Neural Density Estimation of Response Times in Layered Software Systems

Zifeng Niu and Giuliano Casale

Abstract—Layered queueing networks are a class of performance models for software systems in which distributed resources are allowed to be possessed simultaneously. Estimating response times in a layered system is an essential but challenging analysis dimension in Quality of Service (QoS) assessment. Current analytic methods are capable of providing accurate estimations of mean response times. However, accurately estimating the response time density based on queueing theory for service level agreements is a demanding task. This paper proposes a novel hybrid framework that leverages phase-type (PH) distributions and neural networks to provide accurate density estimations of response times in layered queueing networks. The core step of this framework is to recursively obtain response time distributions in submodels that are used to analyze the model by means of decomposition. We describe the conditional response time distribution as a mixture of density functions for which we learn the parameters through a Mixture Density Network (MDN). The approach recursively propagates MDN predictions across software layers using PH distributions and performs matrix operations to estimate end-to-end response time densities. Extensive numerical experiment results show that our scheme significantly improves density estimations and delivers an error reduction by more than 70% compared to the state-of-the-art.

Index Terms—Quality of Service, response time density estimation, layered queueing networks, phase-type distributions, mixture density networks.

1 Introduction

During the last few decades, performance has increasingly become an influential factor in software engineering [1]. End-to-end response time is a key performance index, but in some situations mean value estimation is insufficient for system designers, who may want to know the quantile statistics or the extent of variability to assess compliance to service level objectives. Queueing network models (QN) serve as a common performance model used for software analysis [2]. Although the problem of approximating response time distributions in QN models has been extensively studied over many years, it remains a challenging task because conventional analytic methods are inherently limited by their underpinning technical assumptions, such as those that prescribe overtake-free job behavior under first come first serve (FCFS) scheduling [3]. Besides, many services today, such as distributed software applications or business processes, feature concurrent services structured according to a layered architecture. Such services are not simple to model using ordinary queueing networks since layering is a form of simultaneous resource possession that does not meet the product-form solution assumptions [4]. The lack of simple closed-form analytic solutions compounds the challenge of estimating response time distributions.

In this paper, we propose a framework to provide accurate density estimations of response times in software performance models with layered architectures. The framework requires estimating response time distributions in a set of ordinary queueing networks that are constructed from a layered queueing network (LQN). However, current analytic methods for response time distributions are not accurate since they are limited by strict assumptions on service discipline, number of servers, and Markovian networks. Learning-based methods offer a number of opportunities and one of the goals of this paper is to understand how machine learning (ML) can benefit software performance modeling. We utilize Mixture Density Networks (MDNs) to approximate the response time distribution conditional on the known parameters of an ordinary queueing network representing a software layer. MDNs use a mixture model to fit probability distribution and a fully-connected neural network to map the conditional relation. In addition to MDNs for predictions, we use phase-type (PH) distributions to pass density predictions across layers. The proposed framework can estimate response time densities in complex systems and achieve high accuracy in our experiments. This approach therefore offers a novel contribution to the field of QoS assessment for layered models. The paper is an extended version of [5] with the following new contributions:

- We extend activity graphs to include selection and loop patterns. We propose a way to model the duration in a loop using Markov process, and illustrate how phase-type random variables are leveraged to cope with arbitrary activity graphs that consist of sequences, selections and loops.
- We apply Markov analysis to characterize durations of service requests between layers. This leads to an end-to-end analytic framework based on PH distributions that allows tractability of probability distributions throughout the whole layered system. Combined with MDNs, a formal scheme is presented.
- We increase the size of experiments significantly and study error distributions based on a large number

• *Z. Niu and G. Casale are with the Department of Computing, Imperial College London, London, UK, SW7 2AZ.*
E-mail: zifeng.niu19@imperial.ac.uk, g.casale@imperial.ac.uk

of model instances. Extensive numerical results illustrate the effectiveness of our scheme and indicate that it delivers a significant error reduction compared to the state-of-the-art.

The rest of the paper is organized as follows: Section II conducts a review on response time distribution analysis. Section III describes the concepts and related techniques that are used in this paper. Section IV presents how we use MDN analyzers. Section V proposes a scheme for estimating response time densities in layered systems. Section VI evaluates the performance of our MDN analyzers and density estimation framework. Section VII concludes the paper and gives future research directions.

2 Related Work

In queueing theory, tracking a tagged job is a commonly used way to analyze response time distributions [6]. A typical method introduced in [3], [6] is to derive the Laplace–Stieltjes transform (LST) of passage time for the tagged job in ordinary queueing networks and decondition the start state to obtain the product-form LST of the steady-state response time distribution. However, this method can only be applied when the service discipline is FCFS. In addition, its state-space analysis quickly suffers from state explosion. Fluid approximations have been used successfully to approximate response time distributions in closed networks that consist of delay and processor sharing (PS) queueing stations [7], [8]. Here, the mean-field fluid approximation based on Kurtz's theorem [9] is difficult to use in networks with multiclass FCFS stations. All the methods discussed above are limited to Markovian queueing networks. At present, there are few analytic methods that can be applied without assumptions on product-form, Markovian properties, and service discipline.

Layered queueing networks are a type of non-product-form queueing network where the service time of one station may depend on other stations [10]. In the past, Gamma distributions have been applied to approximate response times in LQNs [11], but estimation errors can be large when clients make a non-deterministic number of requests to other servers. A closed-form formula of response times has been then derived in [12] aiming at enhancing estimations for such cases. However, it is mainly restricted by two assumptions, (a) stations have a single server, and (b) service demand distributions of requests are exponential. The main idea of the analytic solution for LQNs is to take the sum of variance estimates of requested services [13], [14] to approximate the variance of end-to-end response time. The response time densities estimated by the state-of-the-art LQN solver [14] can be less accurate in complex LQNs, especially for those with multiple job classes and heavy resource contention.

The limitations of conventional analysis have led to an interest in learning-based methods that parameterize the density via neural networks, thus free the estimation from certain assumptions. Among these methods, the mixture density network [15] is well suited for approximating unknown conditional probability distributions. Over the past few years, MDNs have been widely used in distinct areas and proved to be effective in density estimations [16], [17], [18]. In the queueing literature, [19] first used this technique to predict probability distributions of customer waiting times in an open service system. Then, MDNs are employed to predict the end-to-end delay distribution in a tandem queueing network by [20] and waiting time distributions in open queues with distinct Kendall's notations by [21].

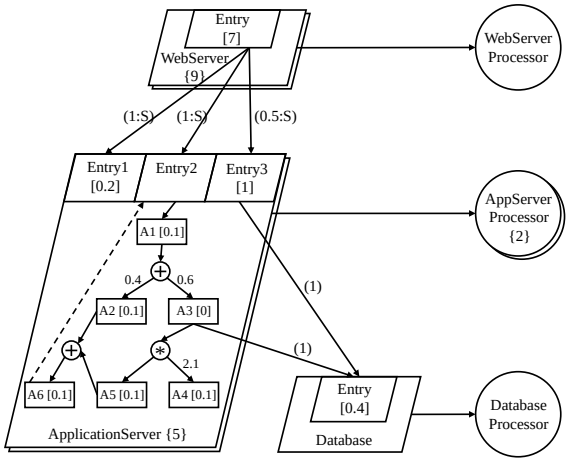

Fig. 1. Example of a LQN model.

3 Background

3.1 Layered Queueing Networks

Layered queueing networks are a form of extended queueing networks primarily used for modeling various distributed software systems [22], [23], [24]. Fig. 1 shows an LQN example that we use to illustrate all of the features considered in this paper. In LQN models, software resources are called *tasks* and shown as parallelograms. Tasks are executed by hardware resources called *host processors*, represented by attached circles. When acting as a server, the tasks usually have an FCFS discipline while the processors are scheduled by PS, a discipline under which the processor runs all jobs simultaneously, and each job experiences an equal fraction of the processor service capacity. The stacked notation makes a task or a processor multiserver, its multiplicity is indicated in the brace. For instance, *ApplicationServer* denotes a task with five threads and runs on *AppServer* which may be a two-CPU multiprocessor. Distinct service classes are called *entries* and represented by smaller parallelograms contained by tasks. The execution logic of each entry is depicted as an *activity graph* in which *activities* represent the smallest operation units and are shown as rectangles. For example, the execution of *Entry2* in Fig. 1 is described by an activity graph consisting of six activities, direct arrows, decision/merge nodes (small circles with + symbols inside), and a loop node (small circle with $*$ symbol inside). The decision node selects one branch to execute, $A2$ with probability 0.4 and $A3$ with probability 0.6. After running $A3$, $A4$ is repeated 2.1 times on average, followed by $A5$. Here, activity graphs that contain fork/join nodes are not considered, we leave this pattern for future work. The host demand of each activity is characterized by mean and variance, and the mean duration is indicated in the square bracket. The activity graphs are

TABLE 1
Submodel Queueing Parameters

Parameter	Definition	Range of Value
N	Population	1-100
Z	Think time	0.1-10
D	Host demand mean	0.05-2
c^2	Host demand SCV	0.1-2.5
m	Number of servers	1-32

not drawn explicitly if they are only composed of a single activity, in which case, the mean host demand is directly shown in the parent entry.

A key feature of layered queueing models is that services may include nested *calls* to other services. Calls model service requests and are represented by directed arrows going from one activity to an entry of another task. There are two main types of call: *synchronous* and *asynchronous*. The synchronous call is a blocking request during which the sender waits for the reply (dashed arrow), while the asynchronous call is a send-no-reply request, i.e., the execution continues after sending the call. In this paper, we focus on the synchronous call which captures the pattern of standard remote procedure calls (RPC) in most layered software architectures. We leave the asynchronous call for future work. The amount of calls indicated in parenthesis alongside the request arrow is either *deterministic* or *stochastic*, both cases are considered in our scheme. Stochastic requests are assumed to be geometrically distributed, and the number in parenthesis is the mean value. In Fig. 1, the task *WebServer* does not receive any calls. It is the system workload generator named *reference tasks*. The interested readers are referred to [25], [26] for more details of LQNs.

In terms of the analysis of a LQN, several techniques exist [14], [27], [28], [29], [30]. The main way is to decompose an overall model into a set of submodels (ordinary queueing networks) that admit a product-form solution. Each submodel is then solved via analytic techniques such as approximate mean value analysis (AMVA) [31]. The result iterates among all submodels until meeting certain convergence criteria, then we can obtain mean performance measures of the LQN [10]. SRVN layering [27] and MOL layering [28] are two ways to construct submodels. In this paper, we adopt SRVN layering in which every submodel consists of a delay and a queueing station. For each submodel, we focus on the queueing parameters defined in Table 1.

3.2 Class Switching and Chains

In a closed queueing network with N stations and K job classes, the routing matrix $R = [r_{iu,jv}]$, $i, j = 1, \ldots, N$, $u, v = 1, \ldots, K$ defines a finite-state discrete-time Markov chain (DTMC). The station-class pair (j, v) can be reached from pair (i, u) with probability $r_{iu,jv}$. Jobs can switch class in the network if there exists $r_{iu,jv} \neq 0$ for $u \neq v$. All job classes can be divided into $0 < L \leq K$ disjoint sets named $chains$. Each chain is a set of classes that can switch jobs with each other, but not with classes in other chains. This means jobs inside a chain will never escape from it. Therefore, the number of jobs in each chain remains constant at all times. It is natural to think of transferring a closed queueing network with L chains to a closed queueing network with L independent job classes and measuring the chain performance [32]. The methods have been developed to calculate the performance measures per chain and then per class in each chain [33], [34], [35, §7.3.6].

3.3 Phase-type Distributions and Closure Properties

In this paper, we cope with various response time distributions in a layered system by phase-type (PH) distributions. A PH distribution is defined as the distribution of the time to absorption in a finite continuous-time Markov chain (CTMC) with one absorbing state. It is determined by a pair $(\boldsymbol{\alpha}, \boldsymbol{T})$, where $\boldsymbol{\alpha}$ and $\boldsymbol{T}$ are the initial probability vector and subgenerator matrix among the transient states of the CTMC. The expectation and variance of PH random variables are expressed in terms of $\boldsymbol{\alpha}$ and $\boldsymbol{T}$ [36] as follows

$$E[X] = (-1)\boldsymbol{\alpha}\boldsymbol{T}^{-1}\mathbf{1} \tag{1}$$

$$Var[X] = 2\boldsymbol{\alpha}\boldsymbol{T}^{-2}\mathbf{1} - (\boldsymbol{\alpha}\boldsymbol{T}^{-1}\mathbf{1})^2 \tag{2}$$

where $\mathbf{1}$ is a column vector of ones. In place of $Var[X]$, the squared coefficient of variation $SCV = Var[X]/(E[X])^2$ is often used in practice.

The same distribution of the time to absorption can be generated by distinct absorbing CTMCs. In other words, one distribution could have multiple $(\boldsymbol{\alpha}, \boldsymbol{T})$ representations. In our research, we adopt the unique and minimal acyclic CTMC representation named *canonical form* [37] for the purpose of limiting the number of states and parameters and reducing the complexity of matrix multiplication.

The closure properties of PH distributions are studied in [36]. Here we cite two properties relevant to this paper:

- If X_i and X_j are statistically independent random variables with PH distributions $(\boldsymbol{\alpha}_i, \boldsymbol{T}_i)$ and $(\boldsymbol{\alpha}_j, \boldsymbol{T}_j)$, then $X_i + X_j$ is a PH random variable $(\boldsymbol{\alpha}, \boldsymbol{T})$, where

$$\begin{aligned} \boldsymbol{\alpha} &= (\boldsymbol{\alpha}_i, (1 - \boldsymbol{\alpha}_i\mathbf{1})\,\boldsymbol{\alpha}_j), \\ \boldsymbol{T} &= \begin{pmatrix} \boldsymbol{T}_i & -\boldsymbol{T}_i\mathbf{1}\boldsymbol{\alpha}_j \\ \mathbf{0} & \boldsymbol{T}_j \end{pmatrix} \end{aligned} \tag{3}$$

- A finite mixture of PH random variables $\sum_{i=1}^{k} p_i X_i$, where $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$, is a PH random variable $(\boldsymbol{\alpha}, \boldsymbol{T})$, where

$$\begin{aligned} \boldsymbol{\alpha} &= (p_1\boldsymbol{\alpha}_1, p_2\boldsymbol{\alpha}_2, \ldots, p_k\boldsymbol{\alpha}_k), \\ \boldsymbol{T} &= \mathrm{diag}(\boldsymbol{T}_1, \boldsymbol{T}_2, \ldots, \boldsymbol{T}_k) \end{aligned} \tag{4}$$

The PH distribution is an effective fitting tool in matching first and second moments. In particular, the PH moment matching technique is always able to return a canonical form $(\boldsymbol{\alpha}, \boldsymbol{T})$ with the same mean and variance. The interested readers are referred to [38] for details.

4 Mixture Density Networks

A Mixture density network (MDN) combines a mixture model and a fully-connected neural network. We start from introducing the mixture model—a weighted sum of N individual probability density functions

$$f(y) = \sum_{i=1}^{N} \pi_i p_i(y|\theta_i) \tag{5}$$

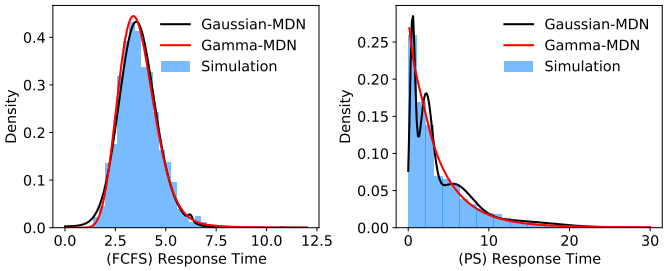

Fig. 2. A comparison between simulation and estimated conditional density functions of response time given the submodel parameters: N = 90, Z = 3, D = 0.6, c^2 = 1.25, m = 8 with (a) FCFS and (b) PS service discipline.

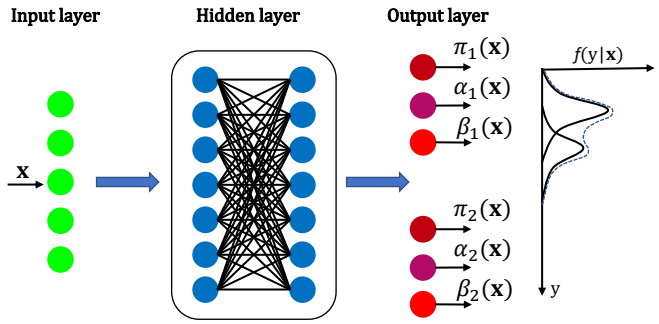

Fig. 3. The structure of a Gamma-MDN.

where π_i are mixing coefficients such that $0 \leq \pi_i \leq 1$ and $\sum_i \pi_i = 1$, and θ_i are parameters that characterize N kernel functions. Mixture model has the potential to express arbitrary probability distributions [15], thus we can use it to model various response time distributions in QN models.

Given the LQN submodel parameter set as an input vector $\mathbf{x}$, we wish to obtain its response time distribution expressed by a mixture model. In other words, our goal is to build a conditional density function of y given $\mathbf{x}$

$$f(y|\mathbf{x}) = \sum_{i=1}^{N} \pi_i(\mathbf{x}) p_i(y|\theta_i(\mathbf{x})) \tag{6}$$

In this case, the weight and density parameters become functions of $\mathbf{x}$: $\pi_i(\mathbf{x})$ and $\theta_i(\mathbf{x})$. The input $\mathbf{x}$ determines a unique mixture model through these two functions. We therefore need to find the functional relationships $\pi_i(\cdot)$ and $\theta_i(\cdot)$ respectively. The central idea of MDNs is to model both functions by a fully-connected neural network. Let O denote the number of observed samples, a MDN is trained by the dataset $\{(\mathbf{x}_j, y_j) \mid 1 \leq j \leq O\}$ and learns the weights of the neural network through maximizing the likelihood

$$L = \prod_{j=1}^{O} f(y_j|\mathbf{x}_j) \tag{7}$$

In MDNs, there are different types of kernel functions that can be used. The majority of applications choose Gaussian kernels as mixture components [17], [18], [19], [20], [21], a few applications have ever used other kernels such as Gamma or Binomial [16]. In this paper, we use both Gaussian and Gamma kernels to predict response time distributions in submodels. We find that a MDN with Gamma kernels is more applicable to parameterizing the PS scheduling density model. For instance, Fig. 2 shows a comparison between one histogram and two density functions obtained from simulation and five-component MDNs respectively. As can be seen from Fig. 2, both MDNs provide accurate predictions for the FCFS case, but for the PS case, the MDN with Gamma kernels provides a more accurate approximation.

The Gamma kernel γ is a two-parameter (θ=$\{\alpha, \beta\}$) probability density function with $\alpha > 0$, $\beta > 0$. Therefore, a Gamma-MDN needs to model three functional relationships $\alpha_i(\cdot)$, $\beta_i(\cdot)$, and $\pi_i(\cdot)$ to determine the conditional density function of variable of interest

$$f(y|\mathbf{x}) = \sum_{i=1}^{N} \pi_i(\mathbf{x}) \gamma_i(y|\alpha_i(\mathbf{x}), \beta_i(\mathbf{x})) \tag{8}$$

Fig. 3 shows the structure of the Gamma-MDN. The output layer consists of three types of nodes that return the function values $\pi_i(\mathbf{x})$, $\alpha_i(\mathbf{x})$, and $\beta_i(\mathbf{x})$. The first type employs softmax as an activation function to turn real values into mixing coefficients that sum to one. For the second and third types, the returned values should be always positive. We employ the following activation function [40] to ensure this condition

$$g(x) = 1 + ELU(\xi, x) \tag{9}$$

where $ELU(\xi, x)$ is Exponential Linear Unit proposed by [41], and ξ is a positive hyperparameter

$$ELU(\xi, x) = \begin{cases} \xi\,(e^x - 1) & x < 0 \\ x & x \geq 0 \end{cases} \tag{10}$$

This activation function is more stable compared to exponential activation function because the growth is not fast on the positive side. This helps to avoid instability during the training process [40]. The training makes the neural network learn its weights through minimizing our objective loss function defined as the negative log-likelihood.

5 Methodology

This section proposes a scheme to estimate densities of response times in layered queueing networks. Neural network analyzers are the basic components of our scheme. We first study SRVN submodels with one and two job classes because later we will reduce submodels with more than two classes to the two-class case. To illustrate the principle, four MDNs are trained by four distinct datasets: I. One class with FCFS; II. One class with PS; III. Two classes with FCFS; IV. Two classes with PS. Therefore, we set the input vector as $\mathbf{x} = (N, Z, D, c^2, m)$ for I, II; and $\mathbf{x} = (N_1, Z_1, D_1, c_1^2, N_2, Z_2, D_2, c_2^2, m)$ for III, IV. When applying our methodology to a given LQN, we build datasets by simulating randomly generated pairs in which the values of parameters N, Z, D, c^2, m fall within the valid parameter ranges of the potential problems we are trying to solve, and train embedded MDN analyzers. In this way, our approach possesses the generalization capabilities to analyze response times in LQN models. The bridge between MDN submodel predictions and LQN response times is an end-to-end analytic framework based on PH distributions

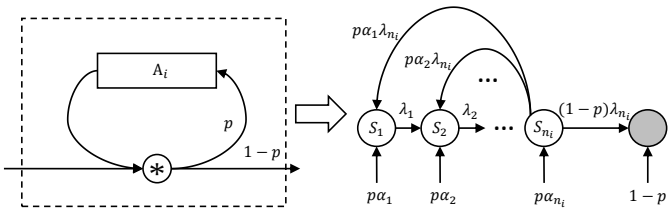

Fig. 4. Underlying Markov chain of the loop structure.

that keeps processing probability distributions within and between layers and interacting with MDNs, recursively propagates densities and transforms MDN output into next round input. The detailed mechanism will be illustrated in the section.

5.1 Host Demands of Activity Graphs

In LQN models, a single activity is the atomic operation. More complex operation patterns can be described by activity graphs. The host demand of each activity is normally characterized by its mean and variance. Our goal is to characterize host demands of arbitrary activity graphs composed of primary software logic structures: sequence, selection, and loop. For this problem, there are two aggregation methods in the existing literature. One work keeps aggregating unit structures until obtaining a single activity that can represent the original activity graph [10, p. 183-186]; during this process, mean and variance are computed based on summation formulas in [42]. However, algorithms to process graphs and detect unit structures are not discussed. The other work finds the weight of each activity in the graph and calculates the weighted sum of host demands [43, p. 31-33], but this cannot be applied to the variance aggregation.

Here, we present our procedure that can automatically characterize host demands of activity graphs using PH distributions. Compared to [10], we replace numerical operations with matrix ones which perform aggregations in a tractable form and have the potential to aggregate more than two moments of host demands.

5.1.1 Graph Processing

As can be observed in Fig. 1, an activity graph consists of nodes and directed edges. Nodes include decision, merge, loop, and activity nodes. Edges can be classified into two types: one indicates the routing probability from one node to another, the other indicates the average repeat count (cnt) of the activity in a loop. We model the loop structure by a probabilistic routing shown on the left side of Fig. 4 where count is transformed into routing probability by $p = cnt/(cnt + 1)$. As a result, a routing probability matrix for the whole activity graph can be built. Since the routing matrix is sparse, we extract non-zero elements from it and build a $m \times 3$ matrix $\mathbb{M}$ where m is the number of non-zero elements, three columns represent source node, target node, and routing probability respectively. We need to detect all primary structures from $\mathbb{M}$.

An activity graph may have nested structures, e.g., a sequence nested within a selection, which contributes to varied operation patterns. We detect primary sequences, selections, and loops that do not nest any structures. The sequence detector starts from building a new matrix $\mathbb{A}$ from $\mathbb{M}$ by extracting rows where the source and target nodes are activities. The $\mathbb{A}$ is a matrix with pure activity nodes, it allows us to figure out the number of sequences in the graph because the initial/final activity node of each sequence only appear once in the matrix. For the detection of every single sequence, the detector builds an initial sequence $[Initial, Final]$ using the first row of $\mathbb{A}$ and explores the rest of the rows to expand the sequence by continuously updating $Initial$ and $Final$ nodes. After detecting one single sequence, we delete all associated rows from $\mathbb{A}$ and initialize another $[Initial, Final]$ pair. This is executed repeatedly so that all sequences are detected in turn. The branch detector locates each decision node and further checks whether the destinations of all target nodes are the same merge node. In this way, every primary selections can be detected. The loop detector recognizes every activity that needs to be executed repeatedly by locating the corresponding loop node.

5.1.2 Updating Host Demands by Matrix Operations

Given an activity graph, we fit the host demand of each activity with the canonical form of acyclic PH distributions. For an activity A_i, suppose the canonical representation $(\boldsymbol{\alpha}_i, \boldsymbol{T}_i)$ of its host demand has n_i transient states, then

$$\boldsymbol{\alpha}_i = (\alpha_1, \alpha_2, \ldots, \alpha_{n_i}),$$
$$\boldsymbol{T}_i = \begin{pmatrix} -\lambda_1 & \lambda_1 & \cdots & 0 \\ 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -\lambda_{n_i} \end{pmatrix} \tag{11}$$

Next, we find the PH expression for each of detected structures by a corresponding matrix operation. The convolution (3) is performed $k-1$ times for a sequence of k activities. The mixture operation (4) is performed for a selection with k branches. As shown in Fig. 4, we construct a CTMC for the loop structure so that we derive its PH expression as follows

$$\boldsymbol{\alpha} = p\boldsymbol{\alpha}_i,$$
$$\boldsymbol{T} = \begin{pmatrix} -\lambda_1 & \lambda_1 & \cdots & 0 \\ 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ p\alpha_1\lambda_{n_i} & p\alpha_2\lambda_{n_i} & \cdots & -\lambda_{n_i} + p\alpha_{n_i}\lambda_{n_i} \end{pmatrix} \tag{12}$$

where $\boldsymbol{T}$ can be summarized as $\boldsymbol{T} = \boldsymbol{T}_i + p(-\boldsymbol{T}_i\mathbf{1}\boldsymbol{\alpha}_i)$. After matrix operations, every detected structure is removed from $\mathbb{M}$ and replaced by a single activity whose host demand is a canonical representation obtained by matching the first two moments of the matrix operation result. In this way the activity graph saved in $\mathbb{M}$ is updated. Then, we repeat this process until we obtain the final host demand PH expression for the whole activity graph.

5.2 Main Algorithm

5.2.1 Internal Mechanism of the Hybrid Scheme

As mentioned earlier, we use SRVN-layering to build submodels. Each SRVN submodel is an ordinary queueing network with only one resource. Therefore, the submodel of layered queueing networks linked via synchronous calls

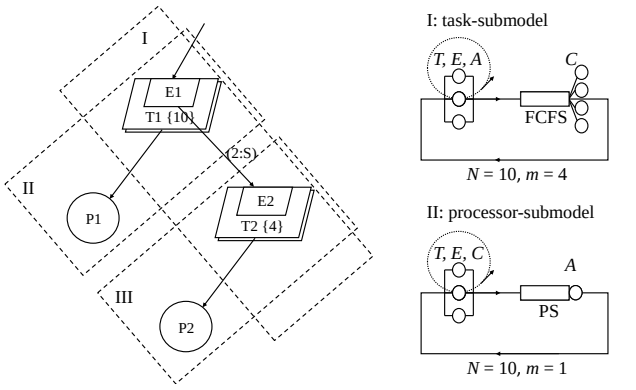

Fig. 5. Task-submodel and Processor-submodel.

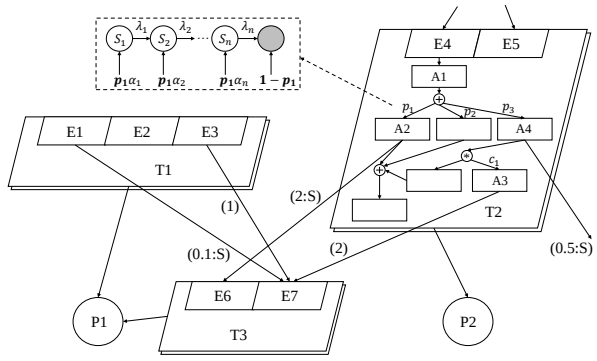

Fig. 6. Requesting distinct services from lower servers

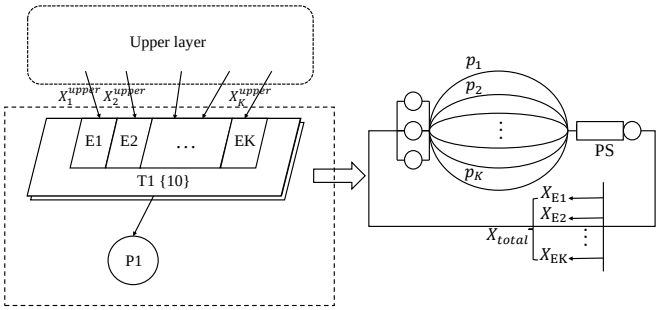

Fig. 7. A submodel with multiple entries.

is a closed queueing network consisting of a delay and a queueing station. The queueing station is either a hardware or a software server. According to the type of queueing stations, we classify submodels into two categories, (a) $processor\text{-}submodel$ and (b) $task\text{-}submodel$. For example, as shown in Fig. 5, submodels II and III belong to (a) while the submodel I belongs to (b). Our method starts from analyzing processor-submodels (Algorithm 1), then analyzes task-submodels and passes densities layer by layer, from bottom to top (Algorithm 2). We give each submodel four main job classes: $Task(T)$, $Entry(E)$, $Activity(A)$, and $Call(C)$; the class switching happens between each other within the closed network. In a layered queueing network, the activities of each task are executed on the host processor. For this reason, in a processor-submodel we let queueing station serve A-class jobs and delay station serve the rest of job classes. Meanwhile, a task may send nested requests to other software servers. Therefore, in a task-submodel we let queueing station serve C-class jobs and delay station serve the rest. In Fig. 5, the switching circle for processor-submodel II is $(Delay, T) \rightarrow (Delay, E) \rightarrow (Queueing, A) \rightarrow (Delay, C) \rightarrow (Delay, T)$, while the switching circle for task-submodel I is $(Delay, T) \rightarrow (Delay, E) \rightarrow (Delay, A) \rightarrow (Queueing, C) \rightarrow (Delay, T)$.

First, we apply the iterative algorithm in [10] to compute mean performance measures for each submodel and use the technique introduced in Section 3.2 to further obtain performance measures for each of our job classes (lines 1-3 in Algorithm 1). In this step we get think times of classes T, A, and C named Z_T, Z_A, and Z_C. The think times of processor-submodels and task-submodels are $Z_T + Z_C$ and $Z_T + Z_C + Z_A$ respectively. In a task-submodel, Z_C is zero if the client sends no service requests outside this submodel. Then, we employ the method presented in Section 5.1 to analyze all activity graphs in the layered queueing network and calculate the aggregated host demand for each entry (lines 8-11 in Algorithm 1). Once this is done, we build the input vector $\mathbf{x}$. We feed $\mathbf{x}$ to a MDN analyzer and get a mixture distribution with mean and variance as follows

$$\mu = \sum_{i=1}^{N} \pi_i \mu_i, \quad \sigma^2 = \sum_{i=1}^{N} \pi_i \Big(\sigma_i^2 + \mu_i^2 - \mu^2 \Big) \tag{13}$$

where π_i, μ_i, and σ_i^2 are kernel parameters returned by this MDN. We therefore can compute its mean and variance by (13) and fit it with the PH distribution (lines 13-22 in Algorithm 1). Next, we begin analyzing $bottom\ task\text{-}submodels$ (line 4 in Algorithm 2). We define this term as a task-submodel in which all services provided by the software server have a known response time distribution. For instance, in Fig. 5 the submodel I becomes a bottom task-submodel after analyzing the processor-submodel III because the software server provides E2 service in submodel I and the response time distribution of E2 is given by a MDN in the analysis of submodel III.

The acquisition of the service demand (lines 8-13 in Algorithm 2) will be discussed in details in next Section 5.2.2, it enables us to form the input vector $\mathbf{x}$ in task-submodels. We feed $\mathbf{x}$ to a MDN analyzer to forecast the response time of a client entry in the bottom task-submodel and fit a PH distribution to this prediction (lines 16-22 in Algorithm 2). One client entry may have several software servers that execute its requests in distinct task-submodels. For example, in addition to the software server T3 in Fig. 6, we can observe that there must be another sever that executes the requests sent by E4 (A4), and we need to analyze this behaviour in a new bottom task-submodel. Once we complete such analysis for each of the associated submodels, we convolve all of the fitted response time distributions that are related to its host processor and software servers. In the end, we get the response time of the targeted client entry (lines 23-26 in Algorithm 2). We repeat this procedure until we obtain end-to-end response time distributions for all client entries in reference tasks.

5.2.2 Modeling Service Requests Between Layers

The first step to analyze a bottom task-submodel is to compute the service demand of each client entry. In a bottom task-submodel, let $(\boldsymbol{\alpha}_{se}, \boldsymbol{T}_{se})$ and Φ represent the response time of a server and the number of requests made by a client entry to this server, we can calculate the contribution of this request to the service demand of the client entry using PH matrix operations according to the request distribution. If the distribution is deterministic, the client entry makes exact Φ requests thus we convolve Φ response time representations $(\boldsymbol{\alpha}_{se}, \boldsymbol{T}_{se})$ by a repeated execution of (3). If the distribution is stochastic, Φ is a geometrically distributed random variable with mean $\mu_\Phi = (1 - p_\Phi)/p_\Phi$, where p_Φ is the probability of sending no further request. We model this process by a CTMC which has the same structure as that in Fig. 4 but with the following PH expression

$$\boldsymbol{\alpha} = (1 - p_\Phi)\boldsymbol{\alpha}_{se}, \quad \boldsymbol{T} = \boldsymbol{T}_{se} + (1 - p_\Phi)(-\boldsymbol{T}_{se}\mathbf{1}\boldsymbol{\alpha}_{se}) \quad (14)$$

The $(\boldsymbol{\alpha}, \boldsymbol{T})$ from (3) or (14) is reduced into the canonical form. This is the intermediate result and we denote it by Λ.

For each service request r_i, we need to consider its weight w_i that is defined as the average number of visits to the parent activity per visit to the parent entry. This consideration is necessary because the actual number of requests is related directly to the execution times of the parent activity. Let $(\boldsymbol{\alpha}_i, \boldsymbol{T}_i)$ represent the intermediate result Λ, the PH representation of the weighting process can be derived as follows

$$\boldsymbol{\alpha} = w_i\boldsymbol{\alpha}_i, \quad \boldsymbol{T} = \boldsymbol{T}_i \quad (15)$$

This is because every client will enter the transient states to hold time with probability w_i. However, (15) cannot be used directly if the parent activity of the service request is in a loop because the weight is partially determined by repeat count, which makes the underlying CTMC contain cycles and thus no longer equivalent to the PH representation above. In this case, we apply (12) to $(\boldsymbol{\alpha}_i, \boldsymbol{T}_i)$ first, then reduce it into the canonical form $\mathbb{C}$, and finally use (15) in which the weight is updated as $w_i = w_i/count$. This forms the following equation

$$(\boldsymbol{\alpha}, \boldsymbol{T}) = \begin{cases} \Lambda \rightarrow (12) \rightarrow \mathbb{C} \rightarrow (15) & a_p \in loop^a \\ \Lambda \rightarrow (15) & a_p \notin loop^a \end{cases} \quad (16)$$

where a_p is the parent activity of a service request, $loop^a$ is the set of activities in loops.

In a task-submodel, a client entry may require multiple services from the software server. For instance, as shown in Fig. 6, the entry E4 sends requests to two services E6 and E7 in the task-submodel (T1,T2)→T3. Suppose response times of E6 and E7 are known, (14) and (3) give us intermediate results of the service requests A2→E6 and A3→E7 respectively. As can be observed, the weight of A2→E6 is p_1, so the intermediate CTMC is directly updated by modifying the initial probability distribution over the set of states. In terms of the request A3→E7, the parent activity A3 is in a loop and the weight is $p_3 \times c_1$, thus we build its CTMC by a two-step way: (a) update the intermediate CTMC to the form shown in Fig. 4 with $p = c_1/(c_1+1)$ and reduce it into the canonical form (b) modify the initial probabilities using p_3. Once the weighting process is completed, we are able to compute the service demand of the client entry by convolving all PH distributions of associated service requests.

TABLE 2
Main Notations of Algorithms

s	submodel
c	chain
e	entry
t_c	client task in c
r	service request
$\varepsilon(t_c)$	the set of client entries in t_c
$\varepsilon(s)$	the set of client entries in s
$Z_T(t_c)$	mean idle time spent by t_c from response until next invocation
$Z_C(t_c)$	mean delay spent by t_c acquiring services from other software servers
$Z_A(t_c)$	mean delay that t_c spends on its host processor
$\mathcal{H}$	host demand PH representation of an activity graph
Θ	mean and variance of a probability distribution
$\mathcal{R}_e$	response time distribution of e
$\mathcal{D}_e$	the set of probability distributions that determine $\mathcal{R}_e$
$\mathcal{E}_e$	the set of submodels that execute requests sent by e
$\Omega_{e\rightarrow s}$	the set of requests sent by e and executed in s
Δ	service time distribution in a task-submodel
$\mathcal{P}$	the set of processor-submodels in a LQN
$\mathcal{T}$	the set of task-submodels in a LQN
$\mathcal{B}$	the set of bottom task-submodels at present
$\mathcal{C}$	the set of chains in s
Ψ	service discipline (FCFS, PS)
$\lvert\cdot\rvert$	number of elements in a set
$x \leftarrow a$	assign a to x
$x \xrightarrow{f} y$	map x onto y by f

5.3 Class Aggregation

A submodel may include an arbitrary number of chains. In that case, we employ a two-class MDN to make predictions for each chain. Suppose there are $n+1$ chains in a submodel and we want to compute the response time in chain $n+1$, we need to aggregate the first n chains so that a two-class MDN can be used. We use the following aggregation method inspired by the utilization law

$$N_{aggr} = \sum_{i=1}^{n} N_i, \quad M_{aggr} = \sum_{i=1}^{n} X_i M_i \Big/ \sum_{i=1}^{n} X_i \quad (17)$$

where N_i and X_i are the population and throughput of the i-th chain, M_i can be Z, D, or c^2 that is defined in Table 1.

Because we are interested in response times of client entries, we build the submodel in which each chain contains only one client entry. This enables us to employ a MDN to predict the response time distribution for the client entry in each chain. For example, as shown in Fig. 6, there should be three chains in the task-submodel $(\mathrm{T}_1, \mathrm{T}_2)\rightarrow\mathrm{T}_3$ because of the three client entries E1, E3, and E4. Likewise, the processor-submodel (T1,T3)→P1 incorporates five chains represented by E1, E2, E3, E6, and E7 respectively. In this way, we can do class aggregation in terms of client entry and focus on entry-level predictions.

Here, we illustrate the way of obtaining an independent chain for each client entry. Let us suppose that the task in a submodel has multiple client entries, we model the task and its server by an ordinary queueing network where a T-class job switches to each of the E-class jobs with a probability determined by the throughput of the entry. For

Algorithm 1

Input: A LQN model
Output: $\mathcal{D}_e$ for each e

construct $\mathcal{P}$ and $\mathcal{T}$, create empty $\mathcal{D}_e$ for every e
derive the parameters of c for each s
iterate to solve LQN, obtain performance measures per chain in s and then per class in each c
for each $s \in \mathcal{P}$ **do**
  for each $c \in \mathcal{C}$ **do**
    $Z = Z_T(t_c) + Z_C(t_c)$
    split c into $|\varepsilon(t_c)|$ chains and determine N of each chain by (18)
    for each $e \in \varepsilon(t_c)$ **do**
      analyze the activity graph to get $\mathcal{H}$
      $\mathcal{H} \xrightarrow{(1)(2)} \Theta$
    end for
  end for
  for each $e \in \varepsilon(s)$ **do**
    if $|\varepsilon(s)| > 1$ **then**
      aggregate chains by (17)
      two-class input $\mathbf{x}$ with Ψ(PS) $\xrightarrow{MDN^{\text{IV}}} \mathcal{W}$
    else
      one-class input $\mathbf{x}$ with Ψ(PS) $\xrightarrow{MDN^{\text{II}}} \mathcal{W}$
    end if
    $\mathcal{W} \xrightarrow{(13)} \Theta \xrightarrow{fitPH} \mathbb{W}$
    put $\mathbb{W}$ into $\mathcal{D}_e$
  end for
end for

instance, as shown in Fig. 7, the task T1 with K entries receives service requests from the upper layer and executes them on its host processor. This task acts as a client in the processor-submodel T1→P1 shown within the dashed rectangle, thus the job class T switches to job classes (i.e., entries) E1, E2,...,EK with probabilities $p_1, p_2, \ldots, p_K$ respectively, where $p_k = X_k^{upper}/X_{total}^{upper}$. We compute mean performance measures for such a model. After this, as the MDN prediction requires constant multiclass populations as an input, we split the main chain into K chains by applying the following transformation to the model

$$N_k = \left(\frac{X_{E_k}}{X_{total}}\right) N_T + N_{E_k} \tag{18}$$

where N_k is the population of the k-th chain, X_{total}, X_{E_k}, N_T and N_{E_k} are the performance measures obtained via analytic computation.

6 Evaluation

6.1 MDN submodel analyzer

In our framework, the key to estimate response times in a layered queueing network is to obtain response time distributions in each of the closed queueing submodels. Therefore, we first evaluate if MDNs are capable of providing accurate predictions for unknown response time distributions. We generate training and test pairs falling within the ranges shown in Table 1 and employ the tool JMT 1.2.1 [44] via LINE [45] to simulate the pairs under

Algorithm 2

Input: Output of Algorithm 1
Output: $\mathcal{R}_e$ of LQN

$Done \leftarrow \emptyset$
while $|Done| < |\mathcal{T}|$ **do**
  for each $s \in \mathcal{T} \setminus Done$ **do**
    if $s \in \mathcal{B}$ **then**
      for each $c \in \mathcal{C}$ **do**
        $Z = Z_T(t_c) + Z_C(t_c) + Z_A(t_c)$
        implement line 7 of Algorithm 1
        for each $e \in \varepsilon(t_c)$ **do**
          for each $r \in \Omega_{e \to s}$ **do**
            $\mathbb{W} \xrightarrow{(3)\ or\ (14)} \Lambda \xrightarrow{(16)} \Delta \xrightarrow{(1)(2)} \Theta$
          end for
          convolve all Θ to get the final Δ
        end for
      end for
      for each $e \in \varepsilon(s)$ **do**
        if $|\mathcal{C}| > 1$ **then**
          aggregate chains by (17)
          two-class input $\mathbf{x}$ with Ψ(FCFS) $\xrightarrow{MDN^{\text{III}}} \mathcal{W}$
        else
          one-class input $\mathbf{x}$ with Ψ(FCFS) $\xrightarrow{MDN^{\text{I}}} \mathcal{W}$
        end if
        implement lines 20-21 of Algorithm 1
        if $|\mathcal{D}_e| = |\mathcal{E}_e| + 1$ **then**
          convolve $\mathcal{D}_e$ by repeated execution of (3)
          $\mathcal{R}_e \leftarrow$ convolution result
        end if
      end for
      put s into $Done$
    end if
  end for
end while

FCFS and PS discipline respectively. For each pair, we tag a particular job and measure its response time at the queueing station. Because the network is closed, the measurements can be made constantly so that we obtain a steady response time distribution. Each training set has 20000 pairs and evaluations are conducted on the corresponding test set which consists of 5000 unseen pairs.

We consider two performance metrics: *mean absolute percentage error* (MAPE) and *mean KS distance* (MKS). The first metric is defined as

$$\text{MAPE} = \frac{100\%}{N_{test}} \sum_{n=1}^{N_{test}} \left| \frac{P_n - G_n}{G_n} \right| \tag{19}$$

where P and G are the predictions and ground-truth values, N_{test} is the number of test pairs. We calculate MAPE for mean, standard deviation, and 95th percentile of response times respectively.

KS distance is defined as the maximum vertical distance between two cumulative density functions (CDFs), thus our second metric is

$$\text{MKS} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} \max_x \left| F_n^G(x) - F_n^P(x) \right| \tag{20}$$

TABLE 3
Neural Network Hyperparameters

Hyperparameter	Values
Number of hidden layers	3
Hidden layer neurons	64
Activation function	ReLU
Number of components	3, 5, 7
Kernel type	Gaussian, Gamma
Optimizer	RMSprop
Learning rate α	0.001
Decay factor of α	50% per 10 epochs

TABLE 4
The MAPE and MKS results of MDNs

Analyzer	Scenario	MAPE			MKS
		Mean	Std	95-th	
MDN I	FCFS, 1 class	3.0%	4.4%	3.2%	0.036
MDN II	PS, 1 class	4.6%	5.6%	5.2%	0.023
MDN III	FCFS, 2 classes	3.6%	5.5%	3.5%	0.055
MDN IV	PS, 2 classes	5.0%	5.8%	5.5%	0.031

TABLE 5
Case Study Parameters

Parameter	Range of Value
System population	20-100
Think time	3-10
Multiplicity	1-8, ∞
Activity host demand mean	0.05-0.1
Activity host demand SCV	0.4-1.2
Number of calls	0.1-2
Loop counts	0.1-2.5

where $F^G(x)$ is the empirical CDF from observed samples and $F^P(x)$ is the CDF of a mixture model from prediction.

We use the neural network structure depicted in Table 3, and only tune the number of components and kernel type. Therefore, for each of the four analyzers I-IV, we train neural networks with six settings and select the one with the lowest validation loss. All selected analyzers are seven-component MDNs, the analyzer I, II, and IV are with Gamma kernels while the analyzer III adopts Gaussian kernels. Then, our analyzers are evaluated on test sets and the results are shown in Table 4. As can be observed, all MAPE values are below 6%, the trained MDNs show an overall good performance. Besides the evaluations above, we can visualize the prediction performance via different types of plots. Fig. 8a shows an interesting phenomenon that the vast majority of KS distances are below 0.10, which means MDNs get very close to the ground-truth response time distributions. Meanwhile, we can observe that PS cases are simpler for neural networks to approximate with regard to the KS distance. Fig. 8b gives a comparison between simulations and sampling distributions from MDN predictions. In this experiment, we simulate a pair and obtain 1758 and 1781 response time samples from FCFS and PS queue respectively, then we employ MDN I and II to make predictions and generate the same number of samples from the returned Gamma mixture distributions. As can be seen from this figure, estimated distributions capture the similar symmetric-like and right-skewed trend, the trained MDNs cope well with emulating unknown distributions. The next experiment is to compare mean, variance, and percentiles

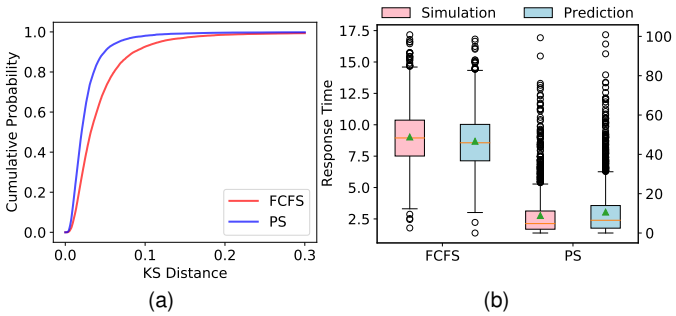

Fig. 8. (a): The CDFs of the KS distances between simulations and predictions for all FCFS and PS test pairs. (b): Summary statistics (the orange line and green triangle inside box indicate the median and mean value) on the response time given $\mathbf{x}$ = (85, 8, 0.4, 1.5, 2). The left and right y-axis represent the response time under FCFS and PS respectively.

between simulations and predictions through varying the values of distinct input variables. Fig. 12a to 12f show a "what if" instance where we can observe that our MDNs give acceptable approximations over varying values of population, number of servers, think time, and host demand mean. This means MDNs are capable of discriminating between the different variables and calculating how each type influences the response time behaviour. Finally, we visualize the prediction performance in a multiclass submodel with FCFS service discipline and different service rates. This is the situation where conventional BCMP theorem cannot be used [4]. Fig. 12g to 12h display two predictions which are below the average level. Their KS distances are 0.07 and 0.12, around 1.3 and 2.2 times longer than the MKS value, 0.055, but the overall trends of predictions are still reasonable.

6.2 Analytic Framework Based on PH Distributions

In previous section, we have illustrated how the proposed analytic framework analyzes activity graphs, models connections between layers, and propagates densities across LQNs based on PH distributions. Combined with MDN analyzers, it can estimate the response time densities in layered models effectively. We conduct evaluations on two cases: E-commerce [46] and interconnected network. For each case, 500 model instances are randomly generated using the parameters in Table 5. The end-to-end response times are estimated by our framework and LQNS 6.2 [14], [26], the de facto standard analytic tool for LQN models. The accuracy of estimations is measured by means of absolute percentage error with respect to the simulation results obtained by LQSIM tool [26]. Both our framework and LQNS can provide accurate estimations of mean response times, their absolute percentage errors are less than 10%. In terms of density estimations, the proposed framework shows a huge improvement over the state-of-the-art.

An instance in Fig. 9 displays the topology of E-commerce system. Users are classified into three categories according to the behaviour in $CustomerProcess$ task. The activity graphs (a), (b), and (c) shown in Fig. 10 are assigned to the entry $browse$, $cart$, and $placeOrder$ respectively, service requests are sent by the dashed activities (A3→catInfo, A4→update, A6→delivOrder, A9→custInfo, A13→cartInfo,

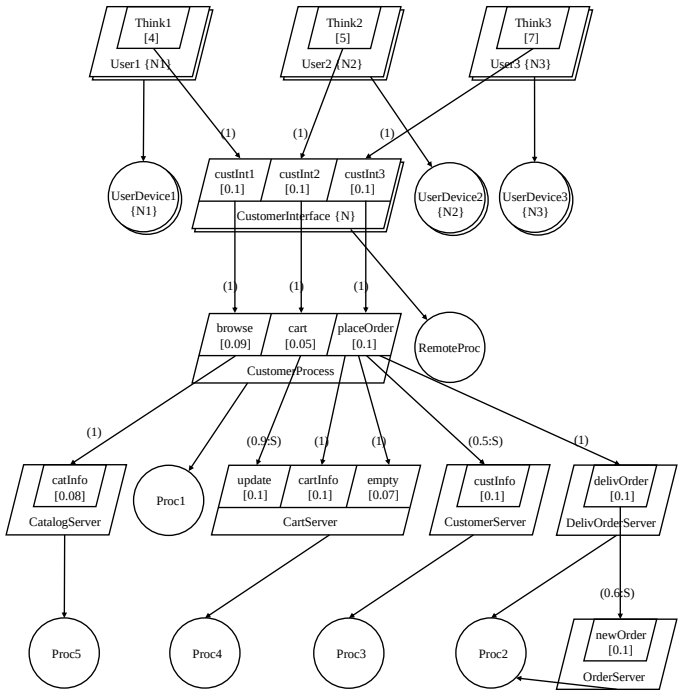

Fig. 9. LQN model of an E-commerce system.

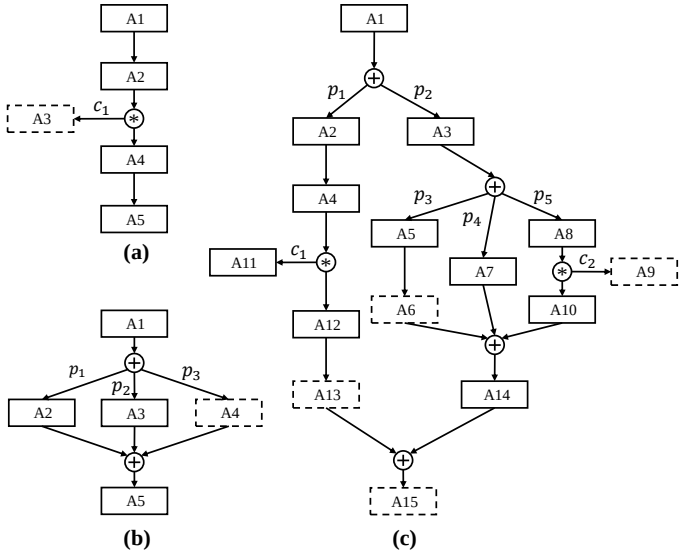

Fig. 10. Three activity graphs used in the case study of Fig. 9.

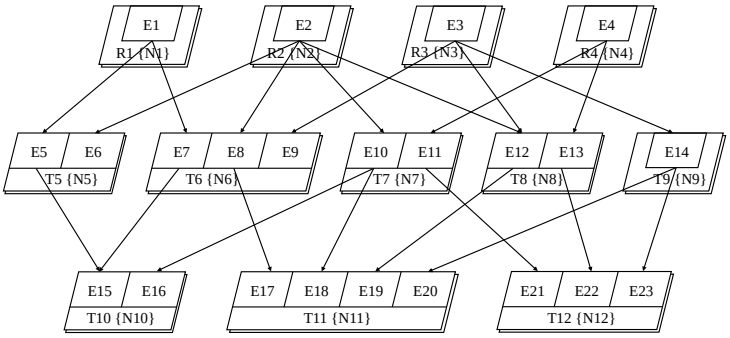

Fig. 11. LQN model of an interconnected system.

A15→empty). Table 6 reports absolute percentage errors of standard deviation (σ) estimates across 500 model instances by the median and average values as well as the percentiles. For an organized presentation, we divide all model instances into four groups with different ranges of system population. As can be observed, the reduction of average error by our framework is between 85% and 95% compared with LQNS. Our median absolute percentage errors are at most 0.07 while the median errors given by LQNS reach around 0.90. The percentile statistics shows that error distributions of our framework are positively skewed, but those of LQNS are negatively skewed. The numerical results indicate that the proposed framework is effective and significantly outperforms LQNS in estimating response time densities in models incorporating activity graphs.

Next, we apply our framework to analyzing an interconnected network shown in Fig. 11. To help the visualization, we omit processor notations. There are two shared processors, one is for R1 and R2, the other is for R3 and R4. The remaining tasks have their own host processors. The main feature of this network is multichain interconnection, which requires estimating response time densities in a set of multiclass queues. Fig. 13 compares the prediction performance of our framework and the state-of-the-art. As observed in Fig. 13a and Fig. 13b, our framework can provide accurate density estimations in multiclass LQN models, the error reduction is over 75% compared to LQNS. Then, we visualize the prediction performance on distinct population groups in Fig. 13c and Fig. 13d. As the system population gets larger, the absolute percentage errors given by LQNS show a quick increase. In contrast, the performance of our framework is unlikely to have a sharp change, we can observe that errors remain on a stable level.

TABLE 6
Summary statistics on absolute percentage errors of σ estimates across 500 model instances with topology as in Fig. 9

System Population		5th	Med	95th	99th	Avg
20-40	lqns	0.143	0.881	0.933	0.945	0.763
(112 instances)	mdn	0.006	0.070	0.308	0.680	0.105
40-60	lqns	0.631	0.905	0.944	0.956	0.878
(140 instances)	mdn	0.004	0.041	0.207	0.541	0.066
60-80	lqns	0.872	0.915	0.952	0.978	0.892
(121 instances)	mdn	0.005	0.049	0.254	0.567	0.080
80-100	lqns	0.840	0.923	0.962	0.983	0.897
(127 instances)	mdn	0.004	0.060	0.417	0.639	0.104

7 Conclusion

In this paper, we propose a novel scheme to provide accurate density estimations of response times in layered software systems. The approach estimates response time densities by a combined use of neural network analyzers and an end-to-end analytic framework based on PH distributions. Extensive numerical results demonstrate that our scheme improves significantly in the accuracy of density estimations

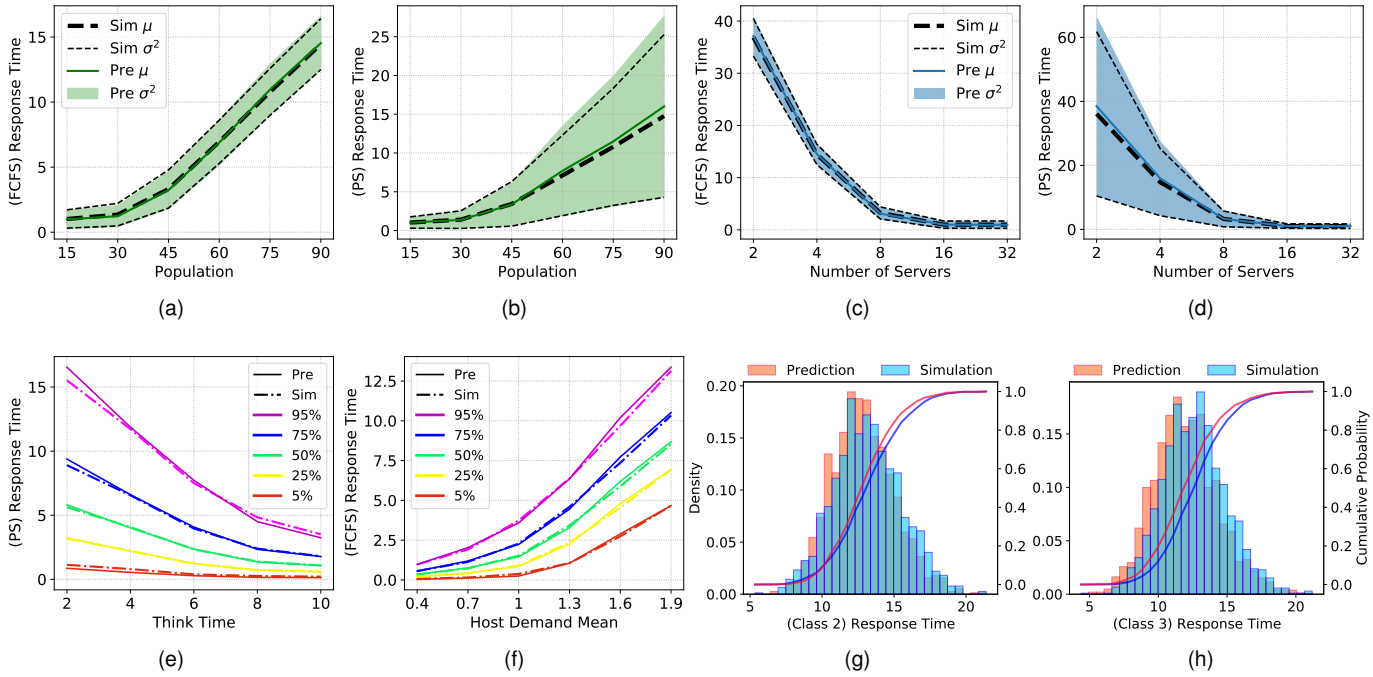

Fig. 12. (a) to (d): Starting with $\mathbf{x}$ = (90, 8, 1, 0.5, 4), varying population and number of servers, comparing the mean and variance between simulations and predictions. (e) to (f): Starting with $\mathbf{x}$ = (35, 8, 1, 0.5, 4), varying think time and host demand mean, comparing the percentile lines between simulations and predictions. (g) to (h): Analyzing a submodel with four job classes and FCFS scheduling: N = (15, 25, 30, 20), Z = (3, 10, 9, 5), D = (0.7, 0.8, 0.1, 0.3), c^2 = (1, 0.6, 1, 1.4), m = 2.

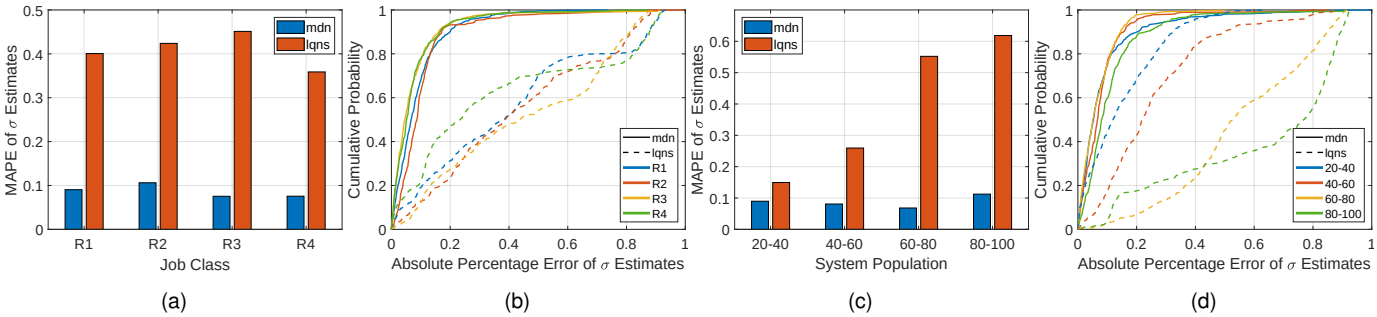

Fig. 13. Prediction performance visualization on absolute percentage errors of σ estimates across 500 model instances with topology as in Fig. 11.

compared to the state-of-the-art method. In future work we will explore the modeling of asynchronous calls and fork/join nodes in our hybrid analytic scheme for layered software systems.

REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.8

[2] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 171–187.

[3] P. G. Harrison and W. J. Knottenbelt, "Passage time distributions in large markov chains," in *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2002, pp. 77–85.

[4] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 248–260, 1975.

[5] Z. Niu and G. Casale, "A mixture density network approach to predicting response times in layered systems," in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2021, pp. 1–8.

[6] P. G. Harrison, "Response time distributions in queueing network models," in *Performance Evaluation of Computer and Communication Systems*. Springer, 1993, pp. 147–164.

[7] L. Zhu, G. Casale, and I. Perez, "Fluid approximation of closed queueing networks with discriminatory processor sharing," *Performance Evaluation*, vol. 139, p. 102094, 2020.

[8] J. Ruuskanen, T. Berner, K.-E. Årzén, and A. Cervin, "Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing," *Performance Evaluation*, vol. 151, p. 102231, 2021.

[9] T. G. Kurtz, "Solutions of ordinary differential equations as limits of pure jump markov processes," *Journal of applied Probability*, vol. 7, no. 1, pp. 49–58, 1970.

[10] G. Franks, "Performance analysis of distributed server systems." Ph.D. dissertation, Carleton University, 2000.

[11] T. Zheng and M. Woodside, "Fast estimation of probabilities of soft deadline misses in layered software performance models," in *Proceedings of the 5th International Workshop on Software and Performance*, 2005, pp. 181–186.

[12] T. Omari, S. Derisavi, and G. Franks, "Deriving distribution of thread service time in layered queueing networks," in *Proceedings of the 6th International Workshop on Software and Performance*, 2007, pp. 66–77.
[13] M. Reiser, "A queueing network analysis of computer communication networks with window flow control," *IEEE transactions on Communications*, vol. 27, no. 8, pp. 1199–1209, 1979.
[14] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2008.
[15] C. M. Bishop, "Mixture density networks," 1994.
[16] C. N. Davis, T. D. Hollingsworth, Q. Caudron, and M. A. Irvine, "The use of mixture density networks in the emulation of complex epidemiological individual-based models," *PLoS computational biology*, vol. 16, no. 3, p. e1006869, 2020.
[17] C. Li and G. H. Lee, "Generating multiple hypotheses for 3d human pose estimation with mixture density network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9887–9895.
[18] X. Wang, S. Takaki, and J. Yamagishi, "An autoregressive recurrent mixture density network for parametric speech synthesis," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 4895–4899.
[19] M. Raeis, A. Tizghadam, and A. Leon-Garcia, "Predicting distributions of waiting times in customer service systems using mixture density networks," in *2019 15th International Conference on Network and Service Management (CNSM)*. IEEE, 2019, pp. 1–6.
[20] S. S. Mostafavi, G. Dán, and J. Gross, "Data-driven end-to-end delay violation probability prediction with extreme value mixture models," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021, pp. 416–422
[21] H. Q. Nguyen and T. Phung-Duc, "Mixture density networks as a general framework for estimation and prediction of waiting time distributions in queueing systems," in *Performance Engineering and Stochastic Modeling*. Springer, 2021, pp. 148–161.
[22] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1994–2004.
[23] M. Woodside, "Performance models of event-driven architectures," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2021, pp. 145–149.
[24] D. C. Petriu, "Integrating the analysis of multiple non-functional properties in model-driven engineering," *Software and Systems Modeling*, vol. 20, no. 6, pp. 1777–1791, 2021.
[25] M. Woodside and G. Franks, "Tutorial introduction to layered modeling of software performance," *Carleton University, Ottawa, Canada*, 2002.
[26] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz, "Layered queueing network solver and simulator user manual," *Dept. of Systems and Computer Engineering, Carleton University (December 2005)*, pp. 15–69, 2005.
[27] M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, "The stochastic rendezvous network model for performance of synchronous client-server-like distributed software," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 20–34, 1995.
[28] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE transactions on software engineering*, vol. 21, no. 8, pp. 689–700, 1995.
[29] G. Franks, M. Woodside, and J. Rolia, "Multi-threaded servers with high service time variation for layered queueing networks," in *Computer System Performance Modeling In Perspective: A Tribute to the Work of Prof Kenneth C Sevcik*. World Scientific, 2006, pp. 137–154.
[30] M. Tribastone, "A fluid model for layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 744–756, 2012.
[31] K. M. Chandy and D. Neuse, "Linearizer: A heuristic algorithm for queueing network models of computing systems," *Communications of the ACM*, vol. 25, no. 2, pp. 126–134, 1982.
[32] S. C. Bruell, G. Balbo *et al.*, *Computational algorithms for closed queueing networks*. Elsevier North-Holland, 1980.
[33] M. Reiser and H. Kobayashi, "Queuing networks with multiple closed chains: theory and computational algorithms," *IBM journal of Research and Development*, vol. 19, no. 3, pp. 283–294, 1975.
[34] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queuing networks," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 313–322, 1980.
[35] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.
[36] M. F. Neuts, *Matrix-geometric solutions in stochastic models: an algorithmic approach*. Courier Corporation, 1994.
[37] A. Cumani, "On the canonical representation of homogeneous markov processes modelling failure-time distributions," *Microelectronics Reliability*, vol. 22, no. 3, pp. 583–602, 1982.
[38] A. Bobbio, A. Horváth, and M. Telek, "Matching three moments with minimal acyclic phase type distributions," *Stochastic models*, vol. 21, no. 2-3, pp. 303–326, 2005.
[39] R. Marie, "Calculating equilibrium probabilities for λ (n)/ck/1/n queues," in *Proceedings of the 1980 international symposium on computer performance modelling, measurement and evaluation*, 1980, pp. 117–125.
[40] A. Brando Guillaumes, "Mixture density networks for distribution and uncertainty estimation," Master's thesis, Universitat Politècnica de Catalunya, 2017.
[41] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.
[42] C. U. Smith, *Performance engineering of software systems*. Addison-Wesley Longman Publishing Co., Inc., 1990.
[43] F. Islam, "Simplifying layered queuing network models." Ph.D. dissertation, Carleton University, 2018.
[44] M. Bertoli, G. Casale, and G. Serazzi, "JMT: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 10–15, 2009.
[45] G. Casale, "Integrated performance evaluation of extended queueing network models with LINE," in *2020 Winter Simulation Conference (WSC)*. IEEE, 2020, pp. 2377–2388.
[46] V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-based software performance analysis*. Springer, 2011, vol. 980.

Zifeng Niu received the BSc (Hons) degree in Mathematics with Applied Mathematics from the University of Nottingham in 2019 and the MSc degree in Computing (Artificial Intelligence and Machine Learning) from Imperial College London in 2020. He is currently pursuing the PhD degree with the supervision of Giuliano Casale at the Department of Computing. His research interests focus on performance modeling, cloud computing, machine learning and operations research.

Giuliano Casale joined the Department of Computing at Imperial College London in 2010, where he is currently a Reader. Previously, he worked as a research scientist and consultant in the capacity planning industry. He teaches and does research in performance engineering and cloud computing, topics on which he has published more than 150 refereed papers. He has served on the technical program committee of several conferences in the area of performance and dependability. His research work has received multiple recognitions, recently the best paper award at ACM SIGMETRICS. He serves on the editorial boards of IEEE TNSM and ACM TOMPECS and as current chair of ACM SIGMETRICS.