ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Characterization of Microservice Response Time in Kubernetes: A Mixture Density Network Approach

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Manca, L., Borsatti, D., Poltronieri, F., Zaccarini, M., Scotece, D., Davoli, G., et al. (2023). Characterization of Microservice Response Time in Kubernetes: A Mixture Density Network Approach. IEEE [10.23919/CNSM59352.2023.10327842].

Availability:
This version is available at: https://hdl.handle.net/11585/961705 since: 2024-02-26

Published:
DOI: http://doi.org/10.23919/CNSM59352.2023.10327842

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

Characterization of Microservice Response Time in Kubernetes: A Mixture Density Network Approach

Lorenzo Manca*, Davide Borsatti*, Filippo Poltronieri‡, Mattia Zaccarini‡, Domenico Scotece*, Gianluca Davoli*, Luca Foschini*, Genady Ya. Grabarnik†, Larisa Shwartz§, Cesare Stefanelli‡, Mauro Tortonesi‡, Walter Cerroni*
* University of Bologna, Bologna, Italy
Email: {lorenzo.manca,davide.borsatti,domenico.scotece,gianluca.davoli,luca.foschini,walter.cerroni}@unibo.it
‡ Distributed Systems Research Group, University of Ferrara, Ferrara, Italy
Email: {filippo.poltronieri,cesare.stefanelli,mauro.tortonesi,mattia.zaccarini}@unife.it
† Department of Mathematics and Computer Science, St. John's University, Queens, NY, USA
Email: grabarng@stjohns.edu
§ Operational Innovation, IBM TJ Watson Research Center, NY, USA
Email: lshwart@us.ibm.com

***Abstract*—The use of microservice-based applications is becoming more prominent also in the telecommunication field. The current 5G core network, for instance, is already built around the concept of a "Service Based Architecture", and it is foreseeable that 6G will push even further this concept to enable more flexible and pervasive deployments. However, the increasing complexity of future networks calls for sophisticated platforms that could help network providers with their deployments design. In this framework, a central research trend is the development of digital twins of the physical infrastructures. These digital representations should closely mimic the behavior of the managed system, allowing the operators to test new configurations, analyze what-if scenarios, or train their reinforcement learning algorithms in safe environments. Considering that Kubernetes is becoming the de-facto standard platform for container orchestration and microservice-based application life cycle management, the implementation of a Kubernetes digital twin requires an accurate characterization of the microservice response time, possibly leveraging suitable Machine Learning techniques trained with measurement data collected in the field. In this paper we introduce a new methodology, based on Mixture Density Networks, to accurately estimate the statistical distribution of the response time of microservice-based applications. We show the improvement in performance with respect to simulation-based inference procedures proposed in literature.**

***Index Terms*—Service Management and Orchestration, Kubernetes, Simulation, Optimization, Digital twins.**

I. Introduction

Modern communication infrastructures are undergoing a rapid shift toward a massive integration of software components. This process began with solutions like Network Function Virtualization (NFV) and Software Defined Networking (SDN). These new technologies quickly became one of the enabling pillars of the 5G architecture, enabling features like network slicing, thanks to the flexibility level they offer compared to traditional approaches [1]. As can be seen from preliminary studies, in the evolution toward 6G it is foreseen that the adoption of cloud-native technologies (e.g., microservice-based applications) will increase to reach the extreme requirements of future services [2]. For example, the increasing interest in the use of container-based solutions is confirmed by the latest updates in the NFV Management and Orchestrator architecture by ETSI [3], with the introduction of elements such as Container Infrastructure Service Management, Container Image Registry, and Container Infrastructure Service Cluster Management. In the context of orchestrating container-based applications, Kubernetes has emerged as the de-facto standard platform for these tasks. Notably, the Global System for Mobile Communications Association (GSMA) has recognized Kubernetes as one of the reference open-source solutions for telco clouds [4]. This recognition underscores the growing importance of Kubernetes in supporting telecommunication network deployments.

As future networks become more intricate and complex, there is a pressing need for sophisticated platforms that can assist network providers in designing and optimizing their deployments. Within this framework, one central research trend revolves around the development of *digital twins* for physical infrastructures [5]. These digital representations aim to closely mimic the behavior of managed systems, enabling operators to experiment with new configurations, explore what-if scenarios, and train reinforcement learning algorithms in safe environments. However, creating a digital representation that accurately mimics the behavior of its physical counterpart is a challenging task. This applies also to the case of Kubernetes as a platform for orchestrating microservice-based applications. With regard to a Kubernetes digital twin, one of the most challenging parts is the accurate modeling of microservice response times, which in many cases cannot be approximated as constant values or with simplistic probabilistic distributions.

Recently we presented the initial design and the evaluation of *KubeTwin*, a Kubernetes digital twin [6], showing results on its capacity to reenact the behavior of a microservice-based application running on a real Kubernetes cluster. From those preliminary results it was clear that, to achieve a more accurate simulation of the system, we need to improve the characterization of the single microservice response time,

possibly leveraging a suitable Machine Learning technique. In this paper, we focus on that issue and investigate a different statistical description method for the response time based on Mixture Density Networks (MDNs), which combine the use of conventional Feedforward Neural Networks (FNN) with characterization by means of mixture models. We show how MDNs allow us to accurately estimate the statistical distribution of the response time of an illustrative microservice-based application. Experimental results show that, by using the improved MDN-based model, KubeTwin significantly improves its accuracy in mimicking the physical Kubernetes application compared to the methodology presented in [6].

The paper is organized as follows. Section II presents related work in the field of application response time characterization. Section III introduces some background on MDNs. Section IV describes the layered queueing system describing the microservice-based application under analysis, and displays the measurement data obtained from a real Kubernetes system. Section V describes the MDN-based model we propose in this paper, and shows the performance of this characterization method. Finally, Section VI concludes the paper, including possible future research directions.

II. Related work

An efficient representation of microservice-based scenarios requires a mathematical characterization of all the quantities related to the components involved in the architecture (e.g., containers, links, queues, etc.) and their dependencies. This description is necessary to have an analytical or virtual representation as close as possible to the real system, making it possible to derive relevant performance metrics such as application response time or throughput experienced by users and to allow the service providers to take optimization decisions. A microservice-based architecture, such as the one implemented in Kubernetes, is well suited to be represented by a so-called *Layered Queuing Network* (LQN) model, as we show in Section IV.

The estimation of performance metrics in LQNs is widely explored in the literature, and many of the works are devoted to service management applications for Quality of Service (QoS) assessment. The characterization of these systems can be derived by analytical models based on queuing theory or by data-driven methods; in this context, data-driven approaches allow to avoid complex analytical modeling when dealing with large layered queuing systems, taking advantage of a variety of machine learning methods (e.g., supervised and reinforcement learning) to process data and model the performance metrics of the aforementioned systems, enabling the possibility to discover relations between data that may not be known a priori. The analytical approach in [7] merges different results of queuing theory and provides a model to characterize sophisticated distributed systems. In this case, the solution of the LQN model is obtained through an iterative method involving a succession of layers' submodels whose composition describes the whole system with a top-down procedure.

The LQN decomposition through submodels is again exploited in [8] for the characterization of mean and variance of the service response time: to make the analysis more realistic an MDN supervised machine learning model is adopted to approximate the unknown Probability Density Function (PDF) of the response time in each layer conditioned to some parameters, then the overall response time PDF is obtained performing convolutions of all the involved PDFs to reduce the complexity of the convolution operation, each of the layer's PDF is fitted on a phase-type distribution to benefit of its closure property.

Adopting the same Machine Learning model, the service response time of the LQN can be modeled avoiding theoretical queue models, where the conditioning parameters can be explored in the space of available system's measurements [9], such as previous realizations of response times or lengths of queues when a request comes to the system, or server statistics [10] such as CPU and memory utilization. The necessity to account for uncertainty with probabilistic models can be extended with online learning to dynamically predict the latency requirement with varying workloads over time [11]. However, these ML models treat the whole application monolithically, while we model each microservice singularly.

Independently on the adopted machine learning model, once the data-driven method is applied, the characterization is available, and it helps the service management to meet the objectives by adopting different control actions: autoscaling of replicas, request routing, request admission/blocking are common strategies [9], [11], [12]. In contrast to the supervised learning models, another possibility is to employ reinforcement learning: this ML class helps to directly map control parameters (in addition to statistics) on performance metrics. However, these models need to interact with the environment, and a simulator of the LQN system is needed to speed up the procedure by avoiding the overhead in setting up a test-bed and collecting results from it. Again, supervised learning may help in these situations [13]–[15].

From a different perspective, our work aims to merge the benefits introduced by a Machine Learning model accounting for uncertainty in each component of the system architecture with the benefits of leveraging a digital twin: by characterizing each component in a modular way inside the digital twin and accounting for their uncertainty it is possible to have a full and close to reality description of them that can be reflected in the digital twin. At the same time, the digital twin enables the possibility to easily integrate evolving architectural and decision strategies that could be time-consuming to be managed in an analytical or in a full machine learning model of the real system (e.g., update of specific microservices, introduction of new microservices or implementation of specific scheduling strategies).

III. Mixture density networks

The prediction of a continuous random variable (*target*) with conventional FNNs for regression often leverages the

minimization of the Mean Squared-Error (MSE) cost function, providing an approximation of a target value with its conditional average, limiting its description. A more complete description of a random variable, in particular for targets with high variability, may be evaluated through the conditional PDF of the target data, given an input vector (*features*).

In order to characterize the conditional probability of the target, a model combining a conventional FNN and a mixture model can be adopted. This model is called MDN and provides a general framework to approximate conditional density functions of targets by modeling the probability parameters as a function of the features [16].

Given a feature vector $\bar{x}$ and a generic random variable T the set of parameters that characterize it would be $(\theta_1, ..., \theta_n)$, making possible to write its PDF as:

$$p(t|\theta_1(\bar{x}), ..., \theta_n(\bar{x})) \tag{1}$$

where t is a possible value for T. By omitting the parameters in the notation, it can be referred to as $p(t|\bar{x})$ simply.

The MDN aims to model the conditional probability density as a linear combination (*mixture model*) of kernel functions:

$$p(t|\bar{x}) = \sum_{c=1}^{C} \alpha_c(\bar{x})\phi_c(t|\bar{x}) \tag{2}$$

where $p(\bullet)$ is the PDF of the target variable, C is the number of kernel components in the mixture, $\alpha_c(\bar{x})$ is the mixing coefficient of the c-th kernel (it represents the prior probability of that kernel conditioned on $\bar{x}$), $\phi_c(t|\bar{x})$ is the c-th kernel (representing the conditional density function of the target t, conditioned on $\bar{x}$).

The model allows different kernel choices (e.g., Normal, Uniform, Exponential, or other distributions); in the following, we opted for the Weibull distribution as kernel of choice, which is characterized by two parameters, γ and k, called *scale* and *shape*, respectively. We chose the Weibull distribution since it generalizes the Exponential distribution and is suitable for describing non-negative random variables, such as the time between occurrences of events.

The generic Weibull kernel conditioned to $\bar{x}$ is:

$$\phi_c(t|\bar{x}) = \begin{cases} \frac{k_c(\bar{x})}{\gamma_c(\bar{x})}\left(\frac{t}{\gamma_c(\bar{x})}\right)^{k_c(\bar{x})-1} e^{-(t/\gamma_c(\bar{x}))^{k_c(\bar{x})}}, & t \geq 0 \\ 0, & t < 0 \end{cases} \tag{3}$$

Hence, in the mixture model, each term is characterized by a set of three parameters (mixture parameters): $\alpha_c(\bar{x})$, $\gamma_c(\bar{x})$ and $k_c(\bar{x})$, assumed to be unknown continuous functions of the features $\bar{x}$.

In order to model the unknown functions:

- the first stage of the model is a *conventional FNN*, which takes as input a vector of features $\bar{x}$ and transforms them through proper weights $\bar{w}$, providing as output a vector $\bar{y}(\bar{x}; \bar{w})$;
- the output vector of the first stage is provided as input of a second stage represented by a *mixture model*.

In the mixture model, the transformed features $\bar{y}(\bar{x}; \bar{w})$ are processed through appropriate activation functions to model the mixture parameters (i.e., mixing coefficients, scales, and shapes). The final output of the model turns out to be the conditional probability density of the target (2). The combination of the two stages is referred to as a MDN, whose basic structure is represented in Fig 1.

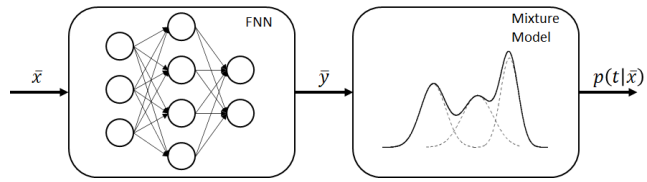

Fig. 1. Structure of a MDN.

The adoption of this model requires the definition of some hyper-parameters (i.e., fixed parameters defining the specific implementation of the neural network), such as the number of mixture components C, the kernel functions, and the number of hidden layers and units in the FNN.

The mixing coefficients $\alpha_c(\bar{x})$ must satisfy the condition:

$$\sum_{c=1}^{C} \alpha_c(\bar{x}) = 1 \tag{4}$$

which can be achieved by choosing the *softmax* activation function for these outputs, leading to the generic c-th mixing coefficient expressed as:

$$\alpha_c = \frac{e^{y_{\alpha_c}}}{\sum_{i=1}^{C} e^{y_{\alpha_i}}} \tag{5}$$

where y_{α_c} represents the FNN output related to the c-th mixing coefficient.

The scale $\gamma_c(\bar{x})$ and shape $k_c(\bar{x})$ parameters must satisfy the conditions:

$$\gamma_c(\bar{x}) > 0 \qquad k_c(\bar{x}) > 0 \tag{6}$$

They can be achieved by choosing the *Exponential Linear Unit* (ELU) activation function:

$$\gamma_c = \mathrm{ELU}(y_{\gamma_c}) + 1 \tag{7}$$

$$k_c = \mathrm{ELU}(y_{k_c}) + 1 \tag{8}$$

where y_{γ_c} and y_{k_c} represent the FNN outputs related to the scale and the shape of the c-th component, respectively. The ELU is defined as:

$$\mathrm{ELU}(y) = \begin{cases} y, & y > 0 \\ e^y - 1, & y \leq 0 \end{cases} \tag{9}$$

The weights $\bar{w}$ in $\bar{y}(\bar{x}; \bar{w})$ are learned during the training of the MDN through a training set $\left\{\bar{x}^{(q)}, t^{(q)}\right\}$ of cardinality m by Maximum Likelihood Estimation (MLE). The objective of MLE is to find the set of parameters for which the observed

data (the training set) have the highest joint probability. Assuming that the training examples are drawn independently from the PDF given by (1), the likelihood function of the set can be written as:

$$\mathcal{L} = \prod_{q=1}^{m} p(t^{(q)}, \bar{x}^{(q)}) = \prod_{q=1}^{m} p(t^{(q)}|\bar{x}^{(q)}) p(\bar{x}^{(q)}) \tag{10}$$

The maximum likelihood estimate is:

$$\widehat{\bar{w}} = \arg\max_{\bar{w}} \mathcal{L}(\bar{w}) \tag{11}$$

From the likelihood it is possible to derive the error function:

$$E = -\log \mathcal{L}(\bar{w}) = -\sum_{q=1}^{m} \log p(t^{(q)}|\bar{x}^{(q)}) \tag{12}$$

Formula (12) is called negative log-likelihood (NLL), in its expression the $p(\bar{x})$ factor is neglected since it does not depend on the NLL parameters. Its minimization is equivalent to the maximization of the likelihood.

By taking into account the mixture model (2), the NLL becomes:

$$E = \sum_{q=1}^{m} E^{(q)} = -\sum_{q=1}^{m} \log \sum_{c=1}^{C} \alpha_c(\bar{x}^{(q)}) \phi_c(t^{(q)}|\bar{x}^{(q)}) \tag{13}$$

where $E^{(q)} = -\log \sum_{c=1}^{C} \alpha_c(\bar{x}^{(q)}) \phi_c(t^{(q)}|\bar{x}^{(q)})$ is the error contribution of the q-th element in the training set.

Back-propagation is the standard procedure to minimize the error function. For this purpose, the gradient of the error function concerning the FNN output needs to be computed as reported in [17]. The software implementation of the back-propagation algorithm for this model can be inspired by regression FNN based on the MSE function. The modification of the error function is required to apply standard optimization procedures such as gradient descent.

At the end of the training, the MDN can approximate the conditional density function of the target data given the input features, allowing it to have a probabilistic description of the data generation process. The MDN results are useful to derive the specific moments (e.g., mean and variance).

IV. Representation of Microservice Applications

The cloud-native deployment scenario under analysis in the experimental work can be represented through a multi-layered queuing system.

A. System description

The general implementation of the system can be described by several components.

- *Requests*: the system receives incoming requests sent by users, represented by a Poisson process with average request rate λ [req/s].
- *Replicas*: they represent multiple processing entities providing a given *microservice* (i.e., an independent portion of processes related to a more complete composite service) at a time; as they can operate in parallel and the execution is non-threaded, a section with n_{rep} replicas can contemporarily execute up to n_{rep} instances of microservices. The processing time related to a microservice performed by each replica can be represented by a random variable T_{ms} with unknown distribution.
- *Queues*: each replica in the system has its own queue in which requests are put on hold in case they cannot be immediately served.
- *Load balancer*: it routes the incoming requests to the available replicas; the simplest way to do it is to perform equal load balancing. In standard Kubernetes clusters, the load balancing is probabilistic, with each replica having the same probability of being chosen to serve a request.

In a cloud-native deployment, a service is obtained through the chaining of several microservices, and this composition is strictly related to the kind of service provided to the users.

With reference to a particular service, the structure described above can be stacked as many times as the number of microservices that compose the overall service, where each layer is responsible for a specific microservice. Figure 2 shows an example of a two-layered queuing system.

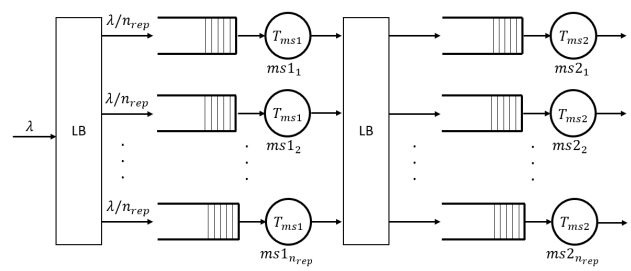

Fig. 2. Example: a two-layered queuing system with n_{rep} replicas for both layers. In the first layer, on average, the load balancer (LB) equally distributes requests among replicas.

The multi-layered queue system during its operations (e.g., load balancing, queuing, microservices processing) introduces a random response time to the users, which we will refer to as TTR (time-to-resolution). The distribution of the TTR is unknown and depends on the kind of service and the configuration of the system.

In our experiments, we implemented a two-layered queuing system with different values of $n_{\text{rep}} \in \{1, 2, 3, 4\}$. The system is managed by the *Kubernetes* platform, and the replicas are deployed as *containers* distributed on a cluster.

The designated service for that system is an image processing application, in particular:

- the first layer is dedicated to an image pre-processing microservice (ms1) to simplify the operations of the second microservice;
- the second layer is dedicated to the actual image processing microservice (ms2).

The random processing times of a generic replica in the first and second layers are indicated as T_{ms1} and T_{ms2}, respectively.

B. Training Data Collection

Before delving into the results of the proposed solution, we believe it is important to present how we obtained the training data. Furthermore, we will discuss their behavior since we think they can give meaningful insight into the performance of microservices in realistic Kubernetes deployments and motivate the choice of more sophisticated methods for the characterization of microservice processing time. Firstly, the system we employed for the measurements is a small two-node Kubernetes cluster. Each node runs Ubuntu 20 LTS and is equipped with 4 vCPU, 8G RAM, and 70G of disk. The only plugin we added to the Kubernetes deployment was Calico as Container Network Infrastructure (CNI), without any other custom module. Therefore, it could be considered as an ordinary standard deployment.

On top of this cluster, we deployed the application described in the previous section. Both microservices are Python-based software components exposing HTTP endpoints to enable the interaction between them. Furthermore, both microservices log the duration of each HTTP request in their internal log. By retrieving these logs, we can discover the time needed to complete each request, thus allowing us to measure the time a request spends in each of the two microservices. Then, by feeding these data to the MDN network, we can obtain the statistical distribution of the two microservices response time. It is important to mention that this process could be highly automated with well-known Kubernetes plugins, such as log-collecting tools like Fluentd or tracing operators like Jaeger (built-in in the Istio service mesh). Finally, we developed a Python script to generate a fixed number of HTTP requests with a configurable Poisson arrival rate.

Thanks to these tools, we performed several measurement campaigns of 1000 requests each, at increasing arrival rates and with different application configurations. In detail, we increased the number of replicas of each microservice from 1 to 4, and we also varied the behavior of each application component by introducing a “slow-down" factor sd. This value affects the response time of each microservice by making them repeat their computing operations sd times before sending out the reply. The reason behind this choice was that, in this way, we could obtain microservices with different execution times in a controlled manner. Furthermore, by increasing these execution periods, we could verify that some peculiar behaviors were not depending on the time scale of the measured execution time of a single function.

We performed the experiments scaling the sd factor from 1 to 10. For each value, we normalized the request rate over the sd factor to keep the server utilization (i.e., the ratio of the arrival rate to the service rate) constant. Furthermore, we compared the behavior of microservices for the multiple replicas configuration with an sd factor equal to 10. As before, during the experiments we tuned the Requests per Second (RPS) value to keep the utilization constant (i.e., by multiplying the RPS value by the number of replicas considered). Focusing on the first microservice in the single replica configuration, Fig. 3 shows a decreasing mean for increasing values of the request rate, which applies for all the sd values. This behavior is counterintuitive since it would be reasonable to believe that the average processing time of a microservice would either remain constant or increase with a higher number of incoming requests.

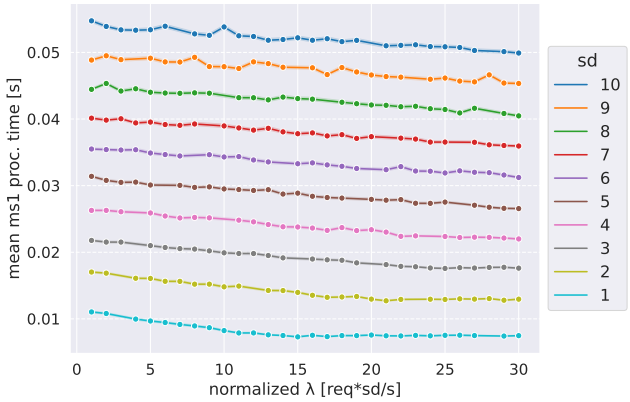

Fig. 3. Comparison of ms1 mean proc. time, conditioned to RPS and sd factor from 1x to 10x (non threaded).

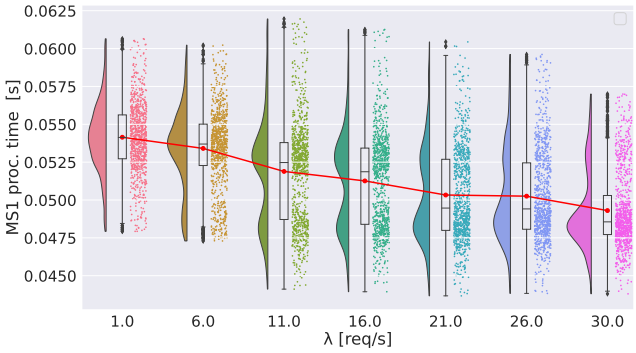

Fig. 4. Raincloud plot [18] of ms1 proc. time, conditioned to RPS, for $sd = 10$ and one replica (non threaded).

As can be seen from the kernel density estimations depicted in Fig. 4, the processing time of the first microservice exhibits strong multimodality in which the two main kernels vary their prior-probability with the arrival rate: for increasing arrival rate the prior-probability of the kernel with higher processing times decrease in favor of the kernel with lower processing times. The reason behind these results requires further investigation, but we believe it might depend on specific CPU allocation policies that try to optimize highly active processes. This result was one of the main reasons that motivated us to search for modeling tools capable of learning these hidden dependencies. The first microservice shows the same dependency even in the case of multiple replicas (Fig. 5). In particular, for the cases with one and three replicas, the difference in the mean processing time between the extreme values of request rate is about 8%. On the other hand, the same behavior is no longer appreciable for the second microservice, at least for the single replica deployment, as shown in Fig. 6 where the processing

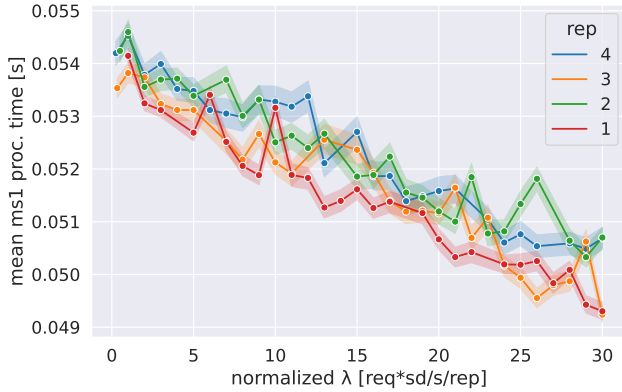

Fig. 5. Comparison of ms1 mean proc. time (95% confidence interval) for $sd = 10x$. The processing time is plotted for different combinations of RPS values and numbers of replicas. The RPS (λ) is equally balanced among replicas.

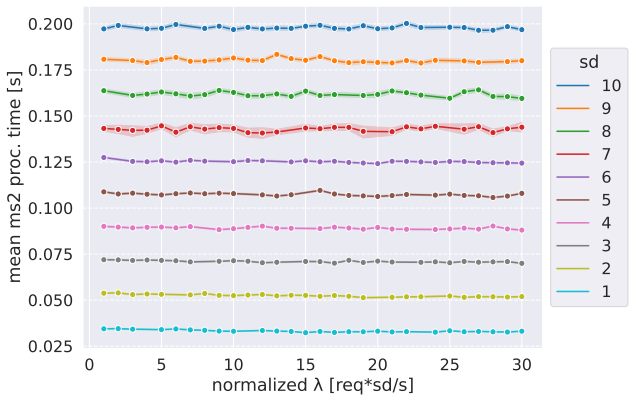

Fig. 6. Comparison of ms2 mean proc. time, conditioned to RPS and sd factor from 1x to 10x (non threaded).

times seem to have no relationship with the request rate. Going deeper into the statistics of the second microservice in the multi-replica deployment, Fig. 7 shows an opposite behavior of ms2 with multiple replicas; this is highlighted in Fig. 8 showing some processing times for the two replicas deployment.

In general, Figures 5 and 7 show that for both microservices there are slight variations in the response time by changing the number of replicas considered. Again, this was an additional element that pushed us into this research of a more complex modeling tool.

V. Models for Characterization of Microservice Response Time

In this section, we discuss the implementation of the model for the characterization of microservice-based applications in Kubernetes, according to the methodology described in Section III. After, we will show its performance on the test set and its field evaluation with the Kubernetes digital twin. The model aims to characterize, employing a MDN, the specific

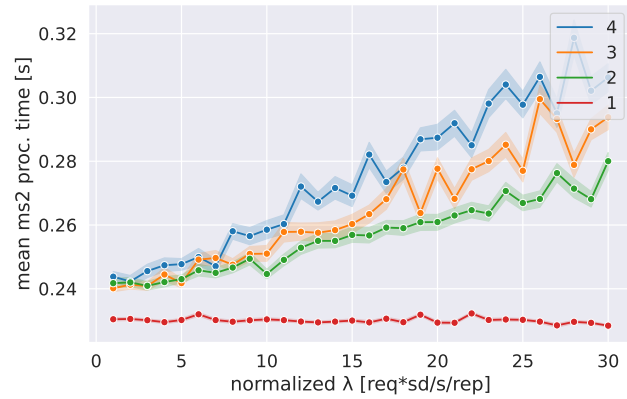

Fig. 7. Comparison of ms2 mean proc. time (95% confidence interval) for $sd = 10x$. The processing time is plotted for different combinations of RPS values and numbers of replicas. The RPS (λ) is equally balanced among replicas.

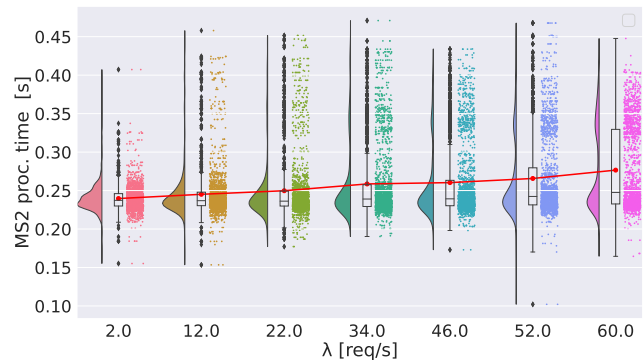

Fig. 8. Raincloud plot of ms2 proc. time, conditioned to RPS, for $sd = 10x$ and two replicas (non threaded).

microservice processing time, given the knowledge of the request rate to the system and the number of available replicas.

A. Models implementation

As previosly discussed, in our microservice framework the mathematical objective of the model is to approximate:

$$p(t_{ms,i}|\lambda, n_{\text{rep}}) \tag{14}$$

Where $p(\bullet)$ indicates the PDF, $t_{ms,i}[s]$ is the i-th microservice processing time ($i = 1, 2$), $(\lambda, n_{\text{rep}})$ are the request rate and the number of available replicas for that microservice, respectively. In other words, the model looks for and generalizes a probabilistic (rather than point estimate) relationship between processing time, request rate, and number of available replicas, which proves useful to make simulations and perform analysis by the digital twin.

The model is a FNN implemented with the Keras API [19] and the TensorFlow probability library [20], with the following characteristics:

- 2 input neurons: one input neuron for RPS and one for n_{rep};

- 4 mixture components: the approximated PDF is the superposition of 4 Weibull's PDFs;[1]
- 12 output neurons: the outputs represent the parameters of the components in the mixture model, considering that every Weibull's PDF is characterized by 3 parameters (the mixture parameter in the superposition, the scale and the shape);
- 2 hidden layers: each hidden layer has 8 neurons.

The hidden neurons are set with *LeakyReLU* (Leaky Rectified Linear Unit) activation function. The activation functions for the output neurons depend on the parameter they represent, as discussed in Section III.

At the end of the training phase the microservice processing time is characterized in the following way:

$$p(t_{ms,i}|\lambda, n_{\text{rep}}) = \\ = \sum_{c=1}^{4} \alpha_c(\lambda, n_{\text{rep}})\phi_c(t_{ms,i}|\lambda, n_{\text{rep}}) \tag{15}$$

B. Performance evaluation on test set

The model architecture described in Section V-A was applied to both microservices ms1 and ms2 with the data set presented in Section IV-B. The available data set was partitioned in the following way: 70% for training set, 20% for validation set and 10% for test set. The performance of the model must be evaluated though a *goodness of fit* measurement, quantifying the disagreement between the prediction made by the model and a sample of observed values. The goodness of fit of our MDN model was evaluated through the Kolmogorov-Smirnov (KS) test [21] and the Wasserstein distance [22] on the test set, represented by 12 samples with 1000 elements each.

The Kolmogorov-Smirnov test compares the empirical Cumulative Distribution Function (eCDF) $F_n(t)$ provided by a sample with n elements with a reference CDF $F(t)$ through the KS-distance, defined as:

$$D_n = \sup_t |F_n(t) - F(t)| \tag{16}$$

The hypotheses of the test are:

$$H_0 : F(t) = F_n(t), \ \forall t \tag{17}$$

$$H_1 : F(t) \neq F_n(t), \ \text{for some } t \tag{18}$$

Under the null hypothesis, the KS-distance (16) is distributed according to the Kolmogorov distribution and converges to 0 in case $n \to \infty$. The statistical significance of the test is represented by the p-value p, that is the probability of having a KS-distance at least as extreme as the one observed under the null hypothesis. By setting a significance level α (typically $\alpha = 0.05$), the null hypothesis H_0 is rejected in favour of the alternative H_1 in case $p \leq \alpha$, otherwise there is a lack of evidence to reject H_0.

Figures 9 and 10 represent the performance of models for ms1 and ms2 on the test set, in terms of the KS test. For ms1,

[1]During initial trials, the Normal, Gamma and Weibull PDFs were tried, then the latter was chosen because it provides the best fit among them.

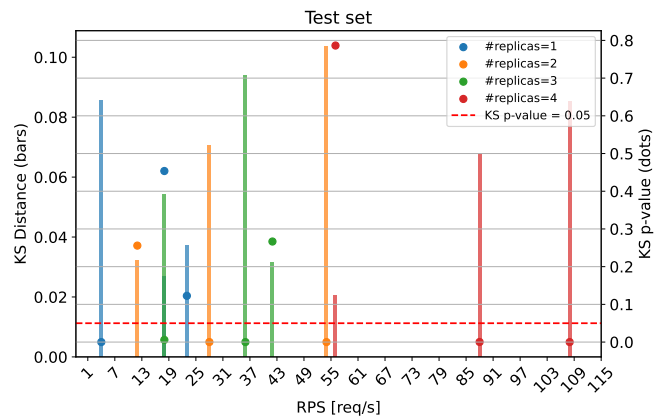

Fig. 9. Performance evaluation of MDN model for ms1 in terms of KS on the test set.

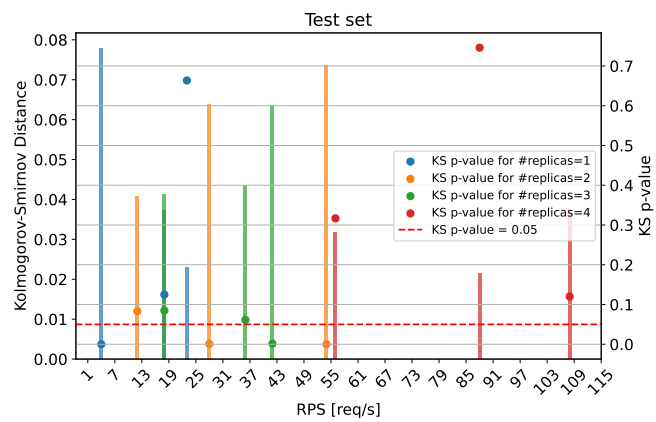

Fig. 10. Performance evaluation of MDN model for ms2 in terms of KS on the test set.

despite most distances being below 0.1, 7 out of 12 sets show the null hypothesis is rejected with p-values below 0.05. For ms2 the maximum distance is below 0.08 and 4 out of 12 sets show the null hypothesis is rejected. Since the KS test compare the maximum distance between distributions, while the objective of our models is to generate samples to emulate the microservices inside the digital twin, we introduced the Wasserstein metric to take into account the overall distance between test samples and samples generated by our models. In particular, this metric represents the minimum "cost" to transform the distribution of one sample to the other one.

In the one dimensional case, for two empirical measures P and Q with respective samples $t_1, ..., t_n$ and $u_1, ..., u_n$ of cardinality n (in ascending order), the p-Wasserstein distance is defined as:

$$W_p(P, Q) = \left(\frac{1}{n} \sum_{i=1}^{n} \|t_i - u_i\|^p \right)^{1/p} \tag{19}$$

In our framework, the 1-Wasserstein distance (1-WD) implemented in [23] was employed to quantify the distance between a sample in the test set for a couple $(\lambda, n_{\text{rep}})$ and the sample

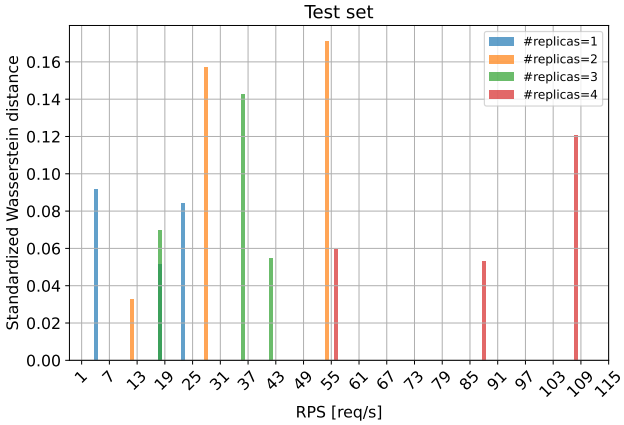

Fig. 11. Performance evaluation of MDN model for ms1 in terms of Standardized Wasserstein metric on the test set.

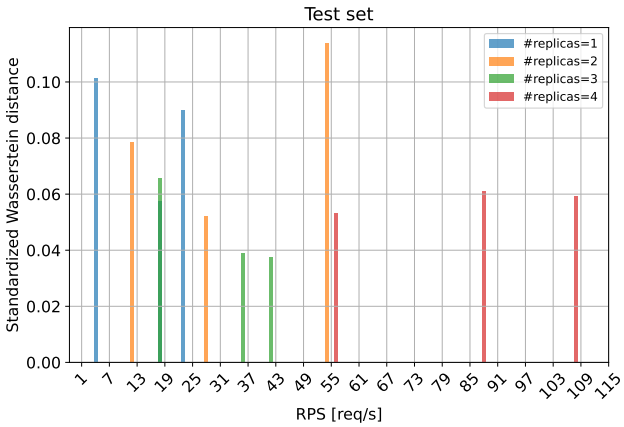

Fig. 12. Performance evaluation of MDN model for ms2 in terms of Standardized Wasserstein metric on the test set.

generated by our model with the same couple as input. In order to relate this metric with the test sample under analysis, we introduced the Standardized Wasserstein distance as the ratio between the 1-WD previously described and the standard deviation of the test sample under analysis. In this way, the metric can be interpreted as the "distance in terms of standard deviations". Figures 11 and 12 show performance evaluation performed with Standardized Wasserstein distance. For ms1 all the distances are below 0.2, with 8 out of 12 test samples showing a distance below 0.1. The performance on ms2 shows even smaller distances, with 10 out of 12 test samples below 0.1. Since most of the test samples show distances close or lower to 0.1, we expect the error in the approximation of the true microservices samples with the ones generated by our mixture models in the digital twin to be relatively small compared to the variability of the true data. However, in a production scenario, the approximation error restrictions need to be related to the specific application requirements.

C. Digital Twin evaluation

After obtaining the model of the two microservices from the MDN, we plugged them into the KubeTwin simulator

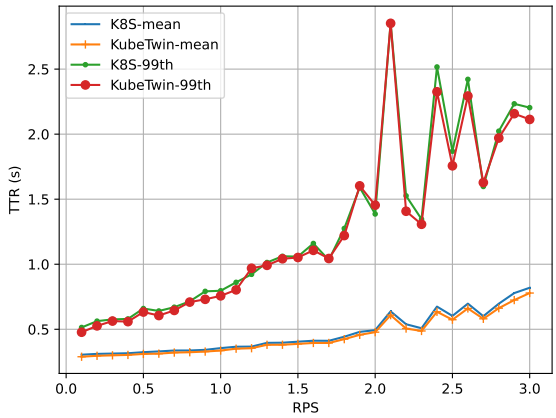

Fig. 13. Comparison between the mean and 99th percentile response time of the real application and KubeTwin, with $sd = 10$ and 1 replica for each microservice.

to compare the overall application response times from the Kubernetes cluster with the ones generated by the simulator. In this experiment, we reenacted with KubeTwin the same request generation patterns as during the measurement campaigns. At each request generation time, the simulator samples from each microservice distribution a response time using as input the RPS of the last eight requests (tunable parameter). Then, the simulator computes the total request time by summing up the two distribution samples and the queueing time of the request. The details on how KubeTwin manages the internal queues can be found in [6].

Figure 13 depicts the mean and 99th percentile of the total application response time measured from the Kubernetes system and Kubetwin, with $sd = 10$ and a single replica for each microservice. The results show that the simulator can reenact the system behavior with good accuracy. In detail, the average MSE measured between the two is around 0.00263 for the mean response time, as reported in Table I. We repeated the test with the same configurations but considering the microservices modeled with the methods presented in [6] (i.e., a three-component Gaussian Mixture Model fitted over the data measured with RPS = 1). Applying this modeling method, we obtained a much larger MSE value of about 0.20881 for the mean response time. These results show a strong improvement in the performance of KubeTwin, thus proving the soundness of the proposed characterization of the response time with a MDN approach.

Finally, we made the same tests with three replicas. Figure 14 reports the results of the MDN-based model with this application configuration. Comparing them to those obtained with a single replica, there is a slight performance degradation. We believe this might be caused by the randomness introduced by the load-balancing feature of Kubernetes. Since vanilla Kubernetes distributes the request among the available replicas using equiprobable iptables rules, some microservices might receive higher request rates than others, thus leading to diverse response times. Nevertheless, the comparison with the method

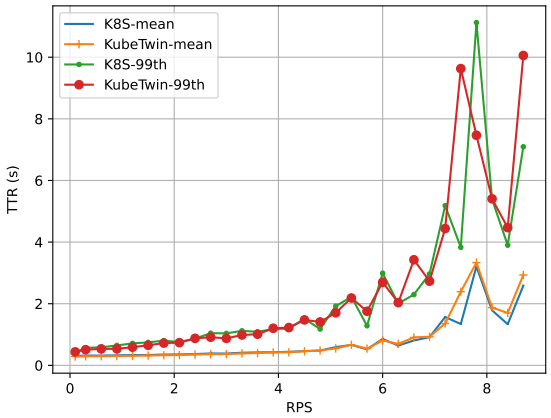

Fig. 14. Comparison between the mean and 99th percentile response time of the real application and KubeTwin, with $sd = 10$ and 3 replicas for each microservice.

TABLE I
AVERAGE MEAN SQUARE ERROR (MSE) COMPARISON BETWEEN THE PROPOSED SOLUTION AND THE GAUSSIAN MIXTURE PRESENTED IN [6] FOR BOTH MEAN AND 99TH PERCENTILE OF THE TTRS.

	MDN		Gaussian Mixture [6]	
n_{rep}	Mean	99th percentile	Mean	99th percentile
1	0.00263	0.00423	0.20881	1.02729
3	0.04830	1.95080	10.38479	56.5381

presented in [6] still shows a substantial MSE reduction.

VI. CONCLUSION

In this work, we showed how an MDN-based model for the estimation of the statistical distribution of the response time of microservices could improve the performance of simulation-based approaches like KubeTwin, improving its effectiveness as a digital twin for Kubernetes environments. In detail, we observed a meaningful performance improvement, with an MSE reduction of about two orders of magnitude compared with the results obtained in [6]. As mentioned in the previous sections, as future research direction, we want to investigate the reasons behind the counterintuitive behavior of some microservices (e.g., Fig. 4). Furthermore, we plan to test the proposed solution with more complex applications and to enrich the ML model by adding more input features (e.g., overall cluster load or request types) that might further improve the statistical description of the microservices. Lastly, we want to automate this learning process by implementing Continual Learning mechanisms to train the model at runtime, exploiting tools like Istio to gather the training data.

ACKNOWLEDGEMENTS

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRPP) of Next Generation EU (NGEU), partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"), and by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 - NGEU.

REFERENCES

[1] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.
[2] Hexa-X, "D1.4 - Hexa-X Architecture for B5G/6G Networks (Final Release)," Tech. Rep., 2023.
[3] ETSI, "ETSI GS NFV 006 – Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Architectural Framework Specification (Version 4.4.1)," Tech. Rep., 2022.
[4] GSMA, "NG.126 – Cloud Infrastructure Reference Model (Version 3.0)," Tech. Rep., 2022.
[5] S. Mihai *et al.*, "Digital Twins: A Survey on Enabling Technologies, Challenges, Trends and Future Prospects," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2255–2291, 2022.
[6] D. Borsatti *et al.*, "Modeling Digital Twins of Kubernetes-Based Applications," in *28th IEEE Symposium on Computers and Communications (ISCC)*, 2023.
[7] 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, 2009.
[8] 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)*, 2021, pp. 1–8.
[9] M. Raeis, A. Tizghadam, and A. Leon-Garcia, "Probabilistic bounds on the end-to-end delay of service function chains using deep mdn," in *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*, 2020, pp. 1–6.
[10] F. S. Samani, R. Stadler, C. Flinta, and A. Johnsson, "Conditional density estimation of service metrics for networked services," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 2350–2364, 2021.
[11] P. Kang and P. Lama, "Robust resource scaling of containerized microservices with probabilistic machine learning," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 122–131.
[12] D. Spatharakis *et al.*, "Distributed resource autoscaling in kubernetes edge clusters," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 163–169.
[13] F. Shahab Samani and R. Stadler, "Dynamically meeting performance objectives for multiple services on a service mesh," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 219–225.
[14] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35 464–35 476, 2021.
[15] H. Fang, P. Yu, Y. Wang, W. Li, F. Zhou, and R. Ma, "A novel network delay prediction model with mixed multi-layer perceptron architecture for edge computing," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 191–197.
[16] C. M. Bishop, "Mixture density networks," Aston University, WorkingPaper, 1994. [Online]. Available: https://research.aston.ac.uk/en/publications/mixture-density-networks
[17] ——, "Neural networks and their applications," *Review of Scientific Instruments*, vol. 65, pp. 1803–1832, 1994.
[18] M. Allen *et al.*, "Raincloud plots: a multi-platform tool for robust data visualization," *Wellcome Open Research*, vol. 4, p. 63, 01 2021.
[19] F. Chollet *et al.*, "Keras," https://keras.io, 2015.
[20] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/
[21] R. Legin, Y. Hezaveh, L. P. Levasseur, and B. Wandelt, "Simulation-based inference of strong gravitational lensing parameters," *arXiv preprint arXiv:2112.05278*, 2021.
[22] A. Ramdas, N. Garcia, and M. Cuturi, "On Wasserstein Two Sample Testing and related families of nonparametric tests," 2015. [Online]. Available: https://arxiv.org/abs/1509.02237
[23] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.