distribution. Each replica has its exponential distribution parameter and concurrency level to make the case study more interesting.

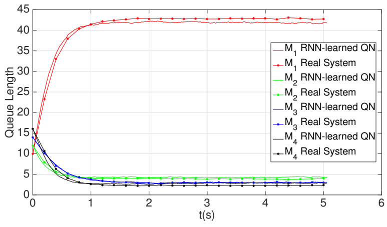
Even though the service time is parametric, we do not know their exact time distribution, as factors like communication details or NodeJs internals influence it.

We computed the queue lengths of the components (the input traces of the learning procedure, see Section 3.1.3) by parsing their access logs, an approach followed by [Wan+16]. Other works, like [ITT17], record the TCP backlog of the components. Our proposed setup allows sampling data every $\Delta t = 0.01$ s, capturing all the system dynamics without slowing the application cycle. The replication package for this evaluation is publicly available at `https://zenodo.org/record/3679251`.

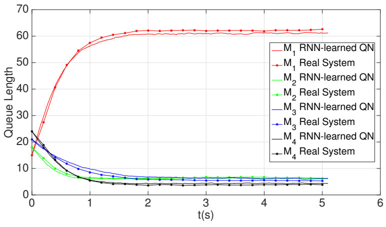
Model Learning The training dataset comprises 50 traces, each corresponding to a different population vector with between 0 and 30 clients in each station. As we did for the synthetic case studies (see Section 3.2.1), each trace is the average queue length dynamics observed in 500 independent executions starting with the same configuration.

The case study is modeled with the QN on the right side of Figure 13. The components **W** and **LB** (workload generator and load balancer) are

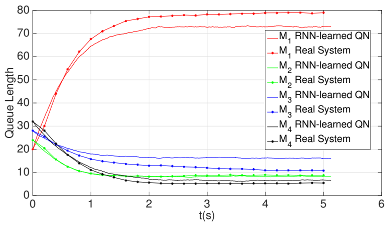
[1]`https://nodejs.org/api/cluster.html`



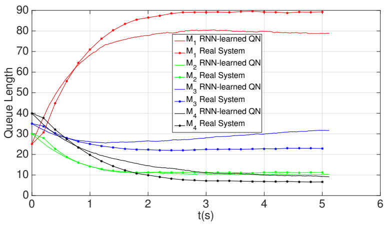
(a) $k = 2$, err=6.46%



(b) $k = 3$, err=5.03%



(c) $k = 4$, err=6.45%

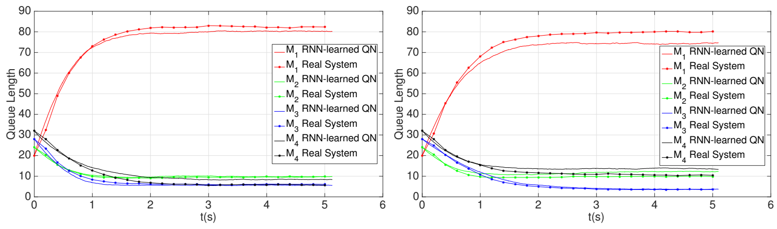


(d) $k = 5$, err=9.05%

Figure 14: Real system dynamics (i.e., marked lines) versus the RNN-learned QN (i.e., solid lines) in the what-if cases over increased population $N = 26k$.

condensed in the station M_1, as the delay introduced by the load balancer is negligible; the other stations, M_2, M_3, M_4, each model a component (respectively **C1**, **C2**, and **C3**). The parameters in the bottom right part of Figure 13 will be learnt by the RNN. As we did in Section 3.2.1, the learning uses half of the traces while the remainder acts as a validation set. We used the Adam [KB15] learning algorithm with the learning rate set to 0.01, repeated until the validation set error in the last 50 iterations does not improve by at least 0.01%. On average, Adam takes 27 minutes to learn, with a validation error of 3.89%.

What-if analysis We will now evaluate the prediction quality of the learned QN versus the real system when changing the number of clients, the **LB** forwarding policy (hence, the QN's routing probabilities), and the three web-server's concurrency levels (i.e, the QN's concurrency levels). Since we are confronted with a real system, in this section, we follow



(a) err=5.98%

(b) err=6.10%

Figure 15: Real and predicted dynamics after bottleneck solution. a) By increasing the concurrency level of M_3. b) By changing **LB**'s routing probability.

a realistic scenario where we discover the bottleneck and solve it using only the information provided by the model. The bottleneck is found by increasing the original number of clients (26, in this example) in the QN model by a factor $k = 2, \dots, 5$ until we find that station M_3 is saturated. We validate the findings by running the real system with the increased loads and plot the queue length dynamics in Figure 14. We observe that the most used component is **C2** (corresponding to the station M_3), and the prediction error is less than 10% across all instances.

We consider two alternative strategies for solving the bottleneck on **C2**/M_3:

(a) we increase the concurrency level of **C2**/M_3 from 5 to 8;

(b) we change the routing probabilities from **LB**/M_1 to the web-servers **C1**/M_2, **C2**/M_3, **C3**/M_4 to, 0.35, 0.20 and 0.45 respectively.

Figure 15 compares the predicted dynamics with the real system dynamics when applying each strategy, starting from the case $k = 4$. Figure 14c shows the original dynamics, Figure 15a applies the (a) strategy and Figure 15b applies the (b) strategy. As expected, both strategies relieved the pressure on **C2**/M_3, and the learned model predicts so with an error of about 6% on the real system.

Chapter 4

Service Demands Estimation in Layered Queueing Networks

Layered Queueing Network (LQN, see Section 2.2 for a discussion) is an expressive model for performance evaluation of software systems. However, the accuracy of the performance prediction depends on a precise estimation of LQN service demands [Bau+18]. Service demands estimation (also known as *model calibration*) is an open problem, and Section 2.4.1 gives an overview of the many attempts. The literature lacks an approach for layered systems with many interacting software and hardware layers. For example, an approach based on single-user workload [KN17] fails with fork/join synchronizations or asynchronous communication as they create multiple simultaneous requests in the system that generate contention.

This chapter proposes the first methodology, to the best of our knowledge, where using queue-length measurements only, we can calibrate an LQN, i.e., automatically derive LQN service demands. As in the QN approach (presented in Chapter 3) we use queue lengths as inputs since they can be obtained without altering the system being measured: we can use operating system primitives for TCP traffic inspection [Wan+18;

ITT17], or through log inspection. We retained Recurrent Neural Networks (RNNs) as the estimator, as it proved successful in Chapter 3 and nonlinear models in general [Mit97].

This task is hard from a mathematical point of view, as there is no closed-form analytic expression for LQNs, unlike QNs. Therefore, we cannot encode the estimation problem as an optimization problem, where the model's dynamics are constrained against observed traces [KLK20].

Similarly to Chapter 3, we use a compact system of approximate equations (the *fluid approximation* [Tri13]), instead of the exact but exponentially large system of equations. Like the QN fluid approximation, the LQN fluid approximation is a set of non-linear ordinary differential equations (ODEs) that grows with the number of tasks and processors describing each resource's mean queue length dynamics. Its complexity is independent of the number of clients in the system and can be encoded in an RNN. See Section 3.1.2 for a more exhaustive discussion about the connection between RNNs and ODEs.

This chapter concludes with an analysis of the effectiveness and accuracy of our approach, using a wealth of already validated LQN models taken from the literature, together with a real application. The prediction quality is assessed by how accurately the learned models match the steady-state response times when the system is altered (what-if analysis) and runs in configurations such as the system load not seen at learning time. The analysis reported prediction errors of less than 5% across a validation set of hundred instances.

4.1 Approach

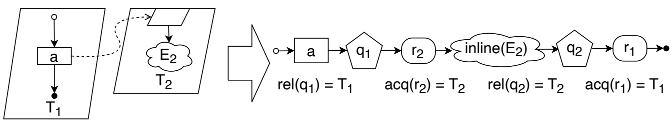
This section presents an algorithm to create the fluid approximation from an LQN. In [Tri13] a process algebra is used to do the LQN to fluid approximation conversion; instead, we take a different approach that keeps a more direct relation between the LQN components and the resulting differential equations leading to a more straightforward RNN encoding. This approach highlights easy-to-probe quantities, hence it is possible to produce the learning traces for the RNN. Also, as we will see in Section 4.3, it leads to accurate models of considered case studies.

The first phase of the LQN to fluid approximation transformation is called *inlining*, described in Section 4.1.1. Inlining replaces the entry calls with the activities that describe them. We can now define the inlined LQN stochastic process (see Section 4.1.2). From the stochastic process, we can build the fluid approximation (see Section 4.1.3), more straightforward to study than the original process, and simulate the LQN (Section 4.3). We then convert the fluid approximation into a Recurrent Neural Network (RNN; Section 4.2.1) to learn the model from the data.

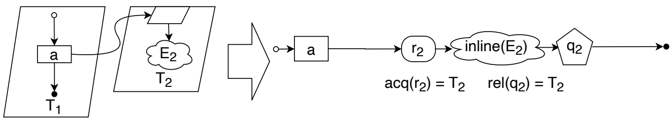
4.1.1 LQN inlining

LQN inlining transforms the original LQN model into an equivalent that explicitly models the execution paths. We need this procedure to define a new version of fluid approximation that distinguishes entry executions that originate from different calls, unlike the one presented in [Tri13]. Inlining starts with the main entry, i.e., the one that represents the main workload and calls the others, which usually represents the behavior of a system user. It recursively replaces every synchronous and asynchronous call with a copy of the called entry and some extra nodes to manage the caller and called entry threadpools. The effect is similar to the inlining expansion procedure presented in [Aho+07]. Inlining also adds nodes to the fork/join structures. The other activities, nodes, and functions presented in Section 2.2 (i.e., $rate$, $proc$, and $prob$) are preserved in the inlined variant. Since we are considering closed systems, the last activity in the inlined version is connected to the first one. In the remainder of this section, we detail the replacement procedure for the considered patterns: asynchronous communication, synchronous communication, and fork/join structure (see also Figure 16).

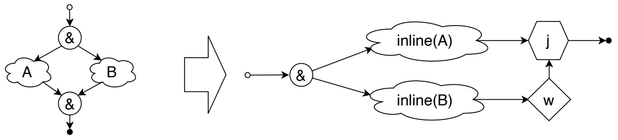
Asynchronous communication Asynchronous communication is performed when an entry calls the destination entry, releases the thread before invocation, and reacquires it after the call (see Section 2.2). The LQN graphical representation realizes this pattern with an activity linked to



(a) Asynchronous communication.



(b) Synchronous communication.



(c) Fork and join.

Figure 16: Unrolling patterns. The left part depicts the original LQN, while the right one the corresponding unrolled graph. Hollow dots mark the beginning of the pattern, filled dots mark the end.

the called entry through a dashed arrow. Inlining introduces two node types, *release* and *acquire* nodes, representing when a thread is released into or acquired from a threadpool, respectively. Inlining also introduces two functions, rel and acq that map the release and acquire nodes with the threadpool they operate on. Release nodes are represented with pentagons, while acquire nodes use the rounded rectangle symbol. Intuitively, the acquire node represents the requests directed to the entry that are queued on the task's port, while the release nodes represent the entry's responses traveling on the network.

Figure 16a represents the inlining for the LQN asynchronous communication pattern. On the left, we see an LQN fragment with an activity a in task T_1 that calls an entry inside task T_2 whose body is E_2 (the cloud). On the right, we see the inlined variant with the following elements: i) activity a, left unchanged; ii) the release node q_1 that releases a thread to T_1's threadpool (i.e., $rel(\mathsf{q}_1) = \mathsf{T}_1$); iii) the acquire node r_2 that acquires a thread from T_2's threadpool (i.e., $acq(\mathsf{r}_2) = \mathsf{T}_2$); iv) the inlined version of E_2 (i.e., $inline(\mathsf{E}_2)$); v) the release node q_2 that releases the thread to T_2's threadpool (i.e., $rel(\mathsf{q}_2) = \mathsf{T}_2$); vi) the acquire node r_1 that reacquires a thread from T_1's threadpool (i.e., $acq(\mathsf{r}_1) = \mathsf{T}_1$). The function definitions presented in Section 2.2 for the activities and nodes arising from the inlining of E_2 are left unchanged (e.g., for activities $x \in inline(\mathsf{E}_2)$, $proc(x)$ is the processor assigned to task T_2).

Synchronous communication We have synchronous communication when an entry calls the destination entry without releasing the thread during the invocation (see Section 2.2). The LQN graphical representation realizes this pattern with an activity linked to the called entry through a solid arrow. Figure 16b represents the inlining for the LQN synchronous communication pattern. On the left, we see the LQN fragment representing this communication pattern, on the right the inlined variant. Inlining is similar to asynchronous communication without the q_1 and r_1 nodes.

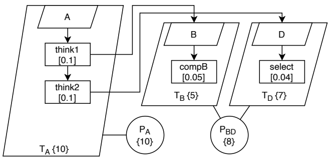


Figure 17: Running example: a distributed system.

Fork and join Fork and join pattern is used when the execution flow is split into two concurrent paths that eventually synchronize and merge (see Section 2.2). The LQN graphical representation represents this pattern through an AND-node with two arcs that lead to the two concurrent branches, with a further AND-node at the synchronization point. Figure 16c depicts the fork and join pattern inlining between the two branches A and B. The inlining defines two types of nodes, *join* and *wait* nodes, that represent the executions that completed one of the two branches and wait for the other branch completion to synchronize. We depict join and wait nodes with a hexagon and a diamond, respectively. The inlining replaces the A branch with its inlined variant $inline(\mathsf{A})$ and a join node j, while B is replaced with its inlined variant $inline(\mathsf{B})$ and the wait node w. The nodes j and w represent the terminated executions of A and B, respectively. If there is an execution both in j and w, the synchronization is completed, and then merge and continue with the activity after j. The pairing of the fork branches to A and B is arbitrary and leads to equivalent representations.

Running Example We apply inlining to the running example in Figure 17. Figure 17 represents a distributed system where the clients ($\mathsf{T_A}$ task) calls in sequence a computation service B (in $\mathsf{T_B}$ task) and a database query D (in $\mathsf{T_D}$ task). In the considered deployment, the two tasks $\mathsf{T_B}$ and $\mathsf{T_D}$ share a common processor $\mathsf{P_{BD}}$. We observe that this system is not representable through a QN model (as two tasks share the same processor): therefore the technique presented in Chapter 3 is not applicable in this context. The unrolling produces the sequence of nodes think1,

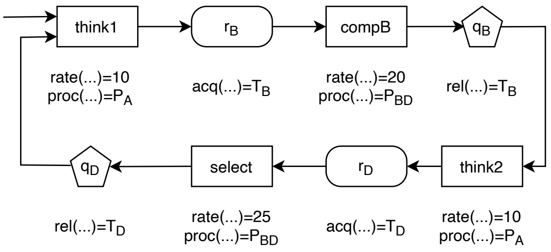


Figure 18: Example after applying inlining. We reported the value of the functions $rate$, acq, rel, $proc$ under the corresponding node.

r_B, compB, q_B, think2, r_D, select, q_D. The nodes represent, respectively: when clients think, clients wait for a thread to execute B, T_B runs the computation service, releases the thread, clients think once more, clients wait for a thread to execute D, T_D executes the database query, releases the thread. The jobs start at think1 and repeat after q_D. Figure 18 contains the details of the inlining.

4.1.2 Inlined LQN stochastic process

We now define the stochastic process underlying the inlined LQN. Similar to what we did in Section 2.1.1 for QNs, we define it through the inlined LQN state and the transitions as a set of jump vectors h with associated propensities $q(\vec{X}, \vec{X} + h)$.

The number of elements in the inlined LQN state vector $\vec{X}$ is the number of nodes and activities in the inlined LQN plus the number of tasks in the original LQN. For each inlined node or activity n, $\vec{X}_n$ is the number of jobs on n, while for each task T, $\vec{X}_T$ is the number of free threads in task T's threadpool. Therefore, the number of elements in $\vec{X}$ is the number of nodes and activities in the inlined LQN plus the number of tasks. Similarly to the definition for QNs, it is independent of the number of jobs circulating in the system.

The following will define the inlined LQN stochastic process as jump vectors (representing the action induced by each node or activity) and associated propensities. For each node, action, or task i, we indicate as $\vec{X}_{\mathsf{i}}$ the multiplicity of i in the inlined LQN state, and with h_{i} the jump vector element corresponding to i. Given two nodes or activities x and y, $\mathsf{x} \rightarrow \mathsf{y}$ indicates an arc between x and y in the inlined LQN. We conclude this section by showing the stochastic process induced by the running example inlined LQN.

Activities After completing the computation, every job in activity a evolves in the successor x (i.e., $\mathsf{a} \rightarrow \mathsf{x}$). The probability of completing each job is proportional to i) the number of jobs in a (i.e., $\vec{X}_{\mathsf{a}}$); ii) the length of a's computation, represented by $rate(\mathsf{a})$; iii) the contention factor on a's processor (i.e., s_{P} where $\mathsf{P} = proc(\mathsf{a})$). Generally, several activities share P, and we need to define a scheduling policy to share P's CPUs. If there are more CPUs in P than jobs in activities willing to use it, all jobs proceed unaffected (each job gets a dedicated CPU). Conversely, we apply the CFS sharing policy that proportionally slows the jobs (i.e., each job gets the same ratio of access to the CPU). s_{P} is defined as:

$$s_{\mathsf{P}} = \min\left(\frac{M_{\mathsf{P}}}{\sum_{\mathsf{b}|proc(\mathsf{b})=\mathsf{P}} \vec{X}_{\mathsf{b}}}, 1\right).$$

where M_{P} is the multiplicity of P (i.e., the number of CPUs). We observe that without contention $s_{\mathsf{P}} = 1$, conversely $s_{\mathsf{P}} < 1$.

Formally, the jump vector h induced by a and the corresponding propensity $q(\vec{X}, \vec{X} + h)$ are:

$$h_{\mathsf{i}} = \begin{cases} -1 & \text{if } \mathsf{i} = \mathsf{a} \\ +1 & \text{if } \mathsf{i} = \mathsf{x} \\ 0 & \text{otherwise} \end{cases} \qquad ; \qquad q(\vec{X}, \vec{X} + h) = \vec{X}_{\mathsf{a}} \cdot rate(\mathsf{a}) \cdot s_{\mathsf{P}} \quad (4.1)$$

where $\mathsf{a} \rightarrow \mathsf{x}$, $\mathsf{P} = proc(\mathsf{a})$, and s_{P} is the CFS sharing policy defined before.

Acquire nodes Jobs in acquire nodes r evolve in the subsequent node x (i.e, $\mathsf{r} \rightarrow \mathsf{x}$) as soon as a thread is available in the affected threadpool

$\mathsf{T} = acq(\mathsf{r})$. The thread acquisition is immediate, i.e., it happens with a rate (speed) much higher than any computation in the LQN. We model this rate with the constant H such as $H >> rate(\mathsf{a})$ for each activity a. This ensures that the scheduling activities happen before the computations as far as they are enabled (i.e., positive propensity) [Tri13]. This allows modeling transitions in two timescales, one representing very fast events like processor scheduling and the other for regular events like application service demands. Similarly to the activity case, the propensity is the product of i) the number of jobs in r (i.e., $\vec{X}_\mathsf{r}$); ii) the immediate transition rate H; iii) the contention factor on the threadpool T, g_T. g_T models the division of the free threads $\vec{X}_\mathsf{T}$ among all the requests, according to the GPS policy. g_T is defined as:

$$g_\mathsf{T} = \min\left(\frac{\vec{X}_\mathsf{T}}{\sum_{\mathsf{n}|acq(\mathsf{n})=\mathsf{T}} \vec{X}_\mathsf{n}}, 1\right).$$

where $\vec{X}_\mathsf{T}$ is the number of free threads in T. If there are enough free threads, each request is fulfilled ($g_\mathsf{T} = 1$); otherwise, they are divided proportionally to the number of requests ($g_\mathsf{T} < 1$).

We are now ready to formally define the jump vector h induced by r and its propensity $q(\vec{X}, \vec{X} + h)$:

$$h_\mathsf{i} = \begin{cases} -1 & \text{if } \mathsf{i} \in \{\mathsf{r}, \mathsf{T}\} \\ +1 & \text{if } \mathsf{i} = \mathsf{x} \\ 0 & \text{otherwise} \end{cases} \quad ; \quad q(\vec{X}, \vec{X} + h) = \vec{X}_\mathsf{r} \cdot H \cdot g_\mathsf{T} \quad (4.2)$$

Release nodes Release nodes q immediately release the thread $\mathsf{T} = rel(\mathsf{q})$ and evolve to the next node x (i.e., $\mathsf{q} \to \mathsf{x}$). Alike acquire nodes, this action is immediate. The propensity of this node is also proportional to the number of jobs in q (i.e., $\vec{X}_\mathsf{q}$). The jump vector h induced by q and its propensity $q(\vec{X}, \vec{X} + h)$ are:

$$h_\mathsf{i} = \begin{cases} -1 & \text{if } \mathsf{i} = \mathsf{q} \\ +1 & \text{if } \mathsf{i} \in \{\mathsf{x}, \mathsf{T}\} \\ 0 & \text{otherwise} \end{cases} \quad ; \quad q(\vec{X}, \vec{X} + h) = H \cdot \vec{X}_\mathsf{q} \quad (4.3)$$

OR-nodes OR-nodes o evolve immediately into one branch according to their probabilities. We encode this by creating a jump vector h for each branch, that moves to the branch's first node x ($\mathsf{o} \to \mathsf{x}$). The propensity is proportional to the number of jobs in o ($\vec{X}_{\mathsf{o}}$), the immediate rate H, and the probability of choosing that branch ($prob(\mathsf{o}, \mathsf{x})$). We remark that each job chooses a branch independently from the others. We define the jump vectors h and propensities $q(\vec{X}, \vec{X} + h)$ as:

$$h_{\mathsf{i}} = \begin{cases} -1 & \text{if } \mathsf{i} = \mathsf{o} \\ +1 & \text{if } \mathsf{i} = \mathsf{x} \\ 0 & \text{otherwise} \end{cases} \quad ; \quad q(\vec{X}, \vec{X} + h) = \vec{X}_{\mathsf{o}} \cdot H \cdot prob(\mathsf{o}, \mathsf{x}) \tag{4.4}$$

Fork and join structures: AND, join, wait nodes To clarify the discussion, we present the nodes involved in fork and join structures (i.e., AND, join, wait nodes) together. AND-nodes f, which represent where the forking happens, evolve immediately into both branches. For each branch, we define a jump vector h that leads to the first node of the branch x ($\mathsf{f} \to \mathsf{x}$) and the corresponding propensity $q(\vec{X}, \vec{X} + h)$ as follows:

$$h_{\mathsf{i}} = \begin{cases} -1 & \text{if } \mathsf{i} = \mathsf{f} \\ +1 & \text{if } \mathsf{i} = \mathsf{x} \\ 0 & \text{otherwise} \end{cases} \quad ; \quad q(\vec{X}, \vec{X} + h) = H \cdot \vec{X}_{\mathsf{f}} \tag{4.5}$$

Join nodes j wait for a job in corresponding wait node w (i.e., the one with $\mathsf{w} \to \mathsf{j}$) to ensure that both branches are completed; after that, the job goes in x, i.e., the successor of j ($\mathsf{j} \to \mathsf{x}$). The propensity of this transition is proportional to the number of jobs that completed both branches, i.e., $\min(\vec{X}_{\mathsf{j}}, \vec{X}_{\mathsf{w}})$, and the immediate action rate H. Therefore, we define the jump vector h and propensity $q(\vec{X}, \vec{X} + h)$ as follows:

$$h_{\mathsf{i}} = \begin{cases} -1 & \text{if } \mathsf{i} \in \{\mathsf{j}, \mathsf{w}\} \\ +1 & \text{if } \mathsf{i} = \mathsf{x} \\ 0 & \text{otherwise} \end{cases} \quad ; \quad q(\vec{X}, \vec{X} + h) = \min\left(\vec{X}_{\mathsf{j}}, \vec{X}_{\mathsf{w}}\right) \cdot H \tag{4.6}$$

We do not define a jump vector for w, as j's jump vector already models its effect.

Running example We continue the running example (Figure 17) and define the stochastic process induced by the inlined LQN of Figure 18. To simplify the discussion, we enumerate the jump vectors as $h^{(1)}$, $h^{(2)}$, ..., $h^{(8)}$, with the corresponding propensities $q(\vec{X}, \vec{X} + h^{(1)})$, $q(\vec{X}, \vec{X} + h^{(2)})$, ..., $q(\vec{X}, \vec{X} + h^{(8)})$.

We observe that the number of clients is at most 10, as the concurrency level of the processor. Therefore $\vec{X}_{\mathsf{think1}} + \vec{X}_{\mathsf{think2}} \leq 10$, hence:

$$s_{\mathsf{P_A}} = \min\left(\frac{10}{\vec{X}_{\mathsf{think1}} + \vec{X}_{\mathsf{think2}}}, 1\right) = 1$$

We also get:

$$s_{\mathsf{P_{BD}}} = \min\left(\frac{8}{\vec{X}_{\mathsf{compB}} + \vec{X}_{\mathsf{select}}}, 1\right)$$

$$g_{\mathsf{T_B}} = \min\left(\frac{\vec{X}_{\mathsf{T_B}}}{\vec{X}_{\mathsf{r_B}}}, 1\right) \qquad g_{\mathsf{T_D}} = \min\left(\frac{\vec{X}_{\mathsf{T_D}}}{\vec{X}_{\mathsf{r_D}}}, 1\right)$$

We get the following jump vectors:

$$\begin{array}{rccccccccccc}
 & & \mathsf{think1}, & \mathsf{r_B}, & \mathsf{compB}, & \mathsf{q_B}, & \mathsf{think2}, & \mathsf{r_D}, & \mathsf{select}, & \mathsf{q_D}, & \mathsf{T_B}, & \mathsf{T_D} \\
h^{(1)} = & (& -1, & +1, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0\,) \\
h^{(2)} = & (& 0, & -1, & +1, & 0, & 0, & 0, & 0, & 0, & -1, & 0\,) \\
h^{(3)} = & (& 0, & 0, & -1, & +1, & 0, & 0, & 0, & 0, & 0, & 0\,) \\
h^{(4)} = & (& 0, & 0, & 0, & -1, & +1, & 0, & 0, & 0, & +1, & 0\,) \\
h^{(5)} = & (& 0, & 0, & 0, & 0, & -1, & +1, & 0, & 0, & 0, & 0\,) \\
h^{(6)} = & (& 0, & 0, & 0, & 0, & 0, & -1, & +1, & 0, & 0, & -1) \\
h^{(7)} = & (& 0, & 0, & 0, & 0, & 0, & 0, & -1, & +1, & 0, & 0\,) \\
h^{(8)} = & (& +1, & 0, & 0, & 0, & 0, & 0, & 0, & -1, & 0, & +1)
\end{array} \tag{4.7}$$

with the following propensities:

$$\begin{aligned}
q(\vec{X}, \vec{X} + h^{(1)}) &= 10\vec{X}_{\mathsf{think1}} \\
q(\vec{X}, \vec{X} + h^{(2)}) &= H\vec{X}_{\mathsf{r_B}} \min\left(\frac{\vec{X}_{\mathsf{T_B}}}{\vec{X}_{\mathsf{r_B}}}, 1\right) \\
q(\vec{X}, \vec{X} + h^{(3)}) &= 20\vec{X}_{\mathsf{compB}} \min\left(\frac{8}{\vec{X}_{\mathsf{compB}} + \vec{X}_{\mathsf{select}}}, 1\right) \\
q(\vec{X}, \vec{X} + h^{(4)}) &= H\vec{X}_{\mathsf{q_B}} \\
q(\vec{X}, \vec{X} + h^{(5)}) &= 10\vec{X}_{\mathsf{think2}} \\
q(\vec{X}, \vec{X} + h^{(6)}) &= H\vec{X}_{\mathsf{r_D}} \min\left(\frac{\vec{X}_{\mathsf{T_D}}}{\vec{X}_{\mathsf{r_D}}}, 1\right) \\
q(\vec{X}, \vec{X} + h^{(7)}) &= 25\vec{X}_{\mathsf{select}} \min\left(\frac{8}{\vec{X}_{\mathsf{compB}} + \vec{X}_{\mathsf{select}}}, 1\right) \\
q(\vec{X}, \vec{X} + h^{(8)}) &= H\vec{X}_{\mathsf{q_D}}
\end{aligned} \tag{4.8}$$

We briefly describe each jump vector:

- $h^{(1)}$ is induced by the activity think1 and represents jobs completing that activity with propensity proportional to the number of jobs in think1 (i.e., $\vec{X}_{\mathsf{think1}}$), the rate of think1 ($rate(\mathsf{think1}) = 1/0.1 = 10$), and the processor scheduling factor $s_{\mathsf{P_A}} = 1$;

- $h^{(2)}$ is induced by the acquire node $\mathsf{r_B}$ and represents jobs competing for a thread from task $\mathsf{T_B}$ with propensity proportional to the number of jobs in $\mathsf{r_B}$ (i.e., $\vec{X}_{\mathsf{r_B}}$), the immediate rate H, and the scheduling factor $g_{\mathsf{T_B}}$;

- $h^{(3)}$ is induced by the activity compB and represents jobs completing that activity with propensity proportional to the number of jobs in compB (i.e., $\vec{X}_{\mathsf{compB}}$), the rate of compB ($rate(\mathsf{compB}) = 1/0.05 = 20$), and the processor scheduling factor $s_{\mathsf{P_{BD}}}$;

- $h^{(4)}$ is induced by the release node $\mathsf{q_B}$ and represents jobs that release their thread to task $\mathsf{T_B}$'s threadpool with propensity proportional to the number of jobs in $\mathsf{q_B}$ ($\vec{X}_{\mathsf{q_B}}$) and the immediate rate H;

- $h^{(5)}$ is induced by the activity think2 and represents jobs completing that activity with propensity proportional to the number of jobs in think2 (i.e., $\vec{X}_{\mathsf{think2}}$), the rate of think2 ($rate(\mathsf{think2}) = 1/0.1 = 10$), and the processor scheduling factor $s_{\mathsf{P_A}} = 1$;

- $h^{(6)}$ is induced by the acquire node $\mathsf{r_D}$ and represents jobs competing for a thread from task $\mathsf{T_D}$ with propensity proportional to the number of jobs in $\mathsf{r_D}$ (i.e., $\vec{X}_{\mathsf{r_D}}$), the immediate rate H, and the scheduling factor $g_{\mathsf{T_D}}$;

- $h^{(7)}$ is induced by the activity select and represents jobs completing that activity with propensity proportional to the number of jobs in select (i.e., $\vec{X}_{\mathsf{select}}$), the rate of select ($rate(\mathsf{select}) = 1/0.04 = 25$), and the processor scheduling factor $s_{\mathsf{P_{BD}}}$;

- $h^{(8)}$ is induced by the release node $\mathsf{q_D}$ and represents jobs that release their thread to task $\mathsf{T_D}$'s threadpool with propensity proportional to the number of jobs in $\mathsf{q_D}$ ($\vec{X}_{\mathsf{q_D}}$) and the immediate rate H.

4.1.3 Fluid approximation

Once we have all the jump vectors and the propensities, we can define the fluid approximation analogously to Section 2.1.2. The fluid approximation is defined as:

$$\dot{X}(\vec{X})_{\mathsf{i}} = \sum_{l=1}^{R} h_{\mathsf{i}}^{(l)} q(\vec{X}, \vec{X} + h^{(l)}) \tag{4.9}$$

where i is an activity, node or task, and R is the number of jump vectors (see Section 4.1.2).

4.2 Learning methodology

We estimate the LQN service demands through a Recurrent Neural Network (RNN). The approach is similar to the one presented in Chapter 3: the RNN considers a discretized version of the fluid approximation presented in Section 4.1.1, where each cell computes one advancement step from the (projected) LQN state to the next one. Unlike the QN approach, it is limited to estimating the service demands without learning the structure.

4.2.1 The integration method and the RNN

The RNN cell is built to mimic one step of the time-discretized LQN according to the fluid approximation. The cell has a structure that depends on the structure of the LQN being studied, with the service rates left as weights to be learned. This way, we directly connect the RNN weights and the system parameters, as in Chapter 3. Owing to the connection between RNN weights and the LQN model (i.e., we have a white-box model), we also keep the advantages of Chapter 3's approach: we can predict the dynamics of the system under different conditions than the ones seen at learning time, as we can replicate the system modifications on the model.

We apply the Strang Splitting method [MS16] to deal with stiffness in the fluid approximation. The fluid approximation introduces the term H to approximate instantaneous events, a large constant wrt. the rates to be learned (see Section 4.1.1). With such a different magnitude from other terms, this term introduces stiffness in the solution [AP98], which should, in principle, be dealt with using tiny integration steps. Such tiny integration steps create a deep RNN, with many cells that are slow to integrate and require vast amounts of memory. Thanks to Strang Splitting, we decompose differential equations as a sum of differential operators, each with its own timescale, which allows us to treat them differently.

We apply Strang Splitting by separating the equation terms into two operators: the ones that embed the H coefficient (that we call *fast operators*) are integrated over a shorter horizon. In contrast, the remaining ones (the *slow operators*) are integrated over a longer time horizon. We realize this by defining a vector F of length R where $F_l = 1$ if $q(\vec{X}, \vec{X} + h^{(l)})$ contains H (i.e., the propensity associated to the jump vector $h^{(l)}$ contains H), otherwise $F_l = 0$. We now incorporate F into the fluid approxima-

Algorithm 1 Forward Euler plus Strang splitting schema.

```
Require: $\hat{X}(t), \Delta t$
Ensure: $\hat{X}(t + \Delta t)$
1: $X_1 \leftarrow \hat{X}(t) + \dot{X}^f(\hat{X}(t))0.5\Delta t$
2: $X_2 \leftarrow X_1 + \dot{X}^s(X_1)\Delta t$
3: $\hat{X}(t + \Delta t) \leftarrow X_2 + \dot{X}^f(X_2)0.5\Delta t$
```

tion equation (see Equation (4.9)) in this way:

$$\begin{aligned}\dot{X}^f(\vec{X})_i &= \sum_{l=1}^{r} h_i^{(l)} F_l q(\vec{X}, \vec{X} + h^{(l)}) \\ \dot{X}^s(\vec{X})_i &= \sum_{l=1}^{r} h_i^{(l)} (1 - F_l) q(\vec{X}, \vec{X} + h^{(l)}) \end{aligned} \tag{4.10}$$

for each node, activity, and task i.

Algorithm 1 computes the integration of $\dot{X}$ on the interval $[0, \Delta]$ with $X(0) = x$ using the Strang Splitting technique and the forward Euler method [AP98]. The fast operator is repeated multiple times, interleaved with the slow ones, to compensate for the different integration horizons. With this approach, we reduce the computation costs where the equation is less stiff. The RNN cell computes $\hat{X}(t + \Delta t)$ given $\hat{X}(t)$ with one fast integration using $0.5\Delta t$ horizon, followed by a slow integration using Δt horizon and another fast integration using $0.5\Delta t$ horizon. This method does not guarantee the integration quality but proved satisfactorily precise if the integration step Δt is small enough.

Running example We apply Algorithm 1 to the running example. We observe that, among the propensities (see Equation (4.8)), $q(\vec{X}, \vec{X} + h^{(2)})$, $q(\vec{X}, \vec{X} + h^{(4)})$, $q(\vec{X}, \vec{X} + h^{(6)})$, and $q(\vec{X}, \vec{X} + h^{(8)})$ belong to the fast operator, while the remainder ones to the slow operator; hence, $F = (0, 1, 0, 1, 0, 1, 0, 1)$.

We now obtain the fluid approximation for fast and slow operators (see Equation (4.10)) as follows:

$$\begin{aligned}\dot{X}^f(\vec{X}) &= h^{(2)}q(\vec{X},\vec{X}+h^{(2)}) + h^{(4)}q(\vec{X},\vec{X}+h^{(4)}) + \\ &\quad + h^{(6)}q(\vec{X},\vec{X}+h^{(6)}) + h^{(8)}q(\vec{X},\vec{X}+h^{(8)}) \\ &= \textstyle\sum_{l\in\{2,4,6,8\}} h^{(l)}q(\vec{X},\vec{X}+h^{(l)})\end{aligned}$$

and

$$\begin{aligned}\dot{X}^s(\vec{X}) &= h^{(1)}q(\vec{X},\vec{X}+h^{(1)}) + h^{(3)}q(\vec{X},\vec{X}+h^{(3)}) + \\ &\quad + h^{(5)}q(\vec{X},\vec{X}+h^{(5)}) + h^{(7)}q(\vec{X},\vec{X}+h^{(7)}) \\ &= \textstyle\sum_{l\in\{1,3,5,7\}} h^{(l)}q(\vec{X},\vec{X}+h^{(l)})\end{aligned}$$

We are ready to incorporate the operators into the algorithm. We compute the integration of the LQN on $\hat{X}(t)$ over a step of size Δt (i.e., $\hat{X}(t+\Delta t)$). Firstly, we integrate the fast operator on $\hat{X}(t)$ over the first half of the step ($0.5\Delta t$):

$$\begin{aligned}X_1 &= \hat{X}(t) + \dot{X}^f(\hat{X}(t))0.5\Delta t \\ &= \hat{X}(t) + \left(\textstyle\sum_{l\in\{2,4,6,8\}} h^{(l)}q(\hat{X}(t),\hat{X}(t)+h^{(l)})\right) 0.5\Delta t\end{aligned}$$

Now, we integrate the slow operator on X_1 over the whole integration step Δt:

$$\begin{aligned}X_2 &= X_1 + \dot{X}^s(X_1)\Delta t \\ &= X_1 + \left(\textstyle\sum_{l\in\{1,3,5,7\}} h^{(l)}q(X_1,X_1+h^{(l)})\right)\Delta t\end{aligned}$$

We conclude by integrating the fast operator on X_2 over the second half of the integration step ($0.5\Delta t$):

$$\begin{aligned}\hat{X}(t+\Delta t) &= X_2 + \dot{X}^f(X_2)0.5\Delta t \\ &= X_2 + \left(\textstyle\sum_{l\in\{2,4,6,8\}} h^{(l)}q(X_2,X_2+h^{(l)})\right) 0.5\Delta t\end{aligned}$$

4.2.2 RNN setting

The RNN we propose learns the service rates from a set of traces. Intuitively, they represent the speed of the computational part of the systems: in the running example, we learn the rates μ_{think1}, μ_{think2}, μ_{compB}, and

μ_{select}. The traces contain the queue lengths, sampled at regular time intervals, and are all of the same length. The traces set is partitioned in the training set (80%) and the validation set (the remaining 20%).

The RNN is trained with the Adam algorithm [KB15] using the learning rate set at 0.1 and the integration step $\Delta t = 0.002$. The learning continues until the error computed on the validation set does not improve for 30 iterations. We use the Mean Absolute Error (MAE) as the error function between the ground truth traces and the RNN-predicted ones, divided by the total number of jobs in the system (which is fixed, thanks to the closed system assumption, see Section 2.1). Intuitively, it measures the proportion of misplaced jobs in the queues by the prediction.

4.3 Numerical Evaluation

We evaluate our method's estimation accuracy through synthetic and real case studies. Specifically, we measure the relative prediction error on the response time between the one observed in the original system against the one predicted by the LQN using the learned service demands. The replication package for this experimentation is available at `https://doi.org/10.5281/zenodo.8330607`.

4.3.1 Synthetic case studies

We considered three LQN models taken from the literature:

- Distributed Shop (`shop`) [Tri13];
- Microservices (`msrv`) [GCW19];
- Group Communication Server (`gcs`) [PS02].

We generated 40 different instances for each of the three case studies by sampling random values for each model parameter: number of clients, task threadpool size, service rates, processor multiplicities, and branch probabilities. Table 1 shows the sampling range of each model's parameters. To improve the coverage of the domain of our approach,

Case study	Clients	Threadpool sizes	Processor multiplicities	Activity service times
`shop`	$[10, 100]$	$[2, 60]$	$[1, 60]$	$[0.002, 1]$
`msrv`	$[50, 200]$	$[1, 70]$	$[1, 20]$	$[0.004, 1]$
`gcs`	$[80, 150]$	$[5, 100]$	$[5, 100]$	$[0.002, 0.1]$

Table 1: Instances generation ranges. OR-nodes probabilities are generated in $[0, 1]$ and form sound probability distributions.

some synchronous calls are transformed into asynchronous ones. As said, the RNN must learn each activity service rate in the LQNs.

Setup We now describe how we make the learning dataset traces (the sum of training and validation set) for each LQN instance. We considered 100 distinct initial population vectors, each one being a valid state in the LQN fluid approximation (see Section 4.1.2 for more details), plus the extreme ones where all system user start in each of the LQN entries. Each of the starting points, together with the LQN fluid approximation, is simulated 100 times using StochKit [San+11], sampled every 0.2 time-units, to obtain traces 5 time-units long, and we use the average of such traces. For the learning phase, we used a Google Cloud Compute Engine instance with 96 vCPUs, where each Compute Engine instance learned up to 5 instances in parallel. The learning phases took (on average) 6h45m for each `shop` instance, 5h09m for each `msrv` instance, and 6h34m for each `gcs` instance.

What-if To evaluate the quality of the model in scenarios not used in the learning phase, we perform a *what-if* analysis on the number of jobs circulating the system. Namely, we compared the response times using the true parameters against the learned parameters where the LQN has $k \in \{2, \cdots, 10\}$ times the jobs used at learning time. This range allows us to test the system under different saturation levels (i.e., moderate to severe). This allows us to see if our model fits the system correctly, rather than just replicating the observed inputs. In the first three boxplots of

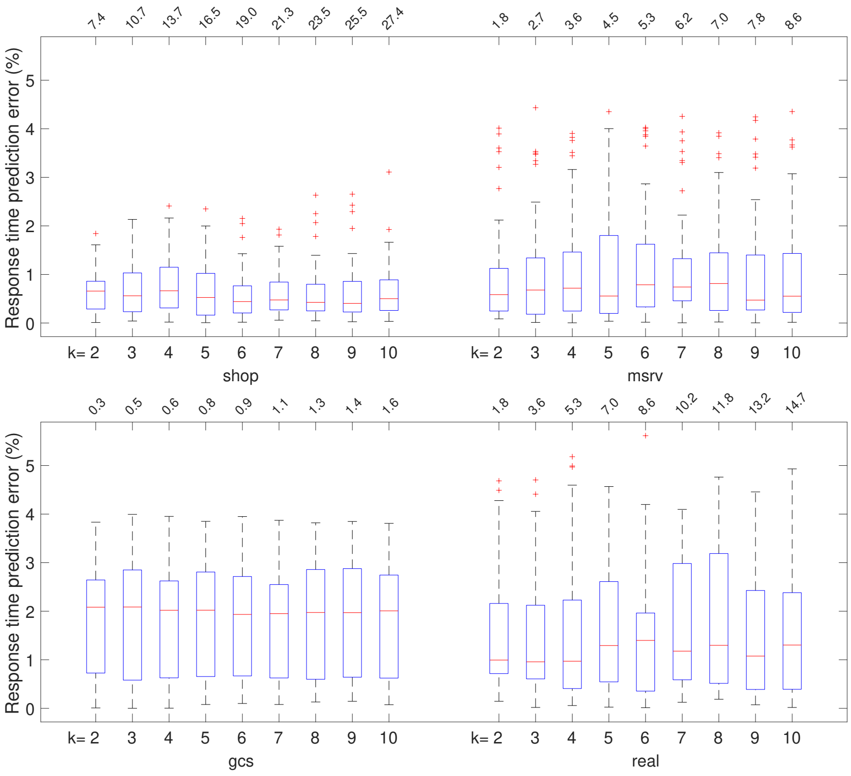


Figure 19: Response time prediction accuracy with population sizes not used during the learning.

Figure 19, we see: on the top line the average response time of all instances, on the x-axis of the plot the job multiplication factor k, on the y-axis the response time prediction error (in percentage). In each box plot, the red line is the average prediction error, the box represents values between the 25th and 75th percentile, whiskers extend to extreme values, and red crosses are outliers; on the top x axis, the average observed response times. We observe that the response time is always predicted with at most a 5% error even when the response time increases, thus with a good generalization across all the case studies.

4.3.2 Real Case Study

We now apply our approach to a real case study, i.e., on the data extracted from a Java implementation of the `msrv` [GCW19] case study. The objective of this section is to show that:

1. it is possible to measure, in a real system, the number of jobs in each state of the LQN fluid approximation;

2. the GPS thread scheduler is easily implementable in a real system;

3. (as a consequence of 1 and 2) this approach is effectively usable in a real context;

In the Java-based implementation of `msrv`, we associate each task to a Java `HttpServer` [1] where each context corresponds to an entry, while entry calls are realized through (synchronous or asynchronous) HTTP requests. Each `HttpServer` stores the incoming requests in a set and associates each to the requested entry. `HttpServer` is associated with a threadpool of workers that continuously pick one request from the incoming set uniformly at random (realizing GPS) and serve it. The Java implementation stores the number of clients in each fluid approximation state in a Redis database [2]. The Redis database is a large-scale key-value database with many industrial applications that executes operations on different elements concurrently: therefore, its impact on performance is negligible. We obtain the learning traces by periodically sampling the queue lengths from the database during the executions. We run our case study on a machine using the Linux kernel, where processor cores are scheduled using the CFS policy [3]. The Java implementation is parametric, alike the synthetic case study, such as to test the learning under many configurations.

[1]`https://javadoc.scijava.org/Java14/jdk.httpserver/module-summary.html`

[2]`https://redis.io/`

[3]`https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html`

Setup For the real case study, we generated 40 instances alike to Section 4.3.1. We generated 100 traces for each instance, with between 50 and 200 clients. Each trace is 5 seconds long, with the queue lengths sampled from Redis every 0.2 seconds. The RNNs completed the learning procedure in about 5 hours on Google Cloud Compute Engine.

What-if We repeated the analysis of Section 4.3.1 on the real case study, performing a what-if analysis on the number of clients that circulate the system. The plot labeled `real` of Figure 19 shows the prediction error on the real case study, in line with the synthetic counterpart (`msrv`). We observe analogous prediction errors, with few outlier instances between 5% and 6%.

4.3.3 Discussion of the results

In this section, we proved that, for the synthetic case studies:

- we can calibrate three different LQN models taken from the literature, each one instantiated with 40 different choices of parameters, using synthetic data;
- we can perform what-if analysis over the number of clients circulating the system and accurately predict the new response time.

With the real case study, we proved that:

- we can easily implement the GPS scheduling policy;
- we can extract the performance data needed to calibrate the LQN from a running real system;
- the procedure exposed in this chapter is applicable in a real context;
- the calibration procedure leads to an LQN model that produces reliable predictions of the real system.

Chapter 5

μP: A Development Framework for Predicting Performance of Microservices by Design

The technique proposed in Chapter 4 requires significant user intervention in the modeling process. The developer must provide an accurate description of the system internals (e.g., deployment schema, and interaction patterns) to enjoy faithful performance predictions. This requirement is a barrier to adopting performance evaluation practices: the developer must be knowledgeable of both the software domain and the modeling techniques to craft accurate models. Moreover, modern software engineering dictates continuous development cycles that imply frequent revisions to the performance model.

In this chapter, we want to overcome this requirement by automatically deriving the full model of the system. We already attempted this in Chapter 3's approach, where we require minimal information about the system (the queue lengths), but it is limited to simple systems. We need to relax those assumptions by adopting a more expressive model, such as LQNs, to study industrial cloud applications. Conversely to what we

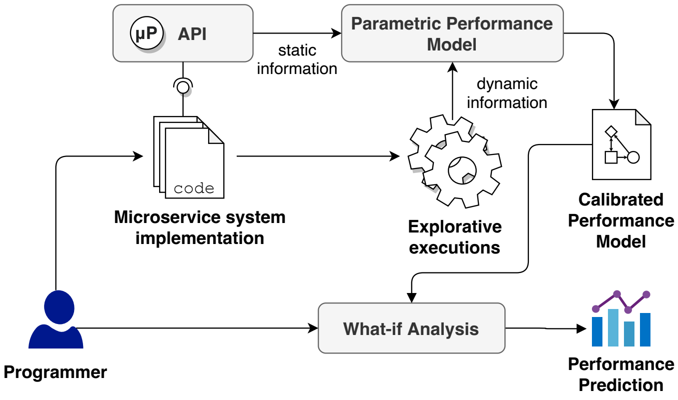


Figure 20: Overview of μP.

did in Chapter 4, we extract details about the internal structure of the system from its source code.

We concentrate on the microservice (MS) architecture, which is ubiquitous in cloud-based software system design. Its key concept is to develop applications as a suite of small interacting components, i.e., microservices, each running in its own process and communicating with lightweight mechanisms such as restful API [RR08]. This paradigm has attracted the attention from the industry; indeed, major vendors such as Amazon, Netflix, and Spotify have reportedly embraced MS [Blo13; sta15; Gol15]. We remark that LQNs provide constructs that directly map over MS primitives such as fork/join behavior, synchronous and asynchronous communication.

This chapter proposes μP, a framework to develop MS systems with predictable performance without further developer intervention beyond writing the actual code. The developer must adopt the APIs made available by μP, whose interface is similar to several state-of-the-art frameworks (e.g., Node.js and Java Spring Boot). To make this *performant-by-construction* objective possible, μP automatically generates an underlying performance model based on a fragment of layered queuing networks (LQNs) [FAW+09].

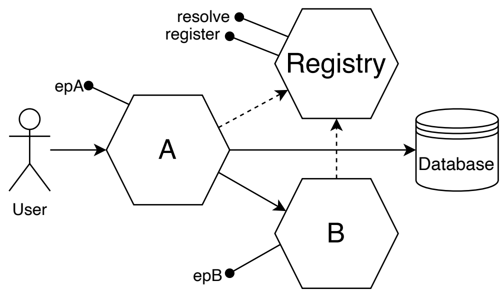


Figure 21: A simple translation service.

Figure 20 depicts an overview of μP. The starting point is the source code written using μP's API. From this, μP derives the *static* part of the performance model, i.e., which microservices and endpoints are available and their communication patterns. The static part of the model is an LQN with all the task, entries, activities, and arcs but lacking the execution time parameters and branching probabilities, which are learned at run-time. To this end, the system is deployed on the target platform and exercised through short explorative executions issued by a single user. During this phase, only application-level information is recorded, i.e., the execution time of each endpoint. This is readily available from lightweight monitoring tools (e.g., for the Java-based μP API, we rely on the `nanoTime` method) and represents a major difference with respect to state-of-the-art approaches based on profiling and monitoring, which discover the relationship between system's performance and its configurations by using low-level information such as CPU timing or I/O ratio [COQ21], which may not always be accessible. At the end of this analysis, we get a *calibrated* LQN model (i.e., with branching probabilities and execution times) that represents each part of the distributed system. μP provides several tools for developers to conduct *what-if* analyses without modifying the code. What-if analysis can answer questions regarding load variation (which will be the response time and throughput if we double the number of users), horizontal scalability (what will be the system dynamics after replicating one or more of its microservices) and vertical scalability (what happens if we assign to some microservices an increased amount of CPU cores).

Using a Java prototype, we analyze the predictive power of μP generated performance models of four benchmark MS applications. Our experiments show prediction errors consistently lower than $\sim$10% also under operating conditions not included in the datasets used for learning which exposed significant variations to response time and CPU utilization.

Running example Figure 21 shows the architecture of the running example used in this chapter. We will indicate microservice instances using the monospaced font. It has two microservices (`A` and `B`), plus a `Registry`. This running example is similar to the one analyzed in Chapter 4, plus the `Registry` microservice. The *registry* is a standard utility microservice that allows for a highly flexible MS deployment where components might come and leave the system at runtime. It is the only component that knows the identity of each instance of the microservices and the network topology. Each MS is equipped with different endpoints, denoted by an oblique line. `A` offers the `epA` endpoint that interacts with `B` and the database. `B` offers the `epB` endpoint that computes and returns a text. `Registry` offers the `register` endpoint to register a new live instance of a microservice and the `resolve` endpoint to return the URI of an instance.

The remainder of the chapter is organized as follows. Section 5.1 presents the μP framework API. Section 5.2 describes how μP builds the performance model from the code and the log of the explorative executions. Finally, Section 5.3 presents the numerical evaluation.

5.1 μP: Framework API

The API of μP is designed as a lightweight API that embeds the main concepts of the MS paradigm. Its API is similar to the most popular frameworks, like Spring Boot and Node.js. Figure 22 shows the class diagram. μP is language-agnostic, i.e., any object-oriented language with

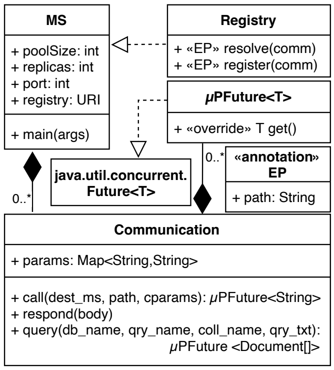


Figure 22: UML class diagram of μP's API.

message-passing technology is suitable for implementation. In this chapter, we consider a Java-based prototypical implementation and, in addition to the methods discussed in this section, μP applications are compatible with all the libraries of a standard Java application.

The abstract class `MS` represents the central actor of μP: developers subclass it to write their microservices (see Section 5.1.1). Methods that encode endpoints are annotated with `EP`. `Registry` is the microservice that implements the registry. The `Communication` class handles the communication between microservice and databases. Our implementation directly supports MongoDB databases; as future work, we plan to support other DBMSs. Finally, μ`PFuture` is our specialization of Java's *Future*.

5.1.1 Microservices Definition

In μP, each microservice is defined by subclassing `MS` and by specifying the TCP `port` on which it accepts incoming requests; `replicas`, the number of independent copies of the microservice (*replicas*); and `poolSize`, the threadpool size of each replica. Here we assume that microservice replicas are equivalent and independent; we leave non-equivalent replicas as future work.

The body of each endpoint is written as a regular `void` method with the `@EP` annotation and takes a `Communication` object as input. The `@EP` annotation has one attribute, called `path`, that specifies the endpoint's path. The `Communication` class encapsulates the communication infrastructure of the framework.

Call to endpoints The `Communication.call(dest_ms, path, cparams)` method establishes an asynchronous point-to-point channel between the calling microservice and the called one, i.e., `dest_ms`, at path `path`. At each invocation, μP spawns an auxiliary thread (in addition to those that form the threadpool of a microservice) responsible for managing the communication. The auxiliary thread contacts the registry through `resolve` to get the URI of a random replica of the microservice `dest_ms` (each replica of `dest_ms` has the same probability of being chosen). It then contacts the chosen replica and calls the specified endpoint (`path`) with the parameters stated in `cparams`. Each time the `call` method is invoked, it returns a μ`PFuture`, i.e., an object representing the communication. The μ`PFuture` class has the method `get()` which blocks until the auxiliary thread gets the callee's response. As said, μP's default communication pattern is asynchronous; the developer can realize a synchronous communication by invoking the `get()` method of the μ`PFuture` object [DCJ07].

Database communication We treat databases as blackbox entities with internals unknown to μP. The developers must register database instances through the `Registry.register`, passing as parameters the database name and the URI at which it is reachable. μP applications then interact with databases using the `Communication.query` method. `Communication.query(db_name, qry_name, coll_name, qry_txt)` represents a specialized version of `call` to perform a query on a MongoDB database. The `db_name` argument specifies the name of the database to contact, `qry_name` is a label that identifies the query, `coll_name` is the name of the collection in the database to query, and `qry_txt` is a string with the JSON encoding of the query. The return value of `Communica-`

```
1   class A extends MS {
2     poolSize = 5; replicas = 1;
3     @EP("/epA/") public void epA(comm){
4       bfuture = comm.call("B", "/epB/", comm.params);
5       bstr = bfuture.get();
6       dq = comm.query("db", "qry", "tr",
            "{\"_id\":\""+comm.params["word"]+"\"}");
7       qstr = dq.get();
8       ans = bstr + "=>" + qstr[0];
9       comm.respond(ans);}
10  }

11  class B extends MS {
12    poolSize = 2; replicas = 1;
13    @EP("/epB/") public void epB(comm){
14      hw = "hello" + "world";
15      comm.respond(hw);}
16  }
```

Figure 23: Microservices definitions using μP.

`tion.query` is a future that terminates with the returned records from the database.

End of the endpoint Once the endpoint is completed, it sends the response to the caller via the `Communication.respond` method.

Running Example Figure 23 shows the code for our running example, using an object-oriented pseudolanguage. The `A` microservice (with 5 threads) is implemented at lines 1–10. `A`'s only endpoint, `epA` (lines 3–9), calls the `B` microservice (line 4) and the database (line 6), then it returns the response. The `B` microservice is implemented at lines 11–16. `B` defines the `epB` endpoint (lines 13–15) that computes a string (line 14) and returns it (line 15).

5.1.2 Microservice Instantiation

μP microservices are launched as regular Java programs. `Registry` is launched without arguments, while the user-defined microservices require the URI at which the `Registry` is listening (i.e., `MS.registry`).

Upon startup, each `MS` instantiates an HTTP server listening on the microservice's port and advertises itself to the registry by calling the `register` endpoint. Upon receiving a request, a microservice determines the target endpoint to call (i.e., the annotated method) by looking at the request path and executes requests one after the other in a first-come-first-served fashion.

5.2 μP: LQN Model Derivation

The μP framework produces the LQN model of an MS system in two phases: static and dynamic analysis. The former derives the structure of the corresponding LQN model. The latter calibrates the LQN parameters (i.e., service demands and branch probabilities) using the logs of explorative execution runs.

5.2.1 Static analysis

The static analysis parses the MS system source code using JavaParser [1], looks for syntactic features (e.g., annotations, declarations, or `if` statements), and maps them into LQN constructs. The output of this phase is an uncalibrated LQN, i.e., an LQN model with all tasks, entries, nodes, activities, and arcs, but without any numeric value such as service-time distribution and branch probability (see Section 2.2 for a detailed description of the LQN paradigm). In the following, we describe the static analysis process of μP.

Network communication Network communication is directly represented by a task (called Net) that describes all the network communications between the system components. Unlike other tasks, Net does not have a software code counterpart but represents a hidden system resource.

We assume the network bandwidth is large enough to deliver the messages without contention. The μP generated LQN model represents

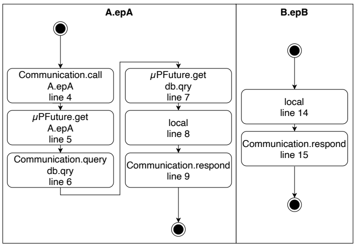
[1] `https://javaparser.org/`

this by assigning an infinite concurrency level to the Net task. Section 6.2 mentions a possible future research direction to mitigate such assumption; however, in our case studies, it did not affect the model prediction quality even under severe load (see Section 5.3).

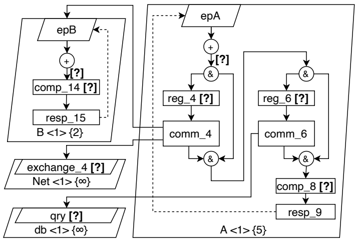
Each endpoint call (i.e., an invocation of `Communication.call`) generates a specific entry in the Net task that models the network time required for sending the request and receiving the corresponding response.

Net entries allow us to separate the time spent communicating on the network from the time spent computing the response. We distinguish the two times as the contention affects only computation time, not communication time. This is important for accurate predictions under various load levels (see Section 5.3.1). Instead, this distinction does not apply to communication involving `Registry` or databases as those entities are concurrent enough to not suffer from contention. Thus, we include network time inside the `Registry` and query service time without degrading model accuracy.

Databases Each database is modeled as a task whose entries represent queries. Queries are performed through the `Communication.query` API (see Section 5.1.1). We identify databases by looking for the individual `db_name` arguments of `Communication.query`; then we populate each task with an entry for each query (i.e., the distinct `qry_name` arguments among `Communication.query` calls that share the same `db_name`). Designing an accurate concurrency model for databases is an open problem [OK12]; therefore, we assume that the database is not the critical component of the system. Our solution models each database as an infinite server (i.e., each database task has an infinite threadpool). Each entry in the database task models the time to perform a query and its network communication without considering the inner details. The LQN model is flexible enough to incorporate a more accurate database description if available: we refer the reader to Section 6.2 for more details. However, in the case studies considered in Section 5.3, this assumption did not impair the model prediction power.



(a) Annotated CFGs.



(b) Uncalibrated model. Values indicated with '?' will be estimated by the μP dynamic analysis (see Section 5.2.2).

Figure 24: Static analysis applied to Figure 23.

Microservices μP translates each microservice class to an LQN task with the same name. The parser identifies the microservice classes by looking for class declarations that inherit from μP's `MS` class (see Section 5.1.1). The threadpool size is the number of software threads assigned to them (i.e., `MS.poolSize` of Figure 22); the replication number is the number of replicas of the microservice (i.e., `MS.replicas` of Figure 22). Although those parameters are statically identified, the developer can use μP tools to change them to perform what-if analysis under new deployment scenarios.

Endpoints Each Endpoint is translated in an entry inside the corresponding microservice's task. The parser identifies endpoints by looking for `EP`-annotated methods within microservice classes. The body of each such method is analyzed to produce its LQN representation.

Endpoint's logic From the endpoint source code, we derive the corresponding entry's control flow (i.e., the LQN encoding of how the endpoint interacts with each other). As a first step we transform the original MS code into a semantically equivalent version but without loops (via loop unwinding [Aho+07]) and method invocations (via inline expansion [Aho+07]).

We then analyze the inlined and unrolled endpoint code to generate its control flow graph (CFG) [All70], i.e., a graph where each node corresponds to a program statement, and arcs connect adjacent statements. In our case, nodes of the CFG form a direct acyclic graph (DAG) and generally have one outbound arc, except for `if` and `switch` statements that have multiple outbound arcs (one for each possible branch), and the last statement that does not have any outbound arc.

To carry out the translation of an endpoint in the corresponding LQN, we divide the CFG nodes into two categories: *local* and *remote*. Statements performing computations that do not contact other microservices or databases (e.g., assignments, insertion in a list, ...) are annotated with `local`. Remote nodes, instead, represent invocations to one of the following μP methods: `Communication.call`, `Communication.query`, μ`PFuture.get`, or `Communication.respond` (i.e., the μP communication APIs). We annotate remote nodes with the name of the invoked method. In μP, we model explicitly how microservices communicate. On the other hand, we collate groups of adjacent `local` nodes together. For example, complex algorithms that do not involve communication (e.g., sorting) appear as a single node in the final LQN model while their possibly variable runtimes will be captured through service time distributions (see Section 5.2.2).

Once the CFG is built, we traverse it to extract all the possible paths, each modeling a distinct sequence of operations. In our benchmarks, we observed at most two paths in an endpoint. This is because the MS architecture favors compact reusable endpoints [Ric18] with limited concerns, leading to simple control flows. Each path is then converted into its LQN representation by replacing path nodes with the corresponding LQN components. The LQN translation of local and remote nodes is detailed below.

A `Communication.call` node is translated into an LQN AND-node which splits the logic of the entry into two concurrent branches (i.e., forking behavior): a main thread that follows the regular execution flow of the endpoint and an auxiliary thread that takes care of the communication asynchronously. This mapping mechanism allows the modeling of the synch/asynch communication paradigm underlying modern MS frameworks (i.e., Node.js asynch/await [LML17] or Java callable future [UMF18]). The auxiliary thread induced by `Communication.call` node c has two activities: *i)* reg_c, that represents the interaction with `Registry`; *ii)* comm_c, that calls Net's exchange_c (i.e., the entry that represent c's network communication time) and the destination endpoint's entry (i.e., the one whose `@EP` annotation bears the `Communication.call`'s `path` argument). We remark that reg_c is a load independent delay (i.e., invariant to the active users in the system) as `Registry` functionality is simple.

A `Communication.query` node is translated similarly to `Communication.call` without the call to exchange_c, as μP factors the communication time into the database query's service time (see the *Network communication* paragraph).

The CFG nodes labeled with μ`PFuture.get` are translated into a join node that terminates the corresponding `Communication.call` or `Communication.query` auxiliary branch.

A `local` node n is translated into an activity comp_n representing the local computation with a stochastic computational weight as described in Section 5.2.2.

A CFG path ends with a `Communication.respond` node, namely r. When we encounter it, we create an activity resp_r and mark it as conclusive.

Finally, we connect the endpoint's LQN entry to an OR-node with an arc towards the first activity of each path, to encode the path choice probabilistically.

Running example We apply static analysis on the source code of Figure 23 to derive the (uncalibrated) LQN model of Figure 24b. First, μP creates an LQN model containing a task for each microservice definition, i.e., the A and the B tasks that correspond to the `A` and `B` microservices defined at line 1, and 3 of Figure 23. We then complete the declaration of those tasks by assigning the corresponding replication factor and threadpool size (i.e., the attributes poolSize and replicas in Figure 23).

Now that the outer structure of the LQN model is defined, μP analyzes the code of each endpoint, i.e., `epA` at lines 4–9, and `epB` at lines 14–15 of Figure 23. Figure 24a depicts the annotated CFGs of those endpoints. Each node shows the annotation, the invoked endpoint or query name (in case of remote nodes), and the CFG node identifier. Here we use the line number as the node identifier such that the CFG node generated by the source code in line 4 has the 4 suffix. As expected, μP finds only one path in both the CFGs since there are no conditional statements neither in `epA` nor in `epB`. To simplify the discussion, we focus on building the LQN corresponding to `epA` while `epB` is similarly constructed.

The `epA` CFG (see Figure 24a) begins with a `Communication.call` node which maps to the call towards the endpoint `epB` of line 4 of Figure 23. It identifies a microservice communication, which generates: the task Net (as it is the first communication) and the Net's exchange_4 entry that models the corresponding network time. In the epA's LQN entry, this `Communication.call` triggers the creation of an AND-node and an auxiliary thread with two activities: *i)* reg_4 (registry invocation); *ii)* comm_4, that calls Net's exchange_4 entry and the B's epB entry (i.e., the destination entry). The next node in the CFG corresponds with a μ`PFuture.get` and is translated into a join node that closes the previous communication (i.e., by blocking the main thread until a response is received).

Proceeding with the encoding, μP translates the next CFG node, i.e., the one with the `Communication.query` annotation corresponding to the statement in line 6 of Figure 23. This node is a database interaction and generates a new task (db) which models the `db` database with the qry entry. In the `epA` entry, the `Communication.query` is translated similarly to a `Communication.call` node, i.e., we add in the model the activities reg_6 (registry interaction), and comm_6 that calls the db's qry (i.e., the entry modeling the query). Analogously, we translate the subsequent μ`PFuture.get` node into a join node ending the database communication. The next CFG node, i.e., the one with the `local` annotation, is mapped into the comp_8 activity, which stands for the local computation at line 8 of Figure 23. Finally, the node annotated with

`Communication.respond`, concludes this path and is mapped in the LQN model with the resp_9 activity terminating the logic of the entry.

To consider endpoints with multiple paths, μP adds a probabilistic choice as the first action of each entry (the initial OR node). This allows us to model a probabilistic choice of the path to execute, and the probability is proportional to the number of times a path has been observed in the explorative phase.

The uncalibrated LQN model of the running example is shown in Figure 24b, where values indicated with '?' will be estimated by dynamic analysis of Section 5.2.2.

5.2.2 Dynamic analysis

Dynamic analysis completes (calibrates) the LQN model of Section 5.2.1 by computing service time distributions and branch probabilities using the log of the system's exploratory runs. We generate the logs by exciting the endpoints of interest using a single-user workload to avoid resource contention between concurrent requests. For each endpoint, we compute the confidence interval of the observed execution time (i.e., the time span between the beginning of a service and when it sends back the response) using the batch means procedure [FY97]. We stop this exploratory phase when, for each endpoint, the 95% confidence interval is wide at most 1% of its average. μP does not directly instrument database queries; rather, we use an indirect approach to quantify database performance through endpoints' logs that invoke them. This approach allows using commercial DBMSs, where modifying their source code is impossible or hard.

We observe that considering a single-user workload is not a limitation. More precisely, if there are multiple classes of users, we can perform the explorative execution with a workload that probabilistically chooses which behavior to take among the possible ones.

This section first discusses the log structure, then details the model calibration procedure. We apply this procedure to the running example to obtain its final LQN model.

Log Structure Each endpoint automatically keeps a distinct log (i.e., a sequence of events) for each of its executions. We call *event* a step in the endpoint's execution: those are generated and stored by μP framework implementation. μP defines five types of events: `begin`, `end`, `commStart`, `contactDest` and `commFinish`. The first two events delimit each log: `begin`, emitted when the endpoint execution begins; `end`, emitted by the invocation of `Communication.respond` (that concludes each endpoint code). Thus, the overall endpoint's *execution time* is the interval between the `begin` and `end` events. Each invocation of `Communication.call` and `Communication.query` produces three events: *i)* `commStart`, immediately after invocation; *ii)* `contactDest`, when the microservice/database request is sent to the destination; *iii)* `commFinish`, when the microservice/database response is received. The events have an associated set of attributes. All the events have the `time` attribute, the timestamp of when the event was emitted. Communication events have an extra attribute, called `dest`, that identifies the called endpoint or database query.

Model calibration Once the exploratory phase is completed, we perform a quantitative analysis of its execution logs. First, μP estimates the mean execution time of each endpoint by averaging the difference between the `time` attribute of the `begin` and the `end` events for each execution of the same endpoint. Then, the logs corresponding to several executions of the same endpoint are partitioned according to the CFG path that generated them. For example, to calibrate the LQN model of an endpoint with two distinct execution paths (e.g., due to an if statement in the code), the logs generated by that endpoint will be partitioned into two subsets, each corresponding to a specific path. Then by following the corresponding CFGs, μP identifies in the log the events necessary for calibrating the activities that represent them in the LQN model.

μP first calibrates the LQN activities related to communication, i.e., the CFG nodes annotated with `Communication.call` and `Communication.query`. In particular, for each communication node c present in an endpoint's CFG, we scan each log of each partition looking for the communication events whose `dest` attribute is the destination of c. To finalize the calibration μP tracks the following time intervals: *i)* reg_c (i.e., the registry interaction time) is the time between the `commStart` and `contactDest` events; *ii)* $exchange\_c$ (i.e., the network communication time in case the node is contacting another endpoint) is the time between the `commStart` and `contactDest` events minus the mean execution time of contacted endpoint; *iii)* db_c (i.e., the contacted query execution time in case the node is submitting a query) the time between the `commStart` and `contactDest` events. We then calibrate the activities reg_c, exchange_c, and the contacted query entry by fitting a service time distribution on, respectively, reg_c, $exchange\_c$, and db_c among all the logs of a partition. In our case studies (see Section 5.3), the normal distribution leads to precise models. comm_n does not perform computational work; hence it does not need calibration.

Subsequently, μP calibrates all the activities comp_n that correspond to a `local` node n in the CFG. Similarly to the database's case, μP does not explicitly track CPU times, which must be indirectly estimated. We do this because a direct estimate of the CPU times of each activity would be hardly applicable in practice (i.e., a developer would have to explicitly delimit CPU-bound computation). Instead, μP uses the communication API events to indirectly infer the length of the local computation, relieving the developer from the logging onus. In μP an endpoint is assumed to perform local computation whenever it is not communicating, and vice versa. For each `local` CFG node n, we need to find its the delimiting events, i.e., the pre_n and $post\_n$ events. To do this, we look at the nodes adjacent to n in the path. For example, if n is between the nodes m and p annotated, respectively, with μ`PFuture.get` and `Communication.respond`, this means that n happens after the end of the communication m (hence, pre_n is the `commFinish` event with the `dest` attribute set to m's destination) and immediately before terminating the endpoint (hence, $post\_n$ is the `end` event). Then, we calibrate comp_n using the time between the pre_n and $post\_n$ events. The resp activities, generated from `Communication.respond` CFG nodes do not need calibration.

#	Endpoint	Event	`time`	`dest`
①	A.epA	begin	68	—
②		commStart	68	B.epB
③		contactDest	74	B.epB
④		commFinish	88	B.epB
⑤		commStart	88	db.qry
⑥		contactDest	95	db.qry
⑦		commFinish	107	db.qry
⑧		end	111	—
⑨	B.epB	begin	77	—
⑩		end	86	—

Table 2: Running example logs.

As final step μP calibrates branch probabilities (i.e., the OR nodes generated by path translations see Section 5.2.1). We compute those probabilities as the ratio between each partition size over the sum of the sizes of all the partition of an endpoint (i.e., how many execution logs of that endpoint followed each path).

Running example We now calibrate the model of Figure 24b, using one explorative execution produced by a workload generator that contacts the `epA` endpoint. Table 2 shows the generated logs, separated by a horizontal line. The first column contains an identifier we use to pinpoint individual events throughout this section. The second column states the name of the endpoint that generated that log. The third, fourth, and fifth columns indicate the event type, `time`, and `dest` attribute (for communication events only).

As a first step, we calibrate the communications. The call of line 4 appears in the log with the events ②, ③, and ④. We assign to reg_4 the time interval between events ② and ③ (②–③, for short) and to exchange_4 the interval ③–④ minus the `B.epB` execution time. Similarly, the database invocation of line 6 corresponds to events ⑤, ⑥, and ⑦ : we calibrate reg_6 with the interval ⑤–⑥ and epB with the interval ⑥–⑦.

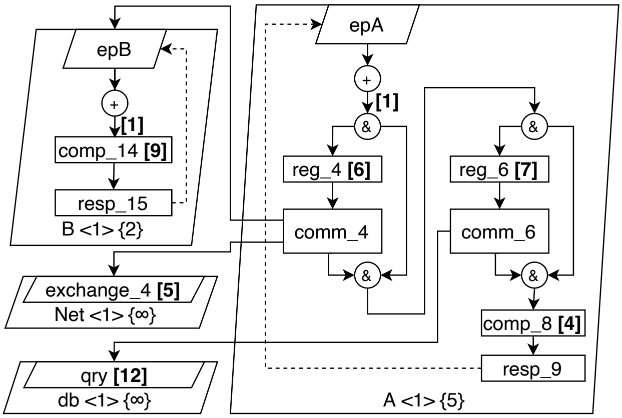


Figure 25: Running example's final model.

We consider the local computation activities. The activity comp_8 is calibrated with the interval ⑦–⑧, and comp_14 with ⑨–⑩ .

The probability calibration phase assigns 1 to all the arcs out of OR-nodes, as each endpoint has only one path.

Figure 25 shows the final running example's calibrated LQN model.

5.3 Numerical Evaluation

Case studies We evaluated μP using four benchmarks often used in performance evaluation literature: Acme Air [TS21], JPetStore [JA18], Tea Store [KES+18], and Teacher Management System [WDC20]. Acme Air is a booking system for a fictional airline carrier. It offers account management, flight search, booking, and check-in procedures. This case study is analyzed by the scientific community [Ade+17; RPT19]. JPet-Store is an online shop for exotic animal retailers. The website offers account management, animal catalog browsing, cart management, and checkout. Several papers use this case study [EH11; HRH08; Kal+21]. Tea Store is an e-shop for tea retailers. The website offers account management, tea catalog browsing, a recommendation system, cart management, and checkout. This case study is often used in performance evaluation, e.g. [COQ21], which we compare against (see Section 5.3.3). Unlike the previous ones, Teacher Management System (TMS) is a real case study used by the Texas Educator Certification training and testing program.

To evaluate the accuracy of the μP generated models, we implemented each case study using μP API such as their functionality matched the publicly available implementations. Then, we stressed the models' prediction power by predicting the systems' behaviors (response time and utilization) under load and deployment conditions far from the one explored at learning time (what-if analysis). Those values were compared to the ground-truth performance indices observed when concretely applying the new loads and deployment conditions and observed low discrepancy with the predicted indices. Our implementation of μP, of the case studies, the generated models, logs and data used to perform this numerical analysis is available online [2].

Performance indices We evaluated the generated models using two steady-state indices: *1)* mean end-to-end response time, i.e., the average time of a full user-system interaction; *2)* microservices' utilization, i.e., the ratio of threads that are busy servicing requests for each microservice.

Deployment The case studies were deployed on the Google Cloud Platform. Each microservice and workload generator had a dedicated `c2-standard-16` machine (16 CPUs and 64GB of RAM).

Model learning We learned the performance models of each case study using one exploratory execution run lasting 500 seconds after a discarded warmup phase of 50s. This time horizon attained the steady state condition in all cases on our platform. Table 3 gives details about the size of the learned models.

[2]`https://github.com/giulio-garbi/mup`

	AcmeAir	**JPetStore**	**TeaStore**	**TMS**
Tasks	4	6	7	6
Entries	17	23	70	18
Activities	35	50	192	36
Nodes	23	32	110	24
OR-nodes	11	14	30	12
Arcs	64	91	341	66
Paths	11	14	31	12

Table 3: LQN model sizes of the case studies.

Model validation We considered three kinds of *what-if* analysis:

- W1 (*system load*): can the models predict response time under increasing load?
- W2 (*vertical scaling*): can the models predict microservices' utilization when varying processors multiplicity and threadpool sizes?
- W3 (*horizontal scaling*): can the models predict utilization when using replicated instances of each microservice?

During the learning phase, μP uses traces obtained from a workload composed of only one client and deployed with a replication factor equal to 1 (i.e., one software thread and one replica for each microservice). The changes induced by W1, W2, and W3 expose contention behavior unseen in the learning phase, which will be predicted by the μP LQN model. Therefore, if the predictions under the tested scenarios are accurate, we can conclude that the generated LQN model is resilient enough to produce reliable performance analyses under operating conditions significantly different from those used in learning.

We validated our approach by comparing the system's measurements against the simulation of the corresponding LQN model. To accommodate minor semantical differences between the models generated by μP compared to the general-purpose LQN, we developed a custom LQN simulator (i.e., tailored on the LQN fragment considered by μP).

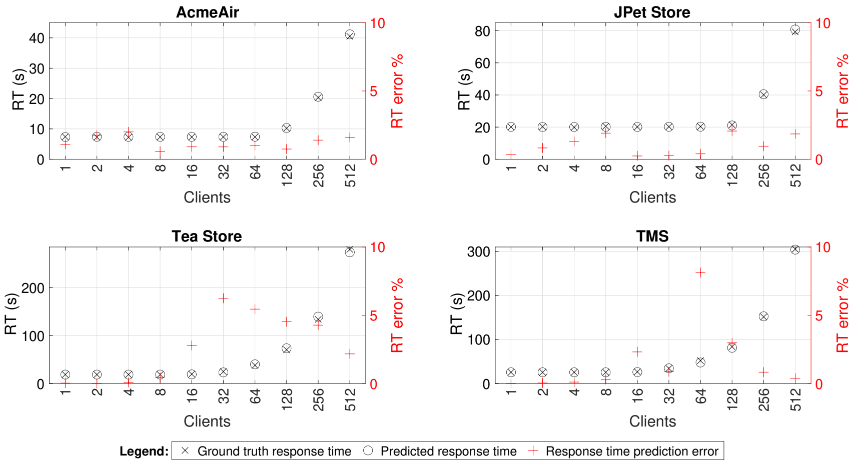


Figure 26: System load what-if (W1).

For each considered scenario, we simulated its model to collect the steady-state measures, and we observed that 6000 (simulated) seconds is enough to reach such state: indeed, those simulations (each one lasting less than 30s wall clock time) yielded confidence intervals whose width is at most 0.76% of the statistical means (at a 95% confidence level). To validate our results, we assume that each software thread corresponds to a core of the underlying hardware processor; anyways, μP models can accommodate any combination of threadpool size and processor multiplicity.

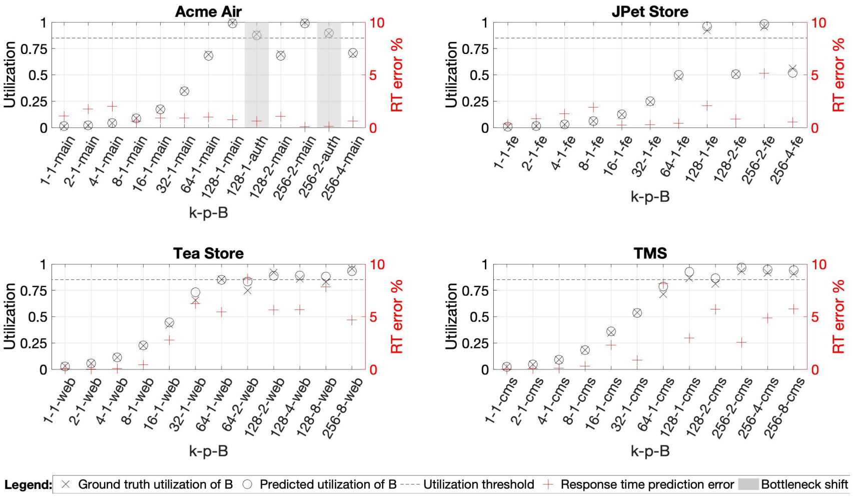
5.3.1 W1: System Load What-if

We validated μP models' prediction capabilities under system load variation by scaling the number of clients up to 512. Figure 26 shows the prediction error results. This scaling significantly changed the response time (axis *RT*) up to 15 times the original one. In Figure 26, the black $\times$ and $\bigcirc$ markers refer to the ground-truth and predicted system response time, respectively. The red $+$ markers indicate the prediction errors. Despite significant changes in the performance indices, μP models accurately track the system response time with a prediction error of less than 10% in all cases (up to 8.13%).

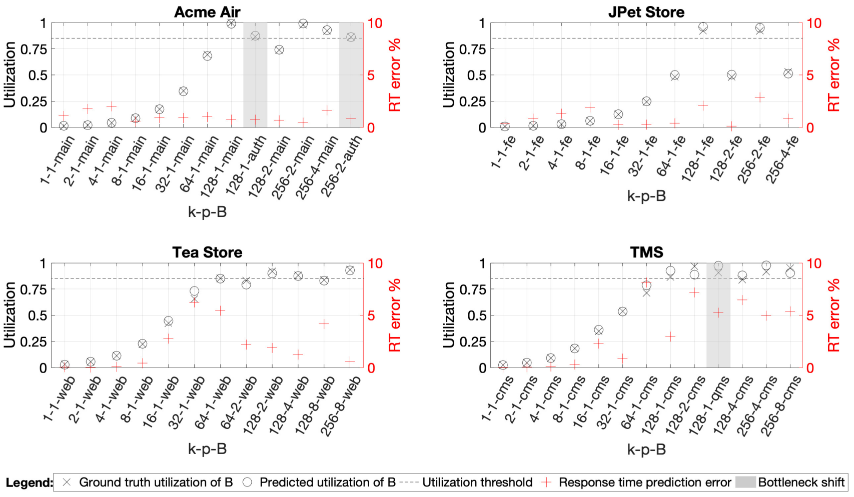
5.3.2 Performance Scaling What-if

We now stress the predictive power of the μP-generated models by considering two different performance scaling scenarios: *vertical* and *horizontal*. Vertical scalability varies the microservices' threadpool size (i.e., `poolSize` in Figure 22), while the horizontal scalability affects microservices' number of independent copies (i.e., `replicas` in Figure 22). This experiment reproduces the logic underlying a performance-driven *autoscaler* [GCW19] that uses μP model predictions to change the deployment schema of each case study. The considered autoscaler works as follows: if a microservice is a *bottleneck* (i.e., its utilization level is $\geq 85\%$), we double its resources (`poolSize` in the *vertical autoscaler*, `replicas` in the *horizontal autoscaler*); otherwise we double the number of clients. This experimentation tests two aspects: *i)* it mimics a typical microservices use case where the system is gaining new users and reacts by changing the deployment schema in the most effective way (i.e., relieving the load on the most stressed component); *ii)* we verify that μP effectively produces white-box models, i.e., each component is fitted individually rather than the system as a whole.

Vertical Scaling What-if (W2) Figure 27a shows the vertical scalability results obtained by applying the vertical autoscaler on the case studies. Unlike Figure 26, for each analyzed benchmark, the black markers (i.e., $\times$ and $\bigcirc$) identify the absolute value of the ground-truth and predicted microservices' utilization, respectively. In contrast, the red $+$ identifies the relative percentage error between the ground truth and predicted response times. The x-axis is labelled using the `k-p-B` pattern, where: *i)* `k` is the number of clients; *ii)* `B` is the most utilized microservice; *iii)* `p` is parallelism level of `B` (i.e., `poolSize`). Correspondingly we plot `B`'s utilization in the left y-axis (in black) and the error between the measured and simulated response time in the right y-axis (in red). The model accu-



(a) Vertical Scaling (W2).



(b) Horizontal Scaling (W3).

Figure 27: Performance Scaling What-ifs. Each label in the x-axis is in the form `k-p-B`, where: `k` is the number of clients; `B` is the most utilized microservice; `p` is parallelism level of `B`.

rately predicted the changes in threadpool sizes, with prediction errors always less than 10% (i.e., at most 8.61%) in all the considered configurations. Interestingly, these experiments produced different *bottleneck shifts*, i.e., when the autoscaler behavior induced a variation in the bottleneck microservice. For example, in Acmeair, 128 users caused the bottleneck microservice to be `main`, which then becomes `auth` after scaling the previous bottleneck. This behavior was accurately predicted by the μP generated model. In Figure 27 we highlighted bottleneck shifts by using a light gray background in the plot.

Finally, the prediction error over `B`'s utilization is at most (in absolute terms) 0.01 in AcmeAir, 0.04 in JPetStore, 0.09 in TeaStore, and 0.07 in TMS.

Horizontal Scaling What-if (W3) We applied the horizontal autoscaler and obtained the results of Figure 27b. Here, in the x-axis labels, `p` is `B`'s `replicas` number. Similarly to W2, there are bottleneck shifts in this experiment, and it induced a different scaling pattern (e.g., in AcmeAir the second bottleneck shift happens at the last steps, and in the TMS case study, the `qms` microservice is a bottleneck in one step). This allowed us to test the model under different conditions (i.e., a different multiplicity ratio between the microservices). The prediction accuracy is in line with W2. The response time prediction errors are less than 10% (i.e., at most 8.13%), while the prediction error over `B`'s utilization is at most 0.01 in AcmeAir, 0.04 in JPetStore, 0.08 in Tea Store, and 0.08 in TMS.

5.3.3 Discussion of the Results

W1, W2, and W3 strongly support the claim that, through μP, conducting highly accurate performance predictions for microservices is possible. Low errors in predicting the system response time (constantly under 10%) witness that the model is very accurate even under critical operating conditions (i.e., bottleneck shifts among microservices with task utilization very close to 1, see Figure 27). We can confidently say that μP is suitable for performance-driven self-adaptive applications [ITT17;

GCW19], even when the system's operating point moves around its saturation. In this scenario, μP significantly outperforms state of the art approaches [BHK11; COQ21] that reports prediction errors $\geq 21\%$ and $\geq 50\%$, respectively. Nonetheless, differently from [IWF07; BHK11], μP does not require developer intervention beyond writing the actual systems code, as these levels of accuracy are achieved by design.

Chapter 6

Conclusions and Future Work

This thesis presented three different methodologies for learning white-box performance models of distributed systems. The advantage of white-box models is a clear connection between each piece of the model and a specific counterpart in the system.

- Chapter 3 uses a recurrent neural network (RNN) to generate a queueing network (QN) model of the system by monitoring the number of pending requests at each system component over time. The novelty of this approach is that the RNN structure mimics an approximation of the QN dynamics (the fluid approximation) as a set of ordinary differential equations (ODE), where the learned weights map directly onto the QN parameters.

- Chapter 4 employs the RNN to calibrate a layered queueing network (LQN) of the system, i.e., calculate the speed of each LQN component, by observing how many requests are in each state of the system in an exploratory run. Similarly to the QN approach, the RNN is based on the fluid approximation (albeit extended to cater to LQN models), still enjoying the weights-to-LQN parameter mapping.

- Chapter 5 describes μP, a framework to develop a microservice-based system that by-design automatically generates a performance model without further developer intervention beyond writing the actual systems code.

The three methodologies are applied to several case studies, synthetic and real implementation of benchmarks from the literature. The results prove that the generated models are accurate.

As the three methodologies generate white-box models, we can use them to do what-if analysis, i.e., predict the impact of a change in the original system's performance indices by applying the same change to the model. We demonstrated low prediction errors (under 10% on the considered performance indices) in all the methodologies and tested case studies even though the changes significantly altered the system behavior. This means that the produced models are accurate to each component (instead of just modeling the overall system), thus catching the spirit of white-box models.

6.1 Threat to validity

Here we briefly discuss the criticalities of the three approaches.

Chapter 3's approach targets a fragment of the QN model, which restricts its applicability. This in principle rules out virtualized deployments where more than one node shares the same computation unit, or where the communication between the nodes is both synchronous and asynchronous. Although the QN modeling language cannot describe the inner behavior of each node, it might be enough for a coarser description of the system where each computation unit is treated as black box. This model proves useful if we have software that is too complex to model in detail where we can only act on the deployment (e.g., to scale up or down the number of processors available to each node), or where we do not have information about the inner structure (e.g., closed source software). We can resort to more expressive models if we need a detailed description of each computation unit: the approaches presented in Chapters 4 and 5 use the LQN modeling language.

Chapter 4 overcomes the limitations of Chapter 3's approach by employing a more expressive language. In comparison to Chapter 3, it requires that the user provides an uncalibrated LQN model of the system (i.e., a description of its internal structure). This might seem a strong limitation, but the literature provides several methods to automatically produce such models (see Section 2.4.1).

Chapter 5 relieves the user from this task by analyzing the system source code. It provides the μP framework, whose internal structure has a direct mapping on the LQN model, that the programmer uses to develop the distributed software and obtain the associated model. At present μP does not handle systems that do not conform to its API. Although this seems necessary to achieve performance *by construction*, various reasons could dictate interactions with components outside the developer's control: legacy code, third-party components, or systems that are too complex to be modeled explicitly. To mitigate this limitation, it is possible to exploit an indirect modeling method similar to the one used with databases or the network (see Section 5.2.1). Alternatively, encoding previous knowledge about them in the model is always feasible, e.g., by using techniques that produce queueing networks (see Section 2.4.1), or by employing the approaches of Chapters 3 and 4 on the legacy parts of the software.

More generally, the model calibration phase depends on the specific hardware platform used during learning: to overcome this issue, one could use transfer learning techniques to adapt the runtimes to a different platform [Jam+17].

6.2 Future Work

For the techniques of Chapters 3 and 4 we plan to improve the learning procedure acting from two different angles:

- reducing the amount of exploratory trace used by the learning infrastructure, by stressing the system parts that require more information using e.g. active learning [Kal+19]

- improving the learning infrastructure itself, using other learning methodologies such as residual networks [ZK16];
- automating the choice of the NN learning rate, which is a critical parameter to attain good prediction quality.

We also plan to improve μP (Chapter 5) by considering other DBMSs with known concurrency models (e.g., [DCS16]) and by using different logging strategies [Lin+16] to face the clock drift problem.

Appendix A

Proof of Theorem 2.1.1

Proof of Theorem 2.1.1. We construct $\hat{\mathbf{P}}$ and $\hat{\boldsymbol{\mu}}$ as follows:

$$\hat{\mathbf{P}}_{k,i} = \begin{cases} \pi_k & \text{if } i = k \\ \frac{\mathbf{P}_{k,i}}{1-\mathbf{P}_{k,k}}(1-\pi_k) & \text{if } \mathbf{P}_{k,k} < 1 \text{ and } i \neq k \\ \frac{1-\pi_k}{M-1} & \text{otherwise} \end{cases}$$

$$\hat{\boldsymbol{\mu}}_k = \begin{cases} \frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k & \text{if } \mathbf{P}_{k,k} < 1 \\ 0 & \text{otherwise} \end{cases}$$

We prove that, for each $i \neq k$ we have $\hat{\mathbf{P}}_{k,i}\hat{\boldsymbol{\mu}}_k = \mathbf{P}_{k,i}\boldsymbol{\mu}_k$ and $(\hat{\mathbf{P}}_{k,k} - 1)\hat{\boldsymbol{\mu}}_k = (\mathbf{P}_{k,k} - 1)\boldsymbol{\mu}_k$. Then (1) follows by substitution.

We now consider the case $\mathbf{P}_{k,k} < 1$.

$$\begin{aligned} \hat{\mathbf{P}}_{k,i}\hat{\boldsymbol{\mu}}_k &= \frac{\mathbf{P}_{k,i}}{1-\mathbf{P}_{k,k}}(1-\pi_k)\frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k \\ &= \frac{\mathbf{P}_{k,i}}{\mathbf{P}_{k,k}-1}(\pi_k-1)\frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k \\ &= \mathbf{P}_{k,i}\boldsymbol{\mu}_k. \\ (\hat{\mathbf{P}}_{k,k}-1)\hat{\boldsymbol{\mu}}_k &= (\pi_k-1)\frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k \\ &= (\mathbf{P}_{k,k}-1)\boldsymbol{\mu}_k. \end{aligned}$$

We now consider the case $\mathbf{P}_{k,k} = 1$. We remark that, in this case, $\mathbf{P}_{k,i} = 0$

if $i \neq k$.

$$\hat{\mathbf{P}}_{k,i}\hat{\boldsymbol{\mu}}_k = \frac{1-\pi_k}{M-1}0 = 0 = 0\boldsymbol{\mu}_k = \mathbf{P}_{k,i}\boldsymbol{\mu}_k.$$
$$(\hat{\mathbf{P}}_{k,k}-1)\hat{\boldsymbol{\mu}}_k = (\pi_k - 1)0 = 0 = 0\boldsymbol{\mu}_k = (\mathbf{P}_{k,k}-1)\boldsymbol{\mu}_k.$$

The point (2) is true by definition of $\hat{\mathbf{P}}$. Statement (3) can be shown as follows. When $\mathbf{P}_{k,k} < 1$:

$$\begin{aligned}\sum_i \hat{\mathbf{P}}_{k,i} &= \hat{\mathbf{P}}_{k,k} + \sum_{i\neq k}\hat{\mathbf{P}}_{k,i} \\ &= \pi_k + \sum_{i\neq k}\frac{\mathbf{P}_{k,i}}{1-\mathbf{P}_{k,k}}(1-\pi_k) \\ &= \pi_k + 1 - \pi_k = 1\end{aligned}$$

where the last statement follows because $\sum_i \mathbf{P}_{k,i} = 1$, $\sum_{i\neq k}\mathbf{P}_{k,i} = 1 - \mathbf{P}_{k,k}$. When $\mathbf{P}_{k,k} = 1$:

$$\begin{aligned}\sum_i \hat{\mathbf{P}}_{k,i} &= \hat{\mathbf{P}}_{k,k} + \sum_{i\neq k}\hat{\mathbf{P}}_{k,i} \\ &= \pi_k + \sum_{i\neq k}\frac{1-\pi_k}{M-1} \\ &= \pi_k + \frac{M-1}{M-1}(1-\pi_k) \\ &= \pi_k + 1 - \pi_k = 1\end{aligned}$$

Statement (4) can be shown observing that $0 \leq \pi_k < 1$, $1 - \mathbf{P}_{k,k} \geq 0$ (since $\mathbf{P}_{k,k} \leq 1$) and $1 - \pi_k > 0$. Statement (5) can be shown observing that $\boldsymbol{\mu}_k \geq 0$, $\mathbf{P}_{k,k} - 1 \leq 0$ and $\pi_k - 1 < 0$. $\square$

Bibliography

[Ade+17] Carlos M. Aderaldo et al. "Benchmark Requirements for Microservices Architecture Research". In: *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. 2017.

[Aho+07] Alfred V Aho et al. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[All70] Frances E. Allen. "Control flow analysis". In: *Proceedings of a Symposium on Compiler Optimization*. 1970.

[AM17] Mahmoud Awad and Daniel A. Menasce. "Deriving Parameters for Open and Closed QN Models of Operational Systems Through Black Box Optimization". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017.

[AP98] Uri M Ascher and Linda R Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. Siam, 1998.

[Arc+15] Davide Arcelli et al. "Control Theory for Model-based Performance-driven Software Adaptation". In: *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. 2015.

[Bal+04] Simonetta Balsamo et al. "Model-Based Performance Prediction in Software Development: A Survey". In: *IEEE Transactions on Software Engineering* 30.5 (2004).

[Bau+18] André Bauer et al. "On the value of service demand estimation for auto-scaling". In: *International Conference on Measurement, Modelling and Evaluation of Computing Systems*. 2018.

[BFW+19] Justus Bogner, Jonas Fritzsch, Stefan Wagner, et al. "Microservices in industry: insights into technologies, characteristics, and software quality". In: *2019 IEEE International Conference on Software Architecture Companion*. 2019.

[BHK11] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. "Automated extraction of architecture-level performance models of distributed component-based systems". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering*. 2011.

[Bix12] Joshua Bixby. *4 awesome slides showing how page speed correlates to business metrics at Walmart.com*. 2012. URL: `https://web.archive.org/web/20130116131103/http://www.webperformancetoday.com/2012/02/28/4-awesome-slides-showing-how-page-speed-correlates-to-business-metrics-at-walmart-com/`.

[BKK09] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. "Automated Extraction of Palladio Component Models from Running Enterprise Java Applications". In: *4th International Conference on Performance Evaluation Methodologies and Tools*. 2009.

[Blo13] Netflix Technology Blog. *Announcing Ribbon: Tying the Netflix Mid-Tier Services Together*. 2013. URL: `https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62`.

[Bol+05] Gunter Bolch et al. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley, 2005.

[Bor+13] Luca Bortolussi et al. "Continuous approximation of collective system behaviour: A tutorial". In: *Performance Evaluation* 70.5 (2013).

[BWB+19] Liang Bao, Chase Wu, Xiaoxuan Bu, et al. "Performance modeling and workflow scheduling of microservice-based applications in clouds". In: *IEEE Transactions on Parallel and Distributed Systems* 30.9 (2019).

[CBL15] F. Cecchi, S. C. Borst, and J. S.H. van Leeuwaardena. "Mean-Field Analysis of Ultra-Dense CSMA Networks". In: *ACM SIGMETRICS Performance Evaluation Review* 43.2 (2015).

[CDS10] Paolo Cremonesi, Kanika Dhyani, and Andrea Sansottera. "Service time estimation with a refinement enhanced hybrid clustering algorithm". In: *International Conference on Analytical and Stochastic Modeling Techniques and Applications (ASMTA)*. 2010.

[Che+18] Tian Qi Chen et al. "Neural ordinary differential equations". In: *Advances in neural information processing systems*. 2018.

[Cho+15] François Chollet et al. *Keras*. `https://keras.io`. 2015.

[CLL16] Bihuan Chen, Yang Liu, and Wei Le. "Generating performance distributions via probabilistic symbolic execution". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016.

[CMI11] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[COQ21] Clément Courageux-Sudan, Anne-Cécile Orgerie, and Martin Quinson. "Automated performance prediction of microservice applications using simulation". In: *29th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2021.

[CS14] Paolo Cremonesi and Andrea Sansottera. "Indirect estimation of service demands in the presence of structural changes". In: *Performance Evaluation* 73 (2014).

[DCJ07] Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. "A complete guide to the future". In: *European Symposium on Programming*. 2007.

[DCS16] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. "A Queueing Network Model for Performance Prediction of Apache Cassandra". In: *10th EAI International Conference on Performance Evaluation Methodologies and Tools*. 2016.

[DI04] A. Di Marco and P. Inverardi. "Compositional generation of software architecture performance QN models". In: *Fourth Working IEEE/IFIP Conference on Software Architecture*. 2004.

[EH11] Jens Ehlers and Wilhelm Hasselbring. "A Self-adaptive Monitoring Framework for Component-Based Software Systems". In: *Proceedings of the 5th European conference on Software architecture*. 2011.

[Ent19] HYS Enterprise. *Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience*. 2019. URL: `https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/`.

[FAW+09] Greg Franks, Tariq Al-Omari, Murray Woodside, et al. "Enhanced Modeling and Solution of Layered Queueing Networks". In: *IEEE Transactions on Software Engineering* 35.2 (2009).

[FY97] George S Fishman and L Stephen Yarberry. "An implementation of the batch means method". In: *INFORMS Journal on Computing* 9.3 (1997).

[Gar+13] Joshua Garcia et al. "Obtaining ground-truth software architectures". In: *Proceedings of the 2013 International Conference on Software Engineering*. 2013.

[GB10] Nicolas Gast and Gaujal Bruno. "A mean field model of work stealing in large-scale systems". In: *ACM SIGMETRICS Performance Evaluation Review* 38.1 (2010), pp. 13–24.

[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*. 2019.

[GDV12] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. "Probabilistic Symbolic Execution". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 2012.

[Gil07] Daniel T. Gillespie. "Stochastic Simulation of Chemical Kinetics". In: *Annual Review of Physical Chemistry* 58.1 (2007).

[GIT] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. "Service Demands Estimation in Layered Queueing Networks".

[GIT20] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. "Learning Queuing Networks by Recurrent Neural Networks". In: *11th ACM/SPEC International Conference on Performance Engineering*. 2020.

[GIT23] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. "μP: A Development Framework for Predicting Performance of Microservices by Design". In: *IEEE International Conference on Cloud Computing*. 2023.

[GKP19] Antonio Gulli, Amita Kapoor, and Sujit Pal. *Deep Learning with TensorFlow 2 and Keras*. Packt, 2019.

[GLD+21] Yu Gan, Mingyu Liang, Sundar Dev, et al. “Sage: Using Unsupervised Learning for Scalable Performance Debugging in Microservices”. In: *arXiv* (2021).

[Gol15] Kevin Goldsmith@GOTO. *Microservices at Spotify*. 2015. URL: `https://thenewstack.io/led-amazon-microservices-architecture/`.

[Goo18] Google. *Using page speed in mobile search ranking*. 2018. URL: `https://developers.google.com/search/blog/2018/01/using-page-speed-in-mobile-search`.

[Gra+20] Martin Grambow et al. “Benchmarking microservice performance: a pattern-based approach”. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 2020.

[Gro+21a] Johannes Grohmann et al. “SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation”. In: *ACM Transactions on Autonomous and Adaptive Systems* 15.2 (2021).

[Gro+21b] Johannes Grohmann et al. “SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications”. In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2021.

[Gro19] Groupon. *Highlights from Groupon Bengaluru's Tech Talk Series: From Monoliths to Microservices*. 2019. URL: `https://people.groupon.com/2019/bengaluru-tech-talk-journey-monoliths-microservices-geekfest/`.

[Guo+13] J. Guo et al. “Variability-aware performance prediction: A statistical learning approach”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering*. 2013.

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[HRH08] André van Hoorn, Matthias Rohr, and Wilhelm Hasselbring. “Generating Probabilistic and Intensity-Varying Workload for Web-Based Software Systems”. In: *Performance Evaluation: Metrics, Models and Benchmarks*. Ed. by Samuel Kounev, Ian Gorton, and Kai Sachs. 2008.

[HRW95] C Hrischuk, J Rolia, and C Murray Woodside. “Automatic generation of a software performance model using an object-oriented prototype”. In: *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 1995.

[INT18] Emilio Incerto, Annalisa Napolitano, and Mirco Tribastone. “Moving Horizon Estimation of Service Demands in Queuing Networks”. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2018.

[ITT17] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. “Software performance self-adaptation through efficient model predictive control”. In: *32nd IEEE/ACM International Conference on Automated Software Engineering*. 2017.

[IWF07] Tauseef Israr, Murray Woodside, and Greg Franks. “Interaction tree algorithms to extract effective architecture and layered performance models from traces”. In: *Journal of Systems and Software* 80.4 (2007).

[JA18] Reiner Jung and Marc Adolf. “The JPetStore suite: A concise experiment setup for research”. In: *Symposium on Software Performance*. 2018.

[Jam+17] Pooyan Jamshidi et al. “Transfer learning for performance modeling of configurable systems: An exploratory analysis”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. 2017.

[Jam+18] Pooyan Jamshidi et al. “Learning to sample: exploiting similarities across environments to learn performance models for configurable systems”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 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*. 2019.

[Kal+11] Amir Kalbasi et al. “MODE: Mix driven on-line resource demand estimation”. In: *Proceedings of the 7th International Conference on Network and Services Management*. 2011.

[Kal+19] Christian Kaltenecker et al. "Distance-based sampling of software configuration spaces". In: *Proceedings of the 41st International Conference on Software Engineering*. 2019.

[Kal+21] Anup K. Kalia et al. "Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021.

[KB15] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations*. 2015.

[KES+18] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2018.

[Kha+16] Hamzeh Khazaei et al. "Efficiency analysis of provisioning microservices". In: *2016 IEEE International Conference on Cloud Computing Technology and Science*. 2016.

[Kha+20] Hamzeh Khazaei et al. "Performance Modeling of Microservice Platforms". In: *IEEE Transactions on Cloud Computing* 10 (2020).

[KLK20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. "Resource Demand Estimation". In: *Systems Benchmarking*. 2020.

[KN17] Ajay Kattepur and Manoj Nambiar. "Service demand modeling and performance prediction with single-user tests". In: *Performance evaluation* 110 (2017), pp. 1–21.

[Koz10] Heiko Koziolek. "Performance evaluation of component-based software systems: A survey". In: *Performance Evalutation* 67.8 (2010).

[Kur70] Thomas G. Kurtz. "Solutions of ordinary differential equations as limits of pure Markov processes". In: *Journal of Applied Probability*. Vol. 7. 1. 1970.

[Lin+16] Qingwei Lin et al. “Log clustering based problem identification for online service systems”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. 2016.

[Lit19] Marin Litoiu. “Panel: AI and Performance”. In: *International Conference on Performance Engineering (ICPE)*. 2019.

[Liu+06] Zhen Liu et al. “Parameter inference of queueing models for IT systems using end-to-end measurements”. In: *Performance Evaluation* 63.1 (2006).

[Llo+18] Wes Lloyd et al. “Serverless computing: An investigation of factors influencing microservice performance”. In: *2018 IEEE International Conference on Cloud Engineering*. 2018.

[LML17] Matthew C. Loring, Mark Marron, and Daan Leijen. “Semantics of Asynchronous JavaScript”. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*. 2017.

[MAA15] Martin Abadi, Ashish Agarwal, and Paul Barham et Al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[Men08] Daniel A. Menascé. “Computing Missing Service Demand Parameters for Performance Models”. In: *International Computer Measurement Group Conference*. 2008.

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

[Mod14] ModerWeb. *How Microservices Saved the Day: Sean McCullough on Explosive Growth at Groupon*. 2014. URL: `https://modernweb.com/groupon-microservices/`.

[MS16] Shev MacNamara and Gilbert Strang. “Operator splitting”. In: *Splitting methods in communication, imaging, science, and engineering*. Springer, 2016.

[Obj07] Object Management Group. *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE). Beta 1*. Omg, 2007.

[OK12] Rasha Osman and William J Knottenbelt. “Database system performance evaluation models: A survey”. In: *Performance evaluation* 69.10 (2012), pp. 471–493.

[Pac+08] Giovanni Pacifici et al. "CPU demand for web serving: Measurement analysis and dynamic estimation". In: *Performance Evaluation* 65.6-7 (2008).

[Pea89] Barak A Pearlmutter. "Learning state space trajectories in recurrent neural networks". In: *Neural Computation* 1.2 (1989).

[PS02] Dorina C. Petriu and Hui Shen. "Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications". In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. 2002.

[Ric18] Chris Richardson. *Microservices patterns*. Manning Publications Company, 2018.

[RL19] Joy Rahman and Palden Lama. "Predicting the end-to-end tail latency of containerized microservices in the cloud". In: *2019 IEEE International Conference on Cloud Engineering*. 2019.

[RPT19] Mohammad Imranur Rahman, Sebastiano Panichella, and Davide Taibi. "A curated Dataset of Microservices-Based Systems". In: *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. 2019.

[RR08] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly Media, 2008.

[San+11] Kevin R. Sanft et al. "StochKit2: software for discrete stochastic simulation of biochemical systems with events". In: *Bioinformatics* 27.17 (2011).

[Sha+08] Abhishek B Sharma et al. "Automatic request categorization in internet services". In: *ACM SIGMETRICS Performance Evaluation Review* 36.2 (2008).

[Sie+12] Norbert Siegmund et al. "Predicting performance via automated feature-interaction detection". In: *2012 34th International Conference on Software Engineering*. 2012.

[Sie+15] Norbert Siegmund et al. "Performance-influence Models for Highly Configurable Systems". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015.

[SJ11] Charles Sutton and Michael I Jordan. "Bayesian inference for queueing networks and modeling of Internet services". In: *The Annals of Applied Statistics* (2011).

[Spi+15] Simon Spinner et al. “Evaluating approaches to resource demand estimation”. In: *Performance Evaluation* 92 (2015).

[sta15] The news stack. *What Led Amazon to its Own Microservices Architecture*. 2015. URL: `https://thenewstack.io/led-amazon-microservices-architecture/`.

[Ste07] William J Stewart. “Performance modelling and markov chains”. In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. 2007.

[SW18] Akshitha Sriraman and Thomas F Wenisch. “μTune: Auto-Tuned Threading for OLDI Microservices”. In: *13th USENIX Symposium on Operating Systems Design and Implementation*. 2018.

[SWM17] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. “Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models”. In: *arXiv* (2017).

[TGH12] Mirco Tribastone, Stephen Gilmore, and Jane Hillston. “Scalable Differential Analysis of Process Algebra Models”. In: *IEEE Transactions on Software Engineering* 38.1 (2012).

[TR14] Alexander Tarvo and Steven P. Reiss. “Automated analysis of multithreaded programs for performance modeling”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 2014.

[Tri+12] Mirco Tribastone et al. “Fluid Rewards for a Stochastic Process Algebra”. In: *IEEE Transactions on Software Engineering* 38 (2012).

[Tri13] Mirco Tribastone. “A Fluid Model for Layered Queueing Networks”. In: *IEEE Transactions on Software Engineering* 39.6 (2013).

[TS21] Doug Tollefson and Andrew Spyker. *Acme Air Sample and Benchmark*. 2021. URL: `https://github.com/acmeair/acmeair`.

[TV10] Stefan Tilkov and Steve Vinoski. “Node. js: Using JavaScript to build high-performance network programs”. In: *IEEE Internet Computing* 14.6 (2010).

[UMF18] Raoul-Gabriel Urma, Alan Mycroft, and Mario Fusco. *Modern Java in Action*. Manning publications, 2018.

[Val+17] Pavel Valov et al. “Transferring Performance Prediction Models Across Different Hardware Platforms”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017.

[Wan+16] Weikun Wang et al. “Maximum likelihood estimation of closed queueing network demands from queue length data”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. 2016.

[Wan+18] Weikun Wang et al. “QMLE: A methodology for statistical inference of service demands from queueing data”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3.4 (2018).

[WC13] Weikun Wang and Giuliano Casale. “Bayesian service demand estimation using Gibbs sampling”. In: *IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. 2013.

[WDC20] Andrew Walker, Dipta Das, and Tomas Cerny. “Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study”. In: *Applied Sciences* 10.21 (2020).

[WFP07] Murray Woodside, Greg Franks, and Dorina C Petriu. “The future of software performance engineering”. In: *Proceedings of the Future of Software Engineering*. 2007.

[Xie+15] Qiaomin Xie et al. “Power of d choices for large-scale bin packing: A loss model”. In: *ACM SIGMETRICS Performance Evaluation Review* 43.1 (2015), pp. 321–334.

[ZH12] Dmitrijs Zaparanuks and Matthias Hauswirth. “Algorithmic Profiling”. In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 2012.

[Zhe+05] Tao Zheng et al. “Tracking time-varying parameters in software systems with extended Kalman filters.” In: *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. 2005.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. “Wide residual networks”. In: *arXiv* (2016).

[ZLW11] Tao Zheng, Marin Litoiu, and Murray Woodside. “Integrated Estimation and Tracking of Performance Model Parameters with Autoregressive Trends”. In: *2nd ACM/SPEC International Conference on Performance Engineering*. 2011.

[ZWL08] Tao Zheng, C. Murray Woodside, and Marin Litoiu. “Performance Model Estimation and Tracking Using Optimal Filters”. In: *IEEE Transactions on Software Engineering* 34.3 (2008).